The Google Play Billing Handbook

A Complete Guide to In-App Purchases and Subscriptions

Contents

Preface

Who This Book Is For

This book is for Android developers who want to sell digital goods and services through the Google Play Store. You already know how to build Android apps with Kotlin. You know your way around Activities, ViewModels, Gradle, and the Android SDK. What you do not know, or know only partially, is how to integrate Google Play Billing into your app.

Maybe you have tried before. You read a few pages of documentation, got a basic purchase working, and then discovered that subscriptions have seven different states, grace periods, account holds, and a notification system that requires Cloud Pub/Sub. You realized this is not a weekend project.

This book takes you from zero billing knowledge to production readiness. It covers everything: the client library, the server API, the Play Console configuration, subscription lifecycle management, testing, security, and the edge cases that only surface in production.

What You Will Learn

By the end of this book, you will be able to:

- Set up a complete billing environment from scratch (developer account, products, API access, notifications)
- Sell one time products (consumable and non consumable) with proper verification and acknowledgement
- Implement subscriptions with free trials, introductory pricing, and multiple plan options
- Handle subscription upgrades, downgrades, and plan changes correctly
- Build a backend that verifies purchases, processes notifications, and maintains accurate entitlement state
- Manage the full subscription lifecycle: renewals, grace periods, account holds, pauses, cancellations, and recovery
- Change subscription prices for existing subscribers without losing them
- Test every billing scenario before shipping to production
- Manage a product catalog at scale using the catalog management APIs
- Protect your revenue with server side verification and fraud detection
- Implement alternative billing where regional regulations require it

How to Read This Book

This book is organized in seven parts that build on each other:

Part I: Foundations (Chapters 1-2) gives you the big picture and walks you through environment setup. Start here if you are new to Google Play Billing.

Part II: Products (Chapters 3-4) covers the two product types: one time products and subscriptions. You will understand how each works and how to configure them.

Part III: Client Side Integration (Chapters 5-8) is where you start writing code. You will learn to integrate the Play Billing Library, build purchase flows, handle plan changes, and implement error handling.

Part IV: Server Side Integration (Chapters 9-10) covers your backend: purchase verification, the Google Play Developer API, and Real Time Developer Notifications.

Part V: The Subscription Lifecycle (Chapters 11-14) is the deepest part of the book. It covers the subscription state machine, payment recovery, cancellations, pauses, winback strategies, and price changes.

Part VI: Production Readiness (Chapters 15-17) prepares you for launch: security, testing, and catalog management at scale.

Part VII: Beyond Standard Billing (Chapter 18) covers alternative billing and external offers for apps operating in regions with special regulations.

The appendices provide reference material: a complete RTDN reference table, response code reference, API endpoint reference, a PBL 7 to PBL 8 migration guide, and a pull out subscription state diagram.

If you are experienced with billing but upgrading to PBL 8, start with Appendix D (Migration Guide) and then read the chapters that cover your knowledge gaps.

If you are building a new integration from scratch, read Parts I through IV in order. Then read Part V as you prepare for production, and Part VI before launch.

Prerequisites

This book assumes you have:

- Working knowledge of Android development with Kotlin
- Familiarity with coroutines (the code examples use suspend functions and Flows)
- Basic understanding of REST APIs and JSON
- Access to a backend server (for purchase verification and notifications)
- A Google Play Developer Account (or the ability to create one)

You do not need prior experience with billing, payments, or the Google Play Console.

Play Billing Library 8 Focus

This book targets Play Billing Library (PBL) version 8.x, specifically versions 8.0.0 through 8.3.0. PBL 8 introduced several important changes from PBL 7, including auto service reconnection, pending purchase parameter changes, the `UnfetchedProduct` API, sub response codes, and new replacement mode APIs.

If you are currently on PBL 7, Appendix D provides a complete migration guide. Note that Google has announced PBL 7 will reach end of support on August 31, 2026. Migrating to PBL 8 is not optional.

All code examples in this book use Kotlin and the `billing-ktx` artifact, which provides coroutine extensions for the billing library's async operations.

Conventions Used in This Book

Code blocks show Kotlin code targeting PBL 8.x. Client side code runs on Android. Server side code runs on your backend (also shown in Kotlin for consistency).

Diagrams illustrate architecture, state machines, and flows. They follow a consistent visual style and are available as both inline images and full page reference cards in the appendices.

Bold terms indicate first use of important vocabulary or emphasis. Terms defined in Chapter 1 are used consistently throughout.

Tables summarize reference information that you will want to look up later. The appendices contain comprehensive reference tables.

Acknowledgments

This book would not exist without the Google Play Billing documentation team, whose work across dozens of pages formed the foundation for this consolidated guide. The Android developer community's questions, bug reports, and Stack Overflow answers also shaped the practical focus of every chapter.

Chapter 1: Understanding Google Play's Billing System

If you have ever tried to sell digital goods in an Android app distributed through the Google Play Store, you have encountered the Google Play Billing system. It is not optional. Google requires every app on the Play Store to use its billing infrastructure for in app digital purchases. Understanding this system is the first step toward building a reliable, production ready purchase experience.

This chapter gives you the big picture. You will learn what Google Play Billing is, who the key actors are, what vocabulary you need, and how the pieces fit together before you write a single line of billing code.

What Google Play Billing Is and Why It Exists

Google Play Billing is the payment infrastructure that handles all in app purchases of digital goods and services on the Google Play Store. It processes payments, manages subscriptions, handles refunds, and provides purchase verification, all through a combination of a client side library (the Play Billing Library), a server side API (the Google Play Developer API), and a management console (the Google Play Console).

Google built this system for a few reasons:

- **Consistent user experience**: Every purchase in every app follows the same flow. Users see the familiar Google Play purchase dialog, can use their saved payment methods, and manage all subscriptions in one place.
- **Trust and security**: Google handles payment processing, fraud detection, and dispute resolution. You do not need to become PCI compliant or store credit card numbers.
- **Revenue share**: Google takes a percentage of each transaction (typically 15% for the first $1M in annual revenue, 30% after that for most developers). This is the business reason the system exists.

As a developer, you get reliable payment processing across 170+ countries and regions, automatic tax handling, and a subscription management system that handles renewals, grace periods, and payment recovery. The trade off is that you must work within Google's framework.

The Big Picture: Three Pillars

Google Play Billing rests on three pillars that work together:

1. The Play Billing Library (PBL)

This is the Android client library you add to your app. It communicates with Google Play Services on the device to show purchase dialogs, query available products, and report purchase results back to your app. This book focuses on PBL version 8.x, the latest major version.

PBL handles the client side of the billing experience. It provides APIs to query product details (names, prices, descriptions), launch the Google Play purchase dialog, and receive purchase results. It also lets you acknowledge purchases, consume products, and check what the user currently owns.

The library ships as a Gradle dependency (`com.android.billingclient:billing-ktx`) and communicates with Google Play Services through interprocess communication (IPC). This means the user must have a Google account signed in and a reasonably recent version of the Google Play Store installed on their device.

2. The Google Play Developer API

This is the server side REST API that your backend uses to verify purchases, check subscription status, acknowledge transactions, and manage your product catalog. It is the source of truth for purchase state.

The API lives at `androidpublisher.googleapis.com` and uses OAuth 2.0 authentication with service accounts. It provides endpoints for verifying purchases (`purchases.subscriptionsv2.get` , `purchases.products.get`), managing subscriptions (cancel, defer, revoke), and performing catalog operations (create, update, delete products and offers).

You should treat this API as the authoritative source of truth for every purchase. The client library provides convenience, but your backend should always verify with the API before granting access to content or features.

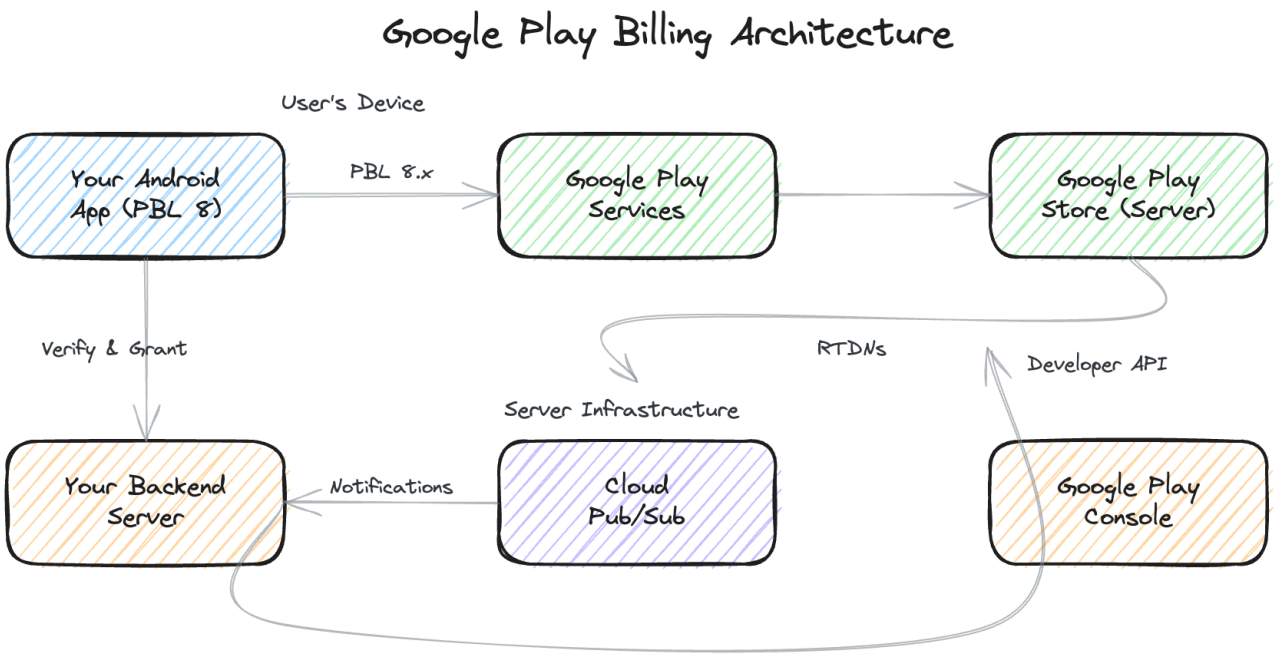
3. The Google Play Console

This is the web dashboard where you create and configure your products, set prices, define subscription offers, view financial reports, and manage your app's billing settings. It is also where you configure Real Time Developer Notifications and API access.

The Console is where you define your product catalog: creating one time products, subscriptions, base plans, and offers. You set default prices, configure regional pricing, define offer eligibility rules, and manage your app's billing settings. It also provides financial reports, refund management, and subscriber analytics.

For large catalogs, you can skip the Console UI and manage products programmatically through the catalog management APIs. But the Console remains essential for initial setup, monitoring, and configuration changes that require visual review.

How they work together: Your app uses PBL to initiate purchases, your backend uses the Developer API to verify and manage them, and you use the Console to configure everything. A fourth piece, Cloud Pub/Sub, acts as the communication channel for Real Time Developer Notifications (RTDNs), which push purchase and subscription state changes from Google to your backend in near real time.



Key Actors

Four actors participate in every billing transaction:

Your App

Your Android application uses the Play Billing Library to query products, launch purchase flows, and detect completed purchases. It communicates with your backend to verify purchases and grant entitlements.

Your app is responsible for:

- Querying available products and displaying them to the user
- Building and launching purchase flows when the user decides to buy
- Listening for purchase results and forwarding purchase tokens to your backend
- Acknowledging or consuming purchases (as a fallback; prefer doing this on your backend)
- Querying existing purchases on app launch and foreground resume to stay in sync

Google Play Services

This is the system level service running on the user's Android device. When your app calls PBL methods, PBL delegates to Google Play Services, which handles the actual communication with Google's servers. The user sees this as the Google Play purchase dialog.

Google Play Services handles the payment UI, payment method selection, and secure communication with Google's servers. Your app never sees credit card numbers or payment details. The purchase dialog is a system level overlay that your app cannot customize beyond the product information you provide.

One consequence of this architecture is that billing functionality depends on the version of Google Play Services installed on the device. If the user has an old version, some billing features may not be available. PBL

provides the `isFeatureSupported()` method to check for specific feature availability before calling feature specific APIs.

Your Backend Server

Your server verifies purchases by calling the Google Play Developer API, stores entitlement records in your database, and processes Real Time Developer Notifications (RTDNs) for subscription state changes. Server side verification is essential. Without it, a user could modify local purchase data and get free access to your content.

Your backend is responsible for:

- Receiving purchase tokens from your app and verifying them against the Google Play Developer API
- Storing purchase records and entitlement state in your database
- Processing RTDNs for subscription lifecycle events (renewals, cancellations, payment failures)
- Acknowledging purchases after verifying and granting entitlements
- Managing subscription operations (cancel, defer, revoke) when needed

You can build your backend in any language. The Google Play Developer API is a standard REST API, and Google provides client libraries for Java, Python, Node.js, Go, and other languages. This book shows server side examples in Kotlin for consistency, but the concepts translate directly to any backend language.

The Google Play Store (Server Side)

Google's servers process payments, manage subscription renewals, handle payment recovery (grace periods, account holds), send RTDNs through Cloud Pub/Sub, and serve as the authoritative source of truth for all purchase state.

Google's servers handle the parts of billing that happen without your app's involvement: automatic subscription renewals, payment retry logic during grace periods and account holds, sending renewal receipts to users, and managing the Play Store's subscription management screen where users can cancel, pause, or modify their subscriptions.

Core Vocabulary

Before going further, you need to understand the key terms that appear throughout this book:

Product ID (also called SKU)

A unique string identifier you assign to each product in the Play Console. For example, `premium_monthly` or `100_coins`. Once created, a Product ID cannot be reused, even after deletion. Choose your naming convention carefully.

Purchase Token

A string that Google generates when a user completes a purchase. This is the primary identifier you use to verify and manage a purchase on your backend. Purchase tokens are unique per transaction. When a user

upgrades or downgrades a subscription, a new purchase token is generated.

Order ID

A Google generated identifier for a financial transaction. For subscriptions, the Order ID includes a sequence number that increments with each renewal (e.g., `GPA.1234-5678-9012..0` for the initial purchase, `GPA.1234-5678-9012..1` for the first renewal). One purchase token maps to many Order IDs over the life of a subscription.

Entitlement

The access or content you grant to a user after a verified purchase. Google does not manage entitlements for you. Your app and backend are responsible for tracking what each user has access to based on their purchase history.

Acknowledgement

A required confirmation you send to Google after granting an entitlement. You must acknowledge every purchase within 3 days, or Google will automatically refund it. This mechanism ensures that users only pay for purchases that your app has actually fulfilled.

Consumption

For consumable products (like virtual currency), consumption tells Google that the user has "used up" the product and can purchase it again. Unlike acknowledgement, consumption is specific to consumable one time products. Consuming a product implicitly acknowledges it.

Base Plan

A pricing configuration within a subscription product. Each base plan defines a billing period (weekly, monthly, yearly, etc.) and a price. A single subscription product can have multiple base plans, for example a monthly plan at $9.99 and an annual plan at $79.99.

Offer

A promotional pricing phase attached to a base plan. Offers include free trials, introductory pricing, and upgrade discounts. Each offer generates an offer token that your app uses when launching the purchase flow.

Real Time Developer Notification (RTDN)

A push notification sent by Google through Cloud Pub/Sub when a purchase or subscription event occurs. RTDNs tell your backend about renewals, cancellations, payment failures, and other lifecycle events without requiring you to poll the API.

Linked Purchase Token

When a user upgrades or downgrades a subscription, Google creates a new purchase token. The new token contains a `linkedPurchaseToken` field pointing back to the original token. Your backend must follow these links to properly deactivate old entitlements and activate new ones.

Product Types Overview

Google Play Billing supports two main product types:

One Time Products

These are products the user purchases once. They come in two flavors:

- **Consumable**: Products that can be purchased repeatedly because they get "used up." Virtual currency, extra lives, or boost items are typical examples. After the user consumes the product, they can buy it again.
- **Non consumable**: Products purchased once that provide permanent access. Removing ads, unlocking a premium level, or buying a filter pack are common examples. The user owns these indefinitely.

Subscriptions

Subscriptions grant access to content or features for a recurring period. Google Play supports several subscription models:

- **Auto renewing subscriptions**: The standard model. Google automatically charges the user at the end of each billing period until they cancel.
- **Prepaid plans**: The user pays upfront for a fixed period with no automatic renewal. They can manually "top up" to extend access.
- **Installment subscriptions**: Available in select markets (Brazil, France, Italy, Spain). Users commit to a minimum number of payments but can cancel after the commitment period.

Subscriptions have a hierarchical structure: a Subscription contains one or more Base Plans, and each Base Plan can have Offers attached. You will explore this hierarchy in depth in Chapter 4.

How Product Types Differ in Practice

The key difference between one time products and subscriptions goes beyond "one payment vs. recurring payments." Subscriptions are significantly more complex because they have a lifecycle. A subscription can be active, paused, in a grace period, on hold, canceled but not yet expired, or fully expired. Each state requires different handling in your app and on your backend.

One time products have a simpler lifecycle: the user buys them, you verify and acknowledge (or consume), and you grant access. The main complexity is handling pending transactions, where the payment has not yet completed.

For most apps that sell subscriptions, you will spend the majority of your billing development time handling subscription lifecycle events. Parts III through V of this book cover this in detail.

When You Must Use Google Play Billing

Google's policy is straightforward: if your app is distributed through the Google Play Store and sells digital goods or services, you must use Google Play Billing. This includes:

- App features or functionality (premium upgrades, ad removal)
- Digital content (ebooks, music, videos, game levels)
- Subscription services (streaming, cloud storage, dating services)
- Virtual currencies and in game items
- Access to premium content or services

You do **not** need to use Google Play Billing for:

- Physical goods (use any payment processor)
- Services performed in the physical world (ride sharing, food delivery, cleaning services)
- Peer to peer payments
- Content that can be consumed outside the app (e.g., songs that can play on external devices)
- Certain regional exceptions where alternative billing is permitted (covered in Chapter 18)

Violating this policy risks your app being removed from the Play Store. When in doubt, consult the Google Play developer policy documentation or contact Google Play support.

There are exceptions to this rule. In certain regions (South Korea, the European Economic Area, India, and the United States), regulatory requirements allow developers to offer alternative billing systems alongside or instead of Google Play Billing. Chapter 18 covers these alternative billing options in detail, including the specific APIs and compliance requirements for each region.

The Purchase Flow at a High Level

Before diving into the details in later chapters, it helps to understand the end to end flow of a purchase:

1. **Product query**: Your app asks PBL for the current details of one or more products. PBL returns pricing, descriptions, and available offers.
2. **Display**: Your app shows the products to the user with localized prices.
3. **Purchase initiation**: The user taps "Buy." Your app builds purchase parameters and calls `launchBillingFlow()`.
4. **Google Play dialog**: The system shows the Google Play purchase dialog. The user selects a payment method and confirms.
5. **Payment processing**: Google processes the payment. For most payment methods, this is instant. For some (like cash payments at convenience stores), the purchase enters a "pending" state.
6. **Callback**: PBL notifies your app of the result through the `PurchasesUpdatedListener`.
7. **Backend verification**: Your app sends the purchase token to your backend. Your backend calls the Google Play Developer API to verify the purchase is legitimate.
8. **Entitlement grant**: After verification, your backend records the entitlement and tells your app to unlock the content.
9. **Acknowledgement**: Your backend (or your app as a fallback) acknowledges the purchase within 3 days.
10. **Ongoing management**: For subscriptions, your backend processes RTDNs for renewals, cancellations, and other lifecycle events.

This flow applies to both one time products and subscriptions. The complexity difference lies in step 10: one time products are done after acknowledgement, while subscriptions require ongoing lifecycle management that can span months or years.

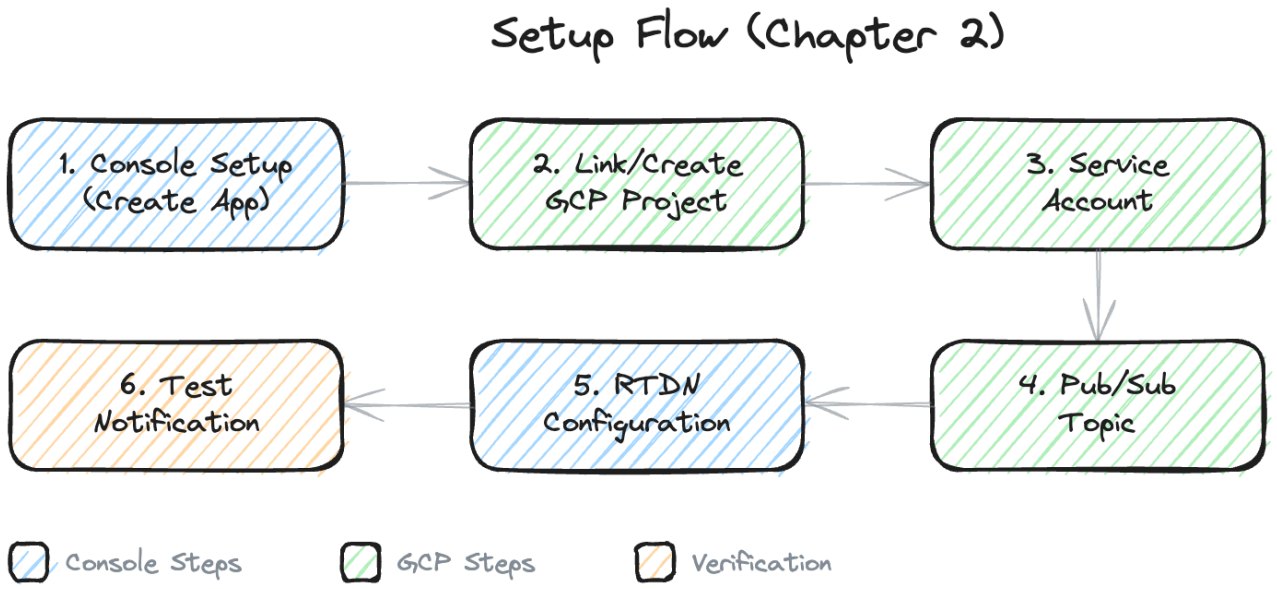
What is Ahead

Now that you understand what Google Play Billing is and how its pieces fit together, the next chapter walks you through setting everything up: your developer account, your first product, API access, and Real Time Developer Notifications. By the end of Chapter 2, you will have a working environment ready for development.

13

Chapter 2: Setting Up Your Environment

Before you write any billing code, you need a working environment. This means a Google Play Developer Account, a configured app on the Play Console, products to sell, API access for your backend, and Real Time Developer Notifications for keeping your server in sync. This chapter walks you through each step.



Google Play Developer Account

If you do not already have a Google Play Developer Account, you need one. Go to the Google Play Console (play.google.com/console) and follow the registration process. There is a one time $25 registration fee.

Individual vs Organization Accounts

Google offers two account types, and the choice matters more than you might expect.

An **individual account** is tied to a single person. You provide your legal name, and that name may appear on your Play Store listing. Individual accounts work fine for solo developers shipping free apps, but they come with limitations. If you leave a company or transfer ownership of an app, the process is more complicated because the account belongs to you personally.

An **organization account** is tied to a business entity. You provide your organization name, a DUNS number (for businesses in the US), and documentation proving the organization exists. Organization accounts unlock additional Play Console features, including managed publishing and the ability to add team members with granular role permissions. For any app that sells subscriptions or has multiple contributors, an organization account is the better choice.

Identity Verification

Both account types require identity verification, but the timeline differs. Individual accounts typically verify within a few days. Organization accounts can take up to two weeks because Google verifies the business

entity independently. Plan for this delay if you are setting up a new account. You cannot create products or upload apps until verification completes.

During verification, Google may request additional documentation. Keep your business registration documents, government issued ID, and proof of address accessible. Responding quickly to these requests shortens the overall timeline.

Setting Up the App Listing

Once your account is verified, create your app in the Play Console. Go to **All apps > Create app** and fill out the basic details: app name, default language, app type (app or game), and whether it is free or paid. You also need to complete several declarations about content, ads, and target audience.

For the store listing itself, you need a short description, full description, app icon, feature graphic, and at least two screenshots. If you are only setting up for internal testing, placeholder content works. However, certain fields like the privacy policy URL are required before you can publish to any track, including internal testing. A simple hosted privacy policy page is sufficient to unblock the process.

Google Payments Center Profile

Before you can sell anything, you need a Google Payments Center merchant profile linked to your developer account. This is where Google sends your revenue.

1. In the Play Console, go to **Setup > Payments profile**.
2. If you do not have a payments profile yet, follow the prompts to create one.
3. Provide your business information, bank account details, and tax information.
4. Wait for verification. Google may take a few days to verify your payment details.

Without a verified payments profile, you cannot create paid products or publish apps with in app purchases.

Supported Countries and Tax Requirements

Google supports merchant profiles in over 60 countries, but availability varies. Check Google's current list of supported merchant locations before assuming your country qualifies. If your business operates in a country that is not supported, you may need to establish a legal entity in a supported region.

Tax ID requirements depend on your country. In the United States, you need an EIN (Employer Identification Number) for businesses or an SSN for individual accounts. Google uses this information for tax reporting and will issue a 1099 at year end if your earnings exceed the IRS threshold. In the European Union, you need a valid VAT number if your business is VAT registered. Other countries have their own requirements. Fill out the tax information forms completely during setup because incomplete tax details can block payouts later.

Payout Schedule and Thresholds

Google pays out revenue on a monthly basis. Earnings from a given month become available for payout approximately 30 days after the month ends. For example, revenue earned in March is typically paid out by the end of April. The minimum payout threshold varies by currency, but for USD it is $1.

Payouts go to the bank account you configure in the Payments Center. You can set up wire transfers for most countries. Double check your bank details before saving because incorrect information can delay payments by an entire cycle. You can also set a custom payout threshold if you prefer to accumulate earnings before receiving a transfer.

Keep in mind that Google takes a 15% commission on the first $1M of annual revenue per developer account, and 30% beyond that. These percentages apply to subscription revenue as well, though subscriptions drop to 15% after 12 months of continuous paid service from a given subscriber.

Adding the Play Billing Library Dependency

Add the Play Billing Library to your app module's `build.gradle.kts`:

```
dependencies {
    val billingVersion = "8.0.0"
    implementation(
        "com.android.billingclient:billing-ktx:$billingVersion"
    )
}
```

The `billing-ktx` artifact includes the core billing library plus Kotlin coroutine extensions. If you prefer callback based APIs, you can use `billing` instead of `billing-ktx`, but the KTX extensions make async operations significantly cleaner. Throughout this book, all client side code examples use the KTX variant.

Version Management

PBL 8.x is the current major version at the time of writing. Google follows semantic versioning for the billing library, so minor version bumps (8.0 to 8.1) add features without breaking changes, while major version bumps (7.x to 8.x) can include breaking API changes.

Pin your billing library version explicitly rather than using dynamic version ranges like `8.+`. Dynamic versions can cause builds to break unexpectedly when a new release introduces changes you have not tested against. When a new PBL version ships, read the release notes, update the version string, test your purchase flows, and then commit the change.

You can check for new releases on the Google Maven repository or in the Android developer documentation. If your project uses a version catalog (`libs.versions.toml`), define the billing version there:

```
// In libs.versions.toml
[versions]
billing = "8.0.0"

[libraries]
billing-ktx = {
    module = "com.android.billingclient:billing-ktx",
    version.ref = "billing"
}
```

Then reference it in your `build.gradle.kts`:

```
dependencies {
    implementation(libs.billing.ktx)
}
```

Compatibility and minSdk Requirements

PBL 8 requires a `minSdk` of at least 21 (Android 5.0). If your app targets a lower API level, you need to either raise your `minSdk` or stay on an older PBL version. In practice, Android 5.0 covers over 99% of active devices, so this is rarely a meaningful constraint.

PBL 8 also requires the Google Play Store app to be installed and up to date on the user's device. This is relevant if you distribute your app outside of Google Play (for example, through sideloading or alternative stores). Devices without the Play Store cannot use the Play Billing Library at all.

Sync your Gradle project and verify the dependency resolves. If you are behind a corporate proxy or using a custom Maven repository, ensure that `google()` and `mavenCentral()` are in your `settings.gradle.kts` repository configuration. If the dependency fails to resolve, check that your project's repository block includes both sources and that your network allows access to `dl.google.com`.

Uploading Your App to the Internal Test Track

You cannot test in app purchases with a locally installed debug build. Google Play Billing requires your app to be uploaded to the Play Console, even for testing.

Understanding Test Tracks

Google Play offers three test tracks, each serving a different purpose:

Internal testing is limited to up to 100 testers that you add by email. Releases publish in minutes, sometimes seconds. There is no review process. This is the track you should use during active development. It gives you the fastest iteration cycle and requires the least amount of store listing completeness.

Closed testing lets you create tester groups that are larger than the internal testing limit. You can define up to 200 email lists, each containing thousands of testers. Closed test releases go through a lightweight review process that typically takes hours, not days. Use closed testing when you need broader beta feedback from a controlled audience before a public launch.

Open testing makes your app available to anyone who visits your store listing and opts in. Open test releases go through the standard review process, similar to production releases. Use open testing when you want public beta feedback, stress testing at scale, or when you want to validate your store listing and conversion rates before a full launch.

For the purpose of setting up your billing development environment, the internal test track is sufficient. You will expand to other tracks later when you are ready for broader testing.

Uploading to Internal Testing

1. In the Play Console, create a new app (or use an existing one).
2. Fill out the required store listing information. For testing purposes, placeholder content works fine, but you still need a privacy policy URL.
3. Build a signed APK or App Bundle. You need a release signing configuration, even for the internal test track. If you use Play App Signing (recommended), you upload your app signing key during your first upload.
4. Go to **Testing > Internal testing** and create a new release.
5. Upload your signed artifact.
6. Add testers by email address. These email addresses must be Google accounts. Each tester must accept the testing invitation link before they can install the app from the Play Store.
7. Roll out the internal testing release.

After rolling out, it takes a few minutes for the release to become available. Your testers can then install the app from the Play Store using the opt in link. The important thing is that Google Play recognizes your app and its package name, which enables the billing APIs to function.

Creating Your First Product in the Play Console

With your app uploaded, you can create products. Let us start with a simple one time product.

1. Go to **Monetize > Products > In-app products** in the Play Console.
2. Click **Create product**.
3. Enter a **Product ID** (e.g., `remove_ads`). This ID is permanent and cannot be changed or reused after activation, so choose carefully.
4. Enter a name and description. These are what users see in the purchase dialog.
5. Set a **default price**. Google will auto convert to other currencies, but you can customize individual country prices.
6. Set the **tax and compliance settings** appropriate for your product type.
7. **Activate** the product.

Naming Conventions

Product IDs are permanent. Once you create a product ID and activate it, you cannot change it, delete it, or reuse it, even if you deactivate the product later. This means your naming convention matters from day one.

A common pattern is `<category>_<descriptor>`, for example `premium_monthly`, `coins_500`, or `remove_ads`. Keep IDs lowercase with underscores. Avoid encoding prices or promotional details in the ID because those things change over time while the ID does not.

For subscriptions, you will also name base plans and offer IDs. A consistent scheme like `premium_monthly` (subscription), `standard` (base plan), and `intro_7d_free` (offer) keeps things readable when you are debugging purchase flows six months from now.

Pricing Strategies and Regional Pricing

When you set a default price in your home currency, Google automatically converts it to local prices in all other countries. These auto converted prices follow Google's price tiers, which are designed to land on clean numbers in each currency (for example, $4.99 USD might convert to 549 JPY or 4.49 EUR).

You can override the auto converted price for any country individually. This is useful when you want to optimize for purchasing power in specific markets. A price that feels reasonable in the United States might be too expensive in Southeast Asia or South America. Adjusting regional prices can significantly increase conversion rates in those markets.

Google also lets you run pricing experiments through the Play Console, where you A/B test different price points for a given product. This is covered in more detail in later chapters, but be aware that the feature exists when you are initially setting up products.

Activating Products

For subscriptions, the creation process is similar but involves creating a Subscription, then adding Base Plans and Offers within it. You will learn the subscription hierarchy in Chapter 4.

A product must be in the **Active** state before your app can query or sell it. New products may take a few minutes to propagate after activation. If you query for a product immediately after activating it and get an empty result, wait a few minutes and try again. This propagation delay is normal and only affects newly created or recently modified products.

Configuring Google Play Developer API Access

Your backend needs to call the Google Play Developer API to verify purchases, check subscription status, and acknowledge transactions. This requires a service account.

Creating a Service Account

1. Go to the Google Cloud Console (console.cloud.google.com).
2. Select or create a project. Link this GCP project to your Play Console app.
3. Go to **IAM & Admin > Service Accounts**.

4. Click **Create Service Account**.
5. Give it a descriptive name (e.g., `play-billing-backend`).
6. Create a JSON key for this service account and download it securely.
7. Go to the Play Console, then **Setup > API access**.
8. Link your GCP project if you have not already.
9. Grant the service account the appropriate permissions.

Role Permissions

The Play Console lets you assign granular permissions to each service account. For billing verification and subscription management, you need at least:

- **View financial data**: Allows reading revenue reports and financial information.
- **Manage orders and subscriptions**: Allows verifying purchases, acknowledging transactions, revoking subscriptions, and issuing refunds via the API.

Avoid granting broader permissions like **Admin** unless you have a specific need. Follow the principle of least privilege. If your backend only needs to verify purchases and check subscription status, those two permissions are sufficient.

If you have multiple services that access the Play Developer API for different purposes (for example, one service for purchase verification and another for analytics), create separate service accounts with appropriate permissions for each.

Key Management

The JSON key file is a secret. Do not commit it to version control, do not share it over email, and do not embed it in client side code. Store it in a secrets manager like Google Secret Manager, AWS Secrets Manager, or HashiCorp Vault.

Rotate your service account keys periodically. Google allows up to 10 keys per service account, so you can create a new key, deploy it, and then delete the old one without downtime. If a key is ever compromised, delete it immediately from the Google Cloud Console and create a replacement.

For production environments running on Google Cloud (Cloud Run, GKE, App Engine), consider using workload identity federation instead of downloaded key files. This approach lets your service authenticate as the service account without managing key files at all.

Testing API Access

Verify your setup by making a simple API call. Using the Google API client library for your backend language, authenticate with the service account and call any read endpoint, such as listing in app products:

```
// Server side Kotlin using google-api-services
val transport = GoogleNetHttpTransport.newTrustedTransport()
val credential = GoogleCredentials
    .fromStream(FileInputStream("service-account.json"))
    .createScoped(
        "https://www.googleapis.com/auth/androidpublisher"
    )
val publisher = AndroidPublisher.Builder(
    transport,
    GsonFactory.getDefaultInstance(),
    HttpCredentialsAdapter(credential)
).setApplicationName("your-app").build()
```

If the call succeeds, your service account is properly configured.

Setting Up Cloud Pub/Sub for Real Time Developer Notifications

Real Time Developer Notifications (RTDNs) let Google push purchase and subscription events to your backend as they happen. Without RTDNs, you would need to poll the API constantly to detect changes like renewals, cancellations, or payment failures.

RTDNs flow through Google Cloud Pub/Sub. Here is how to set it up:

Step 1: Create a Pub/Sub Topic

1. In the Google Cloud Console, go to **Pub/Sub > Topics**.
2. Click **Create Topic**.
3. Name it something descriptive (e.g., `play-billing-notifications`).
4. Leave default settings and create.

Step 2: Grant Google Publish Access

Google needs permission to publish messages to your topic. Add the Google Play service account as a publisher:

1. On your topic, click the **Permissions** tab.
2. Add the principal: `google-play-developer-notifications@system.gserviceaccount.com`
3. Assign the role: **Pub/Sub Publisher**.

Step 3: Create a Subscription

You need a subscription to receive messages from the topic. You have two options, and the choice depends on your infrastructure.

Push subscriptions work by having Google Cloud send an HTTP POST request to your endpoint for each message. The message body contains a base64 encoded JSON payload. Your endpoint must return a 2xx status code to acknowledge the message. If it returns an error or times out, Pub/Sub retries delivery with exponential backoff. Push subscriptions are the simpler option if you have a publicly accessible HTTPS endpoint. You do not need to manage polling loops or maintain persistent connections.

Pull subscriptions work by having your server explicitly request messages from the subscription. Your server calls the Pub/Sub API (or uses a client library) to pull a batch of messages, process them, and then acknowledge each one. Pull subscriptions give you more control over processing rate and are a better fit if your server is behind a firewall, if you want to process messages in batches, or if you need to control concurrency precisely.

For most production setups, a push subscription is the simplest approach and the one this book assumes in later chapters.

Dead Letter Queues

If your endpoint repeatedly fails to acknowledge a message, the message keeps getting retried. By default, Pub/Sub retries indefinitely, which can create noise and waste resources. To handle this, configure a dead letter topic. After a configurable number of delivery attempts (the default maximum is between 5 and 100, you choose), Pub/Sub forwards the undeliverable message to the dead letter topic instead of retrying forever.

Monitor your dead letter topic regularly. Messages that land there represent events your system failed to process, which could mean missed subscription renewals or unacknowledged purchases that need manual attention.

Message Format Preview

Each RTDN message from Google contains a JSON payload with several fields. The most important are `version`, `packageName`, `eventTimeMillis`, and then one of `subscriptionNotification`, `oneTimeProductNotification`, or `testNotification`. Here is a simplified example of what a subscription notification looks like:

```
// Decoded Pub/Sub message payload
{
    "version": "1.0",
    "packageName": "com.example.app",
    "eventTimeMillis": "1234567890123",
    "subscriptionNotification": {
        "version": "1.0",
        "notificationType": 4,
        "purchaseToken": "abc123...",
        "subscriptionId": "premium_monthly"
    }
}
```


22

The `notificationType` integer tells you what happened (renewal, cancellation, pause, etc.). You will learn how to handle each notification type in Chapter 10.

Step 4: Enable RTDNs in Play Console

1. In the Play Console, go to **Monetize > Monetization setup**.
2. Under **Real time developer notifications**, enter the full Pub/Sub topic name: `projects/your-project-id/topics/play-billing-notifications`.
3. Save.

Verifying Your Setup with a Test Notification

After configuring RTDNs, send a test notification to verify the pipeline works:

1. In the Play Console's RTDN settings, click **Send test notification**.
2. Check your Pub/Sub subscription for the test message.
3. If using a push subscription, verify your endpoint received the HTTP POST.

The test notification has a `testNotification` field instead of the usual `subscriptionNotification` or `oneTimeProductNotification` fields. Your notification handler should recognize and acknowledge test messages without attempting to process them as real events.

If the test notification does not arrive, check:

- The Pub/Sub topic name in the Play Console matches exactly.
- The Google Play service account has Publisher permissions on the topic.
- Your subscription is active and your endpoint (for push) is reachable and returns a 200 status.

Troubleshooting Common Setup Problems

Setting up the billing environment involves multiple systems (Play Console, Google Cloud, Pub/Sub, your app, your backend), and things can go wrong at any junction. Here are the most common problems and how to resolve them.

"This version of the app is not configured for billing through Google Play." This error appears when you try to launch a purchase flow from an app that Google Play does not recognize. The usual causes are: your app is not uploaded to any test track, the version code of your locally installed build does not match the one on the Play Console, or the signing key does not match. Make sure you are running the exact same signed build that you uploaded, or at minimum ensure the package name and signing certificate match.

Products return empty when queried. If `queryProductDetails` returns an empty list, check that the product is in the Active state in the Play Console. Also verify that you are querying with the correct product ID and product type (INAPP vs SUBS). Newly activated products can take several minutes to propagate. If you just activated a product, wait and try again.

Service account API calls return 403 Forbidden. This means your service account does not have the right permissions, or the GCP project is not linked to the correct Play Console app. Go to the Play Console under **Setup > API access** and verify the link. Then check that the service account has the necessary permissions granted in the Play Console, not just in the GCP IAM settings. Play Console permissions and GCP IAM permissions are separate systems.

Pub/Sub test notification never arrives. First confirm the topic name in the Play Console matches the fully qualified topic name exactly (`projects/your-project-id/topics/your-topic-name`). Then verify that the Google Play service account (`google-play-developer-notifications@system.gserviceaccount.com`) has the Pub/Sub Publisher role on your topic. If using a push subscription, check that your endpoint is publicly accessible over HTTPS and returns a 200 status code. Check the Pub/Sub monitoring dashboard in the Google Cloud Console for delivery error metrics.

"The developer account associated with this app is not eligible for testing." This happens when the Google account you are testing with is the same as the developer account. The account that owns the Play Console cannot make test purchases. Use a different Google account for testing, and add that account as a license tester under **Setup > License testing** in the Play Console.

Gradle fails to resolve the billing dependency. Verify that your `settings.gradle.kts` includes both `google()` and `mavenCentral()` in the `repositories` block under `dependencyResolutionManagement`. If you are behind a corporate firewall, you may need to configure a proxy in your `gradle.properties` file or use an internal Maven mirror that proxies the Google Maven repository.

Your Environment Checklist

Before moving on, verify you have completed each step:

- ☐ Google Play Developer Account registered and verified
- ☐ Google Payments Center profile linked and verified
- ☐ Play Billing Library 8.x added to your app dependencies
- ☐ App uploaded to the internal test track
- ☐ At least one product created and activated in the Play Console
- ☐ Service account created with appropriate Play Console permissions
- ☐ Cloud Pub/Sub topic created with Google publish permissions
- ☐ Pub/Sub subscription (push or pull) configured
- ☐ RTDNs enabled in Play Console pointing to your topic
- ☐ Test notification sent and received successfully

With all of this in place, you have a complete environment for developing, testing, and eventually shipping your billing integration. In the next chapter, you will start building: creating and selling one time products.

Chapter 3: One Time Products

One time products are the simplest form of monetization on Google Play. A user pays once, and you deliver something: a bag of coins, a premium feature unlock, or access to a content pack. Despite that simplicity, the implementation has real depth. You need to understand the purchase lifecycle, handle edge cases like pending transactions, manage consumption correctly, and work with newer features like multi quantity purchases and pre orders.

This chapter covers everything you need to build a complete one time product integration with the Play Billing Library 8.x.

Consumable vs. Non Consumable Products

One time products fall into two categories based on whether the user can purchase them again.

Consumable products are items the user "uses up." Once consumed, the product is no longer in the user's purchase history, and they can buy it again. Virtual currency, extra lives, temporary boosts, and limited use items are all consumables. When a user buys 100 coins, you add those coins to their balance and then consume the purchase. The next time they want coins, they can buy the same product again.

Non consumable products are items the user buys once and keeps permanently. Removing ads, unlocking a level pack, or enabling a premium theme are typical examples. You do not consume these purchases. They remain in the user's purchase history indefinitely, and the user cannot repurchase them (unless you revoke or refund the purchase).

The distinction matters because it determines which API call you make after granting the entitlement. For consumable products, you call `consumePurchase()`. For non consumable products, you call `acknowledgePurchase()`. Both calls satisfy Google's requirement that you process every purchase within 3 days, but they have different outcomes. You will see the details of each later in this chapter.

One practical note: Google Play does not enforce the consumable vs. non consumable distinction at the product configuration level. You decide how to treat each product in your code. A product becomes consumable because your app calls `consumePurchase()` on it, not because of a flag in the Play Console. That said, PBL 8.x does introduce product configuration options in the Console that help you signal your intent, which you will see next.

Creating One Time Products in the Play Console

To sell a one time product, you first need to create it in the Google Play Console. You did this briefly in Chapter 2. Here is the full process with all the details.

1. Open the Play Console and navigate to your app.
2. Go to **Monetize > Products > In-app products**.
3. Click **Create product**.

4. Enter a **Product ID**. This is a permanent identifier. Once you create a product with a given ID, you cannot reuse that ID even if you delete the product later. Use a clear naming convention like `coins_100`, `remove_ads`, or `level_pack_desert`.
5. Enter the product **Name** and **Description**. These appear in the Google Play purchase dialog that users see, so write them clearly.
6. Configure pricing (covered in the next section).
7. Set the tax category and regional availability.
8. Save and **Activate** the product.

A product must be in the **Active** state before your app can query it or sell it. After activation, it can take a few minutes for the product to propagate through Google's systems.

Naming Conventions

Choose a naming scheme and stick with it across your entire product catalog. Some approaches that work well:

- **Category prefix**: `consumable_coins_100`, `permanent_remove_ads`
- **Feature based**: `coins_100`, `coins_500`, `unlock_themes`, `remove_ads`
- **Versioned**: `coins_100_v2` (useful if you need to retire and replace a product)

Avoid overly generic IDs like `product_1` or `item_a`. When you are debugging a purchase issue six months from now, a descriptive ID saves you time.

Product Configuration: Pricing, Availability, Tax Categories

Pricing

When you create a one time product, you set a **default price** in your primary currency. Google automatically converts this price to all supported currencies using its own exchange rates and rounding rules. You can override individual country prices if you want more control.

A few things to keep in mind:

- Google adjusts converted prices to "pretty" price points (e.g., $0.99, $1.49) rather than exact exchange rate conversions.
- You can set country specific prices manually for any market where you want precise control.
- Price changes take effect immediately for new purchases. Existing pending transactions retain the old price.
- The minimum price varies by country. In the US, it is $0.49 for one time products.

Regional Availability

By default, a product is available in all countries where your app is published. You can restrict availability to specific countries if needed. This is useful when a product only makes sense in certain markets or when legal

restrictions apply.

Tax Categories

Google handles tax collection and remittance in most countries. You need to assign each product a tax category that matches its content type. The available categories include:

- **Digital content**: Games, apps, music, video, books
- **Software as a service**: Cloud storage, productivity tools
- **Digital newspapers/periodicals**: News subscriptions, magazines

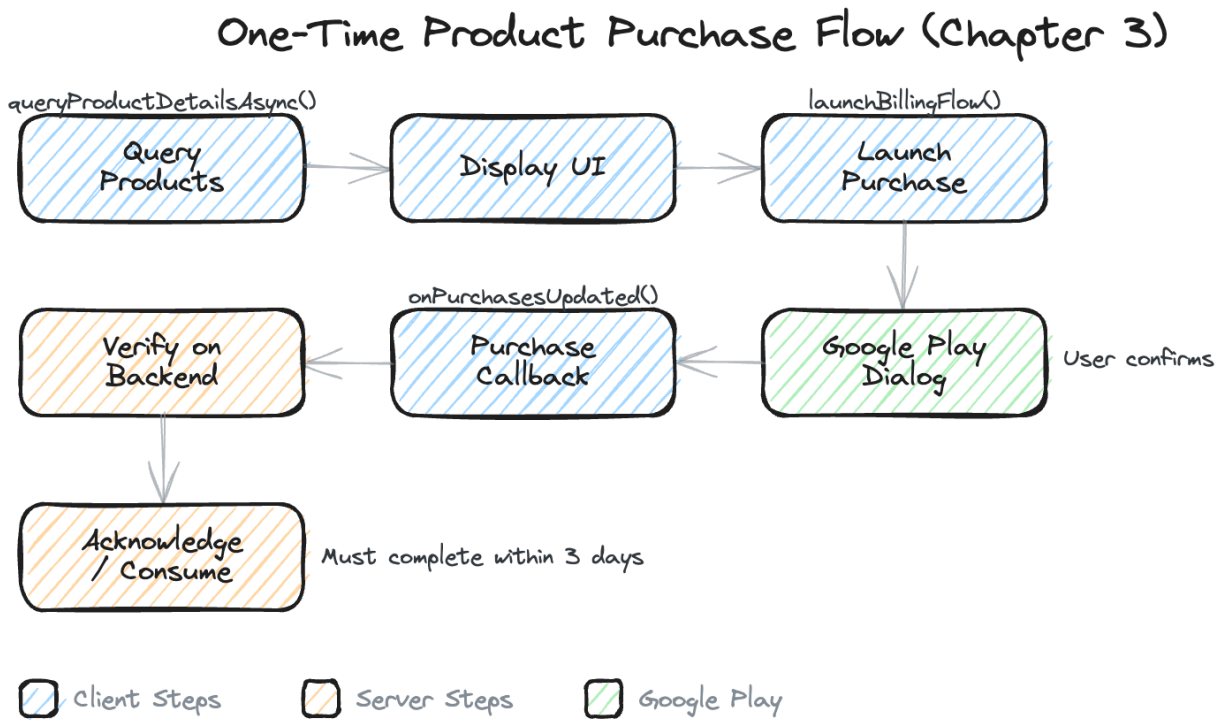
The tax category affects which tax rates Google applies in different jurisdictions. Choosing the wrong category can lead to incorrect tax collection, so pick the one that most accurately describes your product.

The One Time Product Purchase Lifecycle

Every one time product purchase moves through a predictable lifecycle. Understanding this flow is important because your code needs to handle each stage correctly.

1. **Query products**: Your app calls `queryProductDetailsAsync()` to fetch product information (name, price, description) from Google Play.
2. **Display products**: You show the product information to the user in your UI.
3. **Launch purchase flow**: When the user taps "Buy," your app calls `launchBillingFlow()` to start the Google Play purchase dialog.
4. **Purchase result**: Google Play returns the result through your `PurchasesUpdatedListener`. The purchase state is either `PURCHASED` or `PENDING`.
5. **Verify on server**: Your backend verifies the purchase token with the Google Play Developer API.
6. **Grant entitlement**: After verification, you give the user what they paid for.
7. **Consume or acknowledge**: You call `consumePurchase()` for consumable products or `acknowledgePurchase()` for non consumable products. This must happen within 3 days or Google refunds the purchase.

If a purchase enters the `PENDING` state (common for payment methods like cash or delayed payment in certain regions), your app must wait for the purchase to transition to `PURCHASED` before granting the entitlement. You will find coverage of pending transactions in detail later in this chapter.



Querying Product Details with queryProductDetailsAsync()

Before you can show products to the user or launch a purchase, you need to fetch product details from Google Play. This gives you the localized product name, description, and price.

```
suspend fun queryOneTimeProducts(
    billingClient: BillingClient
): List<ProductDetails> {
    val productList = listOf(
        QueryProductDetailsParams.Product.newBuilder()
            .setProductId("coins_100")
            .setProductType(ProductType.INAPP)
            .build(),
        QueryProductDetailsParams.Product.newBuilder()
            .setProductId("remove_ads")
            .setProductType(ProductType.INAPP)
            .build()
    )
    val params = QueryProductDetailsParams.newBuilder()
        .setProductList(productList)
        .build()
    val result = billingClient.queryProductDetails(params)
    return result.productDetailsList.orEmpty()
}
```

A few things to note about this code:

- One time products use `ProductType.INAPP`. Subscriptions use `ProductType.SUBS`.
- The `queryProductDetails()` function is the coroutine extension from the `billing-ktx` artifact. The callback version is `queryProductDetailsAsync()`.
- Always check the `BillingResult` response code. If it is not `BillingResponseCode.OK`, the product list may be empty or null.
- Product IDs must match exactly what you configured in the Play Console, and those products must be in the Active state.

Using ProductDetails

The `ProductDetails` object contains everything you need to display the product to the user:

```
fun displayProduct(productDetails: ProductDetails) {
    val name = productDetails.name
    val description = productDetails.description
    val price = productDetails
        .oneTimePurchaseOfferDetails
        ?.formattedPrice
    // Show name, description, and price in your UI
}
```

The `formattedPrice` string is already localized with the correct currency symbol and formatting for the user's locale. Use it directly in your UI instead of formatting the price yourself.

Launching the Purchase Flow

Once the user decides to buy, you launch the purchase flow with `launchBillingFlow()`. This opens the familiar Google Play purchase dialog.

```
fun launchPurchase(
    activity: Activity,
    billingClient: BillingClient,
    productDetails: ProductDetails
): BillingResult {
    val productDetailsParams =
        BillingFlowParams.ProductDetailsParams
            .newBuilder()
            .setProductDetails(productDetails)
            .build()
    val billingFlowParams = BillingFlowParams.newBuilder()
        .setProductDetailsParamsList(
            listOf(productDetailsParams)
        )
        .build()
    return billingClient.launchBillingFlow(
        activity, billingFlowParams
    )
}
```

Key points about `launchBillingFlow()`:

- It requires an `Activity` reference, not a `Context`. The purchase dialog is presented as a bottom sheet overlay on your activity.
- The method is synchronous and returns a `BillingResult` immediately, but this only tells you whether the purchase flow launched successfully. The actual purchase result arrives asynchronously through your `PurchasesUpdatedListener`.
- You can only have one purchase flow active at a time. Launching a second flow while one is already active returns `BillingResponseCode.DEVELOPER_ERROR`.
- The user can cancel the purchase dialog at any time. Your listener will receive `BillingResponseCode.USER_CANCELED` in that case.

Obfuscated Account and Profile IDs

You can attach identifiers to the purchase for fraud detection and backend correlation:

```
val billingFlowParams = BillingFlowParams.newBuilder()
    .setProductDetailsParamsList(listOf(productDetailsParams))
    .setObfuscatedAccountId(hashedUserId)
    .setObfuscatedProfileId(hashedProfileId)
    .build()
```

These values appear in the purchase record and in your RTDN notifications. Use hashed or obfuscated values, never raw user IDs. Google may use these for fraud detection signals. Setting them also helps your backend correlate purchases to users, especially in cases where the client cannot report the purchase result back to your server.

Detecting Purchases: PurchasesUpdatedListener and queryPurchasesAsync()

There are two ways your app learns about purchases: the real time listener and on demand queries.

PurchasesUpdatedListener

When you build your `BillingClient`, you provide a `PurchasesUpdatedListener`. This listener fires whenever a purchase flow completes, whether the user bought something, canceled, or hit an error.

```
private val purchasesUpdatedListener =
    PurchasesUpdatedListener { billingResult, purchases ->
        when (billingResult.responseCode) {
            BillingResponseCode.OK -> {
                purchases?.forEach { purchase ->
                    handlePurchase(purchase)
                }
            }
            BillingResponseCode.USER_CANCELED -> {
                // User backed out of the purchase flow
            }
            else -> {
                // Log the error for debugging
            }
        }
    }
```

The listener is your primary mechanism for handling purchases in real time. When the response code is `OK`, you receive a list of `Purchase` objects to process.

queryPurchasesAsync()

The listener only fires during active purchase flows. To catch purchases that happened while your app was closed, or purchases that were completed on another device, you need to query for them. Call `queryPurchasesAsync()` whenever your app starts or resumes:

```
suspend fun queryExistingPurchases(
    billingClient: BillingClient
): List<Purchase> {
    val params = QueryPurchasesParams.newBuilder()
        .setProductType(ProductType.INAPP)
        .build()
    val result = billingClient.queryPurchasesAsync(params)
    return result.purchasesList
}
```

This returns all purchases for the given product type that have not been consumed or acknowledged. You should call this method in the following situations:

- **On app launch**: To process any purchases made while the app was not running.
- **On `BillingClient` reconnection**: If the billing connection drops and you reconnect, query purchases again.
- **On `onResume()`**: The user might have completed a purchase through a notification or system dialog while your app was in the background.

The combination of the listener and periodic queries ensures you never miss a purchase.

Consuming vs. Acknowledging: When to Use Each

After verifying a purchase and granting the entitlement, you must tell Google you have processed it. The mechanism depends on whether the product is consumable.

Consuming a Purchase

For consumable products, call `consumePurchase()`. This does two things: it acknowledges the purchase (satisfying the 3 day requirement), and it removes the product from the user's purchase history so they can buy it again.

```
suspend fun consumePurchase(
    billingClient: BillingClient,
    purchaseToken: String
) {
    val params = ConsumeParams.newBuilder()
        .setPurchaseToken(purchaseToken)
        .build()
    val result = billingClient.consumePurchase(params)
    if (result.billingResult.responseCode
        == BillingResponseCode.OK
    ) {
        // Purchase consumed successfully
    }
}
```

Always consume on your server side if possible. If you consume on the client but the server never recorded the entitlement (due to a network issue), the user loses their purchase. Server side consumption through the Google Play Developer API is safer because your server can verify, record, and consume in a single transaction.

Acknowledging a Purchase

For non consumable products, call `acknowledgePurchase()`. This tells Google you have fulfilled the purchase, but the product stays in the user's purchase history. They cannot buy it again.

```
suspend fun acknowledgePurchase(
    billingClient: BillingClient,
    purchaseToken: String
) {
    val params = AcknowledgePurchaseParams.newBuilder()
        .setPurchaseToken(purchaseToken)
        .build()
    val result = billingClient.acknowledgePurchase(params)
    if (result.billingResult.responseCode
        == BillingResponseCode.OK
    ) {
        // Purchase acknowledged successfully
    }
}
```

Check `purchase.isAcknowledged` before calling acknowledge. If the purchase was already acknowledged (perhaps by your server), calling it again returns an error.

The 3 Day Window

Google gives you 3 days to consume or acknowledge a purchase. If you fail to do so, Google automatically refunds the purchase to the user. This protects users from paying for something that was never delivered.

