# 31 · Payments

Flutter Practice Notes

## Contents

# 31 · Payments

## Introduction

This module covers accepting money in a Flutter app: the **security/PCI ground rules** (never touch raw card data; the server is the source of truth), **payment gateways** for physical goods/services (Stripe, Razorpay — the `PaymentIntent` /order flow), **in-app purchases** for digital goods (Apple App Store / Google Play, consumables/non-consumables/subscriptions), **wallets** (Google Pay / Apple Pay), and **server-side verification** (webhooks, receipt validation) that ties it together securely.

## Why this module exists

Payments are where bugs cost real money and mistakes create fraud/compliance liability. The rules differ sharply by *what* you sell: **digital goods must use the platform IAP billing** (Apple/Google take a cut and mandate it), while **physical goods/services must use a gateway** (and are *banned* from IAP). Getting the flow, the store rules, and the security model right is essential and non-obvious.

## Module map

| # | File | Topic | Level |
|---|---|---|---|
| 1 | 01_payments_overview_and_security.md | Payment models, PCI/security, IAP-vs-gateway rule, server-as-source-of-truth | 🔴 |
| 2 | 02_payment_gateways.md | Stripe/Razorpay, `PaymentIntent` /order flow, cards & wallets for goods/services | 🔴 |
| 3 | 03_in_app_purchases.md | `in_app_purchase`, consumables/non-consumables/subscriptions, store setup | 🔴 |
| 4 | 04_wallets_google_apple_pay.md | Google Pay / Apple Pay integration & when they apply | 🟡 |
| 5 | 05_payments_integration.md | Capstone: server verification, webhooks, receipts, reconciliation | 🔴 |

> **Cross-references:** Networking (calling your backend): Module 16. Auth (identifying the buyer): Module 17. Secure storage (tokens, never cards): Module 15. Error handling: Module 38. Firebase (Cloud Functions for webhooks): Module 18. Deployment/store review: Module 51.

## Prerequisites

16 Networking, 17 Authentication, 15 Local Storage (secure storage), a backend you control (or Cloud Functions — Module 18).

## What you'll be able to do after this module

- Choose the correct payment path (IAP vs gateway) for what you're selling and pass store review.
- Integrate a gateway (Stripe/Razorpay) with the client-secret/order flow — no raw card data in your app.
- Implement in-app purchases (consumables, non-consumables, subscriptions) with restore.
- Add Google Pay / Apple Pay where appropriate.
- Verify payments server-side (webhooks, receipt validation) as the source of truth.

## Capstone

**Payment slice:** A checkout that (a) buys a **digital** item via IAP with server receipt verification and (b) pays for a **physical** order via a gateway using a server-created `PaymentIntent`, with the backend confirming via webhook and unlocking entitlement/fulfilling the order — client never trusts its own success callback.

## Summary

Payments = pick the right rail (IAP for digital, gateway for physical), never handle raw card data, and make the **server the source of truth** via webhooks/receipt verification. This module covers the flows, the platform store rules, wallets, and the security/verification backbone.

# Payments Overview & Security (The Ground Rules)

> Two hard rules govern all mobile payments: **(1)** your app must **never see or store raw card data** — a PCI-compliant SDK tokenizes it and your server works with tokens/intents; **(2)** the **platform decides the rail** — *digital goods/services* consumed in-app **must** use Apple/Google in-app purchase (they take ~15–30% and mandate it), while *physical goods/real-world services* **must** use an external gateway and are **banned** from IAP. And always: **the server, verified server-side, is the source of truth** — never the client's success callback.

## Introduction

Before any SDK, you must internalize the payment model: what you're selling determines the rail, PCI dictates you never touch cards, and trust must live on the server. This file establishes those rules, the actors (client, gateway/store, your backend), and the canonical secure flow every later file builds on.

## Why this concept exists

Payments involve money, fraud, and regulation. PCI-DSS exists so card data isn't leaked; Apple/Google billing rules exist to enforce their cut and consumer protections; server-side verification exists because a mobile client is untrusted (it can be tampered, spoofed, or fail after charging). These constraints are non-negotiable and shape the entire architecture.

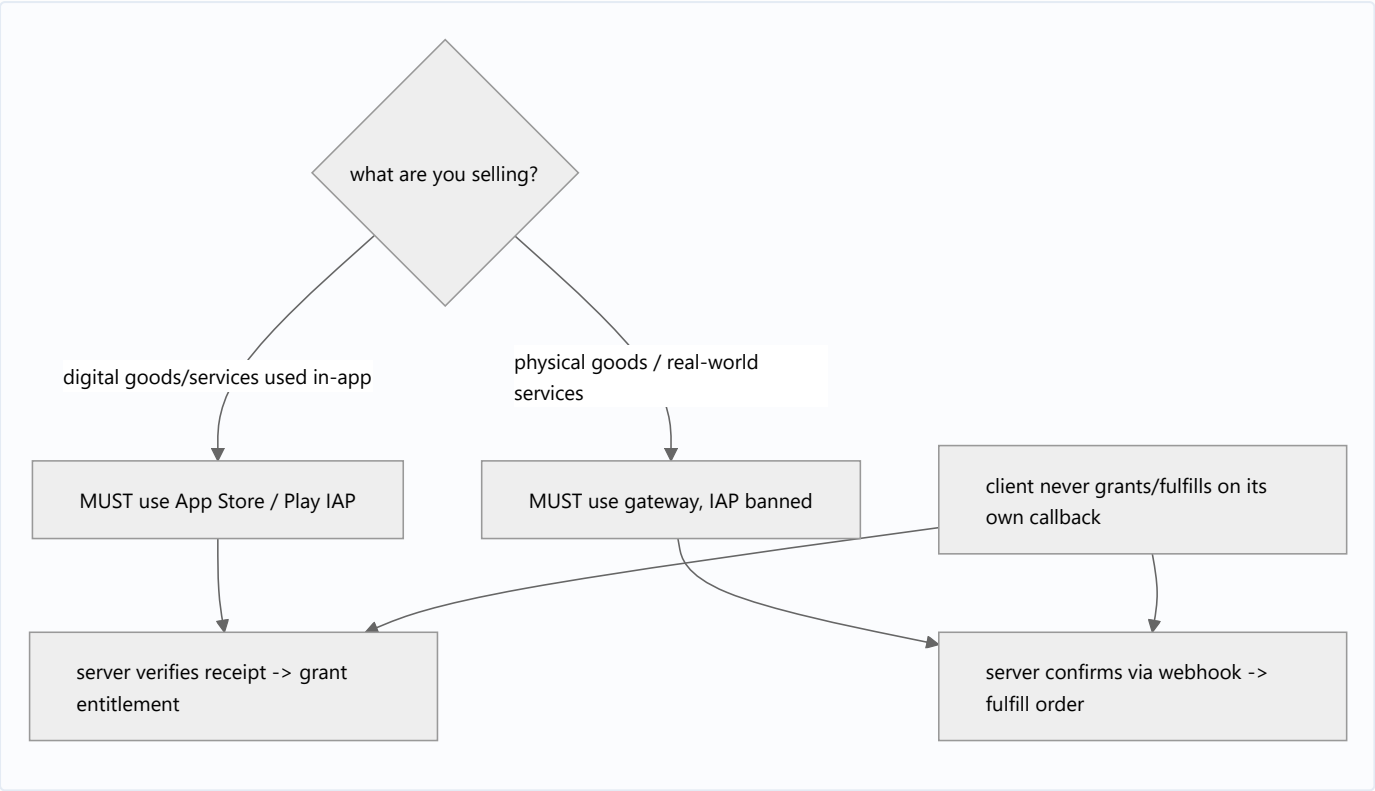
## Real-world analogy

You're a shop that **never handles cash directly**: a **bonded cashier** (PCI SDK) takes the customer's card behind glass and hands you a **receipt token**. For **downloadable goods** the mall (Apple/Google) *requires* you use **its checkout** and takes a cut; for **furniture you deliver**, the mall *forbids* its checkout and you use your **own card terminal** (gateway). And you only ship once your **accountant confirms the bank actually settled** (server verification) — not because the customer *said* they paid.

## Problem Statement

Decide, for each thing you sell (a coin pack, a pro subscription, a physical order, a service booking), which rail is legal/required, ensure no card data touches your app, and design a flow where entitlement/fulfillment happens only after server verification. You'll classify items and design the trust boundary.

## Internal Working




- **Rail selection (critical, App Store/Play policy)**:
  - **Digital content/services consumed in the app** (coins, pro features, subscriptions, unlockables) → **must** use **in-app purchase** (03_in_app_purchases.md). Using a gateway for these = rejection/ban.
  - **Physical goods, real-world services** (food delivery, ride, e-commerce, ticket to a physical event) → **must** use a **gateway** (Stripe/Razorpay/wallets) — IAP is **not allowed** for these (02_payment_gateways.md).
  - Grey areas (reader apps, external-link entitlements, regional rules) evolve — check current policy before shipping.
- **PCI / card data**: your app **never** collects raw PAN/CVV into your own variables/storage. The gateway SDK (Stripe/Razorpay) presents its own secure input or tokenizes; you only ever handle **tokens/intents/nonces**. This keeps you out of PCI scope. **Never log or store card data.**
- **Server as source of truth**: the client is untrusted. The **backend creates the charge intent/order**, and **verifies completion server-side** (webhook / receipt validation) before granting entitlement or fulfilling. A client "payment succeeded" callback is a *hint*, not proof (it can be faked or the app can die mid-flow).
- **Actors**: **client** (collects intent, launches SDK UI, shows result), **gateway/store** (charges, tokenizes, sends webhooks/receipts), **your backend** (creates intents/orders, holds secret keys, verifies, grants/fulfills, reconciles).
- **Secret keys** live only on the server; the client holds only publishable/public keys.

- **Idempotency**: use idempotency keys/order ids so retries don't double-charge or double-grant.

## Memory Representation

Not a data-structure topic. The security-relevant point: card data should have **no representation** in your app's memory/logs/storage — only opaque tokens.

## Compiler Behavior

Not applicable.

## Runtime Behavior

The charge happens via the SDK/store; the authoritative state transition (entitlement/fulfillment) happens on your server after verification — often slightly *after* the client sees "success" (webhook latency), so design for eventual confirmation.

## Flutter Engine Behavior

Payment SDKs often present native UI (platform views / native sheets — 26 · platform channels) for PCI-compliant card entry / wallet sheets.

