The RevenueCat Handbook

In-App Subscriptions Without the Boilerplate

Contents

Preface

Building in-app purchases on Android from scratch means writing your own `BillingClient` connection management, product queries, purchase flow orchestration, server-side receipt verification, RTDN processing, subscription state machines, grace period logic, account hold handling, and error retry strategies. Getting all of that right is genuinely complex work, and the surface area is large.

This handbook shows you the same problems solved with RevenueCat.

The structure mirrors the original chapter for chapter. Where RevenueCat dramatically simplifies the implementation, you will see the code. Where RevenueCat makes an entire problem disappear, where the answer to "how do I implement this?" is "you don't have to", you will find a short explanation of what RevenueCat handles on your behalf and what, if anything, you still need to do.

The goal is not to sell you on RevenueCat. It is to give you an honest map of the tradeoffs. RevenueCat does not make in-app purchases trivial. It moves complexity from your code into a managed service. That tradeoff is worth understanding clearly before you commit to either approach.

What RevenueCat Is

RevenueCat is a backend service and client SDK for managing in-app purchases and subscriptions. On Android, the SDK wraps `BillingClient` internally. You never call `BillingClient` directly. Instead you call `Purchases`, the RevenueCat SDK singleton, and it handles the Google Play Billing protocol on your behalf.

The SDK does three things:

1. **Client-side purchase orchestration.** `Purchases` wraps `BillingClient`, manages the connection lifecycle, launches purchase flows, handles acknowledgement and consumption, and retries transient errors automatically.
2. **Backend receipt verification.** After every purchase, the SDK posts the purchase token to the RevenueCat backend. RevenueCat calls the Google Play Developer API to verify the receipt, records the transaction, and returns a `CustomerInfo` object describing the user's current entitlements.
3. **Entitlement management.** RevenueCat maintains a server-side record of every user's active entitlements. Your app checks `customerInfo.entitlements["pro_access"]?.isActive` to decide whether to grant access. You never query the Google Play Developer API directly.

What This Handbook Assumes

This handbook assumes you have read the companion handbook or have equivalent experience with Google Play Billing. Concepts like base plans, offers, replacement modes, RTDNs, purchase tokens, and subscription states are used here without full re-explanation. The focus is on how RevenueCat maps to those concepts, not on explaining the concepts themselves.

You should be comfortable with Kotlin coroutines. The code examples use the RevenueCat coroutine extension functions (`awaitOfferings()`, `awaitPurchase()`, etc.) throughout.

SDK Version

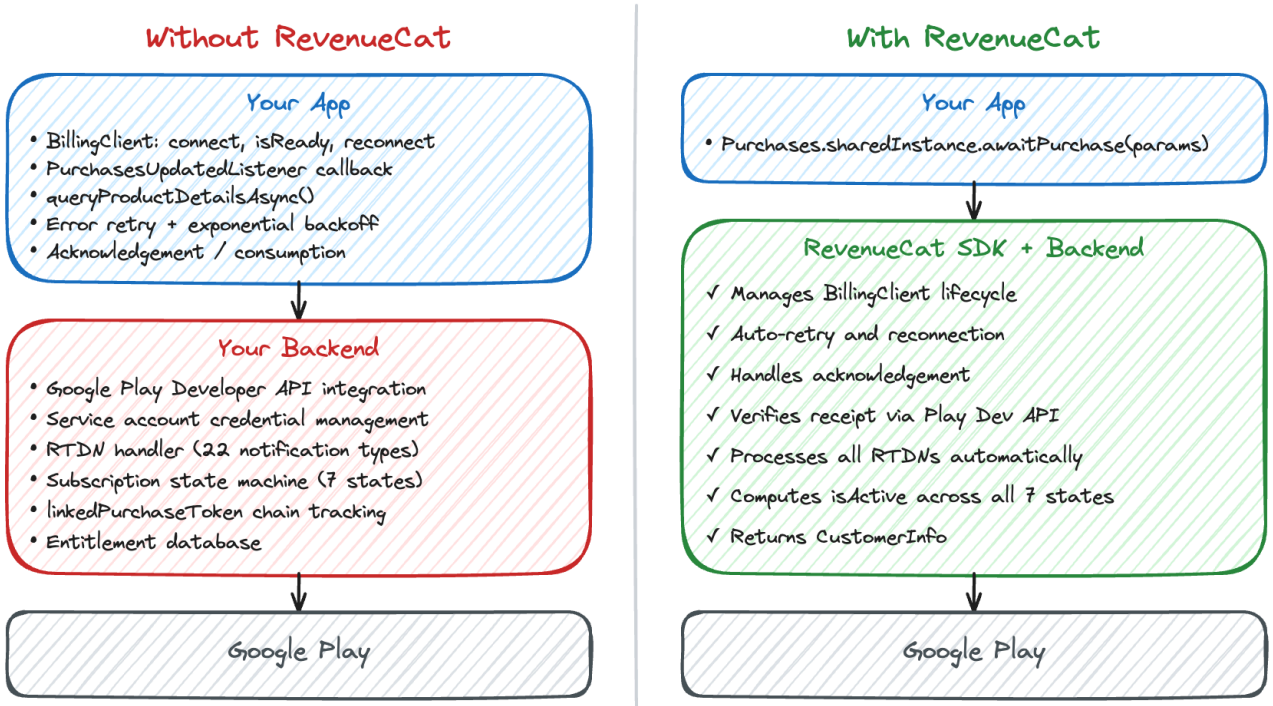
All code examples in this handbook target **RevenueCat Android SDK 9.x**, which wraps Google Play Billing Library 8.x internally. SDK 9.x requires Kotlin 1.8.0 or higher and no longer supports querying expired subscriptions or consumed one-time products, a limitation imposed by Play Billing Library 8.

Chapter 1: Understanding RevenueCat

Building in-app purchases on Android from scratch requires three separate systems: the Play Billing Library on the client, the Google Play Developer API on the server, and Real Time Developer Notifications via Cloud Pub/Sub. Each requires its own setup, its own code, and its own expertise to operate correctly.

RevenueCat reorganizes those three pillars into one. The SDK replaces the client-side `BillingClient` integration. The RevenueCat backend replaces your server-side Google Play Developer API calls. RevenueCat webhooks replace RTDNs. You interact with one SDK and one dashboard instead of three systems.

The RevenueCat Architecture




```
Your App
  ↓  Purchases SDK
RevenueCat Backend
  ↓  Google Play Developer API
Google Play
```

Your app calls the `Purchases` SDK. The SDK communicates with both Google Play (for the purchase UI and payment) and the RevenueCat backend (for verification and entitlement storage). Your server communicates with the RevenueCat backend, not directly with Google Play.

This architecture has a specific implication: **RevenueCat is in the critical path for purchase verification.** After a user completes a purchase, the SDK posts the token to RevenueCat before your app receives confirmation. This adds a network round trip but eliminates your need to implement server-side verification.

Key Concepts

Offerings and Packages

Instead of querying `ProductDetails` directly from Google Play, you fetch **Offerings** from RevenueCat. An Offering is a collection of **Packages**. A Package maps a RevenueCat logical product (monthly, annual, lifetime) to a specific Google Play product and base plan.

You configure Offerings in the RevenueCat dashboard. Your code fetches them at runtime:

```
val offerings = Purchases.sharedInstance.awaitOfferings()
val monthly = offerings.current?.monthly
```

This means your paywall code does not hardcode product IDs. You change what to show users by updating the Offering in the dashboard, with no app update required.

CustomerInfo and Entitlements

Instead of managing subscription state through the Google Play Developer API, you read **CustomerInfo**. `CustomerInfo` is a snapshot of the user's purchase history and active entitlements, computed and cached by RevenueCat.

An **Entitlement** is a logical access level you define in the RevenueCat dashboard, for example, `"pro_access"` or `"premium"`. You map one or more products to each entitlement. When a user purchases any of those products, the entitlement becomes active.

Your access check becomes:

```
val customerInfo = Purchases.sharedInstance.awaitCustomerInfo()
val hasPro = customerInfo.entitlements["pro_access"]?.isActive == true
```

This is the entire entitlement check. The seven subscription states (ACTIVE, IN_GRACE_PERIOD, ON_HOLD, PAUSED, CANCELED, EXPIRED, PENDING) are resolved server-side. `isActive` already reflects the correct result.

RevenueCat In-App Purchases Cheat Sheet

The diagram below shows how Products, Packages, Offerings, and Entitlements relate to each other inside RevenueCat.

RevenueCat In-App Purchases Cheat Sheet

Products

Individual in-app purchases configured directly within store platforms (e.g., Apple App Store or Google Play) in three different types: subscription, consumable, non-consumable

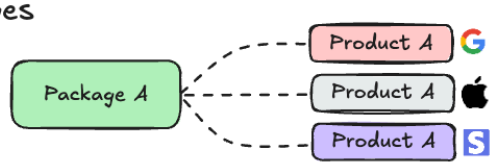
This cheat sheet outlines the key RevenueCat features and settings needed to implement in-app purchases and subscriptions. For more details, refer to the documentation below:

https://www.revenuecat.com/docs/

For an open-source project that demonstrates Google Play and Paywalls on Android using Jetpack Compose, check out the repository below:

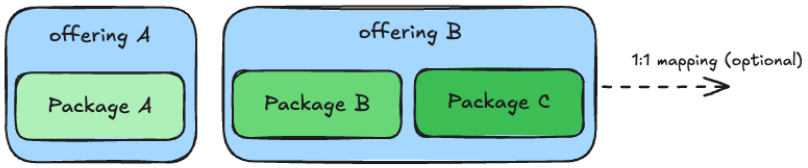
https://github.com/RevenueCat/cat-paywall-compose/

Packages



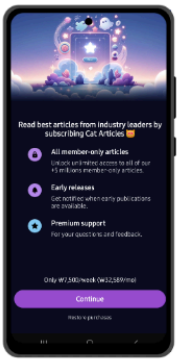
Group equivalent products across iOS, Android, and the web. If your app supports multiple platforms, a Package ensures the same product is available on all of them using matching product identifiers.

Offerings



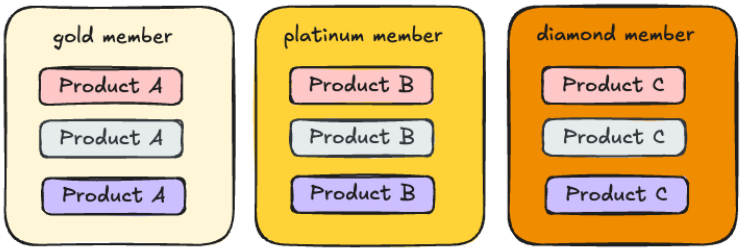
Represent groups of packages(products) presented to users on your paywall. While optional, they are highly recommended as they simplify paywall management, experimentation, and dynamic changes.

Paywalls



A Paywall is an optional feature linked to your Offering that lets you manage both the products being offered and the user interface used to display them. It gives you control over how the offer appears to users, not just what it includes.

Entitlements



Define the specific access, features, or content a customer receives when they purchase any product associated with a given entitlement. For example, if a customer purchases "Product B"—or any package containing it on any platform—they will gain the "Platinum Member" entitlement.

- **Products** are the items you define in the Google Play Console (subscriptions, one-time purchases).
- **Packages** wrap a single product with a type label (Monthly, Annual, Lifetime, or custom). A Package is the unit your paywall displays.
- **Offerings** group one or more Packages. One Offering is marked **Current** and returned by `offerings.current`. You switch the Current Offering from the dashboard without a new app build.
- **Entitlements** are the access levels your app enforces (`"pro_access"` , `"premium"` , etc.). You attach Products to Entitlements in the dashboard. Purchasing any attached Product activates the Entitlement in `CustomerInfo` .

