Great instinct. Do **silent retries**—but only for **transient** errors, and make writes **idempotent** so retries can’t double-apply.

**1) Classify errors fast**

Retry only when it *might* succeed next try:

**Good to retry (transient):**

* Node/network: ETIMEDOUT, ECONNRESET, EPIPE, ENETUNREACH
* PG connection failures: SQLSTATE 08006, 08003, 57P01 (server restarting)
* Contention: 40001 (serialization failure), 40P01 (deadlock)
* Pool hiccups: 53300 (too many connections)—short backoff

**Do NOT retry (client bugs):**

* Validation: 22P02 (bad UUID), 23502 (NOT NULL), 42601 (syntax)
* Business conflicts: 23505 (unique violation) ⇒ handle logically, don’t blind-retry

**2) Make writes idempotent**

So a retried request can’t create duplicates:

* Use **UPSERT** (INSERT ... ON CONFLICT DO NOTHING/UPDATE)
* Guard with **UNIQUE** keys (you already have uniq\_att\_user\_course\_day)
* For “request-level” dedupe, accept a **request\_id** and store it in a table with UNIQUE(request\_id)

**3) A tiny, reusable retry wrapper**

Works with both reads and idempotent writes.

// lib/withPgRetry.js

const TRANSIENT\_CODES = new Set([

'08006','08003','57P01','40001','40P01','53300'

]);

function isTransient(err) {

return TRANSIENT\_CODES.has(err.code)

|| ['ETIMEDOUT','ECONNRESET','EPIPE','ENETUNREACH'].includes(err.code);

}

export async function withPgRetry(fn, {

attempts = 4, // 1 try + 3 retries

baseMs = 100, // backoff base

maxMs = 1500, // cap

jitter = true,

onRetry = (err, i, waitMs) => {}

} = {}) {

let i = 0;

for (;;) {

try {

return await fn();

} catch (err) {

i++;

if (i >= attempts || !isTransient(err)) throw err;

const wait = Math.min(maxMs, baseMs \* 2 \*\* (i - 1));

const waitMs = jitter ? Math.floor(wait \* (0.5 + Math.random())) : wait;

onRetry(err, i, waitMs);

await new Promise(r => setTimeout(r, waitMs));

}

}

}

**Use it (read)**

import { pool } from '../db.js';

import { withPgRetry } from '../lib/withPgRetry.js';

export async function getUserById(userId) {

return withPgRetry(async () => {

const { rows } = await pool.query(`

SELECT user\_id, first\_name, last\_name, phone, email, sign\_in\_times, created\_at

FROM users WHERE user\_id = $1::uuid LIMIT 1

`, [userId]);

return rows[0] ?? null;

}, { onRetry: (e,i,ms) => console.warn('getUserById retry', {i,ms,code:e.code}) });

}

**Use it (idempotent write)**

export async function insertAttendanceIfNew(userId, courseId, date) {

const d = (date ?? new Date()).toISOString().slice(0,10);

return withPgRetry(async () => {

await pool.query(`

INSERT INTO attendance (user\_id, course\_id, signed\_at)

VALUES ($1, $2, $3::date)

ON CONFLICT ON CONSTRAINT uniq\_att\_user\_course\_day DO NOTHING

`, [userId, courseId, d]);

});

}

**4) Timeouts (fail fast to allow retry)**

* Set PG **statement timeout** per session: SET statement\_timeout = '3s' (or via pool/session).
* In pg, you can also use query\_timeout on the client.

**5) Route-level “retry shield”**

Wrap *only* the DB call, not the whole request, so the user doesn’t feel delay if the first try succeeds. Keep total retry budget small (e.g., <2–3s).

**6) Logging & metrics**

Log **first failure** and each retry with code + wait ms. Alert on spikes.

**7) Where it helps in your app**

* **fastCheckInByDevice / reauthCheckInByPhone**: wrap the attendance insert + user fetches. Safe due to UNIQUE.
* **touchDeviceLastSeen**: idempotent UPDATE → ok to retry.
* **createUser**: if you might re-submit, either:
  + Use a **request\_id** UNIQUE to prevent dup users, or
  + Catch 23505 (phone/email taken) and return existing.
* **mergeUsers**: if done in a transaction, retries should target transient *begin/commit* failures; don’t naively re-run half-completed merges—wrap whole op in one withPgRetry + transaction.

**8) Optional: circuit breaker (later)**

If the DB is really down, stop hammering it for a short window and fail fast. Probably overkill now.

If you want, I can drop withPgRetry into your codebase and tag which functions should use it vs. which should just handle 23505/validation without retries.