JUNEPAY

API & Integration Guide

Host-to-host (h2h) UPI collections - authenticate, create a payment, show the customer a QR / UPI app, and receive a signed webhook when the money arrives. API-only; no hosted page required.

Revision **2026-07-13** · see Changelog · `Base URL = https://junepay.co` (the origin you sign in to the portal from)

Changelog

Each revision is dated. ⚠ **affects money** marks a change that can alter what you credit or settle - read those before upgrading a live integration.

Date	Change	
2026-08-30	**ToPay rail is now a `SUCCESS`-only callback stream.** ToPay can settle a payment hours after the order's window, so we suppress the `EXPIRED` callback for `topay` / `topay_intent` orders and reconcile late payments hourly (they still fire one `SUCCESS`). Do not wait for a failure callback on ToPay orders — treat an order you never hear about as unpaid, or poll `check-order-status`. Agent-pool/JunePay orders still get `EXPIRED`. `PENDING`/in-flight orders never call back on any rail.	⚠ **affects money**
2026-07-14	**Webhook signature is now HMAC-SHA256 of the RAW request body** (the Stripe/Razorpay convention) — one `x-signature`, verify it against the exact bytes (e.g. `express.raw()`). The old canonical-field-list scheme and the inline `signature` body field are gone. Because it signs the whole body, every field (incl `test_mode`) is authenticated.	⚠ **affects money**
2026-07-14	`check-order-status` and `reconcile` now return `settled_amount` / `balance_due` / `amount_ordered`, so a poll can express a `PARTIAL` (the safety net for a missed webhook). Documented the `REFUND` webhook payload and that `settled_amount` is gross of commission. Verification examples use constant-time compare; the worked handler processes before ACKing.	⚠ **affects money**
2026-07-13	Documented the `amount_ordered` / `amount_received` webhook fields and clarified that `amount` / `settled_amount` is the *collected* (nudged) value, not your invoice. Added a worked webhook handler and an HMAC test vector.	⚠ **affects money**
2026-07-13	Introduced the `PARTIAL` event/status (underpayments). If your handler credited the full ordered amount on any settle, it would over-credit a partial - branch on the event and credit `settled_amount`.	⚠ **affects money**
2026-07-08	First published revision.	

1. Getting started & authentication

Every server-to-server call is authenticated with your **API key** in the `x-api-key` header. Find your keys on **Portal → API & Integration**.

Key	Prefix	Behaviour
Live	`mk_live_…`	Real collections against your account.
Test (sandbox)	`mk_test_…`	Auto-settles ~5s after create; ledger-isolated (no real balance/agent writes). Use it to build & test end-to-end safely.

Your **API secret** is used only to **verify HMAC signatures** on webhooks and the `hash` field of status responses - never send it in a request. Rotate it any time from the portal.

IP allowlist (optional). You can restrict `POST /api/v1/payment-requests` to specific client IPs (Portal → API & Integration → IP Access Control). Empty = allow all. Requests from other IPs get `403`.

2. Create a payment request

POST `https://junepay.co/api/v1/payment-requests`

```
POST https://junepay.co/api/v1/payment-requests
x-api-key: mk_live_xxxxxxxxxxxxxxx
Content-Type: application/json

{
  "amount": 500.00,
  "orderId": "INV-1001",
  "customerName": "Ravi Kumar",
  "customerMobile": "9999999999",
  "callbackUrl": "https://yourdomain.com/webhook",
  "redirectUrl": "https://yourdomain.com/thank-you"
}
```

Response (abridged):

```
{
  "ok": true,
  "id": "5e3f…",                          // intent id
  "orderId": "INV-1001",
  "amount": "500.00",                     // what you requested
  "amount_charged": "500.01",             // CHARGE THIS — see note below
  "paymentLink": "https://junepay.co/c/5e3f…",       // hosted checkout — ALWAYS present, always safe
  "checkout": { "mode": "upi_intent", "url": "https://junepay.co/c/5e3f…" },  // mode: upi_intent |
hosted_redirect
  "upiDeepLink": "upi://pay?pa=…&am=500.01&…",   // null when mode = hosted_redirect
  "qr": "https://junepay.co/api/v1/transactions/5e3f…/qr",   // UPI QR PNG — null when mode = hosted_redirect
  "vpa": "receiver@bank",                        // null when mode = hosted_redirect
  "payeeName": "…",                              // null when mode = hosted_redirect
  "expiresAt": "2026-07-01T10:20:00Z",
  "status": "pending",
  "testMode": false
}
```