The Purchases Singleton

`Purchases.sharedInstance` is the entry point for all SDK operations. You configure it once at app startup with your RevenueCat API key and it manages the rest of the session:

```
Purchases.configure(
    PurchasesConfiguration.Builder(context, "your_api_key")
        .appUserID("optional_user_id")
        .build()
)
```

Unlike `BillingClient`, the `Purchases` singleton does not require you to manage connection state. It handles reconnection automatically and queues operations that arrive before the connection is ready.

What RevenueCat Does Not Replace

RevenueCat does not replace the Google Play Console. You still create products, base plans, and offers there. RevenueCat reads your Play Console product catalog and lets you organize those products into Offerings.

RevenueCat does not replace your app's UI. You still build your paywall, manage navigation, and display product prices. RevenueCat provides the data.

RevenueCat does not replace your user authentication system. You bring your own user IDs and pass them to the SDK. RevenueCat links purchase history to those IDs.

The Tradeoff

With raw Google Play Billing, you own the entire stack and every failure mode. With RevenueCat, you offload the complexity but take on a dependency. If RevenueCat's backend has an outage, your purchase verification flow is affected.

The SDK keeps a **disk cache** of `CustomerInfo` that allows entitlement checks to work offline using the last-known server state. This cache can become stale. Separately, RevenueCat offers an **Offline Entitlements** feature that computes entitlements client-side from local Play Store purchases, this is a distinct, explicitly-configured feature rather than the standard cache. Both provide resilience but with different semantics; neither is a full substitute for a reachable backend.

The rest of this handbook is about understanding exactly where RevenueCat helps and what remains your responsibility.

Chapter 2: Setting Up RevenueCat

Setting up in-app purchases from scratch means creating a Google Play Developer account, configuring your app in the Play Console, enabling the Google Play Developer API, setting up service account credentials, creating a Cloud Pub/Sub topic for RTDNs, and configuring billing permissions. That is a significant amount of infrastructure before writing a single line of client code.

RevenueCat setup has two parts: the RevenueCat dashboard (where you define your product structure) and the Android SDK (where you initialize the library). Most of the Play Console work still happens, you still need products and base plans there, but the server infrastructure you would otherwise build is replaced by RevenueCat's dashboard.

Step 1: Create a RevenueCat Account

Sign up at `app.revenuecat.com`. Create a project for your app. Within the project, add an Android app. RevenueCat will generate an API key for your Android app, you will need this key to initialize the SDK.

RevenueCat uses separate API keys per platform (Android, iOS, web). Use the Android-specific key in your Android app.

Step 2: Connect Google Play

In the RevenueCat dashboard, go to your app settings and connect your Google Play app. This requires:

1. **Package name**: the same package name as in your Play Console app.
2. **Service account credentials**: a JSON key file from a Google Cloud service account with the Google Play Developer API enabled. RevenueCat uses this to verify receipts and read subscription state.

The service account setup is the same as described in the raw billing handbook. RevenueCat needs a service account with the **Financial data** viewer permission in your Play Console. Once connected, RevenueCat polls the Play Developer API on your behalf, you never call it directly.

Step 3: Create Entitlements

In the RevenueCat dashboard under **Entitlements**, create the access levels your app grants. For example:

- Identifier: `pro_access`
- Description: Premium features including all themes and weather history

Entitlements are the logical access levels that drive your app's feature gates. You define them once here and check `customerInfo.entitlements["pro_access"]?.isActive` everywhere in your code.

Step 4: Import Products

Go to **Products** in the dashboard. Click **Import** to pull in your Google Play products. RevenueCat fetches the product catalog using the connected service account credentials. Select the products you want to manage with RevenueCat.

Attach each product to an entitlement. This tells RevenueCat: "when a user purchases this product, grant them the `pro_access` entitlement."

Step 5: Create Offerings

Go to **Offerings** and create at least one Offering. An Offering groups Packages. Add Packages to your Offering:

- Select a package type (Monthly, Annual, Lifetime, or custom)
- Assign a product to each package

Set one Offering as the **Current Offering**. Your app will fetch `offerings.current` to display the default paywall. You can change which Offering is current from the dashboard at any time, with no app update needed.

Step 6: Add the SDK Dependency

Add the RevenueCat SDK to your app's `build.gradle.kts`:

```
dependencies {
    implementation("com.revenuecat.purchases:purchases:9.x.x")
}
```

The SDK includes `com.android.billingclient:billing` as a transitive dependency. You do not need to add the BillingClient dependency separately. Do not add a conflicting BillingClient version, let RevenueCat control which BillingClient version it uses internally.

Step 7: Initialize the SDK

Initialize RevenueCat in your `Application.onCreate()`. This must happen before any other SDK calls:

```
class MyApplication : Application() {
    override fun onCreate() {
        super.onCreate()

        Purchases.logLevel = LogLevel.DEBUG // remove in production

        Purchases.configure(
            PurchasesConfiguration.Builder(this, "your_revenuecat_api_key")
                .appUserID(null) // or your own user ID
                .build()
        )
    }
}
```

If you pass `null` for `appUserID`, RevenueCat generates an anonymous ID for the user. If you have your own user authentication system, pass the user's ID instead. See the "User Identification" section in Chapter 11 for how to log in, log out, and merge anonymous purchase history.

Step 8: Listen for CustomerInfo Updates

Set an `UpdatedCustomerInfoListener` to receive CustomerInfo whenever the SDK refreshes it:

```
Purchases.sharedInstance.updatedCustomerInfoListener =
    UpdatedCustomerInfoListener { customerInfo ->
        // Update your UI or state based on new entitlements
    }
```

This listener fires after purchases, restores, and SDK-initiated refreshes. It does not fire on every app launch, you should also call `getCustomerInfo()` on launch to get the current state.

What You Do Not Set Up

Compared to raw billing, you do not set up:

- Cloud Pub/Sub topics or subscriptions
- RTDN push endpoint on your server
- Server-side receipt verification code
- Google Play Developer API integration on your backend
- Connection retry logic for `BillingClient`

RevenueCat handles all of these. Your server only needs to receive RevenueCat webhooks if you want server-side events (covered in Chapter 10).

Chapter 3: One-Time Products with RevenueCat

Implementing one-time products from scratch means querying `ProductDetails`, building `BillingFlowParams`, handling the `PurchasesUpdatedListener`, acknowledging purchases, consuming consumables, managing pending states, and verifying receipts server side. Each of those steps requires code you own and maintain.

With RevenueCat, the purchase flow is three method calls: fetch offerings, launch purchase, check the result.

Querying Products

You do not call `queryProductDetailsAsync()`. You fetch Offerings:

```
// Coroutine style
val offerings = Purchases.sharedInstance.awaitOfferings()
val product = offerings.current?.availablePackages?.first()?.product
```

Or if you need a specific product by ID outside of Offerings:

```
val products = Purchases.sharedInstance.awaitGetProducts(
    listOf("your_product_id"),
    type = ProductType.INAPP
)
val product = products.firstOrNull()
```

The `StoreProduct` object returned by the SDK wraps the underlying `ProductDetails` from Google Play and exposes the same price and description fields:

```
val price = product?.price?.formatted    // e.g. "$4.99"
val title = product?.title
val description = product?.description
```

Launching a Purchase

Pass a `Package` or `StoreProduct` to `Purchases.purchase()`:

```
// Using a Package from Offerings (recommended)
val packageToPurchase = offerings.current?.availablePackages?.first()
    ?: return

try {
    val result = Purchases.sharedInstance.awaitPurchase(
        PurchaseParams.Builder(activity, packageToPurchase).build()
    )
    // result.customerInfo has updated entitlements
    // result.storeTransaction has the purchase token
    grantAccess()
} catch (e: PurchasesTransactionException) {
    if (!e.userCancelled) {
        showError(e.error.message)
    }
}
```

When you call `awaitPurchase()`, the SDK:

1. Calls `BillingClient.launchBillingFlow()` internally
2. Waits for the `PurchasesUpdatedListener` result
3. Posts the purchase token to the RevenueCat backend for verification
4. Acknowledges or consumes the purchase automatically
5. Returns updated `CustomerInfo` with the new entitlements

Steps 3, 4, and 5 happen automatically. You do not write any of that code.

Acknowledgement and Consumption

You do not call `acknowledgePurchase()` or `consumeAsync()` yourself. The SDK handles both:

- **Non-consumable products**: acknowledged automatically after the RevenueCat backend confirms the purchase.
- **Consumable products**: you need to tell RevenueCat the product is consumable. In the RevenueCat dashboard, mark the product as a consumable. After that, the SDK calls `consumeAsync()` automatically when the purchase is verified.

This means you no longer risk the 3-day acknowledgement window expiring because of a bug in your code.

Checking Purchase History

To check whether a user has purchased a non-consumable product:

```
val customerInfo = Purchases.sharedInstance.awaitCustomerInfo()

// Via entitlements (recommended)
val hasPurchased = customerInfo.entitlements["lifetime_access"]?.isActive == true

// Via non-subscription transactions
val hasLifetime = customerInfo.nonSubscriptionTransactions
    .any { it.productIdentifier == "your_lifetime_product_id" }
```

The entitlement approach is preferred because it abstracts you from specific product IDs.

Handling Pending Purchases

RevenueCat does not grant entitlements for pending purchases. The `awaitPurchase()` call throws `PurchasesTransactionException` with code `PaymentPendingError` when a purchase enters the pending state. The exception's `userCancelled` property is `false`.

```
} catch (e: PurchasesTransactionException) {
    if (e.error.code == PurchasesErrorCode.PaymentPendingError) {
        showPendingPaymentMessage()
    } else if (!e.userCancelled) {
        showError(e.error.message)
    }
}
```

RevenueCat will automatically detect when the pending payment completes (via RTDN processing on their backend) and update the entitlement at that point. Your app will receive the update through the `UpdatedCustomerInfoListener`.

Server-Side Verification

You do not implement server-side purchase verification. RevenueCat does it for you on every purchase. After `awaitPurchase()` returns successfully, the purchase has already been verified against the Google Play Developer API by RevenueCat's backend.

If you need to know the purchase happened on your own server (for example, to provision access in your database), subscribe to RevenueCat webhooks (Chapter 10). The `INITIAL_PURCHASE` event fires for every new non-subscription purchase.

Common Pitfalls

Calling `queryProductDetailsAsync()` directly. Once you integrate RevenueCat, you should not call BillingClient APIs directly. The SDK owns the `BillingClient` instance. Calling BillingClient methods alongside RevenueCat will produce undefined behavior.

Granting access before `awaitPurchase()` returns. The method only returns after the RevenueCat backend has verified the purchase. The returned `CustomerInfo` is the source of truth, use it, not optimistic local state.