In practice, you should process purchases within seconds, not days. The 3 day window is a safety net, not a target. If you are consistently relying on the 3 day window, something is wrong with your purchase processing flow.

Handling Pending Transactions (PENDING vs. PURCHASED)

Not all purchases complete immediately. In some markets, users can pay with cash at a convenience store, bank transfer, or other delayed payment methods. When a user chooses one of these options, the purchase enters the `PENDING` state.

Detecting Pending Purchases

Check the purchase state to determine how to handle each purchase:

```
fun handlePurchase(purchase: Purchase) {
    when (purchase.purchaseState) {
        Purchase.PurchaseState.PURCHASED -> {
            // Payment complete. Verify, grant, consume/ack.
            verifyAndGrantEntitlement(purchase)
        }
        Purchase.PurchaseState.PENDING -> {
            // Payment not yet complete. Do NOT grant.
            showPendingMessage(purchase)
        }
        Purchase.PurchaseState.UNSPECIFIED_STATE -> {
            // Unknown state. Log and investigate.
        }
    }
}
```

Rules for Pending Transactions

When a purchase is in the `PENDING` state:

- **Do not grant the entitlement.** The user has not paid yet.
- **Do not consume or acknowledge.** You cannot process a pending purchase.
- **Show a clear message.** Let the user know their purchase is being processed and will be fulfilled once payment is confirmed.
- **Track the pending purchase.** Store the purchase token so you can fulfill it later.

When the payment eventually completes (or fails), Google notifies your app in two ways:

1. **RTDN**: Your backend receives a `ONE_TIME_PRODUCT_PURCHASED` or `ONE_TIME_PRODUCT_CANCELED` notification through Cloud Pub/Sub.
2. **queryPurchasesAsync()**: The next time your app queries purchases, the purchase state will have changed from `PENDING` to `PURCHASED` (or it will be gone if the payment failed).

Your backend should handle the RTDN and update the user's entitlement accordingly. Your app should call `queryPurchasesAsync()` on launch and resume to pick up any state changes.

Enabling Pending Transactions

Pending transactions are enabled by default in PBL 8.x. In earlier library versions, you had to opt in by calling `enablePendingPurchases()` on the `BillingClient.Builder`. With PBL 8.x, all apps must handle the `PENDING` state correctly.

Multi Quantity Purchases

PBL 8.x supports multi quantity purchases for consumable products. Instead of buying one bag of coins at a time, a user can buy multiple in a single transaction.

Enabling Multi Quantity in the Play Console

To support multi quantity purchases for a product:

1. Go to your product in **Monetize > Products > In-app products**.
2. Enable the **multi quantity** option for the product.
3. Set the maximum quantity per purchase.

Not all products are good candidates for multi quantity. It makes sense for consumable items like currency packs or resource bundles where a user might want to stock up. It does not make sense for non consumable products like "remove ads."

Handling Multi Quantity in Code

When a user completes a multi quantity purchase, the `Purchase` object includes a `quantity` field:

```
fun handleMultiQuantityPurchase(purchase: Purchase) {
    val quantity = purchase.quantity
    val productId = purchase.products.first()
    // Grant quantity * unitValue to the user
    grantCoins(userId, quantity * COINS_PER_PACK)
    // Then consume the purchase
    consumePurchase(billingClient, purchase.purchaseToken)
}
```

Always use the `quantity` field rather than assuming a quantity of 1. If you do not support multi quantity for a given product, the quantity will always be 1, so your code remains backward compatible.

Price Calculation

The total price the user pays is the single item price multiplied by the quantity. The `Purchase` object does not contain the total price directly. If you need to display or log it, calculate it from the `ProductDetails` price and the `Purchase` quantity.

Pre Order for One Time Products (PBL 8.1+)

Starting with PBL 8.1, Google Play supports pre orders for one time products. This lets users commit to buying a product before it becomes available, similar to how pre orders work for games or apps on the Play Store.

How Pre Orders Work

1. You create a one time product in the Play Console and configure it with a release date.
2. Users can "pre order" the product before the release date. Google does not charge them immediately.
3. On the release date, Google charges the user's payment method and completes the purchase.
4. Your app receives the purchase as a normal `PURCHASED` state purchase and fulfills it.

Use Cases

Pre orders work well for:

- Seasonal content packs released on a specific date
- New game levels or expansion packs announced ahead of time
- Limited edition digital items with a scheduled launch

Implementation Considerations

From a code perspective, pre orders do not require much special handling. The purchase flow uses the same `launchBillingFlow()` API. The difference is in timing:

- Before the release date, the purchase enters a pre order state. The user is committed but not charged.
- On the release date, the purchase transitions to `PURCHASED` and your normal fulfillment flow takes over.

Your app should detect pre ordered products and show appropriate UI, such as "Pre ordered, available on [date]" instead of "Owned." Use `queryPurchasesAsync()` to detect pre ordered items.

You should also handle the case where a pre order charge fails on the release date (expired card, insufficient funds). Google will notify you through RTDNs, and the purchase will not appear as `PURCHASED`.

Multiple Purchase Options and Offers

PBL 8.x introduces purchase options and offers for one time products, bringing some of the flexibility that subscriptions have had for a while. This feature allows you to create multiple ways to buy the same product at different prices or with different conditions.

Purchase Options

A single one time product can have multiple purchase options. Each option can have:

- A different price
- A different availability window (start and end dates)
- Regional restrictions
- Different offer tags for targeting

For example, you might have a "100 Coins" product with a standard purchase option at $1.99 and a promotional purchase option at $0.99 that is only available during a sale event.

Querying Offers

When you query product details, the `ProductDetails` object may contain multiple offer details. You can access them through the `oneTimePurchaseOfferDetailsList`:

```
fun getAvailableOffers(
    productDetails: ProductDetails
): List<ProductDetails.OneTimePurchaseOfferDetails> {
    return productDetails
        .oneTimePurchaseOfferDetailsList
        .orEmpty()
}
```

Each offer detail includes its own `formattedPrice`, `offerToken`, and other metadata. When you display the product to the user, you can show the best available offer or let the user choose between options.

Launching a Purchase with a Specific Offer

To purchase a specific offer, include the `offerToken` in the billing flow parameters:

```
fun launchPurchaseWithOffer(
    activity: Activity,
    billingClient: BillingClient,
    productDetails: ProductDetails,
    offerToken: String
): BillingResult {
    val params = BillingFlowParams.ProductDetailsParams
        .newBuilder()
        .setProductDetails(productDetails)
        .setOfferToken(offerToken)
        .build()
    val flowParams = BillingFlowParams.newBuilder()
        .setProductDetailsParamsList(listOf(params))
        .build()
    return billingClient.launchBillingFlow(
        activity, flowParams
    )
}
```

If you do not specify an offer token, Google Play uses the default purchase option.

Offer Management Strategy

When working with multiple offers, keep these guidelines in mind:

- **Server driven selection**: Have your backend decide which offer to show each user. This lets you run A/B tests and personalized promotions without app updates.
- **Always validate on server**: Verify the purchase price on your backend after the transaction. The user might have received a different offer than you expected.
- **Time limited offers**: Use start and end dates in the Play Console to manage promotional pricing. The offers become available and expire automatically.

Putting It All Together

Here is a condensed example of a complete one time product purchase flow, from querying products to consuming the purchase:

```
class BillingManager(private val activity: Activity) {
    private lateinit var billingClient: BillingClient

    private val listener = PurchasesUpdatedListener {
        result, purchases ->
        if (result.responseCode == BillingResponseCode.OK) {
            purchases?.forEach { handlePurchase(it) }
        }
    }

    fun initialize() {
        billingClient = BillingClient.newBuilder(activity)
            .setListener(listener)
            .enablePendingPurchases(
                PendingPurchasesParams.newBuilder().build()
            )
            .build()
    }
}
```

```
private fun handlePurchase(purchase: Purchase) {
    when (purchase.purchaseState) {
        Purchase.PurchaseState.PURCHASED -> {
            // 1. Send purchase token to your server
            // 2. Server verifies with Play Developer API
            // 3. Server grants entitlement
            // 4. Server consumes or acknowledges
            sendToServer(purchase.purchaseToken)
        }
        Purchase.PurchaseState.PENDING -> {
            showPendingUI(purchase)
        }
    }
}
```

In production, the `sendToServer()` call triggers your backend to verify the purchase with the Google Play Developer API, record the entitlement in your database, and then consume or acknowledge the purchase through the server side API. This keeps your purchase processing reliable even if the user closes the app mid flow.

Common Pitfalls

Here are the mistakes that cause the most issues with one time product purchases:

Not querying purchases on launch. If your app only relies on the `PurchasesUpdatedListener`, you will miss purchases that completed while the app was closed. Always call `queryPurchasesAsync()` when your app starts.

Consuming before verifying. If you consume a purchase on the client before your server has verified and recorded it, and then the network call to your server fails, the user has paid but received nothing, and you cannot recover the purchase because it has been consumed. Always verify and record before consuming.

Ignoring pending state. Granting entitlements for `PENDING` purchases means giving away your product for free. The user might never complete payment.

Hardcoding prices. Always use the `formattedPrice` from `ProductDetails`. Hardcoding prices means showing incorrect amounts to users in different regions, and showing stale prices after you change them in the Play Console.

Not handling BillingClient disconnections. The connection to Google Play Services can drop at any time. Wrap your billing calls in retry logic and always check `isReady` before making calls.

Using product IDs that are too generic. Once created, product IDs are permanent. A descriptive ID saves debugging time in the future.

Skipping server side verification. Always verify purchases on your server and prefer server side consumption or acknowledgement for reliability. Client side acknowledgement is convenient during development but a single network failure between consume and server record means the user paid and got nothing.

Chapter 4: Subscriptions Deep Dive

Subscriptions are the most sophisticated product type in Google Play Billing. A one time product has a single price and a single purchase event. A subscription has billing periods, renewal logic, offers, upgrades, downgrades, grace periods, and a three tier product hierarchy that determines how everything fits together. This chapter breaks down that hierarchy, walks through every subscription model Google Play supports, and shows you how to configure and sell subscriptions in your app.

The Subscription Product Hierarchy

Google Play organizes subscriptions into three levels: Subscription, Base Plan, and Offer. Understanding this hierarchy is essential because it determines how you create products in the Play Console, how you query them with the Play Billing Library, and how you present purchasing options to your users.

Subscription

A Subscription is the top level container. It represents a single subscription product in your app, like "Premium Access" or "Cloud Storage Plan." You give it a Product ID (e.g., `premium`) and a name. The Subscription itself does not define a price or billing period. It is just a container that groups related plans together.

Think of it this way: the Subscription answers the question "what is the user subscribing to?" The details of how they pay come from the levels below.

Base Plan

A Base Plan lives inside a Subscription and defines the actual billing terms: the billing period (weekly, monthly, every two months, quarterly, every six months, yearly), the price, and whether the plan auto renews or is prepaid. A single Subscription can have multiple Base Plans. For example, your "Premium Access" subscription might have a monthly Base Plan at $9.99/month and a yearly Base Plan at $79.99/year.

Each Base Plan has its own identifier (the Base Plan ID) and its own pricing. This is where you decide if the plan auto renews or requires manual top up (prepaid). You can also create installment plans at this level for supported markets.

The key insight is that a single Subscription can offer multiple ways to pay. The user sees one product, but you can present several pricing options.

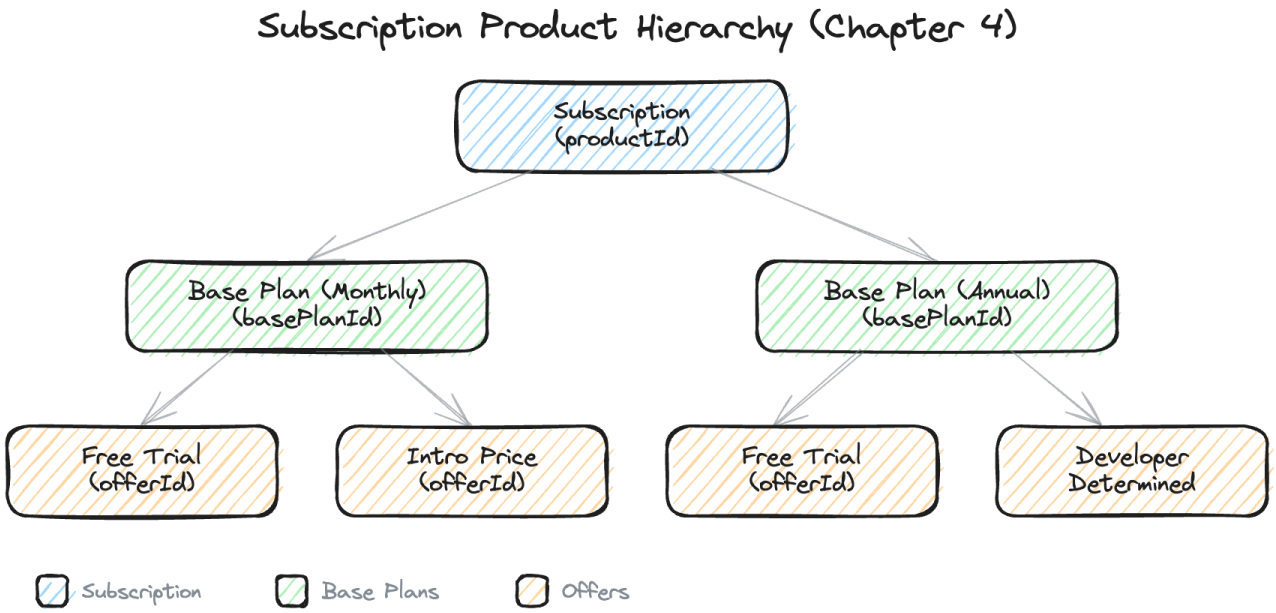
Offer

An Offer attaches to a Base Plan and provides a discount or incentive. This is where free trials, introductory pricing, and other promotional pricing live. Offers are optional. A Base Plan works just fine without any Offers, in which case the user pays the full Base Plan price from the start.

Each Offer has its own identifier (the Offer ID), eligibility criteria, and one or more pricing phases. A pricing phase defines a price and duration for one segment of the offer. For example, a "try then discount" offer might have two phases: 7 days free, then 3 months at $4.99/month, before the user moves to the full Base Plan price.

Here is how the hierarchy maps out in practice:

```
Subscription: "Premium Access" (product ID: premium)
├── Base Plan: "Monthly" (plan ID: monthly)
│   ├── Offer: "Free Trial" (7 days free)
│   └── Offer: "Intro Price" ($2.99 for 3 months)
├── Base Plan: "Yearly" (plan ID: yearly)
│   └── Offer: "First Year Discount" ($49.99 for year 1)
└── Base Plan: "Prepaid Monthly" (plan ID: prepaid-monthly)
    (no offers)
```



When you query product details with PBL 8.x, the `ProductDetails` object reflects this hierarchy. You get the subscription level information, a list of `SubscriptionOfferDetails` entries (one per Base Plan or Offer), and each entry contains pricing phases and an offer token you use to launch the purchase.

Auto Renewing Subscriptions

Auto renewing subscriptions are the standard subscription model. The user subscribes, Google charges them at the end of each billing period, and the subscription continues until the user cancels or a payment fails beyond recovery.

When you create an auto renewing Base Plan, you choose a billing period and set a price. Google handles the rest: charging the user on schedule, sending renewal notifications, and managing payment failures through grace periods and account holds.

From your app's perspective, an auto renewing subscription is straightforward. You query the product, present it to the user, launch the billing flow, and verify the purchase on your backend. After that, Google manages the

renewal cycle. Your backend receives RTDNs for each renewal event, and you can check the subscription status through the Google Play Developer API at any time.

The user can cancel an auto renewing subscription at any time through the Play Store's subscription management screen. Cancellation does not immediately revoke access. The user retains access until the end of the current billing period. After that, the subscription expires.

Prepaid Plans

Prepaid plans work differently from auto renewing subscriptions. The user pays upfront for a fixed period, and there is no automatic renewal. When the period ends, access expires unless the user manually purchases more time.

This model is common in markets where users prefer pay as you go over recurring charges, or where credit card penetration is low and users pay with prepaid cards or carrier billing. It also works well for users who want to control spending precisely.

To create a prepaid plan, you add a Base Plan to your Subscription and set its type to prepaid. You choose a billing period (e.g., 30 days, 1 year) and set the price. Users buy time on this plan, and when it runs out, they can "top up" by purchasing again.

The top up model means your app needs to handle extending an existing prepaid subscription. When a user tops up, Google extends the expiration date. You detect this through a new purchase event and update your entitlement records accordingly.

There are a few important differences between prepaid and auto renewing plans:

- **No free trials or introductory pricing**: Offers are not available for prepaid Base Plans. The user always pays the full price.
- **No grace periods or account holds**: Since there is no automatic renewal, there is no payment retry mechanism. When the period ends, it ends.
- **Shorter acknowledgement windows**: For prepaid plans shorter than 7 days, the acknowledgement window is shorter than the standard 3 days. More on this later.

Here is how you check whether a plan is prepaid when processing `ProductDetails`:

```
val offerDetails = productDetails
    .subscriptionOfferDetails ?: return

for (offer in offerDetails) {
    val installmentInfo = offer.installmentPlanDetails
    val tags = offer.offerTags

    // Prepaid plans have no recurring phase,
    // only a single non-recurring pricing phase
    val phases = offer.pricingPhases.pricingPhaseList
    val isPrepaid = phases.all {
        it.recurrenceMode ==
            RecurrenceMode.NON_RECURRING
    }
}
```

Installment Subscriptions

Installment subscriptions are available in Brazil, France, Italy, and Spain. They let users commit to a subscription for a fixed number of payments (the commitment period), after which the subscription continues on a month to month basis. This model works well in markets where users are accustomed to installment purchasing.

For example, you might create an installment plan where the user commits to 12 monthly payments of $4.99. During those 12 months, the user cannot cancel without consequences (Google may charge an early termination fee, depending on the market). After the 12 payments, the subscription auto renews monthly at the same price, and the user can cancel freely.

To create an installment plan, you add a Base Plan with the installment type and specify:

- The billing period (monthly)
- The price per installment
- The number of committed payments (the commitment count)

The commitment count is the minimum number of billing cycles the user agrees to. Google communicates this commitment clearly to the user in the purchase dialog.

From a code perspective, you can detect installment plans through the `InstallmentPlanDetails` on the offer:

```
val offerDetails = productDetails
    .subscriptionOfferDetails?.firstOrNull()
    ?: return

val installmentDetails =
    offerDetails.installmentPlanDetails

if (installmentDetails != null) {
    val commitmentCount =
        installmentDetails
            .commitmentPaymentsCount
    val renewalCount =
        installmentDetails
            .subsequentCommitmentPaymentsCount
}
```

The `commitmentPaymentsCount` tells you how many payments the user commits to initially. The `subsequentCommitmentPaymentsCount` tells you how many payments apply if the user renews into another commitment period (which may be zero, meaning month to month renewal after the initial commitment).

Subscriptions with Add Ons

Google Play supports the concept of subscription add ons, where users can purchase additional features or content on top of an existing subscription. This is useful when your app has a base subscription tier and you want to offer optional extras.

For example, a music streaming app might have a "Premium" subscription that includes ad free listening, and an add on for "Hi-Fi Audio" that the user can purchase separately on top of their existing subscription. Add ons appear as separate subscription products in the Play Console, but they are logically connected to a parent subscription.

When designing add ons, keep these points in mind:

- Each add on is a separate Subscription product with its own Base Plans and Offers.
- Your app and backend manage the relationship between the parent subscription and the add on. Google does not enforce this relationship automatically.
- The user must have an active parent subscription to use the add on, but your app is responsible for checking this.
- Billing for the add on follows the same rules as any other subscription (auto renewing, prepaid, or installment).

Your backend should validate that the parent subscription is active before granting the add on entitlement. If the parent subscription expires or is cancelled, you should revoke the add on access as well.

Creating Subscriptions and Base Plans in the Play Console

Now that you understand the hierarchy, here is how you create subscriptions in the Google Play Console:

Step 1: Create the Subscription

1. Go to **Monetize > Products > Subscriptions** in the Play Console.
2. Click **Create subscription**.
3. Enter a **Product ID** (e.g., `premium`). This is permanent and cannot be reused.
4. Enter a **Name** and **Description**. Users see these in the purchase dialog and the Play Store subscription management screen.
5. Add any **benefits** you want shown to the user.
6. Save the subscription.

At this point, you have an empty container. It has no plans and cannot be sold.

Step 2: Add Base Plans

1. Within your subscription, click **Add base plan**.
2. Enter a **Base Plan ID** (e.g., `monthly`).
3. Choose the **plan type**: auto renewing, prepaid, or installment.
4. Choose the **billing period**: weekly, monthly, every 2 months, every 3 months, every 4 months, every 6 months, or yearly.
5. Set the **default price**. Google will auto generate prices for other countries based on exchange rates. You can customize individual country prices.
6. For installment plans, set the **commitment count** (minimum number of payments).
7. Activate the Base Plan.

You can add multiple Base Plans to a single Subscription. A common pattern is to create a monthly and a yearly auto renewing plan, with the yearly plan priced at a discount compared to 12 months of the monthly plan.

Step 3: Activate the Subscription

After adding at least one active Base Plan, activate the Subscription itself. It may take a few minutes for the new subscription to propagate and become queryable from your app.

Configuring Offers

Offers let you incentivize users to subscribe by providing discounted pricing for an initial period. You attach Offers to auto renewing Base Plans. Prepaid plans do not support Offers.

Free Trials

A free trial gives the user full access for a period at no charge. After the trial ends, the subscription converts to the full Base Plan price.

To create a free trial:

1. Open a Base Plan in the Play Console.
2. Click **Add offer**.
3. Enter an **Offer ID** (e.g., `free-trial`).
4. Set the eligibility criteria (more on this below).
5. Add a pricing phase with **Free** pricing and the trial duration (e.g., 7 days, 14 days, 1 month).
6. Activate the offer.

Free trials are the most common offer type. They lower the barrier to entry by letting users try your product before committing.

Introductory Pricing

Introductory pricing charges a reduced price for an initial period. For example, $0.99/month for the first 3 months instead of the regular $9.99/month.

To create an introductory price offer:

1. Add an offer to a Base Plan.
2. Add a pricing phase with the discounted price and the number of billing periods it applies to.
3. After this phase, the subscription automatically moves to the full Base Plan price.

You can combine a free trial and introductory pricing in a single offer by adding multiple pricing phases. The phases execute in order: first the free trial, then the intro price, then the full price. An offer can have up to two pricing phases before the regular Base Plan price kicks in.

Upgrade and Downgrade Offers

When a user changes from one Base Plan to another within the same Subscription (or across Subscriptions), you can configure offers that apply specifically to these transitions. These are useful for incentivizing users to move to a higher tier or for softening the blow of a downgrade.

Upgrade and downgrade offers work like any other offer, but their eligibility is tied to the user currently holding a specific subscription. You configure these by selecting the appropriate eligibility criteria when creating the offer.

Win Back Offers

Win back offers target users who have previously subscribed but whose subscription has lapsed. These offers appear in the Play Store to encourage former subscribers to resubscribe at a discounted rate.

Win back offers are configured in the Play Console like other offers, but they target cancelled or expired subscribers. Google surfaces these offers to eligible users in the Play Store's subscription management area, making them a passive re engagement tool that works without requiring users to open your app.

47

Offer Eligibility and Developer Determined Criteria

Not every offer should be available to every user. Google Play supports several eligibility models:

Google Managed Eligibility

For new customer acquisition offers (like free trials), Google can manage eligibility automatically. Google tracks whether a user has previously redeemed a free trial for a given subscription and prevents them from getting another one. You do not need to track trial eligibility yourself.

When you create an offer and select **New customer acquisition** as the eligibility type, Google handles the rest. This is the simplest approach for free trials and introductory pricing.

Developer Determined Eligibility

For more complex scenarios, you can control offer eligibility on your backend. This is called developer determined eligibility.

With developer determined offers, you assign one or more **eligibility tags** to the offer in the Play Console. When your app queries product details, every offer comes back in the response, but your backend decides which tagged offers a specific user qualifies for. Your app then passes only the appropriate offer token when launching the purchase flow.

This model is useful for:

- **Loyalty discounts**: Users who have been active for more than 6 months get a special renewal price.
- **Cohort based pricing**: Users acquired through a specific campaign get different pricing.
- **Cross product offers**: Users who subscribe to Product A get a discount on Product B.
- **Win back with custom logic**: You determine which lapsed users deserve a comeback offer based on their usage history.

Here is how to check offer tags when filtering eligible offers for a user:

```
val offerDetails = productDetails
    .subscriptionOfferDetails ?: return

val eligibleOffers = offerDetails.filter { offer ->
    val tags = offer.offerTags
    // Check against your backend's eligibility
    // response for this user
    tags.isEmpty() ||
        tags.any { it in userEligibleTags }
}
```

Your backend should expose an endpoint that returns the set of eligible offer tags for the current user. Your app calls this endpoint, then filters the offers returned by `ProductDetails` to show only the ones the user

qualifies for.

Combining Eligibility Models

A single Subscription can have offers with different eligibility models. For example, you might have a Google managed free trial for new users and a developer determined loyalty discount for long term subscribers. Both can coexist on the same Base Plan.

Offer Tokens and the Purchase Flow

Every `SubscriptionOfferDetails` entry in the `ProductDetails` response includes an **offer token**. This token is a string that encodes which Base Plan and Offer (if any) the user is purchasing. You must pass this token when launching the billing flow. If you do not pass an offer token, the purchase will fail.

This is one of the most common mistakes when integrating subscriptions with PBL 8.x. Unlike one time products, you cannot simply pass a `ProductDetails` object to the billing flow. You must select an offer token and include it in the `BillingFlowParams`.

Here is the flow:

1. Query `ProductDetails` for your subscription.
2. Read the `subscriptionOfferDetails` list.
3. Choose the appropriate offer (based on eligibility, user preference, or your business logic).
4. Extract the `offerToken` from the chosen `SubscriptionOfferDetails`.
5. Build `BillingFlowParams` with the offer token and launch the flow.

```
val offerDetails = productDetails
    .subscriptionOfferDetails?.firstOrNull()
    ?: return

val productDetailsParams =
    BillingFlowParams.ProductDetailsParams
        .newBuilder()
        .setProductDetails(productDetails)
        .setOfferToken(offerDetails.offerToken)
        .build()

val billingFlowParams =
    BillingFlowParams.newBuilder()
        .setProductDetailsParamsList(
            listOf(productDetailsParams)
        )
        .build()

billingClient.launchBillingFlow(
    activity, billingFlowParams
)
```

Every Base Plan without an offer also has its own `SubscriptionOfferDetails` entry with an offer token. So even if you are not using offers, you still need to extract and pass an offer token. The token for a Base Plan without an offer simply points to the full price plan.

When a user has multiple eligible offers, your app decides which one to present. You might show the best available offer by default, or present a chooser if you want the user to pick. The offer token you pass determines the pricing the user sees in the Google Play purchase dialog.

The 3 Day Acknowledgement Rule

Every purchase, whether a one time product or a subscription, must be acknowledged within 3 days (72 hours) of the purchase. If you do not acknowledge a purchase within this window, Google automatically refunds it and revokes the entitlement.

This rule exists to protect users. Acknowledgement is your app's way of telling Google, "I have received this purchase and granted the user their entitlement." Without it, a user could pay for something and never receive it, with no recourse.

For subscriptions, you acknowledge the initial purchase. You do not need to acknowledge each renewal. Google handles renewals automatically, and they do not require separate acknowledgement.

You can acknowledge a purchase either on the client or the server:

Client side acknowledgement:

```
val acknowledgePurchaseParams =
    AcknowledgePurchaseParams.newBuilder()
        .setPurchaseToken(purchase.purchaseToken)
        .build()

val result = billingClient.acknowledgePurchase(
    acknowledgePurchaseParams
)

if (result.responseCode ==
    BillingResponseCode.OK
) {
    // Purchase acknowledged successfully
}
```

Server side acknowledgement (recommended):

Acknowledging on your backend is more reliable. Your server calls the Google Play Developer API's `purchases.subscriptions.acknowledge` endpoint (or `purchases.products.acknowledge` for one time products) with the purchase token. This way, acknowledgement only happens after your backend has verified the purchase and stored the entitlement. If your app crashes after the purchase but before acknowledgement, your backend can still catch it by processing the RTDN.

Shorter Windows for Short Prepaid Plans

The 3 day acknowledgement window assumes the product grants access for at least 3 days. But what if you sell a 1 day prepaid plan? The user's access would expire before the acknowledgement window closes.

For prepaid plans shorter than 3 days, the acknowledgement window is the same as the plan duration. A 1 day prepaid plan must be acknowledged within 1 day. A 2 day prepaid plan must be acknowledged within 2 days. This ensures Google can refund the user before their access period ends if your app fails to acknowledge the purchase.

The practical takeaway: always acknowledge purchases as soon as possible, ideally within seconds or minutes of the purchase completing. Do not rely on having the full 3 days. Network issues, app crashes, and edge cases can all eat into that window.

Putting It Together: Querying and Displaying Subscription Options

Let us walk through a complete example of querying a subscription's plans and offers, then presenting them to the user. This pulls together the hierarchy, offer tokens, and eligibility concepts from this chapter.

```
suspend fun loadSubscriptionOptions(
    billingClient: BillingClient
): List<SubscriptionOfferDetails> {
    val params = QueryProductDetailsParams
        .newBuilder()
        .setProductList(
            listOf(
                QueryProductDetailsParams.Product
                    .newBuilder()
                    .setProductId("premium")
                    .setProductType(
                        ProductType.SUBS
                    )
                    .build()
            )
        )
        .build()

    val result = billingClient
        .queryProductDetails(params)

    val details = result.productDetailsList
        ?.firstOrNull() ?: return emptyList()

    return details.subscriptionOfferDetails
        ?: emptyList()
}
```

From the returned list, each `SubscriptionOfferDetails` entry gives you:

- `basePlanId` : Which Base Plan this entry belongs to.
- `offerId` : The specific offer, or null if this is the base price.
- `offerToken` : The token you pass to launch the purchase.
- `pricingPhases` : The list of pricing phases (free period, intro price, then full price).
- `offerTags` : Tags for developer determined eligibility filtering.
- `installmentPlanDetails` : Present if this is an installment plan.

Your UI logic iterates through these entries, filters by eligibility, groups by Base Plan, and presents the best option (or multiple options) to the user. When the user taps "Subscribe," you extract the offer token from their chosen entry and launch the billing flow as shown earlier.

Chapter 5: Integrating the Play Billing Library

The Play Billing Library (PBL) is your app's interface to Google Play's billing system. Every purchase, product query, and feature check flows through the `BillingClient` class. This chapter covers how to set up `BillingClient` correctly, query products, and prepare your app for purchases.

Initializing BillingClient

The `BillingClient` is the single entry point for all billing operations. You create one using `BillingClient.Builder`:

```
val billingClient = BillingClient.newBuilder(context)
    .setListener(purchasesUpdatedListener)
    .enablePendingPurchases(
        PendingPurchasesParams.newBuilder().build()
    )
    .enableAutoServiceReconnection()
    .build()
```

Four things happen in this setup:

1. **`newBuilder(context)`**: You pass a `Context` to the builder. This should be your `Application` context or an activity context. If you pass an activity context, the `BillingClient` holds a reference to it, so you need to be careful about leaking activities. The safest choice is `applicationContext`, which avoids lifecycle issues entirely. If you pass a `null` context or a context from a destroyed activity, the builder call will succeed but `startConnection()` will fail later with obscure errors.

2. **`setListener`**: You provide a `PurchasesUpdatedListener` that receives all purchase results. This is a required call. If you skip it, `build()` throws an `IllegalArgumentException`. You can only set one listener per `BillingClient` instance. If you need multiple components to react to purchases, have your single listener dispatch events through a shared event bus, a `Flow`, or a callback list.

3. **`enablePendingPurchases`**: This tells PBL that your app can handle purchases that are not immediately completed (for example, when a user pays with cash at a convenience store). In PBL 8, this method requires a `PendingPurchasesParams` argument. This is mandatory. If you forget to call it, `build()` throws an `IllegalStateException` at runtime. The `PendingPurchasesParams.newBuilder().build()` call uses default settings, which enables pending purchases for both one time and subscription products.

4. **`enableAutoServiceReconnection`**: New in PBL 8, this tells the library to automatically reconnect to Google Play Services if the connection drops. Without this, you would need to manually detect disconnections and call `startConnection()` again. When enabled, PBL uses an exponential backoff strategy internally to retry the connection, so you do not need to implement your own retry logic for the connection itself.

Common Initialization Mistakes

A few patterns cause problems in production:

- **Creating multiple `BillingClient` instances**: Each instance opens its own connection to Google Play Services. If you create a new one every time an activity starts, you waste resources and may hit connection limits. Stick to one instance per process.
- **Forgetting to call `endConnection()`**: When your billing scope is destroyed (for example, when the user logs out or when an activity scoped client is no longer needed), call `billingClient.endConnection()`. Failing to do this leaks the service connection.
- **Using the wrong context**: Passing a `Fragment` context or a context from an inner class can lead to memory leaks. Always prefer `applicationContext` when the `BillingClient` outlives an activity.
- **Calling billing methods before connection is ready**: If you call `queryProductDetailsAsync()` before `onBillingSetupFinished` fires with `OK`, the call returns `SERVICE_DISCONNECTED`. Queue your operations and execute them only after the connection succeeds.

Where to Create BillingClient

Create your `BillingClient` in a scope that survives configuration changes. Good options include:

- A `ViewModel` scoped to your billing screen or activity
- A singleton `BillingRepository` in your dependency injection graph
- An `Application` scoped object if billing is central to your app

Avoid creating `BillingClient` in a `Fragment` or `Activity` directly, as configuration changes would destroy and recreate it, wasting connections.

The PurchasesUpdatedListener

The listener receives results for every purchase initiated by `launchBillingFlow()`:

```
val purchasesUpdatedListener =
    PurchasesUpdatedListener { billingResult, purchases ->
        when (billingResult.responseCode) {
            BillingResponseCode.OK -> {
                purchases?.forEach { purchase ->
                    handlePurchase(purchase)
                }
            }
            BillingResponseCode.USER_CANCELED -> {
                // User backed out of the purchase flow
            }
            else -> {
                // Handle other error codes
                logError(billingResult)
            }
        }
    }
```

This listener fires for purchases made through your app's UI. It does not fire for purchases that happen outside your app, such as subscription renewals or purchases restored from another device. For those, you need `queryPurchasesAsync()`, covered later in this chapter.

Thread Safety and Lifecycle

The `PurchasesUpdatedListener` callback fires on the main thread. This means you should not perform long running operations directly inside it. If you need to verify a purchase with your backend server, launch a coroutine or hand the work off to a background thread. Blocking the main thread in this callback freezes the UI and can cause ANRs.

The listener is tied to the `BillingClient` instance. If you call `endConnection()`, the listener will no longer receive callbacks. If the `BillingClient` reconnects (through auto reconnection), the same listener is reused. You do not need to register it again after a reconnection.

One important consideration: the listener can fire even when your app is in the background. If the user completes a purchase flow and then quickly switches away from your app, the callback may arrive while your activity is stopped. Make sure your listener does not directly update UI elements. Instead, update your data layer (a repository or a state holder) and let your UI observe the changes when it returns to the foreground.

The `purchases` parameter in the callback can be `null` when the `billingResult` response code is not `OK`. Always check for null before iterating. A common mistake is to force unwrap this list and crash when the user cancels the purchase flow.

Connecting to Google Play

After building `BillingClient`, you must connect to Google Play Services before calling any other method:

```
billingClient.startConnection(
    object : BillingClientStateListener {
        override fun onBillingSetupFinished(
            billingResult: BillingResult
        ) {
            if (billingResult.responseCode ==
                BillingResponseCode.OK
            ) {
                // Ready to query products and make purchases
                onBillingReady()
            }
        }

        override fun onBillingServiceDisconnected() {
            // Connection lost. With auto reconnection
            // enabled, PBL will retry automatically.
        }
    }
)
```

With `enableAutoServiceReconnection()` enabled, you do not need to manually call `startConnection()` again in `onBillingServiceDisconnected()`. The library handles reconnection for you. However, any operations that were in progress at the time of disconnection will fail and need to be retried by your code.

Connection States and What Happens When It Drops

The `BillingClient` connection can be in one of three states at any given time: disconnected, connecting, or connected. You can check the current state using `billingClient.isReady`, which returns `true` only when the connection is fully established and ready to handle requests.

When the connection drops, the `onBillingServiceDisconnected()` callback fires. This can happen for several reasons: Google Play Services being updated in the background, the system killing the Play Store process to reclaim memory, or a network interruption. With auto reconnection enabled, PBL retries the connection automatically using exponential backoff. The `onBillingSetupFinished` callback fires again once the connection is restored.

Any API call you make while the connection is down returns `SERVICE_DISCONNECTED` as the response code. Your code should handle this by queuing the failed operation and retrying it once `onBillingSetupFinished` fires again with `OK`. A practical pattern is to maintain a list of pending operations and flush them each time the connection comes back up.

If you are not using auto reconnection (for example, because you need fine grained control), you need to implement your own retry logic in `onBillingServiceDisconnected()`. Use exponential backoff with a cap to

avoid overwhelming the system. A starting delay of 1 second, doubling up to a maximum of 15 seconds, works well for most apps.

If you are using the KTX extensions, the coroutine version is cleaner:

```
// KTX coroutine extension (suspending)
val billingResult = billingClient.startConnection()
if (billingResult.responseCode ==
    BillingResponseCode.OK
) {
    onBillingReady()
}
```

Querying Available Products

Once connected, you can query product details. This fetches current pricing, descriptions, and offer information from Google Play:

```
val productList = listOf(
    QueryProductDetailsParams.Product.newBuilder()
        .setProductId("premium_monthly")
        .setProductType(ProductType.SUBS)
        .build(),
    QueryProductDetailsParams.Product.newBuilder()
        .setProductId("remove_ads")
        .setProductType(ProductType.INAPP)
        .build()
)

val params = QueryProductDetailsParams.newBuilder()
    .setProductList(productList)
    .build()

billingClient.queryProductDetailsAsync(params) {
    billingResult, productDetailsList ->
    if (billingResult.responseCode ==
        BillingResponseCode.OK
    ) {
        displayProducts(productDetailsList)
    }
}
```


57

You must specify both the product ID and the product type (`INAPP` for one time products, `SUBS` for subscriptions). Querying the wrong type for a product ID returns no results for that product.

Batching and Performance

You can include multiple products in a single query, and you should. Each `queryProductDetailsAsync` call is a round trip to Google Play's servers. If you have ten products, querying them all in one call is significantly faster than making ten individual calls. Google Play handles batches of up to 20 products per query efficiently. If you have more than 20 products, split them into batches of 20 and run the queries in parallel.

You can mix `INAPP` and `SUBS` products in the same product list. PBL handles them in one round trip regardless of type.

Handling Empty Results

When the response code is `OK` but the `productDetailsList` is empty, it means none of your product IDs matched anything in Google Play. This usually happens for one of these reasons: the product IDs in your code do not match what you configured in the Google Play Console, the products are in draft status and have not been activated yet, or you are testing on a device that is not signed into a Google account that has access to your app's test track.

If the list is partially empty (you queried five products but only got three back), the missing products may have mismatched IDs or types. Check that each product ID and type pair matches your Play Console configuration exactly. Product IDs are case sensitive.

Understanding QueryProductDetailsResult

In PBL 8, the result of `queryProductDetailsAsync()` can include `UnfetchedProduct` entries. An `UnfetchedProduct` indicates that a product ID was recognized but its details could not be fetched. Each `UnfetchedProduct` carries its own `BillingResult` with a response code that tells you why the fetch failed.

The response codes you may see on an `UnfetchedProduct` include:

- **`SERVICE_UNAVAILABLE`**: Google Play's servers could not be reached. This is a transient error, often caused by network issues. Retry the query after a short delay.
- **`SERVICE_TIMEOUT`**: The request took too long to complete. This typically happens on slow connections. Retry with backoff.
- **`ERROR`**: A general server side error occurred. This is also typically transient. Retry, but if it persists across multiple attempts, check the Play Console for product configuration issues.
- **`DEVELOPER_ERROR`**: The product ID or type is invalid. This is not transient. Check your product configuration in the Google Play Console.

A practical approach is to collect the unfetched product IDs, filter out any with `DEVELOPER_ERROR` (which will never succeed), and retry the remaining ones. Limit your retries to two or three attempts with exponential backoff. If products remain unfetched after that, show the user the products you did successfully fetch and log the failures for later investigation.

Why You Should Never Cache ProductDetails

`ProductDetails` objects contain pricing information that can change at any time. If you cache them, you risk showing outdated prices to users, which leads to a poor experience and potential policy violations. Always query fresh `ProductDetails` before displaying products to the user. The only caching you should do is within a single session or screen lifecycle, and only if you query again when the user navigates back.

Checking Feature Support

Not all devices and Google Play versions support every billing feature. Before using advanced features, check support:

```
val result = billingClient.isFeatureSupported(
    BillingClient.FeatureType.SUBSCRIPTIONS
)
if (result.responseCode == BillingResponseCode.OK) {
    // Subscriptions are supported
}
```

Available feature types include:

- `SUBSCRIPTIONS`: Basic subscription support. Almost universally available on modern devices, but older devices running very old Play Store versions may not support it.
- `SUBSCRIPTIONS_UPDATE`: Subscription upgrades and downgrades. Required before calling `launchBillingFlow()` with a subscription replacement. Without this, users on older Play Store versions cannot switch between subscription tiers.
- `PRICE_CHANGE_CONFIRMATION`: In app price change confirmation flow. Lets you show a dialog when a subscription price changes so the user can accept or decline. If unsupported, you need to handle price changes through other channels like email.
- `IN_APP_MESSAGING`: In app messaging for payment issues such as declined credit cards. When supported, Google Play can show messages directly in your app prompting the user to fix payment problems. This reduces involuntary churn from payment failures.
- `PRODUCT_DETAILS`: The `ProductDetails` API introduced in PBL 5. On very old Play Store versions, only the deprecated `SkuDetails` API is available. In PBL 8, this feature should be available on virtually all active devices.
- `ALTERNATIVE_BILLING`: Support for alternative billing. This is relevant in markets where regulations require apps to offer alternative payment methods.
- `ALTERNATIVE_BILLING_ONLY`: The app uses only alternative billing with no Google Play billing. This is a distinct feature from `ALTERNATIVE_BILLING`, which allows both.
- `EXTERNAL_OFFER`: Support for linking to external offers. Relevant for apps that qualify under specific regional regulations to direct users to web based purchasing.

If a feature is not supported, do not call its related APIs. The call may return `FEATURE_NOT_SUPPORTED` or behave unpredictably. Instead, fall back to alternative behavior or hide the related UI element. A good practice is to check feature support once after the connection is established and store the results in memory for the duration of the session.

Querying the User's Billing Country

Starting with PBL 7, you can query the user's billing country. This is useful for showing region specific offers or adjusting your UI:

```
val params = GetBillingConfigParams.newBuilder().build()
billingClient.getBillingConfigAsync(params) {
    billingResult, billingConfig ->
    if (billingResult.responseCode ==
        BillingResponseCode.OK
    ) {
        val countryCode = billingConfig?.countryCode
        // Use countryCode for region specific logic
    }
}
```

The country code is an ISO 3166-1 alpha-2 code (for example, "US", "KR", "DE"). This reflects the user's Google Play billing country, which may differ from their device locale or physical location. A user living in Germany with a device set to English still returns "DE" if their Google Play account is registered there.

Use Cases for Country Specific Logic

The billing country is useful in several scenarios:

- **Regional offer targeting**: You can show country specific promotions or highlight plans that are popular in a given market. For example, you might feature annual plans in markets where users tend to prefer longer commitments.
- **Compliance filtering**: Some features or pricing models are only available in specific regions. Use the billing country to determine which products or offers to display.
- **Analytics and segmentation**: Knowing the billing country lets you segment conversion rates, revenue, and churn by market without relying on less reliable signals like device locale.
- **Currency display**: While `ProductDetails` already includes localized pricing, you may want to adjust surrounding UI text or layout based on the country.

Caching the Billing Country

The billing country rarely changes for a given user. You can safely cache it for the duration of a session. If you need it across sessions, storing it in local preferences is reasonable, but you should refresh it each time the `BillingClient` connects in case the user changed their Play account. Do not use the cached value as a

source of truth for server side decisions. Your backend should rely on the country information from Google Play's server side APIs instead.

Personalized Pricing

The EU Consumer Rights Directive requires that you disclose when prices are personalized based on automated decision making. If you use personalized pricing, you must set `setIsOfferPersonalized(true)` on your `BillingFlowParams`. This adds a disclosure message to the purchase dialog:

```
val flowParams = BillingFlowParams.newBuilder()
    .setProductDetailsParamsList(listOf(productParams))
    .setIsOfferPersonalized(true)
    .build()
```

Only set this to `true` if you are actually personalizing the price. If all users see the same price, leave it at the default (`false`).

When Personalized Pricing Applies

Personalized pricing means that the price a specific user sees was determined by automated profiling or decision making about that individual. This includes scenarios where you adjust prices based on a user's purchase history, engagement level, geographic data beyond the standard Play Store localization, or any machine learning model that outputs different prices for different users.

Standard regional pricing set in the Google Play Console does not count as personalized pricing, because all users in the same country see the same price. Similarly, promotional offers that are available to all users in a defined segment (such as "all new users") are generally not considered personalized.

EU Law Context

The requirement comes from Article 6(1)(ea) of the EU Consumer Rights Directive (2011/83/EU, as amended by Directive 2019/2161). It mandates that online marketplaces inform consumers when prices are personalized based on automated decision making. Google Play enforces this by requiring the `setIsOfferPersonalized(true)` flag. When you set it, the purchase dialog includes a disclosure that the price was personalized for the user.

If you are selling to users in the EU and using any form of price personalization, you must set this flag. Failing to do so can expose you to regulatory risk. The safest approach is to determine at runtime whether the user's billing country is in the EU (using `getBillingConfigAsync()`) and set the flag accordingly when your pricing logic involves personalization.

Attaching User Identifiers

To help Google detect fraud and to link purchases to your own user accounts, you can attach obfuscated identifiers to purchases:

```
val flowParams = BillingFlowParams.newBuilder()
    .setProductDetailsParamsList(listOf(productParams))
    .setObfuscatedAccountId(hashOf(userId))
    .setObfuscatedProfileId(hashOf(profileId))
    .build()
```

These identifiers are obfuscated, meaning you should hash your actual user IDs before passing them. Google uses these to detect when multiple purchases are associated with the same account, which helps identify fraudulent behavior. The identifiers are also returned in the `Purchase` object and in server side API responses, making it easier to reconcile purchases with your user database.

KTX Coroutine Extensions

The `billing-ktx` artifact provides suspending versions of async methods. Instead of callbacks, you get clean coroutine code:

```
// Callback style
billingClient.queryProductDetailsAsync(params) {
    result, details -> /* handle */
}

// KTX coroutine style
val result = billingClient.queryProductDetails(params)
val details = result.productDetailsList
```

The KTX extensions are available for:

- `queryProductDetails()` (suspending version of `queryProductDetailsAsync`)
- `queryPurchasesAsync()` has a coroutine counterpart
- `consumePurchase()` (suspending version of `consumeAsync`)
- `acknowledgePurchase()` (suspending version of `acknowledgePurchase`)

The real benefit of coroutines becomes clear when you chain multiple billing operations together. With callbacks, querying products and then launching a purchase flow requires nested listeners that are hard to follow. With coroutines, the same logic reads top to bottom:

```
// Callback style: nested and hard to follow
billingClient.queryProductDetailsAsync(params) {
    result, details ->
    if (result.responseCode == BillingResponseCode.OK) {
        val product = details.firstOrNull()
        if (product != null) {
            val flowParams = buildFlowParams(product)
            billingClient.launchBillingFlow(
                activity, flowParams
            )
        }
    }
}
```

```
// KTX coroutine style: sequential and readable
val result = billingClient.queryProductDetails(params)
if (result.billingResult.responseCode ==
    BillingResponseCode.OK
) {
    val product = result.productDetailsList
        ?.firstOrNull()
    if (product != null) {
        val flowParams = buildFlowParams(product)
        billingClient.launchBillingFlow(
            activity, flowParams
        )
    }
}
```

Error handling is also simpler. With callbacks, you handle errors inside each callback. With coroutines, you can use standard `try/catch` blocks or `Result` wrappers. This makes it easier to implement retry logic or to propagate errors up to your UI layer.

Using coroutines makes your billing code more readable and easier to compose with other async operations in your app. If you are already using coroutines elsewhere in your project, adopting the KTX extensions is straightforward and reduces the amount of callback management code you need to maintain.

Connection Lifecycle Management

Deciding when to create and destroy your `BillingClient` depends on how central billing is to your app. There are two common patterns.

Pattern 1: Application Scoped Client

If your app uses billing throughout many screens (for example, checking subscription status to unlock features), keep a single `BillingClient` alive for the entire application lifecycle. Create it in your `Application.onCreate()` or in a singleton provided by your dependency injection framework. Call `startConnection()` once at startup. Call `endConnection()` only when the application is being destroyed or when the user logs out and you want a clean slate.

This pattern is simple and avoids repeated connection setup. The downside is that the connection to Google Play Services stays open even when the user is not interacting with billing features. In practice, this overhead is minimal because idle connections consume very little resources.

Pattern 2: Scoped Client

If billing is only relevant on a few screens (for example, a single purchase screen), you can scope the `BillingClient` to a `ViewModel` or a navigation graph scope. Create the client when the scope starts, connect, and call `endConnection()` when the scope is destroyed. This keeps things tidy but means you pay the connection setup cost each time the user enters the billing flow.

Which Pattern to Choose

For most apps, the application scoped pattern is the better choice. Connection setup takes a few hundred milliseconds on a typical device, and if you need to check subscription status frequently (to gate features, show badges, or customize the UI), having an always connected client avoids repeated delays. Use the scoped pattern only if billing is a rarely accessed corner of your app and you want to minimize the surface area of your Google Play Services interaction.

Regardless of which pattern you use, always call `endConnection()` when you are done with the client. Failing to do so leaks the service connection and can cause issues if your app creates a new `BillingClient` later without ending the previous one.