Field	Required	Notes
`amount`	yes	Rupees (decimal), **not paise** - e.g. `500.00` = ₹500. Must be within your per-transaction min/max.
`orderId`	recommended	Your reference. **Idempotent**: repeating a live/approved order replays it (adds `idempotent:true`) instead of creating a duplicate.
`customerName`, `customerMobile`, `customerEmail`	optional	Shown on the checkout / attached to the order.
`callbackUrl`	optional	Per-order webhook target (overrides the dashboard webhook URL for this order).
`redirectUrl`	optional	Where the hosted checkout sends the customer **after** success. To send them **to** the payment page, use the response's `paymentLink` / `checkout.url` - not this field.
`ttlSeconds` / `expiresInMinutes`	optional	Payment window override, clamped 60–1200s (1–20 min). Default 10 min.

Charge `amount_charged`**, not** `amount`**.** To keep two same-amount payments to the same receiver distinguishable, we may nudge the amount up by up to ₹0.99. Show and collect `amount_charged`. Match the resulting webhook back to your order by `order_id` - **not** by amount. **The webhook's** `amount` **is the *collected* value (i.e. the nudged** `amount_charged`**), not your invoice**; your original invoice is echoed back as `amount_ordered`. Do not compare the webhook `amount` to what you invoiced and expect them to be equal - they differ by the nudge. See Outbound webhook for the exact field meanings and a worked handler.

3. Show the customer the payment

Recommended integration: Hosted checkout + webhook. Create the order, redirect the customer to `paymentLink`, and treat the `SUCCESS` webhook as your source of truth. This is the least code and the most reliable.

Pick whichever fits your flow - all come from the create response:

- **Hosted checkout (recommended)** - redirect the customer to `paymentLink` (`https://junepay.co/c/<id>`). **This page is already built and public** - it renders the QR, UPI-app buttons, live status (SSE + safety poll), and your success redirect, in your chosen theme. You build nothing, and it works no matter how your account is configured.
- **Your own page** - render the QR from `qr` (a PNG URL), and/or offer `upiDeepLink` / `phonepeDeepLink` / `paytmDeepLink` as "Pay with UPI" buttons on mobile. Only available when `checkout.mode` is `upi_intent` - see the note below.

Branch on `checkout.mode`. JunePay serves an order one of two ways, and the create response tells you which:

- `upi_intent` - the host-to-host fields (`upiDeepLink`, `qr`, `vpa`, `payeeName`, `txnRef`) are populated; you may build your own page from them.
- `hosted_redirect` - those fields come back `null`; send the customer to `checkout.url` (= `paymentLink`) instead.

`paymentLink` / `checkout.url` is **always** present and always safe - if you only ever redirect there, you never have to think about the mode.

The customer pays from any UPI app. Credits are matched automatically by our rail feed - no UTR entry needed in the common case.

4. Check order status

POST `https://junepay.co/api/v1/check-order-status` - poll a single order (rate-limited 240/min).

```
POST https://junepay.co/api/v1/check-order-status
x-api-key: mk_live_…
{ "order_id": "INV-1001" }

→ { "ok": true, "status": "SUCCESS", "order_id": "INV-1001",
    "intent_id": "5e3f…",
    "amount": "500.01",          // charged (nudged/rounded); = amount_charged, NOT your invoice
    "amount_ordered": "500.00",  // your invoice
    "amount_charged": "500.01",  // what the customer pays
    "settled_amount": "500.01",  // credit THIS (null until settled; < amount_ordered on PARTIAL)
    "balance_due": "0.00",       // shortfall on a PARTIAL
    "refunded": "0.00", "utr": "4123…",
    "verified_at": "2026-07-01T10:12:30Z",
    "hash": "<HMAC-SHA256(order_id, apiSecret)>" }
```

`status` ∈ `SUCCESS` · `PENDING` · `PARTIAL` · `FAILED` · `EXPIRED` (also `REFUNDED`). `PARTIAL` only occurs if partial settlement is enabled for your account — this response carries `settled_amount` / `balance_due` so a poll (your safety net for a missed webhook) can express it. **Note the word `amount`:** in the **create** response and **reconcile** it's your invoice, but here (and in the webhook) it's the **charged** value — always use the explicit fields (`amount_ordered` for your invoice, `settled_amount` for what to credit). Verify authenticity by recomputing `hash = HMAC-SHA256(order_id, your_api_secret)`.