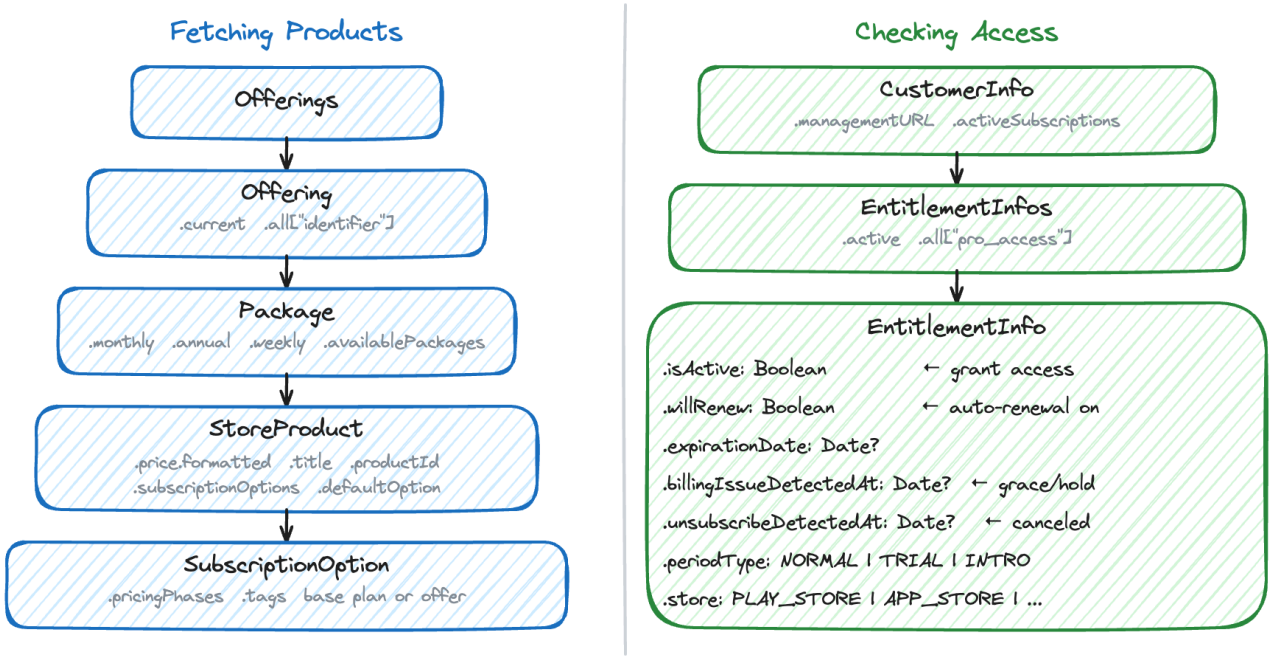
Ignoring `PaymentPendingError`. Pending purchases are real purchases waiting to be completed. Show a message and listen for the `UpdatedCustomerInfoListener` to fire when the payment completes.

Chapter 4: Subscriptions with RevenueCat

Working with subscriptions directly means navigating the three-tier Google Play hierarchy: Subscription → Base Plan → Offer. You call `queryProductDetailsAsync()` and get back nested `SubscriptionOfferDetails` objects you must parse and select from manually.

With RevenueCat, the three-tier hierarchy is still there on the Google Play side, but the SDK presents it through a simpler abstraction: **Offerings → Packages → SubscriptionOptions**.

How RevenueCat Maps the Hierarchy



In the RevenueCat dashboard, you map Google Play products and base plans to Packages inside Offerings. The mapping looks like this:

GOOGLE PLAY	REVENUECAT
Subscription (product ID)	Product
Base Plan	SubscriptionOption (base plan)
Offer	SubscriptionOption (offer)
Group of base plans in an Offering	Package

A `Package` has a `product` property of type `StoreProduct`. A `StoreProduct` for a subscription has a `subscriptionOptions` list, where each `SubscriptionOption` represents either a base plan or an offer.

Fetching Subscription Products

```
val offerings = Purchases.sharedInstance.awaitOfferings()
val offering = offerings.current ?: return

// Access common package types directly
val monthly = offering.monthly
val annual = offering.annual
val weekly = offering.weekly

// Or iterate all packages
for (pkg in offering.availablePackages) {
    val product = pkg.product
    val price = product.price.formatted
    val period = product.period?.iso8601     // "P1M", "P1Y", etc. (null for INAPP)
    val title = product.title
}
```

The `PackageType` enum gives you named access to standard durations: `MONTHLY`, `ANNUAL`, `WEEKLY`, `TWO_MONTH`, `THREE_MONTH`, `SIX_MONTH`, `LIFETIME`. Custom package types have `PackageType.CUSTOM`.

Selecting the Right SubscriptionOption

When you pass a `Package` to `PurchaseParams`, the SDK selects the `defaultOption` automatically. The default option selection logic:

1. Filters out options tagged `"rc-ignore-offer"` or `"rc-customer-center"`
2. Chooses the option with the longest free trial or cheapest first phase
3. Falls back to the base plan if no offers qualify

This means users automatically get the best available offer without you writing selection logic.

If you want to present a specific offer to the user, for example, a win-back offer only for returned subscribers, access the `SubscriptionOption` directly:

```
val product = offering.monthly?.product ?: return
val subscriptionOptions = product.subscriptionOptions

// Find a specific offer by its Google Play offer ID or tag
val winBackOption = subscriptionOptions?.firstOrNull { option ->
    option.tags.contains("win-back")
}

val optionToPurchase = winBackOption ?: product.defaultOption ?: return
val params = PurchaseParams.Builder(activity, optionToPurchase).build()
```

Displaying Trial and Introductory Pricing

Each `SubscriptionOption` has a `pricingPhases` list. The first phase may be a free trial or discounted intro price:

```
val option = pkg.product.defaultOption ?: return
val firstPhase = option.pricingPhases.first()

// Use offerPaymentMode for the clearest free trial detection
val isTrial = firstPhase.offerPaymentMode == OfferPaymentMode.FREE_TRIAL

if (isTrial) {
    // billingPeriod.value is the count of the period's unit (e.g., 1 for P1W, not 7)
    // Use billingPeriod.unit to build a correct label
    val period = firstPhase.billingPeriod
    val trialLabel = when (period.unit) {
        Period.Unit.DAY -> "${period.value}-day"
        Period.Unit.WEEK -> "${period.value}-week"
        Period.Unit.MONTH -> "${period.value}-month"
        Period.Unit.YEAR -> "${period.value}-year"
        else -> period.iso8601
    }
    showLabel("Start your $trialLabel free trial")
} else {
    showLabel(firstPhase.price.formatted)
}
```

Free Trial Eligibility

Google Play returns only the offers a user is eligible for in `queryProductDetailsAsync()`. RevenueCat passes through whatever Google returns in `subscriptionOptions` without additional eligibility filtering. The SDK's `defaultOption` selection logic does filter out options tagged `"rc-ignore-offer"` or `"rc-customer-center"`, but it does not perform trial eligibility checking, that is handled at the Google Play layer before data reaches the SDK.

If a user has already used their free trial for a product, Google simply will not include that trial offer in the returned options. The base plan option will be selected as `defaultOption` in that case.

Prepaid Plans

Prepaid base plans work the same way as standard base plans from the RevenueCat API perspective. The `SubscriptionOption` for a prepaid plan will have `pricingPhases` with `RecurrenceMode.NON_RECURRING`.

To support pending purchases for prepaid plans, enable the option during SDK configuration:

```
PurchasesConfiguration.Builder(context, apiKey)
    .pendingTransactionsForPrepaidPlansEnabled(true)
    .build()
```

Checking Active Subscriptions

After a successful purchase:

```
val customerInfo = result.customerInfo

// Via entitlements (recommended)
val isSubscribed = customerInfo.entitlements["pro"]?.isActive == true

// Via active subscriptions (if you need the product ID)
val activeSubs = customerInfo.activeSubscriptions
// Returns Set<String> of "subscriptionId:basePlanId" strings
```

RevenueCat computes `isActive` server-side, accounting for grace period, account hold, cancellation with remaining time, and expiry. You do not replicate that logic.

Chapter 5: Configuring the SDK

Integrating the Play Billing Library directly means setting up `BillingClient`, managing the connection lifecycle, handling `SERVICE_DISCONNECTED` errors, implementing reconnection logic, setting up the `PurchasesUpdatedListener`, querying products, and querying existing purchases on launch.

With RevenueCat, all of that is replaced by a single `configure()` call. This chapter covers the configuration options and what the SDK does automatically.

The configure() Call

```
Purchases.configure(
    PurchasesConfiguration.Builder(context, apiKey)
        .appUserID(userId)
        .build()
)
```

That is the minimum required setup. The SDK:

- Creates and manages a `BillingClient` instance internally
- Handles all connection lifecycle events
- Reconnects automatically on `SERVICE_DISCONNECTED`
- Queries existing purchases on connection
- Posts unfinished transactions to the RevenueCat backend

You do not call `startConnection()`, `endConnection()`, or check `isReady`. You do not write a reconnection handler.

appUserID

Pass your own user ID if you have an authentication system:

```
.appUserID("user_12345")
```

Pass `null` to let RevenueCat generate an anonymous ID. Anonymous users can be identified later with `Purchases.sharedInstance.logIn("user_12345")`, RevenueCat will merge the anonymous purchase history with the identified user.

If your users are always authenticated before they can purchase, pass the ID at configure time. If anonymous purchases are possible, pass `null` and call `logIn()` after authentication.

showInAppMessagesAutomatically

Defaults to `true`. When enabled, the SDK automatically calls `showInAppMessages()` when the `BillingClient` connects, which displays Google Play's in-app messaging for payment recovery (grace period and account hold prompts). This covers most of what Chapter 12 in the raw billing handbook describes.

To handle it manually:

```
.showInAppMessagesAutomatically(false)
// Then call when you want to show it:
Purchases.sharedInstance.showInAppMessagesIfNeeded(
    activity,
    inAppMessageTypes = listOf(InAppMessageType.BILLING_ISSUES) // default; customize as needed
)
```

purchasesAreCompletedBy

Defaults to `PurchasesAreCompletedBy.REVENUECAT`. In this mode, RevenueCat finishes (acknowledges or consumes) every purchase automatically.

Set to `PurchasesAreCompletedBy.MY_APP` if you have an existing billing implementation and want to use only RevenueCat's backend for analytics and entitlement tracking without changing your purchase flow. In this mode, you are responsible for acknowledging purchases within 3 days, RevenueCat will not do it for you.

entitlementVerificationMode

Defaults to `EntitlementVerificationMode.DISABLED`. Setting it to `INFORMATIONAL` enables response signature verification: the SDK verifies that `CustomerInfo` responses from the RevenueCat backend were not tampered with in transit.

```
.entitlementVerificationMode(EntitlementVerificationMode.INFORMATIONAL)
```

In `INFORMATIONAL` mode, verification failures are logged but do not block access. Set to `ENFORCED` to block access on verification failure (more secure, but can affect users if there are network issues).

diagnosticsEnabled

Defaults to `false`. When enabled, the SDK sends performance and error metrics to RevenueCat. No personal data is collected. Useful for troubleshooting production issues with RevenueCat support.

```
.diagnosticsEnabled(true)
```

Querying Existing Purchases

You do not call `queryPurchasesAsync()` at launch. The SDK does this automatically when the `BillingClient` connection is established. Any purchases that were not yet posted to the RevenueCat backend (for example, from a previous session that lost network connectivity) are posted automatically.

If you want to explicitly sync purchases, for example, to handle a restore flow, call:

```
val customerInfo = Purchases.sharedInstance.awaitRestore()
```

This posts all current Play Store purchases for the user to RevenueCat and returns updated `CustomerInfo`.

The UpdatedCustomerInfoListener

Set this listener to receive `CustomerInfo` updates pushed by the SDK:

```
Purchases.sharedInstance.updatedCustomerInfoListener =
    UpdatedCustomerInfoListener { customerInfo ->
        updateUIForEntitlements(customerInfo)
    }
```

This is called after purchases, restores, and SDK-initiated refreshes. It fires on the main thread. Do not block it.

