UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Diseño e Implementación del Backend para un Sistema de Captura de Datos en Campo en un Ingenio Azucarero

Trabajo de graduación en modalidad de Trabajo Profesional presentado por Daniel Armando Valdez Reyes para optar al grado académico de Licenciado en Ingeniería Ciencia de la Computación y Tecnologías de la Información

Guatemala,

2025

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Diseño e Implementación del Backend para un Sistema de Captura de Datos en Campo en un Ingenio Azucarero

Trabajo de graduación en modalidad de Trabajo Profesional presentado por Daniel Armando Valdez Reyes para optar al grado académico de Licenciado en Ingeniería Ciencia de la Computación y Tecnologías de la Información

Guatemala,

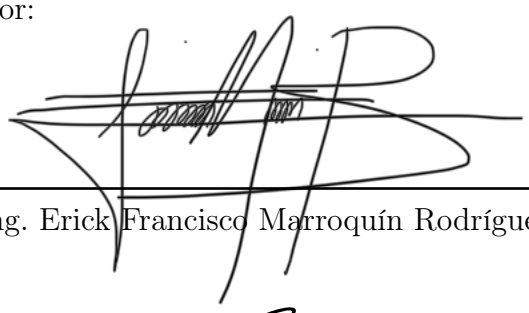
2025

Vo.Bo.:

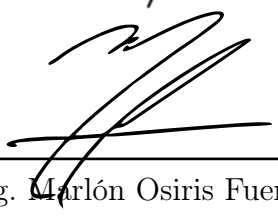
(f) 

Ing. Erick Francisco Marroquín Rodríguez

Tribunal Examinador:

(f) 

Ing. Erick Francisco Marroquín Rodríguez

(f) 

Ing. Marlón Osiris Fuentes López

Fecha de aprobación: Guatemala, 22 de noviembre de 2025.

Prefacio

Este trabajo nace de una necesidad clara del Ingenio: capturar y resguardar datos de campo de manera segura, confiable y operable sin conexión, reduciendo errores y evitando la dependencia de terceros. Está pensado para equipos de Operaciones y TI que coordinan tareas en terreno, para personal de campo que depende de formularios en jornadas largas y con conectividad intermitente, y para lectores técnicos que buscan una ruta práctica para implementar arquitecturas *offline-first*. A partir de esa necesidad se eligieron decisiones que priorizan continuidad operativa y autonomía tecnológica: en el dispositivo, una aplicación en React Native con persistencia local en SQLite que valida y guarda cada captura en el momento; en el servidor, un backend en NestJS con PostgreSQL y estructuras flexibles (JSONB) que permiten evolucionar formularios sin perder propiedades ACID, además de una sincronización idempotente que evita duplicados y contratos documentados que facilitan mantenimiento y auditoría.

Quien lea estas páginas encontrará la justificación del problema y una arquitectura orientada a portabilidad y bajo acoplamiento, junto con un modelo versionado de formularios, mecanismos de sincronización y control de calidad, y un plan de pruebas que cubre tanto la API como la verificación en dispositivo (almacenamiento local y estado de sesión). Deliberadamente quedan fuera, por ahora, un editor visual completo de formularios, tableros analíticos avanzados y un componente de gestión de dispositivos; se señalan como líneas de evolución natural sobre la misma base técnica.

El alcance práctico apunta a asegurar continuidad *offline*, trazabilidad y control local de la información, disminución de errores de transcripción y una plataforma lista para crecer con nuevas necesidades del Ingenio. Las limitaciones responden al contexto: pruebas móviles sin bibliotecas nativas adicionales, validaciones en escenarios controlados y supuestos realistas de trabajo en campo—conectividad variable y dispositivos heterogéneos—que guían el diseño y la interpretación de resultados. A quienes acompañaron el refinamiento de esta propuesta y al equipo del Ingenio que confió en el enfoque, gracias por el apoyo y la apertura para convertir una necesidad operativa en una solución concreta y escalable.

Índice

Lista de figuras

Lista de cuadros

Lista de abreviaturas y siglas

ACID Atomicidad, Consistencia, Aislamiento y Durabilidad. 20, 21, 23

API Interfaz de Programación de Aplicaciones. IX, XI, 38, 39, 44, 46–48, 50–54, 57

CI/CD Integración Continua / Despliegue Continuo. XI, 34, 45

DTO Objeto de Transferencia de Datos. 16, 36, 48, 54

E2E *End-to-End*. VIII, IX, 34, 46–48, 52, 54, 57

JWT *JSON Web Token*. VIII, 17, 30–32

MVCC Control de Concurrencia Multiversión. 21

OAuth 2.0 *Open Authorization* 2.0. 17, 30

ORM *Object-Relational Mapping*. 47

PWA Aplicación Web Progresiva. VII, 13, 15

REST Transferencia de Estado Representacional. 12, 15, 17

RPS Solicitudes por Segundo. 32–34

VU Usuario Virtual. XIII, 52, 53, 55

WAL *Write-Ahead Logging*. 21, 25

Resumen

Este trabajo diseña e implementa una solución de captura de datos en campo para un ingenio azucarero, con énfasis en operar de forma confiable bajo conectividad intermitente. El objetivo central es habilitar formularios adaptables gestionados por un backend móvil, garantizando integridad, consistencia y disponibilidad de la información recolectada.

La arquitectura propuesta sigue un enfoque *offline-first*: en el cliente se desarrolló una aplicación React Native con TypeScript y persistencia local en SQLite, responsable de validar, normalizar y almacenar de inmediato cada sesión de formulario; en el servidor, un backend NestJS con PostgreSQL expone contratos documentados en Swagger y consolida las capturas utilizando JSONB para mantener flexibilidad estructural sin sacrificar propiedades ACID. La sincronización se realiza mediante envíos idempotentes que convergen el estado local y el remoto.

Metodológicamente, el desarrollo se alinea con criterios de elección tecnológica orientados a portabilidad y bajo acoplamiento; se definen supuestos y restricciones operativas propias del contexto rural, y se instrumenta un plan de pruebas que incluye niveles de API (cobertura unitaria y de controladores, métricas globales y por endpoint) y verificación en dispositivo sobre SQLite y el estado de sesión con Redux, con evidencia reproducible.

Los resultados muestran viabilidad técnica y operativa: la aplicación mantiene continuidad sin conexión (reanuda sesiones y respeta el cursor de página), el backend asegura inserciones únicas por intento y la plataforma cumple el alcance propuesto para captura robusta, versionado operativo y sincronización confiable. En conjunto, la solución reduce errores de transcripción, mejora la trazabilidad y sienta una base escalable para analítica y futuras extensiones del sistema.

Abstract

This work designs and implements a field-data capture solution for a sugar mill, engineered to operate reliably under intermittent connectivity. The core objective is to enable adaptable forms managed by a mobile-centric backend while preserving the integrity, consistency, and availability of collected information.

The architecture follows an *offline-first* strategy: on the client side, a React Native application (TypeScript) with local persistence in SQLite validates, normalizes, and immediately stores each form session; on the server side, a NestJS backend with PostgreSQL exposes contract-first APIs documented in Swagger and consolidates submissions using JSONB to retain structural flexibility without sacrificing ACID properties. Synchronization relies on idempotent requests that reconcile local and remote state.

Methodologically, the development prioritizes portability and low coupling, makes explicit the operational assumptions and constraints of rural contexts, and implements a test plan that spans API layers (unit and controller coverage, global and per-endpoint metrics) and on-device verification of SQLite and Redux session state, producing reproducible evidence.

Results demonstrate technical and operational viability: the application sustains offline continuity (resuming sessions and preserving page cursors), the backend ensures unique insertions per attempt, and the platform meets the stated scope for robust capture, operational versioning, and reliable synchronization. Overall, the solution reduces transcription errors, improves traceability, and establishes a scalable foundation for analytics and future system extensions.

CAPÍTULO 1

Introducción

La digitalización de los procesos agroindustriales ha pasado de ser una aspiración a convertirse en requisito operativo: la competitividad, la trazabilidad y la capacidad de reacción del sector dependen, cada vez más, de datos de campo oportunos y confiables. En Guatemala, el sector agrícola y pecuario emplea a una cuarta parte de la población y aportó aproximadamente el 9.3 % del PIB en 2022, con la caña de azúcar entre los rubros líderes de exportación [1]. Este peso económico convive con brechas de conectividad en áreas rurales, donde la captura en papel y la doble digitación elevan costos y riesgos de inconsistencia. A nivel global, la Food and Agriculture Organization of the United Nations [2] subraya que transformar los sistemas agroalimentarios exige cerrar brechas de información y adoptar tecnologías que aumenten productividad con sostenibilidad.

La agroindustria azucarera guatemalteca impulsa iniciativas de innovación —incluida la creación del *Innovation Hub* en 2022— que abren una ventana clara para modernizar la captura y gestión de datos de campo [3]. En este contexto, disponer de un backend móvil *offline-first* deja de ser deseable para volverse crítico: asegura continuidad operativa, integridad de la información y sincronización confiable cuando el enlace retorna [4], [5].

La evidencia en entornos con conectividad intermitente muestra que la recolección móvil y los formularios digitales reducen errores de transcripción y aceleran la disponibilidad de datos. Plataformas como *Open Data Kit* (ODK) fueron diseñadas para capturar y validar información aún con recursos limitados, con resultados positivos documentados en despliegues reales [6]. De forma complementaria, experiencias en agricultura reportan que aplicaciones con capacidades *online/offline* —persistencia local y sincronización posterior— mantienen la continuidad operativa y la calidad de datos en campo [7].

