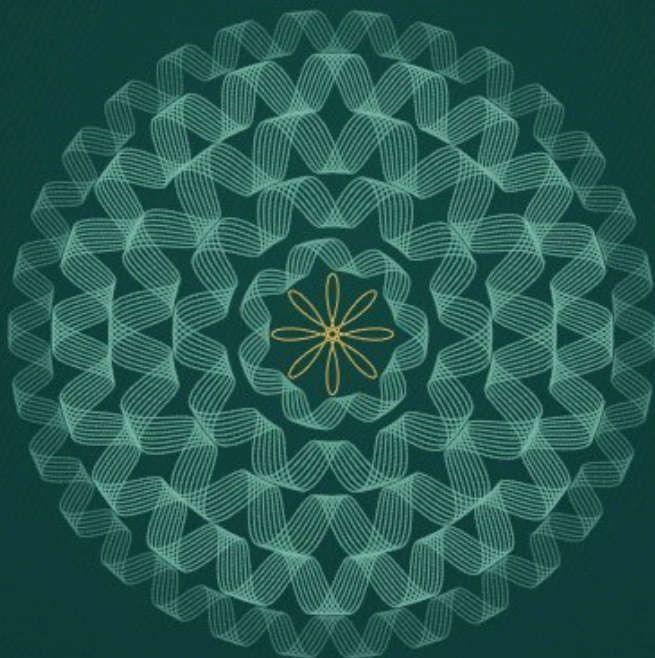


Money Never *Lies*

Designing Payment Systems
That Survive Production

MONEY NEVER LIES - IDEMPOTENCY - RECONCILIATION - DOUBLE-ENTRY - MONEY NEVER LIES - IDEMPOTENCY - RECONCILIATION - DOUBLE-ENTRY



MONEY NEVER LIES - IDEMPOTENCY - RECONCILIATION - DOUBLE-ENTRY - MONEY NEVER LIES - IDEMPOTENCY - RECONCILIATION - DOUBLE-ENTRY

Retries, timeouts, webhooks and month end reconciliation: the parts tutorials skip.

ALWY

MNL 0213 SAT

Chapter 5: Idempotency, the Most Important Word in Payments

READING TIME: ABOUT 15 MINUTES

08:17, the train went into a tunnel

At 08:17 on a Monday, a buyer on a commuter train tapped **Pay** for a USD 64 jacket on Kiosk & Co. The request left the phone, reached Kiosk & Co's API, and was forwarded to the card provider. The provider charged the card.

Then the train went into a tunnel. The response never reached the phone. After ten seconds, the app's HTTP client gave up and, helpfully, retried. The retry arrived at the API as a brand-new request, which created a brand-new payment, which charged the card again.

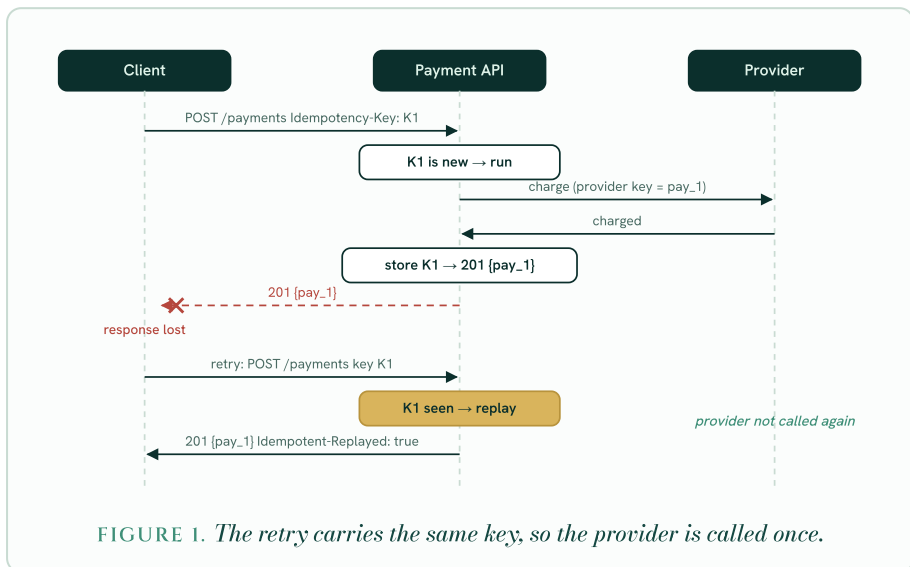
The buyer saw one spinner and one success screen. Their bank app showed two charges of USD 64. Kiosk & Co had two payments, two orders and one very annoyed customer. Every component had behaved correctly. The system as a whole had not.