## Dart VM Behavior

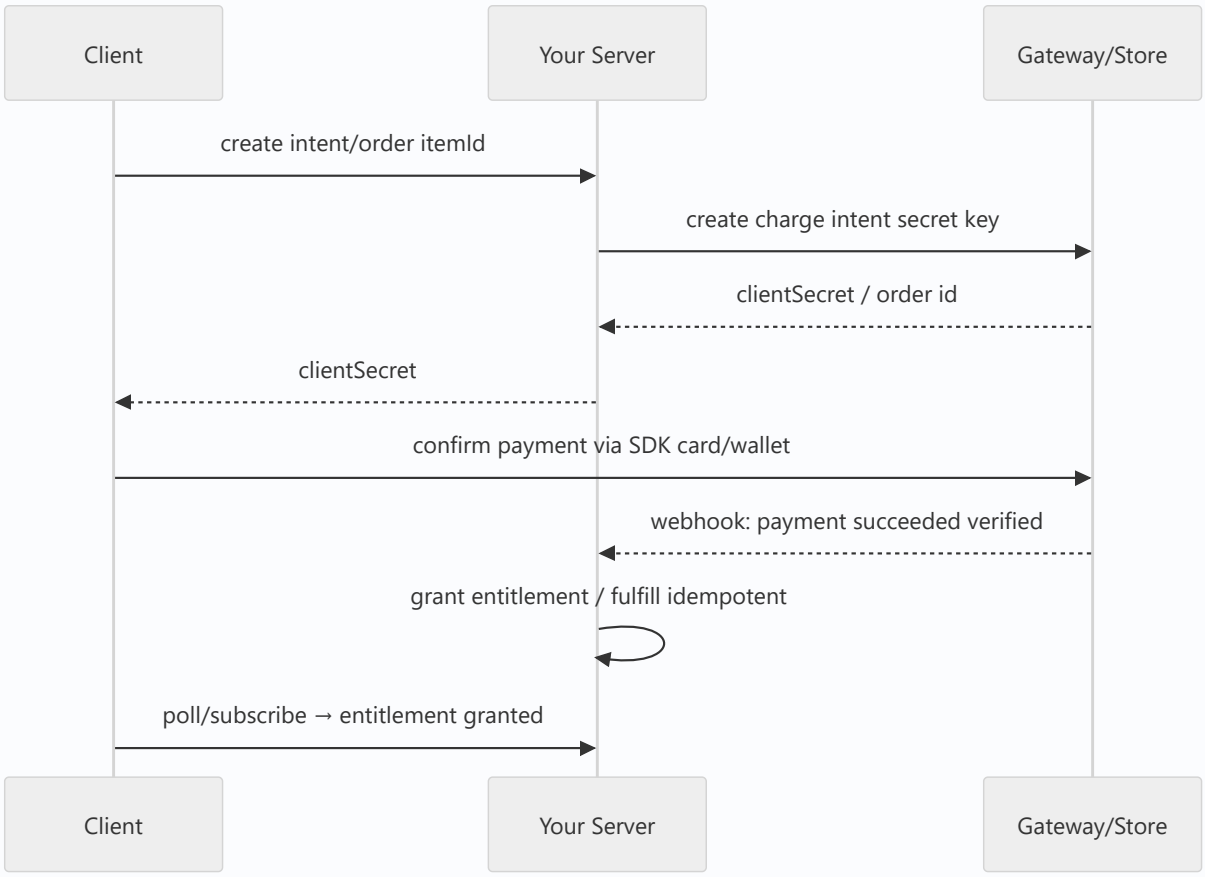
Not applicable.

## Examples

```
// The SAFE shape: client asks server to create an intent, launches SDK, then
// waits for SERVER confirmation — it never grants entitlement itself.
Future<void> checkout(CartItem item) async {
  // 1) Server creates the intent/order (holds secret key, sets amount)
  final intent = await api.createPaymentIntent(itemId: item.id); // returns clientSecret
  // 2) SDK collects card/wallet securely (no raw card data in our code)
  final result = await gatewaySdk.confirm(intent.clientSecret);
  // 3) DO NOT unlock here on `result.success` alone.
  //    Server verifies via webhook; client polls/subscribes for entitlement.
  await api.awaitEntitlement(item.id); // truth comes from the server
}
```

```
Rail decision cheat-sheet:
  Coins / pro unlock / subscription (digital, in-app) -> IAP (App Store / Play)   [mandatory]
  Physical product / food / ride / booking            -> Gateway (Stripe/etc.)     [IAP banned]
  Never: collect raw card numbers into your own code / storage / logs.
  Always: server creates intent, server verifies completion, then grant/fulfill.
```

## Diagrams



## Common Mistakes

| Mistake | Why it's bad | Fix |
|---|---|---|
| Using a gateway for digital goods | App Store/Play rejection/ban | Use IAP for digital content |
| Using IAP for physical goods | Also against policy/rejection | Use a gateway for physical/services |
| Collecting raw card data in-app | PCI liability, breach risk | Use SDK tokenization; never store cards |
| Trusting the client success callback | Fraud / lost revenue / double grants | Verify server-side (webhook/receipt) |
| Secret keys in the app | Key theft, unlimited charges | Secret keys server-only; client uses publishable |
| No idempotency | Double charges/grants on retry | Idempotency keys / order ids |
| Logging card/token PII | Compliance/breach | Never log sensitive payment data |

## Best Practices

- **Classify each item** and pick the mandated rail (digital→IAP, physical→gateway); verify current store policy before shipping.
- **Never handle raw card data** — rely on the PCI-compliant SDK; keep **secret keys server-side**, publishable keys client-side.
- Make the **server create intents/orders and verify completion** (webhook/receipt) before granting/fulfilling; treat client callbacks as hints.
- Use **idempotency keys**; never log/store sensitive payment data; design for **eventual** (webhook-latency) confirmation.

## Performance

Irrelevant vs correctness/security here. The only latency concern is webhook confirmation delay — show a pending state and confirm from the server rather than blocking or optimistically unlocking.

## Advantages / Disadvantages

- **+** (Following the rules) compliant, secure, fraud-resistant, store-approvable, out of PCI scope.
- **−** Requires a backend, webhook/verification complexity, platform cut for IAP, eventual-consistency UX, strict policy constraints.

## Interview Questions

1. 🟢 **When must you use IAP vs a payment gateway?** — Digital goods/services consumed in-app **must** use App Store/Play IAP; physical goods/real-world services **must** use a gateway (IAP is banned for those).

2. 🟢 **Why must your app never handle raw card data?** — PCI-DSS: touching card data puts you in scope and creates breach/compliance liability; SDKs tokenize instead.
3. 🟡 **Why is the client success callback not enough?** — The client is untrusted (tamperable, spoofable, can die mid-flow); only server-side verification (webhook/receipt) proves payment.
4. 🟡 **Where do secret vs publishable keys live?** — Secret keys only on your server; the client holds only publishable/public keys.
5. 🟡 **What is idempotency and why does it matter?** — Using a stable key/order id so retries don't double-charge or double-grant.
6. 🔴 **Design the secure end-to-end flow.** — Server creates intent/order → client confirms via SDK → gateway/store notifies server (webhook/receipt) → server verifies + idempotently grants/fulfills → client reflects server truth.
7. 🔴 **How do you handle webhook latency in UX?** — Show a pending state and confirm entitlement from the server (poll/subscribe); don't optimistically unlock.

## Senior Engineer Tips

- Write down the rail decision per SKU *before* coding — a gateway-for-digital mistake gets the whole app rejected.
- Treat the client as a hostile environment: no secret keys, no card data, no trusting its own "success" — the server grants everything.
- Build idempotent grant/fulfill from day one; payment retries and duplicate webhooks are normal, not edge cases.

## Architect Perspective

Payments are a trust-boundary and compliance architecture problem, not an SDK-wiring problem. The invariants — right rail per item, no card data client-side, server-as-source-of-truth with idempotent verification — dictate that you need a backend and a clean client/server contract. Every later file (gateways, IAP, wallets, verification) is an implementation of these rules (02_payment_gateways.md, 03_in_app_purchases.md, 05_payments_integration.md).

## Summary

- Rail is mandated by *what* you sell: digital→IAP, physical→gateway (each banned from the other).
- Never touch raw card data (PCI); secret keys server-only.
- Server creates intents/orders and verifies completion (webhook/receipt) before granting/fulfilling; idempotent; client callbacks are hints.

## Revision Notes

- Digital in-app → IAP (mandatory, ~15–30% cut); physical/services → gateway (IAP banned); verify current policy.
- No raw card data in app (PCI); SDK tokenizes; secret keys server-only, publishable client-side.
- Server = source of truth: creates intent/order, verifies via webhook/receipt, idempotent grant/fulfill; design for webhook latency.

## Practice Questions

1. Classify five items (coins, pro subscription, T-shirt, ride, e-book) by rail.
2. Why can't the client be trusted to grant entitlement?
3. What keeps your app out of PCI scope?

## Coding Questions

1. Sketch the client `checkout()` that defers entitlement to the server.
2. Design an idempotent server grant given possibly-duplicate webhooks.
3. Write the rail-decision function for a mixed catalog.

## Mini Project

**Payment architecture design (Flutter + backend):** For a catalog mixing digital (coins, pro subscription) and physical (merch order) items, document the rail per SKU, design the client/server contract (server creates intent/order, verifies via webhook/receipt, idempotent grant/fulfill), and implement a client `checkout()` that never grants entitlement itself (waits for server truth). Acceptance: correct rail per item; no card data/secret keys client-side; server-verification flow documented + client defers to it; idempotency addressed; pending-state UX for webhook latency.

# Payment Gateways (Stripe, Razorpay — Physical Goods & Services)

> For physical goods/real-world services, integrate a gateway with the **server-driven intent/order flow**: your backend (holding the **secret key**) creates a `PaymentIntent` (Stripe) or `Order` (Razorpay) with the amount; the client uses the **publishable key** + SDK ( `flutter_stripe` / `razorpay_flutter` ) to collect card/wallet securely and confirm using the returned **client secret/order id**; the gateway then confirms to your server via **webhook**, which fulfills the order — the client never sets the amount or trusts its own result.

## Introduction

Gateways charge cards/wallets for non-digital purchases. Stripe and Razorpay share the same secure shape: server creates the payment object with the amount, client confirms with a secret it can't tamper into a different amount, server verifies via webhook. This file covers that flow, the two SDKs, and the security boundary.

## Why this concept exists

Charging cards requires PCI compliance and fraud control. Gateways provide tokenizing SDKs (so your app never sees card data) and a server API where the **amount is set server-side** (so a client can't pay ₹1 for a ₹1000 order). The webhook exists because the client is untrusted — the gateway tells your server directly.