5. Live status stream (SSE)

GET `https://junepay.co/api/v1/transactions/<id>/stream` - Server-Sent Events; emits `data: {status,utr,verifiedAt}` on change, 15s heartbeat, closes when the order reaches `approved` / `rejected` / `expired`. Handy for a live "waiting for payment" UI without polling.

6. List & batch reconcile

List transactions

GET `https://junepay.co/api/v1/transactions?status=SUCCESS&from=ISO&to=ISO&limit=1..200&offset=` → `{ok,count,transactions:[…]}`. Each row: `order_id, intent_id, status, amount, amount_charged, utr, payer_vpa, payer_name, payer_channel, test_mode, created_at, verified_at`.

Batch reconcile (nightly)

POST `https://junepay.co/api/v1/reconcile` - one round-trip instead of N status calls (≤500 orders, 60/min).

```
{ "orders": [ { "order_id":"INV-1001", "amount":500.00, "utr":"4123…" }, … ] }

→ { "ok":true,
    "summary": { "matched":.., "pending":.., "unsettled":.., "mismatched":.., "missing":.. },
    "results": [ { "order_id":"INV-1001", "found":true, "status":"SUCCESS",
                   "reconciled":true, "amount_requested":"500.00",
                   "amount_charged":"500.01",
                   "settled_amount":"500.01",   // credit THIS (null until settled)
                   "balance_due":"0.00",        // shortfall on a PARTIAL
                   "utr":"4123…", "discrepancy": null } ] }
```

`discrepancy` ∈ `AMOUNT_MISMATCH` · `UTR_MISMATCH` · `NOT_FOUND`. Your input `amount` is checked against what you *requested* (= `amount_requested`, your invoice — not the nudged charge). `settled_amount` / `balance_due` let reconcile express a `PARTIAL` too, so it's a complete safety net for a missed webhook.

7. Webhooks - signed callbacks (the reliable way)

We **POST** to your `callbackUrl` (per-order) or your dashboard **Webhook URL** on every terminal state. This is the recommended source of truth - you don't need to poll. Your endpoint must be **publicly reachable over HTTPS** (we POST from our servers, not the browser) - for local development, expose it with a tunnel (e.g. ngrok / cloudflared) and set that URL, or use the **"Send test webhook"** button on Portal → API & Integration.

```
POST <your callbackUrl>
x-signature: <hmac>            // HMAC-SHA256 of the RAW request body (Stripe/Razorpay-style)
x-event-id: <uuid>
Content-Type: application/json

{ "event_id":"<uuid>", "event":"SUCCESS", "status":"SUCCESS",
  "order_id":"INV-1001", "intent_id":"<uuid>",
  "amount":"499.01",          // COLLECTED value = the nudged amount_charged (NOT your invoice)
  "amount_ordered":"499.00",  // your original invoice (what you sent as `amount` on create)
  "amount_received":"499.01", // rail-verified amount that actually arrived
  "settled_amount":"499.01",  // what your credit is based on (= amount)
  "balance_due":"0.00",       // > 0 only on PARTIAL (amount_ordered − settled_amount)
  "commission":"0.00", "utr":"4123…", "emitted_at":"2026-07-08T12:00:00.000Z",
  "test_mode":false }
```

Which amount is which. `amount` and `settled_amount` are the *collected* value (the nudged `amount_charged`); they are equal. `amount_ordered` is your original invoice. **Match the webhook to your order by** `order_id`**, credit** `settled_amount`**, and never equality-check** `amount` **against your invoice** - they differ by the ≤₹0.99 nudge. On `PARTIAL`, `settled_amount < amount_ordered` and `balance_due` is the shortfall - still credit `settled_amount`, don't assume the order was paid in full.