Este trabajo propone el diseño e implementación de un **backend móvil para la gestión de formularios adaptables** en un ingenio azucarero, guiado por tres principios: (i) **disponibilidad** en condiciones de conectividad intermitente mediante una arquitectura *offline-first* con base de datos local (*SQLite*); (ii) **integridad y consistencia** a través de transacciones ACID y el modo *Write-Ahead Logging* (WAL), que permite lecturas concurrentes mientras se escriben cambios; y (iii) **sincronización robusta** que converge el estado

local con el servidor sin perder cambios del usuario ni corromper datos [8], [9]. En términos prácticos, el aporte se concreta en: (a) un modelo modular y versionado de formularios, (b) un flujo de validación y persistencia local confiable aun sin internet y (c) una estrategia de sincronización y pruebas de desempeño que evidencian viabilidad operativa con consistencia eventual y alta disponibilidad.

CAPÍTULO 2

Antecedentes

En la última década, la transformación digital se ha consolidado como palanca de competitividad y sostenibilidad en el sector agroindustrial latinoamericano [10]. La agricultura de precisión depende de la captura y el análisis oportuno de datos de campo para ajustar prácticas de riego, fertilización y control fitosanitario. El tránsito desde registros en papel hacia flujos digitales reduce errores de transcripción, acelera la disponibilidad de la información y habilita trazabilidad de extremo a extremo [11], [12].

Evidencia temprana sobre plataformas abiertas como *Open Data Kit* (ODK) mostró que el uso de dispositivos móviles en zonas rurales disminuye de forma sustantiva los tiempos de levantamiento y evita reprocesos de digitación aun con presupuestos limitados [13]. En general, las herramientas digitales aportan validaciones en línea, campos obligatorios y chequeos automáticos que elevan la calidad del dato y permiten sincronización casi en tiempo real cuando existe conectividad [12]. Diversos análisis concluyen que las organizaciones agrícolas con gestión basada en datos operan con mayor eficiencia que aquellas guiadas principalmente por la intuición [12].

En América Latina y Centroamérica, la cobertura de internet en áreas agrícolas suele ser precaria o intermitente. Experiencias reportadas muestran que aplicaciones sin soporte fuera de línea fracasan en su adopción, pues técnicos y productores pasan buena parte de la jornada sin señal [14]. De ahí que la operación *offline-first* se considere hoy un requisito: capturar localmente (mediciones, observaciones, fotografías) durante el día y sincronizar con la nube cuando el dispositivo recupere conexión [14]. Esta filosofía garantiza utilidad en el punto de trabajo, no solo en la oficina.

Ante estos desafíos han proliferado aplicaciones de formularios dinámicos orientadas a captura de datos en campo. Como referencia en el contexto del Ingenio, *DigiForms* ha sido la solución usada originalmente para digitalizar formularios, con operación en entornos remotos y sincronización posterior [11], [15]. Plataformas abiertas como ODK y *KoBoToolbox* incorporan validaciones avanzadas, más de veinte tipos de pregunta y lógica condicional (XLSForm); su aplicación móvil (*KoBo Collect*) está diseñada para uso fuera de línea y sincronización diferida [12]. Otras soluciones comerciales y de código abierto agregan inte-

gración con sensores del dispositivo (GPS para georreferenciación, cámara para evidencias fotográficas, lectura de códigos de barras, firmas digitales), mapas en modo desconectado y, en algunos casos, módulos de analítica o inteligencia artificial para acelerar procesos [11], [12]. En Brasil, *Agroclima PRO* prioriza la usabilidad con interfaces simples para consulta agrometeorológica hiperlocal y operación en campo [16]. En evaluaciones comparativas para proyectos agrícolas, *CommCare* ha destacado por su manejo de casos, flexibilidad y desempeño en entornos exigentes [17]. Como complemento, existen aplicaciones verticales como *AgroBase*, que funcionan como bases de conocimiento fitosanitario portátiles para identificación de plagas, enfermedades y malezas [18].

De estos antecedentes se desprenden lineamientos prácticos para iniciativas de captura en campo: sustituir papel por formularios digitales con validaciones y lógica condicional para elevar la calidad del dato [12], [13]; diseñar con enfoque *offline-first*, con persistencia local transaccional y sincronización diferida y confiable [14]; integrar capacidades del dispositivo (GPS, cámara, lectura de códigos) para enriquecer trazabilidad y auditoría [11]; ofrecer interfaces simples y multilenguaje, junto con paneles centrales para análisis y exportaciones (CSV, Excel, APIs) [12], [16]; y apostar por plataformas modulares—abiertas o comerciales—que faciliten la evolución del catálogo de formularios sin migraciones complejas [17].

CAPÍTULO 3

Justificación

La digitalización y el uso de tecnologías móviles en el sector agroindustrial son estrategias esenciales para enfrentar los desafíos contemporáneos, tales como el aumento de la demanda alimentaria, la optimización de recursos y la mejora en la toma de decisiones basada en datos precisos. Según la Organización de las Naciones Unidas para la Alimentación y la Agricultura, se proyecta que la población mundial alcanzará casi los 9 700 millones de habitantes para el año 2050, lo que exige un incremento sustancial en la producción de alimentos y, por ende, la implementación de sistemas que permitan aumentar la eficiencia y sostenibilidad en la agricultura [19].

En este contexto, desarrollar una aplicación móvil que permita la creación, el llenado y el almacenamiento centralizado de formularios adaptables se presenta como una solución integral para el Ingenio Azucarero. Esta herramienta facilitará la recolección de datos en tiempo real en campo, posibilitando la detección temprana de desviaciones en el estado del cultivo de caña de azúcar y permitiendo intervenciones rápidas que optimicen el proceso productivo.

Estudios educativos y de organismos internacionales han demostrado que la adopción de tecnologías digitales en la agricultura puede incrementar la productividad hasta en un 30 % y reducir el uso de insumos en un rango del 15 % al 20 %, lo que no solo se traduce en una mayor eficiencia operativa, sino también en un uso más racional y sostenible de los recursos naturales [20]. Además, la centralización de la información facilita la trazabilidad y el análisis detallado de cada etapa del proceso productivo, apoyando la toma de decisiones estratégicas y mejorando la competitividad del sector.

En el caso del Ingenio Azucarero, esta iniciativa permitirá que los trabajadores en campo capturen datos de forma ágil mediante dispositivos móviles, y que estos se almacenen de manera segura en una base de datos centralizada. Esto posibilitará realizar análisis estadísticos y visualizaciones que orienten a los responsables en la implementación de medidas correctivas, optimizando tanto la calidad del cultivo como los costos de producción.

Por todo lo anterior, la implantación de este sistema digital para la recolección y aná-

lisis de datos no solo impulsa la modernización del ingenio, sino que también contribuye a la sostenibilidad, eficiencia y competitividad del sector agroindustrial, adaptándose a las exigencias de un mercado global en constante evolución.

CAPÍTULO 4

Objetivos

4.1. Objetivo general

Desarrollar el backend móvil para el Ingenio Azucarero que permita gestionar formularios adaptables en la aplicación, garantizando la calidad de los datos recolectados incluso en entornos con conectividad intermitente.

4.2. Objetivos específicos

- Diseñar un módulo backend de formularios dinámicos, mediante contratos API versionados y configuración declarativa, para habilitar cambios sin despliegues y mejorar la captura en campo.
- Implementar un esquema de recolección y almacenamiento seguro, mediante cifrado en tránsito, tokens firmados, idempotencia y respaldos automáticos.
- Sincronizar la operación *offline-first* desde la aplicación móvil, mediante sesión y cursor persistidos, SQLite local y replicación diferida.

CAPÍTULO 5

Alcance

El presente proyecto delimita el desarrollo de una solución de captura de formularios dinámicos con operación *offline-first*, compuesta por una aplicación móvil (React Native + TypeScript con persistencia local en SQLite) y un backend (NestJS + PostgreSQL) documentado con Swagger. El alcance se concentra en la captura robusta, la sincronización idempotente y el versionado operativo de formularios, manteniendo el dato almacenado en `JSONB` con tipos simples y una fotografía de la estructura utilizada en el momento de la captura.

En el alcance

En el cliente móvil se implementa el motor de formularios, la navegación por páginas, la validación y normalización por tipo, el manejo de grupos repetibles, el almacenamiento local inmediato (incluido el cursor de página) y el envío controlado cuando el formulario está listo. La aplicación funciona sin conectividad, cacheando el árbol de formularios y sus datasets, y reanudando exactamente donde el usuario lo dejó. El campo de firma se captura y persiste como cadena Base64; el resto de campos se registran como tipos simples (texto, numéricos y booleanos).

En el backend se proveen módulos de autenticación, formularios y grupos. Se exponen rutas para autenticación con JWT, obtención del árbol de formularios filtrado por rol, consulta de datasets y creación de entradas mediante un payload idempotente que incluye `form_id`, `index_version_id`, `filled_at_local`, `status`, `fill_json` y `form_json`. Se valida el vínculo entre formulario y versión, los requeridos mínimos (incluidos los grupos) y los permisos del usuario. La especificación de la API queda publicada con Swagger. La base de datos mantiene el catálogo estructural de forma relacional y versionada, y centraliza las capturas en `formularios_entry` con índices adecuados para la operación.

A nivel operativo se incorpora **exclusivamente** el flujo de CI/CD **implementado por el autor** para construir y publicar la imagen Docker del backend hacia entornos de *staging*

y producción con aprobación manual. La orquestación de *pipelines* de pruebas en GitHub Actions pertenece a otro módulo del repositorio y no forma parte de este trabajo. Las migraciones del catálogo son aditivas y compatibles; el almacenamiento operativo de entradas no requiere cambios cuando aparecen campos nuevos, al persistirse como tipos simples bajo nombres internos únicos.