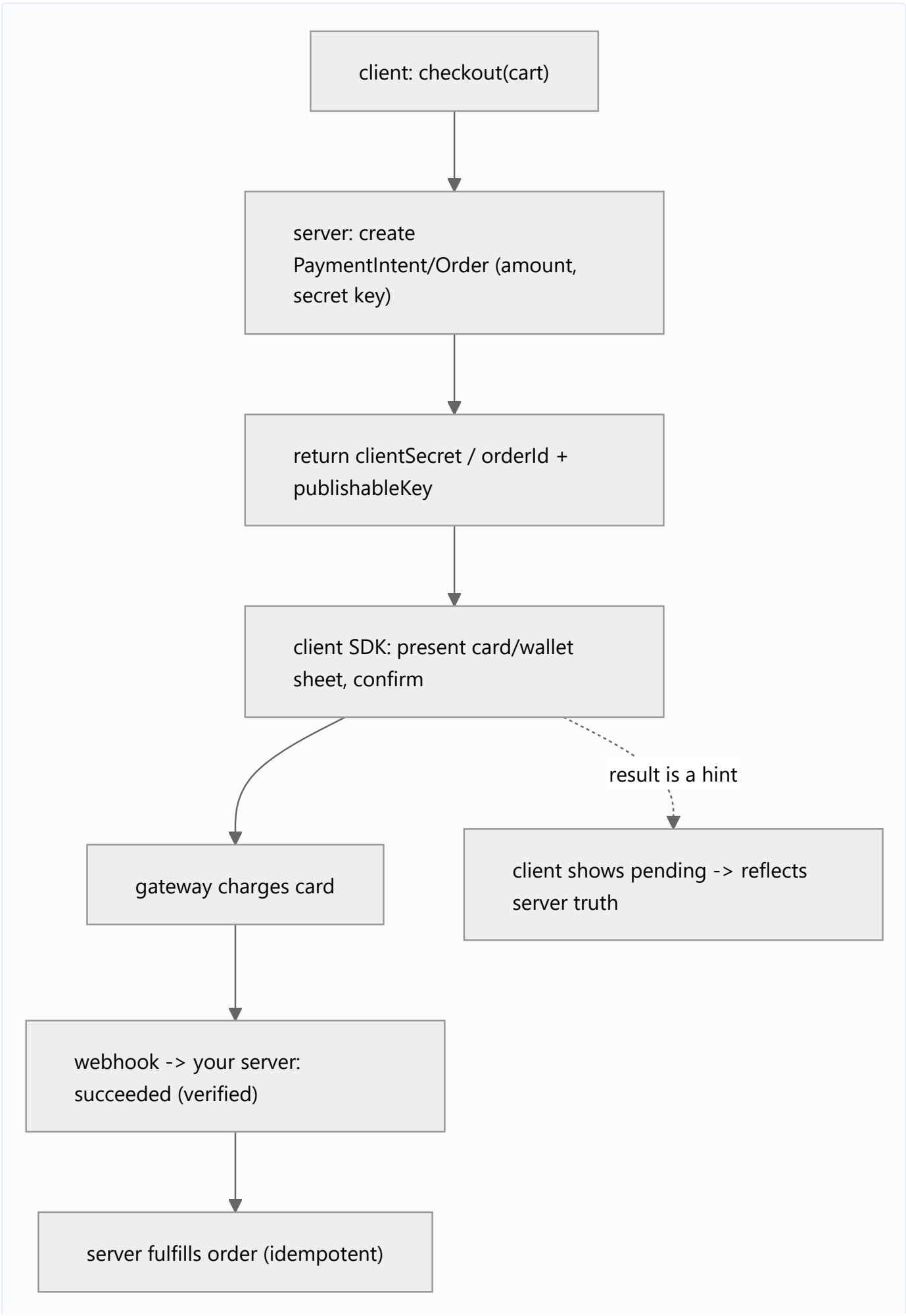
## Real-world analogy

The gateway is a **card terminal you rent**: your **back office** (server) rings up the exact total and prints a **one-time payment slip** (client secret / order id); the customer taps their card on the terminal (SDK) against *that slip*; the **bank calls your back office** (webhook) to confirm settlement before you hand over the goods. The customer can't change the total written on the slip.

## Problem Statement

Charge for a physical order (e.g., ₹899 cart) via Stripe (global) or Razorpay (India): server creates the intent/order with the amount, the app confirms with the card/wallet sheet, and the server fulfills on the verified webhook. You'll wire the SDK + the server-driven flow.

## Internal Working


client: checkout(cart)
server: create PaymentIntent/Order (amount, secret key)
return clientSecret / orderId + publishableKey
client SDK: present card/wallet sheet, confirm
result is a hint
gateway charges card
client shows pending -> reflects server truth
webhook -> your server: succeeded (verified)
server fulfills order (idempotent)

- **Stripe (`flutter_stripe`)**:
  1. Server: `PaymentIntent.create(amount, currency, ...)` with the **secret key** → returns a **client secret**.
  2. Client: init with **publishable key**; `Stripe.instance.initPaymentSheet(paymentIntentClientSecret: ...)` then `presentPaymentSheet()` (Stripe's PCI-compliant sheet handles card entry / Apple/Google Pay).
  3. Server: handle the `payment_intent.succeeded` **webhook** → fulfill.
- **Razorpay (`razorpay_flutter`)**:
  1. Server: create an **Order** (`orders` API) with the **key secret** → returns an **order id**.
  2. Client: `razorpay.open({key, amount, order_id, ...})`; handle `EVENT_PAYMENT_SUCCESS` / `ERROR` / `EXTERNAL_WALLET`.
  3. Server: verify the **payment signature** (HMAC of order_id|payment_id with key secret) and/or webhook → fulfill.
- **Security invariants**: **amount is set server-side** (client never dictates price); **secret key server-only**; **card data only in the SDK sheet**; **fulfillment on verified webhook/signature**, not the client callback.
- **Wallets**: both route Apple Pay / Google Pay for physical purchases through the same sheet (04_wallets_google_apple_pay.md).
- **Refunds/disputes**: handled server-side via the gateway API.
- **Repository**: expose `Future<PaymentResult> pay(cartId)`; UI/domain never see keys or card data.

## Memory Representation

Client holds only a transient client secret/order id + publishable key. No card data. The authoritative record (paid/fulfilled) lives in your DB, set on webhook.

## Compiler Behavior

Not applicable.

## Runtime Behavior

The SDK presents a native sheet; on confirm, the gateway charges and later calls your webhook (may lag the client result by seconds). Fulfillment is triggered by the webhook, so success UX is "payment received, finalizing."

## Flutter Engine Behavior

Payment sheets are native UI (Stripe/Razorpay) presented over Flutter via platform integration (26 · platform channels).

## Dart VM Behavior

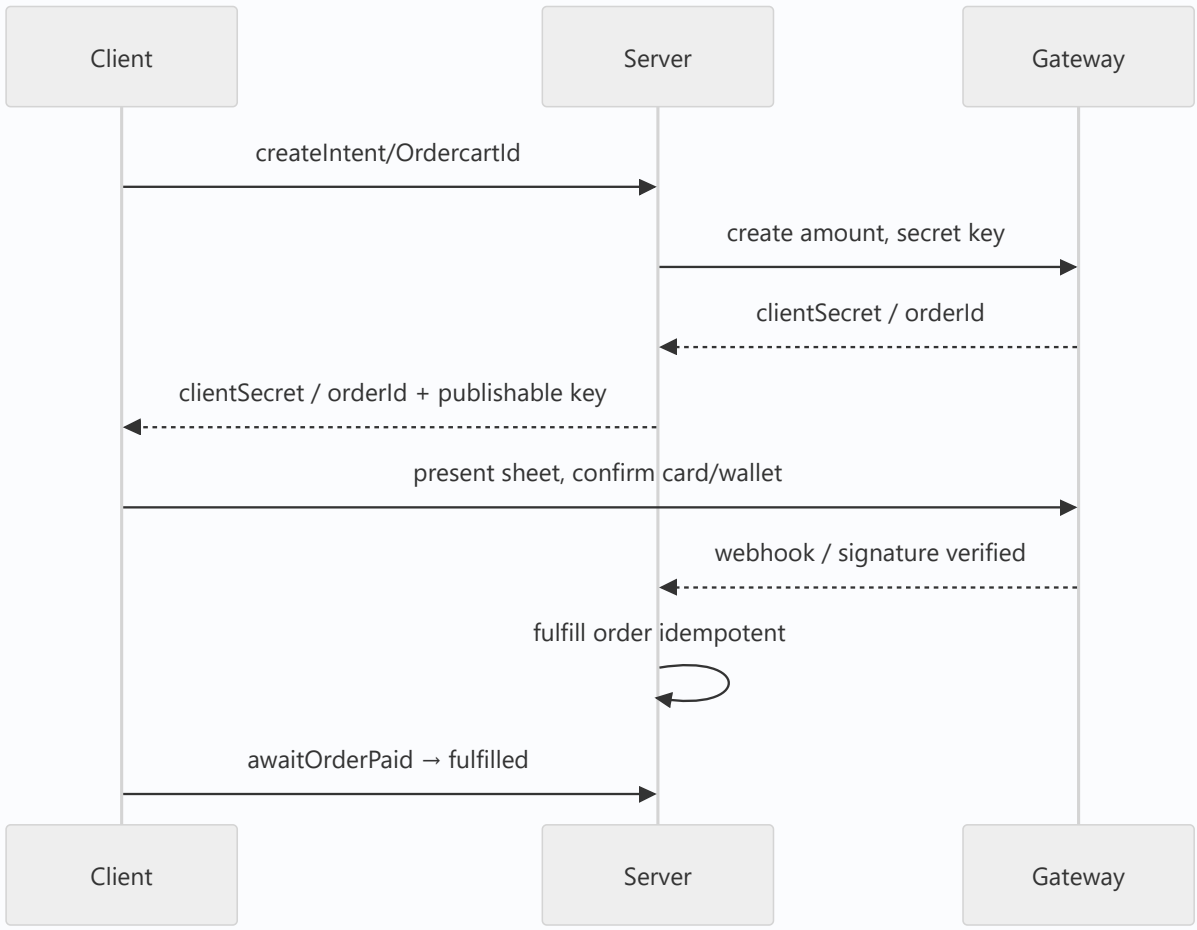
Not applicable.

## Examples

```
// Stripe (flutter_stripe) — server sets the amount; client confirms with the secret
import 'package:flutter_stripe/flutter_stripe.dart';

class StripeGateway {
  Future<void> pay(String cartId) async {
    // 1) Server creates the PaymentIntent (amount decided server-side)
    final clientSecret = await api.createPaymentIntent(cartId: cartId);
    // 2) PCI-compliant sheet handles card entry / wallets (no card data in our code)
    await Stripe.instance.initPaymentSheet(
      paymentSheetParameters: SetupPaymentSheetParameters(
        paymentIntentClientSecret: clientSecret,
        merchantDisplayName: 'My Shop',
      ),
    );
    await Stripe.instance.presentPaymentSheet();
    // 3) Do NOT fulfill here — server does it on payment_intent.succeeded webhook.
    await api.awaitOrderPaid(cartId); // reflect server truth
  }
}
// main(): Stripe.publishableKey = '<publishable_key>'; (secret key stays on server)
```

```
// Razorpay (razorpay_flutter) — server creates the Order; server verifies signature
import 'package:razorpay_flutter/razorpay_flutter.dart';

class RazorpayGateway {
  final _razorpay = Razorpay();
  void init() {
    _razorpay.on(Razorpay.EVENT_PAYMENT_SUCCESS, (r) async {
      // Send payment_id + order_id + signature to server for HMAC verification.
      await api.verifyRazorpay(r.orderId, r.paymentId, r.signature); // server verifies + fulfills
    });
    _razorpay.on(Razorpay.EVENT_PAYMENT_ERROR, (e) {/* show failure */});
  }
  Future<void> pay(String cartId) async {
    final order = await api.createRazorpayOrder(cartId: cartId); // {orderId, amount, key}
    _razorpay.open({
      'key': order.publishableKey, 'order_id': order.orderId,
      'amount': order.amount, 'name': 'My Shop',
    });
  }
}
```

## Diagrams




## Common Mistakes

| Mistake | Why | Fix |
|---|---|---|
| Client sets/sends the amount | Price tampering | Amount set server-side in the intent/order |
| Secret key in the app | Key theft, arbitrary charges | Secret key server-only; client uses publishable |
| Fulfilling on client success | Fraud/no-verify | Fulfill on webhook / verified signature |
| Skipping signature verification (Razorpay) | Spoofed success | Verify HMAC server-side |
| Custom card fields | PCI scope/breach | Use the SDK's sheet/tokenization |
| Non-idempotent fulfillment | Double-ship on retry/dup webhook | Idempotent by order id |
| Using a gateway for digital goods | Store rejection | Gateways are for physical/services only |

## Best Practices

- **Server sets the amount** and creates the intent/order with the **secret key**; client uses only the **publishable key** + returned secret/order id.
- Collect card/wallet **only via the SDK sheet** (PCI); **fulfill on the verified webhook/signature**, idempotently — never on the client callback.
- **Verify Razorpay signature** / handle Stripe webhooks server-side; support refunds/disputes via the gateway API.
- Wrap in a **repository** returning a domain `PaymentResult`; keep keys/card data out of UI/domain; gateways are for **physical/services only** (01_payments_overview_and_security.md).