Event	When	`utr`
`SUCCESS`	Payment approved	set
`PARTIAL`	Underpaid - settled short of the ordered amount. **Opt-in**: only if partial settlement is enabled for your account. Credit `settled_amount` (not `amount`); `balance_due` is the shortfall.	set
`FAILED`	Rejected	null
`EXPIRED`	Payment window lapsed	null
`REFUND`	Order refunded (full/partial)	-

ToPay rail is `SUCCESS` **-only.** If your account settles over ToPay (`topay` / `topay_intent` — "Our Checkout QR" and hosted redirect), a payment can confirm **hours** after the order's window lapses (their bank settles late). To avoid a confusing `EXPIRED`-then-`SUCCESS` sequence, we **suppress the** `EXPIRED` **callback** for ToPay orders and reconcile late payments hourly — so a late payment still credits and still fires a single `SUCCESS`. Practically, ToPay orders emit `SUCCESS` only. **Do not wait for a failure callback on a ToPay order** — treat one you never hear about as unpaid, or poll `check-order-status`. (Agent-pool / JunePay orders *do* get a terminal `EXPIRED`.) A `PENDING` / in-flight order **never** produces a callback, on any rail.

Verify the signature + handle the event (Node)

`x-signature` is **HMAC-SHA256 of the raw request body** (the Stripe/Razorpay convention). Read the **exact bytes** - use `express.raw()`, don't hash a re-serialised parsed object. Or drop in a `/sdk/` client: `verifyWebhook(rawBody, req.headers['x-signature'])`.

```
const crypto = require('crypto');
app.post('/webhook', express.raw({ type: 'application/json' }), async (req, res) => {
  const raw = req.body;                                  // Buffer of the RAW bytes (express.raw)
  const expected = crypto.createHmac('sha256', API_SECRET).update(raw).digest('hex');
  const sig = String(req.headers['x-signature'] || '');
  // Constant-time compare — NOT === (avoids a timing side-channel on the HMAC).
  const ok = expected.length === sig.length &&
    crypto.timingSafeEqual(Buffer.from(expected), Buffer.from(sig));
  if (!ok) return res.sendStatus(401);                   // reject forgeries
  const b = JSON.parse(raw);
  if (b.test_mode) return res.sendStatus(200);           // test ping (Send test webhook / mk_test_) — just ACK

  // PROCESS (or durably persist) BEFORE you ACK. A 2xx tells us to STOP retrying, so if you
  // ACK first and then crash, the event is lost forever. Dedupe on event_id (at-least-once).
  try {
    if (!(await alreadyHandled(b.event_id))) {
      const order = lookupByOrderId(b.order_id);         // match by order_id, NOT amount
      if (b.event === 'SUCCESS' || b.event === 'PARTIAL') {
        credit(order, b.settled_amount);                 // credit what SETTLED (gross of commission)
        if (b.event === 'PARTIAL') flagShortfall(order, b.balance_due);
      } else if (b.event === 'FAILED' || b.event === 'EXPIRED') {
        markUnpaid(order);
      } else if (b.event === 'REFUND') {
        recordRefund(order, b.refund_id, b.amount, b.total_refunded, b.fully_refunded);
      }
      await markHandled(b.event_id);                     // commit BEFORE the ACK below
    }
    res.sendStatus(200);                                 // ACK only once the work is durable
  } catch (e) {
    res.sendStatus(500);                                 // return non-2xx → we retry (up to 8x)
  }
});
```

HMAC test vector

Verify your implementation offline before you deploy - this raw body, signed with this secret, must produce this `x-signature`:

```
secret       = whsec_test_9c1e5b7a2d4f6081
raw body     = {"event":"SUCCESS","order_id":"INV-1001","settled_amount":"499.01"}
x-signature  = 0b690db13c8780f8af65839af8a1369263835cc21d7f556d5d352ced9627dd8c
```

At-least-once delivery - dedupe. Retries use exponential backoff (up to 8 attempts, then a dead-letter queue you can replay from the portal) and carry the **same** `x-event-id`. Make your handler idempotent by deduping on `event_id`. Return `2xx` **only after** you've durably handled the event — a non-2xx has us retry, which is exactly what you want if your processing failed.

The signature covers the whole payload. Because it's an HMAC of the raw body, **every** field is authenticated (including `test_mode`) — a valid signature means nothing was altered. `settled_amount` is **gross** of commission (the collected amount); your balance is credited `settled_amount − commission`, so credit `settled_amount` to the customer's order and treat `commission` as your fee.

Rescue re-fires a fresh event. If a late credit rescues an `EXPIRED` order, the follow-up `SUCCESS` carries a **new** `event_id` (a distinct event, not a retry) — so deduping on `event_id` will **not** drop it. Dedup only ever collapses retries of the same event, which reuse their id.