Fuera del alcance

No se incluyen un editor gráfico de formularios, paneles de administración avanzados ni un módulo de analítica o reportería. La transformación y exportación de las capturas hacia múltiples tablas relacionales para consumo externo corresponde a **otro módulo** del backend y no forma parte de este trabajo. Tampoco se aborda mensajería en tiempo real, auditorías de seguridad avanzadas, multitenencia, internacionalización completa ni accesibilidad más allá de las prácticas básicas de la plataforma. **Los *pipelines* de pruebas en GitHub Actions mantenidos por terceros no se documentan en este entregable.**

Supuestos y dependencias

Se asume la disponibilidad de una instancia de PostgreSQL y credenciales de despliegue, así como dispositivos móviles con soporte a la cámara/pantalla para la firma en pantalla y almacenamiento local suficiente. Los roles de usuario y los datasets necesarios estarán previamente definidos en el sistema; la conectividad puede ser intermitente sin afectar la continuidad del trabajo del usuario. Se asume la disponibilidad de credenciales para el registro de contenedores utilizado en el *pipeline* de imagen. Cualquier integración externa adicional (por ejemplo, identidad corporativa o catálogos de terceros) queda fuera del presente alcance.

Criterios de aceptación

La solución se considera completa cuando: (i) la app móvil captura formularios sin conectividad, reanudando sesiones con su cursor de página; (ii) el backend acepta envíos idempotentes y registra exactamente una entrada por intento válido; (iii) el `fill_json` contiene únicamente tipos simples (y firma Base64), mientras que la estructura aplicada se conserva en `form_json` junto con su `index_version_id`; (iv) el árbol de formularios y datasets puede descargarse según el rol del usuario; (v) la API queda documentada en Swagger; y (vi) el flujo de CI/CD **implementado por el autor** construye y publica la imagen de Docker a *staging* y producción con aprobación manual y verificación básica de integridad del artefacto.

CAPÍTULO 6

Marco teórico

6.1. Tecnologías de captura de datos en campo

Las redes de sensores inalámbricos (Wireless Sensor Networks, WSN) conforman una de las bases para la recolección de información en campo, midiendo parámetros como temperatura, humedad del suelo y luminosidad, y enviando estos datos a un nodo central para su procesamiento [21]. Ruiz-García y Lunadei [21] describen cómo la combinación de RFID con WSN permite rastrear procesos agrícolas en tiempo real, mejorando la trazabilidad y reduciendo pérdidas operativas.

Los vehículos aéreos no tripulados (UAV o drones) equipados con cámaras multiespectrales facilitan el mapeo de grandes extensiones de cultivo y la generación de índices de vegetación como NDVI, que reflejan la salud de la biomasa. Matese y Di Gennaro [22] indican que el empleo de drones en vuelos periódicos permite detectar estrés hídrico, infestaciones de plagas y variabilidad espacial de nutrientes, posibilitando intervenciones localizadas antes de que los daños sean generalizados.

Los dispositivos móviles con aplicaciones específicas para el agro han crecido exponencialmente: permiten a los operarios capturar datos alfanuméricos (p. ej., incidencias fitosanitarias, rendimientos) y georreferenciarlos, incluso en modo offline, sin señal. Beshir, Abdel-Aty y Abdel-Hafeez [23] resaltan que las aplicaciones móviles exitosas integran interfaces amigables, georreferenciación GPS y capacidad de almacenamiento local que sincroniza al restaurarse la conectividad, lo que garantiza la calidad de los datos en el punto de captura.

El IoT aplicado al agro consiste en nodos (sensores, actuadores, equipos de riego) interconectados que envían datos a plataformas en la nube, las cuales procesan la información mediante analítica avanzada y retroalimentan recomendaciones (ajustar riego, activar alertas tempranas) a los gestores agrícolas. Kamilaris, Fonts y Prenafeta-Boldú [24] definen el IoT en agricultura como la interconexión de objetos físicos para recopilar y compartir datos sin intervención humana directa, lo cual mejora la eficiencia y la sostenibilidad de los cultivos.

Finalmente, la tecnología RFID (Radio Frequency Identification) y otras soluciones de identificación automática facilitan el rastreo de activos (maquinaria, lotes de productos) y el monitoreo de condiciones en almacenes; al identificar automáticamente un lote de caña con etiqueta RFID, se fortalece la trazabilidad desde la recolección hasta el procesamiento en el ingenio, asociando cada registro de campo con los datos centrales.

6.2. Arquitecturas de aplicaciones móviles y backend s

En el patrón cliente-servidor tradicional, la aplicación móvil actúa como cliente ligero que consume una API RESTful expuesta por un servidor central, manteniendo separadas la lógica de presentación (frontend) y la lógica de negocio (backend). [25].

Para mitigar la dependencia de la conectividad en zonas rurales, se emplea el patrón de sincronización offline/online: la app mantiene una base de datos local (por ejemplo, SQLite) y una cola de operaciones pendientes que registra todas las acciones de inserción o modificación; cuando la conexión se restablece, se envían en bloque los cambios al servidor y un módulo de resolución de conflictos ("last write wins" u otras estrategias) garantiza la coherencia de los datos [26]. Lawrence, Marais, Herbert et al. [26] muestran cómo un sistema de tipo CQRS combinado con event sourcing permite que la aplicación funcione sin conexión y sincronice eventos con el backend cuando hay conectividad, logrando consistencia e integridad de datos.

En el backend , la tendencia hacia arquitecturas basadas en microservicios facilita la escalabilidad y el despliegue independiente de cada componente. [27] definieron microservicios como unidades autónomas que exponen APIs Transferencia de Estado Representacional (REST) independientes y se pueden desplegar, escalar y actualizar sin afectar al resto del sistema; identificaron que este enfoque complementa la tecnología de contenedores (Docker , Kubernetes) a nivel de PaaS en la nube. Cada microservicio en un sistema de captura de datos en campo podría incluir: servicio de autenticación y gestión de roles (JSON Web Tokens), servicio de plantillas de formularios dinámicos (JSON), servicio de gestión de entradas (entries) y servicio de notificaciones en tiempo real.

Otra alternativa es el patrón Backend-as-a-Service (BaaS), donde se externaliza la gestión de autenticación, base de datos en tiempo real, almacenamiento y notificaciones push a plataformas como Firebase o AWS Amplify. Sin embargo, esto puede generar "vendor lock-in" cuando los desarrolladores dependen de APIs específicas del proveedor y resulta costoso o complejo migrar a otro servicio [28].

Para garantizar escalabilidad y seguridad, independientemente del patrón escogido, es fundamental: escalar horizontalmente los servicios críticos, emplear balanceadores de carga (Nginx, HAProxy), proteger las comunicaciones con HTTPS/TLS y usar JSON Web Tokens con firma RSA o HMAC, y auditar y registrar logs para monitorear accesos y detectar anomalías.

6.3. Tecnologías de desarrollo móvil

En el desarrollo de aplicaciones móviles conviven tres enfoques principales: (i) nativo (código y *tooling* oficial por plataforma), (ii) web/progresivo (PWA) y (iii) multiplataforma (una sola base de código que despliega apps instalables para iOS/Android). A continuación se resumen sus rasgos técnicos y se presentan evidencias recientes de adopción y madurez.

6.3.1. Desarrollo nativo (Android/iOS)

El desarrollo nativo ofrece acceso inmediato a todas las APIs del sistema operativo y el mayor control sobre *rendering*, hilos y memoria (Android Studio/Jetpack Compose/Kotlin; Xcode/SwiftUI/Swift). Es la vía recomendada cuando se requiere exprimir al máximo capacidades específicas del hardware o del sistema (renderizado avanzado, *background tasks* complejas, integración profunda con el SO). Las guías y herramientas oficiales, mantenidas por Google y Apple, favorecen la estabilidad a largo plazo y la depuración con perfiles de rendimiento dedicados.

6.3.2. Aplicaciones web progresivas (PWA)

Las PWA priorizan alcance y velocidad de despliegue: una app web responsiva, instalable desde el navegador y con capacidades *offline* gracias a *service workers*. Reducen el costo de mantener múltiples bases de código, pero su acceso a ciertos recursos del dispositivo y a integraciones profundas del SO continúa siendo más acotado que en apps nativas.

6.3.3. Frameworks multiplataforma modernos

La generación actual de frameworks multiplataforma —principalmente React Native (Meta) y Flutter (Google)— permite compartir la mayor parte del código sin resignar una UX cercana a la nativa.

- **React Native (RN).** RN ejecuta JavaScript/TypeScript y mapea componentes a vistas nativas (UIKit/Android Views). La *Nueva Arquitectura* elimina el puente asincrónico histórico y lo reemplaza por JSI (JavaScript Interface), TurboModules y el renderizador Fabric en C++, reduciendo latencias y sobrecargas de serialización [29]-[31]. Esta evolución mejora el rendimiento y la interoperabilidad con código nativo.

- **Flutter.** Flutter compila Dart a binarios nativos y renderiza toda la UI con su propio motor. El motor *Impeller* precompila *shaders* para evitar compilación en tiempo de ejecución, mitigando *jank* y estabilizando fotogramas [32], [33]. Al controlar el *pipeline* gráfico, ofrece animaciones consistentes y un estilo visual uniforme entre plataformas.