Chapter 6: The Purchase Flow

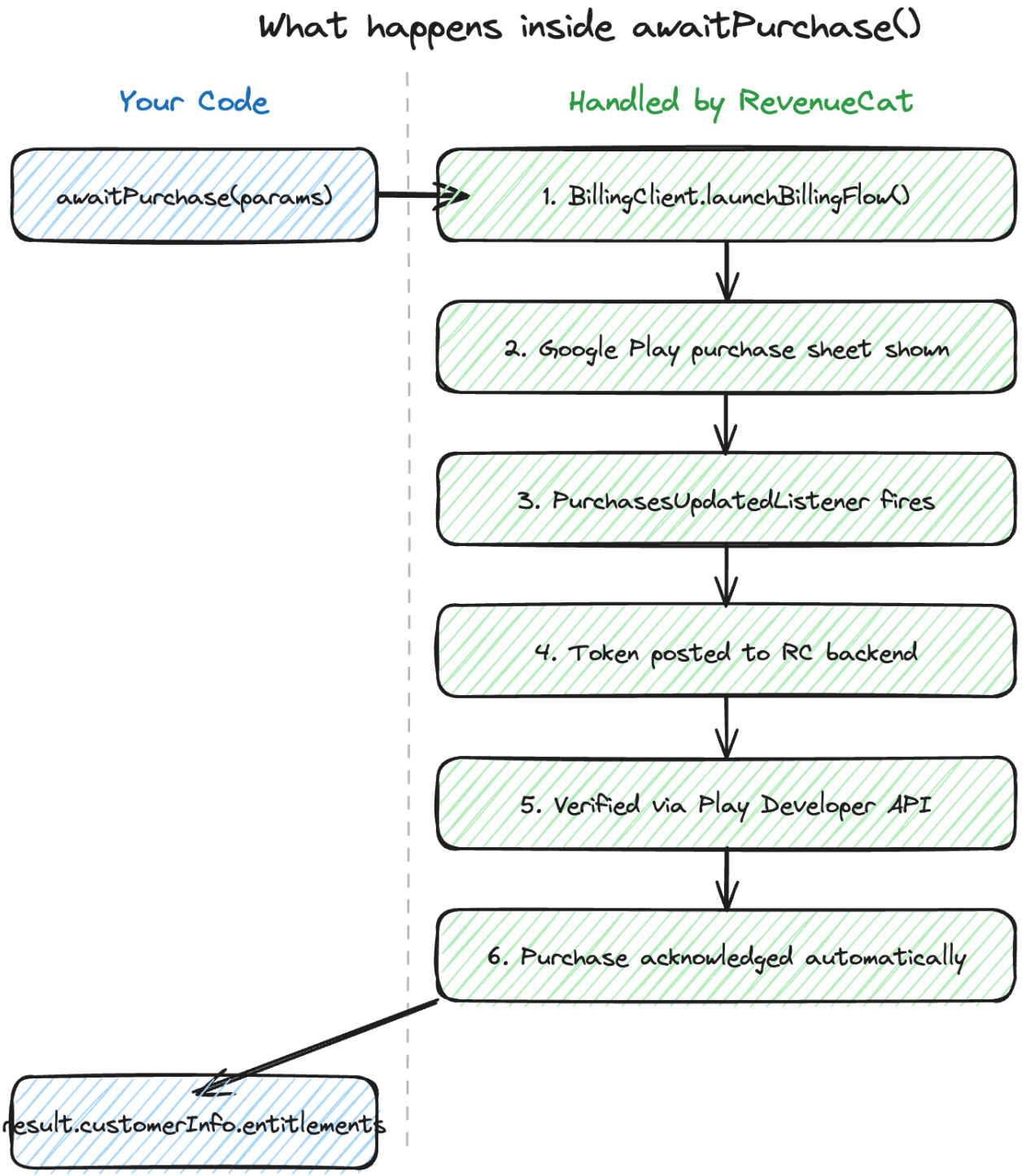
Implementing the purchase flow directly means building `BillingFlowParams`, calling `launchBillingFlow()`, handling the `PurchasesUpdatedListener` callback, dealing with result codes, posting purchases to your backend, acknowledging, and managing edge cases like duplicate purchases and mid-flow disconnections.

The RevenueCat purchase flow is:

```
val params = PurchaseParams.Builder(activity, packageToPurchase).build()
val result = Purchases.sharedInstance.awaitPurchase(params)
```

The rest is handled by the SDK. This chapter explains what happens inside that call and the decisions you still need to make.

The Complete Flow


What happens inside awaitPurchase()
Your Code
Handled by RevenueCat
awaitPurchase(params)
1. BillingClient.launchBillingFlow()
2. Google Play purchase sheet shown
3. PurchasesUpdatedListener fires
4. Token posted to RC backend
5. Verified via Play Developer API
6. Purchase acknowledged automatically
result.customerInfo.entitlements

24

```
// 1. Fetch offerings (can be cached from earlier)
val offerings = Purchases.sharedInstance.awaitOfferings()

// 2. User selects a package from your UI
val pkg = offerings.current?.monthly ?: return

// 3. Launch purchase
try {
    val result = Purchases.sharedInstance.awaitPurchase(
        PurchaseParams.Builder(activity, pkg).build()
    )
    val customerInfo = result.customerInfo

    // 4. Grant access based on updated CustomerInfo
    if (customerInfo.entitlements["pro"]?.isActive == true) {
        navigateToApp()
    }

} catch (e: PurchasesTransactionException) {
    when {
        e.userCancelled -> { /* User backed out, do nothing */ }
        e.error.code == PurchasesErrorCode.ProductAlreadyPurchasedError -> {
            showMessage("You already have this subscription")
        }
        else -> showError(e.error.message)
    }
}
```

What awaitPurchase() Does Internally

1. **Builds `BillingFlowParams` with the correct `ProductDetailsParams`**: maps your `Package` to the right offer token and product details format that Google Play expects.
2. **Calls `BillingClient.launchBillingFlow()`**: opens the Google Play purchase sheet attached to the activity you passed in `PurchaseParams`.
3. **Waits for the `PurchasesUpdatedListener` result**: suspends the coroutine until Google Play delivers the outcome, whether success, cancellation, or an error code.
4. **If result is `OK`, posts the purchase token to the RevenueCat backend**: sends the raw token from Google Play to RevenueCat so it can be recorded and verified server side.
5. **RevenueCat backend calls `purchases.subscriptionsv2.get` or `purchases.products.get` to verify**: RevenueCat hits the Google Play Developer API to confirm the purchase is genuine before granting any entitlement.
6. **SDK acknowledges or consumes the purchase**: calls `acknowledgePurchase()` for subscriptions and non-consumables, or `consumePurchase()` for consumables, within the 3 day window Google requires.
7. **Returns `PurchaseResult(storeTransaction, customerInfo)`**: hands back the verified transaction and updated entitlement state so your code can gate access immediately.

If any step fails with a retriable error, the SDK retries automatically. You do not write retry logic.

Callback Style (without coroutines)

If you prefer callbacks:

```
Purchases.sharedInstance.purchaseWith(
    PurchaseParams.Builder(activity, pkg).build(),
    onError = { error, userCancelled ->
        if (!userCancelled) showError(error.message)
    },
    onSuccess = { transaction, customerInfo ->
        navigateToApp()
    }
)
```

Personalized Pricing (EU)

For EU compliance with personalized pricing:

```
PurchaseParams.Builder(activity, pkg)
    .isPersonalizedPrice(true)
    .build()
```

This passes `isOfferPersonalized = true` to `BillingFlowParams`, which shows the "This price has been customized for you" notice on the Google Play purchase dialog.

Purchasing a Specific SubscriptionOption

When you pass a `Package` to `PurchaseParams`, the SDK uses `defaultOption` (best available offer). To purchase a specific offer:

```
val subscriptionOption = pkg.product.subscriptionOptions
    ?.firstOrNull { it.tags.contains("promo_50_off") }
    ?: pkg.product.defaultOption
    ?: return

val params = PurchaseParams.Builder(activity, subscriptionOption).build()
```

What the StoreTransaction Contains

`PurchaseResult.storeTransaction` is a `StoreTransaction`:

```
val transaction: StoreTransaction = result.storeTransaction
transaction.orderId          // String?, null for restored purchases
transaction.purchaseToken    // the raw Play purchase token
transaction.productIds       // list of product IDs purchased
transaction.purchaseTime     // epoch millis
transaction.type             // ProductType.SUBS or INAPP
```

You typically do not need to use this directly. The `customerInfo` in the same result object is the canonical source of truth for entitlements.

Restoring Purchases

For users who reinstall the app or switch devices:

```
try {
    val customerInfo = Purchases.sharedInstance.awaitRestore()
    if (customerInfo.entitlements["pro"]?.isActive == true) {
        navigateToApp()
    } else {
        showMessage("No active purchases found")
    }
} catch (e: PurchasesException) {
    showError(e.error.message)
}
```

`awaitRestore()` calls `queryPurchasesAsync()` internally, posts all found purchases to RevenueCat, and returns the updated `CustomerInfo`.

Chapter 7: Subscription Upgrades and Downgrades

Handling plan changes directly means choosing among six replacement modes, understanding how each affects billing timing, building `SubscriptionProductReplacementParams`, and managing the `linkedPurchaseToken` chain on your backend after every plan change.

With RevenueCat, the same replacement modes are available and the same billing behaviors apply. The difference is that `PurchaseParams` handles the parameters in a single builder, and RevenueCat manages the `linkedPurchaseToken` chain on the backend for you.

The Replacement Modes

RevenueCat exposes Google's replacement modes through `GoogleReplacementMode`:

GOOGLEREPLACEMENTMODE	BEHAVIOR
`WITH_TIME_PRORATION`	Immediate switch, remaining time credited
`CHARGE_PRORATED_PRICE`	Immediate switch, prorated charge now
`CHARGE_FULL_PRICE`	Immediate switch, full charge now
`WITHOUT_PRORATION`	Immediate switch, no charge until next renewal
`DEFERRED`	Switch at next renewal

Note: `KEEP_EXISTING` is not currently in `GoogleReplacementMode`. For add-on style multi-line purchases, use `PurchaseParams.Builder.addOnPackages()` (experimental API).

Performing a Plan Change

```
// Derive the current product ID from CustomerInfo, do not hardcode it
val currentProductId = customerInfo.activeSubscriptions
    .firstOrNull()
    ?.substringBefore(":")  // activeSubscriptions entries are "productId:basePlanId"
    ?: return

val newPackage = offerings.current
    ?.availablePackages
    ?.firstOrNull { it.identifier == "premium_monthly_package" }
    ?: return

val params = PurchaseParams.Builder(activity, newPackage)
    .oldProductId(currentProductId)
    .googleReplacementMode(GoogleReplacementMode.WITH_TIME_PRORATION)
    .build()

try {
    val result = Purchases.sharedInstance.awaitPurchase(params)
    // result.customerInfo reflects the new subscription
} catch (e: PurchasesTransactionException) {
    if (!e.userCancelled) showError(e.error.message)
}
```

The `oldProductId` should be the subscription product ID only, do not include the base plan ID. If you pass a string like `"basic_monthly:monthly_plan"`, RevenueCat strips the base plan suffix automatically.

Choosing a Replacement Mode

The same logic as the raw billing handbook applies:

SCENARIO	RECOMMENDED MODE
Standard upgrade	`WITH_TIME_PRORATION`
Upgrade, keep billing date	`CHARGE_PRORATED_PRICE`
Switch to/from prepaid	`CHARGE_FULL_PRICE`
Upgrade during free trial	`CHARGE_PRORATED_PRICE`
Downgrade	`DEFERRED`

The default mode in `PurchaseParams` is `WITHOUT_PRORATION` if you do not set `googleReplacementMode()`.

The linkedPurchaseToken Chain

When a user upgrades, Google creates a new purchase token linked to the old one via `linkedPurchaseToken`. On the raw billing stack, your backend must follow this chain to identify the current active token and invalidate old ones.

With RevenueCat, the backend handles this automatically. When RevenueCat verifies the new purchase token and sees a `linkedPurchaseToken`, it resolves the chain, marks the old subscription as replaced, and attributes both tokens to the same user. Your app just reads `customerInfo.entitlements["pro"]?.isActive`, the correct value is already computed.

You do not write token chain traversal code.