Querying Existing Purchases

You need to check for existing purchases in several situations:

- When `BillingClient` first connects (to catch purchases made on other devices)
- When your app returns to the foreground (to catch purchases made outside your app)
- When restoring purchases after an app reinstall

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

val result = billingClient.queryPurchasesAsync(params)
if (result.billingResult.responseCode ==
    BillingResponseCode.OK
) {
    result.purchasesList.forEach { purchase ->
        handlePurchase(purchase)
    }
}
```

You must query separately for `ProductType.INAPP` and `ProductType.SUBS` to get all purchases. For subscriptions, `queryPurchasesAsync()` returns purchases in states that may still grant access: `PURCHASED` and `PENDING`. It does not return expired or fully canceled subscriptions.

Call this method on every app launch and every return to foreground. This is your safety net for catching purchases that your `PurchasesUpdatedListener` might have missed.

Chapter 6: The Purchase Flow

The purchase flow is the core interaction between your app, the user, and Google Play. When a user taps "Buy" in your app, a carefully orchestrated sequence begins: your app builds the purchase parameters, Google Play shows the payment dialog, the user confirms, Google processes the payment, and your app receives the result. Getting this flow right is essential for a reliable billing experience.

Building BillingFlowParams for One Time Products

For a one time product, you build `BillingFlowParams` from the `ProductDetails` you queried earlier:

```
val productDetailsParams =
    BillingFlowParams.ProductDetailsParams.newBuilder()
        .setProductDetails(productDetails)
        .build()

val billingFlowParams = BillingFlowParams.newBuilder()
    .setProductDetailsParamsList(
        listOf(productDetailsParams)
    )
    .build()
```

This is the simplest case. You pass the `ProductDetails` object directly, and Google Play handles pricing, currency, and the purchase dialog.

If the one time product has multiple purchase options (introduced in PBL 8), you can specify which option to present by setting the offer token from the selected purchase option.

Additional BillingFlowParams Options

Beyond the basic product selection, `BillingFlowParams.Builder` gives you several optional fields that are worth understanding.

obfuscatedAccountId: You can attach a hashed or obfuscated version of your own user account identifier. This helps Google Play detect suspicious activity across devices. If the same account ID attempts rapid purchases from different IP addresses or devices, Google can flag the behavior and reduce fraud. You should never pass a raw email address or plain text identifier here. Use a one way hash of your internal user ID instead.

66

```
val billingFlowParams = BillingFlowParams.newBuilder()
    .setProductDetailsParamsList(
        listOf(productDetailsParams)
    )
    .setObfuscatedAccountId(
        hashUserId(currentUser.id)
    )
    .build()
```

obfuscatedProfileId: Similar to the account ID, but intended for apps that support multiple profiles under a single account. A family streaming app, for example, might have one Google account with separate profiles for each family member. Passing the profile ID lets you attribute the purchase to the correct profile on your backend.

isOfferPersonalized: Under EU consumer protection regulations, you must disclose when a price shown to the user is personalized based on automated decision making. If you use dynamic pricing or user specific offers, set `setIsOfferPersonalized(true)`. Google Play will display a disclosure notice in the purchase dialog informing the user that the price was personalized. If you do not personalize prices, you can leave this at the default value of `false`.

Building BillingFlowParams for Subscriptions

Subscriptions require an additional piece: the offer token. Because a subscription can have multiple base plans and offers, you must specify which one the user selected:

```
val offerToken = productDetails
    .subscriptionOfferDetails
    ?.firstOrNull()
    ?.offerToken ?: return

val productDetailsParams =
    BillingFlowParams.ProductDetailsParams.newBuilder()
        .setProductDetails(productDetails)
        .setOfferToken(offerToken)
        .build()

val billingFlowParams = BillingFlowParams.newBuilder()
    .setProductDetailsParamsList(
        listOf(productDetailsParams)
    )
    .build()
```


67

The offer token encodes the specific base plan and offer combination. If you present multiple options to the user (monthly vs. annual, with or without a free trial), each option has its own offer token. Pass the token that corresponds to what the user selected.

Offer Token Selection Logic

In practice, selecting the right offer token requires more thought than calling `firstOrNull()`. A single subscription product can have multiple base plans, and each base plan can have multiple offers attached to it. The `subscriptionOfferDetails` list returned by `ProductDetails` contains every valid combination.

You should think about offer selection in two layers. First, determine which base plan the user wants. If your UI shows a toggle between "Monthly" and "Annual," you need to filter `subscriptionOfferDetails` by the `basePlanId` that matches the user's choice. Second, within that base plan, determine which offer to apply. Google Play returns offers in priority order, with the most favorable offer (typically a free trial or introductory price) listed first, but only if the user is eligible for it. Google automatically filters out offers the user has already redeemed, so you do not need to track trial eligibility yourself.

A practical pattern is to group the offer details by base plan and then present the best available offer for each:

```
val offersByPlan = productDetails
    .subscriptionOfferDetails
    ?.groupBy { it.basePlanId }
    ?: emptyMap()

// For each plan, the first entry is the
// best eligible offer
val monthlyOffer = offersByPlan["monthly"]
    ?.firstOrNull()
val annualOffer = offersByPlan["annual"]
    ?.firstOrNull()
```

When you display these options in your UI, show the user what they are getting. If the first pricing phase is a free trial, say "7 day free trial, then $9.99/month." If it is an introductory price, say "$4.99/month for 3 months, then $9.99/month." You can extract these details from the `pricingPhases` list on each `SubscriptionOfferDetails` object. Each pricing phase includes the price, billing period, and recurrence mode, giving you everything you need to build an informative purchase screen.

If you want to let users choose between an offer with a free trial and one without (perhaps the no trial option costs less per month), present both tokens and let the user decide. The key rule is that whatever the user taps in your UI must map to the exact offer token you pass to `BillingFlowParams`.

If you do not set an offer token for a subscription, the purchase flow will fail with a `DEVELOPER_ERROR`.

Launching the Billing Flow

With your `BillingFlowParams` built, launch the purchase flow:

```
val billingResult = billingClient.launchBillingFlow(
    activity,
    billingFlowParams
)

if (billingResult.responseCode !=
    BillingResponseCode.OK
) {
    // The flow failed to launch
    handleLaunchError(billingResult)
}
```

You must pass a reference to the current `Activity`. The billing flow presents a system dialog on top of your activity, and Google Play needs the activity context to do this.

A successful `launchBillingFlow()` call (returning `BillingResponseCode.OK`) means the purchase dialog was shown to the user. It does not mean the purchase succeeded. The actual purchase result arrives asynchronously through your `PurchasesUpdatedListener`.

What Happens Inside Google Play

When `launchBillingFlow()` returns `OK`, Google Play takes over and presents a system managed bottom sheet dialog on top of your activity. This dialog is not part of your app. You cannot customize its appearance or intercept its events. It runs in a separate process controlled by the Play Store.

The dialog shows the product name, price, and description you configured in the Play Console. The user sees their default payment method and can tap to change it. Google Play supports credit cards, debit cards, Google Play balance, carrier billing, PayPal, and various regional payment methods depending on the user's country.

If the user selects a credit or debit card, the payment may go through additional verification. For transactions that require Strong Customer Authentication (common in the European Economic Area under PSD2 regulations), Google Play handles the 3D Secure (3DS) challenge flow automatically. The user sees a bank verification screen within the Google Play dialog. You do not need to implement anything for this. Google manages the entire challenge, and your app only receives the final result through `PurchasesUpdatedListener`.

During this time, your activity is still running in the background. You should not start any timers or assume the purchase will complete within a certain window. The user might spend a minute picking a payment method, or they might need to complete a 3DS challenge that takes thirty seconds. Your app simply waits for the callback.

If the user presses the back button or dismisses the dialog without completing the purchase, your `PurchasesUpdatedListener` receives `USER_CANCELED`. If the user completes the purchase, you receive `OK` along with the `Purchase` object.

Common Launch Errors

- `ITEM_ALREADY_OWNED`: The user already owns this non consumable product or has an unacknowledged purchase for it. Before showing a purchase button, call `queryPurchasesAsync()` to check existing ownership. If you receive this error unexpectedly, it often means a previous purchase was never acknowledged or consumed, so the item is stuck in the user's inventory.
- `DEVELOPER_ERROR`: Something is wrong with your parameters. Common causes include: the product ID does not match a product in the Play Console, you forgot the offer token for a subscription, the product has not been activated in the Console, or you are testing with a build whose package name does not match the Console listing. This error also fires if you pass an offer token from a different product than the one in `ProductDetails`.
- `FEATURE_NOT_SUPPORTED`: The device or Play Store version does not support the requested product type. This can happen on very old Play Store versions, sideloaded devices without Google Play Services, or certain Android TV and Wear OS devices that lack support for subscriptions.
- `BILLING_UNAVAILABLE`: Google Play billing is not available on this device. The user may not be signed into a Google account, or Play Services may be missing entirely. This is common on custom ROMs, Amazon Fire devices running a sideloaded APK, or emulators without Google Play configured.
- `ITEM_UNAVAILABLE`: The product exists in the Play Console but is not available for purchase in the user's country, or it has been deactivated. Double check your product's availability settings and ensure it is published in the regions where your users are located.
- `NETWORK_ERROR`: The device could not reach Google Play servers. This can happen on flaky connections. You can retry the launch after a short delay, but avoid aggressive retry loops since the user probably knows their connection is poor.

Handling onPurchasesUpdated Results

After the user interacts with the purchase dialog, your `PurchasesUpdatedListener` fires:

```
val purchasesUpdatedListener =
    PurchasesUpdatedListener { billingResult, purchases ->
        when (billingResult.responseCode) {
            BillingResponseCode.OK -> {
                purchases?.forEach { handlePurchase(it) }
            }
            BillingResponseCode.USER_CANCELED -> {
                // No action needed
            }
            BillingResponseCode.ITEM_ALREADY_OWNED -> {
                // Refresh purchases from cache
                refreshPurchases()
            }
            else -> handleError(billingResult)
        }
    }
```

When the response code is `OK`, you receive a list of `Purchase` objects. In most cases, this list contains a single purchase. For multi quantity purchases, you still receive a single `Purchase` object with the quantity set accordingly.

Processing Successful Purchases: The 5 Step Checklist

Every successful purchase should go through these five steps, in order:

Step 1: Verify on Your Backend

Send the purchase token to your backend server. Your backend calls the Google Play Developer API to verify the purchase is legitimate:

- For one time products: `purchases.products.get`
- For subscriptions: `purchases.subscriptionsv2.get`

Never trust the client alone. A modified app could fake a purchase result.

When your backend receives the verification response from Google, you should validate several fields before granting access. Check that the `purchaseState` is `0` (purchased) and not `1` (canceled) or `2` (pending). Confirm that the `productId` matches the product you expected the user to buy. Verify that the `orderId` is not one you have already processed, which protects against replay attacks where someone resubmits the same purchase token. For subscriptions, check the `expiryTimeMillis` to confirm the subscription is still active. You should also verify that the `packageName` matches your app's package name, since a purchase token from a different app would be invalid in your context.

Store the purchase token, order ID, and verification timestamp in your database. You will need these for customer support, refund processing, and audit trails.

Step 2: Check Purchase State

The `Purchase` object has a `purchaseState` property:

```
when (purchase.purchaseState) {
    Purchase.PurchaseState.PURCHASED -> {
        // Payment complete, proceed to grant
    }
    Purchase.PurchaseState.PENDING -> {
        // Payment not yet complete
        // Do NOT grant access yet
        notifyPendingPurchase()
    }
    else -> {
        // UNSPECIFIED_STATE, handle as error
    }
}
```

Only grant access when the state is `PURCHASED`. Pending purchases (covered later) mean the user has initiated payment but has not completed it yet.

The `PurchaseState` enum has three values. `PURCHASED` (value 1) means payment is complete and you can safely grant access. `PENDING` (value 2) means the user selected a delayed payment method and the transaction is not finalized. `UNSPECIFIED_STATE` (value 0) means something unexpected happened, and you should treat this as an error. Log the full purchase object when you encounter `UNSPECIFIED_STATE` so you can investigate. It is rare, but it can occur during edge cases like interrupted transactions or Play Store bugs.

On the server side, the `purchases.products.get` API returns a numeric `purchaseState` field with the same three values. Your server should independently check this field rather than relying on the state reported by the client.

Step 3: Grant Entitlement

After verification, grant the user access to the content or feature they purchased. Update your backend database and notify your app to update its UI.

Your entitlement grant logic must be idempotent. This means that processing the same purchase token twice should produce the same result without granting duplicate access. There are several scenarios where your server might receive the same purchase more than once: the client retries a failed network call, your Real Time Developer Notification (RTDN) handler fires while the client is also sending the token, or the user opens the app on a second device that triggers `queryPurchasesAsync()` and sends the same token again.

To implement idempotency, use the purchase token as a unique key in your database. Before granting access, check if you have already processed that token. If you have, return a success response to the client without modifying the user's entitlements. For consumable products where the user receives currency or items, this check is especially important. Granting 100 coins twice because of a retry would be a costly bug.

72

Step 4: Acknowledge or Consume

For non consumable products and subscriptions, acknowledge the purchase:

```
val params = AcknowledgePurchaseParams.newBuilder()
    .setPurchaseToken(purchase.purchaseToken)
    .build()

billingClient.acknowledgePurchase(params) { result ->
    if (result.responseCode == BillingResponseCode.OK) {
        // Acknowledged successfully
    }
}
```

For consumable products, consume instead of acknowledging:

```
val params = ConsumeParams.newBuilder()
    .setPurchaseToken(purchase.purchaseToken)
    .build()

billingClient.consumeAsync(params) { result, token ->
    if (result.responseCode == BillingResponseCode.OK) {
        // Consumed, user can purchase again
    }
}
```

You can also acknowledge or consume on your backend using the server API. Each approach has tradeoffs worth considering.

Server side acknowledgement is more reliable because you can tie it directly to your entitlement grant logic. Your server verifies the purchase, writes the entitlement to the database, and acknowledges the purchase in a single transaction. If any step fails, you can retry the entire sequence. This guarantees that you never acknowledge a purchase without also granting access. The downside is that if your server is temporarily unreachable, the purchase sits unacknowledged until the client can reach your backend.

Client side acknowledgement is simpler to implement and works even if your server has an outage. The risk is that you might acknowledge a purchase before your server has verified and recorded it. If the client acknowledges but the server call fails, you have a purchase that Google considers fulfilled but your backend knows nothing about. The user might not receive their entitlement.

The best approach is to acknowledge on the server as your primary path, with client side acknowledgement as a fallback. If the server call fails after multiple retries, have the client acknowledge the purchase and queue a background sync to reconcile with the server later.

73

The 3 day rule: You must acknowledge or consume every purchase within 3 days. If you fail to do so, Google automatically refunds the purchase. This protects users from paying for purchases that your app never fulfilled. Three days sounds generous, but it can sneak up on you during a server outage or a holiday weekend. Monitor your unacknowledged purchase count and alert on it.

Step 5: Notify the User

Update your UI to reflect the purchase. Show a success message, unlock features, update subscription status, or add the purchased items to the user's inventory.

The post purchase moment is one of the highest engagement points in your app. The user just made a financial commitment, and how you respond shapes their perception. A few UX best practices:

Show confirmation immediately. Do not wait for the server round trip to display a success state. You already know from the client side `Purchase` object that payment succeeded, so show the user a confirmation dialog or animation while your server verification happens in the background. If the server later rejects the purchase (rare but possible), you can revoke access at that point.

Be specific about what the user received. Instead of a generic "Purchase successful!" message, say "You now have Premium access" or "500 coins added to your balance." This reinforces the value of the purchase and reduces support tickets from confused users.

For subscriptions, show what happens next. Tell the user when their next billing date is and where they can manage their subscription. Linking to the Google Play subscription management page (`https://play.google.com/store/account/subscriptions`) is a good practice that reduces churn from users who feel trapped.

Avoid navigating the user away from their current context unnecessarily. If they were browsing content and bought a subscription to unlock it, take them right back to that content, now unlocked. Do not redirect them to a settings page or a generic home screen.

Handling Pending Transactions

Pending transactions occur when a user chooses a payment method that does not process immediately. There are several scenarios where this happens.

The most common case is cash based payments at convenience stores, which are popular in Japan, Brazil, Mexico, and other markets. The user receives a payment code, walks to a physical store, and pays in cash. The purchase stays in `PENDING` state until the store reports the payment to the payment processor, which can take anywhere from a few hours to several days.

Carrier billing is another source of pending transactions. When a user charges a purchase to their mobile phone bill, the carrier may not confirm the charge instantly. This is more common with smaller regional carriers that batch process transactions.

Slow card authorizations can also produce pending states. While most credit card transactions resolve in seconds, some banks in certain regions take longer to authorize. This is uncommon but not impossible, especially for users with prepaid cards or cards from smaller financial institutions.

74

You must enable pending transactions by calling `enablePendingPurchases(PendingPurchasesParams)` on your `BillingClient.Builder`. If you do not enable this, users with delayed payment methods cannot purchase from your app at all, and you lose those potential customers.

When you receive a purchase with `PurchaseState.PENDING`:

1. **Do not grant access**. The payment has not been processed yet.
2. **Inform the user** that their purchase is pending and they will receive access once payment is confirmed.
3. **Check again later**. The purchase will transition to `PURCHASED` (payment successful) or `CANCELED` (payment failed or expired). Your `PurchasesUpdatedListener` fires when the state changes, and you can also detect the change by calling `queryPurchasesAsync()`.

```
fun handlePurchase(purchase: Purchase) {
    if (purchase.purchaseState ==
        Purchase.PurchaseState.PENDING
    ) {
        // Show "purchase pending" UI
        // Do NOT grant entitlement
        showPendingMessage(purchase)
        return
    }

    if (purchase.purchaseState ==
        Purchase.PurchaseState.PURCHASED
    ) {
        if (!purchase.isAcknowledged) {
            verifyAndGrantAccess(purchase)
        }
    }
}
```

Your server will also receive an RTDN when the pending purchase completes, so you have both client and server side signals.

When to Call queryPurchasesAsync()

You should call `queryPurchasesAsync()` in three situations:

On connect: Right after `startConnection()` succeeds. This is your first opportunity to sync state. It catches purchases made on other devices (the user bought a subscription on their tablet and now opens the app on their phone), purchases restored after a reinstall, and subscription state changes that happened while your app was not running. Without this call, a user who reinstalls your app would see no evidence of their active subscription until they manually restore purchases.

On app launch: Every time your app starts, query for current purchases. This ensures your entitlement state matches the source of truth. Even if you cache entitlements locally, the cache can go stale. A subscription might have expired overnight, or the user might have received a refund through customer support. Querying on launch keeps your local state honest.

On foreground resume: When your activity's `onResume()` fires, query again. This covers a surprisingly common scenario: the user backgrounded your app, opened the Play Store, canceled their subscription or changed their payment method, and then returned to your app. Without this query, your app would still show the stale subscription state. This also catches pending purchases that completed while your app was in the background. If the user completed a cash payment at a convenience store and then opened your app, the foreground resume query picks up the state transition from `PENDING` to `PURCHASED`.

```
// In your ViewModel or repository
fun refreshPurchases() {
    val subsParams = QueryPurchasesParams.newBuilder()
        .setProductType(ProductType.SUBS)
        .build()
    val inappParams = QueryPurchasesParams.newBuilder()
        .setProductType(ProductType.INAPP)
        .build()

    billingClient.queryPurchasesAsync(subsParams) {
        result, subs -> processPurchases(subs)
    }
    billingClient.queryPurchasesAsync(inappParams) {
        result, inapp -> processPurchases(inapp)
    }
}
```

Cart Abandonment Reminders

Google Play can automatically send reminders to users who start but do not complete a purchase. This is the cart abandonment feature and it is enabled by default.

When a user opens the purchase dialog but cancels or does not complete the transaction, Google may send a push notification reminding them about the product. The timing and frequency of these reminders is controlled entirely by Google. You do not configure when or how often they appear. If the user taps the notification, the Play Store opens the purchase flow directly, and if the user completes the purchase, your `PurchasesUpdatedListener` fires as usual.

According to Google's published data, cart abandonment reminders can recover a meaningful percentage of otherwise lost conversions. Industry benchmarks for mobile subscription apps suggest that between 10% and 15% of users who abandon a purchase flow will convert if reminded within 24 hours. Your actual recovery rate

76

depends on your product, price point, and audience, but the feature is essentially free revenue with no engineering cost.

You can opt out of cart abandonment reminders in the Play Console under your app's monetization settings (Monetization setup > Cart abandonment). You can also control the behavior per product. However, there is rarely a good reason to disable this feature. The only scenario where you might consider opting out is if your pricing changes frequently and you do not want users reminded about a price that is no longer valid. Even then, Google typically handles this gracefully by showing the current price when the user returns.

Note that cart abandonment reminders only work for users who reached the Google Play purchase dialog. If the user tapped a button in your app but you never called `launchBillingFlow()` (perhaps due to a validation error), Google has no record of the abandoned purchase and cannot send a reminder. This means your own pre purchase validation logic can inadvertently reduce the pool of users eligible for reminders. Keep pre launch checks minimal and fast.

Multi Line Item Purchases

Starting with Play Billing Library 7.0 and expanded in PBL 8.1, Google Play supports purchasing multiple items in a single transaction. You can bundle multiple products into one `BillingFlowParams` by adding more than one entry to the `ProductDetailsParamsList`. The user sees a single purchase dialog with all items listed, confirms once, and your app receives one `Purchase` object that covers everything.

This is particularly useful for subscription bundles. Imagine your app offers a "Music + Video" plan where the user gets both a music streaming subscription and a video streaming subscription in one purchase. Instead of forcing two separate checkout flows, you bundle them:

```
val musicParams =
    BillingFlowParams.ProductDetailsParams
        .newBuilder()
        .setProductDetails(musicProductDetails)
        .setOfferToken(musicOfferToken)
        .build()

val videoParams =
    BillingFlowParams.ProductDetailsParams
        .newBuilder()
        .setProductDetails(videoProductDetails)
        .setOfferToken(videoOfferToken)
        .build()

val billingFlowParams = BillingFlowParams
    .newBuilder()
    .setProductDetailsParamsList(
        listOf(musicParams, videoParams)
    )
    .build()
```

When the purchase completes, the resulting `Purchase` object contains multiple product IDs in its `products` list. Your processing logic needs to handle this. Instead of assuming a single product per purchase, iterate over `purchase.products` and grant entitlements for each one.

There are a few constraints to keep in mind. All items in a multi line purchase must be of the same product type. You cannot mix a subscription with a one time product in a single transaction. The items must also be compatible in terms of billing periods if they are subscriptions. Google Play enforces these rules at the API level, so you will get a `DEVELOPER_ERROR` if you violate them.

Multi line item purchases also affect how you handle acknowledgement. You acknowledge the purchase once using the purchase token, and it covers all items in the bundle. You do not need to acknowledge each item separately.

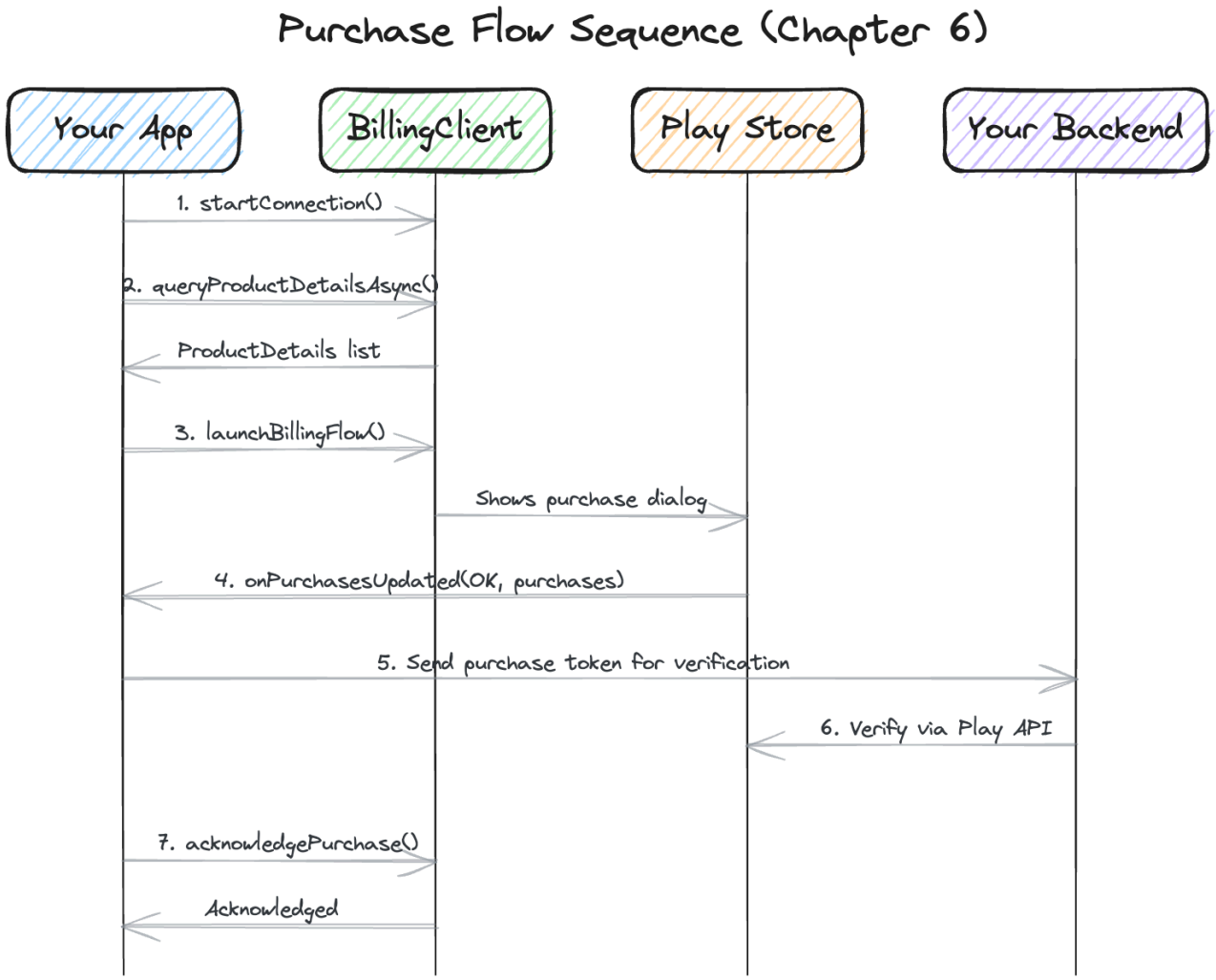
Putting It All Together

Here is the complete purchase flow from start to finish:

1. User taps a product in your UI.
2. Your app queries `ProductDetails` (if not already queried this session).
3. Your app builds `BillingFlowParams` with the product and offer token (for subscriptions).
4. Your app calls `launchBillingFlow()` with the current activity.
5. Google Play shows the purchase dialog.

6. The user confirms or cancels.
7. Your `PurchasesUpdatedListener` fires with the result.
8. For `PURCHASED` state: send the token to your backend, verify, grant access, acknowledge/consume, update UI.
9. For `PENDING` state: inform the user, do not grant access, check again later.
10. For errors: handle according to Chapter 8's error handling strategies.

This flow applies to both one time products and subscriptions. The only difference is in how you build `BillingFlowParams` (subscriptions need an offer token) and what you do after verification (acknowledge for subscriptions, consume for consumable products).



79

Chapter 7: Subscription Upgrades, Downgrades, and Plan Changes

Users change their minds. A subscriber on your basic plan wants premium features. A premium subscriber wants to save money by switching to a lower tier. A user on a monthly cycle wants to switch to annual billing. These are plan changes, and handling them correctly is one of the more nuanced parts of Google Play Billing.

This chapter covers how plan changes work under the hood, the six replacement modes Google provides, and the important details around purchase tokens, proration, and deferred switching. By the end, you will know exactly which replacement mode to use for every scenario your app might encounter.

Why Plan Changes Create a New Purchase Token

When a user changes their subscription plan, Google does not simply modify the existing subscription in place. Instead, it creates an entirely new purchase token. This is a fundamental design decision that affects how you build your backend.

Think of it this way: a purchase token represents a specific subscription agreement between the user and Google. When the terms of that agreement change (different product, different price, different billing period), Google treats it as a new agreement. The old purchase token becomes inactive, and a new one takes its place.

This means your backend must handle the transition. When you receive a new purchase token from a plan change, you need to:

1. Verify the new purchase token with the Google Play Developer API.
2. Look up the old subscription associated with this user.
3. Revoke the old entitlement and grant the new one (or update the entitlement level).
4. Store the new purchase token as the active subscription record.

If you only tracked entitlements by purchase token and ignored the connection between old and new tokens, you might accidentally grant a user two active subscriptions, or worse, lose track of their subscription entirely. Google provides the `linkedPurchaseToken` field specifically to help you manage this transition, which you will learn about later in this chapter.

The 6 Replacement Modes

When you launch a plan change flow, you specify a **replacement mode** that tells Google how to handle the financial transition. Each mode controls two things: when the user gets access to the new plan, and how Google handles the money.

PBL 8.x defines these modes in the `ReplacementMode` interface. Here they are, each with a concrete scenario to show when you would use it.

WITH_TIME_PRORATION

This is the default replacement mode. Google calculates the monetary value of the remaining time on the current plan, converts it into time on the new plan, and switches the user immediately.

Scenario: A user is halfway through a $4.99/month basic plan and upgrades to a $9.99/month premium plan. They have roughly $2.50 in unused value. Google converts that into approximately 7 to 8 days of premium access. The user gets premium immediately, and their next billing date adjusts forward based on the converted time. No additional charge happens right now.

When to use it: This is a good default for any upgrade or downgrade where you want the transition to happen immediately without charging the user. It is the fairest option from the user's perspective because they never lose money they already paid.

Trade off: For upgrades to a more expensive plan, the next billing cycle comes sooner than the user might expect. For example, if the conversion only yields 8 days of premium time, the user gets charged again in 8 days rather than a full month later. This can surprise users if you do not communicate it clearly.

```
val replacementParams = BillingFlowParams
    .SubscriptionProductReplacementParams
    .newBuilder()
    .setOldPurchaseToken(currentPurchaseToken)
    .setReplacementMode(
        ReplacementMode.WITH_TIME_PRORATION
    )
    .build()
```

CHARGE_PRORATED_PRICE

Google charges the user immediately for the price difference between the old and new plan for the remainder of the current billing period. The user gets the new plan immediately, and the billing cycle stays the same.

Scenario: A user is halfway through a $4.99/month basic plan and upgrades to a $9.99/month premium plan. They have about 15 days left. Google charges them $2.50 right now (the $5.00 monthly difference prorated to 15 days), switches them to premium immediately, and the next full $9.99 charge happens on their original renewal date.

When to use it: This works well for upgrades where you want to keep the billing date predictable. The user's renewal date does not shift, which is easier to explain and less surprising. Users see a small charge now and then the full price at their usual renewal.

Important limitation: This mode only works for upgrades where the new plan costs more than the old plan. If you try to use it for a downgrade, the billing flow will fail. Google does not issue refunds through proration charges.

```
val replacementParams = BillingFlowParams
    .SubscriptionProductReplacementParams
    .newBuilder()
    .setOldPurchaseToken(currentPurchaseToken)
    .setReplacementMode(
        ReplacementMode.CHARGE_PRORATED_PRICE
    )
    .build()
```

CHARGE_FULL_PRICE

Google charges the full price of the new plan immediately and switches the user right away. Any remaining value from the old plan is either converted into time on the new plan or applied as a credit toward the new plan's billing period.

Scenario: A user on a $4.99/month basic plan upgrades to a $49.99/year premium plan. Google charges $49.99 immediately, switches the user to premium, and the remaining value from the basic plan extends the annual subscription start date slightly. The user now has roughly a year plus a few extra days of premium.

When to use it: This is the required mode for switching to or from prepaid plans, because prepaid plans do not have recurring billing cycles to adjust. It also works well when switching between plans with different billing periods (monthly to annual), where proration math would be confusing.

Important detail: Unlike the other proration modes, the user sees a full charge in their payment method immediately. Make sure your UI communicates this clearly so users are not surprised.

```
val replacementParams = BillingFlowParams
    .SubscriptionProductReplacementParams
    .newBuilder()
    .setOldPurchaseToken(currentPurchaseToken)
    .setReplacementMode(
        ReplacementMode.CHARGE_FULL_PRICE
    )
    .build()
```

WITHOUT_PRORATION

Google switches the user to the new plan immediately, but does not charge them until the next renewal date. The user gets the new plan's entitlements right away, and the price difference is settled at the next billing cycle.

Scenario: A user is on a $4.99/month basic plan that included a 7 day free trial, and they are currently in the trial period. They decide to upgrade to the $9.99/month premium plan. With `WITHOUT_PRORATION`, they keep the remainder of their free trial, get premium access immediately, and are charged $9.99 when the trial ends.

82

When to use it: This mode shines when you want to preserve free trials or introductory pricing periods. If you used `WITH_TIME_PRORATION` or `CHARGE_PRORATED_PRICE` during a free trial, the trial would effectively end. `WITHOUT_PRORATION` lets you upgrade the user while keeping whatever promotional period they are in.

Trade off: For upgrades outside of trial periods, the user gets the more expensive plan for free until their next renewal date. This means you are giving away premium access for the remainder of the current billing period. That may be acceptable as a goodwill gesture, but understand the revenue impact.

```
val replacementParams = BillingFlowParams
    .SubscriptionProductReplacementParams
    .newBuilder()
    .setOldPurchaseToken(currentPurchaseToken)
    .setReplacementMode(
        ReplacementMode.WITHOUT_PRORATION
    )
    .build()
```

DEFERRED

The user stays on their current plan until the next renewal date. At renewal, Google switches them to the new plan and charges the new price. No immediate change happens from the user's perspective.

Scenario: A premium subscriber at $9.99/month decides to downgrade to the $4.99/month basic plan. With `DEFERRED`, they keep premium access for the remainder of their current billing period (they already paid for it), and the switch to basic happens automatically at renewal.

When to use it: This is the natural choice for downgrades. The user already paid for their current period, so they should keep their current access level until it expires. It is also useful when you want to let users schedule a plan change in advance without any immediate disruption.

Important behavior: Even though the switch does not happen until renewal, Google returns a new purchase token immediately when the billing flow completes. This new token has two line items: one for the current plan (active now) and one for the new plan (pending). You will learn more about this behavior in the section on deferred replacement later in this chapter.

```
val replacementParams = BillingFlowParams
    .SubscriptionProductReplacementParams
    .newBuilder()
    .setOldPurchaseToken(currentPurchaseToken)
    .setReplacementMode(
        ReplacementMode.DEFERRED
    )
    .build()
```


83

KEEP_EXISTING (PBL 8.1+)

This mode was introduced in PBL 8.1 and works differently from the others. Instead of replacing the current plan, it keeps the existing subscription active and adds the new subscription alongside it. The user ends up with both plans active simultaneously.

Scenario: Your app offers a base subscription ($4.99/month) and an add on subscription ($2.99/month for extra storage). A user who already has the base subscription wants to add the storage add on. With `KEEP_EXISTING`, the original base subscription stays active and untouched, and the storage add on becomes a separate active subscription linked to the same purchase.

When to use it: This is designed for subscription add ons and modular subscription architectures. If your app offers a base plan plus optional add on packages, `KEEP_EXISTING` lets users build up their subscription bundle without replacing what they already have.

Important detail: The `linkedPurchaseToken` on the new purchase still points to the original purchase token. Your backend needs to understand that both subscriptions are active simultaneously, rather than treating the new one as a replacement for the old one. This requires different entitlement logic than a standard upgrade or downgrade.

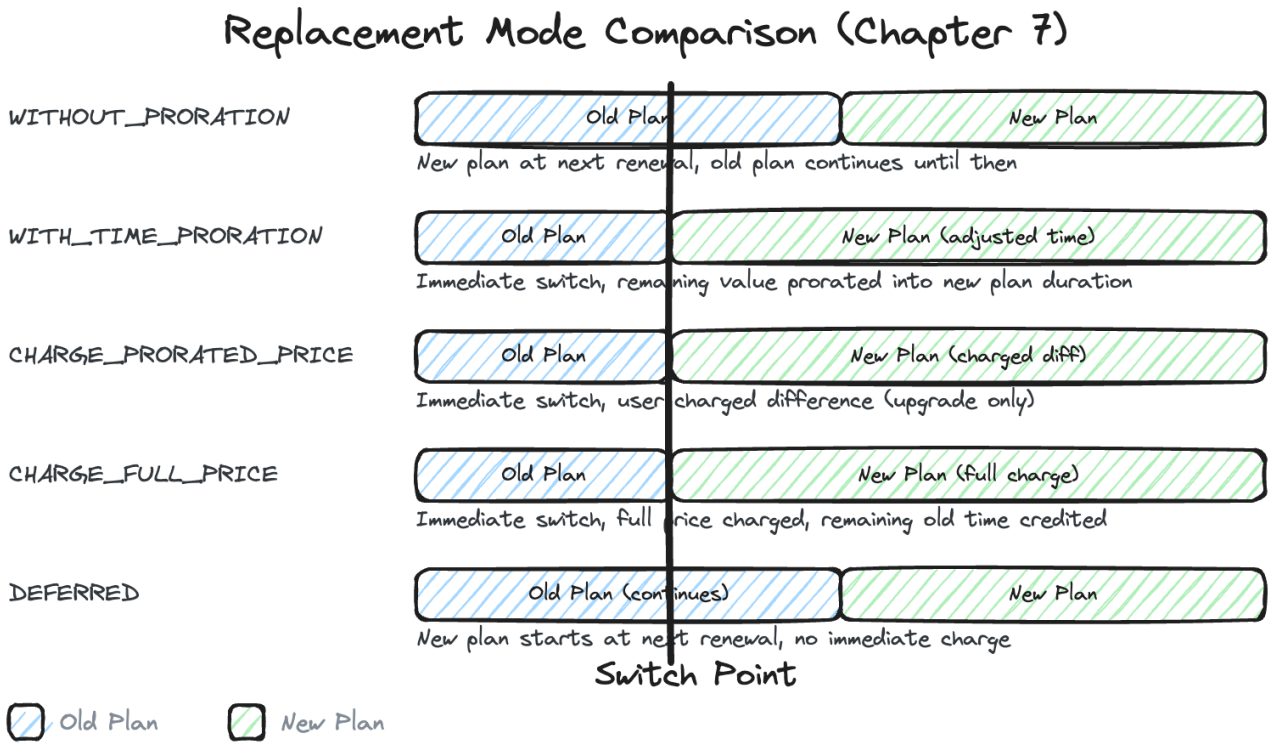
```
val replacementParams = BillingFlowParams
    .SubscriptionProductReplacementParams
    .newBuilder()
    .setOldPurchaseToken(currentPurchaseToken)
    .setReplacementMode(
        ReplacementMode.KEEP_EXISTING
    )
    .build()
```

Choosing the Right Replacement Mode

Here is a quick reference to help you decide:

SCENARIO	RECOMMENDED MODE
Standard upgrade, fair to user	`WITH_TIME_PRORATION`
Upgrade, keep billing date	`CHARGE_PRORATED_PRICE`
Switch to/from prepaid	`CHARGE_FULL_PRICE`
Upgrade during free trial	`WITHOUT_PRORATION`
Downgrade	`DEFERRED`
Subscription add on	`KEEP_EXISTING`

These are recommendations, not rules. Your business logic might call for different choices. For example, you might prefer `DEFERRED` for all downgrades but `CHARGE_PRORATED_PRICE` for upgrades to keep billing dates stable. The key is understanding what each mode does so you can make an informed decision.



SubscriptionProductReplacementParams vs. SubscriptionUpdateParams

If you have worked with older versions of PBL, you likely used `BillingFlowParams.SubscriptionUpdateParams` to configure plan changes. Starting with PBL 8.1, Google introduced `SubscriptionProductReplacementParams` as the preferred way to set up replacement flows.

The older `SubscriptionUpdateParams` is now deprecated. While it still works in PBL 8.x, you should migrate to the new API for any new code. Here is why the new API is better:

1. **Clearer separation of concerns**: `SubscriptionProductReplacementParams` is attached to the product parameters rather than the billing flow parameters. This makes it clearer which product is being replaced and how.
2. **Support for newer features**: `KEEP_EXISTING` mode and other future replacement features are only guaranteed to work with the new API.
3. **Consistent with PBL 8.x patterns**: The new API follows the same builder patterns used elsewhere in PBL 8.x.

Here is how the old approach looks:

```
// Deprecated approach using SubscriptionUpdateParams
val updateParams = BillingFlowParams
    .SubscriptionUpdateParams.newBuilder()
    .setOldPurchaseToken(oldToken)
    .setSubscriptionReplacementMode(
        ReplacementMode.WITH_TIME_PRORATION
    )
    .build()

val flowParams = BillingFlowParams.newBuilder()
    .setSubscriptionUpdateParams(updateParams)
    .setProductDetailsParams(
        listOf(productDetailsParams)
    )
    .build()
```

And here is the new approach with `SubscriptionProductReplacementParams`:

```
// Preferred approach (PBL 8.1+)
val replacementParams = BillingFlowParams
    .SubscriptionProductReplacementParams
    .newBuilder()
    .setOldPurchaseToken(oldToken)
    .setReplacementMode(
        ReplacementMode.WITH_TIME_PRORATION
    )
    .build()

val productParams = BillingFlowParams
    .ProductDetailsParams.newBuilder()
    .setProductDetails(newProductDetails)
    .setOfferToken(selectedOfferToken)
    .setSubscriptionReplacementParams(
        replacementParams
    )
    .build()

val flowParams = BillingFlowParams.newBuilder()
    .setProductDetailsParams(
        listOf(productParams)
    )
    .build()
```

The key structural difference is that replacement parameters are now attached to `ProductDetailsParams` instead of being a top level property on `BillingFlowParams`. This makes the data model more logical: you are saying "I want this product, and it should replace this existing subscription" rather than "I want to update a subscription, and here is a product."

If you are starting a new project with PBL 8.1 or later, use `SubscriptionProductReplacementParams` exclusively. If you are maintaining existing code that uses `SubscriptionUpdateParams`, plan to migrate when convenient, but there is no urgent rush since the deprecated API still functions correctly.

Understanding linkedPurchaseToken and Why It Matters

When a user performs a plan change, the new `SubscriptionPurchaseV2` resource returned by the Google Play Developer API contains a field called `linkedPurchaseToken`. This field holds the purchase token of the subscription that was replaced.

This field is your backend's guide to maintaining a clean subscription history. Here is why it matters:

Preventing duplicate entitlements: Without checking `linkedPurchaseToken`, your backend might see a new purchase token and treat it as a brand new subscription. The user would then appear to have two active subscriptions when they should have one. By checking `linkedPurchaseToken`, you know to deactivate the old subscription record when activating the new one.

Building a subscription chain: Over the life of a user's subscription, they might upgrade, downgrade, and change plans multiple times. Each change produces a new purchase token linked to the previous one. By following the chain of `linkedPurchaseToken` values, you can reconstruct the complete history of a user's subscription.

Handling the token on your server: When your backend receives a new purchase token (either from the client app or through an RTDN), the verification flow should include these steps:

1. Call the Google Play Developer API to get the `SubscriptionPurchaseV2` resource for the new token.
2. Check if `linkedPurchaseToken` is present.
3. If it is present, look up the old purchase token in your database.
4. Mark the old subscription record as replaced.
5. Create a new subscription record with the new token.
6. If using `KEEP_EXISTING` mode, mark both subscriptions as active instead of replacing.

```
// Server-side logic for handling linkedPurchaseToken
fun handleNewPurchaseToken(newToken: String) {
    val subscription = playApi
        .getSubscriptionV2(packageName, newToken)

    val linkedToken = subscription
        .linkedPurchaseToken

    if (linkedToken != null) {
        // This is a plan change, not a new purchase
        val oldRecord = database
            .findByPurchaseToken(linkedToken)
        if (oldRecord != null) {
            oldRecord.status = "REPLACED"
            database.update(oldRecord)
        }
    }

    database.insertSubscription(
        purchaseToken = newToken,
        productId = subscription.productId,
        status = "ACTIVE"
    )
}
```

A common mistake: Some developers only check `linkedPurchaseToken` one level deep. But consider this sequence: the user buys Plan A (token 1), upgrades to Plan B (token 2, linked to token 1), then upgrades to Plan C (token 3, linked to token 2). If your backend only processes notifications for token 3 and token 1 was still marked active (maybe you missed the notification for token 2), you need to walk the chain back and deactivate everything older than the current active token.

The safest approach is to look up `linkedPurchaseToken`, mark it as replaced, and also verify there are no other active subscriptions for the same user and product family.

Deferred Replacement Behavior: Immediate Token, Two Line Items

The `DEFERRED` replacement mode has unique behavior that deserves special attention. When you launch a billing flow with `DEFERRED` mode, Google returns a new purchase token immediately, even though the actual plan switch does not happen until the next renewal date.

This means the `PurchasesUpdatedListener` in your app receives the new purchase right away. Your backend gets a new token to verify right away. But the user is still on their old plan.

How does this work? The new purchase token's `SubscriptionPurchaseV2` resource contains **two line items**:

1. **The current plan**: This line item represents the user's existing subscription. It is active now and has an expiry time matching the current billing period's end.
2. **The new plan**: This line item represents the plan the user is switching to. It is in a "pending" or "deferred" state and will become active when the current plan's line item expires.

```
// Server-side: inspecting a deferred replacement
fun handleDeferredReplacement(token: String) {
    val subscription = playApi
        .getSubscriptionV2(packageName, token)

    for (item in subscription.lineItems) {
        if (item.deferredItemReplacement != null) {
            // This is the NEW plan, pending
            val newProductId = item.productId
            val newBasePlanId = item
                .offerDetails?.basePlanId
            // Store this as the upcoming plan
        } else {
            // This is the CURRENT plan, still active
            val currentProductId = item.productId
            val expiryTime = item.expiryTime
            // User keeps this until expiryTime
        }
    }
}
```

Your backend should handle this by granting the current plan's entitlement and scheduling or noting the upcoming change. When the renewal date arrives, Google sends an RTDN indicating the switch has occurred. At that point, you update the user's entitlement to reflect the new plan.

Why this matters for your UI: If a user initiates a deferred downgrade, your app should show them that the change is scheduled. Something like "You will switch to Basic on March 15" gives the user confidence that their request was processed. You can read the expiry time of the current plan's line item to determine when the switch will happen.

Canceling a deferred replacement: If the user changes their mind before the renewal date, they can initiate another plan change. The new plan change replaces the pending deferred change. There is no separate "cancel deferred change" API.

Default Replacement Mode Configuration in Play Console

Starting with recent versions of the Play Console, you can set a default replacement mode for plan changes at the subscription level. This means you do not always have to specify the replacement mode in your client code, because Google can fall back to the configured default.

To configure this:

1. Go to **Monetize > Subscriptions** in the Play Console.
2. Select the subscription you want to configure.
3. Look for the **Upgrade/downgrade** settings section (this may be under base plan settings depending on the Console version).
4. Set the default replacement mode for upgrades and downgrades.

This is useful if you want consistent behavior across all plan changes without relying on every client version to specify the correct mode. However, specifying the replacement mode explicitly in your client code is still the recommended approach for two reasons:

1. **Clarity**: Anyone reading your code can see exactly what replacement behavior is expected.
2. **Control**: Different plan changes might warrant different modes. An upgrade might use `CHARGE_PRORATED_PRICE` while a downgrade uses `DEFERRED`. The Play Console default is a single setting per subscription, so it cannot express this nuance.

If you do specify a replacement mode in your client code, it overrides the Play Console default. The console setting only applies when no mode is specified in the `BillingFlowParams`.

Think of the Play Console default as a safety net. If a client version ships without specifying a replacement mode (perhaps due to a bug), the console default ensures users still get a reasonable experience rather than an error.

Putting It All Together: A Complete Plan Change Flow

Here is how a typical upgrade flow works end to end, from the user tapping "Upgrade" in your app to the entitlement being updated on your backend.

Step 1: The user taps an upgrade button in your app. Your app already has the current subscription's purchase token (stored locally after the original purchase) and has queried `ProductDetails` for the new plan.

Step 2: Your app builds `BillingFlowParams` with the new product details and the replacement parameters:

```
val replacementParams = BillingFlowParams
    .SubscriptionProductReplacementParams
    .newBuilder()
    .setOldPurchaseToken(currentToken)
    .setReplacementMode(
        ReplacementMode.CHARGE_PRORATED_PRICE
    )
    .build()

val productParams = BillingFlowParams
    .ProductDetailsParams.newBuilder()
    .setProductDetails(premiumDetails)
    .setOfferToken(premiumOfferToken)
    .setSubscriptionReplacementParams(
        replacementParams
    )
    .build()
```

Step 3: Launch the billing flow. Google Play shows the user a confirmation dialog that explains the price change and timing.

Step 4: The `PurchasesUpdatedListener` receives the result. If successful, you get a new `Purchase` object with a new purchase token.

Step 5: Send the new purchase token to your backend for verification.

Step 6: Your backend calls the Google Play Developer API, finds the `linkedPurchaseToken`, deactivates the old subscription record, creates the new one, and acknowledges the purchase.

Step 7: Your backend sends a response to the app confirming the entitlement update. The app updates its UI to reflect premium access.

This entire flow typically completes in a few seconds from the user's perspective.

Chapter 8: Error Handling and Retry Strategies

Every call you make to the Play Billing Library can fail. Network connections drop, Google Play Services restarts, users cancel purchases, and payment methods get declined. If your app does not handle these failures properly, users see broken purchase flows, lose access to content they paid for, or give up and uninstall.

This chapter gives you a complete error handling strategy for Play Billing Library 8.x. You will learn what every `BillingResponseCode` means, which errors are retriable and which are not, and how to implement retry logic that recovers gracefully without hammering Google's servers.

Understanding BillingResult and BillingResponseCode

Every PBL operation returns a `BillingResult` object. This is the primary way Google Play communicates whether an operation succeeded or why it failed. The `BillingResult` contains two pieces of information:

- **`responseCode`**: An integer constant from `BillingClient.BillingResponseCode` that tells you the category of the result.
- **`debugMessage`**: A human readable string with additional context. This is useful for logging, but never show it to users and never parse it programmatically. Google can change these messages at any time.

Here is how you check a billing result:

```
// PBL 8.x
val billingResult = billingClient.acknowledgePurchase(params)

when (billingResult.responseCode) {
    BillingClient.BillingResponseCode.OK -> {
        // Success. Proceed with your logic.
    }
    BillingClient.BillingResponseCode.NETWORK_ERROR -> {
        // Retriable. Schedule a retry.
    }
    else -> {
        Log.e("Billing", billingResult.debugMessage)
    }
}
```

The `OK` response code (value 0) means the operation completed successfully. Everything else indicates either a recoverable problem you can retry or a permanent failure you need to handle differently.

Here is the full set of response codes you will encounter in PBL 8.x:

RESPONSE CODE	VALUE	RETRIABLE?
OK	0	N/A
USER_CANCELED	1	No
SERVICE_UNAVAILABLE	2	Yes
BILLING_UNAVAILABLE	3	User initiated
ITEM_UNAVAILABLE	4	No
DEVELOPER_ERROR	5	No
ERROR	6	Yes
ITEM_ALREADY_OWNED	7	No (refresh cache)
ITEM_NOT_OWNED	8	No (refresh cache)
NETWORK_ERROR	12	Yes
SERVICE_DISCONNECTED	-1	Yes
SERVICE_TIMEOUT	-3	Yes
FEATURE_NOT_SUPPORTED	-2	No

You should handle every one of these in production code. Ignoring even one can lead to silent failures that are hard to debug.

Sub Response Codes in PBL 8.0+

Starting with PBL 8.0, Google added sub response codes that give you more specific information about why certain operations failed. These appear in the `onPurchasesUpdated` callback through the `BillingResult` and provide additional detail beyond the top level response code.

Two sub response codes are particularly useful:

PAYMENT_DECLINED_DUE_TO_INSUFFICIENT_FUNDS

This tells you the user's payment method was declined specifically because of insufficient funds. The top level response code for this is `ERROR`, but the sub response code lets you show a more helpful message. Instead of a generic "purchase failed" screen, you can suggest the user update their payment method or try a different one.

USER_INELIGIBLE

This indicates the user does not qualify for a specific offer. For example, if you have a free trial offer restricted to new subscribers and a returning subscriber tries to claim it, you get this sub response code. You can use this to show the user alternative offers they do qualify for.

93

You access sub response codes through the `getSubResponseCode()` method on `BillingResult`. Not every failure includes a sub response code, so always check whether one is present before acting on it. These sub response codes supplement your existing error handling logic rather than replacing it.

Retriable Errors and Their Strategies

Some billing errors are temporary. The operation failed now, but it might succeed if you try again. The key is knowing which errors to retry and how aggressively to retry them.

NETWORK_ERROR and SERVICE_TIMEOUT

These are the most common transient errors. `NETWORK_ERROR` means the device could not reach Google's servers. `SERVICE_TIMEOUT` means the request was sent but no response came back in time.

For both of these, a simple retry with a short delay works well. Start with a delay of one to two seconds and retry up to three times. If the user is on a flaky connection, a brief pause is often enough for the network to recover.

Do not retry immediately without any delay. Rapid fire retries on a struggling network connection just add noise and drain the user's battery.

SERVICE_DISCONNECTED

This means your `BillingClient` lost its connection to Google Play Services. In PBL 8.x, the library handles reconnection automatically in most cases. If auto reconnection does not resolve the issue, you fall back to calling `startConnection()` again manually.

```
// PBL 8.x
private suspend fun ensureConnected(): Boolean {
    if (billingClient.isReady) return true
    val result = billingClient.startConnection()
    return result.responseCode ==
        BillingClient.BillingResponseCode.OK
}
```

After reconnecting, retry the original operation. In practice, `SERVICE_DISCONNECTED` often happens when the device wakes from sleep or when Google Play Services updates itself in the background.

SERVICE_UNAVAILABLE and ERROR

`SERVICE_UNAVAILABLE` means Google Play Services is temporarily overloaded or unreachable. `ERROR` is a general catch all for unexpected failures on Google's side. Both of these call for exponential backoff rather than simple retry. You start with a longer initial delay and increase it with each attempt, giving Google's services time to recover.

94

Use a base delay of two seconds, double it on each retry, and cap at three attempts. You will see the full implementation later in this chapter.

BILLING_UNAVAILABLE

This means billing is not available on the device at all. Common causes include:

- Google Play Store is not installed (some sideloaded devices or emulators)
- Google Play Store version is too old
- The user's country does not support Google Play purchases
- The user is not signed into a Google account

You cannot retry this automatically because the underlying problem requires user action. Show a message explaining that Google Play is not available and let the user fix the issue, whether that means signing in, updating the Play Store, or switching accounts. You can offer a "try again" button so the user can retry after resolving the problem.

ITEM_ALREADY_OWNED and ITEM_NOT_OWNED

These are not traditional retry candidates, but they indicate your local purchase state is out of sync with Google's records.

`ITEM_ALREADY_OWNED` happens when you try to purchase a non consumable product or subscription the user already owns. `ITEM_NOT_OWNED` happens when you try to consume or acknowledge a purchase that Google does not think the user has.

The fix is to refresh your local cache:

```
// PBL 8.x
suspend fun refreshPurchases() {
    val params = QueryPurchasesParams.newBuilder()
        .setProductType(
            BillingClient.ProductType.SUBS
        )
        .build()
    val result = billingClient.queryPurchasesAsync(params)
    if (result.billingResult.responseCode ==
        BillingClient.BillingResponseCode.OK
    ) {
        // Update your local purchase state
        processPurchases(result.purchasesList)
    }
}
```

After refreshing, check whether the user actually owns the product. If they do, grant them access instead of trying to purchase again. If they do not, the purchase flow should work on the next attempt.

Non Retriable Errors

Some errors mean "stop trying." Retrying these wastes resources and creates a poor user experience.

FEATURE_NOT_SUPPORTED

This means the device or Play Store version does not support the feature you are trying to use. For example, calling a subscriptions API on a very old version of Google Play Services that does not support subscriptions.

The right approach is to check feature support before calling the method:

```
// PBL 8.x
val result = billingClient.isFeatureSupported(
    BillingClient.FeatureType.SUBSCRIPTIONS
)
if (result.responseCode ==
    BillingClient.BillingResponseCode.OK
) {
    // Safe to use subscription features
} else {
    // Hide subscription UI elements
}
```

Check feature support once during initialization and adapt your UI accordingly. Do not show subscription options to users whose devices cannot handle them.

USER_CANCELED

The user dismissed the purchase dialog without completing the purchase. This is the most common "error" you will see, and it is not really an error at all. The user simply changed their mind.

Handle this gracefully. Return the user to wherever they were before the purchase flow started. Do not show an error dialog, do not show a "purchase failed" message, and definitely do not prompt them to try again immediately. A quiet return to the previous screen is the right behavior.

ITEM_UNAVAILABLE

The product you tried to query or purchase does not exist or is not active in the Play Console. This should not happen in production if your product configuration is correct.

If you encounter this, refresh your product details from Google Play:

```
// PBL 8.x
suspend fun refreshProductDetails(
    productId: String
): ProductDetails? {
    val params = QueryProductDetailsParams.newBuilder()
        .setProductList(
            listOf(
                QueryProductDetailsParams.Product
                    .newBuilder()
                    .setProductId(productId)
                    .setProductType(
                        BillingClient.ProductType.INAPP
                    )
                    .build()
            )
        )
        .build()
    val result =
        billingClient.queryProductDetails(params)
    return result.productDetailsList?.firstOrNull()
}
```

If the product still comes back empty, it likely means the product was deactivated in the Play Console. Remove it from your UI.

DEVELOPER_ERROR

This is Google telling you that your API call is malformed. Common causes include:

- Passing an invalid product ID
- Using the wrong product type (e.g., querying a subscription as INAPP)
- Providing invalid parameters to `launchBillingFlow()`
- Calling methods before the `BillingClient` is connected

This error should never reach production. If it does, fix your code. Log the `debugMessage` because it usually contains specific information about what you did wrong.

Implementing Simple Retry

For transient errors like `NETWORK_ERROR` and `SERVICE_TIMEOUT`, a simple retry with a fixed delay handles most cases. Here is a reusable retry wrapper:

```
// PBL 8.x
suspend fun <T> retryBillingCall(
    maxAttempts: Int = 3,
    delayMs: Long = 1_000L,
    block: suspend () -> T
): T {
    var lastException: Exception? = null
    repeat(maxAttempts) { attempt ->
        try {
            return block()
        } catch (e: Exception) {
            lastException = e
            if (attempt < maxAttempts - 1) {
                delay(delayMs)
            }
        }
    }
    throw lastException
        ?: IllegalStateException("Retry failed")
}
```

You use it like this:

```
// PBL 8.x
val result = retryBillingCall(maxAttempts = 3) {
    billingClient.queryProductDetails(params)
}
```

This works well for operations that throw exceptions on transient failures. However, most PBL methods return a `BillingResult` instead of throwing. You need a version that understands billing response codes:

```
// PBL 8.x
suspend fun <T> retryOnBillingError(
    maxAttempts: Int = 3,
    delayMs: Long = 1_000L,
    retriableCodes: Set<Int> = setOf(
        BillingClient.BillingResponseCode.NETWORK_ERROR,
        BillingClient.BillingResponseCode.SERVICE_TIMEOUT,
        BillingClient.BillingResponseCode.ERROR,
        BillingClient.BillingResponseCode.SERVICE_UNAVAILABLE
    ),
    block: suspend () -> T
): T where T : BillingResult {
    var lastResult: T? = null
    repeat(maxAttempts) { attempt ->
        val result = block()
        if (result.responseCode !in retriableCodes) {
            return result
        }
        lastResult = result
        if (attempt < maxAttempts - 1) {
            delay(delayMs)
        }
    }
    return lastResult!!
}
```

This version checks the response code after each attempt. If the code is not in the retriable set, it returns immediately, whether it is `OK` or a non retriable error. Only retriable codes trigger another attempt.

Implementing Exponential Backoff

For errors like `SERVICE_UNAVAILABLE` and `ERROR`, exponential backoff is the better strategy. Instead of waiting the same amount of time between each retry, you double the delay each time. This gives overloaded services room to recover.

99

```
// PBL 8.x
suspend fun <T> retryWithBackoff(
    maxAttempts: Int = 3,
    baseDelayMs: Long = 2_000L,
    factor: Double = 2.0,
    retriableCodes: Set<Int> = setOf(
        BillingClient.BillingResponseCode.SERVICE_UNAVAILABLE,
        BillingClient.BillingResponseCode.ERROR,
        BillingClient.BillingResponseCode.NETWORK_ERROR,
        BillingClient.BillingResponseCode.SERVICE_TIMEOUT
    ),
    block: suspend () -> BillingResult
): BillingResult {
    var lastResult: BillingResult? = null
    repeat(maxAttempts) { attempt ->
        val result = block()
        if (result.responseCode !in retriableCodes) {
            return result
        }
        lastResult = result
        if (attempt < maxAttempts - 1) {
            val delayMs = baseDelayMs *
                factor.pow(attempt.toDouble())
            delay(delayMs.toLong())
        }
    }
    return lastResult!!
}
```

With a base delay of 2 seconds and a factor of 2, the delays look like this:

ATTEMPT	DELAY BEFORE RETRY
1st retry	2 seconds
2nd retry	4 seconds
3rd retry	8 seconds (if you extend to 4 attempts)

Three attempts with this pattern mean the entire sequence takes about 6 seconds in the worst case. That is a reasonable amount of time for the user to wait, and it gives Google's services meaningful breathing room between attempts.

You can add jitter (random variation) to the delay to prevent multiple devices from retrying at exactly the same time. This matters at scale when thousands of devices might hit the same transient error simultaneously:

100

```
// PBL 8.x
val jitter = Random.nextLong(0, baseDelayMs / 2)
val delayMs = baseDelayMs *
    factor.pow(attempt.toDouble()) + jitter
delay(delayMs.toLong())
```

Adding jitter spreads retry attempts across a wider time window, reducing the chance of another wave of failures hitting the service all at once.

Proper Error Handling in onPurchasesUpdated

The `onPurchasesUpdated` callback is where you receive purchase results after a user interacts with the Google Play purchase dialog. This is the single most important place to get error handling right, because it directly affects whether users get the content they paid for.

Here is a complete implementation that handles every response code:

101