6.3.4. Lenguaje y ecosistema: impacto en velocidad de desarrollo

La elección del **lenguaje** afecta directamente la disponibilidad de talento, la curva de aprendizaje y el acceso a librerías. La encuesta de Stack Overflow 2025 reporta que, entre todos los encuestados, **JavaScript** (66.0 %) y **TypeScript** (43.6 %) se sitúan muy por encima de **Dart** (5.9 %) en uso reciente [34]. Dado que React Native usa JavaScript/TypeScript y Flutter usa Dart, este panorama favorece la contratación y el *ramp-up* de equipos con RN.[1]

En cuanto al **ecosistema**, el registro de paquetes `npm` —base natural para proyectos React/Node— se considera *"el registro de software más grande del mundo"* y concentra un ecosistema masivo y diverso de dependencias reutilizables [35], [36]. En RN, la *Nueva Arquitectura* permite integrar módulos nativos tipados mediante *Codegen* y *TurboModules*, combinando productividad de JavaScript con extensibilidad nativa cuando se requiere [30], [37].

6.3.5. Adopción y tendencias (evidencia externa)

Datos recientes de la encuesta de desarrolladores de Stack Overflow sitúan a Flutter y React Native en la parte alta de la categoría *"Other frameworks and libraries"* en 2024, con valores cercanos entre sí; no obstante, en términos de velocidad y disponibilidad de talento, el vector diferencial proviene del dominio de **lenguaje** y **ecosistema** (JS/TS y npm) [38].

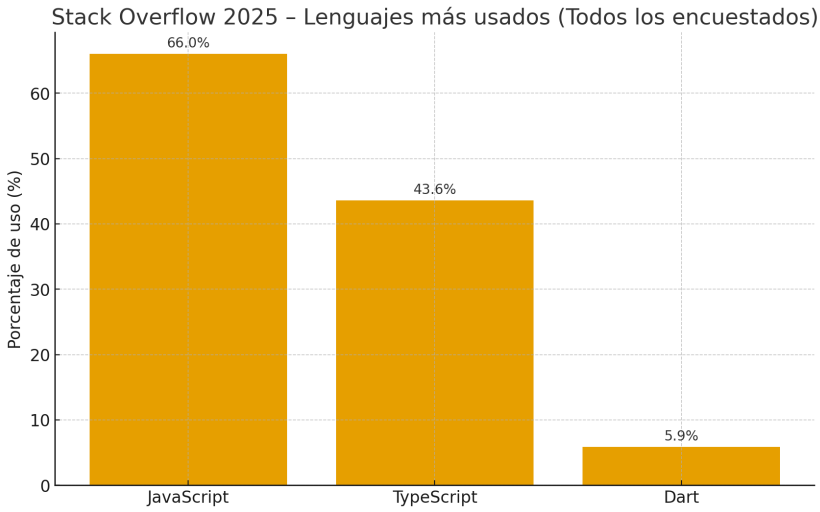


Figura 1: Lenguajes más usados según la encuesta de Stack Overflow 2025

[1] En equipos con base web previa, la transferencia de conocimiento React/TypeScript al entorno móvil reduce tiempos y riesgos iniciales.

6.3.6. Comparativa resumida

Criterio	**Nativo**	**PWA**	**Multiplataforma (RN/Flutter)**
Rendimiento UI	Máximo control y acceso a APIs gráficas; óptimo en cargas extremas	Dependiente del navegador; bueno para apps de negocio	Cercano a nativo (RN con Fabric/JSI; Flutter con Impeller)
Acceso a recursos del dispositivo	Completo e inmediato	Parcial, sujeto a APIs web	Amplio vía plugins/módulos y *bridges* nativos
Velocidad de desarrollo	Menor; dos bases de código (iOS/Android)	Alta; una sola base web	Alta; una base móvil compartida, *hot reload*
Talento disponible	Especialistas por plataforma	Equipo web estándar	**RN:** JS/TS (cuota alta); **Flutter:** Dart (cuota menor)
Persistencia local	Core Data/SQLite /Room, APIs oficiales	IndexedDB/Cache Storage	Acceso a *storages* y BDs (SQLite /SQLCipher en RN; `sqflite`/Hive en Flutter)

Cuadro 1: Comparación de enfoques de desarrollo móvil

6.3.7. Casos y madurez en producción

Organizaciones de gran escala reportan métricas de estabilidad y rendimiento con RN y Flutter. En RN, la *Nueva Arquitectura* (JSI, Fabric, TurboModules) y el uso de *Codegen* han madurado oficialmente [29], [31], [37]. En Flutter, el motor *Impeller* y las guías de perf actualizadas refuerzan su rendimiento [32], [39].

6.4. Comparativa de tecnologías modernas para desarrollo de APIs RESTful

El ecosistema actual de desarrollo backend ofrece múltiples *frameworks* para construir APIs REST en distintos lenguajes (TypeScript/JavaScript, Java, Python, C#, Go). La elección depende de factores como arquitectura, productividad, soporte de estándares (OpenAPI), seguridad y escalabilidad. A continuación se describen opciones representativas y se presenta una comparativa técnica, priorizando documentación oficial y fuentes verificables.

6.4.1. NestJS (Node.js/TypeScript)

NestJS es un *framework* progresivo para Node.js que adopta una arquitectura modular con inyección de dependencias, controladores y servicios, y soporte de *middleware*,

guards e *interceptors*. Ofrece integración nativa con OpenAPI/Swagger mediante el módulo `@nestjs/swagger` para documentar automáticamente rutas y Objeto de Transferencia de Datos (DTO) s, y dispone de módulo oficial para GraphQL [40], [41]. Al basarse en TypeScript, favorece el tipado estático y el mantenimiento de bases de código grandes [42].

Fortalezas: (i) estructura de grado empresarial con DI y módulos; (ii) generación automática de documentación OpenAPI; (iii) integración opcional con GraphQL y compatibilidad con el ecosistema de Node; (iv) patrón de pruebas facilitado por su modularidad. *Consideraciones:* curva de aprendizaje mayor que micro-*frameworks* minimalistas.

6.4.2. Express.js (Node.js/JavaScript)

Express es un *microframework* minimalista y no opinado para Node.js: provee enrutamiento y *middleware* con muy poca sobrecarga y deja la estructura a criterio del equipo. Es sencillo empezar ("Hello World" en pocas líneas) y existen guías oficiales para enrutamiento, uso y escritura de *middleware* [43]-[46]. *Trade-off:* en proyectos grandes, la ausencia de convenciones obligatorias exige mayor disciplina arquitectónica.

6.4.3. Spring Boot (Java)

Spring Boot permite crear aplicaciones *stand-alone* de producción con un enfoque opinado sobre Spring, integrando web MVC, acceso a datos, seguridad y más. La documentación oficial detalla sus características núcleo y su filosofía "production-ready" [47], [48]. Para documentación de APIs, la integración con OpenAPI/Swagger suele hacerse con `springdoc-openapi`, que genera especificaciones y UI de forma automática a partir de anotaciones [49].

6.4.4. FastAPI (Python)

FastAPI es un *framework* moderno asíncrono (ASGI) que aprovecha los *type hints* de Python para validación automática y genera documentación interactiva (*Swagger UI* y ReDoc) sin esfuerzo adicional; todo ello basado en un esquema OpenAPI [50], [51]. Su modelo de dependencias y *routers* permite modularidad, y su enfoque asincrónico resulta adecuado para I/O concurrente.

6.4.5. Flask (Python)

Flask es un *microframework* WSGI ligero, con núcleo pequeño y extensiones para ampliar funcionalidad. La documentación oficial lo presenta como un marco sencillo para comenzar rápido y escalar a aplicaciones complejas mediante *Blueprints* y extensiones [52]. Para OpenAPI/Swagger se emplean extensiones de terceros (p. ej., `Flask-RESTx` o `flasgger`).

6.4.6. ASP.NET Core (C#/.NET)

ASP.NET Core ofrece APIs HTTP tanto con controladores (MVC) como con *Minimal APIs*, recomendadas para nuevos proyectos por su sencillez y alto rendimiento [53], [54]. La generación de documentación OpenAPI/Swagger se habilita fácilmente con *Swashbuckle*, y existen guías oficiales para ello [55]. La plataforma integra autenticación/autorización (JWT , *Open Authorization* 2.0 (OAuth 2.0) /OpenID Connect) y DI de forma nativa [56].

6.4.7. Gin (Go)

Gin es un *framework* web minimalista para Go, construido sobre `net/http`, con enrutamiento eficiente, soporte de *middleware* y utilidades para JSON y validación [57], [58]. La documentación describe su modelo de *router groups* y el contexto de solicitud, facilitando la construcción de APIs REST de alto rendimiento. La documentación OpenAPI/Swagger se integra con herramientas comunitarias (p. ej., `swaggo`).

6.4.8. Notas sobre rendimiento y escalabilidad

Los resultados comparativos dependen del caso de uso y la configuración. En benchmarks públicos de referencia, los *stacks* compilados (Go/ASP.NET Core/Java bien optimizado) suelen figurar entre los de mayor rendimiento bruto, mientras que Node.js y Python asíncrono ofrecen buen desempeño para I/O concurrente con excelente productividad. Para comparar de manera estandarizada, pueden consultarse los *TechEmpower Framework Benchmarks* [59].