Deferred Upgrades

With `DEFERRED` mode, Google issues a new purchase token immediately even though the plan switch does not take effect until the next renewal. The new token has two line items: one active (current plan) and one pending (new plan).

RevenueCat handles this correctly. During the deferral window, `customerInfo.entitlements` reflects the current plan. After the renewal fires and Google sends an RTDN, RevenueCat updates the entitlement to reflect the new plan. Your app just reads `customerInfo`, the state is always correct.

Chapter 8: Error Handling

Handling billing errors directly means dealing with every `BillingResponseCode`, categorizing them into retriable and non-retriable groups, and building an exponential backoff retry system. Different response codes require different actions: some require reconnection, some require user intervention, some should be silently ignored.

With RevenueCat, you handle one error type: `PurchasesError`. The SDK has already absorbed the `BillingResponseCode` layer.

PurchasesError

Every failure in the RevenueCat SDK surfaces as a `PurchasesError`:

```
public class PurchasesError(
    val code: PurchasesErrorCode,
    val underlyingErrorMessage: String? = null,
) {
    val message: String // human-readable description of the error code
}
```

`PurchasesErrorCode` is a cross-platform enum with human-readable codes. The `message` property on `PurchasesError` gives a description suitable for logging (not for showing directly to users).

The Error Codes You Handle in Practice

Most of your error handling code deals with these codes:

CODE	MEANING	WHAT TO DO
`PurchaseCancelledError`	User backed out of the flow	Do nothing; `userCancelled` will also be `true`
`ProductAlreadyPurchasedError`	This product is already active for the user	Refresh `CustomerInfo` and check entitlements
`PaymentPendingError`	Purchase entered pending state	Show pending message, wait for `UpdatedCustomerInfoListener`
`NetworkError`	Request failed due to connectivity	Ask user to retry
`StoreProblemError`	Issue with Google Play	Ask user to retry or update Play Store
`PurchaseNotAllowedError`	Device or user cannot make purchases	Show appropriate message
`IneligibleError`	User ineligible for the offer	Show base plan instead

Handling Purchase Errors

```
try {
    val result = Purchases.sharedInstance.awaitPurchase(params)
    handleSuccess(result.customerInfo)
} catch (e: PurchasesTransactionException) {
    // PurchasesTransactionException adds userCancelled
    if (e.userCancelled) return

    when (e.error.code) {
        PurchasesErrorCode.PaymentPendingError ->
            showPendingMessage()
        PurchasesErrorCode.ProductAlreadyPurchasedError -> {
            val info = Purchases.sharedInstance.awaitCustomerInfo()
            handleSuccess(info)
        }
        PurchasesErrorCode.NetworkError ->
            showRetryDialog()
        else ->
            showGenericError(e.error.message)
    }
}
```

`PurchasesTransactionException` is a subtype of `PurchasesException` that adds `userCancelled: Boolean`. Use it for `awaitPurchase()` only. For `awaitRestore()`, catch `PurchasesException` (the base class), restore does not produce a `PurchasesTransactionException`.

Handling Non-Purchase Errors

For `awaitOfferings()`, `awaitGetProducts()`, and `awaitCustomerInfo()`, catch `PurchasesException`:

```
try {
    val offerings = Purchases.sharedInstance.awaitOfferings()
    displayOfferings(offerings)
} catch (e: PurchasesException) {
    when (e.error.code) {
        PurchasesErrorCode.NetworkError ->
            showOfflineFallback()
        else ->
            logError(e.error)
    }
}
```

Retry Logic

You do not write retry logic for transient errors. The SDK retries `BillingClient` operations internally with backoff for `SERVICE_UNAVAILABLE` and `ERROR` response codes. The errors that surface to your code are already past the retry budget.

For `NetworkError` on RevenueCat API calls (the calls that post purchases or fetch `CustomerInfo`), the SDK also retries internally. A `NetworkError` reaching your catch block means retries were exhausted.

The only retries you implement are user-initiated: a "Try Again" button that re-calls the relevant SDK method.

User-Facing Error Messages

Do not show `e.error.message` to users, it is a technical description. Map to user-facing strings:

```
fun userFacingMessage(error: PurchasesError): String = when (error.code) {
    PurchasesErrorCode.PurchaseCancelledError ->
        "" // show nothing
    PurchasesErrorCode.NetworkError ->
        "Please check your internet connection and try again."
    PurchasesErrorCode.StoreProblemError ->
        "There was a problem with Google Play. Please try again."
    PurchasesErrorCode.ProductAlreadyPurchasedError ->
        "You already have this subscription."
    PurchasesErrorCode.PaymentPendingError ->
        "Your payment is being processed. We'll notify you when it completes."
    else ->
        "Something went wrong. Please try again."
}
```

Chapter 9: Backend Architecture

Building purchase verification from scratch means standing up a server that calls the Google Play Developer API, manages service account credentials, validates receipts, grants entitlements, and protects against token reuse. It is a non-trivial backend service with Google Cloud credential management.

With RevenueCat, you do not build that backend.

What RevenueCat Replaces

RevenueCat's backend is your receipt verification server. After every purchase, the RevenueCat SDK posts the purchase token to RevenueCat's backend, which:

1. Calls `purchases.subscriptionsv2.get` or `purchases.products.get` on the Google Play Developer API
2. Validates the receipt is genuine and matches the expected product
3. Records the transaction in RevenueCat's database
4. Returns `CustomerInfo` to the SDK

Your app never calls the Google Play Developer API. Your server never calls it either, unless you are building a custom integration that bypasses the SDK entirely.

Your Backend's Role with RevenueCat

If you have a backend server, its billing-related responsibilities are:

1. **Receive RevenueCat webhooks** (optional but recommended): RevenueCat sends normalized events for every purchase, renewal, cancellation, grace period, and expiry. Your server consumes these to maintain its own record of subscription state.

2. **Call the RevenueCat REST API** (optional): Instead of calling the Google Play Developer API, you can call `https://api.revenuecat.com/v1/subscribers/{app_user_id}` to get the current `CustomerInfo` for any user. This is authenticated with your RevenueCat secret API key.

3. **Grant/revoke access in your own database**: Based on webhook events or REST API responses, your server updates its own records. This is the same responsibility as before, but the trigger is a RevenueCat webhook instead of a Google Cloud Pub/Sub message.

The RevenueCat REST API

Your backend can check a user's entitlement status without calling Google:

```
GET https://api.revenuecat.com/v1/subscribers/{app_user_id}
Authorization: Bearer {your_secret_api_key}
```

> **Important:** The secret API key is found under RevenueCat dashboard → Project Settings → API Keys. It is **not** the same as the Android public SDK key embedded in your app. Never include the secret API key in client-side code.

The response includes the same `CustomerInfo` structure the SDK returns. This is useful for server-side access decisions, for example, in an API endpoint that serves premium content.

Do You Still Need a Backend?

That depends on your app. If your app only needs to gate features inside the client app, you may not need any server-side component at all. RevenueCat's `CustomerInfo` on the client is verified server-side before it reaches your app, and the optional `EntitlementVerificationMode.INFORMATIONAL` or `.ENFORCED` setting adds cryptographic signature verification.

If your app serves premium content from a server, API responses, files, or data, you should verify entitlement on the server using RevenueCat webhooks or the REST API. Do not trust the client alone for server-side access decisions.

What You Still Own

- Your user authentication system
- Your database of users and their access levels
- Your API endpoints that serve premium content
- The webhook receiver that processes RevenueCat events

What you do not own:

- Google Play Developer API credentials management
- Receipt verification code
- `linkedPurchaseToken` chain traversal
- Subscription state computation across seven states

Chapter 10: Webhooks

Receiving subscription event notifications from scratch means setting up a Cloud Pub/Sub topic, creating a push subscription, deploying an endpoint to receive messages, decoding base64 payloads, dispatching by notification type, and implementing idempotency to handle duplicate deliveries. There are 22 different notification types, each mapping to a specific subscription state change.

RevenueCat webhooks replace all of that. You get one webhook endpoint to configure, and one normalized event schema across all stores.

Setting Up Webhooks

In the RevenueCat dashboard, go to **Integrations → Webhooks**. Add your endpoint URL. RevenueCat sends an HTTP POST with a JSON payload for every relevant event.

There is no Cloud Pub/Sub to configure. No base64 decoding. No service account credentials for your endpoint. RevenueCat sends standard HTTPS POST requests with a `X-RevenueCat-Signature` header for verification.

The Event Schema

Every RevenueCat webhook has this shape:

```
{
  "api_version": "1.0",
  "event": {
    "id": "evt_xxxxxxxxxxxxxxxxxxxxxxxx",
    "type": "INITIAL_PURCHASE",
    "app_user_id": "user_12345",
    "aliases": ["$RCAnonymousID:abc123"],
    "product_id": "premium_monthly",
    "period_type": "NORMAL",
    "purchased_at_ms": 1700000000000,
    "expiration_at_ms": 1702592000000,
    "store": "PLAY_STORE",
    "environment": "PRODUCTION",
    "entitlement_ids": ["pro_access"],
    "transaction_id": "GPA.1234-5678-9012-34567",
    ...
  }
}
```

The key field is `entitlement_ids`, RevenueCat has already mapped the product to your entitlement. Your webhook handler does not need to know which product maps to which access level. It just checks which entitlements were affected.

Event Types

REVENUECAT EVENT	EQUIVALENT GOOGLE RTDN
`INITIAL_PURCHASE`	`SUBSCRIPTION_PURCHASED` (4)
`RENEWAL`	`SUBSCRIPTION_RENEWED` (2)
`CANCELLATION`	`SUBSCRIPTION_CANCELED` (3)
`UNCANCELLATION`	`SUBSCRIPTION_RESTARTED` (7)
`BILLING_ISSUE`	`SUBSCRIPTION_IN_GRACE_PERIOD` (6) or `SUBSCRIPTION_ON_HOLD` (5)
`SUBSCRIBER_ALIAS`	N/A (identity merge)
`EXPIRATION`	`SUBSCRIPTION_EXPIRED` (13)
`PRODUCT_CHANGE`	`SUBSCRIPTION_ITEMS_CHANGED` (17)
`TRANSFER`	N/A (user ID transfer)

A Minimal Webhook Handler

```
// Server-side (example in Kotlin/Ktor)
post("/revenuecat/webhook") {
    val body = call.receiveText()
    val signature = call.request.headers["X-RevenueCat-Signature"]

    if (!verifySignature(body, signature, webhookSecret)) {
        call.respond(HttpStatusCode.Unauthorized)
        return@post
    }

    val event = Json.decodeFromString<RevenueCatEvent>(body)

    when (event.type) {
        "INITIAL_PURCHASE", "RENEWAL", "UNCANCELLATION" ->
            db.grantEntitlements(event.appUserId, event.entitlementIds)
        "EXPIRATION" ->
            // Subscription has ended, revoke access immediately
            db.revokeEntitlements(event.appUserId, event.entitlementIds)
        "CANCELLATION" ->
            // User has opted out of renewal but still has access until expirationAtMs.
            // Schedule revocation, do NOT revoke immediately.
            db.scheduleEntitlementRevocation(event.appUserId, event.entitlementIds, event.expiration
AtMs)
        "BILLING_ISSUE" ->
            db.flagBillingIssue(event.appUserId)
    }

    call.respond(HttpStatusCode.OK)
}
```

Idempotency

RevenueCat may deliver the same event more than once. Use `event.id` as an idempotency key. If you have already processed an event with that ID, return 200 without re-processing.

Retries

RevenueCat retries failed webhook deliveries with exponential backoff. Return a 2xx response immediately if you received the event successfully, even if your processing is asynchronous. If your handler takes too long, RevenueCat may time out and retry.