Inbound source IPs. We don't publish fixed egress IPs for webhook delivery — authenticate inbound with the HMAC signature, not an IP allowlist. (The IP allowlist in the portal applies to your outbound `create` calls only.)

8. Balance

GET `https://junepay.co/api/v1/balance` → `{ balance, pending_settlement, available, lifetime_collected }`. Amounts in rupees.

9. Refunds

A refund **records a reversal** against a collected order - it adjusts your ledger, flips the order to `REFUNDED` (when fully reversed) and fires a `REFUND` webhook. Returning the funds to the customer is handled **separately by the platform**; the API records the reversal, it does not itself push money back.

- **Observe (always available):** **GET** `https://junepay.co/api/v1/transactions/<id>/refunds`; `check-order-status` returns `REFUNDED` + a `refunded` amount.
- **Record (enabled per account, off by default):** **POST** `https://junepay.co/api/v1/transactions/<id>/refund` `{ amount?, reason?, ref? }` - full (omit `amount`) or partial, idempotent on `ref`, never above the collected balance. Returns `403 REFUNDS_NOT_ENABLED` until the platform admin turns it on for you.

A recorded refund fires a `REFUND` webhook (same signature scheme):

```
{ "event_id":"<uuid>", "event":"REFUND", "status":"REFUND",
  "order_id":"INV-1001", "intent_id":"<uuid>",
  "refund_id":"<id>",
  "amount":"100.00",           // THIS refund's amount (the SIGNED value)
  "settled_amount":"100.00",   // = amount (alias, so it's covered by the signature)
  "balance_due":"0.00",        // always 0.00 on a refund
  "total_refunded":"100.00",   // cumulative across all refunds on this order
  "fully_refunded":false,      // true once the whole collected amount is reversed
  "utr":"4123…", "emitted_at":"…", "test_mode":false }
```

Same `x-signature` (HMAC of the raw body) as any webhook — `amount` here is the refunded value. Reconcile the running total via `GET .../refunds`.

10. Errors & rate limits

Error shape: `{ "ok": false, "code": "<CODE>", "message": "…", "error": { "code": "<CODE>", "message": "…" } }` with an HTTP status. Match on `code` (or `error.code`).

HTTP	Code	Meaning
400	`MISSING_PARAMS` / `BAD_AMOUNT`	Missing/invalid fields.
401	`UNAUTHORIZED_KEY`	Bad or missing `x-api-key`.
403	`MERCHANT_NOT_APPROVED`	Account not approved, or caller IP not allowlisted.
429	`RATE_LIMITED`	Slow down. Create 120/min, status 240/min, reconcile 60/min per key.
500	`INTERNAL`	Server-side fallback (generic errors may carry code `error`). Retry idempotently using the same `orderId`.

Idempotency & retries. A repeat create for the same `orderId` on a live/approved order replays the original - safe to retry on timeouts. Only `expired` / `rejected` orders create a fresh intent.

11. Reference SDK & quickstart

Zero-dependency SDKs ship for **Node, Python, PHP, Ruby, Go, Java, and C#** - in your JunePay portal under **API & Integration → SDKs** (tabbed, with copy-paste quickstarts), or download a file directly (e.g. `https://junepay.co/sdk/junepay-node.js`). Each exposes `createPayment`, `getStatus`, `waitForTerminal`, `getBalance`, and `verifyWebhook` (which verifies the signature for you). Any other language generates from the OpenAPI spec at `https://junepay.co/openapi.yaml`.

curl - create & poll

```
# create
curl -s https://junepay.co/api/v1/payment-requests \
  -H "x-api-key: $KEY" -H "Content-Type: application/json" \
  -d '{"amount":500,"orderId":"INV-1001","redirectUrl":"https://you/thanks"}'

# poll
curl -s https://junepay.co/api/v1/check-order-status \
  -H "x-api-key: $KEY" -H "Content-Type: application/json" \
  -d '{"order_id":"INV-1001"}'
```

Verify a webhook (Node)

```
const crypto = require('crypto');
function verify(rawBody, signature, apiSecret) {          // rawBody = the EXACT bytes received
  const expected = crypto.createHmac('sha256', apiSecret).update(rawBody).digest('hex');
  const sig = String(signature || '');
  return expected.length === sig.length &&
    crypto.timingSafeEqual(Buffer.from(expected), Buffer.from(sig));   // constant-time, not ===
}
// Or just drop in an SDK (portal → API & Integration → SDKs) — verifyWebhook(rawBody, sig) does this.
```

Go-live checklist: build against a `mk_test_` key → verify you receive & validate the `SUCCESS` webhook → make the handler idempotent on `event_id` → charge `amount_charged` → switch to your `mk_live_` key.