6.4.9. Comparativa técnica resumida

Framework	Lenguaje (tipado)	Arquitectura base	Comentarios clave
NestJS	TypeScript (estático) [42]	MVC modular, DI, *middleware*, *guards*	Estructura enterprise en Node; pruebas facilitadas por DI.
Express	JavaScript (dinámico)	Enrutamiento + *middleware* minimalista [43], [45]	Arranque ultrarrápido; requiere disciplina en proyectos grandes.
Spring Boot	Java (estático)	MVC con DI (Spring), *auto-configuration* [47], [48]	Ecosistema empresarial completo; *production-ready*.
FastAPI	Python (dinámico con *type hints*)	Endpoints declarativos, dependencias, ASGI	Validación y docs integradas; muy productivo.
Flask	Python (dinámico)	Núcleo mínimo + *Blueprints* [52]	Flexibilidad máxima; ideal para APIs pequeñas/-medias.
ASP.NET Core	C# (estático)	Controladores o *Minimal APIs*; DI nativa [53]	Alto rendimiento; seguridad y herramientas de nivel enterprise.
Gin	Go (estático)	Handlers + *middleware* ligeros [57]	Binarios pequeños; gran eficiencia y concurrencia.

Cuadro 2: Resumen comparativo de *frameworks* modernos para APIs REST.

En Node.js, NestJS aporta una base modular tipada y una experiencia de documentación/seguridad coherente con prácticas de ingeniería de mayor escala [40]. En ecosistemas Java, Python, .NET y Go existen alternativas igualmente maduras con distintos compromisos. Para proyectos que ya privilegian TypeScript y el ecosistema de Node, el balance entre productividad, estructura y extensibilidad que ofrece NestJS resulta especialmente atractivo sin sacrificar estándares como OpenAPI o integraciones como GraphQL.

6.5. Bases de datos embebidas y relacionales

Las bases de datos embebidas en el dispositivo y los motores relacionales en servidor constituyen pilares complementarios para gestionar el ciclo de vida de los datos en apli-

caciones móviles que operan tanto sin conexión como conectadas. En términos prácticos, los motores embebidos priorizan latencia mínima y disponibilidad local —características esenciales en terreno cuando la conectividad es intermitente—, mientras que los motores relacionales en servidor concentran la consolidación transaccional, la gobernanza y la escalabilidad horizontal/vertical. La arquitectura *offline–online* típicamente articula ambas capas: (i) una persistencia local donde se resuelven lecturas y escrituras inmediatas, y (ii) un backend relacional que actúa como fuente de verdad global, aplicando validaciones, controles de acceso y analítica. A continuación, se comparan **SQLite** (motor embebido) y **PostgreSQL** (motor relacional), destacando ventajas, limitaciones y consideraciones de diseño para sincronización y consistencia, y se presenta evidencia de adopción en la industria.

6.5.1. Evidencia de adopción y mercado (2025)

La *Stack Overflow Developer Survey 2025* reporta, para "All Respondents", los siguientes porcentajes de uso por base de datos: PostgreSQL (55.6 %), MySQL (40.5 %), SQLite (37.5 %) y Microsoft SQL Server (30.1 %), seguidos por Redis (28.0 %) y MongoDB (24.0 %) [60]. La Figura 2 reproduce estos resultados y la Tabla 3 los resume. Adicionalmente, el informe destaca que PostgreSQL es "la más admirada y la más deseada" desde 2023 [60]. En el plano embebido, la página oficial de SQLite afirma que es "el motor SQL más ampliamente desplegado en el mundo", con "miles de millones" de copias y presencia en la totalidad de teléfonos móviles y la mayoría de computadoras y aplicaciones de escritorio [61].

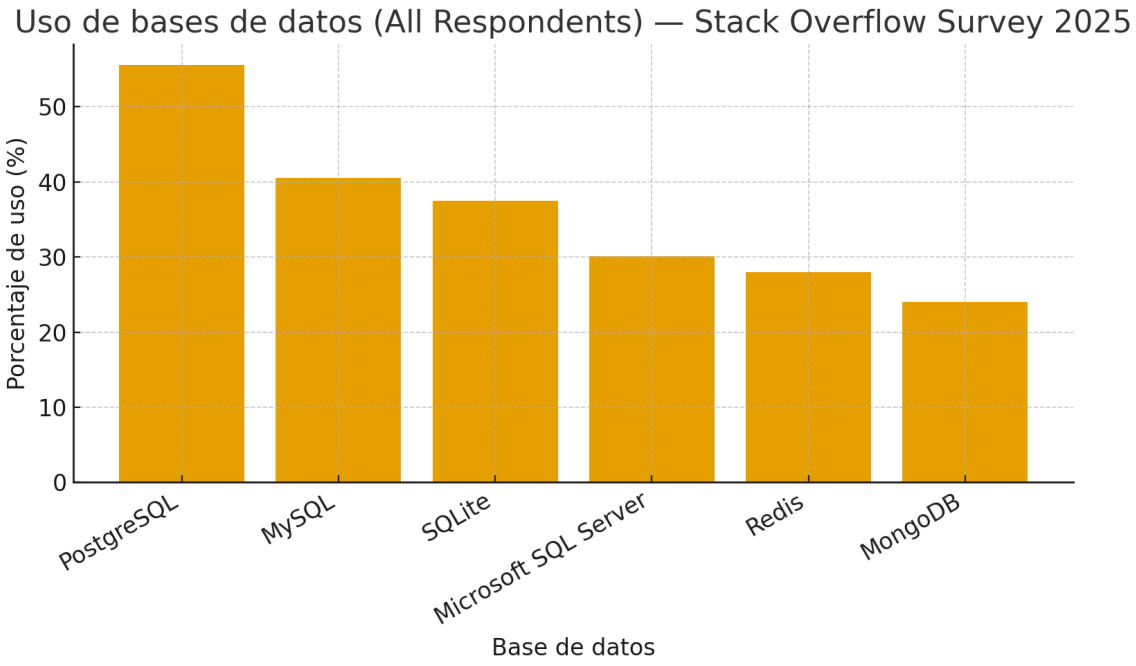


Figura 2: Uso de bases de datos en 2025

Fuente: elaboración propia con datos de Stack Overflow [60].

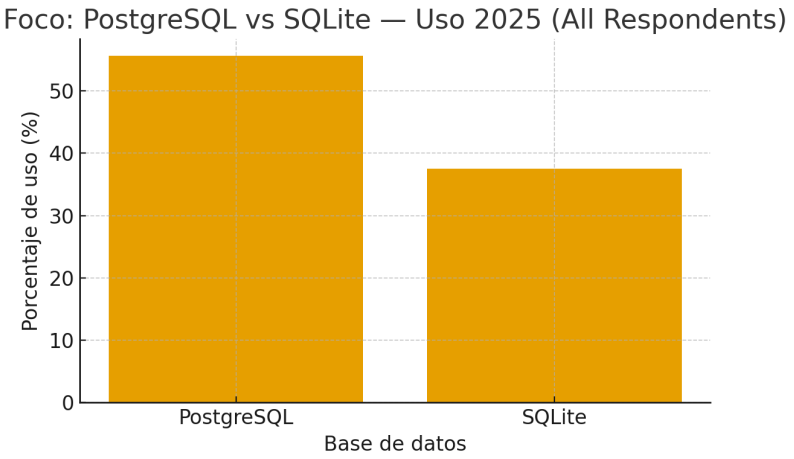


Figura 3: Uso de PostgreSQL y SQLite en 2025

Fuente: elaboración propia con datos de Stack Overflow [60].

Cuadro 3: Porcentaje de uso (All Respondents) por base de datos — 2025

Base de datos	**Uso (%)**
PostgreSQL	55.6
MySQL	40.5
SQLite	37.5
Microsoft SQL Server	30.1
Redis	28.0
MongoDB	24.0

Fuente: elaboración propia con datos de Stack Overflow [60].

6.5.2. SQLite: motor embebido

SQLite es un SGBD relacional ligero distribuido como biblioteca vinculada a la aplicación. No requiere un servidor independiente, almacena los datos en un único fichero local y ofrece propiedades Atomicidad, Consistencia, Aislamiento y Durabilidad (ACID) mediante transacciones atómicas y mecanismos de *journaling*, lo que lo convierte en un candidato natural para persistencia *offline* en dispositivos móviles [62].

Ventajas.

- **Operación offline y latencia mínima.** Las consultas y transacciones no dependen de la red, garantizando disponibilidad continua y respuestas muy rápidas incluso sin cobertura.
- **Integración sencilla.** Al no requerir despliegue de servidores ni configuración adicional, acelera desarrollo, pruebas y empaquetado de la app (menor huella operativa).

- **Adecuado para cargas ligeras y patrones CRUD.** En volúmenes moderados y consultas acotadas, evita viajes a la nube y, por tanto, suele superar en rendimiento a alternativas remotas para operaciones interactivas [62].

Limitaciones.

- **Concurrencia de escritura.** Solo permite un escritor a la vez (p. ej., con *Write-Ahead Logging* (WAL) mejora la lectura concurrente, pero la escritura sigue siendo serial), lo que puede generar cuellos de botella con múltiples productores locales [62].
- **Escalabilidad y funciones avanzadas.** No está orientado a bases de datos masivas ni a tasas de transacciones muy altas; carece de funcionalidades empresariales propias de un motor servidor.
- **Replicación y acceso remoto.** No incluye sincronización nativa; la reconciliación con el backend debe implementarse en la aplicación o vía servicios especializados.

Implicaciones de diseño. En un flujo offline-first , es recomendable: (i) definir transacciones cortas e idempotentes para minimizar bloqueos; (ii) mantener índices estrictamente necesarios para no penalizar escrituras; y (iii) registrar metadatos de sincronización (p. ej., marcas de tiempo y estados *pending/synced/failed*) que permitan reintentos y resolución determinística de conflictos [62].