Chapter 11: Subscription States

Tracking subscription state from scratch means mapping seven distinct states (ACTIVE, IN_GRACE_PERIOD, ON_HOLD, PAUSED, CANCELED, EXPIRED, PENDING), knowing which grant access and which do not, and building a state machine that processes RTDNs and makes correct entitlement decisions.

With RevenueCat, you do not implement a state machine. You read one boolean.

The Entitlement Check

```
val customerInfo = Purchases.sharedInstance.awaitCustomerInfo()
val hasAccess = customerInfo.entitlements["pro_access"]?.isActive == true
```

`isActive` is `true` when:

- The subscription is `ACTIVE`
- The subscription is `IN_GRACE_PERIOD` (grace period grants access)
- The subscription is `CANCELED` but `expirationDate` is in the future

`isActive` is `false` when:

- The subscription is `ON_HOLD`
- The subscription is `PAUSED`
- The subscription is `EXPIRED`
- The subscription is `PENDING` (no payment yet)

RevenueCat computes this on the backend using the `SubscriptionPurchaseV2` resource from the Google Play Developer API. You do not write the seven-state access decision function.

Reading EntitlementInfo

The `EntitlementInfo` object gives you additional context if you need it:

```
val entitlement = customerInfo.entitlements["pro_access"] ?: return
val isActive = entitlement.isActive
val willRenew = entitlement.willRenew         // false if canceled
val expirationDate = entitlement.expirationDate  // null for lifetime
val periodType = entitlement.periodType       // NORMAL, TRIAL, INTRO, PREPAID
val billingIssueAt = entitlement.billingIssueDetectedAt // non-null during grace period / on hold
val unsubscribeAt = entitlement.unsubscribeDetectedAt   // non-null when canceled
val store = entitlement.store                 // PLAY_STORE
```

`billingIssueDetectedAt` being non-null means the user is in grace period or account hold. Show them a payment update prompt.

`unsubscribeDetectedAt` being non-null means the user has canceled but may still have active access until `expirationDate`.

Showing State-Aware UI

Use `EntitlementInfo` to drive UI decisions beyond the basic access gate. The same `isActive` boolean that grants access also lets you surface contextual prompts, such as a renewal reminder for users who have canceled but are still within their paid period.

```
fun updateUI(entitlement: EntitlementInfo?) {
    if (entitlement == null || !entitlement.isActive) {
        showSubscribeScreen()
        return
    }

    showPremiumContent()

    when {
        entitlement.billingIssueDetectedAt != null ->
            showBillingIssueWarning()
        entitlement.unsubscribeDetectedAt != null ->
            entitlement.expirationDate?.let { showExpiryNotice(it) }
        !entitlement.willRenew ->
            showNonRenewingNotice()
    }
}
```

User Identification

For multi-device support and linking purchases to your user accounts, pass your user ID at login:

```
// When user logs in
val loginResult = Purchases.sharedInstance.awaitLogIn("your_user_id")
val customerInfo = loginResult.customerInfo
val created = loginResult.created  // true if this is a new RevenueCat user

// When user logs out
Purchases.sharedInstance.logOut()
// After logOut, an anonymous user session starts
```

`awaitLogIn()` merges any anonymous purchases made before login with the identified user. `created` is `true` if this is a new RevenueCat user.

CustomerInfo Caching

`CustomerInfo` is cached on disk. Subsequent calls to `awaitCustomerInfo()` return the cache immediately, then fetch from the network in the background if the cache is stale. This means your entitlement checks are fast even offline.

For scenarios where you need guaranteed fresh data (for example, after a support interaction that granted a promotional subscription), use:

```
val customerInfo = Purchases.sharedInstance.awaitCustomerInfo(
    fetchPolicy = CacheFetchPolicy.FETCH_CURRENT
)
```

Listening for Changes

The `UpdatedCustomerInfoListener` fires whenever the SDK updates its cache:

```
Purchases.sharedInstance.updatedCustomerInfoListener =
    UpdatedCustomerInfoListener { customerInfo ->
        val isActive = customerInfo.entitlements["pro_access"]?.isActive == true
        updateAccessGate(isActive)
    }
```

This fires after purchases, restores, and background refreshes. It does not fire if the SDK starts with a cache hit and nothing changed. Always call `awaitCustomerInfo()` on launch as well.

Chapter 12: Payment Recovery

Handling payment recovery from scratch means detecting grace periods, building a `checkGracePeriodStatus()` function, displaying in-app messages at the right moments, understanding the silent grace period, and responding to `SUBSCRIPTION_IN_GRACE_PERIOD` and `SUBSCRIPTION_ON_HOLD` RTDNs on the backend.

With RevenueCat, two of these concerns require code from you. The rest are automatic.

In-App Messages: Automatic by Default

RevenueCat shows Google Play's in-app payment recovery messages automatically. The SDK calls `showInAppMessages()` when `BillingClient` connects, which displays the snackbar prompting users to fix their payment method during grace period and account hold.

You do not call any in-app messaging API. This is on by default with:

```
PurchasesConfiguration.Builder(context, apiKey)
    .showInAppMessagesAutomatically(true) // this is the default
    .build()
```

If you want to control when the message appears, for example, only on certain screens, disable the automatic behavior and call it manually:

```
PurchasesConfiguration.Builder(context, apiKey)
    .showInAppMessagesAutomatically(false)
    .build()

// Call where appropriate
Purchases.sharedInstance.showInAppMessagesIfNeeded(activity)
```

Detecting Payment Issues in Your UI

To show your own payment recovery UI (a banner, a dialog, or a dedicated screen), check `EntitlementInfo.billingIssueDetectedAt`:

```
val entitlement = customerInfo.entitlements["pro_access"]
if (entitlement?.isActive == true &&
    entitlement.billingIssueDetectedAt != null) {
    showPaymentIssueWarning()
}
```

`billingIssueDetectedAt` is non-null from the moment RevenueCat receives the `SUBSCRIPTION_IN_GRACE_PERIOD` RTDN until the payment issue is resolved. This covers both grace period (access retained) and account hold (access

revoked). Since `isActive` is still `true` during grace period but `false` during account hold, you can distinguish:

```
val entitlement = customerInfo.entitlements["pro_access"]
when {
    entitlement == null || !entitlement.isActive ->
        showSubscribeScreen()
    entitlement.billingIssueDetectedAt != null && entitlement.isActive ->
        showGracePeriodWarning() // user has access but payment is failing
    entitlement.billingIssueDetectedAt != null && !entitlement.isActive ->
        showAccountHoldScreen() // access revoked, prompt to fix payment
    else ->
        showPremiumContent()
}
```

Opening Payment Management

Send users to fix their payment method:

```
customerInfo.managementURL?.let { url ->
    startActivity(Intent(Intent.ACTION_VIEW, url))
}
```

`managementURL` is a `Uri?` pointing to the subscription management page in Google Play. RevenueCat populates this automatically.

Backend: Webhooks Handle the Rest

On your backend, the `BILLING_ISSUE` webhook event fires when a subscription enters grace period or account hold. The event includes the `app_user_id` and `entitlement_ids`, so you can flag the user's account to show payment recovery prompts.

```
{
  "type": "BILLING_ISSUE",
  "app_user_id": "user_12345",
  "entitlement_ids": ["pro_access"],
  "expiration_at_ms": 1702592000000
}
```

Your webhook handler flags the user's account. No RTDN decoding, no state machine update needed.

Chapter 13: Cancellations, Pauses, and Winback

Handling cancellations from scratch means writing code for user-initiated cancellation, developer-initiated cancellation via API, system cancellation, pause states, restoring canceled subscriptions, resubscribing, subscription deferrals, and revocations.

Most of these are passive events your app observes rather than initiates. RevenueCat reflects them all in `CustomerInfo`. The code you write to handle them is small.

Detecting Cancellation

```
val entitlement = customerInfo.entitlements["pro_access"]

if (entitlement?.unsubscribeDetectedAt != null) {
    // User has canceled but may still have active access
    val expiry = entitlement.expirationDate
    if (entitlement.isActive && expiry != null) {
        showCancellationBanner(expiry)
    }
}
```

`unsubscribeDetectedAt` is set when RevenueCat receives the `SUBSCRIPTION_CANCELED` RTDN. `isActive` remains `true` until the billing period ends. Your app shows a "subscription ends on [date]" message without computing access manually.

Backend Webhooks for Cancellation

Your backend receives:

```
{ "type": "CANCELLATION", "cancel_reason": "UNSUBSCRIBE", "expiration_at_ms": 1702592000000 }
```

`cancel_reason` can be `UNSUBSCRIBE` (user), `BILLING_ERROR` (payment failure), `DEVELOPER_INITIATED`, or `PRICE_INCREASE`. Use this to segment churned users for win-back campaigns.

Paused Subscriptions

RevenueCat sets `isActive = false` when a subscription is paused. You do not need to call `queryPurchasesAsync()` with `setIncludeSuspendedSubscriptions(true)`. The state comes from RevenueCat's server-side subscription state.

```
// Just check isActive — RevenueCat handles the state resolution
val hasPremiumAccess = customerInfo.entitlements["pro_access"]?.isActive == true
```

If you want to show a "your subscription is paused, resuming on [date]" message, call the RevenueCat REST API (`GET /v1/subscribers/{app_user_id}`) and read the `paused_expiration_time_ms` field for the subscription. RevenueCat does not currently send a dedicated webhook event for subscription pauses, the pause state change is reflected via `CustomerInfo` when RevenueCat processes the Google Play RTDN.

Restoring a Canceled Subscription (Before Expiry)

If a user cancels and then re-subscribes before expiry (Google's "resubscribe before expiry" flow), RevenueCat detects the reactivation via the `SUBSCRIPTION_RESTARTED` RTDN and sets `willRenew = true` again. The `unsubscribeDetectedAt` is cleared. Your app reads `customerInfo` and the state is correct, no special handling needed.

Resubscribing After Expiry

A resubscription after expiry sends a `RENEWAL` webhook event, not `INITIAL_PURCHASE` . `INITIAL_PURCHASE` fires only for a user's first-ever purchase of a product. Ensure your webhook handler grants entitlement access on `RENEWAL` as well as `INITIAL_PURCHASE` , a handler that only listens for `INITIAL_PURCHASE` will not restore access for users who resubscribe after expiry.

Subscription Deferral (Extending Access)

Subscription deferral is a developer-side operation done through the Google Play Developer API. RevenueCat does not expose a deferral API from the SDK or dashboard. If you need to defer a subscription, you call `purchases.subscriptionsv2.defer` on the Google Play Developer API directly from your backend.

After deferral, RevenueCat will receive the updated expiry time via RTDN processing and reflect it in `CustomerInfo` automatically.

Revoking a Subscription

Revocation is also done through the Google Play Developer API directly (`purchases.subscriptionsv2.revoke`). RevenueCat processes the resulting RTDN and immediately sets `isActive = false` for the entitlement.

Opening the Subscription Management Screen

Let users cancel or manage their subscription through Google Play:

```
customerInfo.managementURL?.let { url ->
    startActivity(Intent(Intent.ACTION_VIEW, url))
}
```

Win-Back: No SDK Changes Required

Win-back campaigns are configured in the Google Play Console (Play-managed win-back) or in the RevenueCat dashboard (custom targeting). You do not write code to implement win-back. When a win-back offer is accepted, it arrives as a normal purchase and surfaces in `CustomerInfo` like any other subscription.

Chapter 14: Price Changes

Managing price changes from scratch means configuring price increases in the Play Console, understanding how new subscribers and legacy cohorts are affected differently, handling the `SUBSCRIPTION_PRICE_CHANGE_UPDATED` RTDN, managing consent for opt-in price changes, and tracking edge cases for subscribers on old price cohorts.

With RevenueCat, there is nothing to implement in your app or backend for most price change scenarios.