```
// PBL 8.x
override fun onPurchasesUpdated(
    billingResult: BillingResult,
    purchases: List<Purchase>?
) {
    when (billingResult.responseCode) {
        BillingClient.BillingResponseCode.OK -> {
            purchases?.forEach { purchase ->
                handlePurchase(purchase)
            }
        }
        BillingClient.BillingResponseCode.USER_CANCELED -> {
            // User backed out. No action needed.
        }
        BillingClient.BillingResponseCode.ITEM_ALREADY_OWNED -> {
            // Refresh local cache and grant access
            scope.launch { refreshPurchases() }
        }
        else -> {
            Log.e("Billing",
                "Purchase failed: " +
                "${billingResult.responseCode} " +
                billingResult.debugMessage
            )
            showPurchaseError(billingResult)
        }
    }
}
```

A few things to note about this implementation:

1. **OK with null purchases**: Even when the response code is `OK`, the `purchases` list can theoretically be null. Always null check it.
2. **USER_CANCELED does nothing**: No error messages, no prompts, no analytics events marking it as a failure. The user made a deliberate choice.
3. **ITEM_ALREADY_OWNED refreshes state**: Instead of showing an error, you query for the user's current purchases and grant access. This handles the case where a purchase succeeded but your app crashed before processing it.
4. **Everything else gets logged**: The `debugMessage` goes into your logs for investigation. The user sees a generic, friendly error message.

For the `handlePurchase` function called on success, make sure you verify the purchase on your backend and acknowledge it within 3 days. An unacknowledged purchase gets refunded automatically by Google.

102

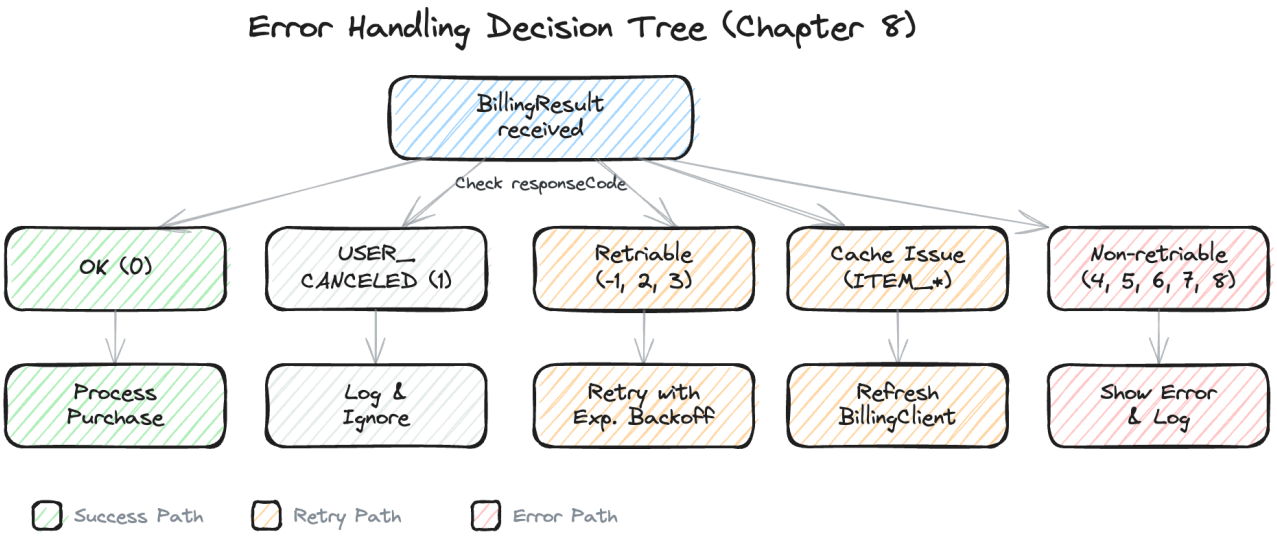
The Complete Decision Tree

Bringing everything together, here is the full decision tree for handling any `BillingResponseCode`. This function takes a billing result and the operation that produced it, then decides what to do:

```
// PBL 8.x
sealed class BillingAction {
    data object Success : BillingAction()
    data object RetrySimple : BillingAction()
    data object RetryBackoff : BillingAction()
    data object Reconnect : BillingAction()
    data object RefreshPurchases : BillingAction()
    data object RefreshProducts : BillingAction()
    data object UserRetry : BillingAction()
    data object Ignore : BillingAction()
    data class Fail(val msg: String) : BillingAction()
}
```


103

```
// PBL 8.x
fun decideBillingAction(code: Int): BillingAction =
    when (code) {
        BillingResponseCode.OK -> BillingAction.Success
        BillingResponseCode.NETWORK_ERROR,
        BillingResponseCode.SERVICE_TIMEOUT ->
            BillingAction.RetrySimple
        BillingResponseCode.SERVICE_UNAVAILABLE,
        BillingResponseCode.ERROR ->
            BillingAction.RetryBackoff
        BillingResponseCode.SERVICE_DISCONNECTED ->
            BillingAction.Reconnect
        BillingResponseCode.ITEM_ALREADY_OWNED,
        BillingResponseCode.ITEM_NOT_OWNED ->
            BillingAction.RefreshPurchases
        BillingResponseCode.BILLING_UNAVAILABLE ->
            BillingAction.UserRetry
        BillingResponseCode.USER_CANCELED ->
            BillingAction.Ignore
        BillingResponseCode.ITEM_UNAVAILABLE ->
            BillingAction.RefreshProducts
        BillingResponseCode.FEATURE_NOT_SUPPORTED ->
            BillingAction.Fail("Feature not supported")
        BillingResponseCode.DEVELOPER_ERROR ->
            BillingAction.Fail("Developer error")
        else -> BillingAction.Fail("Unknown error")
    }
```



Then you wire up the decision tree to your actual retry and recovery logic:

```
// PBL 8.x
suspend fun executeBillingAction(
    action: BillingAction,
    operation: suspend () -> BillingResult
): BillingResult = when (action) {
    is BillingAction.Success -> BillingResult
        .newBuilder()
        .setResponseCode(BillingResponseCode.OK)
        .build()
    is BillingAction.RetrySimple ->
        retryOnBillingError { operation() }
    is BillingAction.RetryBackoff ->
        retryWithBackoff { operation() }
    is BillingAction.Reconnect -> {
        ensureConnected(); operation()
    }
    is BillingAction.RefreshPurchases -> {
        refreshPurchases(); operation()
    }
    is BillingAction.RefreshProducts -> operation()
    is BillingAction.UserRetry,
    is BillingAction.Ignore,
    is BillingAction.Fail -> BillingResult
        .newBuilder()
        .setResponseCode(BillingResponseCode.ERROR)
        .build()
}
```

This pattern separates the decision about what to do from the execution of that decision. You can test the decision logic with simple unit tests against response codes, without needing a real `BillingClient`. The execution layer handles the actual retries, reconnections, and cache refreshes.

Putting It All Together

In production, your billing wrapper ties all of these pieces together. Every operation goes through the decision tree, and the wrapper handles retries transparently:

```
// PBL 8.x
suspend fun queryProducts(
    params: QueryProductDetailsParams
): QueryProductDetailsResult {
    var lastResult: QueryProductDetailsResult? = null
    for (attempt in 1..3) {
        val result =
            billingClient.queryProductDetails(params)
        lastResult = result
        if (result.billingResult.responseCode ==
            BillingResponseCode.OK
        ) return result
        if (result.billingResult.responseCode !in
            retriableCodes
        ) return result
        delay(2_000L * attempt)
    }
    return lastResult!!
}
```

```
// PBL 8.x
suspend fun acknowledgePurchase(
    token: String
): BillingResult {
    val params = AcknowledgePurchaseParams
        .newBuilder()
        .setPurchaseToken(token)
        .build()
    return retryWithBackoff {
        billingClient.acknowledgePurchase(params)
    }
}
```

The calling code does not need to worry about transient errors. The retry logic handles them automatically. When a non retriable error does surface, it propagates up to the UI layer where you can show the appropriate message.

What to Show Users

Error handling is not just about code. It is about the user experience when things go wrong. Here are practical guidelines:

106

- **Network errors during retry**: Show a subtle loading indicator. Do not flash error messages between retry attempts.
- **Final failure after retries exhausted**: Show a clear message like "Something went wrong. Please try again." with a retry button.
- **BILLING_UNAVAILABLE**: Show "Google Play is not available. Please check that you're signed in to the Play Store."
- **ITEM_ALREADY_OWNED**: Skip the error entirely. Refresh your purchase state and grant access silently.
- **USER_CANCELED**: Show nothing. Return to the previous screen.
- **Payment declined (sub response code)**: Show "Your payment method was declined. Please update your payment method in Google Play and try again."

Never show raw error codes or debug messages to users. Never show technical details about what went wrong. Keep messages actionable. Tell the user what they can do, not what the system failed to do.

107

Chapter 9: Backend Architecture for Billing

Everything you have built so far runs on the user's device. The client side code queries products, launches purchase flows, and receives purchase results through the Play Billing Library. But the client is not trustworthy. Any device the user controls can be manipulated: network traffic can be intercepted, APKs can be decompiled and modified, and local purchase data can be forged. If your app grants premium access based solely on what the client reports, someone will exploit it.

This chapter moves the conversation to your backend. You will learn why server side verification is non negotiable, how to set up the Google Play Developer API, and how to build a verification flow that is secure, reliable, and production ready.

Why Server Side Verification Is Non Negotiable

When a purchase completes on the client, the Play Billing Library gives you a `Purchase` object containing a purchase token, order ID, product ID, and a signature. You might think verifying the signature locally is enough. It is not.

Here is why. The signature verification key is embedded in your APK. Anyone can extract it. A modified APK can generate fake `Purchase` objects with valid looking signatures. Freedom, Lucky Patcher, and similar tools have been doing this for years. They intercept the communication between your app and Google Play Services, returning fabricated purchase responses that pass local signature checks.

Server side verification solves this by going directly to Google. Instead of trusting what the client says, your backend takes the purchase token and calls the Google Play Developer API. Google's servers respond with the authoritative purchase state: whether the purchase is valid, what product was purchased, whether it has been acknowledged, and its current subscription status. No client side tool can fake this response because it comes directly from Google's infrastructure over an authenticated API call.

There are three reasons this matters:

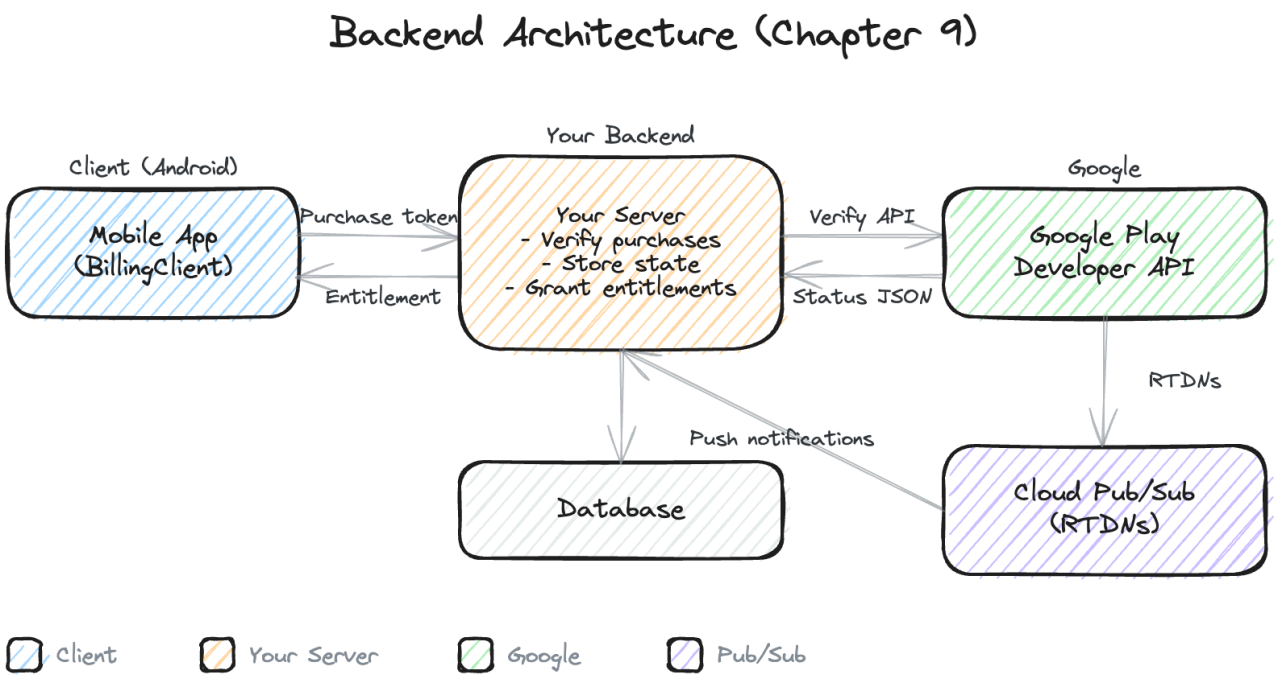
Fraud prevention. Without server verification, attackers can use modified clients to claim purchases that never happened. This means free access to your premium content and lost revenue. For apps with significant user bases, billing fraud can become a serious financial problem.

Accurate entitlement management. Subscription state changes constantly. Users cancel, payment methods fail, grace periods start and end, refunds happen. The client only sees a snapshot from the last time it queried. Your backend, combined with Real Time Developer Notifications, maintains a live view of every user's subscription state.

Auditability. When a user contacts support claiming they paid but lost access, you need server side records to investigate. Client side data alone cannot give you the full picture because the user may have switched devices, reinstalled the app, or cleared data.

The rule is simple: the client sends the purchase token to your backend. Your backend verifies it with Google. Only after verification does your backend grant the entitlement. The client never decides on its own whether a

purchase is valid.



The Canonical Purchase Verification Flow

Every purchase your app processes should follow these seven steps. This is the flow from the moment your client reports a successful purchase to the moment the user gains access to their content.

Step 1: Receive the purchase token from the client.

After `onPurchasesUpdated` fires with a successful result, your client extracts the purchase token from the `Purchase` object and sends it to your backend along with the product ID and the user's account identifier.

```
// Client side, PBL 8.x
fun onPurchaseCompleted(purchase: Purchase) {
    val payload = PurchasePayload(
        purchaseToken = purchase.purchaseToken,
        productId = purchase.products.first(),
        userId = currentUser.id
    )
    apiClient.verifyPurchase(payload)
}
```

Step 2: Call the Google Play Developer API.

Your backend receives the token and calls the appropriate API endpoint. For subscriptions, you call `purchases.subscriptionsv2.get`. For one time products, you call `purchases.products.get`. This chapter covers both endpoints in detail later.

109

```
// Server side Kotlin
fun verifySubscription(
    packageName: String,
    token: String
): SubscriptionPurchaseV2 {
    return androidPublisher
        .purchases()
        .subscriptionsv2()
        .get(packageName, token)
        .execute()
}
```

Step 3: Validate the response fields.

The API returns a detailed response object. You verify that the product ID matches what the client claimed, that the package name is correct, and that the response contains the expected fields. If any of these do not match, reject the purchase.

```
// Server side Kotlin
fun validateSubscriptionResponse(
    response: SubscriptionPurchaseV2,
    expectedProductId: String
): Boolean {
    val lineItem = response.lineItems?.firstOrNull()
        ?: return false
    return lineItem.productId == expectedProductId
}
```

Step 4: Check the purchase state.

For subscriptions, inspect the `subscriptionState` field. A value of `SUBSCRIPTION_STATE_ACTIVE` means the user has a valid, paying subscription. For one time products, check that `purchaseState` is `0` (purchased). Any other state means you should not grant access.

Step 5: Check for duplicate tokens.

Before granting an entitlement, query your database to see if this purchase token has already been processed. Duplicate submissions happen for many reasons: the client retried, the network delivered the request twice, or an attacker is replaying a legitimate token. If the token already exists in your database and is associated with a user, skip the entitlement grant.

110

```
// Server side Kotlin
fun isDuplicate(purchaseToken: String): Boolean {
    return purchaseRepository
        .findByToken(purchaseToken) != null
}
```

Step 6: Grant the entitlement.

If the purchase passes all checks, store the purchase record in your database and grant the user access to the purchased content. This is the moment the user's account state changes from "free" to "premium" (or whatever your product provides).

```
// Server side Kotlin
fun grantEntitlement(
    userId: String,
    purchaseToken: String,
    productId: String
) {
    purchaseRepository.save(
        PurchaseRecord(
            userId = userId,
            purchaseToken = purchaseToken,
            productId = productId,
            grantedAt = Instant.now()
        )
    )
    entitlementService.activate(userId, productId)
}
```

Step 7: Acknowledge the purchase.

After granting the entitlement, acknowledge the purchase with Google. You must acknowledge every purchase within 3 days or Google automatically refunds it. You can acknowledge from the client or the backend. Acknowledging from the backend is safer because it guarantees the acknowledgement only happens after verification and entitlement granting are complete.

```kotlin
// Server side Kotlin
fun acknowledgeSubscription(
    packageName: String,
    subscriptionId: String,
    token: String
) {
    androidPublisher
        .purchases()
        .subscriptions()
        .acknowledge(
            packageName,
            subscriptionId,
            token,
            SubscriptionPurchasesAcknowledgeRequest()
        )
        .execute()
}
```

These seven steps form the backbone of every billing backend. Get this flow right and you have a secure foundation. Skip any step and you open the door to fraud, lost revenue, or broken user experiences.

Setting Up Google Play Developer API Access

Before your backend can verify purchases, you need to configure API access in the Google Play Console. This involves creating a Google Cloud project, enabling the API, and setting up authentication credentials.

Step 1: Link a Google Cloud project.

Open the Google Play Console, navigate to **Setup > API access**, and link a Google Cloud project. If you do not have one, the Console offers to create one for you. This Cloud project is where your API credentials and quotas live.

Step 2: Enable the Google Play Android Developer API.

In the Google Cloud Console, navigate to **APIs & Services > Library**, search for "Google Play Android Developer API", and enable it. This unlocks the REST endpoints your backend will call.

Step 3: Create a service account.

Still in the Google Cloud Console, go to **IAM & Admin > Service Accounts** and create a new service account. Give it a descriptive name like `play-billing-backend`. You do not need to grant any Cloud IAM roles at this point because the permissions come from the Play Console side.

Step 4: Generate a JSON key.

On the service account's detail page, go to the **Keys** tab, click **Add Key > Create new key**, and select JSON. Download the key file and store it securely. This file contains the credentials your backend uses to authenticate. Treat it like a password: never commit it to version control, never expose it in logs, and rotate it periodically.

Step 5: Grant permissions in the Play Console.

Back in the Google Play Console under **Setup > API access**, you will see your service account listed. Click **Grant access** and assign the **Financial data** permission (to view purchases and subscriptions) and **Manage orders** permission (to acknowledge and refund). Apply these permissions to the specific app or to all apps in your account.

After these steps, your backend can authenticate as the service account and call the Google Play Developer API on behalf of your app.

Service Account Authentication (OAuth 2.0)

The Google Play Developer API uses OAuth 2.0 with service account credentials. Unlike user based OAuth flows that require browser redirects and consent screens, service account authentication happens entirely server to server. Your backend uses the JSON key file to generate short lived access tokens, then includes those tokens in API requests.

The Google API client libraries handle this for you. Here is how to set up the authenticated client in server side Kotlin using the `google-api-services-androidpublisher` library:

```
// Server side Kotlin
import com.google.api.client.googleapis.javanet
    .GoogleNetHttpTransport
import com.google.api.client.json.gson.GsonFactory
import com.google.api.services.androidpublisher
    .AndroidPublisher
import com.google.api.services.androidpublisher
    .AndroidPublisherScopes
import com.google.auth.http
    .HttpCredentialsAdapter
import com.google.auth.oauth2
    .GoogleCredentials
import java.io.FileInputStream
```


113

```kotlin
// Server side Kotlin
fun createPublisherClient(): AndroidPublisher {
    val credentials = GoogleCredentials
        .fromStream(
            FileInputStream("service-account.json")
        )
        .createScoped(
            listOf(
                AndroidPublisherScopes.ANDROIDPUBLISHER
            )
        )
    val transport =
        GoogleNetHttpTransport.newTrustedTransport()
    val jsonFactory = GsonFactory.getDefaultInstance()
    return AndroidPublisher.Builder(
        transport,
        jsonFactory,
        HttpCredentialsAdapter(credentials)
    )
        .setApplicationName("your-app-backend")
        .build()
}
```

The `GoogleCredentials.fromStream()` method reads the service account JSON key, and `createScoped()` requests the `androidpublisher` scope, which grants access to the Google Play Developer API. The credentials object handles token generation and refresh automatically. When a token expires, the library fetches a new one before the next API call.

In production, avoid reading the key file from disk on every request. Initialize the `AndroidPublisher` client once at application startup and reuse it. The client is thread safe, and the credentials object caches tokens internally.

For containerized deployments on Google Cloud, you can skip the JSON key file entirely and use Workload Identity Federation or the default service account attached to your compute instance. The `GoogleCredentials.getApplicationDefault()` method picks up credentials from the environment automatically:

114

```
// Server side Kotlin
val credentials = GoogleCredentials
    .getApplicationDefault()
    .createScoped(
        listOf(
            AndroidPublisherScopes.ANDROIDPUBLISHER
        )
    )
```

This is the preferred approach for production because it eliminates the need to manage key files.

Key API Endpoints

The Google Play Developer API provides several endpoints for managing purchases. Each serves a different purpose, and knowing when to use which one is important.

purchases.subscriptionsv2.get

This is the primary endpoint for subscription verification and the one you will use most often. It replaces the older `purchases.subscriptions.get` endpoint and returns richer data in a more consistent format.

You call it with your package name and the purchase token:

```
// Server side Kotlin
fun getSubscription(
    packageName: String,
    token: String
): SubscriptionPurchaseV2 {
    return androidPublisher
        .purchases()
        .subscriptionsv2()
        .get(packageName, token)
        .execute()
}
```

The response includes:

- **`subscriptionState`**: The current state of the subscription. Values include `SUBSCRIPTION_STATE_ACTIVE`, `SUBSCRIPTION_STATE_CANCELED`, `SUBSCRIPTION_STATE_IN_GRACE_PERIOD`, `SUBSCRIPTION_STATE_ON_HOLD`, `SUBSCRIPTION_STATE_PAUSED`, `SUBSCRIPTION_STATE_EXPIRED`, and `SUBSCRIPTION_STATE_PENDING`.
- **`lineItems`**: A list of items in the subscription. Each line item contains the `productId`, `expiryTime`, `offerDetails`, and `autoRenewingPlan` or `prepaidPlan` information.

115

- `linkedPurchaseToken`: If this subscription replaced a previous subscription (via upgrade or downgrade), this field contains the token of the previous purchase. More on this in the linked token section below.
- `acknowledgementState`: Whether the purchase has been acknowledged. `ACKNOWLEDGEMENT_STATE_ACKNOWLEDGED` or `ACKNOWLEDGEMENT_STATE_PENDING`.
- `externalAccountIdentifiers`: The obfuscated account ID and profile ID you passed during the purchase flow. Use these to match the purchase to a user in your system.

This endpoint is the source of truth for subscription state. When you need to know whether a user's subscription is active, call this endpoint. Do not rely on cached client side data.

purchases.products.get

This endpoint verifies one time product purchases (both consumable and non consumable):

```
// Server side Kotlin
fun getProduct(
    packageName: String,
    productId: String,
    token: String
): ProductPurchase {
    return androidPublisher
        .purchases()
        .products()
        .get(packageName, productId, token)
        .execute()
}
```

Note that this endpoint requires the product ID as a path parameter, unlike `subscriptionsv2.get` which only needs the token. The response includes:

- `purchaseState`: An integer where `0` means purchased, `1` means canceled, and `2` means pending.
- `consumptionState`: `0` for not consumed, `1` for consumed.
- `acknowledgementState`: `0` for not acknowledged, `1` for acknowledged.
- `purchaseTimeMillis`: When the purchase was made.
- `orderId`: The order ID for this transaction.

For one time products, the important check is that `purchaseState` is `0`. A purchase in state `1` (canceled) or `2` (pending) should not result in an entitlement grant.

purchases.subscriptions.acknowledge

After verifying a subscription and granting the entitlement, you must acknowledge the purchase. This endpoint tells Google that you have fulfilled the purchase:

116

```kotlin
// Server side Kotlin
fun acknowledgeSubscription(
    packageName: String,
    subscriptionId: String,
    token: String
) {
    val request =
        SubscriptionPurchasesAcknowledgeRequest()
    androidPublisher
        .purchases()
        .subscriptions()
        .acknowledge(
            packageName,
            subscriptionId,
            token,
            request
        )
        .execute()
}
```

You must acknowledge within 3 days of purchase. Failing to acknowledge causes an automatic refund. In a well designed system, acknowledgement happens immediately after entitlement granting in Step 7 of the verification flow.

If acknowledgement fails due to a transient error, retry it. An unacknowledged purchase sitting in your system is a ticking time bomb that will result in a refund and a confused user.

purchases.products.acknowledge and purchases.products.consume

For one time products, you have two options depending on the product type.

Non consumable products use the acknowledge endpoint:

117

```
// Server side Kotlin
fun acknowledgeProduct(
    packageName: String,
    productId: String,
    token: String
) {
    val request = ProductPurchasesAcknowledgeRequest()
    androidPublisher
        .purchases()
        .products()
        .acknowledge(
            packageName,
            productId,
            token,
            request
        )
        .execute()
}
```

Consumable products use the consume endpoint instead. Consuming a product also acknowledges it, so you do not need to call both:

```
// Server side Kotlin
fun consumeProduct(
    packageName: String,
    productId: String,
    token: String
) {
    androidPublisher
        .purchases()
        .products()
        .consume(
            packageName,
            productId,
            token
        )
        .execute()
}
```

The distinction matters. If you acknowledge a consumable product instead of consuming it, the user cannot purchase it again. If you consume a non consumable product, the user loses permanent ownership. Match the API call to the product type.

118

Using purchaseToken as the Primary Key

When you store purchase records in your database, use the `purchaseToken` as the primary identifier, not the `orderId`. This is a common source of confusion because the order ID looks like a natural key, but purchase tokens are the correct choice for several reasons.

First, the purchase token is what the Google Play Developer API accepts as input. Every verification call, every acknowledgement, every consumption call takes a purchase token. If you key your records on order ID, you need a lookup step to find the token every time you interact with the API.

Second, for subscriptions, a single purchase token covers the entire lifecycle of a subscription, including all renewals. The order ID changes with every renewal (appending an incrementing sequence number like `..0`, `..1`, `..2`). If you key on order ID, you end up with dozens of records for what is logically one subscription.

Third, when a subscription is upgraded, downgraded, or otherwise replaced, the new subscription's response includes a `linkedPurchaseToken` pointing to the previous purchase token. This creates a chain that lets you trace the full history of a user's subscription. If you key on order ID, you lose this linkage.

Here is a practical schema:

```
// Server side Kotlin
data class PurchaseRecord(
    val purchaseToken: String,  // Primary key
    val userId: String,
    val productId: String,
    val orderId: String?,
    val purchaseState: String,
    val linkedPurchaseToken: String?,
    val acknowledgedAt: Instant?,
    val grantedAt: Instant,
    val createdAt: Instant
)
```

The `orderId` is still useful for customer support (users see it in their Google Play receipts) and financial reconciliation. Store it, but do not use it as your primary key.

One edge case to watch for: pending purchases. When a user chooses a delayed payment method (like cash payment at a convenience store in some markets), the purchase token is generated immediately but the `purchaseState` is `2` (pending). You should store the token but not grant the entitlement until the purchase transitions to state `0`. You learn about the state change through Real Time Developer Notifications.

Handling linkedPurchaseToken Chains

When a user upgrades or downgrades their subscription, Google creates a new purchase token for the replacement subscription. The new token's API response includes a `linkedPurchaseToken` field that points

to the token it replaced. This creates a chain of tokens that represents the user's subscription history.

Consider a user who starts on a monthly plan, upgrades to annual, and then downgrades back to monthly:

1. Initial monthly subscription: token `A` (no linked token)
2. Upgrade to annual: token `B` (linked to `A`)
3. Downgrade to monthly: token `C` (linked to `B`)

Your backend must handle this chain correctly. Specifically, when you process a new purchase with a `linkedPurchaseToken`, you must:

1. Look up the linked token in your database.
2. Revoke the entitlement associated with the old token (mark it as superseded, not active).
3. Grant a new entitlement for the new token.

If you skip step 2, the user ends up with two active entitlements. This becomes a problem if you track usage limits, concurrent streams, or any feature that depends on a user having exactly one active subscription.

```
// Server side Kotlin
fun processSubscriptionPurchase(
    userId: String,
    purchaseToken: String,
    subscription: SubscriptionPurchaseV2
) {
    val linkedToken = subscription.linkedPurchaseToken
    if (linkedToken != null) {
        purchaseRepository
            .deactivate(linkedToken)
    }
    if (!isDuplicate(purchaseToken)) {
        grantEntitlement(
            userId,
            purchaseToken,
            subscription.lineItems
                .first().productId
        )
    }
}
```

In rare cases, chains can be longer than two tokens. A user who upgrades and downgrades multiple times might have a chain of five or six tokens. Your code does not need to walk the entire chain every time. When you process a new token, you only need to deactivate the immediately linked token. If your system processes tokens in order (which Real Time Developer Notifications help ensure), each previous token in the chain was already deactivated when the next one arrived.

120

However, if you are backfilling data or recovering from a processing failure, you may need to walk the full chain. In that case, follow each `linkedPurchaseToken` until you reach a token with no linked token (the original purchase):

```
// Server side Kotlin
fun resolveTokenChain(
    packageName: String,
    currentToken: String
): List<String> {
    val chain = mutableListOf(currentToken)
    var token = currentToken
    while (true) {
        val sub = getSubscription(packageName, token)
        val linked = sub.linkedPurchaseToken
            ?: break
        chain.add(linked)
        token = linked
    }
    return chain.reversed()
}
```

Be careful with this approach in production. Each step in the chain requires an API call, and long chains can eat into your quota. Cache the chain structure in your database after the first traversal so you do not need to repeat it.

There is another subtlety. When a user resubscribes after cancellation (as opposed to upgrading or downgrading), Google does not always set `linkedPurchaseToken`. A resubscription may produce an entirely new, unlinked token. This means you cannot rely solely on token chains to associate all of a user's subscriptions. Always use the `externalAccountIdentifiers` (the obfuscated account ID you pass during purchase) or your own user to token mapping to tie subscriptions to users.

Rate Limiting and Quota Management

The Google Play Developer API enforces quota limits. As of the current API version, the default quota is 200,000 queries per day for `purchases.subscriptionsv2.get` and `purchases.products.get` combined. This sounds like a lot, but apps with large subscriber bases can hit this limit, especially during incident recovery when you need to re verify many purchases.

Here are the strategies that work well for staying within quota:

Cache verification results. When your backend verifies a purchase, store the result. For the next hour (or whatever window makes sense for your app), serve the cached result instead of calling the API again. Subscription state does not change every second. A cache TTL of 5 to 15 minutes is a reasonable starting point.

121

```kotlin
// Server side Kotlin
fun getSubscriptionCached(
    packageName: String,
    token: String
): SubscriptionPurchaseV2 {
    val cacheKey = "sub:$token"
    val cached = cache.get(cacheKey)
    if (cached != null) return cached
    val result = getSubscription(
        packageName, token
    )
    cache.put(cacheKey, result, ttl = 10.minutes)
    return result
}
```

Use Real Time Developer Notifications. Instead of polling the API to check if subscription state changed, configure RTDN through Google Cloud Pub/Sub. Google pushes notifications to your backend when subscriptions renew, cancel, enter grace periods, or experience payment issues. This eliminates the need for periodic polling and dramatically reduces API calls.

Batch verification during startup. When your app launches, it might query purchases for the current user. Instead of having every app instance hit your backend (which then hits Google), your backend should rely on its local database for entitlement checks. Only call the Google API when processing a new purchase or handling an RTDN.

Implement server side retry with backoff. When the API returns a 429 (Too Many Requests) or 503 (Service Unavailable), back off exponentially. The API response includes a `Retry-After` header that tells you how long to wait.

122

```kotlin
// Server side Kotlin
fun executeWithRetry(
    maxAttempts: Int = 3,
    block: () -> Any
): Any {
    repeat(maxAttempts) { attempt ->
        try {
            return block()
        } catch (e: GoogleJsonResponseException) {
            if (e.statusCode == 429 ||
                e.statusCode == 503
            ) {
                val delay = (1L shl attempt) * 1000L
                Thread.sleep(delay)
            } else {
                throw e
            }
        }
    }
    throw RuntimeException("Retries exhausted")
}
```

Monitor your quota usage. Set up alerts in the Google Cloud Console when your daily API usage approaches 80% of your quota. If you consistently approach the limit, you can request a quota increase through the Cloud Console. Google typically grants reasonable increases for production apps with legitimate traffic.

Avoid verifying on every app launch. A common mistake is calling the Google Play Developer API every time a user opens your app to check if their subscription is still valid. This burns through quota quickly and adds latency to your app launch. Instead, trust your local database for entitlement checks and update it only when processing RTDNs or new purchases. Your database is your working copy of entitlement state. The Google API is the authoritative source you sync from, not a real time query service.

123

Chapter 10: Real Time Developer Notifications (RTDN)

Every subscription lifecycle event that happens outside your app, a renewal, a cancellation, a payment failure, happens on Google's servers. If your backend does not learn about these events quickly, your users see stale entitlement states, your analytics miss subscription churn, and your support team cannot resolve billing disputes. Real Time Developer Notifications solve this by pushing subscription and purchase events to your backend the moment they occur.

This chapter covers the full RTDN pipeline: why polling is insufficient, how the architecture works, how to set up Cloud Pub/Sub, and how to process every notification type in production.

Why Polling Is Not Enough: The Case for Real Time Notifications

Without RTDN, the only way to know what happened to a subscription is to call the Google Play Developer API and ask. This polling approach has several problems.

First, there is latency. If you poll every 15 minutes, a user who cancels their subscription might retain access for up to 15 minutes after cancellation. For most apps this is tolerable, but if you poll every hour or every few hours, the gap becomes noticeable.

Second, there is cost. Every call to the `purchases.subscriptionsv2.get` endpoint counts against your API quota. If you have 100,000 active subscribers and poll each one every 15 minutes, that is 9.6 million API calls per day. Google's default quota for the Android Publisher API is generous but not unlimited, and you are spending compute and bandwidth on requests where most of the time nothing has changed.

Third, there is complexity. You need to maintain a scheduler that iterates through every active subscription, handles pagination, manages rate limits, and deals with transient failures. This is a lot of infrastructure for a problem that has a simpler solution.

RTDN flips the model. Instead of asking Google "has anything changed?", Google tells you "something changed." You receive a notification within seconds of a state change, and you only call the API to fetch the full details for that specific subscription. No wasted calls. No polling infrastructure. No stale states lasting minutes or hours.

In practice, most production billing systems use both approaches. RTDN handles the real time path for immediate state updates. A periodic poll acts as a safety net to catch any notifications that might have been missed due to infrastructure issues. But RTDN is the primary mechanism, and setting it up should be one of the first things you do when building server side billing logic.

RTDN Architecture: Google Play, Cloud Pub/Sub, and Your Backend

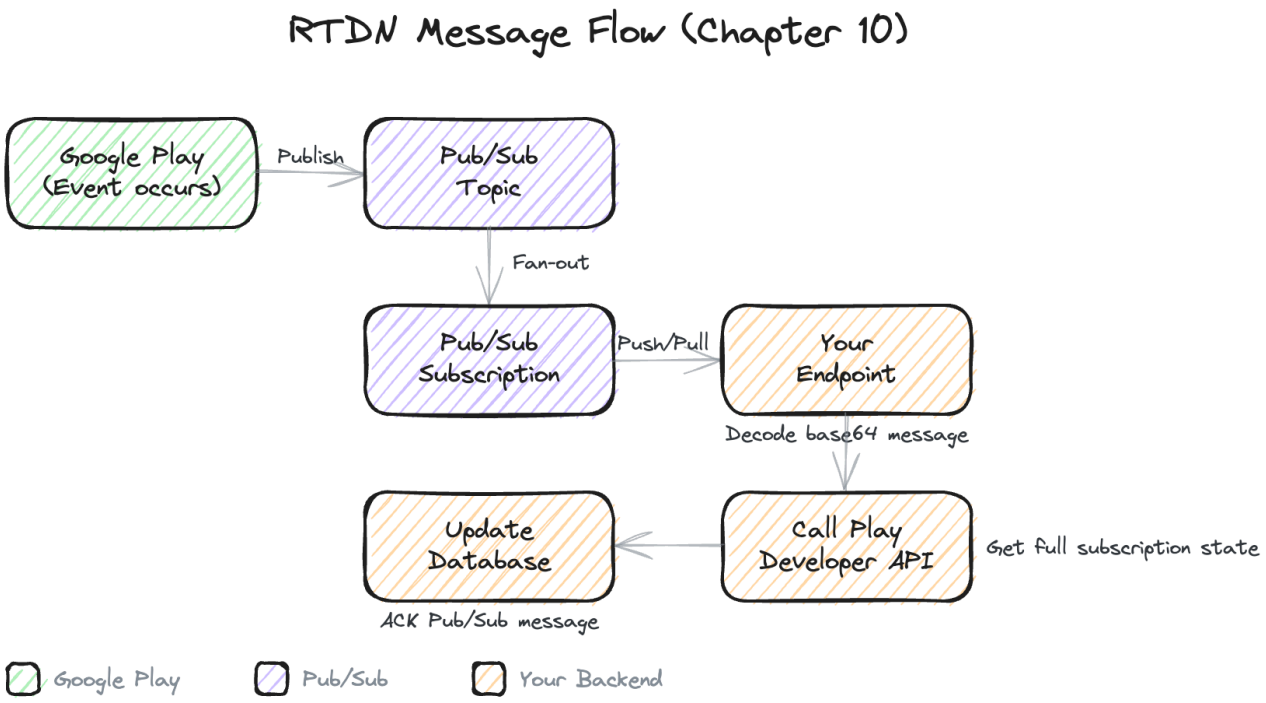
RTDN uses Google Cloud Pub/Sub as the delivery mechanism. Here is how the pieces connect:

1. **Google Play** detects a subscription or purchase state change (renewal, cancellation, refund, etc.).
2. **Google Play** publishes a notification message to a Cloud Pub/Sub topic that you own.

3. **Cloud Pub/Sub** delivers the message to a subscription (push or pull) that your backend consumes.
4. **Your backend** receives the notification, decodes the message, and calls the Google Play Developer API to get the full, current state of the purchase.

The important detail here is that you own the Pub/Sub topic and subscription. You create them in your own Google Cloud project. Google Play simply has publish permission to your topic. This means you control delivery settings, retry policies, dead letter handling, and monitoring.

The notification itself is a lightweight signal. It tells you which purchase changed and what type of change occurred, but it does not contain the full purchase state. You must call the API to get the complete picture. This is by design, and it is the golden rule of RTDN processing that you will see later in this chapter.



Push vs. Pull Subscription Strategies

Cloud Pub/Sub offers two ways to consume messages: push subscriptions and pull subscriptions. Each has trade offs for RTDN processing.

Push Subscriptions

With a push subscription, Pub/Sub sends an HTTP POST request to an endpoint you specify whenever a message arrives. Your backend exposes an HTTPS URL, Pub/Sub delivers the message to that URL, and your server responds with a 200 status code to acknowledge receipt.

Push works well when:

- You already have a web server or API gateway running.
- You want the simplest possible setup with no polling loop on your side.
- You need low latency processing (messages arrive as soon as they are published).

125

The main consideration is reliability. Your endpoint must be publicly accessible and respond quickly. If your server is down or responds with a non 2xx status code, Pub/Sub retries with exponential backoff. You need to handle duplicate deliveries because Pub/Sub guarantees at least once delivery, not exactly once.

Pull Subscriptions

With a pull subscription, your backend actively requests messages from Pub/Sub. You make a pull request, Pub/Sub returns any pending messages, and you acknowledge each one after processing.

Pull works well when:

- Your backend runs in an environment without a public URL (batch processing, internal services).
- You want fine grained control over when and how fast you process messages.
- You want to batch process notifications in groups.

The trade off is that you need to run a polling loop or use streaming pull, which adds infrastructure complexity. For most web backends, push is the simpler choice.

Which Should You Choose?

For most apps, **push** is the better starting point. You get a webhook style integration with minimal setup, and Cloud Pub/Sub handles the retry logic for you. If you later need more control over processing rate or if your architecture is not well suited to accepting inbound HTTP requests, you can switch to pull without changing anything on the Google Play side. The Pub/Sub topic stays the same, and you just create a different type of subscription.

Setting Up Cloud Pub/Sub

Setting up RTDN requires four steps: creating a topic, creating a subscription, granting publish rights to Google, and enabling RTDN in the Play Console.

Creating a Topic

Open the Google Cloud Console for your project and navigate to Pub/Sub. Create a new topic with a descriptive name, for example `play-billing-notifications`. This topic is where Google Play will publish all notification messages for your app.

You can also create the topic using the `gcloud` CLI:

```
gcloud pubsub topics create play-billing-notifications
```

One topic handles all notification types (subscription events, one time product events, voided purchases). You do not need separate topics for different event types.

Creating a Push or Pull Subscription

126

After creating the topic, create a subscription attached to it.

For a **push subscription**, specify the HTTPS endpoint where your backend will receive messages:

```
gcloud pubsub subscriptions create play-billing-sub \
  --topic=play-billing-notifications \
  --push-endpoint=https://api.yourapp.com/billing/rtdn \
  --ack-deadline=60
```

The `ack-deadline` is the number of seconds Pub/Sub waits for your server to acknowledge the message before retrying. Set this based on how long your notification processing takes. 60 seconds is a reasonable starting point.

For a **pull subscription**, omit the push endpoint:

```
gcloud pubsub subscriptions create play-billing-sub \
  --topic=play-billing-notifications \
  --ack-deadline=60
```

Granting Publish Rights to Google's Service Account

Google Play publishes notifications using a specific service account: `google-play-developer-notifications@system.gserviceaccount.com`. You must grant this account the **Pub/Sub Publisher** role on your topic.

In the Google Cloud Console, go to your topic, open the Permissions tab, and add the service account with the `roles/pubsub.publisher` role.

Using the CLI:

```
gcloud pubsub topics add-iam-policy-binding \
  play-billing-notifications \
  --member=serviceAccount:google-play-developer-notifications@system.gserviceaccount.com \
  --role=roles/pubsub.publisher
```

Without this step, Google Play cannot publish to your topic and you will not receive any notifications. If RTDN appears to be set up correctly but you are not receiving messages, missing publisher permissions is the first thing to check.

Enabling RTDN in Play Console

The final step is to tell Google Play which Pub/Sub topic to use. In the Google Play Console:

127

1. Navigate to **Monetization setup** (under "Monetize" in the left menu).
2. Scroll to the **Real time developer notifications** section.
3. Enter your full topic name in the format: `projects/YOUR_PROJECT_ID/topics/play-billing-notifications`.
4. Click **Send test notification** to verify the connection.
5. Save your changes.

The test notification sends a test message to your topic. If you have a push subscription set up, your endpoint should receive a POST request within a few seconds. If you are using a pull subscription, pull from the subscription to confirm the message arrived. A successful test notification confirms that Google Play can publish to your topic and your subscription is wired up correctly.

Message Format: Base64 Encoded JSON in Pub/Sub Data Field

When a Pub/Sub message arrives at your backend (via push or pull), the notification payload is inside the message's `data` field as a base64 encoded JSON string. The structure of the push request body looks like this:

```
{
  "message": {
    "attributes": {
      "key": "value"
    },
    "data": "eyJ2ZXJzaW9uIjoiMS4wIiwicGFja2FnZU...",
    "messageId": "136969346945"
  },
  "subscription": "projects/myproject/subscriptions/play-billing-sub"
}
```

The `data` field contains the base64 encoded notification. You decode it to get the `DeveloperNotification` JSON object:

```
import java.util.Base64

fun decodePubSubMessage(base64Data: String): String {
    val decodedBytes = Base64.getDecoder()
        .decode(base64Data)
    return String(decodedBytes, Charsets.UTF_8)
}
```


128

After decoding, you parse the JSON string into a `DeveloperNotification` object. The next section covers that structure.

The DeveloperNotification Structure

The decoded JSON has the following top level structure:

```
{
  "version": "1.0",
  "packageName": "com.example.app",
  "eventTimeMillis": "1234567890123",
  "subscriptionNotification": { ... },
  "oneTimeProductNotification": { ... },
  "voidedPurchaseNotification": { ... },
  "testNotification": { ... }
}
```

Every notification includes `version`, `packageName`, and `eventTimeMillis`. Exactly one of the four notification type fields will be present in any given message:

- **`subscriptionNotification`**: A subscription lifecycle event occurred.
- **`oneTimeProductNotification`**: A one time product event occurred.
- **`voidedPurchaseNotification`**: A purchase was voided (refunded or charged back).
- **`testNotification`**: A test message sent from the Play Console.

Here is a Kotlin data model for the full structure:

```
data class DeveloperNotification(
    val version: String,
    val packageName: String,
    val eventTimeMillis: Long,
    val subscriptionNotification:
        SubscriptionNotification? = null,
    val oneTimeProductNotification:
        OneTimeProductNotification? = null,
    val voidedPurchaseNotification:
        VoidedPurchaseNotification? = null,
    val testNotification:
        TestNotification? = null
)
```

Your processing logic checks which field is non null and routes to the appropriate handler:

```
fun processNotification(
    notification: DeveloperNotification
) {
    when {
        notification.subscriptionNotification != null ->
            handleSubscription(
                notification.subscriptionNotification
            )
        notification.oneTimeProductNotification != null ->
            handleOneTimeProduct(
                notification.oneTimeProductNotification
            )
        notification.voidedPurchaseNotification != null ->
            handleVoidedPurchase(
                notification.voidedPurchaseNotification
            )
        notification.testNotification != null ->
            handleTest(notification.testNotification)
    }
}
```

Processing Subscription Notifications

The `subscriptionNotification` field contains two pieces of information:

```
data class SubscriptionNotification(
    val version: String,
    val notificationType: Int,
    val purchaseToken: String,
    val subscriptionId: String
)
```

The `notificationType` is an integer that tells you what happened. The `purchaseToken` and `subscriptionId` tell you which subscription was affected. There are 14 subscription notification types, each representing a different lifecycle event.

SUBSCRIPTION_RECOVERED (1)

The user's subscription was recovered from an account hold. This means a payment that had previously failed has now succeeded, and the subscription is active again. When you receive this, call the API to confirm the subscription is active and restore the user's access immediately.

130

SUBSCRIPTION_RENEWED (2)

The subscription successfully renewed for a new billing period. This is the most common notification you will see for healthy subscriptions. Call the API to get the updated expiry time and extend the user's access accordingly.

SUBSCRIPTION_CANCELED (3)

The user canceled their subscription. This does not mean they lost access immediately. The subscription remains active until the end of the current billing period. When you receive this notification, call the API to check the `expiryTimeMillis`. Continue granting access until that time, then revoke it.

Do not revoke access the moment you receive this notification. The user paid for the current period and is entitled to use the service until it expires.

SUBSCRIPTION_PURCHASED (4)

A new subscription was purchased. You will typically process this on the client side first through the Play Billing Library, but this notification serves as a server side confirmation. Use it to verify that your backend recorded the purchase correctly. If you rely solely on client side purchase handling, this notification acts as a safety net for cases where the app crashed or the user switched devices before your client could report the purchase.

SUBSCRIPTION_ON_HOLD (5)

The subscription entered account hold because of a payment failure. During account hold, the user should not have access to subscription content. Revoke access when you receive this notification, but do not cancel the subscription in your system. The user's payment method may recover, and you will receive a `SUBSCRIPTION_RECOVERED` notification if it does.

Account hold can last up to 30 days by default (configurable in the Play Console). If the payment does not recover within the hold period, the subscription expires.

SUBSCRIPTION_IN_GRACE_PERIOD (6)

The subscription entered a grace period because the renewal payment failed. Unlike account hold, the user retains access during the grace period. You should continue granting access but may want to show a message prompting the user to update their payment method.

Grace periods are shorter than account holds (typically 3, 7, 14, or 30 days, configurable in the Play Console). If payment recovers during the grace period, you receive `SUBSCRIPTION_RECOVERED`. If it does not, the subscription moves to account hold or expires directly, depending on your configuration.

SUBSCRIPTION_RESTARTED (7)

A user resubscribed to a subscription that was previously canceled but had not yet expired. The user went into Google Play subscriptions management and tapped "Resubscribe" before the expiry date. When you receive this, call the API to confirm the subscription is active and auto renewing again.

131

SUBSCRIPTION_PRICE_CHANGE_CONFIRMED (8)

The user confirmed a pending price change for their subscription. If you raised the price of a subscription and Google required user opt in, this notification tells you the user accepted the new price. Their next renewal will be at the new price.

SUBSCRIPTION_DEFERRED (9)

The subscription's renewal date was pushed forward, typically through a developer initiated API call using the `defer` endpoint. This is used for granting free extensions (for example, as a customer support gesture). Call the API to get the new expiry time.

SUBSCRIPTION_PAUSED (10)

The subscription entered a paused state. The user chose to pause their subscription, and the pause has now taken effect. Revoke access when you receive this notification. The subscription will automatically resume at the end of the pause period, at which point you will receive a `SUBSCRIPTION_RENEWED` notification.

SUBSCRIPTION_PAUSE_SCHEDULE_CHANGED (11)

The user modified their pause schedule. This could mean they scheduled a pause that has not started yet, changed when the pause will happen, or canceled a pending pause. Call the API to get the current subscription state and update your records accordingly.

SUBSCRIPTION_REVOKED (12)

The subscription was revoked. This happens when a user requests a refund or when Google revokes the subscription for policy reasons. Revoke access immediately. Unlike cancellation, revocation means the user should lose access right away, regardless of how much time was left in the billing period.

SUBSCRIPTION_EXPIRED (13)

The subscription has fully expired. This is the terminal state. The user no longer has an active subscription. Revoke access if you have not already done so. This notification often comes after a cancellation when the billing period ends, or after account hold times out.

SUBSCRIPTION_PENDING_PURCHASE_CANCELED (14)

A pending purchase (such as a pending transaction waiting for parental approval or a slow payment method) was canceled before it completed. No access should have been granted for this purchase, so no revocation is needed. Clean up any pending records you may have created.

A Complete Subscription Handler

Here is a handler that routes each notification type to the appropriate business logic:

```
fun handleSubscription(
    notification: SubscriptionNotification
) {
    val token = notification.purchaseToken
    val subId = notification.subscriptionId

    when (notification.notificationType) {
        1 -> onRecovered(token, subId)
        2 -> onRenewed(token, subId)
        3 -> onCanceled(token, subId)
        4 -> onPurchased(token, subId)
        5 -> onHold(token, subId)
        6 -> onGracePeriod(token, subId)
        7 -> onRestarted(token, subId)
        8 -> onPriceChangeConfirmed(token, subId)
        9 -> onDeferred(token, subId)
        10 -> onPaused(token, subId)
        11 -> onPauseScheduleChanged(token, subId)
        12 -> onRevoked(token, subId)
        13 -> onExpired(token, subId)
        14 -> onPendingPurchaseCanceled(token, subId)
        else -> log("Unknown type: " +
            "${notification.notificationType}")
    }
}
```

Each handler method follows the same pattern: call the Google Play Developer API with the purchase token, get the current subscription state, and update your database accordingly.

133

```
suspend fun onRenewed(
    purchaseToken: String,
    subscriptionId: String
) {
    val subscription = playApi
        .getSubscriptionV2(
            packageName, purchaseToken
        )
    val expiryTime = subscription
        .lineItems
        .first()
        .expiryTime
    database.updateExpiry(
        purchaseToken, expiryTime
    )
    log("Renewed: $subscriptionId, " +
        "expires: $expiryTime")
}
```

Processing One Time Product Notifications

One time product notifications are simpler. The structure contains:

```
data class OneTimeProductNotification(
    val version: String,
    val notificationType: Int,
    val purchaseToken: String,
    val sku: String
)
```

There are two notification types:

ONE_TIME_PRODUCT_PURCHASED (1)

A one time product was purchased. As with subscription purchases, you will usually process this client side first, but this notification confirms the purchase on the server side. Call the API to verify the purchase and grant the entitlement if you have not already.

ONE_TIME_PRODUCT_CANCELED (2)

A pending one time product purchase was canceled. This applies to purchases that were in a pending state (for example, waiting for a cash based payment to complete) and were then canceled. If you had provisionally granted access for the pending purchase, revoke it now.

134

Here is the handler:

```
fun handleOneTimeProduct(
    notification: OneTimeProductNotification
) {
    val token = notification.purchaseToken
    val sku = notification.sku

    when (notification.notificationType) {
        1 -> onOneTimePurchased(token, sku)
        2 -> onOneTimeCanceled(token, sku)
        else -> log("Unknown OTP type: " +
            "${notification.notificationType}")
    }
}
```

```
suspend fun onOneTimePurchased(
    purchaseToken: String,
    sku: String
) {
    val purchase = playApi
        .getProductPurchase(packageName, sku, purchaseToken)
    if (purchase.purchaseState == 0) { // Purchased
        database.grantEntitlement(
            purchaseToken, sku
        )
    }
}
```

Processing Voided Purchase Notifications

Voided purchase notifications tell you that a previously completed purchase has been reversed. This happens when Google issues a refund, the user's payment is charged back, or Google revokes a purchase for policy violations.

The structure:

135

```
data class VoidedPurchaseNotification(
    val purchaseToken: String,
    val orderId: String,
    val productType: Int,
    val refundType: Int
)
```

The `productType` tells you whether the voided purchase was a subscription (1) or a one time product (2). The `refundType` indicates the reason: full refund (1) or quantity based refund (2).

When you receive a voided purchase notification, revoke access to the associated content. The user received their money back, so they should no longer have the entitlement.

```
fun handleVoidedPurchase(
    notification: VoidedPurchaseNotification
) {
    val token = notification.purchaseToken
    val orderId = notification.orderId
    database.revokeEntitlement(token)
    log("Voided: order=$orderId, " +
        "type=${notification.productType}, " +
        "refund=${notification.refundType}")
}
```

Be careful with voided purchase processing. Some businesses choose to let users keep access after a refund as a goodwill gesture, while others strictly revoke. Your business rules dictate the right behavior, but you must at least record the voided purchase in your system for financial reconciliation.

The Golden Rule: RTDNs Signal State Changes, Always Call the API

This is the single most important principle for working with RTDN, and it bears repeating: **never make entitlement decisions based solely on the notification type. Always call the Google Play Developer API to get the full, current state.**

RTDNs are signals, not sources of truth. They tell you "something happened to this purchase token," but they do not give you the complete picture. Here is why this matters:

Notifications can arrive out of order. A `SUBSCRIPTION_CANCELED` notification might arrive after a `SUBSCRIPTION_RESTARTED` notification if there was a delivery delay. If you blindly revoke access on cancellation without checking the API, you would incorrectly revoke access for a user who already resubscribed.

136

Notifications can be duplicated. Pub/Sub guarantees at least once delivery, meaning you may receive the same notification more than once. If your handler is not idempotent, duplicates can cause incorrect state transitions.

The notification does not contain the full state. A `SUBSCRIPTION_RENEWED` notification does not tell you the new expiry time, the current price, or whether a price change is pending. Only the API response has that information.

The correct pattern is:

1. Receive the RTDN.
2. Extract the purchase token.
3. Call `purchases.subscriptionsv2.get` (for subscriptions) or `purchases.products.get` (for one time products) to get the full current state.
4. Update your database based on the API response, not the notification type.

The notification type is useful for logging, metrics, and routing (knowing which handler to call), but the API response is what you use for entitlement decisions.