6.5.3. PostgreSQL : motor relacional en servidor

PostgreSQL es un SGBD relacional de código abierto con soporte completo de transacciones ACID y control de concurrencia multiversión (Control de Concurrencia Multiversión (MVCC)) por defecto, lo que preserva la integridad bajo alta concurrencia. Destaca por su **extensibilidad** (tipos definidos por el usuario, funciones y procedimientos en múltiples lenguajes, métodos de índice y extensiones), sus **tipos de datos avanzados** (p. ej., `JSON` /`JSONB` , arreglos, rangos) e **índices** ricos (parciales, por expresión, búsqueda de texto completo), además de **ejecución paralela**, **particionamiento nativo** y **mecanismos de replicación** física y lógica para alta disponibilidad y escalado de lectura [63].

Ventajas.

- **Consistencia fuerte con alta concurrencia.** ACID + MVCC minimizan bloqueos en lecturas/escrituras simultáneas.
- **Extensibilidad y flexibilidad de modelo.** `JSONB` y tipos ricos permiten combinar esquemas relacionales con documentos semiestructurados manteniendo transacciones y *constraints* [63].
- **Replicación y recuperación robustas.** Replicación *streaming* (asíncrona/síncrona), replicación lógica y *Point-In-Time Recovery* (*PITR*) para resiliencia operativa.
- **Rendimiento en consultas complejas.** Planificador avanzado, índices variados y paralelismo mejoran *analytics* y agregaciones sobre grandes volúmenes [63].

- **Ecosistema maduro y licencia permisiva.** Amplia comunidad, soporte profesional opcional y licencia tipo BSD (costo de licencia cero) [63].

Limitaciones.

- **Opera como servicio.** Requiere despliegue/operación (o usar ofertas gestionadas); por diseño no ofrece modo *offline* en cliente.
- **Latencia de red.** Cada operación desde el dispositivo implica viaje a servidor; se mitiga con caché local, colas y sincronización diferida.

Implicaciones de diseño. En una arquitectura *offline–online*: (i) confirmar localmente en SQLite y propagar al backend mediante lotes idempotentes; (ii) usar claves naturales o identificadores deterministas para evitar duplicados en la consolidación; (iii) diseñar particionamiento por tiempo/tenant y plan de índices orientado a consultas críticas; y (iv) aprovechar replicación y *read replicas* para aislar cargas de lectura analítica sin afectar la escritura transaccional [63].

Síntesis. En aplicaciones móviles con capturas en campo, SQLite aporta disponibilidad y latencia mínima en el dispositivo, mientras que PostgreSQL concentra consistencia global, seguridad y escalabilidad. La evidencia de mercado de 2025 indica alta adopción de ambas tecnologías en sus respectivos dominios: PostgreSQL en el backend transaccional y SQLite en el ecosistema embebido [60], [61].

6.6. Sincronización offline/online en aplicaciones móviles

En contextos rurales o con conectividad intermitente, el diseño de aplicaciones móviles debe priorizar la continuidad operativa incluso sin red, y diferir la comunicación con el servidor hasta que la conexión se restablezca. Este enfoque, conocido como offline-first , asume que el dispositivo es la primera fuente de verdad durante los periodos sin cobertura; por tanto, las operaciones de lectura y escritura se resuelven localmente y se reconcilian con el backend cuando vuelve la red. En dominios agroindustriales, esta capacidad resulta crítica para mantener la captura oportuna de datos en campo, reducir pérdidas por interrupciones y asegurar la trazabilidad de la información recolectada [64]. Más ampliamente, los retos clásicos de la computación móvil —latencia, desconexiones, recursos limitados— refuerzan la necesidad de diseñar aplicaciones tolerantes a fallos de red y con estrategias explícitas de sincronización, consistencia y resolución de conflictos [25].

6.6.1. Almacenamiento local y caché de datos

La base técnica del modo sin conexión es una capa de persistencia local que permita consultar y modificar datos sin depender de la red. En la práctica, esto se implementa con motores embebidos (p. ej., SQLite, Realm, Couchbase Lite) o con cachés persistentes respaldadas en disco. En Android, el patrón recomendado es usar Room como capa de

acceso a datos sobre SQLite para definir entidades, *DAOs* y transacciones de forma segura y tipada [65]. El uso de almacenamiento local no solo mejora la experiencia (baja latencia, disponibilidad), sino que evita pérdida de información al almacenar de inmediato los cambios que el usuario realiza mientras está *offline*.

En backend s administrados, algunos servicios simplifican la persistencia local y el reenvío de operaciones. Firebase Realtime Database, por ejemplo, mantiene una caché local y encola escrituras cuando no hay red, reenviándolas automáticamente al reconectar; esta estrategia permite que la aplicación se comporte "como si" estuviera en línea y que las vistas se hidraten desde la caché hasta que arriben los datos remotos [66], [67]. En paralelo, conocer los límites y el *trade-off* de cada motor es clave: SQLite es ligero y ACID , ideal como almacenamiento embebido, pero su concurrencia de escritura es limitada, por lo que se recomienda un diseño de transacciones cortas y colas de operaciones bien definidas [62].

6.6.2. Colas de operaciones y estrategias offline

Para encapsular la variabilidad de la red, las apps offline-first registran las mutaciones como eventos u operaciones en una cola persistente. Cada operación incluye, como mínimo, un identificador idempotente, marca de tiempo y el tipo de acción a ejecutar en el servidor (crear, actualizar, borrar). Al detectarse conectividad, la cola se procesa en orden lógico, aplicando políticas de reintento (p. ej., *exponential backoff*) y verificación de éxito para marcar definitivamente la operación como sincronizada. En Android, la ejecución diferida y confiable de trabajos en segundo plano suele delegarse en *WorkManager*, que respeta restricciones de batería, red y reinicios del dispositivo [67]. A nivel conceptual, esta separación entre "escritura local inmediata" y "propagación remota eventual" incrementa la resiliencia, disminuye errores percibidos por el usuario y ordena el manejo de fallos temporales de red [68].

6.6.3. Sincronización de datos: manual vs automática

El disparo de la sincronización puede ser automático, manual o mixto. La sincronización automática se activa al recuperar conectividad y procesa en segundo plano las operaciones pendientes (push) y/o consulta cambios del servidor (pull), buscando mantener el estado local lo más fresco posible sin intervención del usuario. Es la estrategia preferida para flujos críticos y colaborativos, donde la frescura de datos es prioritaria. La sincronización manual, en cambio, requiere una acción explícita del usuario (p. ej., un botón "Sincronizar ahora"), otorgando mayor control sobre el uso de datos móviles y ofreciendo transparencia respecto de cuántos registros están pendientes; suele ser útil cuando los costos de conectividad son altos o el usuario debe revisar borradores antes de enviarlos [68]. Entre ambos extremos, existen esquemas programados (por intervalo) o orientados a eventos (p. ej., notificaciones *push* que indican disponibilidad de nuevos datos), que combinan conveniencia y control.

6.6.4. Resolución de conflictos

Cuando múltiples actores modifican el mismo recurso en ventanas de tiempo solapadas —dispositivos distintos o cliente y servidor durante la desconexión— aparecen conflictos. Resolverlos exige instrumentar metadatos (versiones, *timestamps*, *hashes* o *vector clocks*) y definir políticas claras: *last-write-wins* (la escritura más reciente prevalece), "cliente gana" o "servidor gana" (preferencias fijas por origen), o estrategias de fusión específicas del dominio que conserven y combinen cambios compatibles [68]. Algunos ecosistemas facilitan esta tarea: CouchDB/PouchDB mantienen múltiples revisiones, eligen un ganador determinístico y permiten que el desarrollador inspeccione y resuelva divergencias con lógica personalizada [69]. En la familia Couchbase, Sync Gateway incorpora resolutores automáticos y *callbacks* para reconciliación a medida, logrando convergencia eventual con mínimos conflictos "human-in-the-loop" [70]. El principio rector es alinear la estrategia con el caso de uso: en formularios de campo, por ejemplo, puede bastar *last-write-wins* con *audit trail*; en registros sensibles, puede ser preferible preservar ambas versiones para revisión posterior.

6.6.5. Gestión de estado con Redux, persistencia y coordinación con la base de datos

En un enfoque *offline–first*, Redux funge como la "fuente de verdad" de la interfaz mientras la base de datos local (p. ej., SQLite) conserva el estado duradero y auditable. Para evitar inconsistencias y recomputaciones costosas, se recomienda normalizar el estado (entidades por *id*, relaciones por referencia) y diseñar *slices* separados para `ui`, `entities` y `sync` (metadatos, colas, marcas de tiempo) [71], [72]. La normalización reduce duplicidad y simplifica actualizaciones y selectores memoizados.

- **Persistencia y rehidratación.** Para preservar continuidad tras cierres/reinicios, la persistencia del *store* puede implementarse con *Redux Persist*, postergando el renderizado de la app hasta la rehidratación (*PersistGate*) y seleccionando con listas blanca/negra qué *slices* persisten (evitando cachés voluminosas) [73]. Cuando se usa RTK Query, la rehidratación del estado de la API puede integrarse con el ciclo de `REHYDRATE` para mantener coherentes la caché y los datos persistidos [74].
- **Actualizaciones optimistas con RTK Query.** Para minimizar latencia percibida, las mutaciones pueden aplicar *optimistic updates* (actualizar UI/caché antes de la confirmación remota) y revertirse si falla la operación. RTK Query ofrece utilidades (`onQueryStarted`, `updateQueryData`, `upsertQueryData`) para actualizar manualmente entradas de caché y manejar confirmación o *rollback* [75], [76].
- **Cola transaccional ("outbox") e idempotencia .** Para conciliar "escritura local inmediata" con "propagación remota eventual", se recomienda una **cola transaccional** persistida en SQLite: cada operación captura un identificador idempotente, tipo de acción y carga útil. Este patrón evita escrituras dobles inconsistentes y permite reintentos seguros: la app registra la operación en la BD (misma transacción que el cambio local) y un proceso separado la envía al servidor; al confirmarse, se marca como sincronizada [77], [78]. La idempotencia a nivel de API del backend asegura que reintentos no dupliquen efectos.