12. Reference & FAQ (details)

Payment lifecycle (state machine)

Internal status	Public status	Meaning
`pending`	PENDING	Created, awaiting payment.
`paid_submitted`	PENDING	A UTR/credit is attached (customer-entered or auto-matched from the rail feed); awaiting final verify.
`approved`	**SUCCESS**	Verified. Money credited. (approved ⇔ SUCCESS.)
`partial`	**PARTIAL**	Settled short of the ordered amount (opt-in feature). `settled_amount` < `amount`, `balance_due` > 0.
`rejected`	**FAILED**	Verification failed. (rejected ⇔ FAILED.)
`expired`	EXPIRED	Payment window lapsed with no matched credit.
`reversed`	REFUNDED	Fully refunded after success.

Transitions are forward-only: `paid_submitted` never returns to `pending`. The only "backward" path is a merchant **Rescue** that pulls an `expired` order to `paid_submitted` when the customer paid late.

Amount nudging - exact spec

Nudge is **+₹0.00 to +₹0.99, upward only** (never below your requested amount, never more than ₹0.99 over). It guarantees each *active* pending order on the same receiving account within the 30-min window has a unique paise value, so the rail feed can attribute a credit unambiguously. If all 100 slots for an amount are taken, create returns `503 NUDGE_EXHAUSTED` (retry shortly). Nudging is **not merchant-configurable**. **Show and charge** `amount_charged`; reconcile against your original `amount`.

Idempotency - exact semantics

Keyed on `orderId` **only**, scoped **per merchant**, over *active* states (`pending` / `paid_submitted` / `approved`). A repeat create replays the **original** order and adds `idempotent:true` - the new request body (amount, callbackUrl, customer details) is **ignored** on replay. `orderId` is **reusable** once the prior order is `expired` or `rejected` (creates a fresh order). It is **not globally unique** - two merchants may both use `INV-1001`. Store our `intent_id` (globally-unique UUID) as well; quote it to support.

Webhook retry policy

Up to **8 attempts**, then dead-letter (replayable from the portal or `POST /api/v1/webhooks/deliveries/:id/replay`). Backoff is **jittered exponential**: `random(0..1) × 5s × 2^attempt`, capped at **1 hour** per gap (≈ up to a few hours total). Per-attempt HTTP timeout **10s** - return `2xx` fast. Retries are **serial** (not parallel) and carry the **same** `event_id` / `x-event-id` - it is minted once per event and stable across every retry, so dedupe on it.

Signature (HMAC) - exact

Webhooks: `x-signature = HMAC_SHA256(your_api_secret, rawRequestBody)`, output **lowercase hex**. Verify against the **exact bytes** you received (e.g. `express.raw()`), not a re-serialised object. Because it signs the whole body, every field is authenticated.

check-order-status: the response carries a separate `hash = HMAC_SHA256(your_api_secret, order_id)` (UTF-8, lowercase hex) so you can confirm a polled status is authentic.

Rate limits

Endpoint	Limit	Scope
create payment-requests	120 / min	per API key
check-order-status	240 / min	per API key
transactions (list)	120 / min	per API key
reconcile	60 / min	per API key
balance	120 / min	per API key

Fixed 60-second window. Every key-authed response carries `X-RateLimit-Limit`, `X-RateLimit-Remaining` and `X-RateLimit-Reset` (epoch seconds); a `429` also sends `Retry-After` (seconds). No separate burst/daily caps. There is polling headroom alongside SSE - but prefer the webhook as truth and don't poll tighter than a few seconds.

Error responses

Shape: `{ "ok": false, "code": "<CODE>", "message": "<text>", "error": { "code": "<CODE>", "message": "<text>" } }`. Match on the `code` field (mirrored at top level and under `error`).