## Performance

Not perf-sensitive; the UX concern is webhook latency — show "finalizing" and confirm from the server. SDK sheets load quickly.

## Advantages / Disadvantages

- **+** Charge cards/wallets for physical goods/services, PCI-handled by SDK, server-controlled amounts, refunds/disputes, global (Stripe) / India (Razorpay) coverage.
- **−** Requires a backend + webhooks, secret-key/verification discipline, eventual-confirmation UX, gateway fees, not usable for digital goods.

## Interview Questions

1. 🟢 **Who sets the payment amount and why?** — The **server** (in the PaymentIntent/Order) so the client can't tamper the price.
2. 🟢 **What key does the client use vs the server?** — Client: publishable/public key; server: secret key (never shipped).
3. 🟡 **Walk through the Stripe payment-sheet flow.** — Server creates a `PaymentIntent` → returns client secret → client `initPaymentSheet` / `presentPaymentSheet` → server fulfills on `payment_intent.succeeded` webhook.
4. 🟡 **How is Razorpay success verified?** — Server verifies the HMAC signature (order_id|payment_id with key secret) and/or a webhook before fulfilling.
5. 🟡 **Why not fulfill on the client success event?** — It's an untrusted hint; only server verification (webhook/signature) proves payment.
6. 🔴 **How do you prevent double fulfillment?** — Idempotent fulfillment keyed by order/payment id (webhooks can duplicate; clients retry).

7. 🔴 **Why can't you use a gateway for coins/subscriptions consumed in-app?** — Store policy mandates IAP for digital goods; gateways are for physical/real-world only.

## Senior Engineer Tips

- Never let the amount originate on the client — server-created intents are the single most important anti-fraud measure.
- Make fulfillment idempotent and webhook-driven; treat the client result purely as "show pending."
- Verify Razorpay signatures server-side every time; a client-reported success without server verification is an open fraud door.

## Architect Perspective

Gateways implement the "server-as-source-of-truth" rule for physical commerce: amount and verification on the server, card data in the SDK, fulfillment on webhook. Behind a repository returning domain results, the app stays PCI-safe and fraud-resistant while the backend owns money movement — the same trust boundary as auth/networking, applied to payments (01_payments_overview_and_security.md, 05_payments_integration.md, Module 16).

## Summary

- Server creates `PaymentIntent` / `Order` with the amount (secret key); client confirms via SDK sheet with the client secret/order id (publishable key).
- Card data only in the SDK; fulfill on verified webhook/signature, idempotently; never trust the client callback or let it set the amount.
- Stripe (global) / Razorpay (India); wrap behind a repository; gateways = physical/services only.

## Revision Notes

- Stripe: server `PaymentIntent.create` → clientSecret → `initPaymentSheet` / `presentPaymentSheet` → `payment_intent.succeeded` webhook → fulfill.
- Razorpay: server `orders` API → orderId → `razorpay.open` → verify HMAC signature server-side → fulfill.
- Secret key server-only; amount server-set; card data in SDK sheet only; idempotent webhook fulfillment; physical/services only.

## Practice Questions

1. Why must the amount be created server-side?

2. What does the client hold vs the server (keys/secrets)?
3. How is a Razorpay payment verified before fulfillment?

## Coding Questions

1. Implement a Stripe `pay(cartId)` using the server-created client secret + payment sheet.
2. Implement Razorpay `pay` + server-side signature verification.
3. Design an idempotent, webhook-driven fulfillment handler.

## Mini Project

**Physical-order checkout (Flutter + backend):** Implement a `StripeGateway` (or `RazorpayGateway`) where the server creates the intent/order with the amount, the client confirms via the SDK sheet, and the server fulfills on the verified webhook/signature (idempotent). UI shows a pending → confirmed state from server truth. Acceptance: amount set server-side; secret key never in app; card data only in SDK; fulfillment webhook/signature-verified + idempotent; behind a repository; runs end-to-end (test mode).

# In-App Purchases (Digital Goods — Consumables, Non-Consumables, Subscriptions)

> Digital goods consumed in-app **must** use the platform stores via the `in_app_purchase` plugin: define products in App Store Connect / Play Console, query them, launch the store purchase, and **listen to the purchase stream**; **verify the receipt/token server-side** before granting entitlement, then **complete** the purchase — and for **consumables** consume them, for **non-consumables/subscriptions** support **restore**.

## Introduction

IAP is the mandated rail for coins, pro unlocks, and subscriptions. Unlike a gateway, you don't create the charge — the store does — but you *must* verify the receipt on your server and manage product types and restoration. This file covers product setup, the purchase stream, the three product types, receipt verification, and restore.

## Why this concept exists

Apple/Google require their billing for digital goods (consumer protection + their cut) and provide `StoreKit` / `Play Billing`. The `in_app_purchase` plugin unifies them. Server-side receipt verification exists because the client purchase result is untrusted and receipts can be replayed/forged — entitlement must be validated with Apple/Google.

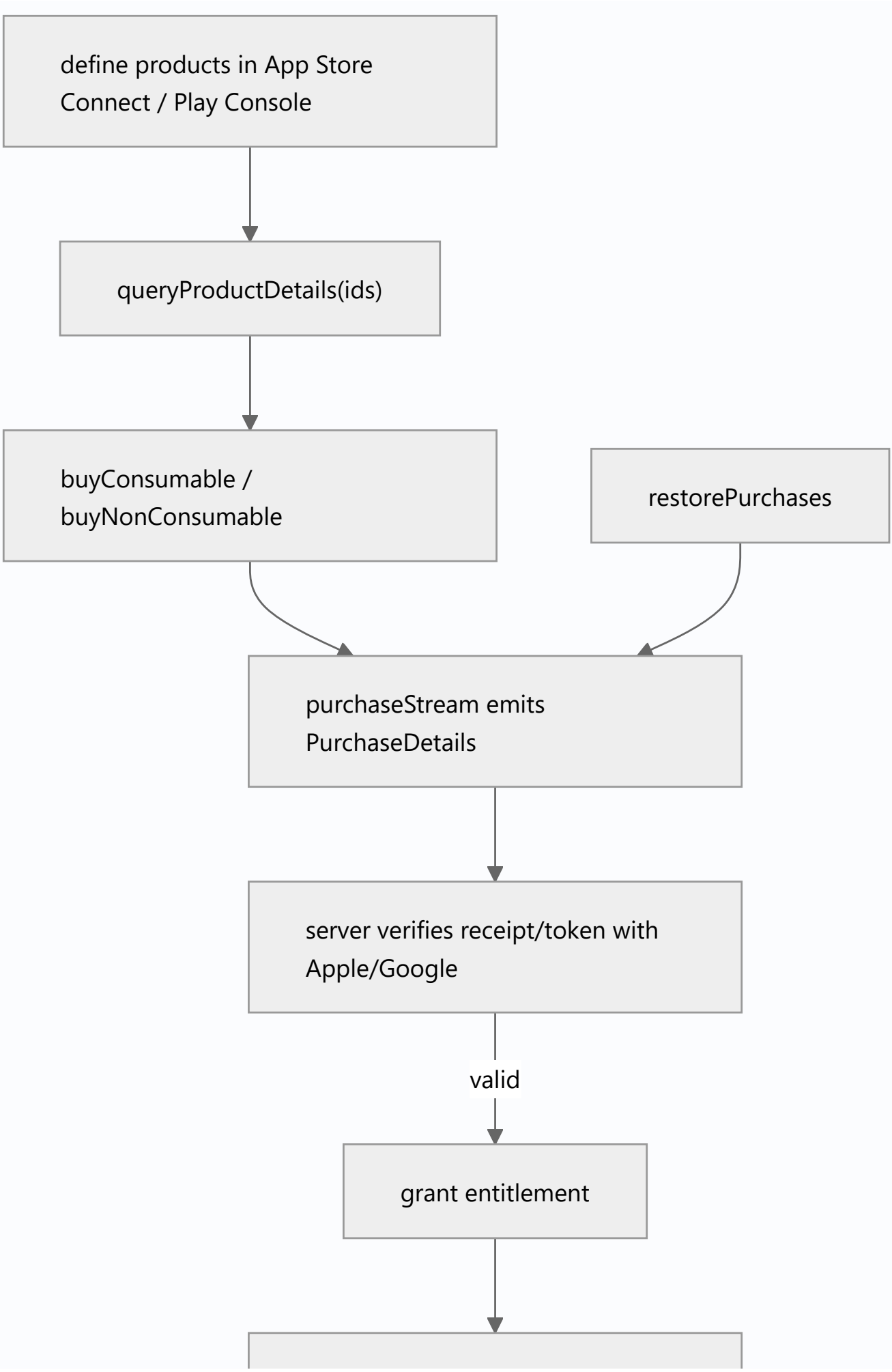
## Real-world analogy

IAP is buying **tokens at an arcade's official booth** (you can't bring outside cash to the machines). The booth prints a **receipt**; before the machine gives you a prize you take the receipt to the **manager's office** (your server) which **calls the booth to confirm it's genuine** (Apple/Google verification). **Consumables** are tokens you spend; **non-consumables** are a permanent membership card; a **subscription** is a monthly pass you can **restore** on a new machine.

## Problem Statement

Sell a **consumable** (coin pack), a **non-consumable** (remove ads forever), and a **subscription** (pro monthly) with a "restore purchases" button — verifying each receipt server-side before granting, completing purchases, and consuming consumables. You'll wire `in_app_purchase` + server verification.

## Internal Working


define products in App Store Connect / Play Console
queryProductDetails(ids)
buyConsumable / buyNonConsumable
restorePurchases
purchaseStream emits PurchaseDetails
server verifies receipt/token with Apple/Google
valid
grant entitlement

completePurchase (+ consume if consumable)

- **Product setup**: create products in **App Store Connect** and **Play Console** with matching **product ids**; types: **consumable** (coins), **non-consumable** (permanent unlock), **auto-renewable subscription** (pro). Products must be approved/active to query.
- **Query**: `InAppPurchase.instance.queryProductDetails({ids})` → `ProductDetails` (localized price/title). Show these (don't hardcode prices).
- **Purchase**: `buyConsumable(purchaseParam)` or `buyNonConsumable(purchaseParam)`. Results arrive **only** via `InAppPurchase.instance.purchaseStream` (set the listener up at app start — purchases can complete/resume out-of-band).
- **Purchase stream states**: `pending` → `purchased` / `restored` (verify + grant) or `error` / `canceled`. **Always** call `completePurchase(details)` after handling (or the store re-delivers).
- **Server verification (mandatory)**: send the receipt/purchase token to your server, which validates with **Apple** (verifyReceipt / App Store Server API) or **Google** (Play Developer API) before granting entitlement. Never grant on the client result alone.
- **Consume vs restore**: **consumables** must be **consumed** (so they can be bought again); **non-consumables/subscriptions** support `restorePurchases()` (required by Apple) to re-grant on reinstall/new device.
- **Subscriptions**: status/renewal/expiry is tracked via server-to-server notifications (App Store Server Notifications / Play RTDN) — the server is the entitlement authority over time.
- **Repository**: expose products + `buy()` + entitlement state; keep store details out of UI.