- **Ejecución en segundo plano y reintentos.** El vaciado de la cola puede programarse con gestores de trabajos que soportan políticas de reintento y *backoff* exponencial (p. ej., WorkManager en Android), respetando restricciones de batería/red y tolerando reinicios del dispositivo [79], [80]. En entornos React Native/Expo, se puede articular con tareas en segundo plano para *flush* periódico o por eventos.
- **Concurrencia y transacciones en SQLite.** Dado que SQLite es de un solo escritor simultáneo, conviene mantener transacciones *cortas*, agrupar escrituras (*batching*) y habilitar WAL para que lectores no bloqueen escritores (y viceversa), manteniendo la UI fluida [81].
- **Invalidez y frescura de datos.** Las mutaciones deben invalidar etiquetas (*tags*) o claves de caché para provocar *refetch* selectivo cuando haya conectividad, re-alineando el estado de Redux con la "verdad" del backend sin descargar más de lo necesario [76].

Resumen práctico. (i) Normalizar entidades en Redux y persistir sólo *slices* críticos; (ii) usar RTK Query para *fetch* y caché, con *optimistic updates* + *rollback*; (iii) registrar mutaciones en una *outbox* idempotente en SQLite; (iv) vaciar la cola con trabajos en segundo plano y reintentos exponenciales; (v) usar WAL y transacciones breves para mitigar la restricción de un único escritor en SQLite; y (vi) invalidar caché de RTK Query tras confirmación remota para garantizar convergencia UI–servidor.

6.6.6. Frameworks y herramientas de sincronización offline

- **Couchbase Lite + Sync Gateway.** Proporciona base embebida en el dispositivo, replicación bidireccional sobre WebSockets y resoluciones de conflicto basadas en revisiones, con opción de resolutores personalizados. Es apropiado para escenarios con múltiples clientes y necesidad de convergencia consistente sin bloquear la UI [70].
- **Firebase Realtime Database.** Habilita persistencia local y *queuing* automático de escrituras fuera de línea; al reconectar, reenvía pendientes y rehidrata vistas. Simplifica el desarrollo al manejar las colas y la detección de red a nivel de SDK, a costa de menos control fino sobre la política de reconciliación [66], [67].
- **PouchDB/CouchDB.** Implementa almacenamiento local (web/híbrido) y sincronización eventual con el servidor CouchDB. El modelo de múltiples revisiones y el algoritmo de convergencia reducen conflictos irreconciliables, y permiten inspección y fusión bajo reglas de negocio [69].
- **AWS AppSync/Apollo (GraphQL).** Ofrecen caché local (a menudo sobre SQLite), *retry* automático, *subscriptions* para cambios en tiempo real y políticas de *merge* configurables en el cliente. Resultan útiles cuando la interfaz de datos se modela como un grafo y se requieren actualizaciones selectivas por tipo/consulta [68].

En síntesis, un diseño offline-first maduro integra: (i) una base local robusta y transacciones cortas; (ii) colas persistentes con idempotencia y reintentos; (iii) disparadores de sincronización coherentes con el contexto de uso; y (iv) una política de conflictos alineada al

dominio. Con estos pilares, la app mantiene continuidad operativa y calidad de datos en campo —y el backend consolida, audita y distribuye los cambios— aun cuando la conectividad sea la excepción y no la regla [25], [64], [67].

6.7. Notificaciones en Tiempo Real en Aplicaciones Móviles Híbridas (React Native)

6.7.1. Fundamentos de la Comunicación en Tiempo Real

En aplicaciones distribuidas modernas, la comunicación en tiempo real se refiere a la capacidad de entregar datos o eventos a múltiples componentes tan pronto como ocurren, minimizando la latencia percibida por el usuario. A diferencia del modelo tradicional cliente–servidor de consulta–respuesta (request/response), en los sistemas en tiempo real el servidor puede empujar datos nuevos hacia los clientes sin esperar una solicitud. Para ello se emplean patrones de diseño y protocolos especializados:

- **Patrón Publish/Subscribe (pub/sub):** modelo de mensajería asíncrona donde los emisores publican en un canal o *topic* y los suscriptores reciben una copia de cada evento, desacoplando completamente productores y consumidores.
- **WebSockets:** protocolo bidireccional persistente (RFC 6455) sobre TCP, que permite al servidor y al cliente enviar datos mutuamente en cualquier momento, creando un canal full-duplex de baja latencia.
- **Server-Sent Events (SSE):** flujo unidireccional basado en HTTP donde el servidor envía eventos continuos al cliente mediante la API `EventSource`. Es sencillo y atraviesa proxies, pero no permite al cliente enviar datos por el mismo canal.
- **Long Polling y otras técnicas de sondeo:** antes de WebSockets/SSE, el cliente reutilizaba peticiones HTTP abiertas (*long polling*) o consultas periódicas (*polling*) para simular tiempo real, pero con mayor sobrecarga y latencia.

En suma, mantener canales abiertos (WebSockets/SSE) o emplear pub/sub en el backend evita el sondeo activo del cliente, logrando que un cambio (p. ej., un nuevo dato o mensaje) se propague inmediatamente a todos los clientes suscritos.

6.7.2. Notificaciones en tiempo real con SQL Server: alternativas y herramientas

Para habilitar experiencias reactivas sin recurrir a *polling* intensivo, es necesario instrumentar mecanismos orientados a eventos que propaguen cambios desde la base de datos hacia la capa de aplicación y, de allí, a los clientes. En el ecosistema de SQL Server existen opciones nativas y patrones complementarios que permiten emitir notificaciones con distintas garantías y costos operativos. A alto nivel, el objetivo es detectar cambios en datos

de interés, transformarlos en eventos de dominio y distribuirlos de forma eficiente a consumidores suscritos (p. ej., aplicaciones móviles), aprovechando *publish/subscribe* cuando sea posible [82].

- **Query Notifications (SqlDependency + Service Broker).** Permiten que SQL Server notifique a la aplicación cuando el resultado de una consulta parametrizada cambia, eliminando la necesidad de *polling* periódico. La aplicación registra una dependencia (`SqlDependency`) asociada a una consulta "suscrita"; cuando ocurre una modificación que afecta el conjunto de resultados, el motor envía una notificación a través de Service Broker y el cliente invalida caché o vuelve a consultar [83]. Es una solución idónea en .NET; fuera de ese ecosistema, su adopción es menos directa y suele requerir capas intermediarias.

- **Service Broker + *triggers* (eventos a colas).** Un *trigger* por tabla puede publicar un mensaje en una cola de Service Broker ante inserciones/actualizaciones. Un proceso consumidor (microservicio) lee la cola, convierte el mensaje a un evento de dominio y lo reemite hacia los clientes. Este patrón materializa un canal *pub/sub* interno en la base de datos y desacopla el origen (tabla) del emisor final, alineándose con los patrones de integración empresarial [82]. Requiere diseño cuidadoso para evitar trabajo excesivo dentro del *trigger* y preservar la latencia de transacciones.

- **Hubs en tiempo real (SignalR).** En entornos .NET, un *hub* SignalR expone métodos que el servidor invoca para *push* hacia clientes conectados (WebSockets/SSE/Long Polling). Combinado con Query Notifications o con un consumidor de Service Broker, el servidor actúa como orquestador: recibe el aviso de la BD y notifica a los clientes a través del *hub* [84]. Ofrece administración de conexiones, grupos y reintentos a nivel de transporte.

- **Emisión desde la lógica de negocio (Node.js/NestJS).** En arquitecturas no-.NET, es habitual que el backend emita eventos tras cada operación CRUD exitosa, usando WebSockets (o un *broker* como Redis/RabbitMQ). Este enfoque traslada la responsabilidad de notificación a la aplicación (fuente de verdad de la operación), y evita acoplarse a mecanismos nativos de la BD. Suele resultar más simple de operar y mantener en *stacks* JavaScript.

En síntesis, las opciones nativas de SQL Server (Query Notifications / Service Broker) reducen el *polling* y encajan mejor con aplicaciones .NET; en Node.js, se privilegia emitir desde la capa de dominio y usar un canal *pub/sub* gestionado por la propia aplicación [82]-[84]. En todos los casos, conviene definir identificadores idempotentes, orden lógico de eventos y políticas de reintento para garantizar consistencia observada por los clientes.

6.7.3. Implementación práctica en React Native (Android e iOS)

En el cliente móvil se combinan canales de *push* (para despertar la app incluso cerrada) con conexiones persistentes (para datos en vivo mientras la UI está activa):

- **WebSockets / Socket.IO.** Permiten recibir eventos de baja latencia mientras la app está en primer plano (o en *background* activo). En React Native puede usarse la API estándar `WebSocket` o `socket.io-client`. La app se suscribe a temas/salas y actualiza la UI en tiempo real.
- **SignalR Client.** Con `@microsoft/signalr`, la app se conecta a un *hub* SignalR (o Azure SignalR Service) y escucha métodos del servidor; resulta natural si el backend es .NET o si existe un microservicio dedicado a *hubs* [84].
- **Firebase Cloud Messaging (FCM).** Para notificaciones *push* nativas —incluso con la app cerrada—, React Native Firebase u OneSignal simplifican la recepción. El backend entrega mensajes vía Admin SDK; Android/iOS los enrutan al dispositivo [85]. FCM es eficiente en reposo (canal del SO) y apropiado para alertas, *badges* o sincronizaciones diferidas.
- **Combinación operativa.** Mientras la app está activa, WebSockets mantienen la UI actualizada; cuando está cerrada o en suspensión profunda, FCM notifica cambios críticos y permite "despertar" flujos de sincronización en segundo plano. En escenarios offline-first , las notificaciones deberían disparar lectura/sincronización diferida y no sustituir el almacenamiento local [67], [85].

