



Subscriptions with RevenueCat

A practical guide to setting up subscriptions and one-time purchases with RevenueCat and StoreKit in a React Native app.

WHO THIS GUIDE IS FOR

This picks up from *How to Launch a React Native App on the App Store*, which covers subscriptions only at a high level. This is the detailed version — how to wire up App Store Connect, RevenueCat, and local StoreKit testing so you can build and test a paywall before your products are approved or you've set up sandbox accounts.

Beginners — every account and file involved, in order.

Technical readers — use this as a reference checklist and troubleshooting guide.

SECTION	COVERS
01 · Why RevenueCat	What it does, and why it beats building receipt validation yourself
02 · Foundations	Why local StoreKit testing first, and the four places product IDs must match
03 · App Store Connect	App record, subscription group, products
04 · RevenueCat	Project, entitlement, offering, SDK key — and the prompts that build the purchase flow
05–06 · Xcode & testing	.storekit file, device signing, run scheme, verifying purchases
07–08 · Checklist & fixes	Full setup checklist, common failure modes



01 · WHAT IS REVENUECAT & WHY CHOOSE IT

Skip the receipts

RevenueCat is a subscription and purchase management layer that sits between StoreKit/Google Play Billing and your app — it turns raw platform receipts into a simple "does this user have access?" check.

WHAT IT ACTUALLY DOES

When a purchase happens, RevenueCat validates the receipt with Apple (or Google) on your behalf, tracks renewal/expiration/refund state server-side, and exposes a single `entitlement` your app can check — instead of your app parsing raw StoreKit transactions and re-implementing Apple's validation and edge cases yourself.

Why it's the default choice here

- 1 Receipt validation is genuinely hard to get right.** Server-side verification, grace periods, refunds, billing retries, and renewal edge cases are easy to get subtly wrong if you build this yourself — and subtly wrong here means users losing access they paid for, or keeping access they didn't.
- 2 One API across platforms.** The same entitlement model works across the App Store, Google Play, and web/Stripe purchases, so adding Android later doesn't mean rebuilding your purchase logic.
- 3 Entitlements decouple access from products.** Your app checks "does this user have `premium`," not "did this user buy `yearly_premium_v1`" — so changing pricing or adding a new product later doesn't require touching your access-control code.
- 4 Dashboards and webhooks come for free.** MRR, churn, and trial-conversion metrics are visible without wiring up your own analytics, and webhooks let your backend stay in sync with subscription state without polling.



Test locally **first**

You can't fully test purchases against Apple's real infrastructure until your App Store Connect products are approved. A local StoreKit configuration file lets you simulate the entire purchase flow without waiting.

HOW THE FLOW WORKS

RevenueCat stays wired into your app exactly as it will be in production — but purchases are simulated locally via a `.storekit` file during development. Use this flow when you want purchases to load immediately, your App Store Connect products aren't approved yet, or you want to avoid sandbox account setup while you're still building the paywall UI/UX.

Important constraints

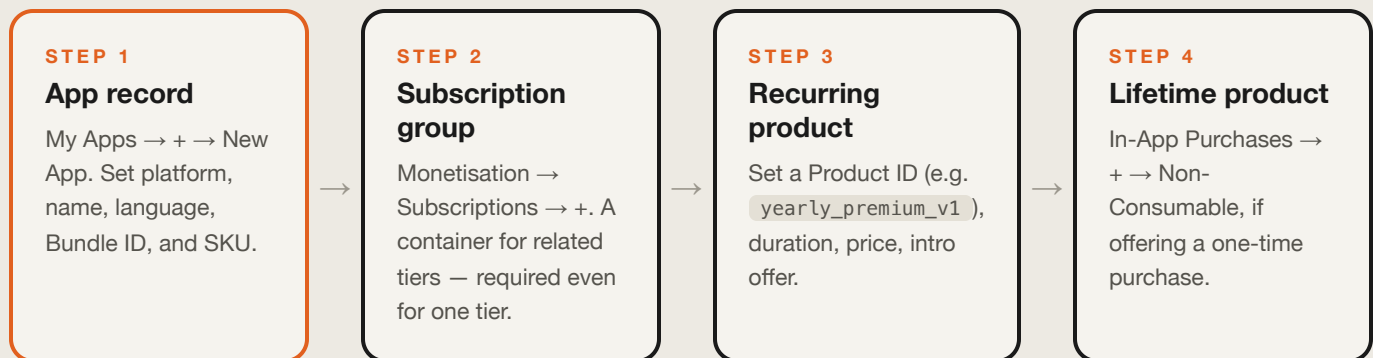
- a Run from Xcode, not Expo Go.** StoreKit testing requires a native build.
- b Use a native development build on a physical iPhone.** Simulators can work for some StoreKit testing, but a physical device is the most reliable.
- c Product IDs must be identical across four places** — your app code, RevenueCat, App Store Connect, and the `.storekit` file. A single typo in any one of these is the most common source of "offerings are empty" bugs.