```
suspend fun processSubscriptionNotification(
    notification: SubscriptionNotification
) {
    // Log the notification type for metrics
    metrics.increment(
        "rtdn.subscription.${notification.notificationType}"
    )

    // Always call the API for the full state
    val subscription = playApi
        .getSubscriptionV2(
            packageName,
            notification.purchaseToken
        )

    // Update entitlements based on API state
    updateEntitlementFromApiState(
        notification.purchaseToken,
        subscription
    )
}
```

Handling Notification Ordering and Deduplication

137

Because Cloud Pub/Sub guarantees at least once delivery and does not guarantee ordering, your RTDN processing must handle two scenarios: duplicate messages and out of order messages.

Deduplication

The simplest deduplication strategy uses the Pub/Sub message ID. Store each processed message ID and skip any message you have already seen:

```
suspend fun onPubSubMessage(
    messageId: String,
    data: DeveloperNotification
): Boolean {
    if (database.hasProcessed(messageId)) {
        log("Duplicate message: $messageId")
        return true // Acknowledge but skip
    }
    processNotification(data)
    database.markProcessed(messageId)
    return true
}
```

Keep the processed message IDs in a cache or database table with a TTL. Pub/Sub typically retries within minutes to hours, so a TTL of 24 to 48 hours covers most cases without unbounded storage growth.

Out of Order Processing

Out of order notifications are trickier. If you receive `SUBSCRIPTION_EXPIRED` before `SUBSCRIPTION_CANCELED`, naive processing would expire the user, then try to cancel an already expired subscription.

The solution is to always defer to the API response rather than the notification type. Since you call the API for every notification (following the golden rule), you always get the current state regardless of which notification triggered the call. If two notifications arrive out of order, both API calls return the same current state, and your database ends up correct.

You can add an extra safety measure by using the `eventTimeMillis` field from the notification. If a notification's event time is older than the last event time you processed for that purchase token, you can still call the API, but you know the state might have already moved past this event:

```
suspend fun processWithOrdering(
    notification: DeveloperNotification,
    subNotification: SubscriptionNotification
) {
    val lastEventTime = database
        .getLastEventTime(subNotification.purchaseToken)

    // Always call the API regardless of order
    val subscription = playApi
        .getSubscriptionV2(
            notification.packageName,
            subNotification.purchaseToken
        )
    updateEntitlementFromApiState(
        subNotification.purchaseToken,
        subscription
    )

    // Update the last event time if this is newer
    if (notification.eventTimeMillis > lastEventTime) {
        database.updateLastEventTime(
            subNotification.purchaseToken,
            notification.eventTimeMillis
        )
    }
}
```

This approach ensures you never miss a state update, even if notifications arrive in an unexpected order.

Estimating Pub/Sub Costs

Cloud Pub/Sub pricing has three components: message ingestion, message delivery, and storage for undelivered messages. For RTDN, the costs are typically very low.

Message volume. Each subscription lifecycle event generates one notification. A healthy monthly subscription generates roughly 12 renewal notifications per year, plus occasional cancellations, grace period events, and hold events. If you have 100,000 subscribers, expect roughly 1.2 million to 2 million messages per year from renewals alone, plus additional messages for lifecycle events.

Pricing. As of the current pricing model, Google Cloud offers the first 10 GB of message data per month free. Each Pub/Sub message for RTDN is tiny (a few hundred bytes). At 2 million messages per year with an average size of 500 bytes, that is about 1 GB per year of total data, well within the free tier.

139

When costs become meaningful. Pub/Sub costs start to matter when you have millions of subscribers generating tens of millions of notifications per month, or when you have many subscriptions attached to your topic. Even then, the per message cost is measured in fractions of a cent per 10,000 messages. For most apps, Pub/Sub costs for RTDN are effectively free.

The real cost of RTDN is not Pub/Sub itself. It is the Google Play Developer API calls you make in response to each notification. These calls count against your API quota, and if you have a very high volume of notifications, you may need to request a quota increase from Google. Monitor your API usage in the Google Cloud Console and request increases proactively before you hit the limit.

140

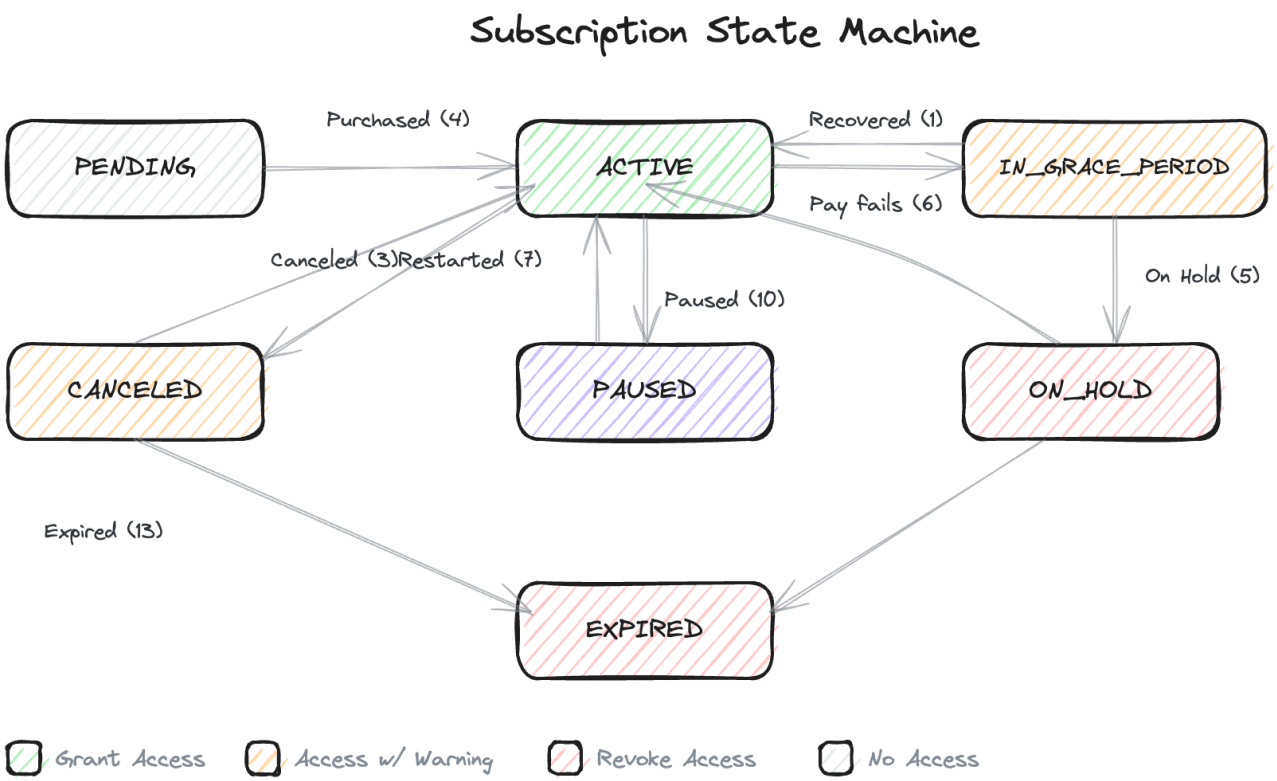
Chapter 11: The Subscription State Machine

A subscription is not simply "active" or "inactive." Between the moment a user subscribes and the moment their subscription ends for good, a subscription can pass through seven distinct states. Each state carries specific rules about whether the user gets access, what your app should show, how the Play Billing Library reports the purchase, and what your backend should do when it receives a notification.

If you get these states wrong, users either lose access they should have, or keep access they should not. Both outcomes cost you money and trust. This chapter is your definitive reference for every subscription state, the transitions between them, and the practical code decisions each state demands.

The 7 Subscription States

Google Play defines subscription state through the `subscriptionState` field on the `SubscriptionPurchaseV2` resource returned by the Google Play Developer API. There are seven possible values. Each one represents a specific phase in the subscription lifecycle.



ACTIVE

The subscription is in good standing. The user has paid, Google processed the payment successfully, and the subscription is either within its current billing period or has just renewed. This is the happy path.

When a subscription is `ACTIVE`, your app should grant full access to the subscribed content. The `expiryTime` field indicates when the current billing period ends. For auto renewing subscriptions, Google will

attempt to charge the user before this time. If the charge succeeds, the subscription stays `ACTIVE` and `expiryTime` advances to the end of the next billing period.

An `ACTIVE` subscription does not necessarily mean the user just paid. It means the most recent payment succeeded and the current period has not expired. A subscription can be `ACTIVE` for months or years as long as every renewal charge goes through.

IN_GRACE_PERIOD

A renewal payment failed, but Google is still trying to collect. The user retains access during this window.

Grace periods are configured in the Play Console under your subscription settings. You can set a grace period of 3, 7, 14, or 30 days. During this time, Google retries the payment using the user's payment method. If the retry succeeds, the subscription returns to `ACTIVE`. If it does not succeed within the grace period, the subscription moves to `ON_HOLD`.

This state exists because payment failures are often temporary. A credit card might be at its limit for a day, or a bank might flag an international transaction that the user then approves. Giving the user continued access while Google retries is better than cutting them off immediately over a transient payment issue.

Your app should still grant access during `IN_GRACE_PERIOD`, but you should also notify the user that their payment failed and ask them to update their payment method. Google provides a deep link you can use to send users directly to the Play Store payment settings.

ON_HOLD

The grace period expired without a successful payment. Access is revoked, but the subscription is not dead yet.

Account hold is also configured in the Play Console. You can enable account hold for up to 30 days. During this time, the user loses access to subscription content, but they can fix their payment method and reactivate. If the user updates their payment method and Google successfully charges them, the subscription returns to `ACTIVE`. If the hold period expires without recovery, the subscription moves to `EXPIRED`.

When a subscription is `ON_HOLD`, revoke access immediately. Show a message explaining that the subscription is on hold due to a payment issue and provide a way for the user to fix it. The goal is to recover the subscriber, not to punish them.

PAUSED

The user voluntarily paused their subscription. Access is revoked for the duration of the pause.

Pausing is an option you can enable in the Play Console. When enabled, users can pause their subscription for a period between one week and three months through the Play Store subscription management screen. During the pause, no billing occurs and the user does not have access.

When the pause period ends, Google resumes the subscription by charging the user. If the charge succeeds, the subscription returns to `ACTIVE`. If it fails, the subscription enters the grace period or account hold flow, just like a normal renewal failure.

142

Your app should recognize `PAUSED` as a voluntary action. Do not show alarming "payment failed" messages. Instead, inform the user that their subscription is paused and when it will resume.

CANCELED

The user canceled their subscription, but the current billing period has not ended yet. The user still has access until `expiryTime`.

This is one of the most commonly misunderstood states. Cancellation does not mean immediate loss of access. When a user cancels an auto renewing subscription, they keep access for the time they already paid for. The `expiryTime` field tells you exactly when access should end.

After `expiryTime` passes, the subscription transitions to `EXPIRED`. Until then, the user should have full access to everything their subscription entitles them to.

Your app can use this state to show retention messaging. A banner like "Your subscription ends on March 15. Resubscribe to keep access" gives the user a chance to change their mind. Google also provides a "Restore" button in the Play Store that lets users resubscribe before their current period ends.

EXPIRED

The subscription has ended. Either the user canceled and the billing period ran out, or the account hold period ended without payment recovery, or a prepaid subscription's time ran out.

When a subscription is `EXPIRED`, revoke access completely. The user is no longer a subscriber. They would need to purchase a new subscription to regain access.

The purchase token for an expired subscription remains valid for querying through the Google Play Developer API for 60 days after expiration. After that, the token becomes invalid. This 60 day window gives your backend time to process any final state changes and clean up records.

PENDING

The initial purchase has not completed yet. This happens with delayed payment methods like cash payments at convenience stores.

When a subscription starts in the `PENDING` state, do not grant access. The user has initiated the purchase but has not actually paid yet. If you grant access at this point, you risk giving away content for free if the payment never completes.

Once the payment processes, the subscription transitions to `ACTIVE` and you grant access at that point. If the payment fails or expires, the purchase is canceled and you never need to grant anything.

State Transitions and Their Triggers

Subscription states do not change randomly. Each transition is triggered by a specific event, and Google notifies you about most of these events through Real Time Developer Notifications (RTDNs). Understanding which notification type maps to which transition is essential for building a reliable backend.

143

Here is the complete map of state transitions, the events that cause them, and the RTDN types you receive:

144

FROM STATE	TO STATE	TRIGGER	RTDN TYPE
(new purchase)	ACTIVE	Successful payment	SUBSCRIPTION_PURCHASED (type 4)
(new purchase)	PENDING	Delayed payment initiated	None (detected via client purchase flow)
PENDING	ACTIVE	Delayed payment succeeds	SUBSCRIPTION_PURCHASED (type 4)
PENDING	EXPIRED	Payment fails or times out	SUBSCRIPTION_PENDING_PURCHASE_CANCELED (type 20)
ACTIVE	ACTIVE	Renewal succeeds	SUBSCRIPTION_RENEWED (type 2)
ACTIVE	IN_GRACE_PERIOD	Renewal payment fails	SUBSCRIPTION_IN_GRACE_PERIOD (type 6)
ACTIVE	CANCELED	User cancels	SUBSCRIPTION_CANCELED (type 3)
ACTIVE	PAUSED	Pause takes effect	SUBSCRIPTION_PAUSED (type 10)
ACTIVE	ON_HOLD	Payment fails (no grace period configured)	SUBSCRIPTION_ON_HOLD (type 5)
IN_GRACE_PERIOD	ACTIVE	Retry payment succeeds	SUBSCRIPTION_RECOVERED (type 1)
IN_GRACE_PERIOD	ON_HOLD	Grace period expires without payment	SUBSCRIPTION_ON_HOLD (type 5)
ON_HOLD	ACTIVE	User fixes payment	SUBSCRIPTION_RECOVERED (type 1)
ON_HOLD	EXPIRED	Hold period expires	SUBSCRIPTION_EXPIRED (type 13)
CANCELED	ACTIVE	User resubscribes	SUBSCRIPTION_RESTARTED (type 7)

145

FROM STATE	TO STATE	TRIGGER	RTDN TYPE
		before expiry	
CANCELED	EXPIRED	Billing period ends	SUBSCRIPTION_EXPIRED (type 13)
PAUSED	ACTIVE	Pause ends and payment succeeds	SUBSCRIPTION_RENEWED (type 2)
PAUSED	ON_HOLD	Pause ends but payment fails	SUBSCRIPTION_ON_HOLD (type 5)
EXPIRED	ACTIVE	User repurchases (new token)	SUBSCRIPTION_PURCHASED (type 4)

A few things to note about this table. First, some transitions produce the same RTDN type. `SUBSCRIPTION_RECOVERED` (type 1) fires both when a grace period retry succeeds and when a user fixes their payment during account hold. Your backend should check the new `subscriptionState` on the `SubscriptionPurchaseV2` resource rather than inferring state from the notification type alone.

Second, the transition from `CANCELED` back to `ACTIVE` via `SUBSCRIPTION_RESTARTED` only applies when the user resubscribes before their current period ends. If the subscription has already expired, a resubscription creates a new purchase token entirely.

Third, not every state transition sends an RTDN. The initial `PENDING` state is communicated through the purchase flow on the client, not through a server notification. You detect it when `onPurchasesUpdated` returns a purchase with `PurchaseState.PENDING`.

Which States Grant Access and Which Revoke It

This is the most practical question for your app: should the user have access right now? The answer depends entirely on the subscription state.

States That Grant Access

ACTIVE: Full access. No qualifications needed.

IN_GRACE_PERIOD: Full access. The user's payment failed, but they are still within the configured grace period. Continue granting access while showing a notification about the payment issue.

CANCELED: Full access until `expiryTime`. The user canceled, but they paid for the current period and should keep access until it ends. Check `expiryTime` and revoke access only after it passes.

States That Revoke Access

ON_HOLD: No access. The grace period expired without payment recovery. Revoke access and prompt the user to fix their payment method.

PAUSED: No access. The user voluntarily paused. Revoke access and show when the subscription will resume.

EXPIRED: No access. The subscription is over. Revoke access and offer a path to resubscribe.

States Where You Should Not Grant Access

PENDING: Do not grant access. Payment has not been completed. Wait for the state to transition to `ACTIVE`.

Here is a clean reference table:

STATE	GRANT ACCESS?	NOTES
ACTIVE	Yes	Standard entitlement
IN_GRACE_PERIOD	Yes	Show payment fix prompt
CANCELED	Yes (until expiryTime)	Show resubscribe prompt
ON_HOLD	No	Show payment fix prompt
PAUSED	No	Show resume date
EXPIRED	No	Show resubscribe option
PENDING	No	Show "payment pending" message

Your entitlement check function should encode these rules directly:

```
fun shouldGrantAccess(
    state: String,
    expiryTimeMillis: Long
): Boolean {
    val now = System.currentTimeMillis()
    return when (state) {
        "SUBSCRIPTION_STATE_ACTIVE",
        "SUBSCRIPTION_STATE_IN_GRACE_PERIOD" ->
            true
        "SUBSCRIPTION_STATE_CANCELED" ->
            now < expiryTimeMillis
        else -> false
    }
}
```


147

This function handles the three access granting states. `ACTIVE` and `IN_GRACE_PERIOD` always grant access. `CANCELED` grants access only if the current time is before `expiryTime`. Everything else (including `ON_HOLD`, `PAUSED`, `EXPIRED`, and `PENDING`) returns `false`.

How queryPurchasesAsync() Behaves for Each State

The `queryPurchasesAsync()` method on `BillingClient` returns purchases that Google Play considers "owned" by the user on the current device. But "owned" does not mean the same thing for every state. Understanding what this method returns for each subscription state is important for building correct client side entitlement logic.

What queryPurchasesAsync() Returns

`queryPurchasesAsync()` returns `Purchase` objects for subscriptions in the following states:

- **ACTIVE**: Returned. `purchaseState` is `PurchaseState.PURCHASED`.
- **IN_GRACE_PERIOD**: Returned. `purchaseState` is `PurchaseState.PURCHASED`. The purchase appears the same as an active subscription from the client's perspective.
- **CANCELED**: Returned (until `expiryTime`). `purchaseState` is `PurchaseState.PURCHASED`. The user still "owns" the subscription until the period ends.
- **ON_HOLD**: Returned. `purchaseState` is `PurchaseState.PURCHASED`. Even though the user should not have access, the purchase still appears in query results. Your app must check with your backend to determine the actual subscription state.
- **PAUSED**: Not returned. A paused subscription does not appear in `queryPurchasesAsync()` results.
- **EXPIRED**: Not returned. Once a subscription expires, it disappears from query results.
- **PENDING**: Returned. `purchaseState` is `PurchaseState.PENDING`.

This behavior has an important implication: you cannot rely on `queryPurchasesAsync()` alone to determine whether a user should have access. The method returns purchases for `ON_HOLD` subscriptions (which should not have access), and does not distinguish between `ACTIVE` and `IN_GRACE_PERIOD` states on the client side.

Why Backend Verification Matters

The `Purchase` object returned by `queryPurchasesAsync()` does not contain a `subscriptionState` field. It only gives you `purchaseState` (either `PURCHASED`, `PENDING`, or `UNSPECIFIED_STATE`). To get the actual subscription state, you need to call the Google Play Developer API on your backend using the purchase token.

This is why every subscription app needs a backend component. The client can tell you "the user has a purchase," but only the server can tell you "the user's subscription is on hold and they should not have access."

Here is a practical pattern for combining client and server checks:

```
suspend fun checkEntitlement(
    billingClient: BillingClient,
    api: YourBackendApi
): Boolean {
    val params = QueryPurchasesParams
        .newBuilder()
        .setProductType(ProductType.SUBS)
        .build()

    val result = billingClient
        .queryPurchasesAsync(params)

    val purchase = result.purchasesList
        .firstOrNull() ?: return false

    if (purchase.purchaseState ==
        PurchaseState.PENDING
    ) {
        return false
    }

    // Ask your backend for the real state
    return api.verifyEntitlement(
        purchase.purchaseToken
    )
}
```

The client side check filters out obvious non entitlements (no purchase at all, or a pending purchase). The backend call determines the actual subscription state and makes the final access decision. This two step approach keeps your app responsive while maintaining accuracy.

The ON_HOLD Gap

The fact that `ON_HOLD` subscriptions appear in `queryPurchasesAsync()` results with `PurchaseState.PURCHASED` is one of the most common sources of entitlement bugs. If your app only checks the client side purchase state, a user whose subscription is on hold will appear to have full access.

Always verify subscription state on your backend, especially when granting access to high value content. For lower stakes content, you might cache the backend verification result and refresh it periodically rather than checking on every app launch, but never skip it entirely.

Purchase Token Validity

A purchase token is the unique identifier for a subscription purchase. You use it to verify purchases with the Google Play Developer API, to acknowledge purchases, and to track subscription state on your backend. But tokens do not last forever.

The 60 Day Window

A purchase token remains valid for querying through the Google Play Developer API for up to 60 days after the subscription expires. During this window, you can call `purchases.subscriptionsv2.get` with the token and receive the full `SubscriptionPurchaseV2` resource, including the final subscription state.

After 60 days, the token becomes invalid. API calls with an expired token return an error. This means your backend must process all subscription lifecycle events and store the relevant state locally. You cannot rely on querying Google's API indefinitely.

Practical Implications

The 60 day window exists primarily for cleanup and reconciliation. Your backend should not be routinely querying expired tokens. Instead, process RTDNs as they arrive and keep your database up to date in real time.

However, the window is useful in a few scenarios:

- **Missed RTDNs**: If your server was down and missed a notification, you can query the token within 60 days to catch up on state changes.
- **Reconciliation jobs**: A periodic batch job can query all recently expired tokens to verify that your database state matches Google's records.
- **Customer support**: When a user reports a billing issue, your support team can look up the purchase token to see what Google's records show, as long as it is within the 60 day window.

Token Reuse After Expiration

Once a subscription expires, the user might resubscribe to the same product. When they do, Google issues a brand new purchase token. The old token and the new token are completely separate. The new subscription does not carry a `linkedPurchaseToken` pointing to the expired token unless it was specifically triggered through a resubscription flow.

Your backend should handle this by treating each purchase token as an independent subscription record. Map tokens to users through your own user account system, not by assuming any relationship between tokens for the same product.

Renewal Date Edge Cases

Subscription renewals happen on the same day of the month as the original purchase, where possible. But calendar months are not all the same length, and this creates edge cases that can confuse your reporting and your users.

Month End Drift

If a user subscribes on January 31st with a monthly billing period, what happens when February comes? February does not have a 31st day. Google handles this by renewing on the last day of the month. So the user renews on February 28th (or February 29th in a leap year).

But here is where it gets interesting. When March comes around, the renewal does not jump back to the 31st. It stays on the 28th. Once a renewal date drifts to a shorter month, it stays at the earlier date for subsequent months.

This means a subscription purchased on January 31st follows this pattern:

MONTH	RENEWAL DATE
January	31st (purchase date)
February	28th
March	28th
April	28th
May	28th

The date does not "recover" to the 31st. This is called month end drift, and it applies to all subscriptions purchased on the 29th, 30th, or 31st of a month.

Impact on Your App

Month end drift affects billing cycle calculations, analytics, and any UI that displays the next renewal date. If your app shows "next billing date" to the user, always read `expiryTime` from the subscription resource rather than calculating it yourself. Do not assume that a monthly subscription renews exactly 30 or 31 days after the previous renewal.

For reporting purposes, be aware that a subscription purchased on January 31st and one purchased on February 28th will share the same renewal date from March onward. If you track cohorts by purchase date, these users end up with different billing patterns despite having subscriptions of the same length.

Yearly Subscriptions and Leap Years

Annual subscriptions have a similar edge case. A subscription purchased on February 29th (leap year) renews on February 28th in non leap years. In the next leap year, it stays on February 28th rather than returning to the 29th. The same drift behavior applies.

What This Means for Your Backend

Never calculate expiry or renewal dates manually. Always use the `expiryTime` field from the `SubscriptionPurchaseV2` resource. Google handles all the calendar math, and their calculation is the source of truth. If your manually calculated date differs from Google's `expiryTime`, Google's value wins, and your user's experience should reflect that.

151

```
// Do this
fun getNextRenewalDate(
    subscription: SubscriptionPurchaseV2
): Instant {
    val expiryMillis = subscription
        .lineItems.first()
        .expiryTime
        .toLong()
    return Instant.ofEpochMilli(expiryMillis)
}

// Do NOT do this
fun calculateNextRenewal(
    purchaseDate: LocalDate
): LocalDate {
    // This will be wrong for month-end dates
    return purchaseDate.plusMonths(1)
}
```

Building a Robust State Handler

Now that you understand every state, let us look at how to build a backend notification handler that processes state changes correctly. Your RTDN handler receives a notification, queries the subscription state, and updates your entitlement database accordingly.

152

```
fun handleSubscriptionNotification(
    notification: SubscriptionNotification
) {
    val token = notification.purchaseToken
    val subscription = playApi
        .getSubscriptionV2(packageName, token)
    val state = subscription.subscriptionState

    when (state) {
        "SUBSCRIPTION_STATE_ACTIVE" ->
            grantAccess(token)
        "SUBSCRIPTION_STATE_IN_GRACE_PERIOD" ->
            grantAccessWithWarning(token)
        "SUBSCRIPTION_STATE_ON_HOLD" ->
            revokeAccess(token)
        "SUBSCRIPTION_STATE_PAUSED" ->
            revokeAccess(token)
        "SUBSCRIPTION_STATE_CANCELED" ->
            scheduleRevocation(token, subscription)
        "SUBSCRIPTION_STATE_EXPIRED" ->
            revokeAccess(token)
        "SUBSCRIPTION_STATE_PENDING" ->
            holdAccess(token)
    }
}
```

The important principle here is: always read the subscription state from the API response rather than inferring it from the notification type. Notification types tell you what happened, but the subscription state tells you what the current reality is. These can diverge if notifications arrive out of order or if your server processes them with a delay.

For the `CANCELED` state specifically, you should not revoke access immediately. Instead, read the `expiryTime` and schedule revocation for that time:

```
fun scheduleRevocation(
    token: String,
    subscription: SubscriptionPurchaseV2
) {
    val expiryMillis = Instant.parse(
        subscription.lineItems.first().expiryTime
    ).toEpochMilliseconds()
    val now = System.currentTimeMillis()

    if (now >= expiryMillis) {
        revokeAccess(token)
    } else {
        database.setExpiry(token, expiryMillis)
        scheduler.scheduleAt(expiryMillis) {
            revokeAccess(token)
        }
    }
}
```

Common Pitfalls

Before wrapping up, here are the mistakes that cause the most problems in production:

Treating CANCELED as immediate revocation. This is the number one subscription state bug. A canceled subscription still has access until `expiryTime`. If you revoke immediately on receiving a `SUBSCRIPTION_CANCELED` RTDN, you are taking away access the user paid for.

Relying solely on queryPurchasesAsync() for entitlement. The client side API does not give you the full picture. `ON_HOLD` subscriptions appear with `PurchaseState.PURCHASED`, which looks like they should have access. Always verify with your backend.

Ignoring IN_GRACE_PERIOD. Some developers treat any payment failure as immediate revocation. If you have grace periods configured, you should continue granting access during this state. Otherwise, you are providing a worse experience than what Google intends.

Not handling PAUSED. If you enable pause in the Play Console but your app does not handle the `PAUSED` state, users who pause will see confusing behavior. Either handle the state properly or do not enable the feature.

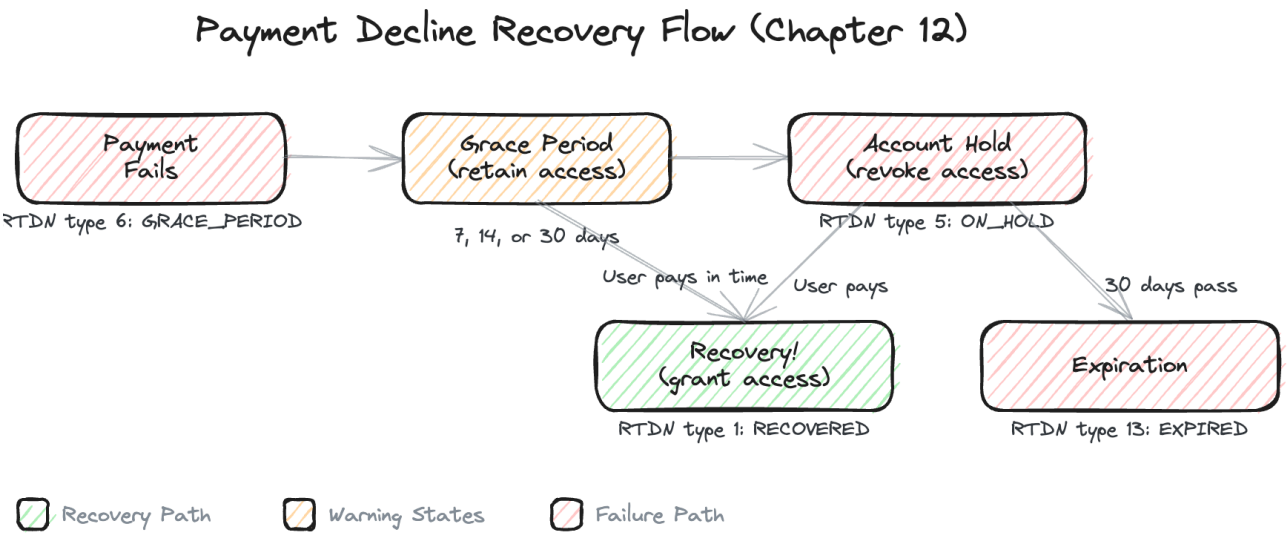
Calculating renewal dates manually. As covered in the renewal date section, calendar math is tricky. Always use `expiryTime` from the API.

Chapter 12: Payment Recovery: Grace Period and Account Hold

Subscriptions fail to renew. Credit cards expire, bank accounts run low, and payment methods get declined. When this happens, Google Play does not immediately cancel the subscription. Instead, it runs a multi stage recovery process designed to give the user time to fix the problem and keep their subscription alive.

This recovery process has two stages: grace period and account hold. Understanding how each stage works, what notifications you receive, and how to communicate with users during these stages directly affects your subscription retention rate. A well implemented recovery flow can save a significant percentage of subscriptions that would otherwise churn due to payment issues.

This chapter covers both stages of payment recovery, the Real Time Developer Notifications (RTDNs) you receive at each transition, what access the user should have during each stage, and how to use the In App Messaging API to help users fix payment problems without leaving your app.



Grace Period: What It Is and How It Works

A grace period is a window of time after a renewal payment fails during which Google continues to retry the payment. During this window, the user keeps full access to their subscription. The idea is simple: most payment failures are temporary. A card might be over its daily limit, a bank might flag an unusual charge, or the user's payment method might need updating. Giving Google a few days to retry often resolves the issue without the user ever noticing.

You configure grace periods in the Google Play Console under **Monetize > Subscriptions > [Your Subscription] > Grace period settings**. You can set the grace period duration per base plan. Google offers preset durations of 3, 7, 14, or 30 days for monthly and longer billing periods. For weekly subscriptions, the options are shorter.

The default behavior depends on when you created your subscription. Newer subscriptions created in the Play Console may have grace period enabled by default. Older subscriptions may have it disabled. You should

155

explicitly check and configure this for every subscription in your app.

During the grace period, Google retries the payment on its own schedule. You do not control the retry timing. Google optimizes retry attempts based on signals like the time of day, payment method type, and historical success rates. Your only job is to keep the user's access active and, ideally, let the user know there is a problem so they can fix it proactively.

The Silent Grace Period

Even if you set your grace period to 0 days in the Play Console, Google still applies a minimum grace period of approximately 1 day. This is sometimes called the "silent" grace period because it happens regardless of your configuration.

During this silent grace period, Google retries the payment at least once before taking any further action. The subscription status in the Google Play Developer API will show that the subscription is still active. You will not receive a `SUBSCRIPTION_IN_GRACE_PERIOD` RTDN during this silent window because, from Google's perspective, the subscription has not formally entered the grace period you configured.

This means you cannot fully eliminate the grace period. Even with a 0 day configuration, there is a brief window where the subscription appears active despite a failed payment. In practice, this is a good thing. It prevents subscriptions from being immediately disrupted by transient payment issues that resolve within hours.

If you have configured a grace period longer than 0 days, the silent grace period is included in that duration. A 7 day grace period means 7 total days of payment retry, not 7 days plus the silent period.

SUBSCRIPTION_IN_GRACE_PERIOD RTDN

When a subscription enters the grace period you configured (not the silent minimum), Google sends a Real Time Developer Notification with the notification type `SUBSCRIPTION_IN_GRACE_PERIOD`. This notification tells your backend that the user's payment failed and Google is actively retrying.

Your backend should handle this notification by:

1. Looking up the subscription using the purchase token from the notification.
2. Calling the Google Play Developer API to get the current `SubscriptionPurchaseV2` resource.
3. Checking the `lineItems` for the subscription state, which will show the subscription in a grace period state.
4. Flagging the user's account internally so your app can show appropriate messaging.
5. Keeping the user's entitlement active. Do not revoke access during grace period.

The `SubscriptionPurchaseV2` resource will have the subscription state set to `SUBSCRIPTION_STATE_IN_GRACE_PERIOD`. The `expiryTime` field reflects the end of the grace period, not the original renewal date. This gives you a clear deadline: if payment is not recovered by this time, the subscription moves to account hold or cancels.

User Access During Grace Period

During the grace period, the user should retain full access to all subscription features. This is not optional. Google's guidelines require that you maintain the user's entitlement during the grace period because Google is still actively trying to collect payment. From the user's perspective, their subscription is still active.

If you revoke access during the grace period, you create a negative experience for users whose payment issue is temporary. Many of these users would have renewed successfully if given a few more days. Cutting them off prematurely drives unnecessary churn.

However, you should use this time to communicate with the user about the payment issue. A subtle banner in your app, a push notification, or an email letting them know their payment failed and suggesting they update their payment method can dramatically improve recovery rates. More on this in the communication section later in this chapter.

Detecting Grace Period and Showing UI

The client side `Purchase` object does not expose grace period state directly. A subscription in grace period still has `isAutoRenewing = true` (Google is still retrying payment) and `purchaseState == PURCHASED`, which is indistinguishable from a healthy active subscription when queried via `queryPurchasesAsync()`. Checking `!purchase.isAutoRenewing` would match canceled subscriptions, not grace period ones, a common mistake that leads to false negatives.

The reliable way to detect grace period status is through your backend. When your server receives the `SUBSCRIPTION_IN_GRACE_PERIOD` RTDN, it flags the user's account. Your app then checks this flag when it syncs with your server:

```
// PBL 8.x
fun showGracePeriodBanner(
    subscriptionStatus: SubscriptionStatus
) {
    if (subscriptionStatus.isInGracePeriod) {
        // Show a non blocking banner
        showBanner(
            message = "There's a problem with your " +
                "payment method. Please update it " +
                "to keep your subscription.",
            action = "Fix Payment",
            onClick = { launchPaymentUpdate() }
        )
    }
}
```

The banner should be visible but not intrusive. You want the user to notice it and take action, but you do not want to degrade their experience while they still have access. A dismissible banner at the top of your main screen works well for this.

In App Messaging API: showInAppMessages()

Google provides a built in way to prompt users to fix payment issues directly inside your app. The `BillingClient.showInAppMessages()` method displays a Google managed dialog that walks the user through resolving their payment problem. This is the simplest way to handle payment recovery UI because Google handles all the messaging, payment method selection, and retry logic.

To use it, call `showInAppMessages()` with the `TRANSACTIONAL` category. This category specifically targets payment related messages, including grace period and account hold notifications:

```
// PBL 8.x
fun showPaymentRecoveryMessage(activity: Activity) {
    val params = InAppMessageParams.newBuilder()
        .addInAppMessageCategoryToShow(
            InAppMessageParams
                .InAppMessageCategoryId
                .TRANSACTIONAL
        )
        .build()

    billingClient.showInAppMessages(
        activity, params
    ) { result ->
        val code = result.responseCode
        if (code == InAppMessageResult
            .InAppMessageResponseCode
            .SUBSCRIPTION_STATUS_UPDATED
        ) {
            // User fixed their payment. Refresh.
            refreshSubscriptionStatus()
        }
    }
}
```

Call `showInAppMessages()` at natural points in your app, such as when the app launches, when the user navigates to a subscription feature, or when your backend indicates the user is in grace period or account hold. If there is no pending payment issue, the method returns `NO_ACTION_NEEDED` and shows nothing to the user. If there is an issue, Google displays the appropriate dialog automatically.

When the user successfully resolves their payment through this dialog, you receive `SUBSCRIPTION_STATUS_UPDATED` in the callback. At that point, refresh the user's subscription status from your backend because the subscription has been recovered.

A few things to keep in mind about `showInAppMessages()`:

- You must pass a valid `Activity` reference. The dialog is presented as an overlay on the activity.
- The `TRANSACTIONAL` category covers both grace period and account hold scenarios.
- Google controls the content and appearance of the dialog. You cannot customize the messaging.
- This is not a replacement for your own notifications. Use it as one part of a broader communication strategy.

Account Hold: When Grace Period Expires

If the grace period ends without a successful payment, the subscription enters account hold. Account hold is the second and more aggressive stage of payment recovery. Google continues to retry the payment, but the user's access is now revoked.

Account hold can last up to 30 days, depending on your configuration in the Play Console. You configure the account hold duration alongside the grace period settings. If you do not enable account hold, the subscription cancels immediately after the grace period expires.

Enabling account hold is strongly recommended. It gives Google additional time to recover the payment and gives the user additional time to fix their payment method. A subscription that enters account hold still has a meaningful chance of being recovered, especially if you communicate effectively with the user.

The account hold period starts the moment the grace period expires. If you had a 7 day grace period and a 30 day account hold, the total recovery window is 37 days from the initial payment failure. That is over a month for the user to update their payment method and restore their subscription.

SUBSCRIPTION_ON_HOLD RTDN

When a subscription transitions from grace period to account hold, Google sends a `SUBSCRIPTION_ON_HOLD` RTDN. This is your signal to revoke the user's access to subscription features.

Your backend should handle this notification by:

1. Looking up the subscription using the purchase token.
2. Calling the Google Play Developer API to confirm the subscription state is `SUBSCRIPTION_STATE_ON_HOLD`.
3. Revoking the user's entitlement immediately.
4. Flagging the user's account so your app can show account hold messaging.
5. Triggering a push notification or email to the user explaining that their subscription has been suspended.

Here is the key difference from grace period handling: during account hold, you must revoke access. The user should not be able to use premium features. However, you should still make it easy for them to restore their subscription. Show a clear message explaining what happened and provide a direct path to fixing their payment method.

Your app should check for account hold status and display appropriate UI:

```
// PBL 8.x
fun handleSubscriptionState(
    subscriptionStatus: SubscriptionStatus
) {
    when {
        subscriptionStatus.isOnHold -> {
            // Revoke access, show recovery UI
            revokeEntitlement()
            showAccountHoldScreen(
                message = "Your subscription is " +
                    "paused due to a payment issue.",
                action = "Update Payment Method",
                onClick = {
                    launchPaymentUpdate()
                }
            )
        }
        subscriptionStatus.isInGracePeriod -> {
            // Keep access, show warning
            showGracePeriodBanner(subscriptionStatus)
        }
        subscriptionStatus.isActive -> {
            grantEntitlement()
        }
    }
}
```

When a user is in account hold and opens your app, do not just show a blank screen or silently remove features. Show a dedicated screen that explains the situation and gives the user a clear call to action. The goal is to get them to fix their payment method, not to punish them.

Recovery: SUBSCRIPTION_RECOVERED

When Google successfully collects a payment during either the grace period or account hold, the subscription is recovered. Google sends a `SUBSCRIPTION_RECOVERED` RTDN to your backend. This is the notification you want to see because it means the subscription is back to normal.

Your backend should handle this notification by:

1. Looking up the subscription using the purchase token.
2. Calling the Google Play Developer API to confirm the subscription state is `SUBSCRIPTION_STATE_ACTIVE`.
3. Restoring the user's entitlement immediately.
4. Clearing any grace period or account hold flags on the user's account.
5. Optionally sending the user a confirmation that their subscription has been restored.

Here is a backend handler that processes the recovery notification:

```
// Server-side: handling SUBSCRIPTION_RECOVERED
fun handleSubscriptionRecovered(
    purchaseToken: String,
    packageName: String
) {
    val subscription = playApi
        .getSubscriptionV2(packageName, purchaseToken)
    if (subscription.subscriptionState !=
        "SUBSCRIPTION_STATE_ACTIVE") return

    val record = database
        .findByPurchaseToken(purchaseToken)
        ?: return

    record.status = "ACTIVE"
    record.gracePeriod = false
    record.onHold = false
    record.renewalDate = subscription.lineItems
        .firstOrNull()?.expiryTime
    database.update(record)

    notificationService.send(
        userId = record.userId,
        message = "Your subscription is restored."
    )
}
```

An important detail about recovery: the billing date resets. When a subscription recovers from grace period or account hold, Google sets the next billing date based on the recovery date, not the original renewal date. If a user's subscription was supposed to renew on March 1, failed, entered a 7 day grace period, and recovered on March 5, the next renewal date will be approximately April 5, not April 1. The user gets a full billing period from the point of recovery.

This billing date reset means you should update the renewal date in your database when you process the `SUBSCRIPTION_RECOVERED` notification. If you rely on a cached renewal date from before the payment failure, it will be wrong.

Failure: Auto Cancel After Account Hold

If the account hold period expires without a successful payment, Google automatically cancels the subscription. Your backend receives a `SUBSCRIPTION_CANCELED` RTDN (or `SUBSCRIPTION_EXPIRED`, depending on the specific flow).

At this point, the recovery process is over. The subscription is no longer active, and the user has lost their subscription. They would need to purchase a new subscription to regain access.

The total timeline from initial payment failure to cancellation is:

1. Silent grace period: approximately 1 day (always applies).
2. Configured grace period: 0 to 30 days (your setting).
3. Account hold: 0 to 30 days (your setting).

If you configured a 7 day grace period and a 30 day account hold, the subscription survives up to 38 days of payment failure before canceling. If you configured neither, the subscription cancels after the silent grace period of approximately 1 day.

For most apps, enabling both grace period (7 or 14 days) and account hold (30 days) is the right choice. The additional retention from recovering subscriptions during these windows far outweighs the small operational cost of managing the states.

When the subscription finally cancels after account hold, handle it the same way you handle any other cancellation: revoke access, update your database, and optionally send the user a final notification with a link to resubscribe.

Communicating with Users During Payment Recovery

The difference between recovering a subscription and losing it often comes down to whether the user knows there is a problem. Many users have no idea their payment failed. They do not check their email from Google Play, and they do not notice the subtle "payment issue" notification in their device's notification drawer. Your app is the best channel to reach them.

Here is a layered communication strategy:

Push notifications: Send a push notification as soon as your backend receives the `SUBSCRIPTION_IN_GRACE_PERIOD` RTDN. Keep the message clear and actionable: "Your subscription payment could not be processed. Please update your payment method to avoid losing access." Include a deep link that takes the user directly to the Play Store's subscription management page or to your app's payment recovery screen.

Email: If you have the user's email address, send an email explaining the payment issue. Email works well because it persists. The user can come back to it later. Include clear instructions on how to update their payment method.

In app messaging: Use `showInAppMessages()` with the `TRANSACTIONAL` category as described earlier. Also consider building your own in app UI that appears when the user opens your app during grace period or account hold. Your own UI gives you control over the messaging and can be more specific about what the user will lose if they do not act.

Escalation during account hold: When the subscription moves from grace period to account hold, increase the urgency of your messaging. During grace period, a gentle banner works. During account hold, the user has lost access, so a full screen message explaining the situation and how to fix it is appropriate.

Timing matters: Do not send all your communications at once. Space them out. A push notification on day 1, an email on day 3, another push notification on day 7 (when account hold starts if your grace period is 7 days), and a final email partway through account hold. This cadence keeps the issue on the user's radar without being overwhelming.

Deep links to the Play Store: You can open the Play Store's subscription management page where the user can update their payment method:

```
// PBL 8.x
fun openSubscriptionManagement(
    context: Context,
    packageName: String
) {
    val intent = Intent(Intent.ACTION_VIEW).apply {
        data = Uri.parse(
            "https://play.google.com/store/" +
                "account/subscriptions?package=" +
                packageName
        )
    }
    context.startActivity(intent)
}
```

This takes the user directly to their subscription settings in the Play Store, where they can update their payment method with just a few taps.

Changing Grace Period Duration for Existing Subscribers

If you change the grace period duration in the Play Console, the change affects existing subscribers differently depending on their current state.

163

Subscribers not currently in grace period: The new duration applies to their next payment failure. If a subscriber is currently active and renewing normally, they will get the new grace period duration if and when their next payment fails.

Subscribers currently in grace period: This is where it gets nuanced. If you shorten the grace period, subscribers who are already in grace period and whose current grace period window extends beyond the new shorter duration may have their grace period cut short. The subscription could transition to account hold (or cancel) sooner than originally expected.

If you extend the grace period, subscribers currently in grace period generally benefit from the longer window. Google extends their recovery window to match the new duration.

The practical advice: If you are going to change grace period duration, prefer extending it rather than shortening it. Extending the grace period is always safe for existing subscribers. Shortening it can cause unexpected transitions for subscribers who are actively in recovery.

If you must shorten the grace period, do it during a period when you expect fewer subscribers to be in grace period (if your analytics can tell you this). And monitor your `SUBSCRIPTION_ON_HOLD` and `SUBSCRIPTION_CANCELED` RTDN volume after the change to make sure you have not caused a spike in unexpected cancellations.

The same general principles apply to account hold duration changes: extending is safe, shortening can cause early cancellations for subscribers already in account hold.

Putting It All Together

Here is the complete lifecycle of a subscription that encounters a payment failure and eventually recovers:

1. **Renewal attempt fails**: Google's initial charge attempt is declined. The silent grace period begins. The subscription still appears active.
2. **Grace period starts**: After the silent period, the configured grace period begins. Google sends `SUBSCRIPTION_IN_GRACE_PERIOD` RTDN. Your backend flags the user. Your app shows a payment warning banner. The user retains full access.
3. **Recovery during grace period**: If Google collects payment during this window, it sends `SUBSCRIPTION_RECOVERED`. Your backend restores the normal status, updates the billing date, and clears the flag. The user may never have noticed the issue.
4. **Grace period expires**: If payment is not recovered, the subscription moves to account hold. Google sends `SUBSCRIPTION_ON_HOLD` RTDN. Your backend revokes the entitlement. Your app shows an account hold screen when the user opens it.
5. **Recovery during account hold**: If the user updates their payment method and Google collects payment, it sends `SUBSCRIPTION_RECOVERED`. Your backend restores access and updates the billing date. The user is back to normal.

6. **Account hold expires**: If the entire account hold period passes without recovery, Google cancels the subscription. Your backend receives `SUBSCRIPTION_EXPIRED` or `SUBSCRIPTION_CANCELED`. The subscription is over.

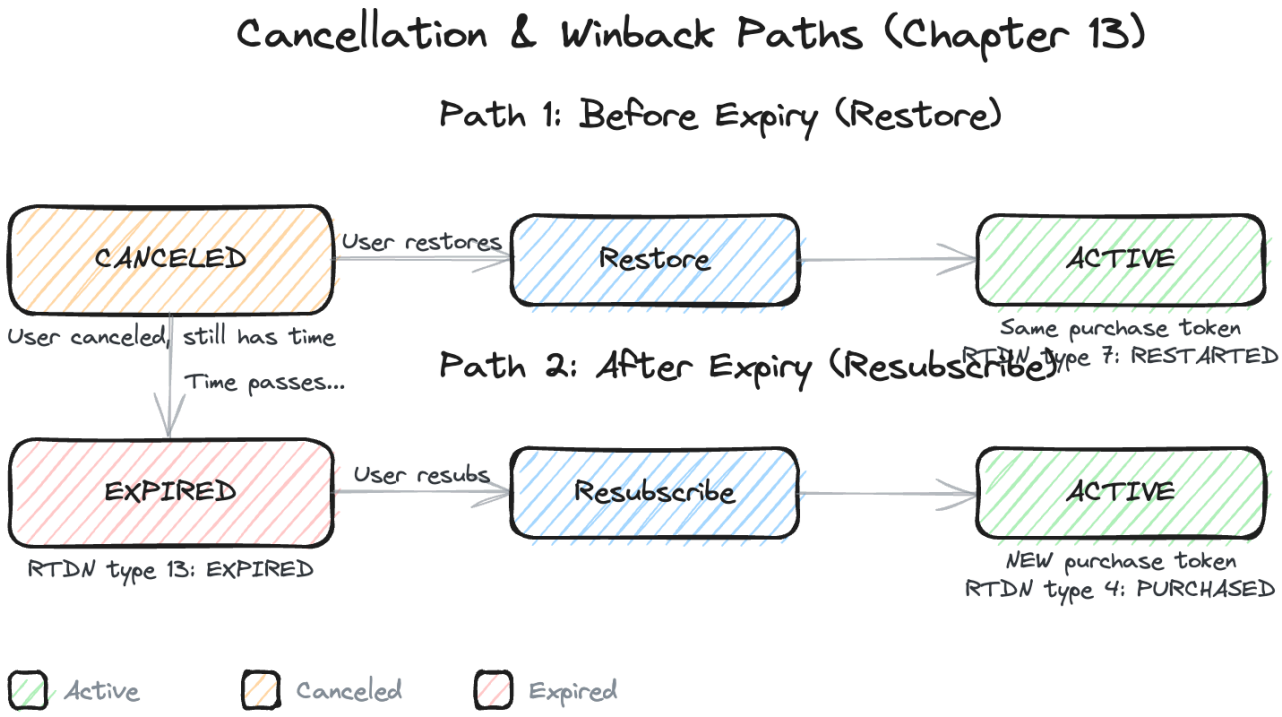
At each stage, your app should communicate clearly with the user about what is happening and what they need to do. The combination of Google's automatic payment retries, the In App Messaging API, and your own push notifications and in app UI gives you the best chance of recovering failed subscriptions.

165

Chapter 13: Cancellations, Pauses, and Winback

Subscriptions do not last forever. Users cancel for all kinds of reasons: they no longer need the feature, they want to save money, or they simply forgot what your app does. Sometimes the system cancels on their behalf after repeated payment failures. And sometimes you, the developer, need to cancel a subscription through your own backend.

This chapter covers every way a subscription can end or be interrupted, what happens to user access in each case, and the tools Google Play gives you to win subscribers back. You will learn how to handle cancellations, pauses, restores, resubscribes, deferrals, and revocations with the confidence that your entitlement logic stays correct throughout.



User Initiated Cancellation via Play Subscriptions Center

The most common cancellation path is the user doing it themselves. Users manage their subscriptions through the Google Play Store app under **Settings > Payments & subscriptions > Subscriptions**. They find your subscription, tap it, and tap "Cancel subscription." Google walks them through a brief cancellation flow that may include a survey asking why they are leaving.

From your app's perspective, this cancellation is silent. Your app does not receive a callback or in app notification at the moment the user cancels. Instead, you learn about it through two channels:

1. **Real Time Developer Notifications (RTDNs)**: Google sends a `SUBSCRIPTION_CANCELED` notification to your backend. This is your primary signal.
2. **Polling the API**: When you call `purchases.subscriptionsv2.get` for the subscription, the response includes a `canceledStateContext` field that tells you the subscription has been canceled and why.

166

The important thing to understand is that cancellation does not mean immediate loss of access. When a user cancels an auto renewing subscription, they retain full access until the end of their current billing period. The `expiryTime` field in the subscription resource tells you exactly when access ends. Until that time passes, the user is still a paying subscriber who deserves full access.

Your app should reflect this clearly. If a user cancels and then opens your app the next day, they should still see all premium features. You might show a banner saying "Your subscription ends on March 15" to remind them, but do not revoke anything early.

Developer Initiated Cancellation via API

Sometimes you need to cancel a subscription from your backend. A user might contact your support team asking to cancel, or your business logic might require canceling a subscription as part of a fraud mitigation workflow.

You cancel a subscription server side by calling `purchases.subscriptionsv2.cancel` on the Google Play Developer API. This endpoint requires the package name, the subscription's purchase token, and a cancellation reason.

Google provides two cancellation reasons for developer initiated cancellations:

USER_REQUESTED_STOP_RENEWAL: Use this when the user asked you to cancel on their behalf. Perhaps they contacted your support team, or your app provides its own cancellation button that calls your backend. The behavior is identical to the user canceling through the Play subscriptions center: the subscription stops renewing, but the user keeps access until `expiryTime`.

DEVELOPER_REQUESTED_STOP_PAYMENTS: Use this when you are canceling for your own reasons, not at the user's request. This might happen if you detect fraudulent activity, if the user violated your terms of service, or if you are winding down a product. The behavior is the same as the other reason in terms of access, but the `canceledStateContext` in the subscription resource records a different cancellation reason. This distinction matters for analytics and for understanding your churn patterns.

In both cases, the subscription does not renew at the next billing cycle. The user retains access until `expiryTime`. If you need to revoke access immediately, you need the separate `revoke` endpoint, which is covered later in this chapter.

System Initiated Cancellation

The third way a subscription gets canceled is when Google's payment system gives up trying to collect payment. This happens after the grace period and account hold mechanisms have been exhausted.

The sequence works like this:

1. A renewal payment fails (e.g., expired credit card, insufficient funds).
2. If you have configured a grace period, Google retries the payment for the grace period duration (typically 3, 7, 14, or 30 days). The user keeps full access during grace period.

167

3. If the grace period expires without successful payment, the subscription enters account hold. The user loses access, but the subscription is not yet canceled.
4. Google continues retrying the payment during account hold (up to 30 days).
5. If account hold expires without successful payment, Google cancels the subscription. The subscription is now terminated.

When the system cancels a subscription this way, Google sends a `SUBSCRIPTION_CANCELED` RTDN to your backend. The `canceledStateContext` will indicate a system initiated cancellation due to payment failure.

Your backend should update the user's entitlement status to reflect the cancellation. If the user was already in account hold (no access), this is a final state change from "on hold, might recover" to "canceled, will not recover."

canceledStateContext: Understanding Why a Subscription Was Canceled

When you retrieve a canceled subscription through `purchases.subscriptionsv2.get`, the response includes a `canceledStateContext` object. This tells you exactly why the subscription was canceled, which is valuable for analytics, customer support, and deciding how aggressively to pursue win back.

The `canceledStateContext` contains a `cancellationReason` field with one of the following values:

REASON	WHAT IT MEANS
USER_CANCELED	The user canceled through the Play Store or your app
SYSTEM_CANCELED	Payment failure after grace period and account hold
DEVELOPER_CANCELED	You canceled via the API
REPLACED	The subscription was replaced by a plan change (new token issued)

For user and developer cancellations, there is additional context. A user cancellation includes the survey response if the user filled one out. A developer cancellation records whether it was `USER_REQUESTED_STOP_RENEWAL` or `DEVELOPER_REQUESTED_STOP_PAYMENTS`.

This information helps you segment your churned users. A user who canceled because "it's too expensive" is a good candidate for a discount win back offer. A user whose payment failed might come back if you prompt them to update their payment method. A user you canceled for fraud should not get a win back offer at all.

```
// Server side: inspecting cancellation reason
fun logCancellationReason(
    subscription: SubscriptionPurchaseV2
) {
    val context = subscription.canceledStateContext
        ?: return

    when (context.cancellationReason) {
        "USER_CANCELED" -> {
            val survey = context.userInitiatedCancellation
                ?.cancelSurveyResult
            analytics.trackChurn(
                reason = "user",
                surveyResponse = survey?.reason
            )
        }
        "SYSTEM_CANCELED" -> {
            analytics.trackChurn(reason = "payment_failure")
        }
        "DEVELOPER_CANCELED" -> {
            analytics.trackChurn(reason = "developer")
        }
    }
}
```

User Access After Cancellation: Until expiryTime

Regardless of how a subscription gets canceled, the user retains access until the `expiryTime` in the subscription resource. This is a fundamental rule of Google Play subscriptions.

When a user cancels on day 5 of a 30 day billing period, they have already paid for the full 30 days. Google does not issue a prorated refund, and the user keeps access for the remaining 25 days. Your app must respect this.

On your backend, the entitlement check should look something like this:

169

```
fun hasActiveSubscription(
    subscription: SubscriptionPurchaseV2
): Boolean {
    val now = System.currentTimeMillis()
    val expiry = subscription.lineItems
        .firstOrNull()
        ?.expiryTime
        ?.toEpochMilliseconds() ?: return false

    // User has access if expiry is in the future,
    // even if subscription is canceled
    return now < expiry
}
```

This means your entitlement logic should not check cancellation status alone. A canceled subscription with a future `expiryTime` is still an active subscription from the user's perspective. Only when `expiryTime` passes does the user lose access.

On the client side, `queryPurchasesAsync()` returns canceled subscriptions that have not yet expired. The `Purchase` object still appears in the results with a valid purchase token. Once `expiryTime` passes, the purchase disappears from `queryPurchasesAsync()` results.

Installment Subscription Cancellation

Installment subscriptions have special cancellation rules because of the commitment period. When a user subscribes to an installment plan, they agree to a minimum number of payments. Canceling before fulfilling that commitment is not straightforward.

If a user attempts to cancel during the commitment period, one of two things happens depending on the market and the plan configuration:

1. **Cancellation is blocked**: The user cannot cancel until the commitment period ends. Google Play shows a message explaining that they have remaining committed payments.
2. **Cancellation is scheduled**: Google accepts the cancellation request but schedules it to take effect after the commitment period ends. The user continues to be charged for the remaining committed payments.

When a cancellation is scheduled for the end of the commitment period, Google sends a `SUBSCRIPTION_CANCELLATION_SCHEDULED` RTDN. This is different from the standard `SUBSCRIPTION_CANCELED` notification. The scheduled cancellation means the subscription will continue to renew through the remaining commitment payments and then stop.

Your backend should handle `SUBSCRIPTION_CANCELLATION_SCHEDULED` by noting that the user intends to leave but is still an active, paying subscriber. Do not revoke access. Do not remove them from subscriber

170

segments. The actual cancellation happens later, and you will receive a `SUBSCRIPTION_CANCELED` notification at that point.

After the commitment period ends and the subscription transitions to month to month renewal, the user can cancel freely, just like any other auto renewing subscription.

Pausing Subscriptions

Sometimes users do not want to cancel permanently. They just need a break. Google Play supports subscription pausing, which lets users temporarily stop their subscription and resume it later.

Pausing is different from canceling in an important way: a paused subscription is expected to resume. The user has signaled intent to come back. This makes paused subscribers a fundamentally different cohort from canceled subscribers in your analytics.

Duration Options by Billing Period

The pause duration depends on the subscription's billing period:

BILLING PERIOD	AVAILABLE PAUSE DURATIONS
Weekly	1, 2, or 3 weeks
Monthly	1, 2, or 3 months
Quarterly or longer	Not eligible for pausing

The user selects their preferred pause duration through the Play Store's subscription management screen. You cannot programmatically pause a subscription from your app or backend. Pausing is always user initiated through the Play Store.

To enable pausing for your subscription, you must opt in through the Play Console. Go to your subscription settings and enable the pause option. You can configure the maximum pause duration allowed.

The Pause Lifecycle: RTDN Sequence

When a user pauses a subscription, your backend receives a sequence of RTDNs that tracks the entire lifecycle:

SUBSCRIPTION_PAUSE_SCHEDULE_CHANGED: This fires immediately when the user schedules the pause. At this point, the subscription is still active. The pause has not started yet. It will begin at the end of the current billing period. Think of this as the user saying "pause my subscription at the end of this month."

SUBSCRIPTION_PAUSED: This fires when the pause actually begins, which is at the end of the current billing period. At this point, the user loses access. No payment is collected. The subscription enters a dormant state.

SUBSCRIPTION_RECOVERED: This fires when the pause ends and the subscription resumes. Google charges the user for the new billing period, and the user regains access.

Here is how to handle each notification:

```
fun handlePauseNotification(
    notificationType: Int,
    purchaseToken: String
) {
    when (notificationType) {
        11 -> { // SUBSCRIPTION_PAUSE_SCHEDULE_CHANGED
            // User scheduled a pause. Subscription
            // is still active. Optionally show
            // "pausing soon" in your UI.
            markPauseScheduled(purchaseToken)
        }
        10 -> { // SUBSCRIPTION_PAUSED
            // Pause has started. Revoke access.
            revokeAccess(purchaseToken)
        }
        1 -> { // SUBSCRIPTION_RECOVERED
            // Pause ended, payment collected.
            // Restore access.
            grantAccess(purchaseToken)
        }
    }
}
```

Between `SUBSCRIPTION_PAUSED` and `SUBSCRIPTION_RECOVERED`, the user should not have access to premium features. Your entitlement logic must account for the paused state.

Querying Paused Subscriptions with includeSuspendedSubscriptions

By default, `queryPurchasesAsync()` on the client side does not return paused (suspended) subscriptions. This means if a user pauses their subscription and opens your app, the purchase does not appear in the query results, and your app has no local signal that the user was ever a subscriber.

Starting with PBL 8.1, you can include suspended subscriptions in your query by using the `includeSuspendedSubscriptions` option:

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

val result = billingClient
    .queryPurchasesAsync(params)

for (purchase in result.purchasesList) {
    if (purchase.purchaseState ==
        Purchase.PurchaseState.PURCHASED
    ) {
        // This is a paused/suspended subscription
        showPausedSubscriptionUI(purchase)
    }
}
```

This is useful for showing a "your subscription is paused" message in your app, or for presenting a "resume now" option. Without this flag, you would need to rely entirely on your backend to know whether a user has a paused subscription.