What RevenueCat Handles Automatically

RevenueCat processes `SUBSCRIPTION_PRICE_CHANGE_UPDATED` and related RTDNs on its backend. When a subscriber's price cohort changes, or when consent is resolved, RevenueCat updates the `CustomerInfo` accordingly. Your app reads `customerInfo.entitlements["pro"]?.isActive` and gets the correct result regardless of which price cohort the user is on.

Price Increase Consent

For opt-in price increases (where Google requires the user to consent before the price change takes effect), Google sends the in-app consent dialog to the user through Google Play's own UI. You do not implement a consent flow in your app.

If the user consents: the subscription continues at the new price, the RTDN fires, RevenueCat records it.

If the user does not consent: the subscription cancels at the end of the current period, the `SUBSCRIPTION_CANCELED` RTDN fires, RevenueCat records it. Your webhook receives a `CANCELLATION` event.

Fetching Updated Prices

When you update product prices in the Play Console, the `StoreProduct.price` returned by `awaitOfferings()` or `awaitGetProducts()` reflects the current price for new subscribers. Existing subscribers on a legacy cohort continue to pay their old price, this is a Google Play concern, not something visible in the RevenueCat SDK.

What Requires Action

If you display the subscription price in your paywall, the price from `pkg.product.price.formatted` is always correct for new purchases. You do not need to handle legacy cohort pricing in your UI, that is between Google and the existing subscriber.

If you want to notify users of an upcoming price increase (for example, an in-app banner before the change takes effect), you can use a RevenueCat webhook event or a Targeting rule to show a specific Offering to affected users. This is an optional enhancement, not a requirement.

Chapter 15: Security

Building a secure purchase system from scratch means implementing server-side receipt verification, protecting against client-side tampering, validating purchase tokens, and ensuring the client never makes its own access decisions.

With RevenueCat, server-side verification is automatic, every purchase is verified before `awaitPurchase()` returns. This chapter covers the additional security features RevenueCat provides.

Trusted Entitlements (Response Verification)

The default configuration trusts `CustomerInfo` responses from RevenueCat's backend without cryptographic verification. An attacker with network access could theoretically intercept and modify the response.

Enable response signature verification to prevent this:

```
PurchasesConfiguration.Builder(context, apiKey)
    .entitlementVerificationMode(EntitlementVerificationMode.INFORMATIONAL)
    .build()
```

In `INFORMATIONAL` mode, RevenueCat signs every `CustomerInfo` response. The SDK verifies the signature and attaches the result to `EntitlementInfos.verification`:

```
val verification = customerInfo.entitlements.verification
when (verification) {
    VerificationResult.VERIFIED -> { /* response is authentic */ }
    VerificationResult.FAILED -> { /* possible tampering, log and alert */ }
    VerificationResult.NOT_REQUESTED -> { /* verification disabled */ }
    VerificationResult.VERIFIED_ON_DEVICE -> { /* verified locally */ }
}
```

In `INFORMATIONAL` mode, a failed verification is logged but does not block access. Change to `ENFORCED` to block access on verification failure:

```
.entitlementVerificationMode(EntitlementVerificationMode.ENFORCED)
```

In `ENFORCED` mode, `EntitlementInfo.isActive` returns `false` for any response that fails signature verification. This is the highest client-side security setting, but be aware that legitimate network issues (proxies, some VPNs) could cause verification failures for real users.

The Server Is Always the Authority

Even with `EntitlementVerificationMode.ENFORCED`, do not make server-side access decisions based solely on what the client reports. For any API that serves premium content, verify entitlement using the RevenueCat REST API or your own database updated by webhooks.

API Key Security

Your RevenueCat Android API key is embedded in your app binary. This is expected, it is a public API key that allows reading and purchasing but not administrative operations. Guard your **secret API key** (used for the REST API) on your server and never embed it in the client.

Anonymous Users

RevenueCat's anonymous user IDs (`$RCAnonymousID:...`) are generated on the device and are not secret. They are simply identifiers, not credentials. If a user shares their device, both users can read the anonymous purchase history. For production apps with real users, always identify users with your own authenticated user IDs by calling `Purchases.sharedInstance.logIn("your_user_id")`.

Purchase Token Security

The purchase token is posted to RevenueCat's backend immediately after the purchase. If the network call fails, the SDK retries on the next app launch. The token is verified against Google's servers by RevenueCat, a fabricated or reused token will fail verification and not grant entitlements.

What RevenueCat Handles

- Receipt validation against Google Play Developer API on every purchase
- Protection against token reuse (RevenueCat deduplicates tokens)
- Cryptographic response signing (`INFORMATIONAL` / `ENFORCED` modes)
- Secure transmission to RevenueCat backend (HTTPS)

What You Handle

- Identifying users with authenticated IDs instead of relying on anonymous IDs
- Server-side entitlement checks using RevenueCat REST API for premium content
- Protecting your RevenueCat secret API key on your server

Chapter 16: Testing

Testing billing from scratch means setting up license tester accounts, navigating Play Console test tracks, and dealing with shortened subscription intervals that still require real Google Play sandbox flows. RevenueCat gives you a faster alternative: the **Test Store**.

What Is the Test Store

The Test Store is a RevenueCat-managed testing environment that runs entirely without Google Play. When your app is configured with a `test_` API key, calling `awaitPurchase()` shows a local dialog instead of launching the Google Play purchase sheet:

- **Successful Purchase**, SDK returns a successful `PurchaseResult` with active entitlements
- **Failed Purchase**, SDK throws a `PurchasesTransactionException` with a payment failure error
- **Cancel**, SDK throws with `userCancelled = true`

No sandbox accounts. No test tracks. No network dependency on Google's servers. Every outcome is deterministic.

Setup

1. Enable the Test Store

In the RevenueCat dashboard → your app → **Apps & providers** → **Create Test Store**. This generates a separate API key prefixed with `test_`.

2. Use the Test Key in Debug Builds

```
// build.gradle.kts
android {
    buildTypes {
        debug {
            buildConfigField("String", "RC_API_KEY", "\"test_YOUR_TEST_KEY_HERE\"")
        }
        release {
            buildConfigField("String", "RC_API_KEY", "\"goog_YOUR_PRODUCTION_KEY_HERE\"")
        }
    }
}
```

3. Initialize with the Build Config Key

```
class MyApplication : Application() {
    override fun onCreate() {
        super.onCreate()

        if (BuildConfig.DEBUG) Purchases.logLevel = LogLevel.DEBUG

        Purchases.configure(
            PurchasesConfiguration.Builder(this, BuildConfig.RC_API_KEY).build()
        )
    }
}
```

That is all the setup required. The same `awaitPurchase()` call you write for production will show the Test Store dialog in debug builds.

Testing Each Purchase Outcome

The Test Store dialog lets you select any outcome before your code runs: successful purchase, cancellation, or specific error codes like `BILLING_UNAVAILABLE` and `ITEM_ALREADY_OWNED`. This means you can exercise every branch of your error handling code without needing to trigger real billing failures.

```
// In your paywall or test, this call triggers the Test Store dialog in debug builds
try {
    val result = Purchases.sharedInstance.awaitPurchase(
        PurchaseParams.Builder(activity, pkg).build()
    )
    val customerInfo = result.customerInfo
    val isActive = customerInfo.entitlements["pro_access"]?.isActive == true
    // → Test: select "Successful Purchase" in the dialog

} catch (e: PurchasesTransactionException) {
    if (e.userCancelled) {
        // → Test: select "Cancel" in the dialog
    } else {
        showError(e.error.message)
        // → Test: select "Failed Purchase" in the dialog
    }
}
```

Unit Testing

For unit tests, wrap `Purchases` behind an interface so you can mock it without the SDK. The `Purchases` singleton is not directly mockable, and unit tests should not depend on a real SDK instance or network. Introducing a thin

`BillingService` interface between your ViewModels and the SDK lets you inject a fake in tests and verify your business logic independently of RevenueCat's internals.

```
interface BillingService {
    suspend fun getOfferings(): Offerings
    suspend fun purchase(activity: Activity, pkg: Package): CustomerInfo
    suspend fun getCustomerInfo(): CustomerInfo
}

class RevenueCatBillingService : BillingService {
    override suspend fun getOfferings() =
        Purchases.sharedInstance.awaitOfferings()

    override suspend fun purchase(activity: Activity, pkg: Package): CustomerInfo {
        val result = Purchases.sharedInstance.awaitPurchase(
            PurchaseParams.Builder(activity, pkg).build()
        )
        return result.customerInfo
    }

    override suspend fun getCustomerInfo() =
        Purchases.sharedInstance.awaitCustomerInfo()
}
```

Your ViewModel takes a `BillingService`. In tests, inject a mock:

```
class PaywallViewModelTest {
    private val billing = mockk<BillingService>()
    private val viewModel = PaywallViewModel(billing)

    @Test
    fun `purchase success grants access`() = runTest {
        val mockInfo = mockk<CustomerInfo> {
            every { entitlements["pro_access"]?.isActive } returns true
        }
        coEvery { billing.purchase(any(), any()) } returns mockInfo

        viewModel.purchase(mockActivity, mockPackage)

        assertTrue(viewModel.state.value is PaywallState.Success)
    }

    @Test
    fun `purchase failure shows error`() = runTest {
        coEvery { billing.purchase(any(), any()) } throws PurchasesException(
            PurchasesError(PurchasesErrorCode.StoreProblemError)
        )

        viewModel.purchase(mockActivity, mockPackage)

        assertTrue(viewModel.state.value is PaywallState.Error)
    }
}
```

CI/CD

The Test Store works without a device or Google Play account, which makes it suitable for automated test runs in CI. Because the Test Store API key is distinct from your production key, you can safely expose it to your CI environment without risking live purchases. Store it as a repository secret and inject it at build time via a `BuildConfigField`. Your unit tests then run against the mock billing layer, while instrumented tests can drive the Test Store dialog programmatically.

```
// build.gradle.kts
val testKey = System.getenv("RC_TEST_STORE_KEY") ?: "test_placeholder"
buildConfigField("String", "RC_API_KEY", "\"$testKey\"")
```


53

```
# .github/workflows/test.yml
- name: Run tests
  env:
    RC_TEST_STORE_KEY: ${{ secrets.RC_TEST_STORE_KEY }}
  run: ./gradlew testDebugUnitTest
```

When to Use Google Play Sandbox Instead

The Test Store covers most development and CI testing. Use the Google Play Sandbox when you specifically need to test:

SCENARIO	USE
Purchase success / failure / cancel	Test Store
Unit and integration tests	Test Store
CI/CD automated tests	Test Store
Subscription renewal cycles	Google Play Sandbox
Pending purchase (parental approval)	Google Play Sandbox
Actual payment flow end-to-end	Google Play Sandbox

Debug Dashboard

After any purchase (Test Store or Sandbox), inspect the result in the RevenueCat dashboard under **Customers → [user ID]**: full purchase history, active entitlements, and raw `CustomerInfo` JSON. The **Events** tab shows every webhook sent. This is the fastest way to confirm a test purchase was recorded correctly.

Pre-Ship Checklist

- ☐ `Purchases.logLevel = LogLevel.DEBUG` removed or gated on `BuildConfig.DEBUG`
- ☐ Release build type uses production `goog_` API key, not `test_` key
- ☐ `test_` key is not committed to version control (use env variable or local properties)
- ☐ Happy path, error path, and cancellation tested via Test Store dialog
- ☐ At least one end-to-end flow verified via Google Play Sandbox
- ☐ Webhook endpoint receives events for sandbox purchases

Chapter 17: Catalog Management

Managing your product catalog programmatically from scratch means calling the `monetization.subscriptions` and `monetization.onetimeproducts` REST API endpoints directly, handling latency for catalog changes, and writing tooling to manage product configurations at scale.