The question: How do you guarantee that a customer is charged exactly once, when every layer between their phone and the bank is allowed to retry?

The mental model: a numbered ticket at the counter

Imagine a counter where you hand over a form and a numbered ticket. If you come back with the same ticket, the clerk does not process the form again. They look up the ticket and hand you the same receipt as last time. If you come back with the same ticket and a *different* form, the clerk refuses: that ticket is taken.

That ticket is an **idempotency key**. The client generates it once per attempt to do something, and sends it with every retry of that attempt. The server remembers each key, runs the operation the first time, and replays the stored answer every time after.



Networks are at-least-once. Plan for it.

There is no such thing as an exactly-once network. Every reliable system you depend on achieves reliability by retrying: mobile HTTP clients, load balancers, service meshes, message queues, your provider's webhook sender, and your own background jobs. Each of them, at some point, will send the same thing twice.

You cannot remove the retries. They are what makes the system reliable. What you can do is make the receiving side treat a repeat as the same request. That property, "doing it twice has the same effect as doing it once", is **idempotency**.

RULE OF THUMB

Exactly-once delivery is a myth. Exactly-once *effect* is engineering: at-least-once delivery plus an idempotent receiver.

Some operations are naturally idempotent. Setting a field to a value is idempotent; adding to a balance is not. `GET` is idempotent by HTTP's definition; `POST` is not. Creating a payment is the most important non-idempotent operation in your system, so you have to make it idempotent on purpose.

The client owns the key, and it means "this attempt"

The key must come from the client, because only the client knows whether a request is a retry or a new intention. The server cannot tell the difference between "I tapped Pay twice by accident" and "I want two jackets".

The scope of a key is one **attempt to do one thing**. In practice:

- Generate a random key (a UUID is fine) when the user starts a checkout attempt.
- Reuse it for every retry of that attempt, including after an app restart if you can persist it.
- Generate a new key only when the user deliberately starts a new attempt, for example after a definite failure and an explicit "try again with another card".

Do not derive the key from the order ID alone. If the first attempt definitely failed (card declined) and the customer tries a different card for the same order, a key based on the order would replay the decline forever.

COMMON TRAP

Using a key that is too broad (per customer, per order) or too narrow (per HTTP request, regenerated on every retry). Too broad blocks legitimate new attempts. Too narrow makes the key useless.

The server stores the first answer, not just the key

Remembering "I've seen K1" is not enough. If a retry arrives and the server only says "duplicate, rejected", the client still does not know whether it was charged. The server must store the **response** of the first execution and replay it exactly: the same status code, the same body, the same payment ID.

That means a stored record has four parts:

FIELD	WHY
Key (scoped per merchant or API client)	The lookup
Request fingerprint	Detects the same key reused with a different request
State: in progress or completed	Handles the retry that arrives while the first is still running
Stored response (status, body)	Replayed to every retry

The unique constraint does the hard part

Two retries can arrive at the same millisecond, on two different servers. A check like "SELECT the key; if it is not there, INSERT it" has a race between the SELECT and the INSERT. Both servers see nothing, both insert, both charge.

The database already has a tool that resolves this race correctly: a unique constraint. Let the insert itself be the check.

```
INSERT INTO idempotency_keys (merchant_id, key, fingerprint, state)
VALUES ($1, $2, $3, 'in_progress')
ON CONFLICT (merchant_id, key) DO NOTHING
RETURNING key;
```

If the insert returns a row, this request created the key and must do the work. If it returns nothing, someone else already owns the key: read the existing record and decide whether to replay, refuse or tell the client to wait. Exactly one request can ever win the insert, no matter how many servers you run.