03 · APP STORE CONNECT SETUP

Real products, **first**

Even though you'll be testing locally, you still need real product records in App Store Connect so RevenueCat and your eventual production setup stay aligned from day one.

**WHY THE `_v1` SUFFIX**

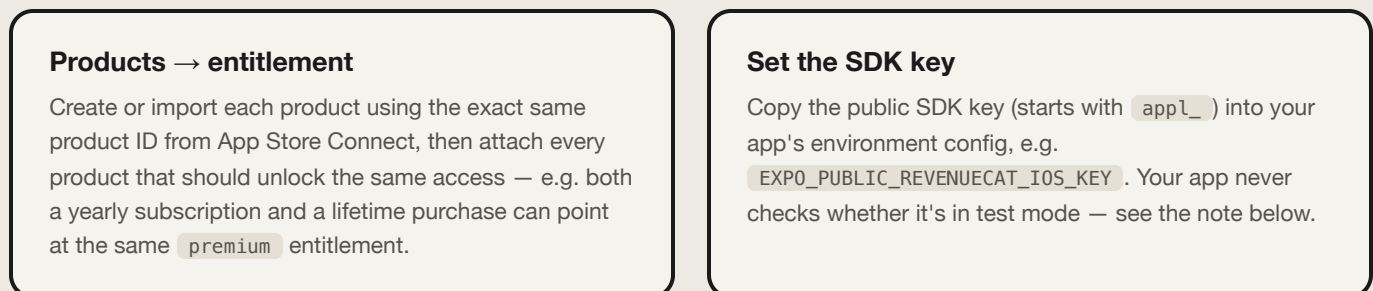
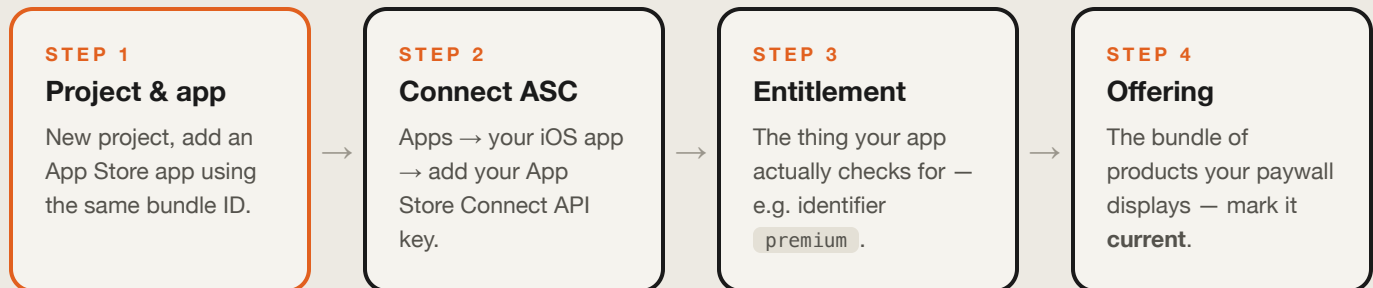
Product IDs can never be reused once retired, and you may need a `_v2` later — worth adopting the versioned naming convention even on your first product. Once created, keep the product ID constants in your app code (typically a dedicated purchases service file) as the single source of truth — this is the first of the four places these IDs must match exactly.



04 · REVENUECAT SETUP

Entitlements, not products

RevenueCat sits between StoreKit and your app, handling receipt validation and entitlement state so you don't have to build that logic yourself.



"NO STOREKIT-SPECIFIC BRANCH IS NEEDED" — IN PLAIN TERMS

You do **not** write anything like `if ( __DEV__ ) { useFakePurchases() }`. Your app calls the same `Purchases.configure()` and the same `purchasePackage()` in every environment. What changes is *underneath* it: when a `.storekit` file is attached to your Xcode Run scheme, Apple's own StoreKit framework answers those calls from that local file instead of from the App Store. The swap happens in the Xcode scheme, not in your code — which is why the code you test is literally the code you ship.

THE SILENT FAILURE MODE

If your app fetches `offerings.current` (a common pattern), marking the offering current is what determines what shows up on the payroll — an offering that isn't marked current will silently not appear.