With RevenueCat, you do not call the catalog management APIs directly. Product catalog changes happen in two places: the Google Play Console (for the underlying products) and the RevenueCat dashboard (for how those products are organized and presented).

Product Changes in RevenueCat

When you add a new product in the Play Console, import it into RevenueCat:

1. Go to **Products** in the RevenueCat dashboard
2. Click **Import** to sync from the Play Console
3. Attach the new product to an entitlement
4. Add the product to an Offering as a Package

No code changes are required in your app. Your app calls `awaitOfferings()` and fetches whatever Offering is current. When you update the Offering in the dashboard, users see the new configuration on their next app launch without an app update.

Offering Targeting

RevenueCat supports serving different Offerings to different users through **Targeting**. You can show a discounted Offering to users on a specific OS version, country, or custom attribute.

```
// Access a placement-specific offering
val offering = offerings.getCurrentOfferingForPlacement("paywall_upsell")
    ?: offerings.current // fallback
```

Configure placements and targeting rules in the RevenueCat dashboard. No code changes are required when you update targeting rules.

Programmatic Access via RevenueCat REST API

If you need to manage products programmatically from your backend (for example, to automate catalog updates), the RevenueCat REST API has endpoints for managing products, entitlements, and offerings. This is an alternative to the Google Play catalog management API, you call RevenueCat rather than Google directly.

Google Play Console remains the source of truth for the underlying products. RevenueCat is the layer that controls which products are surfaced to users and when.

Chapter 18: Alternative Billing Programs

Supporting Google's alternative billing programs from scratch, including User Choice Billing, Alternative Billing Only, External Offers, and External Payment Links, means implementing specific Play Billing Library APIs, handling regional eligibility requirements, and building specialized server-side flows for each program.

RevenueCat has limited support for alternative billing programs as of SDK 9.x. This chapter describes the current state.

Web Billing (RevenueCat Web)

RevenueCat's own web billing product allows you to sell subscriptions outside the Play Store via a web checkout flow. This is separate from Google's alternative billing programs. The RevenueCat SDK supports web purchases through `Offering.webCheckoutURL` and `Package.webCheckoutURL`:

```
val pkg = offerings.current?.monthly
pkg?.webCheckoutURL?.let { url ->
    // Open RevenueCat's web checkout for this package
    startActivity(Intent(Intent.ACTION_VIEW, Uri.parse(url.toString())))
}
```

After a web purchase completes, the user can redeem it in-app through deep link handling. This is RevenueCat's proprietary alternative to Play Store billing, not Google's alternative billing programs.

Google's Alternative Billing Programs

For Google's alternative billing programs (User Choice Billing, Alternative Billing Only, External Offers, External Payment Links), RevenueCat does not currently expose dedicated SDK APIs. If your app participates in these programs, you would need to implement the Google Play Billing Library APIs for those flows directly alongside RevenueCat.

Using raw `BillingClient` APIs alongside the RevenueCat SDK requires care, the SDK owns the `BillingClient` instance. RevenueCat recommends setting `purchasesAreCompletedBy(PurchasesAreCompletedBy.MY_APP)` if you are managing some purchase flows yourself, so the SDK does not interfere with acknowledgement.

Check with RevenueCat Documentation

Alternative billing support evolves with each SDK version. Before implementing an alternative billing flow, check the current RevenueCat documentation and changelog for the most up-to-date guidance. What is unsupported in SDK 9.x may be supported in a later release.

- **RevenueCat Docs**: https://www.revenuecat.com/docs/ covers the full SDK reference, dashboard configuration, and platform-specific guides.
- **RevenueCat Codelabs**: https://revenuecat.github.io/ provides step-by-step implementation guides for common integration scenarios.

Appendix A: RevenueCat API Quick Reference

SDK Initialization

```
// Minimum setup
Purchases.configure(
    PurchasesConfiguration.Builder(context, "api_key").build()
)

// Full options
Purchases.configure(
    PurchasesConfiguration.Builder(context, "api_key")
        .appUserID("user_id")                    // null for anonymous
        .showInAppMessagesAutomatically(true)    // default: true
        .purchasesAreCompletedBy(PurchasesAreCompletedBy.REVENUECAT) // default
        .entitlementVerificationMode(EntitlementVerificationMode.INFORMATIONAL)
        .diagnosticsEnabled(false)               // default: false
        .pendingTransactionsForPrepaidPlansEnabled(false) // default: false
        .build()
)

Purchases.logLevel = LogLevel.DEBUG // before configure()
```

Core Operations (Coroutine)

57

```
// Fetch offerings
val offerings: Offerings = Purchases.sharedInstance.awaitOfferings()

// Fetch specific products
val products: List<StoreProduct> = Purchases.sharedInstance.awaitGetProducts(
    listOf("product_id"),
    type = ProductType.INAPP // or SUBS, or null for all
)

// Purchase
val result: PurchaseResult = Purchases.sharedInstance.awaitPurchase(
    PurchaseParams.Builder(activity, packageOrStoreProductOrSubscriptionOption).build()
)

// Purchase upgrade/downgrade
val result = Purchases.sharedInstance.awaitPurchase(
    PurchaseParams.Builder(activity, newPackage)
        .oldProductId("old_product_id")
        .googleReplacementMode(GoogleReplacementMode.WITH_TIME_PRORATION)
        .build()
)

// Get customer info
val customerInfo: CustomerInfo = Purchases.sharedInstance.awaitCustomerInfo()
val fresh: CustomerInfo = Purchases.sharedInstance.awaitCustomerInfo(
    fetchPolicy = CacheFetchPolicy.FETCH_CURRENT
)

// Restore
val customerInfo: CustomerInfo = Purchases.sharedInstance.awaitRestore()

// Log in / log out
val loginResult: LogInResult = Purchases.sharedInstance.awaitLogIn("user_id")
// loginResult.customerInfo, loginResult.created (Boolean)
val customerInfo: CustomerInfo = Purchases.sharedInstance.awaitLogOut()
```

Offerings Structure

```
val offerings: Offerings
offerings.current          // current Offering
offerings.all              // Map<String, Offering>
offerings["my_offering"]   // by identifier
offerings.getCurrentOfferingForPlacement("placement_id") // targeting

val offering: Offering
offering.identifier
offering.monthly           // Package? (shortcut)
offering.annual            // Package?
offering.weekly            // Package?
offering.availablePackages // List<Package>
offering.metadata          // Map<String, Any>

val pkg: Package
pkg.identifier             // "$rc_monthly", "$rc_annual", custom
pkg.packageType            // PackageType enum
pkg.product                // StoreProduct
pkg.webCheckoutURL         // URL? (RevenueCat web billing)

val product: StoreProduct
product.productId
product.title
product.description
product.price.formatted    // "$4.99"
product.price.amountMicros
product.price.currencyCode
product.period             // Period? (null for INAPP); use period.iso8601 for "P1M", "P1Y", etc.
product.subscriptionOptions // List<SubscriptionOption>? (null for INAPP)
product.defaultOption       // SubscriptionOption? best available offer
```

Entitlements

59

```
val customerInfo: CustomerInfo
customerInfo.entitlements.active         // Map<String, EntitlementInfo> (active only)
customerInfo.entitlements.all            // Map<String, EntitlementInfo> (all)
customerInfo.entitlements["pro_access"]  // EntitlementInfo?
customerInfo.activeSubscriptions         // Set<String> of "productId:basePlanId"
customerInfo.nonSubscriptionTransactions // List<Transaction>
customerInfo.managementURL               // Uri? to Play Store management

val entitlement: EntitlementInfo
entitlement.isActive                     // Boolean, the main access gate
entitlement.willRenew                    // false if canceled
entitlement.expirationDate               // Date? (null for lifetime)
entitlement.periodType                   // NORMAL, TRIAL, INTRO, PREPAID
entitlement.billingIssueDetectedAt       // Date? (non-null = grace/hold)
entitlement.unsubscribeDetectedAt        // Date? (non-null = canceled)
entitlement.store                        // PLAY_STORE, APP_STORE, etc.
entitlement.productIdentifier            // subscription product ID
entitlement.productPlanIdentifier        // base plan ID (Google only)
entitlement.verification                 // VerificationResult
```

Error Handling

```
// Purchase errors
try {
    Purchases.sharedInstance.awaitPurchase(params)
} catch (e: PurchasesTransactionException) {
    e.userCancelled     // Boolean
    e.error.code        // PurchasesErrorCode
    e.error.message     // description string
}

// Other errors
try {
    Purchases.sharedInstance.awaitOfferings()
} catch (e: PurchasesException) {
    e.error.code        // PurchasesErrorCode
}

// Key error codes
PurchasesErrorCode.PurchaseCancelledError
PurchasesErrorCode.ProductAlreadyPurchasedError
PurchasesErrorCode.PaymentPendingError
PurchasesErrorCode.NetworkError
PurchasesErrorCode.StoreProblemError
PurchasesErrorCode.IneligibleError
PurchasesErrorCode.ConfigurationError
```

Listeners

```
// CustomerInfo updates
Purchases.sharedInstance.updatedCustomerInfoListener =
    UpdatedCustomerInfoListener { customerInfo -> }

// Show in-app messages manually
Purchases.sharedInstance.showInAppMessagesIfNeeded(activity)
```

Appendix B: What RevenueCat Replaces vs. What Remains Your Responsibility

A quick reference comparing which billing concerns RevenueCat handles and which still require your code or infrastructure.

Client-Side

CONCERN	RAW PBL	REVENUECAT
BillingClient setup and configuration	You write	Handled internally
Connection lifecycle and reconnection	You write	Handled internally
PurchasesUpdatedListener	You write	Replaced by purchase callbacks
queryProductDetailsAsync()	You write	Replaced by awaitOfferings() / awaitGetProducts()
launchBillingFlow()	You write	Called internally by awaitPurchase()
Acknowledgement after purchase	You write	Handled automatically
Consumption of consumables	You write	Handled automatically (mark product as consumable in dashboard)
queryPurchasesAsync() on launch	You write	Handled internally on connection
Retry logic for transient errors	You write	Handled internally by SDK
BillingResponseCode handling	You write	Abstracted to PurchasesErrorCode
In-app payment recovery messages	You write	Automatic (showInAppMessagesAutomatically = true)
Subscription option selection for offers	You write	defaultOption selected automatically

Server-Side

CONCERN	RAW PBL	REVENUECAT
Google Play Developer API integration	You build	RevenueCat backend
Service account credential management	You manage	Configured once in RC dashboard
Receipt verification on every purchase	You build	Automatic
Purchase token validation and deduplication	You build	RevenueCat backend
linkedPurchaseToken chain traversal	You build	RevenueCat backend
RTDN processing and dispatch	You build	RevenueCat processes, sends webhooks
Cloud Pub/Sub setup	You set up	Not needed
Subscription state machine on backend	You build	CustomerInfo computed by RC
Entitlement computation across 7 states	You build	isActive computed by RC
Grace period / account hold tracking	You build	billingIssueDetectedAt
Cancellation with access-until-expiry logic	You build	isActive + unsubscribeDetectedAt
Price cohort tracking	You build	RC backend handles
Product-to-entitlement mapping	You build	Configured in RC dashboard

What Remains Your Responsibility

CONCERN	NOTES
User authentication system	You bring your own; pass user ID to RC
Your own database of users	RC is not your primary user database
Premium content server-side	Verify via RC REST API or webhooks
Webhook receiver endpoint	You build; RC sends, you receive
Play Console product creation	Still done in Play Console
Subscription deferral	Direct Google Play API call
Subscription revocation	Direct Google Play API call
Alternative billing programs	Limited RC support; may need raw PBL
App UI (paywalls, onboarding)	You build (RC Paywalls UI optional)
Push notifications for payment issues	You build on top of webhook events

63