HTTP	Code	When

400	`MISSING_PARAMS` · `BAD_AMOUNT` · `TX_TOO_SMALL` · `TX_TOO_LARGE`	Bad/missing fields or amount outside your limits.
401	`UNAUTHORIZED_KEY`	Missing/invalid `x-api-key`.
403	`MERCHANT_NOT_APPROVED` · `IP_NOT_ALLOWED` · `REFUNDS_NOT_ENABLED`	Account not approved / caller IP not in your allowlist / refunds not enabled for your account.
409	`DUPLICATE_ORDER`	Rare create race (normally you get an idempotent replay instead).
429	`RATE_LIMITED` · `DAILY_CAP`	Rate limit / merchant daily collection cap hit.
503	`NO_ACCOUNT_CAPACITY` · `NO_ACTIVE_ACCOUNTS` · `NO_RULE_MATCHED_ACCOUNTS` · `NUDGE_EXHAUSTED`	No receiver has headroom / none are active / none match your routing rule / all nudge slots for that amount are taken. Transient - retry shortly.
500	`INTERNAL`	Server-side fallback (generic errors may carry code `error`). Retry idempotently with the same `orderId`.

Edge cases

Scenario	What happens
Customer pays **after expiry**	Not auto-approved (the matcher only looks at active orders in a 30-min window). The credit is real and sits in the receiving account; the merchant can **Rescue** the expired order (attach the UTR) → operator approves → SUCCESS + webhook. Not auto-refunded.
Customer pays the **wrong amount** (e.g. ₹499.00 vs charged ₹499.01)	No automatic amount-match; the order stays pending and expires. The credit is unattributed pending manual reconciliation. Always collect `amount_charged` to avoid this.
Customer **pays twice**	A duplicate UTR is blocked at approval (`409`, never double-credited). A genuine second payment lands as an unattributed overpayment (each active order has a unique nudged amount) and is handled by a manual refund.

Test mode

Same environment and endpoints - distinguished only by a `mk_test_…` key. A test order **auto-settles ~5s** after create (synthetic UTR), fires the **real webhook** with `test_mode:true`, and is **ledger-isolated** (no real balance/agent/account writes). Filter test orders client-side by the `test_mode` flag.

QR & deep links

QR is a **server-rendered PNG** (320px, medium error-correction), `Cache-Control: public, max-age=300`. No SVG. It encodes the UPI intent; the *payment* expires with the window (the image URL keeps serving). `upiDeepLink` is a standard `upi://pay` link and works with **every** UPI app (Google Pay, PhonePe, Paytm, BHIM, Amazon Pay, CRED, Navi, …) via the OS chooser. `phonepeDeepLink` / `paytmDeepLink` are enhanced native links (Paytm's requires platform partner config, else null).

Payment methods, settlement, security & retention

- **Methods:** UPI only - UPI intent (deep link) + UPI QR. No UPI collect-request push, UPI Lite, autopay, wallets, cards, or netbanking.

- **Commission = your own fee (MDR).** The `commission` field is *your* cost on the order (deducted at approval - your ledger is credited `amount − commission`), shown so you can reconcile net vs gross. It is not a platform-internal number, and the platform's own economics (what it pays its liquidity agents, its margin) are never exposed to you. GST is not modelled.
- **What you don't see:** the specific receiving VPA/bank that collected a payment is a platform supply-side identity and is **not** exposed on your transactions - you see your customer (payer), the UTR, and your amount/fee/net.
- **Settlement:** request-based (dashboard/API `/api/merchant/settlements`), **admin-approved**, paid out by **bank transfer (NEFT/IMPS)** - not instant/auto T+0/T+1, no UPI payout. Reports via `/api/merchant/export.csv`.
- **txnRef:** the UPI `tr=` reference we generate for the order (our internal reference) - **not** the bank UTR (the UTR arrives with the actual payment).
- **Security:** the IP allowlist applies to **create** only; the webhook signature is provided (verifying it is your responsibility - do it); secrets are rotatable; dedupe on `event_id` for replay protection. Callback URLs are not pre-validated - set them carefully.
- **Branding:** hosted checkout supports themes, a custom (whitelabel) domain, and your success/failed redirect URLs. Set them on Portal → API & Integration / Checkout theme.
- **Retention:** transaction & webhook history are retained indefinitely (no auto-purge); SSE is live-only (no history).
- **Reconciliation flow:** webhook = real-time truth → nightly `POST /api/v1/reconcile` as the safety net → `check-order-status` for one-offs.