In Code: a middleware that runs the handler at most once

The companion repository implements this as HTTP middleware, so individual handlers do not have to think about it. Here is the decision at its centre:

```

    fp := Fingerprint(r, body)

    rec, created, err := store.Begin(r.Context(), key, fp,
now())

    switch {
    case err != nil:
        http.Error(w, "idempotency store unavailable",
http.StatusServiceUnavailable)
    case !created && rec.Fingerprint != fp:
        http.Error(w, "Idempotency-Key reused with a different
request", http.StatusUnprocessableEntity)
    case !created && rec.InProgress():
        http.Error(w, "a request with this key is still in
progress", http.StatusConflict)
    case !created:
        replay(w, rec)
    default:
        run(w, r, store, key, next)
    }

```

Excerpt from `internal/idempotency/middleware.go`. There are exactly five outcomes, and each one is a decision you would otherwise make inconsistently across handlers:

1. **The store is down:** refuse with `503`. Never run the payment without idempotency "just this once".
2. **Same key, different request:** a client bug. Refuse with `422` so it shows up in their logs.
3. **Same key, first request still running:** `409`. The client should wait and retry with the same key.
4. **Same key, completed:** replay the stored response, with an `Idempotent-Replayed: true` header.

5. **New key:** run the handler and store its response.

The repository's test fires ten concurrent requests with the same key at a slow handler and asserts that the card is charged once.

Not every error should be remembered

What if the handler fails? It depends on whether anything durable happened.

If the handler returns a `4xx` (invalid card number, amount too large), the result is final for that request. Store and replay it: retrying the same bad request will not make it good.

If the handler returns a `5xx` because it could not reach the database before doing anything, nothing happened, and a retry should be allowed to try again. The companion middleware releases the key in that case:

```
func run(w http.ResponseWriter, r *http.Request, store Store, key
string, next http.Handler) {
    rec := &recorder{header: http.Header{}, code: http.StatusOK}
    next.ServeHTTP(rec, r)
    if rec.code ≥ 500 {
        store.Release(r.Context(), key) // nothing durable
        happened; let the client retry
    } else {
        store.Complete(r.Context(), key, rec.code, rec.header,
rec.body.Bytes())
    }
    copyHeader(w.Header(), rec.header)
    w.WriteHeader(rec.code)
    w.Write(rec.body.Bytes())
}
```

*From `internal/idempotency/middleware.go`. * Releasing on `5xx` is only safe if your handlers follow one rule: **never return a 5xx after a side effect may have happened**. If the provider call timed out, the charge may have happened. That is not a server error; it is a payment in `UNKNOWN` (Chapter 7), and the response should say so with a `2xx` or `202` and the payment ID, so the client can poll instead of retrying.

RULE OF THUMB

A `5xx` must mean "nothing happened, retry is safe". Anything that might have moved money returns a payment in `PENDING` or `UNKNOWN` instead.

Keys expire, payments do not

Idempotency records cannot live forever; the table would grow without bound. Pick an expiry that covers every realistic retry window: mobile clients retrying for minutes, background jobs retrying for hours, a user reopening the app the next day. Twenty-four hours is a common default; some providers keep keys for longer.

Expiry is safe only because idempotency is not your last line of defence. The payment itself carries a unique reference to the provider, the state machine refuses duplicate captures, and reconciliation catches anything that slips through. Idempotency keys stop the common case cheaply; the other layers stop the rare one.

You are also a client

Everything above describes Kiosk & Co as the **receiver** of retries. It is also a **sender**: when it calls the provider, its own HTTP client may retry after a timeout.

Most mature payment APIs accept an idempotency key or a unique merchant reference on create calls. Send one, always, and derive it from your own payment ID so that every retry of the same payment carries the same key. If a provider does not support idempotency on a call that moves money, treat every timeout on that call as `UNKNOWN` and resolve it by inquiry, never by resending.

```
Phone —key K1→ Kiosk API —key pay_1→ Provider → Bank
      (client's key)           (our payment id as the provider key)
```

AROUND THE WORLD

The same idea appears under different names. Card and PSP APIs call it an idempotency key. Bank and national-standard APIs usually call it a partner or external reference number, and reject a second request with the same reference. Instant-payment rails rely on a unique end-to-end identifier. Whatever it is called, find it in your provider's docs before you write the first line of integration code.

What goes into the fingerprint, and what doesn't

The fingerprint detects a key reused for a different request. If it is too strict, legitimate retries are rejected; if it is too loose, a genuinely different request slips through and receives someone else's stored response.

