

Pawsi Development and Security of a Production Application

Diego Puzio · Case Study · October 2026

I developed Pawsi to help people post pets for adoption, report missing pets, and share sightings. It includes a web client and an Android application built with Capacitor. During a security review, I prepared a local recovery environment and corrected two issues in the messaging system: trust in notification requests and excessive update permissions.

My work covered development, deployment, backend review, testing, and documentation. This case describes the changes and the evidence supporting them. Its scope is limited; it is not a comprehensive audit or a security certification.

From Prototype to Product

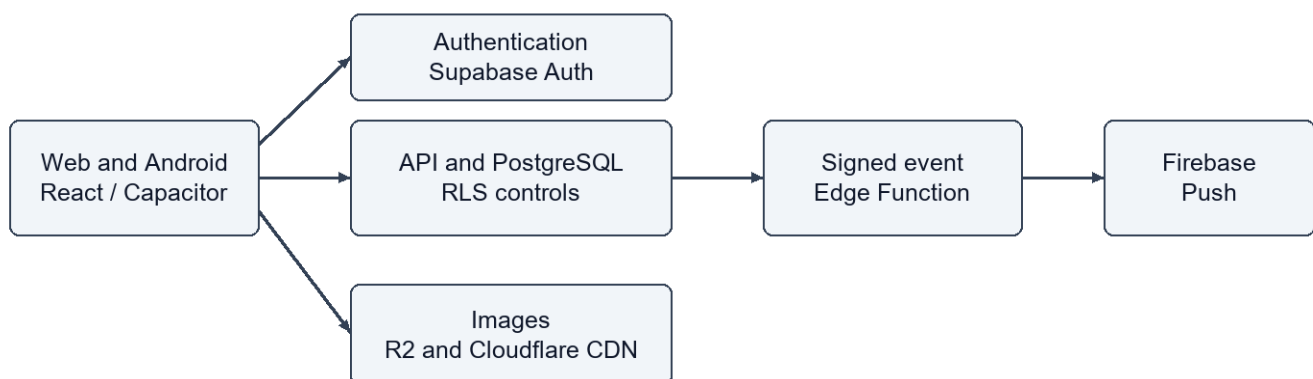
Pawsi launched as a web application in September 2025 and won the Hackathon del Milagro in Salta, Argentina, with its web MVP. I continued development and published the Capacitor-based Android application on Google Play in October 2025. The project received local media coverage, and I took part in more than 30 interviews to present it. The application remains operational, although activity declined after its initial user peak.

This involved building the product, publishing it, maintaining its operation, and explaining its purpose to nontechnical audiences. The security review grew out of that maintenance responsibility.

Architecture and Decisions

The interface uses React and TypeScript. Capacitor enables Android distribution. Supabase provides authentication, PostgreSQL, row-level security policies, and Edge Functions. Images are served through R2 object storage and Cloudflare's CDN. Firebase Cloud Messaging delivers push notifications.

Image storage and the database require separate recovery procedures: restoring records does not automatically recover the files. The media domain's cache policy sets a one-year TTL at both Cloudflare and the browser. Replacing or removing images therefore requires considering cached copies; purging the CDN does not clear browser caches.



Simplified service architecture for this case study.

A Lab Before Production Changes

I set up local Supabase using Docker and WSL. The project groups several containers, including PostgreSQL, authentication, API, storage, and the administration interface. The frontend used the local environment's URL and credentials. External integrations were disabled or replaced to prevent deliveries and operations against production during testing.

The initial startup showed that the available migrations could not rebuild an empty database by themselves: one migration tried to alter a table that did not yet exist. For the exercise, I selectively restored the PostgreSQL backup and compared table and record counts. I also checked that later tests left no synthetic data in the restored database.

The exercise validated partial recovery of schema and data. It did not include stored files, active sessions, or all provider configuration. It therefore does not demonstrate complete recovery or establish measured RTO or RPO values.

Securing Notification Delivery

The previous flow allowed the client to request a notification send. I moved that decision to the backend: a valid insert into the messages table creates the event that authorizes delivery. The recipient comes from that event, rather than an arbitrary identity submitted to the endpoint.

The current flow works as follows:

1. The authenticated client saves a message subject to database access controls.
2. An insert trigger creates an in-app notification and records a private event.
3. The database signs the request using HMAC and a timestamp. The shared secret is stored in Vault and in the function's private configuration.
4. The Edge Function checks the signature and freshness. An atomic operation allows the event to be claimed only once.
5. When delivery is allowed, the function calls Firebase and records the outcome.

The previous function remains as a compatibility endpoint without sending notifications. This preserved support for published clients without requiring an Android update. Read-status updates do not generate new notifications.

I added rate limits, event expiration, and duplicate prevention. Enabling delivery does not process historical messages. Events exceeding the limits are omitted, rather than accumulating in a queue that could later release a burst of notifications.

The lab ran 23 checks using local Auth, PostgreSQL, pg_net, and Edge Runtime, with a simulated push provider. Scenarios included invalid or expired signatures, repeated requests, concurrency, rate limits, and configuration or provider failures.

During deployment, the function returned 503 because its secret had a different name from the one expected by the code. I corrected the configuration and confirmed delivery on Android. An unsigned request received 401. These production checks complement the local tests; they do not repeat the full test suite in production.

I chose a single delivery attempt per event, without automatic retries. This limits duplication but can lose a push if the process fails after claiming the event. The in-app notification and the push are separate mechanisms.

Restricting Message Update Permissions

The RLS policy identified the recipient as the user allowed to update a message. However, SQL UPDATE privileges covered every column. The intended action was marking a message as read, but a recipient could also change its subject or other fields.

I reproduced a subject change using synthetic accounts in the lab. I then revoked broad update privileges and granted the authenticated role UPDATE on read_at only. I retained RLS to determine which rows each user may modify.

The controls serve two purposes: RLS restricts accessible messages, while column privileges restrict what can change within them. The frontend already updated only read_at, so the fix required no frontend changes.

The 15 local checks covered the initial reproduction, legitimate read marking, blocking the other nine columns, attempts by the sender and a third party, anonymous access, and the absence of new events when marking a message as read. In production, I checked effective permissions and persistent read status between two accounts. I did not alter real messages to demonstrate abuse.

Results and Scope

Local recovery. A selective restore and record-count comparison verified partial database recovery. They do not demonstrate recovery of stored files or every integration.

Notifications. Evidence includes 23 local checks, Android delivery, and rejection of an unsigned request in production. The lab used a simulated provider; these results do not certify the entire system.

Message permissions. Evidence includes 15 local checks, effective privileges, and legitimate read marking in production. The correction specifically addresses message updates.

The review helped me turn implicit product rules into backend controls and verifiable checks. It also reinforced a recovery lesson: having a backup and demonstrating what it can restore are different tasks.

The public material is limited to this scope. Operational logs, backups, credentials, and user information remain private.

Press and Contact

- [InSalta — A Salta-built app reuniting lost pets with their owners](#)
- [Qué Pasa Salta — Pawsi and missing pets in Salta](#)
- [Agenda Salta — A local app helping people find missing pets](#)
- [Diego Puzio on LinkedIn](#)

These Spanish-language articles provide public context about the product; they do not verify the technical results of this review. The operational code and backups remain private.