EcomLabs

STEP-BY-STEP GUIDE

A Postgres job queue with SKIP LOCKED

The jobs table, a function that enqueues inside the order's transaction, and a Node worker with FOR UPDATE SKIP LOCKED, exponential backoff retries and stuck-job recovery. All of it tested, with a checklist for production.

FOR	Node.js apps on PostgreSQL that send emails or handle webhooks
TESTED WITH	Node 22.22.3 · PGlite 0.5.10 (PostgreSQL 18.6) · Postgres.js 3.4.9 · Windows 11
UPDATED	November 3, 2026
DOCS	postgresql.org/docs/current/sql-select.html
ARTICLE	ecomlabs.dev/g/colas-en
AUTHOR	Luis Morales · @luismoralesdev

Contents

Before you start	What you will build
Step 01	The jobs table
Step 02	Enqueue in the order's transaction
Step 03	The worker
Step 04	Start the worker
Step 05	What we tested
Final	Checklist

BEFORE YOU START

What you will build

A `jobs` table, a function that enqueues in the order's transaction, and a Node worker that claims each job with `FOR UPDATE SKIP LOCKED`, retries it and recovers stuck ones. We tested it on November 2, 2026 with Node 22.22.3, Postgres.js 3.4.9 (`npm install postgres`) and PGlite 0.5.10, which is PostgreSQL 18.6 in WebAssembly.

STEP 01

The jobs table

Save this as `schema.sql` and run it against your database (or turn it into a migration):

```
CREATE TABLE jobs (
  id           bigint      GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  type         text        NOT NULL,
  payload      jsonb       NOT NULL,
  status       text        NOT NULL DEFAULT 'pending'
               CHECK (status IN ('pending', 'running', 'done', 'failed')),
  attempts     int         NOT NULL DEFAULT 0,
  max_attempts int         NOT NULL DEFAULT 5,
  run_at       timestamptz NOT NULL DEFAULT now(),
  locked_at    timestamptz,
  last_error   text,
  created_at   timestamptz NOT NULL DEFAULT now(),
  updated_at   timestamptz NOT NULL DEFAULT now()
);

CREATE INDEX jobs_pending_run_at ON jobs (run_at) WHERE status = 'pending';
```

PostgreSQL only uses the partial index when the query's `WHERE` implies its predicate, so the worker asks for `status = 'pending'` exactly.

STEP 02

Enqueue in the order's transaction

Save `enqueue.mjs`:

```
export async function enqueue(sql, type, payload) {
  const [job] = await sql`
    INSERT INTO jobs (type, payload)
    VALUES (${type}, ${sql.json(payload)})
    RETURNING id`;
  return job.id;
}
```

Call it with the `tx` of the transaction that creates the order, so both commit together or neither does:

```
async function placeOrder(email) {
  return sql.begin(async (tx) => {
    const [order] = await tx`INSERT INTO orders (email) VALUES (${email}) RETURNING id`;
    await enqueue(tx, "order-confirmation", { orderId: order.id, email });
    return order.id;
  });
}
```

Use `sql.json(payload)`: with `JSON.stringify`, our `jsonb` column stored a string.

STEP 03

The worker

Save the whole `worker.mjs`:

```
export async function claimJob(sql) {
  const [job] = await sql`
    UPDATE jobs
    SET status = 'running', locked_at = now(), attempts = attempts + 1, updated_at = now()
    WHERE id = (
      SELECT id FROM jobs
      WHERE status = 'pending' AND run_at <= now()
      ORDER BY run_at
      LIMIT 1
      FOR UPDATE SKIP LOCKED
    )
    RETURNING *`;
  return job ?? null;
}

export async function runOne(sql, handlers, { backoffSeconds = 30 } = {}) {
  const job = await claimJob(sql);
  if (!job) return null;
  try {
    const handler = handlers[job.type];
    if (!handler) throw new Error(`no handler for job type ${job.type}`);
    await handler(job.payload, job);
    await sql`
      UPDATE jobs SET status = 'done', locked_at = NULL, updated_at = now()
      WHERE id = ${job.id} AND attempts = ${job.attempts}`;
    return job;
  } catch (err) {
    const status = job.attempts >= job.max_attempts ? "failed" : "pending";
    const delay = backoffSeconds * 2 ** (job.attempts - 1);
    await sql`
      UPDATE jobs SET status = ${status}, locked_at = NULL, last_error = ${String(err?.message ?? err)},
        run_at = now() + make_interval(secs => ${delay}), updated_at = now()
```

```
      WHERE id = ${job.id} AND attempts = ${job.attempts}`;
    return job;
  }
}

export async function recoverStuck(sql, minutes = 15) {
  return sql`
    UPDATE jobs
    SET status = CASE WHEN attempts >= max_attempts THEN 'failed' ELSE 'pending' END,
        locked_at = NULL, last_error = 'stuck in running, lock expired', updated_at = now()
    WHERE status = 'running' AND locked_at < now() - make_interval(mins => ${minutes})
    RETURNING id, status`;
}

const sleep = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

export async function work(sql, handlers, { idleMs = 1000, signal } = {}) {
  while (!signal?.aborted) {
    if (!(await runOne(sql, handlers))) await sleep(idleMs);
  }
}
```

- `claimJob` claims a job in one statement; `SKIP LOCKED` skips rows another transaction has already locked.
- `runOne` retries after 30, 60, 120 and 240 s and leaves the fifth failure `failed`. `AND attempts = ...` stops a late worker from overwriting another.
- `recoverStuck` frees jobs that have been `running` for more than 15 minutes.

STEP 04

Start the worker

Save `start.mjs` and swap the fake sender for your email provider's client:

```
import postgres from "postgres";
import { work, recoverStuck } from "./worker.mjs";

// Fake sender: replace it with your email provider's client.
async function sendOrderEmail({ orderId, email }) {
  await new Promise((resolve) => setTimeout(resolve, 300));
  console.log(`email sent to ${email} for order ${orderId}`);
}

const sql = postgres(process.env.DATABASE_URL, { max: 1 });
const handlers = { "order-confirmation": sendOrderEmail };

setInterval(() => recoverStuck(sql).catch(console.error), 60_000);
await work(sql, handlers);
```

Run it apart from your web server, with `DATABASE_URL` pointing at the same database: `node start.mjs`. Each process is one worker with one connection.

STEP 05

What we tested

With `node --test` against PGlite, all 8 tests passed:

- 4 workers, 50 jobs: 50 emails. Claiming in two statements (`SELECT`, then `UPDATE`): 200.
- A handler that always fails: waits of 30, 60, 120 and 240 s, then `failed`.
- A stuck job is recovered and the late worker cannot overwrite it; without the `attempts` guard, the test fails.
- A rolled-back transaction leaves no job, and the query uses the partial index.

> **CAUTION**
> PGlite runs one query at a time and never overlaps two transactions. The tests prove every job is processed once, not that `SKIP LOCKED` avoids waiting; that comes from the PostgreSQL documentation. Repeat the concurrency test on a real PostgreSQL, never on your production database.

FINAL

Checklist

- [ ] The request stores the job in the same transaction as the order and responds.
- [ ] The table has a `CHECK` on `status` and the partial index on `pending`.
- [ ] The worker claims each job in one statement with `FOR UPDATE SKIP LOCKED`.
- [ ] The email is sent outside any transaction, and failures end in `failed`.
- [ ] `recoverStuck` runs every minute, with a limit longer than the slowest job.
- [ ] Final writes carry `AND attempts = ...` and handlers are idempotent.
- [ ] Old `done` jobs are removed with a periodic `DELETE`.