When a suspended subscription appears in the query results, its `purchaseState` will be `PURCHASED`, the subscription was successfully purchased, it is just currently paused. Do not confuse this with `PENDING`, which is for purchases where payment has not yet completed (such as cash payments). You should not grant access for paused purchases. Use the presence of the purchase in results (with `includeSuspendedSubscriptions = true`) to drive UI that acknowledges the user's paused subscription and encourages them to resume.

Restoring a Canceled Subscription Before Expiration

If a user cancels their subscription but changes their mind before `expiryTime`, they can restore it. Restoration reactivates the auto renewal so the subscription continues beyond the current billing period.

The user restores their subscription through the Play Store's subscription management screen. There is a "Resubscribe" or "Restore" button that appears for canceled subscriptions that have not yet expired.

When a user restores a canceled subscription, two important things happen:

1. **The purchase token stays the same**. Unlike a plan change, restoration does not create a new token. The existing token simply becomes active again.
2. **Google sends a `SUBSCRIPTION_RESTARTED` RTDN** to your backend.

Because the token does not change, your backend logic is straightforward. When you receive `SUBSCRIPTION_RESTARTED`, look up the existing subscription record by purchase token and flip it back to active:

```
fun handleSubscriptionRestarted(
    purchaseToken: String
) {
    val record = database
        .findByPurchaseToken(purchaseToken)
        ?: return

    record.canceledAt = null
    record.status = "ACTIVE"
    database.update(record)
}
```

On the client side, `queryPurchasesAsync()` continues to return this purchase (it was already returned before restoration, since the subscription had not yet expired). The user's access is uninterrupted. The only difference is that the subscription will now renew at the end of the billing period instead of expiring.

This is the simplest recovery path. No new tokens, no re verification, no changes to entitlement logic. The user picks up right where they left off.

Resubscribing After Expiration

If a subscription has already expired (past `expiryTime`), the user cannot restore it. Instead, they must resubscribe, which creates an entirely new subscription with a new purchase token.

A user can resubscribe through two paths:

1. **Through your app**: You present subscription options and the user goes through the standard purchase flow. This is no different from a new subscription purchase.
2. **Through the Play Store**: Google may show resubscribe options in the Play Store's subscription management screen. When the user resubscribes this way, the purchase happens outside your app.

When a resubscribe happens outside your app, the purchase context is captured in the `outOfAppPurchaseContext` field of the subscription resource. This field tells your backend that the subscription was purchased through the Play Store directly, not through your app's billing flow. Your `PurchasesUpdatedListener` will not fire for these purchases. You learn about them through RTDNs or by polling the API.

Because resubscribing creates a new purchase token, your backend treats it as a new subscription. There is no `linkedPurchaseToken` connecting the new subscription to the old one (unlike plan changes). This makes it harder to associate the resubscribe with the user's previous subscription history.

Linking Resubscribes to Original Accounts

When a user resubscribes after expiration, you want to connect the new subscription to their existing account. Google provides the `expiredExternalAccountIdentifiers` field to help with this.

If you passed an `externalAccountId` when the user originally subscribed (using `BillingFlowParams.Builder.setExternalAccountId()`), that identifier is stored with the subscription. When the subscription expires and the user resubscribes through the Play Store, the `expiredExternalAccountIdentifiers` field on the new subscription resource contains the external account IDs from the user's expired subscriptions.

This lets your backend map the new subscription back to the correct user account:

```
fun handleResubscribe(
    newSubscription: SubscriptionPurchaseV2
) {
    val expiredIds = newSubscription
        .expiredExternalAccountIdentifiers

    if (expiredIds != null) {
        // Find the user by their old account ID
        val user = database.findByExternalAccountId(
            expiredIds.externalAccountId
        )
        if (user != null) {
            database.addSubscription(
                userId = user.id,
                purchaseToken = newSubscription
                    .purchaseToken,
                status = "ACTIVE"
            )
            return
        }
    }

    // No match found. Create a new record
    // and handle account linking separately.
    handleNewSubscription(newSubscription)
}
```

The key prerequisite is that you must set `externalAccountId` during the original purchase. If you did not, this field will be empty on the resubscribe, and you have no automatic way to link the new subscription to the old account. This is why setting `externalAccountId` is a best practice for every subscription purchase, even if you do not think you need it right now.

175

```
val flowParams = BillingFlowParams.newBuilder()
    .setProductDetailsParamsList(
        listOf(productDetailsParams)
    )
    .setExternalAccountId(userId)
    .build()
```

Deferred Billing: subscriptionsv2.defer

Deferred billing lets you extend a subscription's next renewal date without charging the user. This is useful for customer support scenarios where you want to give a user free time as compensation, or for promotional campaigns where you reward loyal subscribers with bonus time.

You call `purchases.subscriptionsv2.defer` on the Google Play Developer API with the purchase token and a new desired expiry time. The constraints are:

- **Minimum deferral**: 1 day beyond the current expiry time.
- **Maximum deferral**: 1 year beyond the current expiry time.
- The new expiry time must be in the future.

When you defer a subscription, the user keeps full access through the extended period. No payment is collected until the new expiry time arrives. The subscription then renews normally from the deferred date.

Here is a practical example. A subscriber contacts support because they experienced a bug that prevented them from using a premium feature for two weeks. You want to give them two free weeks as compensation:

176

```
// Server side: defer a subscription by 14 days
fun deferSubscription(
    packageName: String,
    purchaseToken: String
) {
    val currentExpiry = getSubscriptionExpiry(
        packageName, purchaseToken
    )
    val newExpiry = currentExpiry
        .plus(14, ChronoUnit.DAYS)

    playApi.purchases()
        .subscriptionsv2()
        .defer(
            packageName,
            purchaseToken,
            DeferRequest(newExpiry)
        )
        .execute()
}
```

A few things to keep in mind:

- Deferral does not change the subscription's purchase token. The same token remains valid.
- You can defer a subscription multiple times. Each deferral extends from the current (possibly already deferred) expiry time.
- Deferral does not work on canceled subscriptions. The subscription must be active (set to renew) for deferral to apply.
- Google sends an RTDN when a deferral is processed, so your backend can update its records.

Deferral is a powerful tool for customer retention. A well timed free extension can turn a frustrated subscriber into a loyal one.

Revocations: subscriptionsv2.revoke

Revocation is the most aggressive action you can take on a subscription. Unlike cancellation, which lets the user keep access until `expiryTime`, revocation removes access immediately.

You call `purchases.subscriptionsv2.revoke` on the Google Play Developer API. When you revoke a subscription:

- The user loses access immediately, regardless of how much time they had left in their billing period.
- Google issues a prorated refund for the unused portion of the billing period.

177

- The subscription is terminated. It cannot be restored.

Revocation is appropriate in limited circumstances:

- **Fraud**: You have confirmed that the subscription was purchased fraudulently.
- **Terms of service violations**: The user violated your terms in a way that warrants immediate removal.
- **Chargebacks**: In some cases, you may want to proactively revoke access when you detect a chargeback.

Do not use revocation as a substitute for cancellation. If a user contacts support asking to cancel and wants a refund, the appropriate path is to cancel the subscription (letting them keep access until `expiryTime`) and process a refund separately through the Play Console or the Voided Purchases API. Revocation is a blunt instrument that should be reserved for situations where immediate access removal is justified.

```
// Server side: revoke a subscription
fun revokeSubscription(
    packageName: String,
    purchaseToken: String,
    reason: String
) {
    playApi.purchases()
        .subscriptionsv2()
        .revoke(
            packageName,
            purchaseToken,
            RevokeRequest(reason)
        )
        .execute()

    // Immediately update your entitlement records
    val record = database
        .findByPurchaseToken(purchaseToken)
    record?.status = "REVOKED"
    record?.let { database.update(it) }

    analytics.track(
        "subscription_revoked",
        mapOf("reason" to reason)
    )
}
```

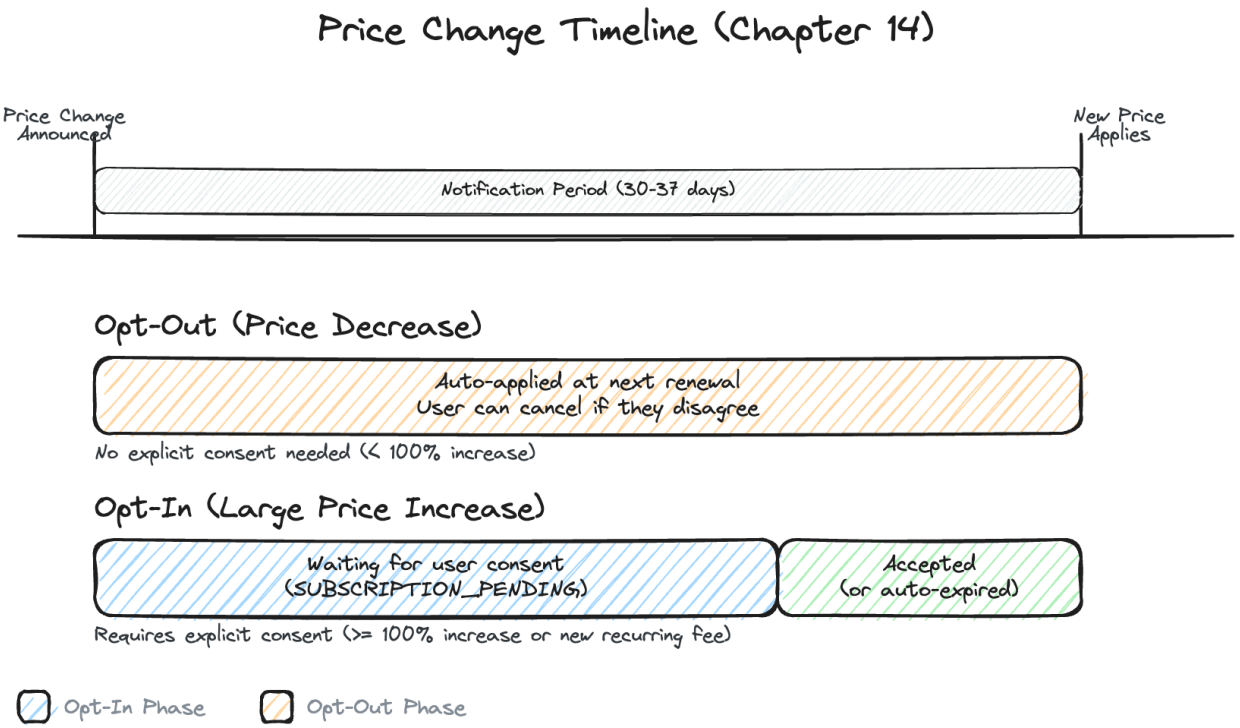
When you revoke a subscription, Google sends a `SUBSCRIPTION_REVOKED` RTDN. Your backend should handle this notification to ensure entitlement records are updated, even as a safety net in case your initial database update did not complete.

178

Chapter 14: Changing Subscription Prices

Pricing is never permanent. Market conditions shift, costs change, competitors adjust their pricing, and your own product evolves. At some point, you will need to change the price of a subscription that already has active subscribers. Google Play provides a system for this, but it is one of the most nuanced parts of the billing platform. Different rules apply depending on whether the price goes up or down, whether the subscriber is new or existing, and which country the subscriber lives in.

This chapter walks through every type of price change Google Play supports, the timelines and notification requirements for each, the APIs you use to execute them, and the edge cases you need to handle on your backend.



How Pricing Works: New Subscribers vs. Legacy Price Cohorts

When you change the price of a base plan, the new price takes effect for new purchases within a few hours. Any user who subscribes after the change sees and pays the updated price. No additional work is required on your part for new subscribers.

Existing subscribers are a different story. By default, they are completely unaffected by a price change. Google places them into a **legacy price cohort**, and they continue paying their original base plan price at every renewal. This is an intentional design decision. Users agreed to pay a specific price when they subscribed, and Google does not change that agreement without an explicit action from you.

This means that after a price change, you can have multiple groups of subscribers paying different amounts for the same base plan. A subscriber who joined in January at $4.99/month and a subscriber who joined in

179

June at $6.99/month both have the same entitlement, but they are in different price cohorts. Google supports up to 250 concurrent legacy price cohorts per base plan.

There are two important exceptions to this behavior:

1. **Offer pricing phases cannot be migrated.** If you change the price of a free trial or introductory offer, the change only affects new subscribers. You cannot push existing subscribers from one offer phase price to another.
2. **Installment subscriptions freeze prices during the commitment period.** If a subscriber committed to 12 monthly payments at $4.99, you cannot change their price until the commitment ends. Any price migration you initiate takes effect only after the commitment period concludes.

To move existing subscribers to the current price, you must explicitly end the legacy price cohort. This triggers Google's price change notification system, which varies depending on whether the new price is higher or lower.

Programmatic Price Changes via monetization.subscriptions.patch

You can change a base plan's price through the Play Console UI, but for large catalogs or automated pricing workflows, the Google Play Developer API is more practical.

To update the price of a base plan programmatically, use the `monetization.subscriptions.patch` method. You send a `Subscription` object with the updated configuration, setting the new price in the `RegionalBasePlanConfig` under the appropriate base plan.

Here is a simplified example of the request structure:

```
// Server-side: updating a base plan price
val regionConfig = RegionalBasePlanConfig(
    regionCode = "US",
    price = Money(
        currencyCode = "USD",
        units = 6,
        nanos = 990_000_000 // $6.99
    )
)

val basePlan = basePlan.copy(
    regionalConfigs = listOf(regionConfig)
)

val subscription = subscription.copy(
    basePlans = listOf(basePlan)
)

androidPublisher.monetization()
    .subscriptions()
    .patch(packageName, productId, subscription)
    .execute()
```

After the patch completes, the new price applies to all new purchases within a few hours. Existing subscribers remain in their legacy price cohort until you explicitly migrate them.

The `patch` method updates the subscription configuration itself. It does not trigger any notifications to existing subscribers. Think of it as changing the sticker price on the shelf. Only the next customer who picks up the product sees the new price.

Price Change Timeline and Propagation

After you call `monetization.subscriptions.patch` or update the price in the Play Console, propagation follows a predictable pattern:

1. **Immediate**: The API confirms the price change.
2. **Within a few hours**: New purchases reflect the updated price. The exact propagation time varies, but it is typically under 4 hours.
3. **No impact on existing subscribers**: Legacy cohort subscribers continue at their original price indefinitely until you migrate them.

If you are running a time sensitive promotion and need the price to be live by a specific hour, give yourself a buffer. Initiate the change well ahead of your target time. There is no way to force instant propagation.

181

Ending a Legacy Price Cohort

To move existing subscribers from their old price to the current base plan price, you end the legacy price cohort. This is the action that triggers Google's notification and consent system.

Via the Play Console

1. Go to **Monetize > Subscriptions** and select the subscription.
2. Open the base plan whose price you changed.
3. Navigate to the **Legacy price points** section.
4. Select the cohort you want to migrate and initiate the price change.
5. Choose opt in or opt out (if eligible) for price increases.

Via the API: basePlans.migratePrices

For programmatic migration, use `monetization.subscriptions.basePlans.migratePrices`. This method accepts a list of `RegionalPriceMigrationConfig` objects that specify which regions to migrate and how.

```
// Server-side: migrating a legacy price cohort
val migrationConfig = RegionalPriceMigrationConfig(
    regionCode = "US",
    oldestAllowedPriceVersionTime =
        "2025-01-01T00:00:00Z",
    priceIncreaseType =
        PriceIncreaseType.PRICE_INCREASE_TYPE_OPT_IN
)

val request = MigratePricesRequest(
    regionalPriceMigrations =
        listOf(migrationConfig),
    regionsVersion = RegionsVersion(version = "2022/02")
)

androidPublisher.monetization()
    .subscriptions()
    .basePlans()
    .migratePrices(
        packageName, productId,
        basePlanId, request
    )
    .execute()
```

The `oldestAllowedPriceVersionTime` field is a timestamp cutoff. All subscribers in legacy cohorts created before this timestamp will be migrated to the current price. This lets you target specific cohorts without

182

affecting more recent ones.

The `priceIncreaseType` field determines whether the migration uses opt in or opt out rules. If you set it to `PRICE_INCREASE_TYPE_OPT_OUT` but the migration does not meet regional, frequency, or amount requirements, Google automatically downgrades it to opt in. The first opt out price increase for each app must be initiated through the Play Console, not the API.

A successful request returns an empty response body, confirming the migration was queued. Google then begins the notification process according to the rules described in the following sections.

Price Decrease: Automatic, Email Notification, Authorization Window

When the current base plan price is lower than a legacy cohort's price, migrating that cohort is a price decrease. This is the simplest type of price change because it requires no consent from the subscriber.

Here is what happens:

1. You initiate the migration (Console or API).
2. Google sends an email notification to affected subscribers informing them their price is going down.
3. At their next renewal, subscribers are charged the lower price.

There is no opt in dialog, no freeze period, and no risk of cancellation. Users are simply notified and then charged less.

The authorization window matters here. Google authorizes payment up to 48 hours before a subscriber's renewal date in most regions. In India and Brazil, authorization happens up to 5 days before renewal. If a subscriber's payment was already authorized at the old (higher) price before the decrease took effect, they pay the old price for that cycle and the decrease applies at the following renewal.

For example, suppose you decrease the price on March 10. A subscriber whose renewal date is March 11 may have already been authorized at the old price on March 9. That subscriber pays the old price on March 11 and the new lower price on April 11.

License testers also receive email notifications for price decreases, so you can verify the notification flow during development.

Opt In Price Increase: 37 Day Notice, 7 Day Freeze, Auto Cancel

The default behavior for price increases is opt in. Subscribers must explicitly accept the higher price, or Google cancels their subscription. This is the most protective approach for users, and it is the only option in many regions.

The Timeline

The opt in price increase follows a precise timeline:

Day 0: You initiate the price migration. Google records the effective date as 37 days from now.

Days 0 through 7 (the freeze period): Nothing visible happens to subscribers. Google holds the price change for 7 days. This is your window to communicate the change to users through your own channels before Google starts sending notifications. You can show in app banners, send push notifications, or email users from your own systems. No Google notifications go out during this period.

Day 7 onward: Google begins sending email and push notifications to affected subscribers. The exact timing of notifications depends on each subscriber's individual renewal date. Google starts notifying a subscriber 30 days before their first renewal at the new price.

Day 37 (effective date): The price change becomes enforceable. Any renewal that occurs on or after this date is subject to the new price.

What Happens at Renewal

When a subscriber's renewal date arrives after the effective date:

- If the subscriber **accepted** the price increase, they are charged the new price. Life continues normally.
- If the subscriber **did not accept** (and did not cancel), Google automatically cancels their subscription. The user retains access until the end of the current billing period, then the subscription expires.

Concrete Example: Monthly Subscription

Suppose you have a $1/month subscription and you raise the price to $2/month on March 3.

- **Effective date**: April 9 (37 days later).
- **Alice** (renews on the 5th): Renews at $1 on March 5 and April 5 (both before the effective date). Receives notifications starting around April 5 (30 days before her May 5 renewal). If she accepts, she pays $2 on May 5.
- **Bob** (renews on the 29th): Renews at $1 on March 29 (before effective date). Receives notifications starting around March 30 (30 days before his April 29 renewal). If he accepts, he pays $2 on April 29.

Notice that Bob's first renewal at the new price comes sooner than Alice's, even though the migration was initiated on the same day. Each subscriber's timeline is relative to their own renewal date.

Concrete Example: Quarterly Subscription

For a quarterly (3 month) subscription at $1, raised to $2 on March 3:

- **Alice** (renews March 5): Pays $1 on March 5. Next renewal is June 5 (after the effective date). Receives notifications starting around May 6. Pays $2 on June 5 if she accepts.
- **Bob** (renews April 11): April 11 is after the April 9 effective date. Receives notifications starting around March 12. Pays $2 on April 11 if he accepts.

Longer billing periods mean fewer renewal opportunities, so subscribers may not encounter the new price for months after you initiate the change.

Showing Price Change Information in Your App

During the 7 day freeze period, use your own messaging to prepare users. After day 7, Google handles notifications, but you should also surface the price change in your app. Provide a deep link to the Play Store subscription management screen so users can easily review and accept the change.

Opt Out Price Increase: Conditional Availability

Opt-out price increases let you raise prices without requiring explicit acceptance. Subscribers are notified, but unless they actively cancel or change plans, they are charged the new price automatically. This is a less disruptive approach for your business, but it comes with significant restrictions.

Eligibility Requirements

Opt-out price increases are not universally available. They are subject to:

1. **Regional restrictions**: Only certain countries support opt out increases. The available regions and their specific rules are maintained by Google and can change over time.
2. **Amount limits**: There is a maximum allowed increase amount per region. The exact thresholds vary by country.
3. **Frequency limits**: Each base plan can only have one opt out price increase per country in any 365 day rolling window.
4. **Developer requirements**: When initiating an opt out increase, you must certify in the Play Console that your app's terms of service reserve the right to increase subscription prices with advance notice and provide clear, valid reasons for the increase.
5. **First opt out must use the Console**: The first opt out price increase for each app must be initiated through the Play Console. Subsequent opt out increases can use the API.

If you attempt an opt out increase via the API and it does not meet regional, frequency, or amount requirements, Google automatically converts it to an opt in increase. Your migration will still proceed, but subscribers will need to accept the new price.

Notification Timeline

The notification period for opt out increases depends on the subscriber's country:

- **30 day notification countries**: Subscribers receive email and push notifications starting 30 days before their first charge at the new price.
- **60 day notification countries**: Subscribers receive notifications starting 60 days before the first charge.

Unlike opt in increases, there is no 7 day freeze period for opt out increases. Google begins sending notifications immediately after you initiate the migration.

What Happens at Renewal

When a subscriber's renewal date arrives after the notification period:

- If the subscriber **took no action**, they are charged the new price. The subscription continues normally.

- If the subscriber **canceled or changed plans** during the notification period, the price increase does not apply. The subscription follows normal cancellation or plan change behavior.

Concrete Example: 30 Day Opt Out Region

You increase a $1/month subscription to $1.30 on January 2, in a region with a 30 day notification period.

- **Alice** (renews on the 14th): Renews at $1 on January 14. Receives notifications starting January 15. Pays $1.30 on February 14 unless she cancels.

The effective date is calculated as January 2 plus 30 days, which is February 1. Alice's first renewal after February 1 is February 14, so that is when the new price applies.

Overlapping Price Changes: Only the Latest Applies

If you initiate multiple price migrations for the same base plan before the first one fully resolves, only the latest migration applies. Google does not stack price changes or require users to respond to each one individually.

Here is what happens when price changes overlap:

1. The older price migration is marked as `CANCELED` in the `SubscriptionPurchaseV2` resource's `priceChangeDetails`.
2. You receive a `SUBSCRIPTION_PRICE_CHANGE_UPDATED` RTDN for the cancellation of the old migration.
3. The new price migration takes over. Its status appears as `OUTSTANDING` (for opt in increases) or `CONFIRMED` (for opt out increases and decreases).
4. You receive another `SUBSCRIPTION_PRICE_CHANGE_UPDATED` RTDN for the new migration.
5. The subscriber only needs to respond to the latest change.

Concrete Example: Two Opt In Increases

You raise the price from $1 to $2 on March 3 (effective April 9). Then on March 10, you raise it again from $2 to $3 (effective April 16).

- **Alice** (renews March 5): The first migration ($1 to $2) is canceled during the 7 day freeze period of the second migration. Alice never receives notifications about the $2 price. She receives notifications about the $3 price starting around April 5. Her May 5 renewal is at $3 if she accepts.

The 7 day freeze period of the second migration effectively absorbs the first migration. From the subscriber's perspective, only one price change ever existed.

Why This Matters for Your Backend

When you receive a `SUBSCRIPTION_PRICE_CHANGE_UPDATED` RTDN, always call `purchases.subscriptionsv2.get` to check the current state. Do not assume the RTDN corresponds to the most recent migration you initiated. It might be a cancellation notice for an older migration. Check the `priceChangeState` field to understand what actually happened.

Accidental Price Change Recovery

Mistakes happen. You might set the wrong price, migrate the wrong cohort, or change the price in the wrong direction. Google provides recovery paths for each scenario.

Recovering from an Accidental Opt In Price Increase

1. Change the base plan price back to the original value using the Console or API.
2. Navigate to the legacy price points page for the base plan.
3. Initiate a price decrease migration to move subscribers back to the original price.

Timing matters. If you catch the mistake within the 7 day freeze period, existing subscribers were never notified. The original migration is canceled cleanly. If you catch it after the freeze period but before any subscriber has paid the new price, the migration is canceled for subscribers who have not yet been charged. Subscribers whose payment was already authorized at the new price (within the 48 hour or 5 day authorization window) may still be charged the higher amount for one cycle.

Recovering from an Accidental Opt Out Price Increase

The recovery process is the same as for opt in:

1. Revert the base plan price to the original value.
2. Initiate a price decrease from the legacy price points page.

Since opt out increases have no freeze period, subscribers may have been notified immediately. If any subscriber has already been charged the higher price, the decrease applies at their next renewal.

Recovering from an Accidental Price Decrease

This is trickier because you are trying to move the price back up:

1. Revert the base plan price to the original (higher) value using the Console or API.
2. Navigate to the legacy price points page.
3. Initiate a price increase migration (opt in or opt out if eligible) to move subscribers back to the original price.

Whether the reversal is effective depends on timing relative to each subscriber's renewal date:

- **Valid cancellation**: If the period between your reversal and the subscriber's next renewal at the lower price exceeds the country specific notification window (30 or 60 days), the subscriber renews at the original higher price. The accidental decrease is effectively canceled.
- **Invalid cancellation**: If the period is shorter than or equal to the notification window, the subscriber pays the lower price for at least one cycle. After that, they go through the standard price increase process (opt in or opt out) to return to the original price.

The practical takeaway: catch pricing mistakes as quickly as possible. The longer you wait, the more subscribers will have been charged at the wrong price, and the more complex the recovery becomes.

Price Step-Up Consent (South Korea)

South Korean regulations require a special consent mechanism for price transitions that occur within a subscription's offer phases. Specifically, when a subscriber's free trial or introductory offer period ends and the subscription transitions to the higher base plan price, the subscriber must explicitly consent to the step up.

This is different from a developer initiated price change. A price step up happens because the subscriber's offer expired and the full price kicks in. The subscriber knew about this when they signed up, but Korean regulations require Google to collect explicit consent before the higher charge.

How It Works

When a subscriber in South Korea is approaching the end of a free trial or introductory pricing phase:

1. Google notifies the subscriber about the upcoming price step up.
2. The subscriber has a consent window (up to 30 days before the step up) to accept or decline.
3. If the subscriber consents, the subscription transitions to the base plan price normally.
4. If the subscriber does not consent before the step up date, Google automatically cancels the subscription.

Example: Free Trial to Base Price

A subscriber in South Korea signs up on March 3 with a 10 day free trial for a $4.99/month subscription.

- **March 3**: Subscription starts. Free trial begins.
- **March 13**: Free trial ends. If the subscriber consented during the trial, they are charged $4.99. If not, the consent period continues for up to 30 days (until April 12). If they still have not consented by the step up date, the subscription is canceled.

Example: Free Trial to Introductory Price to Base Price

If your offer has multiple phases (free trial, then intro price, then full price), consent is required at each transition:

- **First consent**: Free trial to introductory price.
- **Second consent**: Introductory price to base plan price.

Each transition requires its own consent from the subscriber.

Identifying Offer Phases in the API

Use the `offerPhase` field in `SubscriptionPurchaseLineItem` to determine which phase a subscriber is currently in:

```
// Server-side: checking offer phase
val lineItem = subscription.lineItems.first()
val phase = lineItem.offerPhase

when {
    phase?.freeTrial != null -> {
        // Subscriber is in free trial
    }
    phase?.introductoryPrice != null -> {
        // Subscriber is in intro pricing
    }
    else -> {
        // Subscriber is at base plan price
    }
}
```

This field helps you distinguish between a price step up (offer phase transition) and a developer initiated price change, which require different handling in your backend.

RTDN Notification Types for Price Changes

Google provides two Real Time Developer Notification types for price change events. Both arrive as `SubscriptionNotification` objects containing a `purchaseToken` and a `notificationType` integer.

SUBSCRIPTION_PRICE_CHANGE_UPDATED (Type 19)

This notification fires when:

- A price change migration is initiated for a subscriber.
- A subscriber accepts or declines an opt in price increase.
- An overlapping price change cancels a previous migration.
- The price change status updates for any reason.

When you receive this RTDN, call `purchases.subscriptionsv2.get` with the included `purchaseToken`. Inspect the `priceChangeDetails` on the relevant line item:

```
// Server-side: handling price change RTDN
fun handlePriceChangeRtdn(purchaseToken: String) {
    val subscription = playApi
        .getSubscriptionV2(packageName, purchaseToken)

    for (item in subscription.lineItems) {
        val priceChange = item.priceChangeDetails
            ?: continue

        when (priceChange.priceChangeState) {
            "OUTSTANDING" -> {
                // Opt-in increase pending acceptance
                val newPrice = priceChange.newPrice
                val chargeTime =
                    priceChange.expectedNewPriceChargeTime
                // Store and display to user
            }
            "CONFIRMED" -> {
                // User accepted, or opt out/decrease
                // Change will apply at next renewal
            }
            "APPLIED" -> {
                // Price change already took effect
            }
            "CANCELED" -> {
                // Migration was canceled
                // (overlapping change or reversal)
            }
        }
    }
}
```

The `priceChangeMode` field tells you the type of change:

MODE	MEANING
`PRICE_DECREASE`	Price is going down
`PRICE_INCREASE`	Opt-in price increase
`OPT_OUT_PRICE_INCREASE`	Opt-out price increase

The `expectedNewPriceChargeTime` field tells you when the subscriber will first be charged at the new price. This field is only populated when the change has not yet been applied.

190

SUBSCRIPTION_PRICE_STEP_UP_CONSENT_UPDATED (Type 22)

This notification fires only for subscriptions in South Korea (or other regions where price step up consent is required). It is sent when:

- The consent period for a price step up begins (the subscriber is approaching the end of a free trial or introductory phase).
- The subscriber provides or declines consent.

When you receive this RTDN, call `purchases.subscriptionsv2.get` and check the subscription's line item for the current offer phase and consent state. If the subscriber has not consented and the step up date is approaching, you may want to surface a reminder in your app.

Backend Processing Pattern

A clean way to handle both RTDN types is to route them through a single handler:

```
// Server-side: RTDN router
fun handleSubscriptionRtdn(
    notificationType: Int,
    purchaseToken: String
) {
    val subscription = playApi
        .getSubscriptionV2(packageName, purchaseToken)

    when (notificationType) {
        19 -> handlePriceChange(subscription)
        22 -> handlePriceStepUpConsent(subscription)
        // ... other notification types
    }
}
```

Always fetch the latest subscription state from the API after receiving an RTDN. The notification tells you something changed, but the API response is the source of truth for the current state.

Testing Price Changes with Play Billing Lab

Changing subscription prices in production to test your handling is risky. You could accidentally affect real subscribers. Google provides Play Billing Lab specifically for testing price changes safely.

Setting Up a Price Change Test

1. Install the Play Billing Lab app on your test device.
2. Make sure your test account is configured as a license tester in the Play Console.
3. Purchase a subscription with your license tester account.

4. In Play Billing Lab, open the **Subscription settings** card and click **Manage**.
5. Select the active subscription you want to test.
6. Enter the new price.
7. Check or uncheck the **User opt out** checkbox depending on which flow you want to test.
8. Click **Apply**.

The price change applies only to your test subscription at the next renewal. Other subscribers (including other testers) are not affected.

Compressed Notification Timelines

License tester subscriptions already use compressed billing periods (a monthly subscription renews every 5 minutes, for example). Play Billing Lab compresses notification timelines to match:

ACTUAL BILLING PERIOD	TEST BILLING PERIOD	OPT-IN/OPT-OUT (30 DAY)	OPT-OUT (60 DAY)
1 week	5 minutes	5 minutes	10 minutes
1 month	5 minutes	5 minutes	10 minutes
3 months	10 minutes	3 minutes	6 minutes
6 months	15 minutes	2 minutes	4 minutes
1 year	30 minutes	3 minutes	6 minutes

This lets you observe the full notification and renewal cycle in minutes rather than weeks.

Testing Tips

Defer billing after triggering a price change. Because license tester renewal periods are so short, the subscription might renew before the price change notification is even registered. Defer the billing by at least one hour after applying the price change to give the system time to process.

Test all three scenarios. Create separate test flows for price decreases, opt in increases, and opt out increases. Each has different notification behavior and user interaction requirements.

Verify your RTDN handling. After applying a price change in Play Billing Lab, check that your server receives the `SUBSCRIPTION_PRICE_CHANGE_UPDATED` RTDN and processes it correctly. Verify that your `purchases.subscriptionsv2.get` call returns the expected `priceChangeDetails`.

Test the cancellation path. For opt in increases, let the test subscription renew without accepting the price change. Verify that your backend correctly handles the automatic cancellation.

Test overlapping changes. Apply one price change, then immediately apply another before the first resolves. Verify that your backend processes the cancellation of the first change and the activation of the second.

Chapter 15: Security and Fraud Prevention

Every purchase that flows through your app is a target. Attackers modify APKs to bypass purchase checks, replay stolen purchase tokens, abuse refund policies, and share accounts at scale. If your security model relies on the client alone, you have already lost. The Play Billing Library runs on the user's device, and anything running on the user's device can be tampered with.

This chapter gives you a practical, layered defense strategy for securing your billing implementation. You will learn how to move sensitive logic to your backend, verify every purchase through a 7 step workflow, protect your content from extraction, detect and respond to voided purchases, and defend against the most common attack vectors targeting Android apps.

Moving Sensitive Logic to Your Backend

The most important security decision you can make is this: never trust the client. Your Android app runs in an environment you do not control. Users can root their devices, install modified versions of your APK, intercept network traffic, and manipulate local storage. Any check that happens only on the device can be bypassed.

This means you should move all of the following to your backend server:

Purchase verification. When a purchase completes, the client should send the purchase token to your server. Your server calls the Google Play Developer API to confirm the purchase is real. The client never decides whether a purchase is valid.

Entitlement decisions. Your server maintains the authoritative record of what each user has access to. The client queries your server to find out what content to unlock. If the client stores entitlement state locally, treat it as a cache that your server can override at any time.

Acknowledgement and consumption. While the Play Billing Library supports client side acknowledgement, performing it on your server ties the acknowledgement directly to your entitlement grant. This guarantees you never acknowledge a purchase you have not fulfilled, and you never fulfill a purchase you cannot acknowledge.

Subscription state management. Subscription lifecycles are complex. Renewals, cancellations, grace periods, account holds, and pauses all generate state changes. Your server should process Real Time Developer Notifications (RTDNs) and maintain the current subscription state in your database. The client queries your server to determine access, not the Play Billing Library directly.

The general principle is straightforward: the client handles UI and user interaction, your server handles business logic and trust decisions. The client tells your server what happened (a purchase token was received), and your server decides what to do about it (verify, grant, acknowledge).

The 7 Step Purchase Verification Workflow

Purchase verification is the backbone of your security model. Every purchase that arrives from the client should pass through these seven steps on your backend before the user gets access to anything.

Step 1: Receive the Purchase Token from the Client

After a successful purchase, your app receives a `Purchase` object in the `PurchasesUpdatedListener`. Extract the purchase token and send it to your backend along with the user's account identifier:

```
override fun onPurchasesUpdated(
    billingResult: BillingResult,
    purchases: List<Purchase>?
) {
    if (billingResult.responseCode ==
        BillingResponseCode.OK
    ) {
        purchases?.forEach { purchase ->
            if (purchase.purchaseState ==
                PurchaseState.PURCHASED
            ) {
                sendToBackend(
                    purchase.purchaseToken,
                    purchase.products,
                    currentUserId
                )
            }
        }
    }
}
```

Send the token over HTTPS with certificate pinning if possible. Include your own authentication token so your server knows which user is claiming the purchase.

Step 2: Call Google Play Developer API to Verify

Your backend uses the purchase token to query the Google Play Developer API. For one time products, call `purchases.products.get`. For subscriptions, call `purchases.subscriptionsv2.get`.

The API response contains the authoritative purchase record from Google. This is the source of truth. It tells you whether the purchase actually happened, what was purchased, what the current state is, and when it occurred.

If the API returns an error (such as a 404), the purchase token is invalid. Reject it immediately.

Step 3: Validate Response Fields

Once you have the API response, validate that the purchase matches what you expect:

Package name. Confirm the `packageName` in the response matches your app's package name. A valid purchase token from a different app should not grant access in yours.

194

Product ID. Confirm the `productId` (or subscription ID) matches what the client claimed was purchased. An attacker could send a token for a cheaper product and claim it was for a more expensive one.

Order ID. Store the `orderId` for your records. You will use it for refund tracking and customer support. Each unique transaction should have a unique order ID.

If any of these fields do not match the expected values, reject the purchase. Log the discrepancy for investigation.

Step 4: Check Purchase State

The API response includes a purchase state field. For one time products, the `purchaseState` field should be `0` (PURCHASED). For subscriptions, check the `acknowledgementState` and `expiryTimeMillis`.

Do not grant access for purchases in the `PENDING` state (value `2`). Pending purchases mean the user has chosen a delayed payment method like cash at a convenience store. You will receive an RTDN when the payment completes.

For subscriptions, verify that the subscription has not expired and is not in a revoked state. The `subscriptionState` field from `purchases.subscriptionsv2.get` tells you the current lifecycle state.

Step 5: Check for Duplicate Tokens

Before granting entitlement, check your database for the purchase token. If this exact token has already been processed, do not grant a second entitlement. This prevents replay attacks where an attacker sends the same valid purchase token multiple times, possibly from different accounts.

Your database should have a unique constraint on the purchase token column. When a duplicate arrives, return the existing entitlement status rather than creating a new one. If the duplicate comes from a different user account than the original, flag it for investigation.

Step 6: Grant Entitlement

If the purchase passes all five checks above, grant the user access to the purchased content or feature. Write the entitlement to your database with the purchase token, user ID, product ID, order ID, and timestamp.

This is the only point at which the user should gain access. Everything before this step is validation. Everything after is cleanup.

Step 7: Acknowledge the Purchase

After granting entitlement, acknowledge the purchase through the Google Play Developer API on your server. For one time products, call `purchases.products.acknowledge`. For subscriptions, call `purchases.subscriptions.acknowledge`.

You must acknowledge within 3 days of the purchase. If you fail to do so, Google automatically refunds the purchase. Performing acknowledgement on your server, after entitlement grant, ensures these two actions stay in sync.

195

If the acknowledgement call fails, retry it with exponential backoff. A purchase that has been granted but not acknowledged will eventually be refunded, so treat acknowledgement failures as high priority.

Protecting Unlocked Content

Verification stops fake purchases. But what about the content itself? If your premium content is bundled inside the APK, anyone with a file extractor can pull it out without paying.

Never Bundle Premium Content in the APK

An APK is a ZIP file. Anyone can unzip it and access every resource, asset, and raw file inside. If your premium wallpapers, audio files, level data, or documents are sitting in the `assets/` or `res/` directories, they are free for the taking regardless of your purchase checks.

Instead, host premium content on your server and download it only after your server confirms the user has a valid entitlement. This keeps the content off the device until the user has paid for it.

For apps where offline access is important, download the content to internal storage (not external storage, which is readable by other apps) after purchase verification. Use Android's encrypted file storage or encrypt the files yourself with a key derived from a server provided secret.

Device Specific Encryption

When you must store premium content on the device, encrypt it with a key that is specific to both the user and the device:

1. After purchase verification, your server generates an encryption key tied to the user's account and the device's unique identifier.
2. Your server sends this key to the client over HTTPS.
3. Your app encrypts the content with this key before writing it to disk.
4. Your app requests the key from your server when it needs to decrypt the content.

This approach means that even if someone copies the encrypted files to another device, they cannot decrypt them without the correct key. If the user's entitlement is revoked, your server stops providing the key, and the content becomes inaccessible.

For the encryption itself, use AES-256-GCM through Android's `javax.crypto` APIs. Do not invent your own encryption scheme.

Detecting Voided Purchases

A voided purchase is one that was originally valid but has since been reversed. This happens when Google issues a refund, a chargeback occurs, or Google's fraud detection system flags the transaction. The user got access to your content, but the payment is no longer valid.

The Voided Purchases API

The Google Play Developer API provides the Voided Purchases API at `purchases.voidedpurchases.list`. This endpoint returns a list of purchases that have been voided for your app within a specified time range.

You should poll this endpoint regularly, at least once per day. For each voided purchase in the response:

1. Look up the purchase token or order ID in your database.
2. Identify the user who was granted the entitlement.
3. Decide on the appropriate enforcement action (covered in the next section).

The API returns a `voidedReason` field that tells you why the purchase was voided:

- **0 (Other)**: General void, no specific reason provided.
- **1 (Remorse)**: The user requested a refund and Google granted it (buyer's remorse).
- **2 (Not received)**: The user reported not receiving the purchased content.
- **3 (Defective)**: The user reported the content was defective.
- **4 (Accidental purchase)**: The user claimed the purchase was unintentional.
- **5 (Fraud)**: Google's fraud detection flagged the transaction.
- **6 (Friendly fraud)**: The user disputed the charge through their bank.
- **7 (Chargeback)**: A chargeback was filed against the transaction.

The `voidedSource` field tells you whether the void came from the user (value `0`) or from Google (value `1`). This distinction matters for your enforcement decisions. A user requesting a single refund is different from a pattern of chargebacks flagged by Google.

Processing Voided Purchases at Scale

For apps with significant transaction volume, batch your voided purchase processing. Query the API with a pagination token to handle large result sets. Store the latest `voidedTimeMillis` value so your next query picks up where the last one left off.

Do not revoke access in real time as you discover each voided purchase. Batch the revocations and apply them during your regular entitlement sync. This reduces the chance of revoking access for a purchase that was voided by mistake (which does occasionally happen, and Google may reverse the void).

Graduated Enforcement for Fraud

Not every voided purchase means fraud. Users make legitimate refund requests, accidental purchases happen, and sometimes Google's fraud detection generates false positives. A good enforcement strategy is graduated, applying increasingly severe consequences based on the pattern of behavior.

Level 1: Warning

For a user's first voided purchase, especially one with a reason of "remorse" or "accidental purchase," take no enforcement action on the entitlement. Instead, log the event and flag the account for monitoring. If the user is currently subscribed through a different active purchase, there is nothing more to do.

197

If the voided purchase was the user's only entitlement, revoke access to the specific content but do not penalize the account. The user may have had a genuine issue.

Level 2: Disable Features

If a user accumulates multiple voided purchases within a short time window (for example, three voids within 30 days), begin restricting access more aggressively. Revoke all entitlements associated with voided purchases. You may also temporarily disable the ability to make new purchases within your app for that account.

At this stage, display a message explaining that purchase privileges have been restricted due to unusual refund activity, and direct the user to customer support if they believe this is an error.

Level 3: Revoke and Ban

For users with a clear pattern of abuse, such as repeated purchase and refund cycles, chargebacks, or purchases flagged as fraud by Google, revoke all entitlements and ban the account from future purchases. This is the appropriate response for confirmed fraud.

Store the device identifiers and account information associated with banned users so you can detect ban evasion. If a banned user creates a new account on the same device, your server can apply restrictions immediately.

Implementing Graduated Enforcement

Track each user's voided purchase history in your database:

- Total number of voided purchases
- Voided reasons and sources
- Time span of voided purchases
- Current enforcement level

Run your enforcement logic on your server during the voided purchase processing batch. Evaluate each user's history against your thresholds and apply the appropriate level. Log every enforcement action for auditing and customer support purposes.

The specific thresholds (how many voids trigger each level, the time windows, which void reasons are weighted more heavily) depend on your business. Start conservative, monitor the results, and adjust over time.

Helping Google Detect Fraud

Google has its own fraud detection systems that analyze purchase patterns across all apps. You can make these systems more effective by providing additional signals when launching the billing flow.

obfuscatedAccountId

The `obfuscatedAccountId` is a one way hash of your user's internal account identifier. You set it on the `BillingFlowParams` before launching the purchase:

198

```
val flowParams = BillingFlowParams.newBuilder()
    .setProductDetailsParamsList(
        listOf(productDetailsParams)
    )
    .setObfuscatedAccountId(
        hashAccountId(currentUser.id)
    )
    .build()
```

This allows Google to correlate purchases across sessions and devices for the same user. If a single account is making purchases from dozens of different devices, that is a fraud signal. Without the obfuscated account ID, Google can only correlate by Google account, which misses cases where attackers use multiple Google accounts.

Use a consistent, irreversible hash function (SHA-256 is a good choice) so the value cannot be reversed to identify the user but stays the same across sessions.

obfuscatedProfileId

The `obfuscatedProfileId` serves a similar purpose but at the profile level. If your app supports multiple profiles per account (common in streaming and gaming apps), set this to a hash of the active profile ID:

```
val flowParams = BillingFlowParams.newBuilder()
    .setProductDetailsParamsList(
        listOf(productDetailsParams)
    )
    .setObfuscatedAccountId(
        hashAccountId(currentUser.id)
    )
    .setObfuscatedProfileId(
        hashProfileId(currentUser.activeProfileId)
    )
    .build()
```

Both identifiers appear in the purchase record returned by the Google Play Developer API. You can use them on your backend to correlate purchases with your internal user records, even if the user switches Google accounts.

Set both identifiers on every purchase flow. They cost nothing to include and give both you and Google better visibility into purchase patterns.

Common Attack Vectors and Defenses

199

Understanding how attackers target in app purchases helps you build defenses that actually work. Here are the most common attack vectors and how to defend against each one.

Modified APKs Bypassing Client Side Checks

The attack. An attacker decompiles your APK, removes or modifies the purchase verification code, repackages the app, and distributes it. Users who install the modified APK get full access without paying. Some tools automate this process entirely, requiring zero technical skill from the end user.

The defense. If your purchase verification runs entirely on the client, this attack is trivial. The primary defense is server side verification, which renders client side tampering irrelevant. The modified APK can fake any local check it wants, but it cannot forge a valid response from the Google Play Developer API.

Additional layers of defense:

- **Google Play Integrity API.** Call the Play Integrity API before or during the purchase flow. It tells your server whether the app binary is genuine and unmodified, whether the device passes basic integrity checks, and whether the app was installed from the Play Store. Reject purchases from devices that fail integrity checks.
- **Certificate pinning.** Pin your server's TLS certificate in the app so attackers cannot intercept traffic between your app and your backend with a proxy.
- **Code obfuscation.** Use R8 (or ProGuard) with aggressive obfuscation settings. This does not prevent decompilation, but it raises the effort required to understand and modify your code.

None of these secondary measures replace server side verification. They complement it.

Fake Purchase Tokens

The attack. An attacker sends fabricated or stolen purchase tokens to your backend, hoping your server will accept them and grant entitlements without proper validation.

The defense. Your 7 step verification workflow handles this. When your server calls the Google Play Developer API with a fake token, the API returns an error. When the server calls with a stolen token from a different app or product, the package name and product ID validation catches it. When the server sees a replayed token that was already processed, the duplicate check catches it.

The key is to never skip any of the seven steps. Each step catches a different type of invalid token.

Refund Abuse

The attack. A user purchases content, consumes or downloads it, and then requests a refund from Google. They keep the content and get their money back. Some users do this systematically, cycling through purchase and refund for every piece of premium content.

The defense. Poll the Voided Purchases API regularly and apply your graduated enforcement strategy. For consumable content (like in game currency), track what the user spent the currency on. If a purchase is refunded, deduct the equivalent amount from the user's balance. If the balance goes negative, restrict access until the balance is resolved.

200

For non consumable content, revoke access to the specific content associated with the refunded purchase. Your server should maintain a mapping between purchase tokens and the content they unlocked.

Google also provides the Play Console's "order management" section where you can block users from requesting refunds for your app. Use this for confirmed abusers.

Account Sharing

The attack. One user purchases a subscription and shares their account credentials with multiple people. Alternatively, they share a session token or authentication cookie. Instead of paying for five subscriptions, five people share one.

The defense. There is no perfect solution for account sharing, but you can limit its impact:

- **Concurrent device limits.** Track the number of devices actively using each account. When the limit is exceeded, force a sign out on the oldest device or require re authentication.
- **Device fingerprinting.** Record the devices associated with each account. If an account is used on an unusually high number of distinct devices over a short period, flag it for review.
- **Streaming/access restrictions.** If your app delivers content in real time (streaming media, live data), limit concurrent streams per account.

The `obfuscatedAccountId` and `obfuscatedProfileId` you set during the purchase flow help here as well. If a single purchase token shows activity from many different device fingerprints, that is a signal of sharing.

Be careful not to punish legitimate use cases. A user may have a phone, tablet, and Chromebook. Your limits should accommodate normal multi device usage while catching abuse.

Timezone Manipulation

The attack. A user changes their device's timezone or date settings to exploit time based offers, extend free trials, or manipulate expiration checks. For example, setting the clock forward to skip a waiting period, or backward to extend a trial.

The defense. Never use the device clock for time sensitive decisions. Your server should be the source of truth for all time related checks:

- **Trial eligibility.** Check trial eligibility on your server based on server timestamps, not client reported times.
- **Offer windows.** Determine whether a promotional offer is active based on your server's clock.
- **Expiration checks.** When the client asks whether a subscription is active, your server compares the subscription's `expiryTimeMillis` (from the Google Play Developer API) against the server's current time.

The Google Play Developer API returns all timestamps in server time (milliseconds since epoch, UTC). Use these values for your comparisons, never the device's `System.currentTimeMillis()` for business logic that determines access.

201

If your app must make offline time decisions (for example, caching access for a period when the device is offline), use a combination of server provided timestamps and monotonic clocks (`SystemClock.elapsedRealtime()`) that cannot be manipulated by changing the device's wall clock.

Trademark and Copyright Infringement Considerations

Security is not only about protecting your revenue from fraud. You also need to protect your app from legal risk related to intellectual property.

Protecting Your Own Content

If your app sells digital content such as images, music, text, or video, ensure you have the rights to distribute that content commercially. Licensing agreements with content creators should explicitly cover in app distribution on Android. Selling content you do not have distribution rights for exposes you to takedown requests and legal action, regardless of how well your billing implementation works.

Register trademarks for your app name, brand, and any distinctive product names. This gives you legal standing to pursue copycats who clone your app and sell it under a similar name, potentially stealing your customers and revenue.

Avoiding Infringement of Others' IP

Do not use trademarked names, logos, or copyrighted material in your product listings, subscription descriptions, or promotional materials without authorization. This includes:

- Using another company's brand name in your product titles to attract searches.
- Including copyrighted images or icons in your app's marketing materials.
- Describing your product by reference to a competitor's trademarked product name.

Google enforces intellectual property policies on the Play Store. Violations can result in your app being suspended, your developer account being terminated, or both. An app suspension halts all billing and voids your active subscriber relationships, which is a far greater business impact than the original infringement may seem worth.

Dealing with Clones and Copycat Apps

If someone clones your app and sells it on the Play Store, you have several options:

- **DMCA takedown.** File a DMCA takedown request through the Play Console's reporting tools. Google is required to respond to valid DMCA requests and will typically remove infringing apps.
- **Trademark complaint.** If the clone uses your trademarked name or branding, file a trademark complaint through Google's legal channels.
- **Play Integrity as a signal.** While not a direct anti piracy tool, the Play Integrity API can tell your server whether the requesting app was installed from the Play Store with your genuine signing certificate. Requests from apps with a different signature are clones.

Monitor the Play Store periodically for apps that mimic yours. Automated monitoring services can alert you when new apps appear with similar names, descriptions, or screenshots.

203

Chapter 16: Testing Your Integration

A billing integration that works on your development machine does not mean it works in production. Users have different payment methods, live in different countries, and encounter subscription states your app may never have seen during development. If you ship without thorough testing, you will find bugs when real money is involved, and those bugs are expensive.

Google provides a comprehensive set of testing tools for Play Billing. You can simulate purchases without being charged, accelerate subscription lifecycles that would normally take weeks, test specific error scenarios, and validate behavior across regions. This chapter walks you through all of them and gives you a concrete set of test cases to run before every release.

License Testers

License testers are Google accounts that can make test purchases in your app without being charged real money. This is the foundation of all billing testing. Without license testers configured, every purchase attempt either costs real money or fails entirely.

Setting Up License Testers

To add a license tester, go to the Google Play Console, navigate to **Setup > License testing**, and add the Gmail address of the Google account you want to use for testing. The account must be a valid Gmail or Google Workspace address.

A few things to know about license testers:

- License testers are configured at the developer account level, not per app. Once you add a tester, they can test purchases across all your apps.
- The tester must be signed into the Google account on the test device. If the device has multiple accounts, the account used in the Google Play Store app is the one that matters.
- Changes take effect within minutes. You do not need to publish a new version of your app after adding a tester.
- License testers can test purchases on apps that are not yet published, as long as the app has been uploaded to a test track (internal, closed, or open) at least once.

Test Payment Methods

When a license tester initiates a purchase, the Google Play purchase dialog shows a special set of test payment instruments instead of real payment methods. These test instruments let you simulate different payment outcomes:

Test card, always approves: This instrument completes the purchase immediately with a successful result. Use this for the majority of your testing. It validates that your purchase flow, acknowledgment logic, and entitlement granting all work correctly.

Test card, always declines: This instrument causes the payment to fail. Your `PurchasesUpdatedListener` receives a `BillingResponseCode.ERROR` result. Use this to verify that your app handles payment failures gracefully, shows appropriate error messages, and does not grant entitlements when payment fails.

Slow test card, always approves: This instrument introduces a delay before the purchase completes. The purchase initially enters a pending state, then completes after a few minutes. Use this to test your pending purchase handling, which is especially important if you have enabled pending transactions.

Slow test card, always declines: This instrument introduces a delay and then fails. The purchase enters a pending state and eventually transitions to a failed state. Use this to verify your app handles the case where a pending purchase is ultimately declined.

These test instruments only appear for license tester accounts. Regular users never see them.

Auto Refund for Unacknowledged Test Purchases

When a license tester makes a purchase and your app does not acknowledge it within 3 minutes, Google automatically refunds and revokes the purchase. This mirrors the production behavior where unacknowledged purchases are refunded after 3 days, but the dramatically shorter window makes it practical for testing.