## Memory Representation

`ProductDetails` / `PurchaseDetails` are small value objects. Entitlement truth lives server-side (validated), cached locally as a derived flag — not the source of truth.

## Compiler Behavior

Not applicable.

## Runtime Behavior

Purchases may arrive asynchronously (including after app restart / interrupted flows) — hence the always-on stream + `completePurchase`. Subscriptions renew/expire over time, updated via server notifications.

## Flutter Engine Behavior

The store purchase UI is native (StoreKit / Play Billing) presented over Flutter via the plugin's platform channels (26 · platform channels).

## Dart VM Behavior

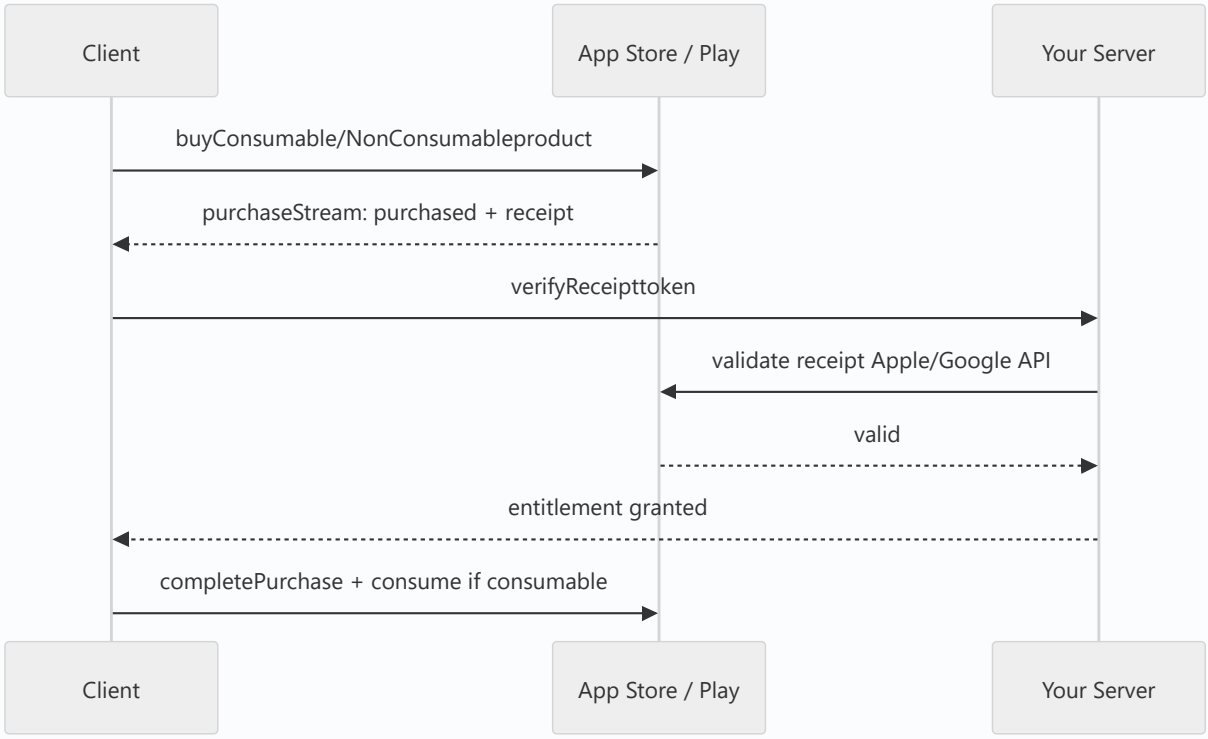
Not applicable.

## Examples

```
import 'package:in_app_purchase/in_app_purchase.dart';


class IapRepository {
  final _iap = InAppPurchase.instance;
  late final Stream<List<PurchaseDetails>> _stream = _iap.purchaseStream;


  // Set up the listener at APP START (purchases can resume out-of-band)
  void listen(void Function() onEntitled) {
    _stream.listen((purchases) async {
      for (final p in purchases) {
        if (p.status == PurchaseStatus.pending) continue;
        if (p.status == PurchaseStatus.purchased || p.status == PurchaseStatus.restored) {
          final ok = await api.verifyReceipt(              // SERVER verifies with Apple/Google
            productId: p.productID,
            token: p.verificationData.serverVerificationData,
          );
          if (ok) onEntitled();                            // grant only if server says valid
        }
        if (p.pendingCompletePurchase) {
          await _iap.completePurchase(p);                  // ALWAYS complete (else re-delivered)
        }
      }
    });
  }


  Future<List<ProductDetails>> products(Set<String> ids) async {
    final resp = await _iap.queryProductDetails(ids);
    return resp.productDetails;                            // localized prices
  }
```

```
  Future<void> buyCoins(ProductDetails coins) =>
      _iap.buyConsumable(purchaseParam: PurchaseParam(productDetails: coins));

  Future<void> buyProSubscription(ProductDetails pro) =>
      _iap.buyNonConsumable(purchaseParam: PurchaseParam(productDetails: pro)); // subs use
buyNonConsumable

  Future<void> restore() => _iap.restorePurchases();       // required for non-consumables/subs
}
```

## Diagrams



## Common Mistakes

| Mistake | Why | Fix |
|---|---|---|
| Granting on client purchase result | Forgeable/replayable | Verify receipt server-side first |
| Not calling `completePurchase` | Store re-delivers / stuck | Always complete after handling |
| Listener set up late/on one screen | Missed/resumed purchases lost | Listen at app start, app-wide |
| Not consuming consumables | Can't rebuy | Consume after grant |
| No restore button | Apple rejection; users lose access | Implement `restorePurchases()` |
| Hardcoding prices | Wrong/localized prices | Use `ProductDetails` price |
| Using IAP for physical goods | Policy violation | Gateway for physical (02_payment_gateways.md) |

## Best Practices

- **Set up the purchase-stream listener at app start** (app-wide) and **always** `completePurchase`; handle `pending` / `error` / `canceled` / `restored`.
- **Verify every receipt server-side** with Apple/Google before granting; treat the client result as a hint. Server is the entitlement authority (esp. subscriptions, via server notifications).
- **Consume consumables**; implement `restorePurchases()` for non-consumables/subscriptions (Apple requires it); show **localized** `ProductDetails` **prices**.
- Wrap in a **repository** exposing products + entitlement; use IAP **only for digital goods** (01_payments_overview_and_security.md).

## Performance

Not perf-sensitive. The concern is correctness/robustness: out-of-band and interrupted purchases must be handled by the always-on stream + completion, and entitlement must survive reinstalls via server-validated restore.

## Advantages / Disadvantages

- **+** Mandated/approved rail for digital goods, native store UX, subscriptions + restore, familiar to users.
- **−** ~15–30% store cut, server verification required, async/out-of-band purchase handling, per-store product setup, subscription lifecycle complexity.

## Interview Questions

1. 🟢 **When must you use IAP?** — For digital goods/services consumed in the app (coins, unlocks, subscriptions); it's store-mandated.

2. 🟢 **What are the three product types?** — Consumable (rebuyable, must consume), non-consumable (permanent), auto-renewable subscription.
3. 🟡 **Why verify receipts server-side?** — Client purchase results are untrusted/replayable; only Apple/Google validation proves the purchase before granting entitlement.
4. 🟡 **Why must the purchase-stream listener be app-wide and set up early?** — Purchases can complete/resume out-of-band (after restart/interruption); a late/screen-local listener misses them.
5. 🟡 **Why is `completePurchase` mandatory?** — Until completed, the store considers it undelivered and will re-deliver it.
6. 🔴 **How do you handle subscription status over time?** — The server is the authority, updated via App Store Server Notifications / Play RTDN (renewals, cancellations, expiries).
7. 🔴 **Why is restore required and for what?** — Apple requires a restore path for non-consumables/subscriptions so users regain entitlements on reinstall/new device (`restorePurchases`).

## Senior Engineer Tips

- Initialize the purchase stream in your app bootstrap, not a store screen — the #1 IAP bug is a purchase that completes while the paywall is closed and is never granted.
- Make the server the durable entitlement store (esp. subscriptions with server notifications); the local flag is just a cache.
- Test the ugly paths: interrupted purchase, reinstall + restore, refund/cancel — these are where real IAP bugs and rejections live.

## Architect Perspective

IAP is an asynchronous, server-verified entitlement system, not a one-shot buy. The robust design: app-wide purchase stream → server receipt verification → durable entitlement (with subscription server-notifications) → local cache, all behind a repository. This survives reinstalls, interruptions, and refunds, and keeps digital-goods commerce store-compliant — complementing the gateway rail for physical goods (02_payment_gateways.md, 05_payments_integration.md).

## Summary

- Digital goods → IAP: define products per store, query (localized prices), buy, handle via an **app-wide purchase stream**, **complete** every purchase.
- **Verify receipts server-side** before granting; server is entitlement authority (subscriptions via server notifications).

- Consume consumables; implement restore for non-consumables/subscriptions; behind a repository; digital only.

## Revision Notes

- `in_app_purchase`: products in App Store Connect/Play Console (matching ids); `queryProductDetails` → `buyConsumable` / `buyNonConsumable`.
- Results via `purchaseStream` (app-wide, early); states pending/purchased/restored/error/canceled; **always** `completePurchase`.
- Server verifies receipt/token (Apple/Google) before grant; consume consumables; `restorePurchases()` (required); subs tracked via server notifications.

## Practice Questions

1. Why must receipts be verified server-side?
2. Where and when do you set up the purchase-stream listener, and why?
3. What's the difference in handling consumables vs non-consumables/subscriptions?

## Coding Questions

1. Implement an `IapRepository` with app-wide stream handling + `completePurchase`.
2. Add server receipt verification before granting entitlement.
3. Implement consume (consumable) + `restorePurchases` (non-consumable/sub).

## Mini Project

**Store purchases (Flutter + backend):** Implement an `IapRepository` selling a consumable (coins), a non-consumable (remove ads), and a subscription (pro), with an app-wide purchase-stream listener, server-side receipt verification before granting, `completePurchase`, consumable consumption, and a "restore purchases" button. Acceptance: three product types work; listener app-wide + early; receipts verified server-side before grant; purchases completed; consumables rebuyable; restore re-grants on reinstall; behind a repository; runs (sandbox/test track).

# Wallets (Google Pay & Apple Pay)

> Google Pay and Apple Pay are **faster checkout methods for physical goods/services** — they return a **tokenized card** you pass to your gateway (Stripe/Razorpay), not a separate payment rail. They boost conversion (no typing, biometric confirm) but the **same rules apply**: server sets the amount, server verifies via webhook, and wallets are **not** a substitute for IAP on digital goods.

## Introduction

Wallets let users pay with a saved card via Face ID/fingerprint instead of typing card details. In Flutter you can use `pay` (the official Google/Apple Pay plugin) or the wallet support built into gateway SDKs (`flutter_stripe`, `razorpay_flutter`). This file covers how wallets fit the gateway flow, setup, and when they apply.

