EcomLabs

STEP-BY-STEP GUIDE

Expo push notifications that arrive

The four places an Expo push notification gets lost, turned into a checklist, and the server code that sends, stores the tickets, reads the delivery receipts and disables tokens that no longer work. With the Postgres tables.

FOR	Expo apps (SDK 58) with a Node server that sends push notifications
TESTED WITH	Node 22.22.3 · expo-server-sdk 8.0.0 against a local copy of the Expo push API · PGlite (Postgres 18.6) · tsc --strict
UPDATED	November 19, 2026
DOCS	docs.expo.dev/push-notifications/sending-notifications
ARTICLE	ecomlabs.dev/g/push-en
AUTHOR	Luis Morales · @luismoralesdev

Contents

Before you start	The four failure points
Step 01	The tables
Step 02	Send and read the receipts
Step 03	The Postgres store
Step 04	Schedule the check

BEFORE YOU START

The four failure points

A notification goes from your server to the Expo push service, then to FCM (Android) or APNs (iOS), then to the phone. Check the four points in this order; each one has its own signal.

- [ ] **1. Permission.** Ask after a user action, not on launch: on iOS the prompt shows only once. On Android 13+ the channel must exist before the permission and the token, or the prompt never appears. If `canAskAgain` is false, send the user to settings.
- [ ] **2. Token.** It comes from `getExpoPushTokenAsync({ projectId })` and is saved on every launch with permission and whenever `addPushTokenListener` fires. On the server the token is the key: one user can have several.
- [ ] **3. Sending.** Up to 100 messages per request and 600 per second per project. Each message returns a ticket; an "ok" ticket only says Expo received it. Retry 429s, 5xx and network errors with growing waits; not 400s.
- [ ] **4. Delivery.** Ask for receipts about 15 minutes later, up to 1,000 IDs per request; they are cleared after 24 hours. An "ok" receipt says FCM or APNs accepted it, not that the phone showed it. `DeviceNotRegistered` disables the token.

Test on a development build (on Android, Expo Go has no push since SDK 53) and, before you ship, on a real phone with the app open, in the background and closed.

STEP 01

The tables

One row per device and one per pending ticket. Tested on PGlite (Postgres 18.6):

```
CREATE TABLE push_tokens (
  token       text PRIMARY KEY,
  user_id     bigint NOT NULL,
  updated_at  timestamptz NOT NULL DEFAULT now(),
  disabled_at timestamptz
);

CREATE TABLE push_pending (
  ticket_id text PRIMARY KEY,
  token     text NOT NULL REFERENCES push_tokens (token),
  sent_at   timestamptz NOT NULL DEFAULT now()
);

-- saveToken on the server
INSERT INTO push_tokens (token, user_id) VALUES ($1, $2)
ON CONFLICT (token) DO UPDATE
SET user_id = EXCLUDED.user_id, updated_at = now(), disabled_at = NULL;
```

The `ON CONFLICT` covers two cases: the same phone signing in to another account (the row changes owner) and a device registering again after a `DeviceNotRegistered` (`disabled_at` is cleared). To send to a user: `SELECT token FROM push_tokens WHERE user_id = $1 AND disabled_at IS NULL`. On sign-out, delete or detach their token.