This 3 minute window is actually one of the most useful testing tools at your disposal. It lets you verify that your acknowledgment logic works correctly:

1. Make a test purchase.
2. Intentionally do not acknowledge it (comment out your acknowledgment code or disconnect from your server).
3. Wait 3 minutes.
4. Verify that the purchase disappears from `queryPurchasesAsync` and that your app revokes the entitlement.

If your app still shows premium content after those 3 minutes, your entitlement logic has a bug. You are probably caching the entitlement locally without rechecking purchase validity.

This behavior applies only to test purchases from license testers. Production purchases have a 3 day acknowledgment window.

Test Tracks

Before a license tester can install and test your app, you need to upload a build to one of Google Play's test tracks. Google provides three test tracks, each designed for a different stage of your release process.

Internal Testing Track

The internal testing track is the fastest way to get a build onto a test device. Builds uploaded to this track are available almost immediately (usually within minutes). You can have up to 100 testers on this track.

Use internal testing for day to day development and QA. Because builds are available quickly, you can iterate rapidly on billing changes without waiting for Google's review process.

Closed Testing Track

Closed testing requires you to create a tester list and manage who has access. Builds may take longer to be available compared to internal testing. You can create multiple closed testing tracks for different groups of testers.

Use closed testing for beta programs or when you want to test with a specific group of external users before a wider release.

Open Testing Track

Open testing makes your app available to anyone who wants to join the test. Users can find your test through the Play Store listing. This track goes through app review, similar to production releases.

Use open testing for large scale pre release testing where you want real user feedback before going to production.

For billing testing specifically, the internal testing track is almost always what you want. It gets builds onto devices fast, and combined with license tester accounts, gives you a complete test environment.

Play Billing Lab

Play Billing Lab is a companion app from Google that gives you fine grained control over billing test scenarios that would otherwise be difficult or impossible to reproduce. Install it from the Google Play Store on your test device.

Changing Play Country

By default, Google determines a user's country based on their IP address and account settings. This makes it difficult to test how your app behaves for users in different regions. Play Billing Lab lets you override the Play country on your test device, so you can see the exact prices, currencies, and available payment methods that users in that country would see.

To change the country, open Play Billing Lab, select your app, and choose a different country from the list. Your app will then receive product details with pricing localized to that country. This is essential for testing multi region pricing configurations and verifying that your UI handles different currencies and price formats correctly.

Testing Trial and Intro Offers Repeatedly

In production, a user can only redeem a free trial or introductory offer once per product. This is a problem for testing because after you use the trial on your test account, you cannot test the trial flow again without creating a new account.

206

Play Billing Lab removes this restriction. You can reset the trial and introductory offer eligibility for your test account, letting you test trial sign up flows as many times as you need. This is invaluable for verifying that your app correctly displays offer pricing, handles the transition from trial to paid, and manages cancellations during a trial period.

Testing Price Changes

When you change the price of a subscription in the Play Console, existing subscribers may need to opt in to the new price (for price increases) or are automatically moved to the new price (for price decreases). Testing this flow in production would require creating a subscription, waiting for a renewal period, and then changing the price.

Play Billing Lab lets you simulate price change scenarios directly. You can test both opt in required price increases and automatic price changes, verifying that your app shows the right messaging and handles the user's response correctly.

Accelerating Subscription State Transitions

One of the most powerful features of Play Billing Lab is the ability to manually trigger subscription state transitions. Instead of waiting for time based events to happen naturally (even with accelerated test timelines), you can force a subscription into specific states: renewal, grace period, account hold, pause, cancellation, and more.

This gives you deterministic control over testing. Rather than hoping the timing works out, you can put a subscription into exactly the state you want to test and verify your app's behavior immediately.

Accelerated Renewal Periods

Test subscriptions renew on a compressed schedule. This is automatic for all license tester purchases and does not require any configuration. The mapping is:

PRODUCTION PERIOD	TEST PERIOD
1 week	5 minutes
1 month	5 minutes
3 months	10 minutes
6 months	15 minutes
1 year	30 minutes

This means a monthly subscription that would take 30 days to renew in production renews every 5 minutes during testing. A yearly subscription renews every 30 minutes. This compression lets you test multiple renewal cycles in a single testing session.

207

The compressed schedule applies to all time based subscription events, not just renewals. Trial periods, grace periods, account holds, and paused states all run on the same accelerated timeline.

Maximum 6 Test Renewals

Test subscriptions do not renew indefinitely. After 6 renewals, Google automatically cancels the test subscription. This means you get the initial purchase plus 6 renewal events before the subscription expires.

Plan your testing around this limit. If you need to test behavior beyond 6 renewals, you will need to create a new test subscription. For most scenarios, 6 renewals is plenty to verify that your renewal handling, entitlement checks, and RTDN processing work correctly.

After the 6th renewal, the subscription enters the expired state as if the user had cancelled. Your app should handle this transition the same way it handles any other expiration.

Time Compression for Subscription States

Beyond renewal periods, Google also compresses the duration of intermediate subscription states during testing:

SUBSCRIPTION STATE	TEST DURATION
Free trial	3 minutes
Grace period	5 minutes
Account hold	10 minutes
Pause	Varies (compressed)

These compressed durations mean you can test an entire subscription lifecycle in minutes rather than days:

1. A user starts a free trial (3 minutes in test).
2. The trial converts to a paid subscription (immediate).
3. A payment fails and the subscription enters grace period (5 minutes in test).
4. Grace period expires without recovery and the subscription enters account hold (10 minutes in test).
5. Account hold expires and the subscription is cancelled.

In production, that sequence would take a billing period plus 7 to 30 days depending on your grace period and account hold configuration. In testing, you can observe the entire flow in under 20 minutes.

11 Subscription Test Cases

Before shipping any billing integration to production, you should verify these 11 scenarios. Each one tests a distinct subscription state transition that real users will encounter.

1. New Subscription Purchase

What to test: A user buys a subscription for the first time.

Steps:

1. Query products and verify that subscription details load correctly.
2. Launch the billing flow with the "always approves" test card.
3. Verify that `PurchasesUpdatedListener` receives the purchase with `BillingResponseCode.OK`.
4. Acknowledge the purchase.
5. Verify that `queryPurchasesAsync` returns the active subscription.
6. Confirm your app grants the correct entitlement.

What can go wrong: Forgetting to acknowledge the purchase (it will be refunded in 3 minutes for test, 3 days in production). Granting the wrong entitlement level. Not handling the case where `PurchasesUpdatedListener` delivers the result after your activity has been destroyed.

2. Renewal

What to test: An active subscription renews successfully.

Steps:

1. Start with an active test subscription.
2. Wait for the accelerated renewal period (5 minutes for monthly).
3. Verify that `queryPurchasesAsync` still returns the subscription as active.
4. Check that your server receives an RTDN with `SUBSCRIPTION_RENEWED` notification type.
5. Confirm the entitlement remains active.

What can go wrong: Your server not processing RTDNs correctly, leading to stale entitlement data. Your app not refreshing purchase state after renewal.

3. Grace Period Entry and Recovery

What to test: A payment fails at renewal, the subscription enters grace period, and the user fixes their payment method.

Steps:

1. Start with an active test subscription.
2. Use Play Billing Lab to trigger a payment failure at renewal.
3. Verify the subscription enters grace period. During grace period, the user should still have access.
4. Verify your app shows a message encouraging the user to fix their payment method.
5. Fix the payment method (or use Play Billing Lab to recover).
6. Verify the subscription returns to active state and the entitlement continues.

What can go wrong: Revoking access during grace period. Not notifying the user that their payment failed. Not re granting access after recovery.

4. Account Hold Entry and Recovery

What to test: A subscription enters account hold after grace period expires (or directly if you have not enabled grace period), and the user recovers.

Steps:

1. Start with a subscription in grace period (or trigger a payment failure).
2. Let the grace period expire (5 minutes in test).
3. Verify the subscription enters account hold. During account hold, the user should not have access.
4. Verify your app shows a message that their subscription is on hold.
5. Simulate the user fixing their payment method.
6. Verify the subscription reactivates and access is restored.

What can go wrong: Still granting access during account hold. Not providing a deep link to the Play Store subscription management page where the user can fix their payment. Not restoring access after recovery.

5. User Cancellation

What to test: A user cancels their subscription through Google Play.

Steps:

1. Start with an active test subscription.
2. Cancel the subscription through the Google Play Store app on the device (Settings > Payments & subscriptions > Subscriptions).
3. Verify that the subscription remains active until the end of the current billing period.
4. Verify your app shows that the subscription will not renew.
5. After the billing period ends (accelerated in test), verify the entitlement is revoked.

What can go wrong: Revoking access immediately at cancellation instead of at period end. Not communicating to the user when their access will end. Not handling the expired state after the period ends.

6. Developer Cancellation

What to test: You (the developer) revoke a subscription through the Google Play Developer API.

Steps:

1. Start with an active test subscription.
2. Call the `purchases.subscriptions.revoke` or `purchases.subscriptionsv2.revoke` API endpoint from your server.
3. Verify that the subscription is immediately revoked (unlike user cancellation, developer revocation takes effect right away).

4. Verify your app revokes the entitlement promptly.

What can go wrong: Your app not checking for revocations frequently enough. Caching entitlements locally without server validation.

7. Pause and Resume

What to test: A user pauses their subscription and later resumes it.

Steps:

1. Configure your subscription in the Play Console to allow pausing.
2. Start with an active test subscription.
3. Pause the subscription through the Play Store app.
4. Verify that the subscription remains active until the current period ends, then enters the paused state.
5. Verify your app revokes access when the paused state begins.
6. Resume the subscription (through the Play Store app or by using Play Billing Lab).
7. Verify the subscription reactivates and access is restored.

What can go wrong: Not supporting pause at all and confusing users who paused through the Play Store. Not restoring access after resume.

8. Upgrade and Downgrade with Each Replacement Mode

What to test: A user changes their subscription plan using each of the available replacement modes.

Steps:

1. Start with an active subscription on one plan.
2. For each replacement mode (`WITH_TIME_PRORATION`, `CHARGE_PRORATED_PRICE`, `CHARGE_FULL_PRICE`, `WITHOUT_PRORATION`, `DEFERRED`), launch a plan change flow.
3. Verify that a new purchase token is generated for each plan change.
4. Verify that the old purchase token is linked via `linkedPurchaseToken`.
5. Verify that your backend correctly transfers the entitlement from the old token to the new one.
6. Verify the financial behavior matches the replacement mode (immediate switch vs. deferred, prorated charge vs. full charge).

What can go wrong: Not handling the new purchase token. Not retiring the old purchase token, leading to duplicate entitlements. Using `CHARGE_PRORATED_PRICE` for a downgrade (it will fail).

9. Pending Transaction Completion

What to test: A purchase that requires additional action before completing (such as cash based payment methods or parental approval).

Steps:

1. Enable pending transactions in your `BillingClient` setup.
2. Make a purchase using the "slow test card, always approves" payment method.
3. Verify that your `PurchasesUpdatedListener` receives a purchase with `PENDING` state.
4. Verify your app does not grant the entitlement yet.
5. Wait for the purchase to complete.
6. Verify that `PurchasesUpdatedListener` fires again with the completed purchase.
7. Acknowledge the purchase and grant the entitlement.

What can go wrong: Granting access for pending purchases. Not listening for the completion event. Not handling the case where a pending purchase is declined.

10. Restore After Cancellation

What to test: A user cancels their subscription but then re subscribes before the current period ends.

Steps:

1. Start with an active subscription.
2. Cancel it through the Play Store.
3. Before the period ends, open the Play Store subscription management and tap "Resubscribe".
4. Verify the subscription returns to active, auto renewing state.
5. Verify no gap in entitlement access.

What can go wrong: Your app showing a "buy" flow instead of directing the user to the Play Store to restore. Misinterpreting the restore as a new purchase.

11. Resubscribe After Expiration

What to test: A user's subscription has fully expired and they purchase again.

Steps:

1. Let a test subscription fully expire (cancel it and wait for the period to end).
2. Purchase the same subscription again.
3. Verify a new purchase token is issued.
4. Verify your app grants a fresh entitlement.
5. Verify your backend treats this as a new subscription, not a continuation of the old one.

What can go wrong: Trying to link the new purchase to the old one incorrectly. Not granting access because your system thinks the user already has an expired subscription for this product.

Testing Pending Transactions

Pending transactions deserve extra attention because they represent a fundamentally different purchase flow. In markets where cash based payment methods are common (Brazil, Japan, Indonesia, and others), a

significant portion of your purchases may go through a pending state before completing.

To enable pending transactions in your `BillingClient`:

```
val billingClient = BillingClient.newBuilder(context)
    .setListener(purchasesUpdatedListener)
    .enablePendingPurchases(
        PendingPurchasesParams.newBuilder()
            .enableOneTimeProducts()
            .enablePrepaidPlans()
            .build()
    )
    .build()
```

When you test pending transactions, verify these specific behaviors:

Your app does not grant access for pending purchases. Check the purchase state before granting entitlements:

```
if (purchase.purchaseState ==
    Purchase.PurchaseState.PURCHASED
) {
    // Grant entitlement and acknowledge
} else if (purchase.purchaseState ==
    Purchase.PurchaseState.PENDING
) {
    // Show "purchase pending" UI, no access yet
}
```

Your app updates when the purchase completes. The `PurchasesUpdatedListener` fires again when the pending purchase transitions to `PURCHASED` or is cancelled. Verify you handle both outcomes.

Your server handles pending RTDNs. If you use Real Time Developer Notifications, you receive a `ONE_TIME_PRODUCT_PURCHASED` or subscription notification when the purchase completes. Verify your server processes these correctly.

Your UI communicates the pending state clearly. Users need to know their purchase is not yet complete and what they need to do (for example, complete payment at a convenience store).

Testing Promo Codes

Promo codes let you give users free access to one time products or subscriptions. You create promo codes in the Google Play Console under **Monetize > Promo codes**.

To test promo code redemption:

1. Create a promo code campaign in the Play Console for your product.
2. Generate one or more codes.
3. On your test device, open the Google Play Store app and go to the Redeem code section.
4. Enter the promo code.
5. Verify that your app receives the purchase through `PurchasesUpdatedListener` or `queryPurchasesAsync`.
6. Verify that the purchase can be acknowledged and that your app grants the correct entitlement.

Promo code purchases behave like regular purchases from your app's perspective. The main difference is that the user does not go through your app's purchase flow. The purchase arrives via the Play Store, and your app discovers it the next time it queries purchases or receives an RTDN.

Test that your app handles this "out of band" purchase discovery. If your app only checks for purchases during its own purchase flow, promo code redemptions will be missed until the user restarts the app.

Testing in Different Regions

Billing behavior varies by region. Prices differ, available payment methods change, and some features (like pending transactions or specific offer types) may only apply in certain markets. You need to verify your app works correctly across your target regions.

Use Play Billing Lab to switch your test device's Play country and verify:

- **Prices display correctly.** Different countries have different currencies and number formats. Verify your UI handles long currency symbols, right to left currencies, and prices with varying decimal precision.
- **Payment methods are appropriate.** Some countries support carrier billing, cash payments, or region specific payment methods. Make sure your purchase flow does not break when users encounter these options.
- **Tax handling works.** Google handles tax calculation, but your receipts and UI may need to display tax information differently depending on the region.
- **Product availability is correct.** If you have restricted certain products to specific countries in the Play Console, verify that users in excluded countries do not see those products and that your app handles empty product lists gracefully.

You should test at minimum with one country from each of your major markets. If you sell globally, test with a US account, a European account (to verify EU specific requirements), and at least one account from a market where cash based payments are prevalent.

Testing with Real Payment Methods

Before your first production release, you should make at least one real purchase. Test purchases with license tester accounts cover most scenarios, but there are behaviors you can only verify with real money:

- **The actual payment flow.** Real payment methods go through Google's payment processing, which can behave differently from test instruments. Card verification, 3D Secure prompts, and payment provider timeouts only happen with real transactions.
- **Refund processing.** Request a refund through Google Play for your test purchase and verify that your app and server handle the refund notification correctly.
- **Receipt validation.** Verify that your server can validate real purchase tokens against the Google Play Developer API and that the response matches what you expect.
- **Financial reporting.** Check that the purchase appears in your Google Play Console financial reports with the correct amount, currency, and tax information.

To minimize cost, use your cheapest product for real payment testing. You can also refund yourself immediately after the purchase. Google processes refunds for self refunding within a few days, and you will only lose the small transaction fee.

One important note: real purchases require the app to be published on a track that the purchasing account has access to. An app only on the internal test track works, but the purchasing account must be part of the internal test track's tester list.

Testing BillingResult Response Codes with the Response Simulator

The Play Billing Library includes a response simulator that lets you force specific `BillingResult` response codes from any PBL method. This is invaluable for testing error handling paths that are difficult to trigger naturally.

To use the response simulator, you configure it through Play Billing Lab. You can set up rules that make specific API calls return specific response codes. For example, you can configure `launchBillingFlow` to return `ITEM_ALREADY_OWNED`, or `acknowledgePurchase` to return `NETWORK_ERROR`.

Here are the scenarios you should test with the response simulator:

`SERVICE_DISCONNECTED` **during purchase.** Force a disconnection and verify your app reconnects the `BillingClient` and retries the operation.

`NETWORK_ERROR` **on acknowledgment.** Force a network error when acknowledging a purchase and verify your app retries the acknowledgment. Remember, unacknowledged purchases are refunded after 3 days in production.

`ITEM_ALREADY_OWNED` **when launching a purchase.** This happens when the user already owns the product. Verify your app refreshes its purchase cache and shows appropriate messaging instead of displaying a confusing error.

`USER_CANCELED` **during the billing flow.** Verify your app returns gracefully to the previous screen without showing an error message. Cancellation is a normal user action, not an error.

`BILLING_UNAVAILABLE` **on connection.** This occurs when Google Play Billing is not available on the device (outdated Play Store, unsupported device, or restricted region). Verify your app hides or disables purchase options gracefully.

`DEVELOPER_ERROR` on launch. This means your parameters are wrong (mismatched product IDs, missing offer tokens). While you should catch these during development, verify your app does not crash and logs enough information for you to diagnose the issue.

`ERROR` (generic) on any operation. This is the catch all error. Verify your app shows a generic retry message and logs the debug message for troubleshooting.

Testing each of these response codes through the simulator takes minutes and saves you from shipping error handling code that has never actually been executed.

A Practical Testing Checklist

Bringing all of this together, here is a checklist you can follow before each release:

Setup:

- ☐ License tester account added in Play Console.
- ☐ App uploaded to internal test track.
- ☐ Test device signed in with license tester account.
- ☐ Play Billing Lab installed on test device.

Core Purchase Flows:

- ☐ New one time product purchase.
- ☐ New subscription purchase.
- ☐ Subscription renewal (wait for accelerated cycle).
- ☐ Upgrade with at least two replacement modes.
- ☐ Downgrade with at least one replacement mode.

Subscription Lifecycle:

- ☐ Grace period entry and recovery.
- ☐ Account hold entry and recovery.
- ☐ User cancellation with access until period end.
- ☐ Restore before expiration.
- ☐ Resubscribe after full expiration.
- ☐ Pause and resume.

Edge Cases:

- ☐ Pending transaction completion and decline.
- ☐ Unacknowledged purchase auto refund (3 minute test window).
- ☐ Promo code redemption.
- ☐ Purchase on one device, verify entitlement on another.

216

Error Handling:

- [ ] At least 4 response codes tested via response simulator.
- [ ] Network disconnection during purchase flow.
- [ ] App process killed during purchase and relaunched.

Regional:

- [ ] At least one non default country tested via Play Billing Lab.
- [ ] Currency formatting verified for target markets.

217

Chapter 17: Managing Your Product Catalog at Scale

Your product catalog is the foundation of your monetization strategy. Every subscription, one time product, base plan, and offer starts as an entry in that catalog. When your catalog contains a handful of products, managing it through the Play Console works fine. But as your app grows and you add regional pricing, seasonal promotions, multiple subscription tiers, and localized listings, the console becomes a bottleneck. You need automation.

Google provides a comprehensive set of server side APIs for managing your product catalog programmatically. This chapter covers those APIs in depth: how to create and update products in bulk, how to stay within quota limits, how to keep your local database in sync with Google Play, and how to handle the full lifecycle of a product from creation through deprecation.

Catalog Management APIs

The Google Play Developer API exposes two primary sets of endpoints for catalog management, split by product type.

One Time Products

One time products have two available API surfaces. The legacy `inappproducts` service and the newer `monetization.onetimeproducts` service. The newer service supports the expanded one time product object model, which includes multiple purchase options and offers per product. If you are starting fresh, use the `monetization.onetimeproducts` endpoints. If you have an existing catalog built on `inappproducts`, you can migrate products to the new model by sending an update through the `onetimeproducts` service. Once migrated, a product can no longer be accessed through the old `inappproducts` service.

The `monetization.onetimeproducts` service provides these methods:

- `create` creates a new one time product
- `patch` updates specific fields on an existing product
- `get` retrieves a single product by its product ID
- `list` retrieves all one time products for your app
- `batchGet` retrieves multiple products by their IDs in one call
- `batchUpdate` creates or updates up to 100 products in a single request
- `batchDelete` deletes up to 100 products in a single request
- `delete` deletes a single product

The legacy `inappproducts` service mirrors most of this functionality with `insert`, `update`, `patch`, `get`, `list`, `batchGet`, `batchUpdate`, `batchDelete`, and `delete`. It remains functional for backward compatibility, but new features only ship on the `monetization.onetimeproducts` surface.

Subscriptions

Subscription management lives under `monetization.subscriptions` and is organized into three layers that match the subscription product hierarchy you learned in Chapter 4.

Subscription level methods manage the top level container:

- `create`, `patch`, `get`, `list`, `batchGet`, `batchUpdate`, `delete`

Base Plan level methods manage billing terms within a subscription:

- `activate`, `deactivate`, `delete`, `batchUpdateStates`
- `migratePrices` migrates subscribers to the latest price version

Offer level methods manage promotional pricing attached to base plans:

- `create`, `patch`, `get`, `list`, `batchGet`, `batchUpdate`
- `activate`, `deactivate`, `batchUpdateStates`

This three layer structure means creating a fully configured subscription requires multiple steps. You create the subscription container first, then add base plans, then optionally attach offers to those base plans. The `batchUpdate` method at the subscription level can handle all of this in a single call when you include nested base plans and offers in the request body.

Batch Methods

Every product endpoint provides batch methods that accept up to 100 individual operations per request. This is the primary tool for managing catalogs at scale.

For one time products, `monetization.onetimeproducts.batchUpdate` accepts a list of up to 100 update requests. Each nested request specifies the product to create or update, along with configuration for how that individual update should behave. The `batchDelete` method works similarly, accepting up to 100 product identifiers to remove.

For subscriptions, `monetization.subscriptions.batchUpdate` accepts up to 100 subscription update requests. Each nested request can include the full subscription definition with base plans and offers, making it possible to create complete subscription products in bulk.

The batch methods offer two important advantages beyond convenience. First, they count as a single query against the per minute and hourly modification quotas (more on this shortly). A batch request containing 100 updates consumes the same quota as a single update in most cases. Second, they support the latency tolerance parameter, which unlocks higher throughput for large catalog operations.

Here is the general structure of a batch update call for one time products:

```
val batchRequest = BatchUpdateOneTimeProductsRequest()
batchRequest.requests = products.map { product ->
    UpdateOneTimeProductRequest().apply {
        oneTimeProduct = product
        allowMissing = true  // create if not exists
        latencyTolerance = "PRODUCT_UPDATE_LATENCY_TOLERANCE_LATENCY_TOLERANT"
    }
}

androidPublisher
    .monetization()
    .onetimeproducts()
    .batchUpdate(packageName, batchRequest)
    .execute()
```

If your catalog has more than 100 products, you chunk the list and send multiple batch requests. With latency tolerant mode, you can process thousands of products without hitting quota limits.

Latency Tolerance Options

Every modification method (create, update, patch, delete) supports a `latencyTolerance` field that controls how quickly your changes propagate to end users. This is a tradeoff between speed and throughput.

Latency sensitive (default). Changes propagate within minutes. Use this when you need updates to be visible quickly, such as fixing an incorrect price or activating a time sensitive promotion. This mode consumes quota from all three quota tiers, including the most restrictive one.

Latency tolerant. Changes may take up to 24 hours to propagate. Use this for bulk operations where immediate visibility is not required, such as initial catalog creation, periodic reconciliation, or large scale price updates across regions. This mode skips the most restrictive quota tier entirely, giving you much higher effective throughput.

You set latency tolerance at the individual request level. In a batch request, each nested operation can specify its own tolerance. This means you can mix latency sensitive and latency tolerant operations in the same batch, though doing so means the batch will consume quota from the sensitive tier for each sensitive operation it contains.

The practical guidance is straightforward. Default to latency tolerant for any bulk operation or background synchronization. Use latency sensitive only when a change must be live quickly, like correcting a pricing error or responding to a compliance issue.

Quota and Rate Limiting

The Google Play Developer API uses a three tier quota system for catalog modification endpoints. Understanding this system is essential for building reliable automation, because hitting quota limits means your updates will fail with HTTP 429 or 403 errors until the quota resets.

Tier 1: Per Minute Query Limit

All API calls, both reads and writes, share a per minute bucket. The default limit is 3,000 queries per minute for the publishing and monetization bucket. A single request counts as one query. A batch request also counts as one query, regardless of how many individual operations it contains. This means a batch of 100 updates costs the same as a single update against this tier.

Tier 2: Hourly Modification Limit

All product modification operations (create, update, patch, delete) share a single hourly limit of 7,200 queries per hour. This limit spans both one time products and subscriptions. A batch modification call counts as one query against this tier, regardless of the number of nested operations or their latency sensitivity. This is where batch methods provide their biggest advantage. Processing 7,200 individual updates would consume your entire hourly modification budget. Processing 72 batch requests of 100 items each achieves the same result while using only 72 queries of your 7,200 hourly limit.

Tier 3: Hourly Latency Sensitive Modification Limit

Latency sensitive modifications have an additional hourly limit of 7,200 per hour. Unlike Tier 2, this limit counts individual operations within a batch request. A single batch request containing 100 latency sensitive updates consumes 1 query from Tier 1, 1 query from Tier 2, and 100 queries from Tier 3. The same batch request with latency tolerant mode consumes 1 query from Tier 1, 1 query from Tier 2, and 0 queries from Tier 3.

This tiered system creates a clear incentive structure. Batch methods with latency tolerant mode give you the highest throughput. Here is how the math works for different approaches:

APPROACH	PRODUCTS PER HOUR	TIER 1	TIER 2	TIER 3
Individual latency sensitive calls	7,200	7,200	7,200	7,200
Individual latency tolerant calls	7,200	7,200	7,200	0
Batch of 100, latency sensitive	720,000	7,200	7,200	720,000 (over limit)
Batch of 100, latency tolerant	720,000	7,200	7,200	0

The last row is the sweet spot. Batches of 100 with latency tolerant mode let you modify up to 720,000 products per hour while staying within all quota limits. For most catalogs, this is more than enough.

If you hit quota limits, the API returns an error response:

221

```
{
  "code": 429,
  "errors": [{
    "domain": "usageLimits",
    "reason": "rateLimitExceeded",
    "message": "Rate Limit Exceeded"
  }]
}
```

Handle these errors with exponential backoff. Monitor your quota usage through the Google Cloud Console's Quotas section. If your default limits are insufficient, you can request an increase through Google Play's support portal.

Initial Catalog Creation Strategies

When you first set up programmatic catalog management, you need to populate Google Play with your entire product catalog. The strategy depends on catalog size.

Small Catalogs (Under 100 Products)

For small catalogs, you can use individual `create` calls or a single batch request. The `monetization.subscriptions.create` and `monetization.onetimeproducts.create` methods work well here. There is no need for latency tolerance optimization when you are creating fewer than 100 products.

For subscriptions, remember that base plans created through the API start in a DRAFT state. You must explicitly activate each base plan using `monetization.subscriptions.basePlans.activate` before users can purchase them. This is a common source of confusion. You create the subscription with its base plans, but nothing is available for purchase until you activate at least one base plan.

Medium Catalogs (100 to 10,000 Products)

Use `batchUpdate` with `allowMissing` set to `true` and latency tolerance set to `PRODUCT_UPDATE_LATENCY_TOLERANCE_LATENCY_TOLERANT`. The `allowMissing` flag tells the API to create the product if it does not exist, making the same call work for both creation and updates. Chunk your catalog into groups of 100 and send batch requests sequentially.

222

```
fun createCatalog(products: List<OneTimeProduct>) {
    products.chunked(100).forEach { chunk ->
        val batchRequest = BatchUpdateOneTimeProductsRequest()
        batchRequest.requests = chunk.map { product ->
            UpdateOneTimeProductRequest().apply {
                oneTimeProduct = product
                allowMissing = true
                latencyTolerance =
                    "PRODUCT_UPDATE_LATENCY_TOLERANCE_LATENCY_TOLERANT"
            }
        }
        androidPublisher.monetization()
            .onetimeproducts()
            .batchUpdate(packageName, batchRequest)
            .execute()
    }
}
```

At 100 products per batch and 7,200 batches per hour allowed against Tier 2, you can create all 10,000 products in under 2 minutes of wall clock time.

Large Catalogs (Over 10,000 Products)

The same batch approach works, but you should add pacing to stay safely within quota limits. Insert a short delay between batch requests if you are processing tens of thousands of products. Monitor your quota consumption and adjust the delay dynamically. For extremely large catalogs (100,000+ products), spread the initial creation over multiple hours or schedule it during off peak hours.

Also consider running a dry run first. Before sending creation requests, validate your product data locally. Check for duplicate product IDs, missing required fields (like listings and pricing), and invalid values. Catching errors before they hit the API saves both quota and debugging time.

Product Updates: patch, update, and batchUpdate

Once your catalog exists, you will need to update it. Prices change, listings get new translations, tax categories shift, and new offers launch. The API provides three distinct update mechanisms, each suited to different use cases.

patch

The `patch` method updates specific fields on an existing product without touching other fields. You specify which fields to modify using an `updateMask` parameter, a comma separated list of field paths. Only the fields listed in the mask are modified. Everything else stays the same.

223

```
// Update only the listings for a subscription
val subscription = Subscription().apply {
    listings = listOf(
        SubscriptionListing().apply {
            languageCode = "en-US"
            title = "Premium Plan (Updated)"
            description = "Access all premium features"
        }
    )
}

androidPublisher.monetization().subscriptions()
    .patch(packageName, productId, subscription)
    .setUpdateMask("listings")
    .execute()
```

Use `patch` when you are modifying a known set of fields and want to avoid accidentally overwriting other configuration. This is the safest update method for targeted changes.

For subscriptions, you must include `regionsVersion` in the request body when patching regional pricing. This version field ensures you are working with the correct region configuration and prevents conflicts.

update (inappproducts only)

The `update` method on the `inappproducts` endpoint replaces the entire product resource. You send a complete product definition, and it overwrites whatever was there before. It also supports `allowMissing`, which creates the product if it does not exist, making it an upsert operation.

This method is useful when your source of truth is a complete product definition, and you want to ensure Google Play matches it exactly. The risk is that any field you omit from the request body gets cleared or reset to defaults. Always send the full product definition when using `update`.

batchUpdate

The `batchUpdate` method combines batch processing with the update or patch behavior. Each nested request in the batch specifies its own update configuration. You can mix creates and updates in the same batch using the `allowMissing` flag.

For subscriptions, `batchUpdate` is particularly powerful because each nested request can include the full subscription hierarchy: the subscription container, its base plans, and attached offers. This means you can update a complex subscription product, including all its plans and offers, in a single nested request within a larger batch.

```
val batchRequest = BatchUpdateSubscriptionsRequest()
batchRequest.requests = subscriptions.map { sub ->
    UpdateSubscriptionRequest().apply {
        subscription = sub
        updateMask = "listings"
        latencyTolerance =
            "PRODUCT_UPDATE_LATENCY_TOLERANCE_LATENCY_TOLERANT"
    }
}

androidPublisher.monetization().subscriptions()
    .batchUpdate(packageName, batchRequest)
    .execute()
```

The general rule: use `patch` for targeted field updates on individual products, and use `batchUpdate` for bulk operations where you are modifying many products at once.

Catalog Reconciliation and Diff Synchronization

If you manage your catalog through an internal CMS, admin panel, or configuration system, you need a way to ensure Google Play's catalog matches your local source of truth. This process is called catalog reconciliation.

The core idea is simple: periodically read the full catalog from Google Play, compare it against your local catalog, compute the differences, and apply the necessary changes. In practice, there are several details to get right.

Step 1: Fetch the Remote Catalog

Use the `list` endpoints to retrieve every product from Google Play. For one time products, call `monetization.onetimeproducts.list` (or `inappproducts.list` for legacy products). For subscriptions, call `monetization.subscriptions.list`. Both endpoints support pagination. Keep fetching pages until you have all products.

```
fun fetchAllSubscriptions(): List<Subscription> {
    val allSubscriptions = mutableListOf<Subscription>()
    var pageToken: String? = null

    do {
        val response = androidPublisher
            .monetization().subscriptions()
            .list(packageName)
            .setPageToken(pageToken)
            .execute()

        response.subscriptions?.let {
            allSubscriptions.addAll(it)
        }
        pageToken = response.nextPageToken
    } while (pageToken != null)

    return allSubscriptions
}
```

Step 2: Build a Diff

Compare each product in your local catalog against the remote catalog. You are looking for three types of differences:

- **Missing remotely.** Products that exist in your local catalog but not on Google Play. These need to be created.
- **Missing locally.** Products that exist on Google Play but not in your local catalog. These may need to be deactivated or deleted, depending on your policy.
- **Divergent.** Products that exist in both places but with different values. These need to be updated.

For the divergent case, compare the specific fields you care about: pricing, listings, tax settings, offer configuration. You do not need to compare every field. Focus on the ones your system manages.

Step 3: Apply Changes

Group your changes into batch requests. Create missing products using `batchUpdate` with `allowMissing: true`. Update divergent products using `batchUpdate` with the appropriate `updateMask`. Handle deletions or deactivations separately.

226

```
fun reconcileCatalog(
    local: List<OneTimeProduct>,
    remote: List<OneTimeProduct>
) {
    val remoteMap = remote.associateBy { it.productId }
    val localMap = local.associateBy { it.productId }

    val toCreate = local.filter { it.productId !in remoteMap }
    val toUpdate = local.filter { product ->
        val remoteProd = remoteMap[product.productId]
        remoteProd != null && hasDifferences(product, remoteProd)
    }
    val toRemove = remote.filter { it.productId !in localMap }

    // Batch create and update
    (toCreate + toUpdate).chunked(100).forEach { chunk ->
        batchUpdateProducts(chunk, allowMissing = true)
    }

    // Handle removals per your policy
    toRemove.chunked(100).forEach { chunk ->
        batchDeleteProducts(chunk)
    }
}
```

Scheduling and Safety

Run reconciliation on a schedule that fits your update frequency. Daily reconciliation works well for most apps. If your catalog changes frequently throughout the day, run it more often.

Always log the diff before applying changes. In a large catalog, a bug in your diff logic could trigger mass deletions or overwrites. Build in safeguards: if the diff shows more than a certain percentage of products changing, flag it for manual review instead of applying automatically. A reconciliation run that wants to delete 50% of your catalog is almost certainly a bug, not a legitimate change.

Use latency tolerant mode for reconciliation. The changes do not need to propagate immediately, and the higher throughput keeps reconciliation fast even for large catalogs.

Product Deprecation and Deletion

Products have a lifecycle. At some point, you may need to retire a product because a feature is being discontinued, you are consolidating subscription tiers, or a product was created by mistake. The approach differs between one time products and subscriptions.

227

One Time Products

One time products can be deleted outright using `monetization.onetimeproducts.delete` or `monetization.onetimeproducts.batchDelete`. Deletion removes the product from your catalog permanently. Existing purchases are not affected, as users who already bought the product retain their entitlement. But the product will no longer appear in purchase flows or product queries.

Before deleting, consider whether any users might still need to restore their purchase on a new device. If your app checks the product catalog during purchase restoration, a deleted product could cause issues. A safer approach for many apps is to keep the product in the catalog but remove it from your storefront so new purchases cannot happen while existing entitlements remain restorable.

Subscriptions

Subscriptions are more constrained. You can only delete a subscription if no base plan has ever been activated. Once any base plan has been activated, even if you later deactivated it, the subscription cannot be deleted. This restriction exists because active subscribers may hold entitlements tied to that subscription, and deleting it would create inconsistencies.

For subscriptions you want to retire, the approach is to deactivate all base plans. Deactivated base plans stop accepting new subscribers but continue serving existing ones through their current billing cycle. Existing subscribers can still renew, but the plan is no longer visible to new users.

```
// Deactivate a base plan to stop new signups
androidPublisher.monetization().subscriptions()
    .basePlans()
    .deactivate(
        packageName,
        productId,
        basePlanId,
        DeactivateBasePlanRequest()
    )
    .execute()
```

For base plans still in DRAFT state (never activated), you can delete them using `monetization.subscriptions.basePlans.delete`. This is useful for cleaning up test configurations or correcting mistakes before going live.

The practical workflow for sunsetting a subscription:

1. Deactivate all base plans on the subscription.
2. Remove the subscription from your app's storefront and purchase flows.
3. Continue processing renewals for existing subscribers until they churn naturally.
4. Monitor active subscriber counts. When all subscribers have churned, the subscription is effectively retired.

228

Do not try to force migrate subscribers by deleting their subscription product. Instead, build migration flows in your app that encourage users to switch to the replacement product on their own terms.

Offers and Promotions

Beyond the standard catalog management APIs, Google Play provides a separate promotion system for distributing products and subscription trials through redeemable codes. This is useful for marketing campaigns, partnership deals, influencer programs, and customer support gestures.

One Time Use Codes

One time use codes are unique, auto generated codes that can each be redeemed once. You create them through the Play Console (not through the API). Each code is a distinct string that grants the user access to a one time product or a subscription free trial.

The quarterly limits depend on the product type:

- **One time products:** 500 codes per quarter across all managed products in your app. This is a shared pool. If you have 10 one time products, the 500 limit applies to all of them combined.
- **Subscriptions:** 10,000 codes per quarter per subscription product. Each subscription product gets its own independent limit.

Unused codes do not carry over to the next quarter. If you generate 500 codes for one time products and only distribute 200, the remaining 300 expire when the quarter ends. Plan your campaigns accordingly.

Users can redeem one time use codes either through the Google Play Store directly or within your app. When redeemed through the Play Store, the user is prompted to open your app (if installed) or download it. When redeemed in app, the purchase flows through your normal Play Billing Library integration.

Custom Codes

Custom codes are codes that you specify rather than having Google auto generate them. Unlike one time use codes, a single custom code can be redeemed multiple times up to a limit you set during creation.

Important restrictions on custom codes:

- Available only for subscriptions, not one time products.
- Only redeemable by users who have never previously subscribed to that product. This means custom codes are an acquisition tool, not a retention tool.
- Redemption limit must be between 2,000 and 99,999 per code.
- Users can only redeem custom codes in app, not through the Play Store.

Custom codes work well for partnership promotions where you want to give a specific, branded code (like "PARTNER2026") to a large audience. The redemption limit ensures you control how many users can take advantage of the promotion.

Deep Link Redemption

You can create a deep link that takes users directly to the Google Play redemption screen with the code pre populated:

```
https://play.google.com/redeem?code=YOUR_PROMO_CODE
```

When a user opens this link, the Play Store opens with the code field already filled in. If the user has the latest version of your app installed, they are prompted to open it after redemption. If not, they are prompted to download or update the app first.

This deep link format works in emails, push notifications, social media posts, QR codes on physical media, and anywhere else you can share a URL. It reduces friction compared to asking users to manually navigate to the Play Store and type in a code.

Creating Promotions in Practice

All promotion creation happens through the Play Console, not the API. Navigate to your app's Monetize section, select Promotions, and create a new promotion. You choose the promotion type (one time use or custom), select the product or subscription, set the promotion duration (up to one year), and specify the number of codes.

Once created, you can download the generated codes as a CSV file for one time use codes, or note the custom code string for distribution. The codes become active immediately unless you set a future start date.

A few practical tips:

- **Track redemption rates.** The Play Console shows how many codes have been redeemed, but integrate this with your own analytics to understand the downstream conversion and retention of promo code users.
- **Set expiration dates.** Every promotion should have an end date. Open ended promotions are hard to forecast and can create support issues when users try to redeem expired codes.
- **Test codes before distribution.** Generate a small batch first, redeem one yourself, and verify the full flow works correctly in your app before sending codes to thousands of users.
- **Plan around quarterly resets.** If you have a major campaign planned, ensure you have enough quota in the current quarter. The 500 per quarter limit for one time products is particularly tight for large campaigns.

Chapter 18: Alternative Billing and External Offers

Google Play's billing system has historically been the only way to sell digital goods in Play distributed apps. That changed. Regulatory pressure in South Korea, the European Economic Area (EEA), India, and the United States has pushed Google to open up alternative paths for processing payments. The Play Billing Library now offers several programs that let you present users with choices beyond Google Play's standard payment flow, or even handle billing entirely outside of it.

This chapter covers every alternative billing and external linking program available in PBL 8.x. You will learn how each program works, which APIs to call, what regions they apply to, and how to report transactions back to Google. These APIs evolve quickly, so pay attention to which PBL version introduced each feature.

When and Why Alternative Billing Applies

Alternative billing exists because regulators in several countries determined that requiring a single payment system in an app store constitutes anticompetitive behavior. Google responded by creating programs that give developers more flexibility in how they charge users for digital goods.

There are several distinct programs, each with different rules:

- **User choice billing**: You offer both Google Play's billing and your own billing system. The user picks which one to use at purchase time. Google renders a choice screen.
- **Alternative billing only**: You bypass Google Play's billing entirely and use only your own payment system. Google shows an information dialog to the user explaining the situation.
- **External offers**: You link users out of your app to a website where they can find deals on digital content or download apps.
- **External content links**: You direct users to an external website for digital content, without processing the payment inside the app.
- **External payment links**: You present users with a choice between Google Play billing and a developer provided billing option that opens an external payment page.

Why would you use these? The most common reason is to reduce service fees. Google charges a 4% lower service fee on transactions processed through an alternative billing system. For high volume apps, that adds up. You might also want to offer payment methods Google does not support, consolidate billing across platforms, or comply with regional regulations that require you to offer alternatives.

The trade off is real: you take on more responsibility. You handle payment processing, fraud detection, refunds, and compliance yourself. You also must report every external transaction back to Google within 24 hours using the `externaltransactions` API. If you miss that window, you risk policy violations.

Alternative Billing with User Choice

User choice billing gives your users a screen where they can pick between Google Play's billing and your own. Google renders and manages this choice screen, so you do not need to build the UI yourself.

Setting Up the BillingClient

To enable user choice billing, call `enableUserChoiceBilling()` on the `BillingClient.Builder` and pass a `UserChoiceBillingListener`. This listener fires when the user selects your alternative billing system instead of Google Play.

```
val userChoiceBillingListener =
    UserChoiceBillingListener { userChoiceDetails ->
        // User chose your billing system.
        // Send the token to your backend.
        val token = userChoiceDetails
            .externalTransactionToken
        val products = userChoiceDetails.products
        backend.startExternalPurchase(token, products)
    }

val billingClient = BillingClient.newBuilder(context)
    .setListener(purchasesUpdatedListener)
    .enablePendingPurchases(
        PendingPurchasesParams.newBuilder().build()
    )
    .enableUserChoiceBilling(userChoiceBillingListener)
    .build()
```

You still set a `PurchasesUpdatedListener` because when the user picks Google Play's billing, the standard purchase flow runs and results come through that listener as usual.

The Choice Screen Flow

When you call `launchBillingFlow()`, Google Play checks whether the user is in a supported country and whether you called `enableUserChoiceBilling()`. If both conditions are met, Google shows the choice screen. If the user is not in a supported country, the standard Google Play purchase dialog appears instead, and no choice screen is shown.

The flow has two outcomes:

1. **User picks Google Play**: The purchase proceeds normally. Your `PurchasesUpdatedListener.onPurchasesUpdated()` callback fires with the `BillingResult` and purchases.
2. **User picks your billing system**: The `UserChoiceBillingListener.userSelectedAlternativeBilling()` callback fires with a `UserChoiceDetails` object. This object contains the list of products the user wants to buy and an `externalTransactionToken` that you must send to your backend.

After the user picks your billing, you are responsible for the entire payment flow. Collect payment through your system, then report the transaction to Google's `externaltransactions` API within 24 hours using the token.

232

Subscription Upgrades and Downgrades

When a user who originally purchased through your alternative billing system wants to change their subscription plan, you can skip the choice screen by using `setOriginalExternalTransactionId()`:

```
val billingFlowParams = BillingFlowParams.newBuilder()
    .setProductDetailsParamsList(
        listOf(
            BillingFlowParams.ProductDetailsParams
                .newBuilder()
                .setProductDetails(newPlanDetails)
                .setOfferToken(newOfferToken)
                .build()
        )
    )
    .setSubscriptionUpdateParams(
        BillingFlowParams.SubscriptionUpdateParams
            .newBuilder()
            .setOriginalExternalTransactionId(
                originalExternalTxId
            )
            .build()
    )
    .build()
```

This tells Google Play that the user already chose alternative billing for the original subscription, so there is no need to ask again. A new `externalTransactionToken` is generated for the upgrade or downgrade transaction.

Version History

The user choice billing concept was introduced in PBL 5.2 under the name `enableAlternativeBilling` with an `AlternativeBillingListener`. In PBL 6.1, these were renamed to `enableUserChoiceBilling` and `UserChoiceBillingListener` for clarity. The old names still exist for backwards compatibility but are deprecated. In PBL 8.0, `UserChoiceDetails` replaced the deprecated `AlternativeChoiceDetails`. Use the newer names in any new integration.

Alternative Billing Only

Alternative billing only means you do not offer Google Play's billing at all. The user pays exclusively through your payment system. Google still requires you to show an information dialog that tells the user they are being billed outside of Google Play.

Setting Up the BillingClient

233

The setup is simpler than user choice billing. You call `enableAlternativeBillingOnly()` and do not need a `PurchasesUpdatedListener`, since no purchases will flow through Google Play.

```
val billingClient = BillingClient.newBuilder(context)
    .enableAlternativeBillingOnly()
    .build()
```

Checking Availability

Before starting a purchase, verify that alternative billing only is available for the current user:

```
billingClient.isAlternativeBillingOnlyAvailableAsync(
    object : AlternativeBillingOnlyAvailabilityListener {
        override fun onAlternativeBillingOnlyAvailabilityResponse(
            billingResult: BillingResult
        ) {
            if (billingResult.responseCode !=
                BillingResponseCode.OK) {
                // Not available. Fall back or show error.
                return
            }
            // Proceed with alternative billing.
        }
    }
)
```

A `BILLING_UNAVAILABLE` response means the user or your account is not eligible. Check your Play Console enrollment status.

Showing the Information Dialog

Before each transaction, you must call `showAlternativeBillingOnlyInformationDialog()`. This shows a Google managed dialog that informs the user they are paying through your system, not Google Play. The dialog content differs slightly between regions (US vs. EEA messaging).

234

```
billingClient
    .showAlternativeBillingOnlyInformationDialog(
        activity,
        object :
            AlternativeBillingOnlyInformationDialogListener {
            override fun onAlternativeBillingOnlyInformationDialogResponse(
                billingResult: BillingResult
            ) {
                when (billingResult.responseCode) {
                    BillingResponseCode.OK ->
                        proceedWithPurchase()
                    BillingResponseCode.USER_CANCELED ->
                        // Show dialog again next attempt
                        Unit
                    else -> handleError(billingResult)
                }
            }
        }
    )
```

The dialog typically does not show again once the user has acknowledged it on the same device. However, if the device cache is cleared, it will appear again. If the response is `USER_CANCELED`, call the dialog again on the next purchase attempt.

Generating the Transaction Token

Before completing a transaction, call `createAlternativeBillingOnlyReportingDetailsAsync()` to get an `externalTransactionToken`:

235

```
billingClient
    .createAlternativeBillingOnlyReportingDetailsAsync(
        object :
            AlternativeBillingOnlyReportingDetailsListener {
            override fun onAlternativeBillingOnlyTokenResponse(
                billingResult: BillingResult,
                details:
                    AlternativeBillingOnlyReportingDetails?
            ) {
                if (billingResult.responseCode !=
                    BillingResponseCode.OK) return
                val token =
                    details?.externalTransactionToken
                // Send to your backend
            }
        }
    )
```

Send this token to your backend. After the user completes payment through your system, report the transaction to Google's `externaltransactions` API within 24 hours.

Alternative billing only requires PBL 6.1 or higher. The APIs work in PBL 8.x without changes.

Regional Eligibility

Not every program is available everywhere. Regional regulations drive which options you can offer and to whom.

South Korea

South Korea was the first country to require alternative billing options, driven by the Telecommunications Business Act amendment in 2021. Developers can offer user choice billing to mobile and tablet users in South Korea. The service fee is reduced by 4% for transactions where users pay through the alternative billing system. Both gaming and non gaming apps are eligible. South Korea is not part of the user choice billing pilot; it operates under its own regulatory framework.

European Economic Area (EEA)

The EEA includes all EU member states plus Iceland, Liechtenstein, and Norway. As of March 2024, both gaming and non gaming apps are eligible for user choice billing and alternative billing only when serving users in the EEA. The EEA also has its own external offers program with a tiered service fee model. The fee structure includes a required Tier 1 ongoing service fee (10% on transactions) and an optional Tier 2 fee that covers additional Play services. Developers in the EEA also have access to external content links and external payment links.

236

India

All developers can offer an alternative billing system alongside Google Play's billing for Indian users making in app purchases on mobile phones and tablets. The 4% service fee reduction applies here as well. When reporting transactions for Indian users, you must include the `administrativeArea` (state or province) in the `userTaxAddress` because tax rates vary by state.

United States

The US programs emerged from the Epic Games v. Google settlement. Developers of mobile and tablet games can offer user choice billing to US users. The external content links program (PBL 8.2+) is available for apps serving US users, allowing you to link out to your website. The external payment links program (PBL 8.3+) enables a choice screen with a developer provided billing option for eligible apps. Google announced updated policies in December 2025 that affect these programs, so check the Play Console for the latest requirements and enrollment deadlines.

Checking Eligibility at Runtime

You do not need to hard code country lists. The PBL APIs handle eligibility checks for you. When you call `launchBillingFlow()` with user choice billing enabled, Google Play only shows the choice screen if the user is in a supported country. The `isBillingProgramAvailableAsync()` method for external programs returns `BILLING_UNAVAILABLE` if the user is not eligible. Let the APIs do the filtering.

External Offers: Configuration and In App Integration

The external offers program lets you direct users outside your app to a website where you offer deals on digital content or app downloads. This program was originally available through dedicated APIs (`enableExternalOffer` , `isExternalOfferAvailableAsync`), but PBL 8.2 replaced those with the unified `enableBillingProgram()` API.

Play Console Configuration

Before writing any code, enroll in the external offers program through the Play Console. You will need to:

1. Accept the program terms
2. Register any external URLs you plan to link to
3. Provide a subscription management link if you offer subscriptions
4. Select which countries to participate in

In App Integration (PBL 8.2+)

Initialize the `BillingClient` with `BillingProgram.EXTERNAL_OFFER` :

```
val billingClient = BillingClient.newBuilder(context)
    .enableBillingProgram(
        BillingProgram.EXTERNAL_OFFER
    )
    .build()
```

Check whether the user is eligible:

```
billingClient.isBillingProgramAvailableAsync(
    BillingProgram.EXTERNAL_OFFER,
    object : BillingProgramAvailabilityListener {
        override fun onBillingProgramAvailabilityResponse(
            billingResult: BillingResult,
            details: BillingProgramAvailabilityDetails
        ) {
            if (billingResult.responseCode !=
                BillingResponseCode.OK) return
            // External offers available for this user
        }
    }
)
```

Generate a token and launch the external link:

238

```
val params = BillingProgramReportingDetailsParams
    .newBuilder()
    .setBillingProgram(BillingProgram.EXTERNAL_OFFER)
    .build()

billingClient
    .createBillingProgramReportingDetailsAsync(
        params,
        object : BillingProgramReportingDetailsListener {
            override fun onCreateBillingProgramReportingDetailsResponse(
                billingResult: BillingResult,
                details: BillingProgramReportingDetails?
            ) {
                val token =
                    details?.externalTransactionToken
                // Persist token, then launch link
                launchOffer(token)
            }
        }
    )
```

To actually direct the user to your external offer, use `launchExternalLink()`:

```
val linkParams = LaunchExternalLinkParams.newBuilder()
    .setBillingProgram(BillingProgram.EXTERNAL_OFFER)
    .setLinkUri(Uri.parse("https://myapp.com/offer"))
    .setLaunchMode(
        LaunchExternalLinkParams.LaunchMode
            .LAUNCH_IN_EXTERNAL_BROWSER_OR_APP
    )
    .build()

billingClient.launchExternalLink(
    activity, linkParams, launchListener
)
```

The `LAUNCH_IN_EXTERNAL_BROWSER_OR_APP` mode lets Google Play handle opening the URL. If you need to control how the link opens (for example, in a WebView or a specific browser), use `CALLER_WILL_LAUNCH_LINK` instead.

Note that PBL 8.2.0 had a bug in `isBillingProgramAvailableAsync()` and `createBillingProgramReportingDetailsAsync()`. Use PBL 8.2.1 or later.

239

External Content Links (PBL 8.2+)

External content links allow you to direct users outside your app to a website that offers digital content or app downloads. This program is currently available for apps serving users in the US.

How It Differs from External Offers

External content links and external offers use the same APIs but different `BillingProgram` constants. The distinction is primarily a policy and program enrollment difference. External content links use `BillingProgram.EXTERNAL_CONTENT_LINK`.

Integration

The integration pattern is identical to external offers, with the constant swapped:

```
val billingClient = BillingClient.newBuilder(context)
    .enableBillingProgram(
        BillingProgram.EXTERNAL_CONTENT_LINK
    )
    .build()
```

Check availability with `isBillingProgramAvailableAsync()`:

```
billingClient.isBillingProgramAvailableAsync(
    BillingProgram.EXTERNAL_CONTENT_LINK,
    object : BillingProgramAvailabilityListener {
        override fun onBillingProgramAvailabilityResponse(
            billingProgram: Int,
            billingResult: BillingResult
        ) {
            if (billingResult.responseCode !=
                BillingResponseCode.OK) return
            // External content links available
        }
    }
)
```

Generate the transaction token and launch the link using the same `createBillingProgramReportingDetailsAsync()` and `launchExternalLink()` methods shown in the external offers section, replacing the `BillingProgram` constant with `EXTERNAL_CONTENT_LINK`.

One important detail: for app download links, the target URL must be registered and approved in the Play Developer Console. Do not include personally identifiable information in the URI. Perform any cleanup

operations before launching the link, because users may not return to your app after the external browser opens.

External Payment Links (PBL 8.3+)

External payment links represent the newest alternative billing program. Introduced in PBL 8.3.0 (released December 2025), this program lets you present users with a choice between Google Play's billing system and a developer provided billing option that opens an external payment page. Unlike user choice billing, which predates PBL 8, external payment links use a more structured API with dedicated parameter classes.

New APIs in PBL 8.3

PBL 8.3.0 introduced several new classes:

- `BillingProgram.EXTERNAL_PAYMENTS`: The program constant
- `EnableBillingProgramParams`: Configuration for enabling the program
- `DeveloperBillingOptionParams`: Parameters for your external payment option
- `DeveloperProvidedBillingListener`: Callback for when the user picks your billing
- `DeveloperProvidedBillingDetails`: Details passed to your listener

Setting Up the BillingClient

The setup uses `EnableBillingProgramParams` to bundle the program type and listener:

```
val devBillingListener =
    DeveloperProvidedBillingListener { details ->
        // User chose your payment option.
        // Process with your billing system.
        backend.handleExternalPayment(details)
    }

val billingClient = BillingClient.newBuilder(context)
    .setListener(purchasesUpdatedListener)
    .enablePendingPurchases(
        PendingPurchasesParams.newBuilder().build()
    )
    .enableBillingProgram(
        EnableBillingProgramParams.newBuilder()
            .setBillingProgram(
                BillingProgram.EXTERNAL_PAYMENTS
            )
            .setDeveloperProvidedBillingListener(
                devBillingListener
            )
            .build()
    )
    .build()
```

You still provide a `PurchasesUpdatedListener` because users who choose Google Play billing go through the standard flow.