## Why this concept exists

Manual card entry kills mobile conversion. Wallets provide a one-tap, biometric-confirmed, tokenized payment that's more secure (no card shown) and faster. They exist on top of the card networks/gateways — the wallet supplies a payment token; your gateway still does the charge.

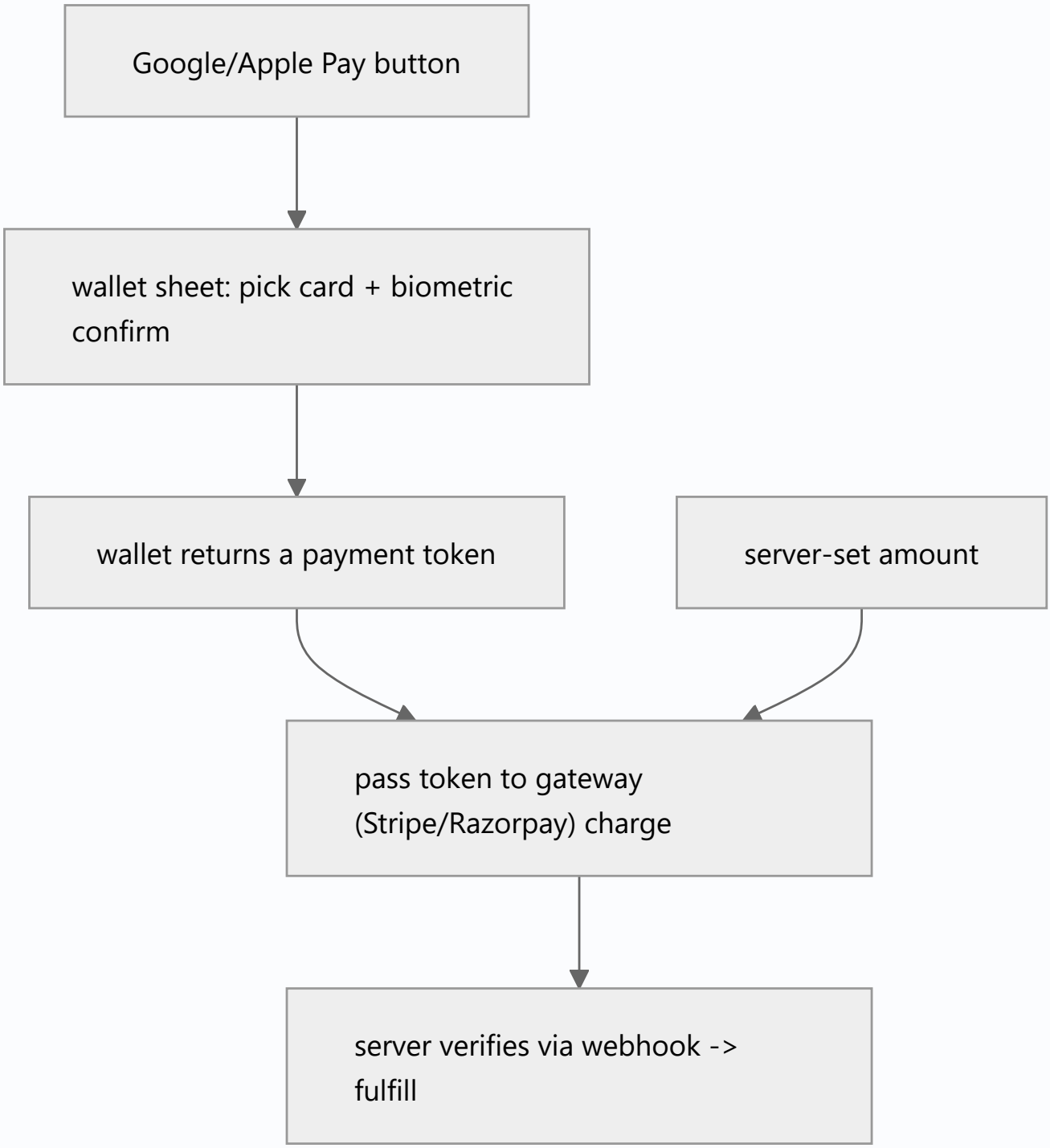
## Real-world analogy

A wallet is **tap-to-pay with your phone** instead of digging out a physical card: the terminal (gateway) still runs the charge and the bank still settles — you just authenticated with your face/thumb and the card number never appeared. The **cashier (server) still rings up the exact total** and confirms the bank settled.

## Problem Statement

Add a "Pay with Google Pay / Apple Pay" button to a physical-order checkout so users pay in one tap, feeding the wallet token into your existing gateway flow (server amount + webhook verification). You'll wire wallet buttons and route their token to the gateway.

## Internal Working




- **Two integration paths**:
  1. **Via the gateway SDK** (recommended): Stripe/Razorpay payment sheets **already offer** Apple/Google Pay when configured — the wallet token flows through the same `PaymentIntent` /order you already built (02_payment_gateways.md). Least extra work.
  2. `pay` **plugin**: renders native Google/Apple Pay buttons, returns a payment token you forward to your gateway (or processor) to charge.

- **Setup**:
  - **Google Pay**: a payment configuration JSON (gateway = your processor, e.g. Stripe), merchant info, allowed card networks; test in Google Pay test env.
  - **Apple Pay**: a **Merchant ID + Apple Pay capability/entitlement** in Xcode (28 · ios_integration), merchant certificate; only on real devices with a card in Wallet.
- **Same invariants**: **amount server-set**, **charge via gateway**, **fulfill on verified webhook** — the wallet only changes *how the card is supplied* (tokenized), not the trust model.
- **Availability**: check device/wallet availability and only show the button when supported (Apple Pay iOS-only, Google Pay Android/web); provide a card fallback.
- **Digital goods**: wallets do **not** bypass IAP — you still can't sell coins/subscriptions via Apple/Google Pay; those go through IAP (03_in_app_purchases.md).

## Memory Representation

Client handles only an opaque wallet **payment token** (not card data). Charge/fulfillment records live server-side as with any gateway payment.

## Compiler Behavior

Not applicable; Apple Pay needs the capability/entitlement at build (28 · ios_integration).

## Runtime Behavior

The wallet sheet is a native biometric flow; on confirm it returns a token instantly; the gateway charge + webhook proceed as normal (with the usual confirmation latency).

## Flutter Engine Behavior

Wallet sheets are native UI presented over Flutter (via `pay` /gateway SDK platform channels — 26).

## Dart VM Behavior

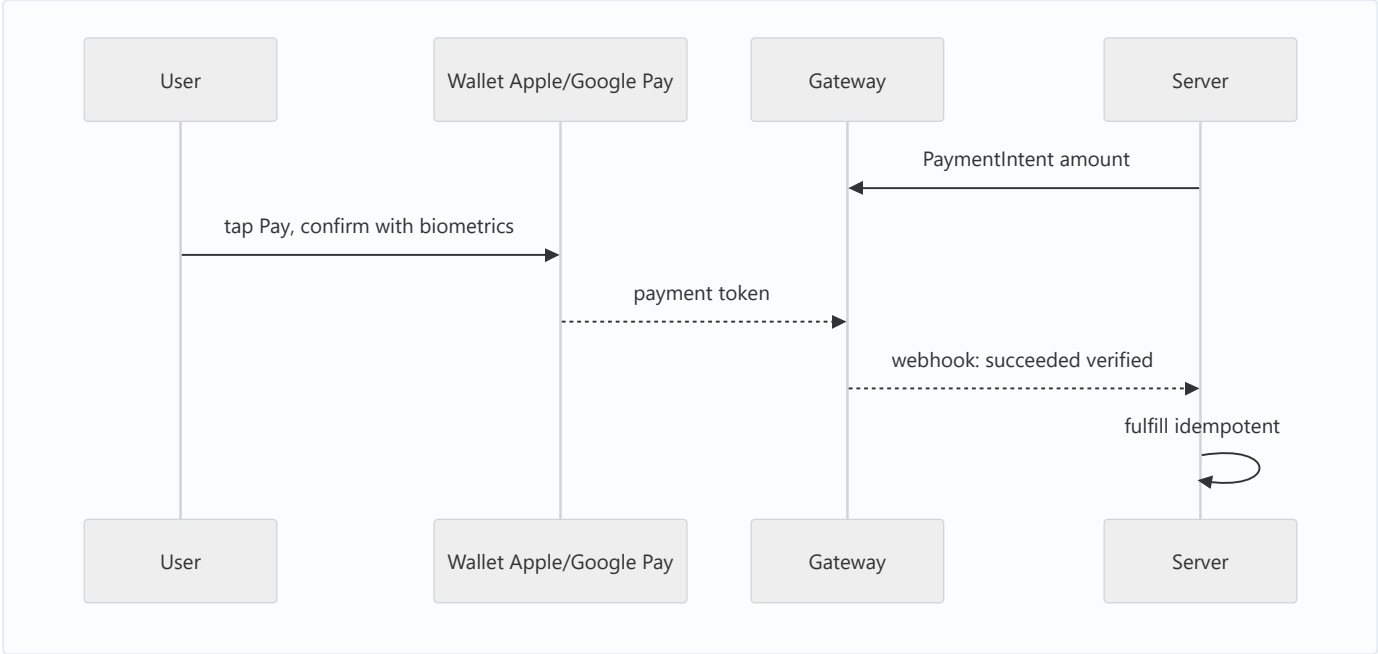
Not applicable.

## Examples

```
// Simplest: enable wallets inside the Stripe payment sheet (reuses your PaymentIntent)
import 'package:flutter_stripe/flutter_stripe.dart';

Future<void> payWithWalletOrCard(String cartId) async {
  final clientSecret = await api.createPaymentIntent(cartId: cartId); // server sets amount
  await Stripe.instance.initPaymentSheet(
    paymentSheetParameters: SetupPaymentSheetParameters(
      paymentIntentClientSecret: clientSecret,
      merchantDisplayName: 'My Shop',
      applePay: const PaymentSheetApplePay(merchantCountryCode: 'US'),   // Apple Pay
      googlePay: const PaymentSheetGooglePay(merchantCountryCode: 'US', testEnv: true), // Google Pay
    ),
  );
  await Stripe.instance.presentPaymentSheet();  // sheet shows Apple/Google Pay + card
  await api.awaitOrderPaid(cartId);             // server truth (webhook)
}
```

```
// Or the `pay` plugin: native button -> token -> your gateway
// import 'package:pay/pay.dart';
// GooglePayButton(
//   paymentConfiguration: PaymentConfiguration.fromJsonString(googlePayConfig),
//   paymentItems: items,                         // amounts (server-authoritative in practice)
//   onPaymentResult: (result) => api.chargeWithWalletToken(result), // forward token to gateway
// );
// Show the button only when the wallet is available; provide a card fallback.
```

## Diagrams




## Common Mistakes

| Mistake | Why | Fix |
|---|---|---|
| Treating wallets as a separate rail | They feed the gateway | Route token through gateway flow |
| Amount from the client wallet items | Tampering | Server-set amount (intent/order) |
| Fulfilling on wallet result | Untrusted | Fulfill on gateway webhook |
| Showing button when unsupported | Broken UX | Check availability; card fallback |
| Missing Apple Pay entitlement/merchant id | Apple Pay unavailable | Configure capability + merchant id |
| Wallets for digital goods | Policy violation | IAP for digital (03_in_app_purchases.md) |

## Best Practices

- Prefer enabling wallets **inside the gateway payment sheet** (reuses your `PaymentIntent`/order + webhook) over a separate integration.
- Keep the **amount server-set** and **fulfill on the webhook** — wallets only change how the card is tokenized, not the trust model.
- **Check availability** and show the wallet button only when supported (Apple Pay iOS, Google Pay Android/web); always offer a **card fallback**.
- Configure **Apple Pay merchant id + entitlement** (28 · ios_integration) / **Google Pay config**; use wallets for **physical/services only** (not IAP goods).