04 · REVENUECAT SETUP

The integration, end to **end**

Three pieces: install the SDK, configure it once at launch, and call one function from your paywall button. You don't write any of it by hand — you specify it.

COPY-PASTE · TERMINAL · INSTALL

```
npx expo install react-native-purchases
# native module – rebuild after installing:
npx expo run:ios --device
```

COPY-PASTE · PROMPT · CONFIGURE ONCE, AT APP LAUNCH

Configure react-native-purchases in this Expo app.

- Call `Purchases.configure()` once, in the root layout (`app/_layout.tsx`), inside a `useEffect` that runs on mount – NOT lazily when the paywall screen opens
- Read the key from `process.env.EXPO_PUBLIC_REVENUECAT_IOS_KEY`
- Enable the SDK's `DEBUG` log level in development only, never in a release build
- Do not add any `__DEV__` branch or mock-purchase path: the same code must run in every environment

Show me the file you changed and where the call sits relative to the rest of the provider tree.

The "not lazily" line is the one that matters. A paywall that configures on mount is the most common cause of "offerings are empty on first open, fine on second" — so it goes in the prompt, not in a comment you'd have to notice afterwards.

WHAT THIS BUYS YOU

That single `configure` call is the entire setup. From here your app never parses a receipt, checks an expiry date, or talks to Apple directly — it asks RevenueCat one question, *is this entitlement active*, and RevenueCat has already handled renewals, refunds, grace periods and billing retries on the server side.



04 · REVENUECAT SETUP

The paywall **button**

Load the offering, sell the package the user tapped, then check the entitlement. Three calls — and one of the three error cases is not an error. Put all of that in the prompt.

COPY-PASTE · PROMPT · THE PAYWALL BUTTON

Implement the purchase flow for my paywall screen using react-native-purchases.

1. Load: `getOfferings()`, read `offerings.current`, render its `availablePackages`. Use each package's localised price string from the product — never format or hardcode a price myself.
2. Buy: `purchasePackage()` for the package the user tapped. On success, check `customerInfo.entitlements.active['premium']` before unlocking anything.
3. Gate: a reusable way to ask "is this user premium?" anywhere else in the app, based on the `ENTITLEMENT` — never on a product ID.

Required behaviours:

- If the error has `userCancelled` set, return silently. That is the user tapping Cancel on Apple's sheet, not a failure, and must not show an error message.
- Show a real message for genuine failures, and log the rest.
- Include a Restore Purchases action calling `restorePurchases()`.
- Handle the loading state: the paywall must not render an empty package list as if there were nothing for sale.

Plan first, then show me the diff.

- 1 **A cancelled purchase is not a failure.** `e.userCancelled` means the user tapped Cancel on Apple's sheet — return silently. Showing an error there is the most common paywall UX bug.
- 2 **Check the entitlement, never the product.** `entitlements.active['premium']` stays correct when you add products or change pricing; `boughtYearlyV1` does not.
- 3 **Add a Restore Purchases button.** `Purchases.restorePurchases()` — Apple requires one for non-consumables and subscriptions, and its absence is a rejection.



05 · LOCAL STOREKIT TESTING IN XCODE

Mirror it **exactly**

The .storekit file is a local mirror of your App Store Connect products — every ID and type has to line up, or testing fails silently.

STEP 1**Create the file**

File → New → File → StoreKit Configuration File. Name it (e.g. `LocalTesting.storekit`) and save into your iOS project. Syncing with App Store Connect is optional.

**STEP 2****Add products**

Match ID, duration, price, and intro offer exactly for each Auto-Renewable Subscription and Non-Consumable product.

**STEP 3****Attach to scheme**

Product → Scheme → Edit Scheme → Run → Options → StoreKit Configuration → select your file.

THE SECOND AND THIRD MATCH POINTS

The `.storekit` file is the second and third places product IDs must match exactly (after your app code). A mismatch means the product simply won't be found during testing — often with an unhelpful or silent failure. Also double-check the active scheme is your actual app target and bundle ID; it's easy to have multiple schemes/targets in a React Native/Expo project and attach the file to the wrong one.



Run it, then **verify**

Building the native app is a one-time step; after that, all your purchase testing happens by pressing Run in Xcode with the .storekit file attached.

BEFORE THIS COMMAND WILL WORK · FIRST-TIMERS

Installing onto a physical iPhone requires code signing, and this is where first attempts stall. You do **not** need the paid \$99 Apple Developer Program for this step — a free Apple ID works, with the caveat that the build expires after 7 days and has to be reinstalled.