Include everything that changes the meaning of the request: the method, the path, and the body fields that define the operation (amount, currency, payment method, order reference). Exclude what can legitimately differ between retries of the same attempt: request timestamps, client-generated trace IDs, and headers added by proxies. The companion code hashes the whole body, which is the simplest correct choice as long as clients send identical bodies on retry. If your clients add a timestamp to every body, fingerprint a canonical subset of fields instead.

What a good client does with the answers

Idempotency is a contract between client and server, and the client has to keep its side. Publish the expected behaviour in your API documentation:

RESPONSE	MEANING	CLIENT SHOULD
2xx	Done, or replayed (Idempotent-Replayed: true)	Use the result; never retry
202 with a payment in PENDING / UNKNOWN	Outcome not known yet	Poll GET /payments/{id} , don't create again
409	Same key still in progress	Wait and retry with the same key
422	Key reused with a different request	Fix the bug; generate a new key only for a new intention
4xx (validation)	Request is wrong	Fix and send with a new key
5xx or network error	Nothing happened, or no answer	Retry with the same key, with backoff

The most common client bug is regenerating the key inside the retry loop, which silently turns idempotent retries into new payments. The second most common is treating 409 as a failure and showing the customer an error while the first request is about to succeed. Both are worth a line in your SDK and a test in your mobile app.

War Story: the retry that came from our own queue

Consider a payout service that had idempotency on its public API, tested and reviewed. It still sent one batch of seller payouts twice.

The public API was fine. The duplicate came from inside: a worker picked a payout job from a queue, called the bank, and crashed before acknowledging the job. The queue, correctly, redelivered it to another worker, which called the bank again with a freshly generated reference. The bank, correctly, treated it as a new transfer.

The fix was to make the bank reference a function of the payout ID, not of the attempt, so that the bank itself rejected the second transfer. Idempotency is not a feature of your API layer. It is a property every hop must have.

Trade-offs: where to enforce idempotency

APPROACH	USE WHEN	AVOID WHEN
Middleware with a key table (this chapter)	Public APIs that create payments, refunds, payouts	Read-only endpoints
Natural unique key on the business record (<code>UNIQUE (merchant_id, order_ref)</code>)	One payment per order is a real business rule	Legitimate repeat payments for the same reference
Provider-side idempotency key	Every outbound call that moves money	Never skip it if the provider supports it
Inquiry before retry	Providers without idempotency support	High-volume paths where an extra call per retry is too costly

Interview Corner

1. "How do you guarantee a customer is charged exactly once?" At-least-once delivery plus idempotent receivers. Client-generated key per attempt; unique constraint on the key; store and replay the first response; fingerprint to catch misuse; idempotency on outbound provider calls too; reconciliation as the final net.

2. "Two identical requests arrive at the same time on two servers. What happens?" Both try to insert the key; the unique constraint lets exactly one succeed. The loser reads the record: if it is still in progress, return `409` and let the client retry; if complete, replay.

3. "Your provider call timed out. Do you return an error to the client?" No. The charge may have happened. Return the payment in `UNKNOWN` (or `PENDING`) with its ID, store that response for the key, and resolve the outcome by inquiry. A `5xx` would invite a retry that the provider may treat as a new charge.

Production checklist

- ☐ Every endpoint that creates a payment, refund or payout requires an idempotency key.
- ☐ Keys are scoped per merchant or API client and enforced by a unique constraint.
- ☐ The first response is stored and replayed byte for byte; replays are marked with a header.
- ☐ A reused key with a different request body is rejected with a clear error.
- ☐ Handlers never return `5xx` after a side effect may have occurred.
- ☐ Every outbound call that moves money carries an idempotency key or unique reference derived from your own ID.
- ☐ Queue consumers derive provider references from business IDs, not from attempts.
- ☐ Key expiry is longer than every client's retry window.

TL;DR

- Every layer retries; exactly-once effect comes from idempotent receivers.

- The client generates one key per attempt; the server stores the first response and replays it.
- A unique constraint, not a SELECT-then-INSERT, decides who runs the operation.
- `5xx` must mean "nothing happened". Anything that may have moved money is `UNKNOWN`, not an error.
- You are also a client: send idempotency keys to your provider, derived from your own IDs.

Next: your API is now safe against retries from your clients. But the provider talks back too, through webhooks, and they arrive early, late, twice, and sometimes from someone who is not your provider at all.