## Performance

Faster checkout (higher conversion) — the main benefit is UX/business, not runtime perf. Confirmation latency is the gateway webhook as usual.

## Advantages / Disadvantages

- + One-tap, biometric, tokenized (secure) checkout; higher conversion; reuses gateway flow.
- − Platform setup (Apple merchant id/entitlement), availability checks + fallback needed, still gateway-bound, not for digital goods.

## Interview Questions

1. 🟢 **Are Google/Apple Pay a separate payment rail?** — No — they supply a tokenized card that you charge through your gateway; they're a faster input method, not a new processor.
2. 🟢 **When do wallets apply vs IAP?** — Wallets are for physical goods/services (via a gateway); digital goods still require IAP.
3. 🟡 **What's the easiest way to add wallets in Flutter?** — Enable Apple/Google Pay inside the gateway's payment sheet (Stripe/Razorpay), reusing the existing PaymentIntent + webhook.
4. 🟡 **Do the security rules change with wallets?** — No — server sets the amount and fulfills on the verified webhook; only card tokenization is handled by the wallet.
5. 🟡 **What does Apple Pay require to work?** — A Merchant ID + Apple Pay capability/entitlement (Xcode), real device, and a card in Wallet.
6. 🔴 **How do you decide whether to show the wallet button?** — Check wallet availability/support for the platform/device and provide a card fallback when unavailable.
7. 🔴 **Why can't you sell a subscription via Apple/Google Pay?** — It's a digital good; store policy mandates IAP regardless of payment method.

## Senior Engineer Tips

- Reuse the gateway sheet's wallet support instead of a bespoke `pay` integration unless you need custom buttons — far less to maintain and it inherits your webhook flow.
- Always gate the wallet button on runtime availability and keep a card fallback; nothing erodes trust like a dead pay button.
- Remember wallets don't change compliance: server amount + webhook fulfillment still apply, and they're never a loophole around IAP.

## Architect Perspective

Wallets are a conversion-optimizing front-end to the existing gateway rail, not a new rail. Architecturally they slot into the same server-amount + webhook-verified-fulfillment flow, adding only availability checks and platform setup. Treating them this way keeps checkout secure and consistent while improving UX for physical commerce — orthogonal to the IAP rail for digital goods (02_payment_gateways.md, 03_in_app_purchases.md).

## Summary

- Google/Apple Pay = fast, biometric, tokenized card input feeding your gateway — not a separate rail and not an IAP substitute.
- Easiest via the gateway payment sheet; same rules (server amount, webhook fulfillment).
- Check availability + card fallback; configure Apple Pay merchant id/entitlement; physical/services only.

## Revision Notes

- Wallets return a tokenized card → charged via gateway; enable inside Stripe/Razorpay sheet (easiest) or via `pay` plugin.
- Same invariants: server-set amount, webhook-verified fulfillment; wallets change only card tokenization.
- Apple Pay: merchant id + entitlement (Xcode), real device; Google Pay: config JSON; availability check + card fallback; not for digital goods (IAP).

## Practice Questions

1. Why are wallets not a separate payment rail?
2. How do wallets fit the server-amount/webhook flow?
3. Why can't wallets be used to sell in-app coins?

## Coding Questions

1. Enable Apple/Google Pay inside the Stripe payment sheet reusing a server PaymentIntent.
2. Gate the wallet button on availability with a card fallback.
3. Forward a `pay`-plugin token to a gateway charge.

## Mini Project

**One-tap wallet checkout (Flutter + backend):** Add Apple/Google Pay to the physical-order checkout by enabling wallets in the gateway payment sheet (server-created PaymentIntent), with an availability check and card fallback, fulfilling on the verified webhook. Acceptance: wallet sheet appears where supported (card fallback otherwise); amount server-set; fulfillment webhook-verified; Apple Pay entitlement/merchant id configured; not used for digital goods; runs on device (test mode).

# Payments Integration (Capstone: Server Verification, Webhooks & Reconciliation)

> The backbone that makes payments trustworthy: the **server is the single source of truth**. It creates intents/orders (amount, secret keys), **verifies completion out-of-band** — gateway **webhooks** (with signature checks) and IAP **receipt validation** (Apple/Google APIs + server notifications) — then **idempotently** grants entitlement/fulfills, and **reconciles** its ledger against the provider. The client only ever *reflects* server-confirmed state.

## Introduction

This module capstone unifies gateways, IAP, and wallets under one server-verification architecture. The recurring theme across every file — don't trust the client — is realized here: a payment isn't "done" until your server has verified it with the provider and recorded it idempotently. This file covers webhooks/receipts, idempotency, entitlement modeling, reconciliation, and the client's (limited) role.

## Why this concept exists

Clients can be tampered, spoofed, disconnected mid-flow, or replay old receipts; networks drop; webhooks duplicate; subscriptions renew/expire independently. Only a server that verifies with the provider and records idempotently can produce correct, fraud-resistant, reconcilable payment state. This is the difference between a demo and a system that handles real money.

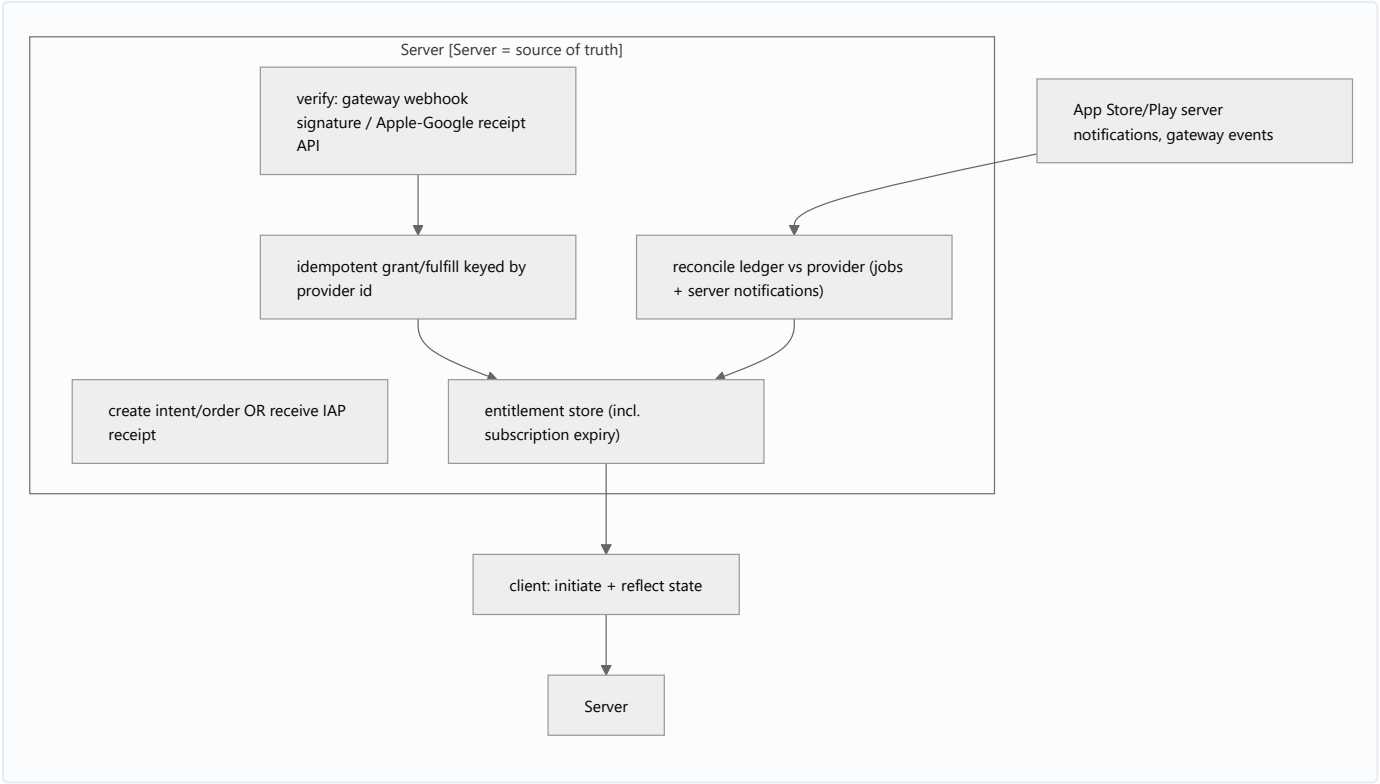
## Real-world analogy

Your server is the **accounting department**. Salespeople (clients) can *say* a sale happened, but accounting only books revenue when the **bank statement** (webhook / verified receipt) confirms settlement — and it **matches its ledger to the bank statement** monthly (reconciliation) so nothing is lost, double-counted, or fraudulent.

## Problem Statement

Build the server-truth layer for a checkout that sells both a **digital** item (IAP) and a **physical** order (gateway): verify each via webhook/receipt, grant/fulfill idempotently, model entitlement (including subscription expiry), reconcile against the provider, and have the client reflect confirmed state. You'll design the verification + entitlement backbone.

## Internal Working




- **Verification per rail**:
  - **Gateway**: receive the **webhook**, **verify its signature** (Stripe signing secret / Razorpay HMAC), then act. Webhooks are the authoritative "it settled" — not the client.
  - **IAP**: send the receipt/token to Apple (App Store Server API) / Google (Play Developer API) to validate; subscribe to **server notifications** (App Store Server Notifications / Play RTDN) for renewals/cancellations/refunds.
- **Idempotency**: every grant/fulfill is keyed by a **stable provider id** (payment_intent id / order id / transaction id) so duplicate webhooks, client retries, and reprocessing don't double-charge, double-ship, or double-grant. Store processed ids.
- **Entitlement model**: a durable server record of *what the user owns* — one-off unlocks, consumable balances, and **subscription state with expiry** (updated by renewal notifications). The client caches a derived flag; the server is authoritative.
- **Reconciliation**: periodically (and via notifications) match your ledger against the provider's records to catch missed webhooks, refunds, chargebacks, and expiries — repair entitlement accordingly.
- **Client role (minimal)**: initiate purchase, present SDK/store UI, then **poll/subscribe for server-confirmed entitlement** and render pending → confirmed/failed. It never grants, never sets amounts, never trusts its own callback.
- **Failure handling**: pending (webhook not yet arrived), failed, refunded, expired — each an explicit state (Module 38).

## Memory Representation

The durable entitlement/ledger lives in your DB (server). The client holds a small cached entitlement snapshot (non-authoritative). Processed-webhook ids are stored to enforce idempotency.

## Compiler Behavior

Not applicable.

