EcomLabs

GUÍA PASO A PASO

Notificaciones push con Expo que sí llegan

Los cuatro puntos donde se pierde una notificación push con Expo, convertidos en checklist, y el código del servidor que envía, guarda los tickets, lee los recibos de entrega y desactiva los tokens que ya no sirven. Con las tablas de Postgres.

PARA	Apps con Expo (SDK 58) y un servidor en Node que envía notificaciones push
PROBADO CON	Node 22.22.3 · expo-server-sdk 8.0.0 contra una copia local de la API de Expo · PGlite (Postgres 18.6) · tsc --strict
ACTUALIZADA	19 de noviembre de 2026
DOCUMENTACIÓN	docs.expo.dev/push-notifications/sending-notifications
ARTÍCULO	ecomlabs.dev/g/push
AUTOR	Luis Morales · @luismoralesdev

Índice

Antes de empezar	Los cuatro puntos de falla
Paso 01	Las tablas
Paso 02	Enviar y leer los recibos
Paso 03	El store en Postgres
Paso 04	Programar la revisión

ANTES DE EMPEZAR

Los cuatro puntos de falla

Una notificación va de tu servidor al servicio de Expo, de ahí a FCM (Android) o APNs (iOS) y luego al teléfono. Revisa los cuatro puntos en este orden; cada uno tiene su propia señal.

- [ ] **1. Permiso.** Se pide después de una acción del usuario, no al abrir la app: en iOS el aviso sale una sola vez. En Android 13+ el canal se crea antes del permiso y del token, o el aviso no aparece. Si `canAskAgain` es falso, manda al usuario a los ajustes.
- [ ] **2. Token.** Sale de `getExpoPushTokenAsync({ projectId })`, se guarda en cada arranque con permiso y cuando `addPushTokenListener` avisa. En el servidor, el token es la llave: un usuario puede tener varios.
- [ ] **3. Envío.** Hasta 100 mensajes por petición y 600 por segundo por proyecto. Cada mensaje devuelve un ticket; un ticket "ok" solo dice que Expo lo recibió. Reintenta 429, 5xx y errores de red con espera creciente; un 400 no.
- [ ] **4. Entrega.** Los recibos se piden unos 15 minutos después, hasta 1,000 IDs por consulta, y se borran a las 24 horas. Un recibo "ok" dice que FCM o APNs lo aceptaron, no que el teléfono lo mostró. `DeviceNotRegistered` desactiva el token.

Prueba en un development build (en Android, Expo Go no tiene push desde el SDK 53) y, antes de publicar, en un teléfono real con la app abierta, en segundo plano y cerrada.

PASO 01

Las tablas

Una fila por dispositivo y una por ticket pendiente. Probado con 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;
```

El `ON CONFLICT` cubre dos casos: el mismo teléfono con otra cuenta (la fila cambia de dueño) y un dispositivo que vuelve a registrarse después de un `DeviceNotRegistered` (se borra `disabled_at`). Para enviar a un usuario: `SELECT token FROM push_tokens WHERE user_id = $1 AND disabled_at IS NULL`. Al cerrar sesión, borra o desasocia su token.

PASO 02

Enviar y leer los recibos

Con `expo-server-sdk` 8.0.0 (Node 22.12 o posterior). `sendPush` guarda los tickets "ok" como pendientes; `checkReceipts` los consulta cuando ya pasaron 15 minutos:

```
// 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;
}
```

El ticket `i` corresponde al mensaje `i`, por eso cada mensaje lleva un solo token. Lo que vuelve en `retry` se reintenta más tarde; el SDK solo reintenta los 429, dos veces. Un ID ausente es un recibo que aún no está listo; a las 24 horas se descarta.

PASO 03

El store en Postgres

Las cuatro operaciones de `PushStore` sobre las tablas del paso 01. `query` es la función de tu cliente, por ejemplo `pool.query` de `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]);
    },
  };
}
```

Los tickets de un grupo se guardan en una sola consulta con `unnest`. `disableToken` no borra la fila: `push_pending` la referencia y el `ON CONFLICT` la reactiva si el dispositivo vuelve.

PASO 04

Programar la revisión

Corre `checkReceipts(expo, pgStore(pool.query.bind(pool)))` cada 5 minutos desde tu programador o tu cola de trabajos, nunca dentro de la petición que envía. Revisa lo que devuelve:

- `ok` y `disabled`: nada que hacer; los tokens muertos ya están desactivados.
- `errors`: alerta. `InvalidCredentials` se arregla con `eas credentials`; `MessageTooBig` es un payload de más de 4,096 bytes; `MessageRateExceeded` pide espaciar los envíos a ese equipo; `MismatchSenderId` es un remitente de FCM que no coincide.
- `waiting`: se consultan en la siguiente vuelta. Si `expired` crece, tu trabajo no corre a tiempo.

> **CUIDADO**
>
> Probado contra una copia local de la API de Expo, nunca contra el servicio real, con la variable `EXPO_BASE_URL` que Expo documenta como uso interno. No la uses en producción.