Checking Availability and Generating Tokens

These steps mirror the pattern from external offers and external content links:

```
billingClient.isBillingProgramAvailableAsync(
    BillingProgram.EXTERNAL_PAYMENTS,
    object : BillingProgramAvailabilityListener {
        override fun onBillingProgramAvailabilityResponse(
            billingProgram: Int,
            billingResult: BillingResult
        ) {
            if (billingResult.responseCode !=
                BillingResponseCode.OK) return
            // External payments available
        }
    }
)
```

Generate a token immediately before launching the billing flow:

```
val params = BillingProgramReportingDetailsParams
    .newBuilder()
    .setBillingProgram(BillingProgram.EXTERNAL_PAYMENTS)
    .build()

billingClient
    .createBillingProgramReportingDetailsAsync(params,
        object : BillingProgramReportingDetailsListener {
            override fun onCreateBillingProgramReportingDetailsResponse(
                billingResult: BillingResult,
                details: BillingProgramReportingDetails?
            ) {
                val token =
                    details?.externalTransactionToken
                // Store token, build billing flow
            }
        }
    )
```

Launching the Billing Flow with a Developer Option

When you launch the billing flow, attach a `DeveloperBillingOptionParams` to present your payment option alongside Google Play's:

```
val devBillingOption = DeveloperBillingOptionParams
    .newBuilder()
    .setBillingProgram(BillingProgram.EXTERNAL_PAYMENTS)
    .setLinkUri(
        Uri.parse("https://myapp.com/checkout")
    )
    .setLaunchMode(
        DeveloperBillingOptionParams.LaunchMode
            .LAUNCH_IN_EXTERNAL_BROWSER_OR_APP
    )
    .build()
```

Include this in your `BillingFlowParams` alongside the product details. Google Play renders a choice screen. If the user picks Google Play, the standard flow runs. If they pick your option, the `DeveloperProvidedBillingListener` fires.

When the Choice Screen Appears

The choice screen only appears when all three conditions are met: the user is in a supported country, you enabled external payments on the `BillingClient`, and you provided `DeveloperBillingOptionParams` in the billing flow. If any condition is missing, the standard Google Play billing dialog shows instead.

External Transaction Tokens and IDs

Every alternative billing and external linking program requires you to generate transaction tokens and report transactions. Understanding the difference between tokens and IDs is important.

External Transaction Token

The `externalTransactionToken` is a string generated by Google Play through the PBL. You obtain it by calling one of these methods, depending on the program:

- `createAlternativeBillingOnlyReportingDetailsAsync()` for alternative billing only
- `createBillingProgramReportingDetailsAsync()` for external offers, external content links, and external payment links (PBL 8.2+)

For user choice billing, the token comes directly from `UserChoiceDetails.externalTransactionToken` when the user selects alternative billing.

The token ties the transaction to a specific user and device context. You send it to your backend and include it when reporting the initial transaction to Google's API. Generate a new token before each transaction. Do not cache or reuse tokens across different transactions.

External Transaction ID

The `externalTransactionId` is a string you generate yourself. It uniquely identifies each transaction in your system. You provide it when reporting transactions to the `externaltransactions` API. For subscription renewals, each renewal gets its own `externalTransactionId`, but references the initial transaction's ID via the `initialExternalTransactionId` field.

Do not include personally identifiable information in the `externalTransactionId`.

Generating Tokens with the Unified API (PBL 8.2+)

For programs that use `createBillingProgramReportingDetailsAsync()`, the call looks the same regardless of the program. Only the `BillingProgram` constant changes:

```
val params = BillingProgramReportingDetailsParams
    .newBuilder()
    .setBillingProgram(
        BillingProgram.EXTERNAL_OFFER
        // or EXTERNAL_CONTENT_LINK
        // or EXTERNAL_PAYMENTS
    )
    .build()

billingClient
    .createBillingProgramReportingDetailsAsync(
        params, reportingDetailsListener
    )
```

This is one of the cleanest parts of PBL 8.2's redesign: a single method handles token generation for all external programs.

Backend Integration for Outside GPB Transactions

Once you collect payment outside of Google Play, you must report it to Google. This is not optional. Google uses these reports to calculate service fees, track compliance, and maintain records.

The externaltransactions API

All external transactions are reported through the Google Play Developer API's `externaltransactions` resource at:

```
POST /androidpublisher/v3/applications/{packageName}/externalTransactions
```

You provide the `externalTransactionId` as a query parameter and the transaction details in the request body.

245

Reporting an Initial Purchase

For a new subscription purchased through your billing system:

```
{
  "originalPreTaxAmount": {
    "priceMicros": "4990000",
    "currency": "USD"
  },
  "originalTaxAmount": {
    "priceMicros": "449100",
    "currency": "USD"
  },
  "transactionTime": "2026-01-15T10:30:00Z",
  "recurringTransaction": {
    "externalTransactionToken": "token_from_pbl",
    "externalSubscription": {
      "subscriptionType": "RECURRING"
    }
  },
  "userTaxAddress": {
    "regionCode": "US"
  }
}
```

Prices use `priceMicros`, which represent one millionth of the currency unit. Multiply the price by 1,000,000: $4.99 becomes `4990000`. Double check your math here, as incorrect values will cause invoicing problems.

Reporting Renewals

For subscription renewals, use a new `externalTransactionId` and reference the original:

```
{
  "originalPreTaxAmount": {
    "priceMicros": "4990000",
    "currency": "USD"
  },
  "originalTaxAmount": {
    "priceMicros": "449000",
    "currency": "USD"
  },
  "transactionTime": "2026-02-15T10:30:00Z",
  "recurringTransaction": {
    "initialExternalTransactionId": "first-txn-id",
    "externalSubscription": {
      "subscriptionType": "RECURRING"
    }
  },
  "userTaxAddress": {
    "regionCode": "US"
  }
}
```

Notice the renewal does not include the `externalTransactionToken`. Only the initial transaction uses the token. Renewals reference the initial transaction by its `externalTransactionId`.

Reporting Refunds

Report refunds by calling the refund endpoint on the specific `externalTransactionId`:

```
POST /androidpublisher/v3/applications/{packageName}/externalTransactions/{externalTransactionId}:refund
```

You can report full or partial refunds. For partial refunds, specify the pre tax amount being refunded.

Timing

Report transactions within 24 hours of the purchase completing. For subscriptions, report each renewal within 24 hours of the renewal charge succeeding. Late reporting can result in policy violations.

API Quotas

The `externaltransactions` API has a quota of 1,200 queries per minute (QPM) for `createexternaltransaction` and `refundexternaltransaction` calls. The `getexternaltransaction` call is excluded from this limit. If you are migrating a large number of existing subscriptions, spread the work across multiple days or request a quota increase from Google.

247

Tax Handling for India

When reporting transactions for users in India, include the administrative area (state or province) because tax rates vary:

```
{
  "userTaxAddress": {
    "regionCode": "IN",
    "administrativeArea": "KARNATAKA"
  }
}
```

Payment Method Image Asset Requirements

If you use user choice billing, Google displays your alternative billing option on the choice screen alongside Google Play's option. Your listing shows your app icon, app name, and an image of your accepted payment methods. Google has specific requirements for this image.

Specifications

PROPERTY	REQUIREMENT
Asset size	192dp x 20dp
Format	PNG with transparent background
Card size	32dp x 20dp per card
Card spacing	8dp between cards
Inner padding	3dp within each card
Card outline	1dp inner stroke, 2dp corner radius, color #E0E0E0
Card background	Solid, preferably white
Maximum cards	5 payment method icons

Guidelines

- Include only payment method logos in the image (Visa, Mastercard, PayPal, etc.)
- Do not add text, promotional messaging, or other graphics
- Use recognizable, standard logos for each payment method
- Upload the asset through the Play Console under your alternative billing configuration
- Images that do not meet these requirements will not be displayed on the choice screen

The payment method image is important for user trust. Users see this alongside Google Play's recognizable brand, so make sure your accepted methods are clearly visible and professional.

249

Appendix A: Complete RTDN Reference Table

This appendix provides a complete reference for all Real Time Developer Notification (RTDN) types. Use this as a quick lookup when building your notification handler.

RTDN Message Structure

Every RTDN message arrives as a Cloud Pub/Sub message with a base64 encoded JSON payload in the `data` field. The decoded JSON has this structure:

```
// Top-level DeveloperNotification
{
  "version": "1.0",
  "packageName": "com.example.app",
  "eventTimeMillis": "1234567890123",
  // One of the following (mutually exclusive):
  "subscriptionNotification": { ... },
  "oneTimeProductNotification": { ... },
  "voidedPurchaseNotification": { ... },
  "testNotification": { ... }
}
```

Only one notification field is present per message.

Subscription Notification Types

CODE	TYPE	TRIGGER	SUBSCRIPTIO STATE AFTER
1	`SUBSCRIPTION_RECOVERED`	Payment recovered from grace period, account hold, or pause resumes	`ACTIVE`
2	`SUBSCRIPTION_RENEWED`	Active subscription successfully renewed	`ACTIVE`
3	`SUBSCRIPTION_CANCELED`	Subscription canceled (user, developer, or system)	`CANCELED`
4	`SUBSCRIPTION_PURCHASED`	New subscription purchased	`ACTIVE`
5	`SUBSCRIPTION_ON_HOLD`	Grace period expired, entered account hold	`ON_HOLD`
6	`SUBSCRIPTION_IN_GRACE_PERIOD`	Renewal payment failed, entered grace period	`IN_GRACE_PE`
7	`SUBSCRIPTION_RESTARTED`	User restored canceled subscription from Play Store before expiration	`ACTIVE`
8	`SUBSCRIPTION_PRICE_CHANGE_CONFIRMED`	(Deprecated) User confirmed a price change	No state chan
9	`SUBSCRIPTION_DEFERRED`	Subscription billing date extended via `subscriptionsv2.defer`	`ACTIVE`
10	`SUBSCRIPTION_PAUSED`	Subscription paused (effective at end of current period)	`PAUSED`
11	`SUBSCRIPTION_PAUSE_SCHEDULE_CHANGED`	Pause schedule modified (pause set, changed, or removed)	Varies
12	`SUBSCRIPTION_REVOKED`	Subscription revoked (refund, developer revocation)	`EXPIRED`
13	`SUBSCRIPTION_EXPIRED`	Subscription expired after cancellation or account hold	`EXPIRED`

CODE	TYPE	TRIGGER	SUBSCRIPTIO STATE AFTER
		failure	
17	`SUBSCRIPTION_ITEMS_CHANGED`	Items in subscription bundle changed (add ons)	Varies
18	`SUBSCRIPTION_CANCELLATION_SCHEDULED`	Installment subscription cancellation scheduled at end of commitment	`ACTIVE` (unt commitment e
19	`SUBSCRIPTION_PRICE_CHANGE_UPDATED`	Price change details updated for existing subscribers	Varies
20	`SUBSCRIPTION_PENDING_PURCHASE_CANCELED`	Pending subscription purchase was canceled	`EXPIRED`
22	`SUBSCRIPTION_PRICE_STEP_UP_CONSENT_UPDATED`	Price step up consent period begun or user provided consent (South Korea)	Varies

Note: Notification type codes 14, 15, 16, and 21 are not assigned. Do not expect to receive these values.

Subscription Notification Payload

```
{
  "version": "1.0",
  "notificationType": 4,
  "purchaseToken": "purchase-token-string",
  "subscriptionId": "premium_monthly"
}
```

One Time Product Notification Types

CODE	TYPE	TRIGGER	REQUIRED ACTION
1	`ONE_TIME_PRODUCT_PURCHASED`	One time product purchase completed	Verify purchase. Grant access. Acknowledge or consume.
2	`ONE_TIME_PRODUCT_CANCELED`	Pending one time product purchase was canceled	Do not grant access. Clean up pending records.

One Time Product Notification Payload

```
{
  "version": "1.0",
  "notificationType": 1,
  "purchaseToken": "purchase-token-string",
  "sku": "remove_ads"
}
```

Note that one time product notifications include the `sku` field (the product ID), while subscription notifications include `subscriptionId`.

Voided Purchase Notification

Voided purchase notifications are sent when a purchase is voided due to a refund, chargeback, or revocation.

```
{
  "purchaseToken": "purchase-token-string",
  "orderId": "GPA.1234-5678-9012-34567",
  "productType": 1,
  "refundType": 1
}
```

FIELD	VALUES
`productType`	`1` = Subscription, `2` = One time product
`refundType`	`1` = Full refund, `2` = Quantity based refund

When you receive a voided purchase notification, revoke access immediately and call the Voided Purchases API for full details.

Test Notification

```
{
  "version": "1.0"
}
```

Test notifications have only a `version` field inside the `testNotification` object. Your handler should acknowledge these without processing them as real events.

Handler Pattern

Every RTDN handler should follow this pattern:

1. Receive the Pub/Sub message
2. Decode the base64 `data` field
3. Parse the JSON `DeveloperNotification`
4. Identify the notification type (subscription, one time, voided, or test)
5. Extract the `purchaseToken`
6. **Call the Google Play Developer API** to get the full, current state
7. Update your database based on the API response (not the notification alone)
8. Acknowledge the Pub/Sub message

Never rely solely on the RTDN payload for business logic. RTDNs signal that something changed. The API tells you what the current state is. Always call the API.

254

Appendix B: BillingResult Response Code Reference

This appendix provides a complete reference for all `BillingResponseCode` values returned by the Play Billing Library, along with the sub response codes introduced in PBL 8.0.

Response Codes

255

CODE	VALUE	RETRIABLE	DESCRIPTION	RECOMMENDED STRATEGY
`OK`	0	N/A	Operation succeeded	Process the result normally.
`USER_CANCELED`	1	No	User dismissed the purchase dialog	No action needed. Do not show message.
`SERVICE_UNAVAILABLE`	2	Yes	Google Play service is temporarily unavailable	Retry with exponential backoff (2x factor, max 3 attempts).
`BILLING_UNAVAILABLE`	3	No (auto)	Billing is unavailable on this device or for this user	Do not auto retry. Check Play St version and user account. Let th try again manually.
`ITEM_UNAVAILABLE`	4	No	The requested product is not available for purchase	Refresh product details with `queryProductDetailsAsync` product may have been deactiv
`DEVELOPER_ERROR`	5	No	Invalid arguments passed to the API	Fix your code. Check product ID tokens, and parameter construc This is a programming error.
`ERROR`	6	Yes	An internal Google Play error occurred	Retry with exponential backoff. persistent, report to Google.
`ITEM_ALREADY_OWNED`	7	No	The user already owns this non consumable item	Call `queryPurchasesAsync()` refresh your local purchase cac user may have purchased on an device.
`ITEM_NOT_OWNED`	8	No	Attempted to consume or acknowledge an item the user does not own	Call `queryPurchasesAsync()` refresh your local purchase cac purchase may have been refunc
`NETWORK_ERROR`	12	Yes	A network error occurred during the operation	Retry with a short delay (1-2 se Check device connectivity.
`SERVICE_DISCONNECTED`	-1	Yes	The `BillingClient` is not connected to Google Play Services	With `enableAutoServiceReconne` (PBL 8), the library retries auto Without it, call `startConnect` and retry.

CODE	VALUE	RETRIABLE	DESCRIPTION	RECOMMENDED STRATEGY
`FEATURE_NOT_SUPPORTED`	-2	No	The requested feature is not supported on this device or Play Store version	Check `isFeatureSupported(` before calling feature specific A Provide fallback behavior.
`SERVICE_TIMEOUT`	-3	Yes (Deprecated)	The request timed out	Deprecated in recent PBL versi as `SERVICE_UNAVAILABLE` an with backoff.

Sub Response Codes (PBL 8.0+)

Sub response codes provide more granular information about why a purchase failed. They are only available in the `PurchasesUpdatedListener.onPurchasesUpdated()` callback, retrieved via `BillingResult.getOnPurchasesUpdatedSubResponseCode()`.

SUB RESPONSE CODE	DESCRIPTION	WHEN IT APPEARS	RECOMMENDED ACTION
`NO_APPLICABLE_SUB_RESPONSE_CODE`	No specific sub response applies	Default value for most responses	Handle based on the main response code only.
`PAYMENT_DECLINED_DUE_TO_INSUFFICIENT_FUNDS`	User's available funds are less than the purchase price	When the main response indicates a payment failure	Show a message suggesting the user check their payment method balance.
`USER_INELIGIBLE`	User does not meet eligibility requirements for an offer	When the user attempts to use an offer they are not eligible for	Show standard pricing without the offer. Check your offer eligibility logic.

Accessing Sub Response Codes

```
val listener = PurchasesUpdatedListener {
    billingResult, purchases ->
    if (billingResult.responseCode !=
        BillingResponseCode.OK
    ) {
        val subCode = billingResult
            .onPurchasesUpdatedSubResponseCode
        when (subCode) {
            OnPurchasesUpdatedSubResponseCode
                .PAYMENT_DECLINED_DUE_TO_INSUFFICIENT_FUNDS
            -> showInsufficientFundsMessage()
            OnPurchasesUpdatedSubResponseCode
                .USER_INELIGIBLE
            -> showStandardPricing()
            else -> handleGeneralError(billingResult)
        }
    }
}
```

Error Handling Decision Tree

Use this decision tree when you receive a `BillingResult`:

1. **Is `responseCode` == `OK`?**
    - Yes: Process the result. Done.
    - No: Continue.
2. **Is `responseCode` == `USER_CANCELED`?**
    - Yes: Do nothing. The user chose to cancel. Done.
    - No: Continue.
3. **Is the error retriable?** (`SERVICE_UNAVAILABLE`, `ERROR`, `NETWORK_ERROR`, `SERVICE_DISCONNECTED`, `SERVICE_TIMEOUT`)
    - Yes: Retry with appropriate strategy (simple retry for network errors, exponential backoff for service errors). If all retries fail, show an error message.
    - No: Continue.
4. **Is it a cache sync issue?** (`ITEM_ALREADY_OWNED`, `ITEM_NOT_OWNED`)
    - Yes: Call `queryPurchasesAsync()` to refresh your purchase cache. Update UI accordingly.
    - No: Continue.
5. **Is it a product issue?** (`ITEM_UNAVAILABLE`)

- Yes: Refresh product details. The product may no longer be available.
- No: Continue.

6. **Is it a device/environment issue?** (`BILLING_UNAVAILABLE` , `FEATURE_NOT_SUPPORTED`)
   - Yes: Show a user friendly message. Do not retry automatically. Let the user initiate a retry.
   - No: Continue.
7. **Is it a developer error?** (`DEVELOPER_ERROR`)
   - Yes: Log the error with full details. Fix the API usage in your code. This should not happen in production.

Retry Implementation Summary

STRATEGY	BASE DELAY	FACTOR	MAX ATTEMPTS	USE FOR
Simple retry	1-2 seconds	1x (fixed)	3	`NETWORK_ERROR`
Exponential backoff	2 seconds	2x	3	`SERVICE_UNAVAILABLE` , `ERROR` , `SERVICE_TIMEOUT`
Auto reconnect	Handled by PBL 8	N/A	N/A	`SERVICE_DISCONNECTED`
User initiated	N/A	N/A	N/A	`BILLING_UNAVAILABLE`
No retry	N/A	N/A	N/A	`USER_CANCELED` , `DEVELOPER_ERROR` , `FEATURE_NOT_SUPPORTED`

Appendix C: Google Play Developer API Quick Reference

This appendix provides a quick reference for all Google Play Developer API endpoints related to billing. The base URL for all endpoints is:

```
https://androidpublisher.googleapis.com/androidpublisher/v3/
```

All requests require OAuth 2.0 authentication with the scope:

```
https://www.googleapis.com/auth/androidpublisher
```

Subscription Purchases

ENDPOINT	METHOD	PATH
`purchases.subscriptionsv2.get`	GET	`applications/{packageName}/purchases/subs`
`purchases.subscriptionsv2.revoke`	POST	`applications/{packageName}/purchases/subs`
`purchases.subscriptionsv2.defer`	POST	`applications/{packageName}/purchases/subs`
`purchases.subscriptionsv2.cancel`	POST	`applications/{packageName}/purchases/subs`
`purchases.subscriptions.get`	GET	`applications/{packageName}/purchases/subs`
`purchases.subscriptions.acknowledge`	POST	`applications/{packageName}/purchases/subs`

SubscriptionPurchaseV2 Response (Key Fields)

FIELD	TYPE	DESCRIPTION
`kind`	string	Always `"androidpublisher#subscriptionPurchaseV2"`
`subscriptionState`	string	Current state: `ACTIVE`, `CANCELED`, `IN_GRACE_PERIOD`, `ON_HOLD`, `PAUSED`, `EXPIRED`, `PENDING`
`lineItems[]`	array	One entry per base plan/offer in the subscription
`lineItems[].productId`	string	The subscription product ID
`lineItems[].expiryTime`	timestamp	When this line item expires
`lineItems[].autoRenewingPlan`	object	Present for auto renewing plans. Contains `autoRenewEnabled`.
`lineItems[].prepaidPlan`	object	Present for prepaid plans. Contains `allowExtendAfterTime`.
`lineItems[].offerDetails`	object	Offer applied to this line item. Contains `offerTags[]`, `basePlanId`.
`linkedPurchaseToken`	string	If this purchase replaces another, the token of the replaced purchase.
`startTime`	timestamp	When the subscription was first purchased.
`regionCode`	string	ISO 3166-1 alpha-2 country code of the user.
`canceledStateContext`	object	Why the subscription was canceled. Contains `userInitiatedCancellation`, `systemInitiatedCancellation`, or `developerInitiatedCancellation`.
`acknowledgementState`	string	`ACKNOWLEDGEMENT_STATE_PENDING` or `ACKNOWLEDGEMENT_STATE_ACKNOWLEDGED`
`externalAccountIdentifiers`	object	Contains `obfuscatedExternalAccountId` and `obfuscatedExternalProfileId`.
`pausedStateContext`	object	Present when paused. Contains `autoResumeTime`.

One Time Product Purchases

ENDPOINT	METHOD	PATH
`purchases.products.get`	GET	`applications/{packageName}/purchase`
`purchases.productsv2.getproductpurchasev2`	GET	`applications/{packageName}/purchase`
`purchases.products.acknowledge`	POST	`applications/{packageName}/purchase`
`purchases.products.consume`	POST	`applications/{packageName}/purchase`

ProductPurchase Response (Key Fields)

FIELD	TYPE	DESCRIPTION
`kind`	string	Always `"androidpublisher#productPurchase"`
`purchaseTimeMillis`	long	Time of purchase in milliseconds since epoch.
`purchaseState`	integer	`0` = Purchased, `1` = Canceled, `2` = Pending
`consumptionState`	integer	`0` = Not consumed, `1` = Consumed
`orderId`	string	The order ID for this transaction.
`acknowledgementState`	integer	`0` = Not acknowledged, `1` = Acknowledged
`purchaseToken`	string	The purchase token.
`productId`	string	The product ID.
`quantity`	integer	Quantity purchased (for multi quantity purchases).
`obfuscatedExternalAccountId`	string	Obfuscated account ID set during purchase.
`obfuscatedExternalProfileId`	string	Obfuscated profile ID set during purchase.
`regionCode`	string	ISO 3166-1 alpha-2 country code.

Orders

ENDPOINT	METHOD	PATH	DESCRIPT
`orders.refund`	POST	`applications/{packageName}/orders/{orderId}:refund`	Issue a full refund. Available within 3 ye of purchas

Voided Purchases

ENDPOINT	METHOD	PATH
`purchases.voidedpurchases.list`	GET	`applications/{packageName}/purchases/voidedpur`

Query Parameters for Voided Purchases

263

PARAMETER	TYPE	DESCRIPTION
`startTime`	long	Milliseconds since epoch. Only return purchases voided after this time.
`endTime`	long	Milliseconds since epoch. Only return purchases voided before this time.
`startIndex`	integer	Pagination offset.
`maxResults`	integer	Maximum results per page (default 1000, max 1000).
`type`	integer	`0` = Both, `1` = Subscriptions only, `2` = One time products only
`includeQuantityBasedPartialRefund`	boolean	Include partial quantity based refunds.

Catalog Management: Subscriptions

ENDPOINT	METHOD	PATH
`monetization.subscriptions.create`	POST	`applications/{packageName}/subscriptio`
`monetization.subscriptions.get`	GET	`applications/{packageName}/subscriptio`
`monetization.subscriptions.list`	GET	`applications/{packageName}/subscriptio`
`monetization.subscriptions.patch`	PATCH	`applications/{packageName}/subscriptio`
`monetization.subscriptions.delete`	DELETE	`applications/{packageName}/subscriptio`
`monetization.subscriptions.batchGet`	GET	`applications/{packageName}/subscriptio`
`monetization.subscriptions.batchUpdate`	POST	`applications/{packageName}/subscriptio`

Catalog Management: Base Plans

ENDPOINT	METHOD	PATH
`monetization.subscriptions.basePlans.activate`	POST	`...basePlans/{basePla`
`monetization.subscriptions.basePlans.deactivate`	POST	`...basePlans/{basePla`
`monetization.subscriptions.basePlans.delete`	DELETE	`...basePlans/{basePla`
`monetization.subscriptions.basePlans.migratePrices`	POST	`...basePlans/{basePla`
`monetization.subscriptions.basePlans.batchMigratePrices`	POST	`...basePlans:batchMig`
`monetization.subscriptions.basePlans.batchUpdateStates`	POST	`...basePlans:batchUpd`

Catalog Management: Offers

ENDPOINT	METHOD	PATH
`monetization.subscriptions.basePlans.offers.create`	POST	`...offers`
`monetization.subscriptions.basePlans.offers.get`	GET	`...offers/{offe`
`monetization.subscriptions.basePlans.offers.list`	GET	`...offers`
`monetization.subscriptions.basePlans.offers.patch`	PATCH	`...offers/{offe`
`monetization.subscriptions.basePlans.offers.delete`	DELETE	`...offers/{offe`
`monetization.subscriptions.basePlans.offers.activate`	POST	`...offers/{offe`
`monetization.subscriptions.basePlans.offers.deactivate`	POST	`...offers/{offe`
`monetization.subscriptions.basePlans.offers.batchGet`	GET	`...offers:batch`
`monetization.subscriptions.basePlans.offers.batchUpdate`	POST	`...offers:batch`
`monetization.subscriptions.basePlans.offers.batchUpdateStates`	POST	`...offers:batch`

265

Catalog Management: One Time Products (In-App Products)

ENDPOINT	METHOD	PATH
`inappproducts.insert`	POST	`applications/{packageName}/inappproducts`
`inappproducts.get`	GET	`applications/{packageName}/inappproducts/{sku}`
`inappproducts.list`	GET	`applications/{packageName}/inappproducts`
`inappproducts.patch`	PATCH	`applications/{packageName}/inappproducts/{sku}`
`inappproducts.update`	PUT	`applications/{packageName}/inappproducts/{sku}`
`inappproducts.delete`	DELETE	`applications/{packageName}/inappproducts/{sku}`
`inappproducts.batchGet`	GET	`applications/{packageName}/inappproducts:batchGet`
`inappproducts.batchUpdate`	POST	`applications/{packageName}/inappproducts:batchUpdat`
`inappproducts.batchDelete`	POST	`applications/{packageName}/inappproducts:batchDelet`

Rate Limiting

The Google Play Developer API uses a three tier quota system:

TIER	DEFAULT QUOTA	ENDPOINTS
Read	Higher limits	All `get`, `list`, `batchGet` operations
Write	Lower limits	All `create`, `update`, `patch`, `delete`, `batchUpdate` operations
Sensitive	Most restricted	`refund`, `revoke`, price migration operations

Quota is measured per project, per minute. When you exceed your quota, the API returns HTTP 429 (Too Many Requests). Implement exponential backoff for quota errors.

266

You can monitor your quota usage in the Google Cloud Console under **APIs & Services > Dashboard > Android Publisher API**.

267

Appendix D: Migrating from PBL 7 to PBL 8

Google Play Billing Library 8.0 is a major release that removes deprecated APIs, renames confusing terminology, and introduces new features that simplify billing integration. If you have a working PBL 7 integration, you cannot simply bump the version number and expect everything to compile. Several APIs that were deprecated in PBL 6 and 7 are now gone entirely, and a few method signatures have changed in ways that require code updates.

This appendix walks you through every breaking change, shows you the before and after code for each migration step, and gives you a checklist to follow so nothing falls through the cracks. If you are starting a new project from scratch, you can skip this appendix. But if you have an existing PBL 7 codebase, read this carefully before upgrading.

Breaking Changes Summary

The following table summarizes every breaking change between PBL 7 and PBL 8. Use it as a quick reference, then read the detailed sections below for migration instructions.

AREA	PBL 7 (REMOVED/CHANGED)	PBL 8 (REPLACEMENT)
Purchase history	`queryPurchaseHistoryAsync()`	`queryPurchasesAsync()`
Product queries	`querySkuDetailsAsync()`	`queryProductDetailsAsync()`
Pending purchases	`enablePendingPurchases()` (no args)	`enablePendingPurchases(PendingPurchasesPara`
Product details result	`ProductDetailsResponseListener` returns `List<ProductDetails>`	Returns `QueryProductDetailsResult`
Alternative billing	`enableAlternativeBilling()`	`enableUserChoiceBilling()`
Alternative billing listener	`AlternativeBillingListener`	`UserChoiceBillingListener`
Alternative choice details	`AlternativeChoiceDetails`	`UserChoiceDetails`
Proration modes	`ProrationMode` constants	`ReplacementMode` constants
Proration setter	`setReplaceProrationMode()`	`setSubscriptionReplacementMode()`
Subscription updates	`SubscriptionUpdateParams`	`SubscriptionProductReplacementParams` (8.1)
Auto reconnection	Manual reconnection required	`enableAutoServiceReconnection()`

Removed APIs and Their Replacements

queryPurchaseHistoryAsync() Removal

`queryPurchaseHistoryAsync()` returned the most recent purchase for each product the user had ever bought, including expired and consumed purchases. Google removed this API in PBL 8 because it encouraged patterns that led to incorrect entitlement logic. Developers often used it to "restore purchases" on new devices, but the results included purchases that were no longer valid, leading to users getting access they should not have.

The replacement is `queryPurchasesAsync()`, which returns only active, unconsumed purchases. This is what you should use for entitlement checks and purchase restoration.

Before (PBL 7):

```
billingClient.queryPurchaseHistoryAsync(
    QueryPurchaseHistoryParams.newBuilder()
        .setProductType(ProductType.SUBS)
        .build()
) { billingResult, historyRecords ->
    if (billingResult.responseCode ==
        BillingResponseCode.OK
    ) {
        historyRecords?.forEach { record ->
            // Process historical purchase
        }
    }
}
```

After (PBL 8):

```
val result = billingClient.queryPurchasesAsync(
    QueryPurchasesParams.newBuilder()
        .setProductType(ProductType.SUBS)
        .build()
)
if (result.billingResult.responseCode ==
    BillingResponseCode.OK
) {
    result.purchasesList.forEach { purchase ->
        // Process active purchase
    }
}
```

If you genuinely need historical purchase data (for analytics or auditing), query the Google Play Developer API from your backend using the Purchases.subscriptionsv2 or Purchases.products endpoints. The client library is no longer the right place for historical queries.

querySkuDetailsAsync() to queryProductDetailsAsync()

`querySkuDetailsAsync()` was the original API for fetching product information. It was deprecated in PBL 5 when Google introduced the new product model with base plans and offers. In PBL 8, it is gone entirely.

The migration is straightforward, but the return types are different. `SkuDetails` was a flat object with a single price. `ProductDetails` has a richer structure with `subscriptionOfferDetails` containing multiple base plans and offers, each with their own pricing phases.

Before (PBL 7):

```
val params = SkuDetailsParams.newBuilder()
    .setSkusList(listOf("premium_monthly"))
    .setType(BillingClient.SkuType.SUBS)
    .build()

billingClient.querySkuDetailsAsync(params) {
    billingResult, skuDetailsList ->
    skuDetailsList?.forEach { skuDetails ->
        val price = skuDetails.price
        val title = skuDetails.title
    }
}
```

After (PBL 8):

```
val params = QueryProductDetailsParams.newBuilder()
    .setProductList(listOf(
        QueryProductDetailsParams.Product
            .newBuilder()
            .setProductId("premium_monthly")
            .setProductType(ProductType.SUBS)
            .build()
    )).build()

val result = billingClient
    .queryProductDetailsAsync(params)
```

Note that `queryProductDetailsAsync` in PBL 8 returns a `QueryProductDetailsResult` rather than delivering results through a callback with a raw list. You access the product details through the result object.

271

See the next section for details on this change.

enablePendingPurchases() Signature Change

In PBL 7, you called `enablePendingPurchases()` with no arguments to opt in to pending purchase support. In PBL 8, Google requires you to pass a `PendingPurchasesParams` object. This gives you explicit control over which types of pending purchases you support.

Before (PBL 7):

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

After (PBL 8):

```
val billingClient = BillingClient.newBuilder(context)
    .setListener(purchasesUpdatedListener)
    .enablePendingPurchases(
        PendingPurchasesParams.newBuilder()
            .enableOneTimeProducts()
            .enablePrepaidPlans()
            .build()
    )
    .build()
```

The `PendingPurchasesParams` builder lets you specify `enableOneTimeProducts()` and `enablePrepaidPlans()` separately. You must call at least one of these methods. If your app supports both one time products and prepaid subscriptions, call both. If you only sell auto renewing subscriptions (no prepaid, no one time products), you still need to pass the params object, but you configure it for the product types you actually sell.

ProductDetailsResponseListener Signature Change

In PBL 7, `queryProductDetailsAsync()` accepted a `ProductDetailsResponseListener` that received a `BillingResult` and a nullable `List<ProductDetails>`. In PBL 8, the method returns a `QueryProductDetailsResult` object instead. This result object wraps both the billing result and the product details list, and also introduces handling for unfetched products (covered in the new features section below).

Before (PBL 7):

```
billingClient.queryProductDetailsAsync(
    params,
    ProductDetailsResponseListener {
        billingResult, productDetailsList ->
        if (billingResult.responseCode ==
            BillingResponseCode.OK
        ) {
            productDetailsList?.forEach { details ->
                displayProduct(details)
            }
        }
    }
)
```

After (PBL 8):

```
val result = billingClient
    .queryProductDetailsAsync(params)

if (result.billingResult.responseCode ==
    BillingResponseCode.OK
) {
    result.productDetailsList.forEach { details ->
        displayProduct(details)
    }
    // Check for products that could not be fetched
    result.unfetchedProductList.forEach { unfetched ->
        logUnfetchedProduct(unfetched)
    }
}
```

The `QueryProductDetailsResult` object gives you three things: the `billingResult` with the response code, the `productDetailsList` containing the products that were successfully fetched, and the `unfetchedProductList` containing products that could not be retrieved. This is a cleaner pattern than the nullable list from PBL 7.

Alternative Billing API Renaming

Google renamed the entire alternative billing API surface to use "user choice" terminology instead of "alternative." This is not just cosmetic. The rename reflects Google's compliance framework and the legal terminology used in various markets. The functionality is identical, but every class, method, and interface name has changed.

273

PBL 7 NAME	PBL 8 NAME
`enableAlternativeBilling()`	`enableUserChoiceBilling()`
`AlternativeBillingListener`	`UserChoiceBillingListener`
`AlternativeChoiceDetails`	`UserChoiceDetails`
`AlternativeChoiceDetails.Product`	`UserChoiceDetails.Product`

Before (PBL 7):

```
val billingClient = BillingClient.newBuilder(context)
    .setListener(purchasesUpdatedListener)
    .enablePendingPurchases()
    .enableAlternativeBilling(
        AlternativeBillingListener { details ->
            val products = details.products
            val token = details.externalTransactionToken
            handleAlternativeBilling(products, token)
        }
    )
    .build()
```

After (PBL 8):

```
val billingClient = BillingClient.newBuilder(context)
    .setListener(purchasesUpdatedListener)
    .enablePendingPurchases(
        PendingPurchasesParams.newBuilder()
            .enableOneTimeProducts()
            .build()
    )
    .enableUserChoiceBilling(
        UserChoiceBillingListener { details ->
            val products = details.products
            val token = details.externalTransactionToken
            handleUserChoiceBilling(products, token)
        }
    )
    .build()
```

If you use alternative billing in your app, this is a find and replace migration. The callback behavior, the data you receive, and the flow you implement are all the same. Just update the names.

274

Replacement Mode Migration from ProrationMode

PBL 7 used `ProrationMode` constants to specify how plan changes should handle billing transitions. PBL 8 replaces this with `ReplacementMode`, which uses clearer names and adds a new mode (`KEEP_EXISTING` in PBL 8.1). The setter method on `SubscriptionUpdateParams` also changed from `setReplaceProrationMode()` to `setSubscriptionReplacementMode()`.

Constant Mapping

PRORATIONMODE (PBL 7)	REPLACEMENTMODE (PBL 8)
`IMMEDIATE_WITH_TIME_PRORATION`	`WITH_TIME_PRORATION`
`IMMEDIATE_AND_CHARGE_PRORATED_PRICE`	`CHARGE_PRORATED_PRICE`
`IMMEDIATE_AND_CHARGE_FULL_PRICE`	`CHARGE_FULL_PRICE`
`IMMEDIATE_WITHOUT_PRORATION`	`WITHOUT_PRORATION`
`DEFERRED`	`DEFERRED`
(N/A)	`KEEP_EXISTING` (PBL 8.1+)

Notice that Google dropped the `IMMEDIATE_` prefix from most modes. The old names were verbose and somewhat misleading (for example, `DEFERRED` was never prefixed with `IMMEDIATE_` even in PBL 7, creating an inconsistency). The new names are shorter and more consistent.

Before (PBL 7):

```
val updateParams = BillingFlowParams
    .SubscriptionUpdateParams.newBuilder()
    .setOldPurchaseToken(oldToken)
    .setReplaceProrationMode(
        ProrationMode.IMMEDIATE_AND_CHARGE_PRORATED_PRICE
    )
    .build()
```

After (PBL 8.0):

```
val updateParams = BillingFlowParams
    .SubscriptionUpdateParams.newBuilder()
    .setOldPurchaseToken(oldToken)
    .setSubscriptionReplacementMode(
        ReplacementMode.CHARGE_PRORATED_PRICE
    )
    .build()
```

SubscriptionUpdateParams Deprecation in PBL 8.1

In PBL 8.0, `SubscriptionUpdateParams` still works with the new `ReplacementMode` constants and the renamed setter. But starting in PBL 8.1, Google deprecated `SubscriptionUpdateParams` entirely in favor of `SubscriptionProductReplacementParams`. The new API attaches replacement configuration to the product parameters rather than to the billing flow parameters.

After (PBL 8.1, recommended):

```
val replacementParams = BillingFlowParams
    .SubscriptionProductReplacementParams
    .newBuilder()
    .setOldPurchaseToken(oldToken)
    .setReplacementMode(
        ReplacementMode.CHARGE_PRORATED_PRICE
    )
    .build()

val productParams = BillingFlowParams
    .ProductDetailsParams.newBuilder()
    .setProductDetails(newProductDetails)
    .setOfferToken(selectedOfferToken)
    .setSubscriptionReplacementParams(
        replacementParams
    )
    .build()
```

This two step migration (ProrationMode to ReplacementMode in 8.0, then SubscriptionUpdateParams to SubscriptionProductReplacementParams in 8.1) can be done all at once if you are jumping straight from PBL 7 to PBL 8.1 or later. There is no reason to stop at the intermediate step.

New Features to Adopt

276

PBL 8 is not just about breaking changes. It introduces several new features that you should adopt during your migration to get the most out of the upgrade.

enableAutoServiceReconnection()

In PBL 7, when the connection to Google Play Services dropped (which happens when the system kills the Play Store process or the user's device switches networks), you had to detect the disconnection in `onBillingServiceDisconnected()` and manually call `startConnection()` again. This was error prone and led to a lot of boilerplate retry logic.

PBL 8 introduces `enableAutoServiceReconnection()` on the `BillingClient.Builder`. When enabled, the library automatically re establishes the connection when it drops, with built in exponential backoff. You no longer need to manage reconnection yourself.

```
val billingClient = BillingClient.newBuilder(context)
    .setListener(purchasesUpdatedListener)
    .enablePendingPurchases(
        PendingPurchasesParams.newBuilder()
            .enableOneTimeProducts()
            .build()
    )
    .enableAutoServiceReconnection()
    .build()
```

After enabling this, your `onBillingServiceDisconnected()` callback still fires, but you do not need to call `startConnection()` inside it. The library handles that for you. You can use the callback purely for logging or UI updates (such as showing a temporary "reconnecting" indicator).

UnfetchedProduct Handling

When you call `queryProductDetailsAsync()` in PBL 8, the result may include an `unfetchedProductList`. This list contains products that you requested but that could not be retrieved. Reasons include a product being deactivated in the Play Console, a product ID typo, or a temporary server issue for a specific product.

In PBL 7, if one product out of ten failed to load, you got no information about which one failed. You just received a shorter list than expected. In PBL 8, the `UnfetchedProduct` object tells you exactly which product ID failed and why.

277

```
val result = billingClient
    .queryProductDetailsAsync(params)

result.unfetchedProductList.forEach { unfetched ->
    Log.w("Billing",
        "Could not fetch: ${unfetched.productId}, " +
        "reason: ${unfetched.reason}"
    )
}
```

Use this information to build better diagnostics. If a product consistently appears in the unfetched list, it might be misconfigured in the Play Console or the product ID in your code might have a typo.

Sub Response Codes

PBL 8 introduces sub response codes that provide more granular information about why a purchase failed. The main `BillingResponseCode` tells you *what* happened (for example, the purchase failed). The sub response code tells you *why* (for example, the user had insufficient funds).

You access sub response codes through `BillingResult.getOnPurchasesUpdatedSubResponseCode()` inside your `PurchasesUpdatedListener`. See Appendix B for the full list of sub response codes and recommended handling strategies.

includeSuspendedSubscriptions (PBL 8.1)

PBL 8.1 adds the `includeSuspendedSubscriptions` option to `QueryPurchasesParams`. When enabled, `queryPurchasesAsync()` returns subscriptions that are in a suspended state (such as account hold or grace period) in addition to fully active subscriptions.

```
val params = QueryPurchasesParams.newBuilder()
    .setProductType(ProductType.SUBS)
    .includeSuspendedSubscriptions(true)
    .build()
val result = billingClient
    .queryPurchasesAsync(params)
```

This is useful when you want to show users that their subscription has a payment issue and prompt them to update their payment method. Without this flag, suspended subscriptions are invisible to `queryPurchasesAsync()`, which means you cannot detect the problem on the client side and must rely entirely on your backend and push notifications to alert the user.

Step by Step Migration Checklist

Follow these steps in order when migrating from PBL 7 to PBL 8.

1. **Update your dependency.** Change your `build.gradle` file to reference PBL 8.x. Set the version to the latest stable release (8.1 or later is recommended, since it includes `SubscriptionProductReplacementParams` and `includeSuspendedSubscriptions`).

2. **Fix the BillingClient builder.** Replace `enablePendingPurchases()` with `enablePendingPurchases(PendingPurchasesParams)`. Add `enableAutoServiceReconnection()`. If you use alternative billing, rename `enableAlternativeBilling()` to `enableUserChoiceBilling()` and update the listener type.

3. **Remove queryPurchaseHistoryAsync() calls.** Replace them with `queryPurchasesAsync()`. If you need historical data, move that logic to your backend.

4. **Remove querySkuDetailsAsync() calls.** Replace them with `queryProductDetailsAsync()`. Update your product display code to handle the `ProductDetails` model with its base plans and offers structure.

5. **Update queryProductDetailsAsync() callbacks.** Switch from the `ProductDetailsResponseListener` callback pattern to the `QueryProductDetailsResult` return type. Add handling for `unfetchedProductList`.

6. **Replace ProrationMode with ReplacementMode.** Find every usage of `ProrationMode` constants and replace them with the corresponding `ReplacementMode` constants. Replace `setReplaceProrationMode()` with `setSubscriptionReplacementMode()`.

7. **Migrate SubscriptionUpdateParams (if targeting PBL 8.1+).** Replace `SubscriptionUpdateParams` with `SubscriptionProductReplacementParams`. Move the replacement configuration from `BillingFlowParams` to `ProductDetailsParams`.

8. **Rename alternative billing classes.** Replace `AlternativeBillingListener` with `UserChoiceBillingListener`, `AlternativeChoiceDetails` with `UserChoiceDetails`, and update all references.

9. **Adopt new PBL 8 features.** Add `enableAutoServiceReconnection()` to your builder. Add `UnfetchedProduct` handling to your product query flow. Consider adopting sub response codes in your purchase error handling. Add `includeSuspendedSubscriptions` if you want to detect payment issues on the client.

10. **Simplify your reconnection logic.** If you have manual reconnection code in `onBillingServiceDisconnected()`, you can remove it (or reduce it to logging) now that auto reconnection is enabled.

11. **Run your test suite.** Verify that all billing flows work: new purchases, subscription renewals, plan changes (upgrades and downgrades), consumable purchases, and pending transactions. Test on a device with the latest Play Store version.

12. **Test with the Google Play Billing Library test tool.** Use the `BillingClient` test methods and license testing accounts to verify each flow without spending real money. Pay special attention to plan change

flows if you migrated from `ProrationMode` to `ReplacementMode`.

13. **Update your ProGuard/R8 rules if needed.** PBL 8 may have different class names that need to be kept. Check the official migration guide for any updated ProGuard rules.

14. **Deploy to internal testing first.** Roll out to your internal test track before pushing to production. Monitor crash reports and billing success rates for at least one full billing cycle.

Version Deprecation Timeline

Understanding Google's deprecation timeline helps you plan your migration schedule.

LIBRARY VERSION	STATUS	KEY DATE
PBL 5 and earlier	Deprecated, unsupported	Already past end of life. Google Play may reject new app submissions using these versions.
PBL 6	Deprecated	Support ended. Migrate as soon as possible.
PBL 7	Active, approaching deprecation	Support ends **August 31, 2026**. After this date, Google may stop accepting app updates that target PBL 7.
PBL 8.0	Stable	Current major version.
PBL 8.1+	Stable, recommended	Latest release with all new features.

What "support ends" means in practice: After the deprecation date, Google does not immediately break your existing app. Users who already have your app installed will still be able to make purchases. However, Google may reject new APK/AAB uploads to the Play Console if they reference the deprecated library version. Google may also stop fixing bugs in deprecated versions, so any issues you encounter will not be patched.

The practical deadline: If you are currently on PBL 7, you have until August 31, 2026 to complete your migration. That might sound like plenty of time, but billing migrations deserve thorough testing across multiple billing cycles. Start your migration early enough that you have at least two to three months of production testing before the deadline. A good target is to have your PBL 8 migration in production by June 2026 at the latest.

If you are still on PBL 5 or 6: You are already past the recommended migration window. Prioritize upgrading to PBL 8 immediately. Skipping PBL 7 entirely is fine. You can migrate directly from PBL 5 or 6 to PBL 8 by following the same steps in this appendix, though you may have additional changes to make if you are still using the original `SkuDetails` based API throughout your codebase.

The migration from PBL 7 to PBL 8 is mechanical, not architectural. Your billing flow structure stays the same. Your backend integration stays the same. The changes are at the API surface level: renamed classes, updated method signatures, and a few removed methods. Budget a few days for the code changes and a few weeks for thorough testing, and you will be in good shape well before the deadline.

Appendix E: Subscription State Diagram (Pull Out Reference)

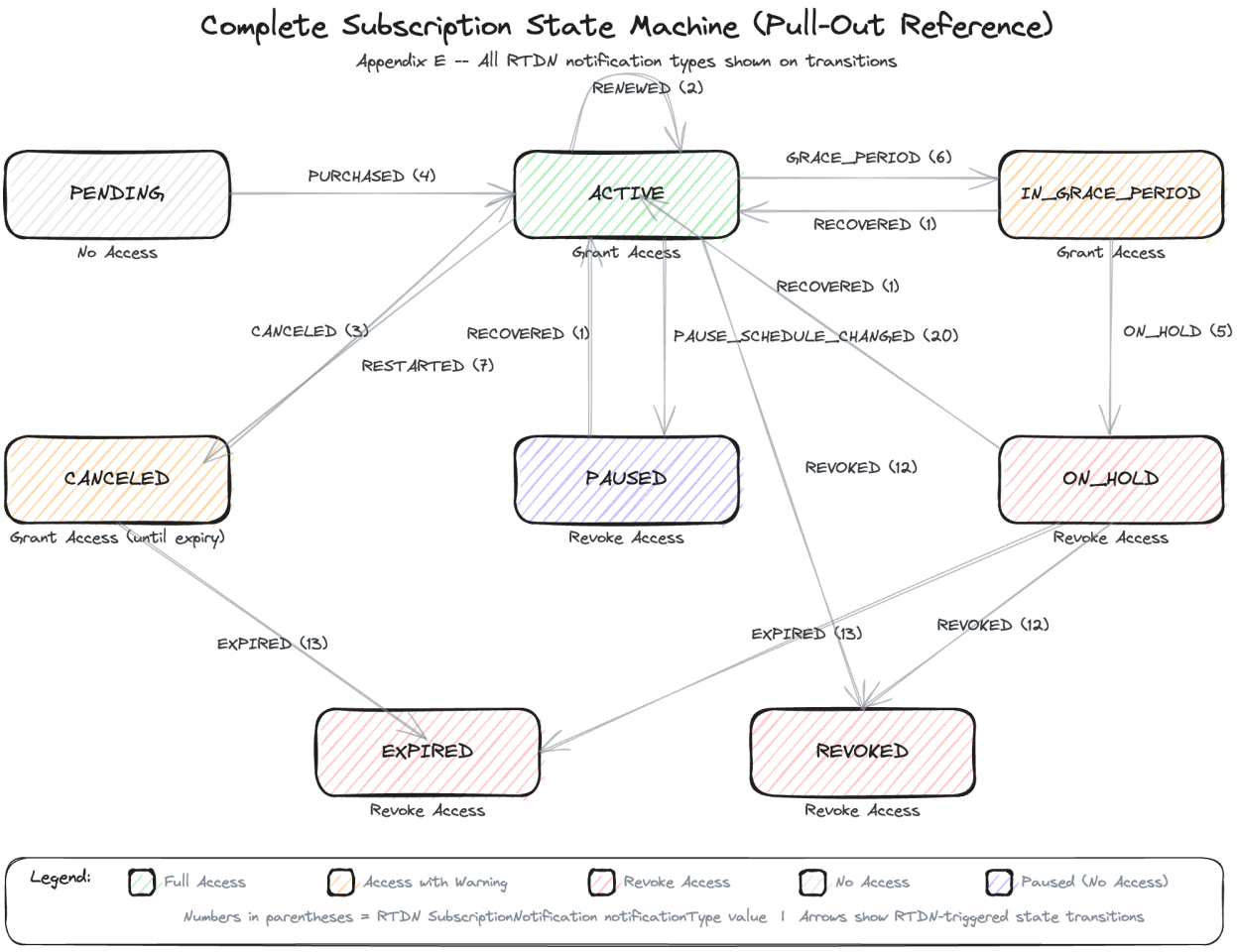
This appendix provides the complete subscription state machine as a quick reference card. Print it, pin it to your wall, or keep it open in a tab while building your billing integration.

States at a Glance

STATE	ACCESS	ICON	DESCRIPTION
PENDING	No	⏳	Purchase initiated but payment not yet processed
ACTIVE	Yes	✅	Subscription is paid and current
IN_GRACE_PERIOD	Yes	⚠️	Payment failed, retrying, user keeps access
ON_HOLD	No	🔒	Grace period expired, access revoked, awaiting payment fix
PAUSED	No	⏸	User requested pause, access revoked
CANCELED	Yes*	🔄	Canceled but not expired, access until `expiryTime`
EXPIRED	No	❌	Subscription has ended

*CANCELED grants access only until `expiryTime`.

281



All State Transitions

From PENDING

TO	TRIGGER	RTDN
ACTIVE	Payment completes	`SUBSCRIPTION_PURCHASED` (4)
EXPIRED	Payment fails or times out	`SUBSCRIPTION_PENDING_PURCHASE_CANCELED` (20)

From ACTIVE

TO	TRIGGER	RTDN
ACTIVE	Successful renewal	`SUBSCRIPTION_RENEWED` (2)
IN_GRACE_PERIOD	Renewal payment fails (grace period enabled)	`SUBSCRIPTION_IN_GRACE_PERIOD` (6)
ON_HOLD	Renewal payment fails (no grace period, account hold enabled)	`SUBSCRIPTION_ON_HOLD` (5)
CANCELED	User or developer cancels	`SUBSCRIPTION_CANCELED` (3)
EXPIRED	Revoked or refunded	`SUBSCRIPTION_REVOKED` (12)
PAUSED	Pause takes effect at end of period	`SUBSCRIPTION_PAUSED` (10)

From IN_GRACE_PERIOD

TO	TRIGGER	RTDN
ACTIVE	Payment recovered	`SUBSCRIPTION_RECOVERED` (1)
ON_HOLD	Grace period expires without payment	`SUBSCRIPTION_ON_HOLD` (5)
EXPIRED	Grace period expires (no account hold)	`SUBSCRIPTION_EXPIRED` (13)

From ON_HOLD

TO	TRIGGER	RTDN
ACTIVE	User fixes payment	`SUBSCRIPTION_RECOVERED` (1)
EXPIRED	Account hold period expires	`SUBSCRIPTION_EXPIRED` (13)

From PAUSED

TO	TRIGGER	RTDN
ACTIVE	Pause ends, payment succeeds	`SUBSCRIPTION_RECOVERED` (1)
ON_HOLD	Pause ends, payment fails	`SUBSCRIPTION_ON_HOLD` (5)

From CANCELED

TO	TRIGGER	RTDN
ACTIVE	User restores before expiration	`SUBSCRIPTION_RESTARTED` (7)
EXPIRED	`expiryTime` reached	`SUBSCRIPTION_EXPIRED` (13)

From EXPIRED

TO	TRIGGER	RTDN
ACTIVE	User resubscribes (new purchase token)	`SUBSCRIPTION_PURCHASED` (4)

Access Decision Flowchart

Use this logic to determine whether to grant access:

```
1. Is subscriptionState == ACTIVE?
   → YES: Grant access

2. Is subscriptionState == IN_GRACE_PERIOD?
   → YES: Grant access (show payment warning)

3. Is subscriptionState == CANCELED?
   → Is current time < expiryTime?
     → YES: Grant access (show expiry notice)
     → NO: Revoke access

4. All other states (ON_HOLD, PAUSED,
   EXPIRED, PENDING):
   → Revoke access
```

Access Decision Code

```
fun shouldGrantAccess(
    state: String,
    expiryTimeMillis: Long
): Boolean = when (state) {
    "SUBSCRIPTION_STATE_ACTIVE",
    "SUBSCRIPTION_STATE_IN_GRACE_PERIOD" -> true
    "SUBSCRIPTION_STATE_CANCELED" ->
        System.currentTimeMillis() < expiryTimeMillis
    else -> false
}
```

Token Lifecycle

EVENT	TOKEN BEHAVIOR
New purchase	New token created
Renewal	Same token, new Order ID
Upgrade/Downgrade	New token, old token linked via `linkedPurchaseToken`
Restore (before expiry)	Same token
Resubscribe (after expiry)	New token
Pause and resume	Same token
Grace period recovery	Same token
Account hold recovery	Same token

RTDN Quick Reference by Number

285

#	NAME	COMMON USAGE
1	RECOVERED	Restore access after hold/pause
2	RENEWED	Extend access period
3	CANCELED	Note cancellation, keep access until expiry
4	PURCHASED	New subscription, grant access
5	ON_HOLD	Revoke access
6	IN_GRACE_PERIOD	Keep access, warn user
7	RESTARTED	Restore access (same token)
8	PRICE_CHANGE_CONFIRMED	(Deprecated)
9	DEFERRED	Extend billing date
10	PAUSED	Revoke access at period end
11	PAUSE_SCHEDULE_CHANGED	Check pause status
12	REVOKED	Immediately revoke access
13	EXPIRED	Revoke access, clean up
17	ITEMS_CHANGED	Check updated line items
18	CANCELLATION_SCHEDULED	Note scheduled cancellation
19	PRICE_CHANGE_UPDATED	Check price change details
20	PENDING_PURCHASE_CANCELED	Clean up pending records
22	PRICE_STEP_UP_CONSENT_UPDATED	Check consent status

Time Windows Reference

286

WINDOW	DURATION	WHAT HAPPENS
Acknowledgement	3 days	Unacknowledged purchases are auto refunded
Grace period	Configurable (1-30 days)	User retains access while Google retries payment
Silent grace period	1 day minimum	Even with 0 day setting, there is a minimum 1 day silent period
Account hold	Configurable (up to 30 days)	User loses access, can fix payment to recover
Token validity	60 days post expiration	API calls still work for this window
Pause duration	1 week to 3 months	Depends on billing period configuration

Checklist: Do I Handle Every State?

- ☐ PENDING: Withhold access, show "pending" message
- ☐ ACTIVE: Grant full access
- ☐ IN_GRACE_PERIOD: Grant access + show payment warning + use In App Messaging
- ☐ ON_HOLD: Revoke access + show "fix payment" message
- ☐ PAUSED: Revoke access + show "paused" message with resume date
- ☐ CANCELED: Grant access until expiry + show expiry countdown
- ☐ EXPIRED: Revoke access + show resubscribe option

287