## Runtime Behavior

Confirmation is **eventually consistent**: the client sees "pending" until the webhook/receipt is verified and entitlement is written, then reflects it. Renewals/refunds arrive over time via notifications and update entitlement asynchronously.

## Flutter Engine Behavior

Not applicable beyond the SDK/store UIs covered in prior files.

## Dart VM Behavior

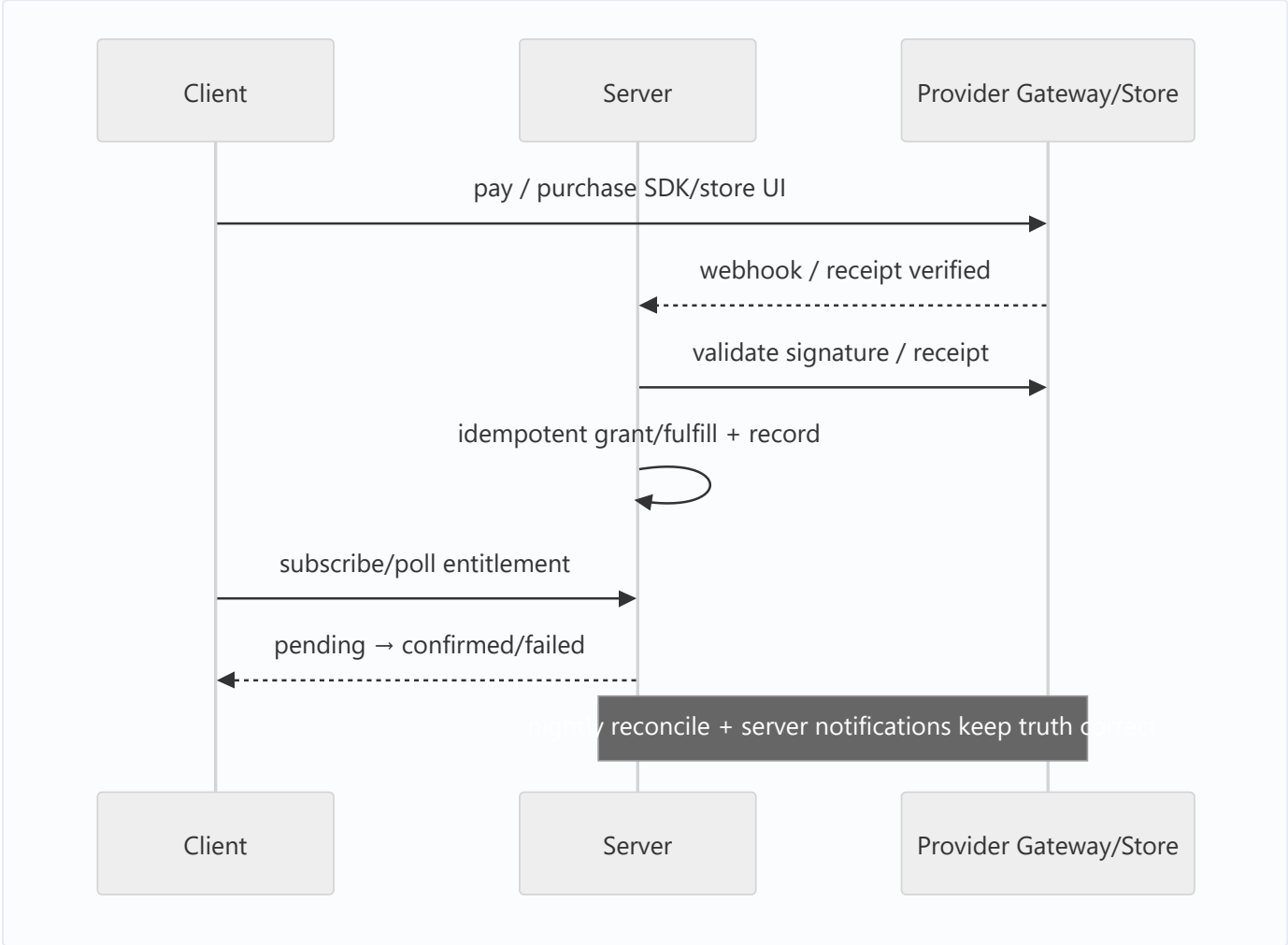
Not applicable (verification/reconciliation are server-side).

## Examples

```
// CLIENT: initiate, then reflect server-confirmed entitlement (never grants itself)
class PaymentFacade {
  Future<Entitlement> buyDigital(String productId) async {
    await iap.buy(productId);                // store flow; receipt verified server-side
    return _awaitEntitlement(productId);     // truth from server
  }
  Future<Entitlement> buyPhysical(String cartId) async {
    await gateway.pay(cartId);               // gateway sheet; fulfilled on webhook
    return _awaitOrderPaid(cartId);          // truth from server
  }
  // Poll/subscribe until the server confirms (handles webhook latency)
  Future<Entitlement> _awaitEntitlement(String id) => api.entitlementStream(id)
      .firstWhere((e) => e.isFinal); // granted | failed | refunded
}
```

```
SERVER (pseudocode) — the part that actually matters:
  on gatewayWebhook(req):
     assert verifySignature(req, SIGNING_SECRET)          # reject spoofed
     if alreadyProcessed(req.paymentId): return 200        # idempotent
     if req.type == 'payment_succeeded':
         fulfillOrder(req.orderId)                         # grant/ship
         markProcessed(req.paymentId)
     return 200

  on iapVerify(receipt):
     result = appleOrGoogle.validate(receipt)              # provider API
     if result.valid and not alreadyProcessed(result.txnId):
         grantEntitlement(result.productId, expiry=result.expiry)
         markProcessed(result.txnId)

  nightly reconcile():
     for record in provider.listSince(lastRun):            # catch missed/refunded
         upsertEntitlement(record)                         # repair truth
```

## Diagrams




## Common Mistakes

| Mistake | Why | Fix |
| --- | --- | --- |
| Skipping webhook signature verification | Spoofed grants | Verify signature before acting |
| Non-idempotent processing | Double grant/ship on dup/retry | Key by provider id; store processed ids |
| Client grants entitlement | Fraud/lost revenue | Server-only grants; client reflects |
| No subscription lifecycle handling | Stale access after cancel/expiry | Use server notifications + expiry |
| No reconciliation | Missed webhooks/refunds unhandled | Periodic ledger vs provider reconcile |
| Optimistic unlock on client success | Wrong on failure/fraud | Show pending → server-confirmed |
| Local flag as source of truth | Diverges from reality | Server authoritative; local is cache |

## Best Practices

- Make the **server the source of truth**: verify **webhook signatures** / **receipts** with the provider before acting; the client only reflects.

- **Idempotent** grant/fulfill keyed by provider id (store processed ids) — assume duplicate webhooks + client retries.
- Model **entitlement durably**, including **subscription expiry** driven by **server notifications**; **reconcile** the ledger against the provider periodically.
- Client: initiate + **poll/subscribe for confirmed state** (pending → confirmed/failed/refunded); never grant/set amount/trust its callback (01_payments_overview_and_security.md).

## Performance

Not throughput-sensitive; the design concern is **eventual consistency** (webhook latency) and **robustness** (idempotency, reconciliation). Client UX handles the pending window; the server converges to correct state via webhooks/notifications/reconciliation.

## Advantages / Disadvantages

- **+** Correct, fraud-resistant, reconcilable payments across both rails; survives retries/duplicates/interruptions; accurate subscription state.
- **−** Requires backend infrastructure (webhooks, provider APIs, notifications, reconciliation jobs), eventual-consistency UX, more moving parts.

## Interview Questions

1. 🟢 **Why is the server the source of truth for payments?** — The client is untrusted (tamperable, replayable, can fail mid-flow); only server-side verification with the provider proves a payment and can be reconciled.
2. 🟢 **What proves a gateway payment vs an IAP purchase?** — A signature-verified **webhook** for the gateway; a provider-validated **receipt** (+ server notifications) for IAP.
3. 🟡 **Why must processing be idempotent, and how?** — Webhooks duplicate and clients retry; key grant/fulfill by a stable provider id and store processed ids so it runs once.
4. 🟡 **How do you keep subscription entitlement correct over time?** — Track expiry in a durable entitlement store updated by App Store/Play server notifications (renewals, cancellations, refunds).
5. 🟡 **What is reconciliation and why do it?** — Periodically matching your ledger against the provider's records to catch missed webhooks, refunds, and chargebacks and repair entitlement.
6. 🔴 **How should the client behave during webhook latency?** — Show pending and subscribe/poll for the server-confirmed final state (granted/failed/refunded) — never optimistically unlock.
7. 🔴 **Design an end-to-end trustworthy payment for both rails.** — Server creates intent/receives receipt → verifies (signature/provider API) → idempotently grants/fulfills →

durable entitlement (+ notifications) → reconciliation → client reflects confirmed state.

## Senior Engineer Tips

- Store every processed provider id and make grant/fulfill idempotent before you write any happy-path code — duplicates and retries are guaranteed, not rare.
- Drive subscriptions from server notifications + an expiry field, not from client checks; a cancelled sub that keeps working is a common, costly bug.
- Add reconciliation early; the day a webhook is missed (and it will be), reconciliation is the only thing that keeps entitlement correct.

## Architect Perspective

Payments integration is a distributed-systems trust problem: an untrusted client, an asynchronous provider, duplicate/lost messages, and time-varying entitlements. The architecture — server-side verification, idempotency, durable entitlement with server notifications, and reconciliation — is what makes money movement correct and auditable. The client shrinks to "initiate + reflect confirmed state," and both rails (gateway/IAP) converge on the same backbone, consistent with the app's overall clean-architecture trust boundaries (01_payments_overview_and_security.md, Module 40, Module 38).

## Summary

- Server = source of truth: verify webhook signatures / receipts with the provider before acting.
- Idempotent grant/fulfill by provider id; durable entitlement with subscription expiry via server notifications; reconcile periodically.
- Client initiates and reflects server-confirmed state (pending → confirmed/failed/refunded); both rails share this backbone.

## Revision Notes

- Gateway → signature-verified webhook; IAP → provider receipt validation + server notifications (App Store Server Notifications / Play RTDN).
- Idempotency keyed by provider id (store processed ids); durable entitlement incl. subscription expiry; periodic reconciliation vs provider.
- Client: initiate + poll/subscribe for confirmed entitlement (pending→final); never grants/sets amount/trusts callback.

## Practice Questions

1. Why is idempotency essential in webhook processing?
2. How is subscription entitlement kept correct after a cancellation?
3. What does reconciliation protect against?

## Coding Questions

1. Design an idempotent gateway-webhook handler (signature check + processed-id store).
2. Design IAP receipt verification + entitlement grant with expiry.
3. Implement a client that shows pending and reflects server-confirmed entitlement.

## Mini Project

**Trustworthy payments backbone (capstone — Flutter + backend):** Build the server-truth layer for a checkout selling a digital item (IAP) and a physical order (gateway): signature-verified webhook + provider receipt validation, idempotent grant/fulfill keyed by provider id, a durable entitlement store with subscription expiry driven by server notifications, and a nightly reconciliation job. The client initiates and reflects server-confirmed state (pending → confirmed/failed/refunded). Acceptance: both rails verified server-side; idempotent (duplicate webhook safe); subscription expiry honored via notifications; reconciliation repairs missed/refunded; client never grants/sets amount; runs end-to-end (test/sandbox).