STEP 02

Send and read the receipts

With `expo-server-sdk` 8.0.0 (Node 22.12 or later). `sendPush` stores "ok" tickets as pending; `checkReceipts` asks for them once 15 minutes have passed:

```
// push.ts: send with expo-server-sdk, keep the tickets, check the receipts later.
import { Expo, type ExpoPushMessage } from "expo-server-sdk";

export type Pending = { id: string; token: string; sentAt: number };

export interface PushStore {
  savePending(rows: Pending[]): Promise<void>;
  pendingSentBefore(time: number): Promise<Pending[]>;
  removePending(ids: string[]): Promise<void>;
  disableToken(token: string): Promise<void>;
}

const WAIT = 15 * 60 * 1000; // Expo: check receipts about 15 minutes after sending
const KEEP = 24 * 60 * 60 * 1000; // Expo clears receipts after 24 hours

export async function sendPush(expo: Expo, store: PushStore, messages: (ExpoPushMessage & { to: string })[]) {
  const retry: ExpoPushMessage[] = [];
  const valid = messages.filter((m) => Expo.isExpoPushToken(m.to));
  for (const chunk of expo.chunkPushNotifications(valid)) {
    let tickets;
    try {
      tickets = await expo.sendPushNotificationsAsync(chunk);
    } catch (e) {
      const status = (e as { statusCode?: number }).statusCode;
      if (status !== undefined && status !== 429 && status < 500) throw e; // a 400 will not fix itself
      retry.push(...chunk); // 429, 5xx or network: try again later, with backoff
      continue;
    }
    const pending: Pending[] = [];
    for (const [i, ticket] of tickets.entries()) {
      const token = chunk[i].to as string; // ticket i answers message i
      if (ticket.status === "ok") pending.push({ id: ticket.id, token, sentAt: Date.now() });
      else if (ticket.details?.error === "DeviceNotRegistered") await store.disableToken(token);
      else console.error("push ticket", ticket.details?.error, ticket.message);
    }
    await store.savePending(pending);
  }
  return { skipped: messages.length - valid.length, retry };
}
```

```
export async function checkReceipts(expo: Expo, store: PushStore, now = Date.now()) {
  const pending = await store.pendingSentBefore(now - WAIT);
  const byId = new Map(pending.map((p) => [p.id, p]));
  const done: string[] = [];
  const count = { ok: 0, disabled: 0, errors: 0, waiting: 0, expired: 0 };
  for (const ids of expo.chunkPushNotificationReceiptIds([...byId.keys()])) {
    const receipts = await expo.getPushNotificationReceiptsAsync(ids);
    for (const id of ids) {
      const receipt = receipts[id];
      const p = byId.get(id)!;
      if (!receipt) {
        if (now - p.sentAt > KEEP) { done.push(id); count.expired++; } else count.waiting++;
        continue;
      }
      done.push(id);
      if (receipt.status === "ok") { count.ok++; continue; }
      if (receipt.details?.error === "DeviceNotRegistered") {
        await store.disableToken(p.token);
        count.disabled++;
      } else {
        console.error("push receipt", id, receipt.details?.error, receipt.message);
        count.errors++;
      }
    }
  }
  await store.removePending(done);
  return count;
}
```

Ticket `i` answers message `i`, so each message carries a single token. Whatever comes back in `retry` is sent again later; the SDK only retries 429s, twice. A missing ID is a receipt that isn't ready; after 24 hours it is dropped.

STEP 03

The Postgres store

The four `PushStore` operations on the tables from step 01. `query` is your client's function, for example `pool.query` from `pg`:

```
// store.ts: PushStore on Postgres. `query` is your client's query(sql, params),
// for example pool.query from pg.
import type { Pending, PushStore } from "./push";

type Query = (sql: string, params?: unknown[]) => Promise<{ rows: Record<string, unknown>[] }>;

export function pgStore(query: Query): PushStore {
  return {
    async savePending(rows: Pending[]) {
      if (!rows.length) return;
```

```
      await query(
        `INSERT INTO push_pending (ticket_id, token, sent_at)
         SELECT id, token, to_timestamp(ms / 1000.0)
         FROM unnest($1::text[], $2::text[], $3::float8[]) AS t(id, token, ms)`,
        [rows.map((r) => r.id), rows.map((r) => r.token), rows.map((r) => r.sentAt)],
      );
    },
    async pendingSentBefore(time: number) {
      const { rows } = await query(
        `SELECT ticket_id, token, extract(epoch FROM sent_at) * 1000 AS ms
         FROM push_pending WHERE sent_at < to_timestamp($1 / 1000.0)`,
        [time],
      );
      return rows.map((r) => ({ id: String(r.ticket_id), token: String(r.token), sentAt: Number(r.ms) }));
    },
    async removePending(ids: string[]) {
      if (ids.length) await query("DELETE FROM push_pending WHERE ticket_id = ANY($1::text[])", [ids]);
    },
    async disableToken(token: string) {
      await query("UPDATE push_tokens SET disabled_at = now() WHERE token = $1 AND disabled_at IS NULL",
[token]);
    },
  };
}
```

A group's tickets are saved in one query with `unnest`. `disableToken` doesn't delete the row: `push_pending` references it, and the `ON CONFLICT` re-enables it if the device comes back.

STEP 04

Schedule the check

Run `checkReceipts(expo, pgStore(pool.query.bind(pool)))` every 5 minutes from your scheduler or job queue, never inside the request that sends. Look at what it returns:

- `ok` and `disabled`: nothing to do; dead tokens are already disabled.
- `errors`: alert. `InvalidCredentials` is fixed with `eas credentials`; `MessageTooBig` is a payload over 4,096 bytes; `MessageRateExceeded` means space out sends to that device; `MismatchSenderId` is an FCM sender that doesn't match.
- `waiting`: asked for on the next run. If `expired` grows, your job is not running on time.

> **CAUTION**
>
> Tested against a local copy of the Expo push API, never the real service, through the `EXPO_BASE_URL` variable that Expo documents as internal. Don't use it in production.