1. Open Xcode → Settings → Accounts, and add your Apple ID.
2. Open the `ios/` workspace, select the app target → Signing & Capabilities → tick *Automatically manage signing* and pick your Team. (Bundle ID conflicts here are common — if signing fails, change the bundle ID to something unique to you.)
3. Plug the iPhone in, unlock it, and tap *Trust This Computer*. Enable Developer Mode on the device: Settings → Privacy & Security → Developer Mode, then restart.
4. First launch of the installed app will refuse to open until you approve the certificate: Settings → General → VPN & Device Management → your Apple ID → Trust.

COPY-PASTE · FIRST NATIVE BUILD

```
npx expo run:ios --device
```

Once the native project exists, do your actual StoreKit purchase testing by pressing **Run** in Xcode with the `.storekit` file attached, rather than re-running the Expo CLI command each time. Open the app on a physical iPhone, walk through to your paywall, trigger each purchase path, and confirm in the local StoreKit sheet — no sandbox Apple ID required.

Verify the result

App behavior

The purchase sheet appears, the purchase completes, and the app's post-purchase flow (onboarding continuing, paywall dismissing) behaves correctly.

Backend state

RevenueCat shows the transaction and updated entitlement for your test user. If you write purchase state to your own backend, confirm that write actually happens — it's easy to check "did the purchase succeed" without checking "did my database update."



07 · QUICK CHECKLIST

Every step, one **page**

The full setup condensed to a single reference checklist — useful to re-run whenever you add a new product.

APP STORE CONNECT

- ☐ Create app record with correct bundle ID
- ☐ Create subscription group
- ☐ Create recurring subscription product
- ☐ Create lifetime/one-time product (if applicable)
- ☐ Confirm product IDs match your app code

REVENUECAT

- ☐ Create iOS app with matching bundle ID
- ☐ Add App Store Connect API key
- ☐ Create entitlement
- ☐ Add all products
- ☐ Attach all products to the entitlement
- ☐ Create and mark an offering as current
- ☐ Set the platform SDK key in your app config

Xcode

- ☐ Create a .storekit configuration file
- ☐ Add each product with matching IDs
- ☐ Attach .storekit file to the Run scheme

Testing

- ☐ Build/run native app from Xcode (not Expo Go)
- ☐ Trigger each purchase path
- ☐ Verify RevenueCat entitlement updates
- ☐ Verify your own backend/database updates



08 · TROUBLESHOOTING

When it **breaks**

Nearly every failure in this flow traces back to a mismatched ID, an unmarked offering, or the wrong build target. Work through these in order.

Offerings are empty

Check your `default` offering is marked current. Check every product ID matches your app code exactly. Check your platform SDK key env var is set and being read.

Product not found

Check the `.storekit` file's product IDs match exactly. Check product types (auto-renewable vs. non-consumable). Check the active Xcode scheme uses the correct `.storekit` file.

Purchase UI doesn't appear

Confirm you're running from Xcode, not Expo Go. Confirm StoreKit Configuration is set on the active scheme. Confirm the build includes native purchase modules.

RevenueCat doesn't update

Confirm the purchases SDK initializes on app launch, not lazily. Confirm the SDK key matches the correct project/app. Confirm every product is attached to the entitlement you're checking.

YOUR OWN BACKEND DOESN'T UPDATE

Confirm your post-purchase handler is actually being called and awaited. Confirm the triggering screen/flow correctly writes the returned subscription data afterward, rather than assuming RevenueCat's state alone is sufficient. Add logging around the purchase success callback if hard to reproduce — intermittent failures here are often a race condition between the purchase resolving and the write happening.



SUMMARY

Setup flow at a glance

The whole sequence, condensed to one reference table.

STAGE	WHAT HAPPENS
App Store Connect	Create app record, subscription group, and products (recurring + one-time)
RevenueCat	Connect App Store Connect, create entitlement, add products, create current offering, set SDK key
Xcode	Create .storekit file with matching products, attach to Run scheme
Testing	Build native app, run from Xcode, test each purchase path, verify RevenueCat + backend state

THE ONE RULE THAT MATTERS MOST

Every product ID — app code, RevenueCat, App Store Connect, and the `.storekit` file — has to match exactly, in all four places, every time you add a product. Nearly every bug in this guide's troubleshooting section traces back to one of these four falling out of sync.

Where to go next

GUIDE	PICKS UP WITH
Launching on the App Store	Submitting the app your paywall now lives in — and the paid-feature disclosure rules that apply to it
The AI Coding Stack	The rest of the build stack around this integration
Idea to App Store	The combined guide — all seven stages end to end