Buenas prácticas del cliente. (i) Reconexión exponencial y *jitter* para enlaces persistentes; (ii) confirmaciones lógicas (ACK) a nivel de dominio para lograr entrega "al menos una vez" sin duplicar efectos; (iii) colas locales y estados *pending/processed* cuando se combine *push* con sincronización *offline* [67].

6.7.4. Comparativa de opciones

- *Latencia.* Conexiones persistentes (WebSockets/SignalR) ofrecen latencia muy baja para flujos de alta frecuencia; *push* (FCM/APNs) puede presentar mayor retardo por el enrutamiento vía los servicios del SO [84], [85].
- *Consumo de recursos.* WebSockets mantienen *keep-alives* y consumen batería/memoria; *push* es más eficiente en reposo al reutilizar el canal del sistema [85].
- *Complejidad operativa.* FCM/APNs exige configuración nativa y manejo de certificados; WebSockets/SignalR requieren infraestructura para conexiones a escala y *stateful fan-out* [84], [85].
- *Escalabilidad.* WebSockets/SignalR escalan con *backplanes* (p. ej., Redis *pub/sub*); *push* delega la escala global a Google/Apple [82], [84], [85].
- *Alineación con Node.js/NestJS.* NestJS soporta WebSockets/SSE de forma nativa; para SignalR suele preferirse un servicio .NET o Azure SignalR Service; si SQL Server no es el emisor, la opción más directa es publicar eventos desde la lógica de negocio y usar un patrón *pub/sub* [82].

Recomendación de diseño. Centralizar la emisión de eventos en el backend de dominio (NestJS), con una *outbox* transaccional ligada a SQL Server para garantizar idempotencia ;

propagar por WebSockets a clientes activos y usar FCM para alertas y reconexiones diferidas. Si el ecosistema es .NET, integrar Query Notifications/Service Broker para invalidar cachés y disparar *fan-out* vía SignalR [82]-[85].

6.8. Validaciones dinámicas mediante código TypeScript

En escenarios de formularios dinámicos es habitual permitir que usuarios avanzados definan fragmentos de código para adaptar reglas de validación sin necesidad de desplegar una nueva versión de la aplicación [86], [87]. No obstante, habilitar JavaScript arbitrario conlleva riesgos de seguridad —acceso al DOM, APIs sensibles o bucles infinitos— que pueden comprometer tanto la estabilidad de la interfaz como la integridad de los datos [87]. Por ello, resulta esencial combinar un **tipado estático** que limite lo que el usuario puede escribir, con mecanismos de **aislamiento en tiempo de ejecución** que contengan cualquier comportamiento malicioso o defectuoso.

6.8.1. Tipado estático en TypeScript

TypeScript permite definir de forma rigurosa la firma de la función de validación que el usuario debe implementar. Por ejemplo, exigir una función

```
function validarCampo(valor: any, formulario: FormData): boolean { ... }
```

hace que el compilador advierta si se accede a propiedades inexistentes o si el retorno no es un booleano [88]. Al exponer solo las interfaces que representan los datos permitidos —por ejemplo, un 'FormData' limitado a campos autorizados— y excluir las definiciones del DOM en la configuración de tipos, se impide al usuario acceder directamente a 'window' o 'document' [88]. Esta capa de validación en tiempo de compilación mejora la confiabilidad del script, guía al desarrollador con autocompletado y detección temprana de errores.

6.8.2. Ejecución aislada en tiempo de ejecución

Iframe con `sandbox` Un `<iframe sandbox>` configura un entorno donde, incluso permitiendo scripts, se bloquean accesos al DOM de la página contenedora y funciones peligrosas, sobre todo si se sirve desde un origen distinto [87]. La comunicación con la página principal se realiza mediante `postMessage`, garantizando un intercambio explícito de datos y evitando accesos no autorizados.

Web Workers Un Web Worker corre en un hilo separado sin acceso al DOM ni a variables globales de la ventana principal, comunicándose exclusivamente por `postMessage`/`onmessage` [89]. Al crear un Blob URL con el código compilado (por ejemplo, `URL.createObjectURL(new Blob([code])`) se logra un "eval seguro" cuya ejecución puede abortarse con `worker.terminate()` en caso de errores o bucles infinitos [90].

6.8.3. Buenas prácticas y consideraciones

- Evitar simultanear `allow-scripts` y `allow-same-origin` en iframes, para no anular las restricciones de seguridad [87].
- En Web Workers, envolver la lógica del usuario en `try/catch` y manejar errores mediante eventos `onerror` o mensajes de fallo, de modo que no se quiebre la aplicación principal [89].
- Establecer tiempos máximos de ejecución y abortar workers que excedan un umbral razonable, previniendo denegaciones de servicio [90].
- Verificar la firma del fragmento de código antes de compilarlo (por ejemplo, con expresiones regulares o un parser ligero) para asegurar que corresponde a la función esperada.
- Para entornos multiusuario o críticos, considerar un lenguaje de dominio específico (DSL) en lugar de JavaScript completo, reduciendo la superficie de ataque.

La combinación de TypeScript como capa de restricción en tiempo de compilación y de sandboxing mediante iframes o Web Workers en tiempo de ejecución proporciona un esquema robusto para habilitar validaciones dinámicas con riesgo controlado. El tipado estático orienta al desarrollador y previene errores, mientras que el aislamiento en contextos separados protege la aplicación principal de comportamientos imprevistos o malintencionados [86], [89].

6.9. Autenticación basada en *Bearer tokens* y códigos QR

Los *Bearer tokens* constituyen un mecanismo ampliamente adoptado para autenticar solicitudes HTTP hacia recursos protegidos: basta la posesión del token para ejercer los privilegios asociados, razón por la cual su protección en tránsito y en reposo resulta crítica [91]. En el ámbito web y de APIs, estos tokens suelen emitirse bajo marcos de autorización como OAuth 2.0 2.0, donde un servidor de autorización concede a un *cliente* permisos delimitados para actuar ante un *recurso* [92]. Entre los formatos más difundidos se encuentran los *JSON Web Tokens* (JWT), definidos como un medio compacto y autocontenido para representar *claims* entre partes, con integridad garantizada mediante firma digital (JWS) o código de autenticación (MAC) y, opcionalmente, confidencialidad mediante cifrado (JWE) [93], [94]. Los algoritmos y parámetros interoperables se estandarizan en JWA, que especifica identificadores y semánticas para HMAC, RSA y ECDSA, entre otros [95]. En flujos modernos, extensiones como PKCE refuerzan la fase de autorización para clientes públicos mitigando el secuestro del *authorization code* [96].

6.9.1. Fundamentos de *Bearer* y JWT

En el esquema *Bearer*, el cliente adjunta el token en la cabecera `Authorization: Bearer <token>` en cada petición; el servidor valida la firma, la vigencia temporal y la audiencia antes

de autorizar el acceso [91]. Un JWT se compone de tres secciones codificadas *Base64URL*: *header*, *payload* y *signature*. El *header* declara el tipo (`typ=JWT`) y el algoritmo de firma (`alg`, p. ej., `HS256`, `RS256`); el *payload* transporta *claims* de identidad y control (`iss`, `sub`, `aud`, `exp`, `nbf`, `iat`) y, cuando aplica, `jti` para facilitar detección de reuso; y la *signature* prueba integridad y autenticidad con la clave compartida o la clave privada del emisor, verificable por el receptor [93], [94]. Este diseño posibilita autenticación *sin estado* en el servidor, ya que la verificación del token no requiere sesión persistente: alcanza con validar la firma, la expiración y la congruencia de *claims* (emisor/audiencia). A nivel criptográfico, JWA define parámetros y curvas aceptables (p. ej., `ES256` sobre P-256) y exige selección de algoritmos robustos frente a ataques conocidos [95]. En arquitecturas con múltiples emisores o rotación de claves, el encabezado `kid` permite identificar la clave concreta utilizada, facilitando *key rotation* segura y trazable [93], [94].

6.9.2. Ciclo de vida y buenas prácticas de seguridad

La robustez de un sistema basado en tokens descansa en la gestión de su ciclo de vida. En primer lugar, conviene diferenciar *access tokens* (corto plazo, alcance acotado) y *refresh tokens* (más longevos, resguardados con mayor rigor), aplicando rotación y revocación de *refresh* ante señales de riesgo [91], [92]. La validación estricta de `exp`/`nbf`, `aud` y `iss` evita usos fuera de contexto (confusión de audiencia) o más allá de la ventana temporal prevista [93]. Directrices de identidad digital recomiendan evidencias de presencia del usuario (cuando el nivel de aseguramiento lo exija), autenticadores con ciclo de vida administrado y resistencia a reprocesamiento (*replay*) mediante identificadores únicos y políticas de reemisión prudentes [97]. En el plano operativo, OWASP aconseja: transportar tokens exclusivamente sobre TLS; evitar incluirlos en URLs; deshabilitar algoritmos no esperados; verificar firmemente el `alg` anunciado; y no sobrecargar el *payload* con datos sensibles innecesarios [98], [99]. En clientes móviles, MASVS sugiere almacenamiento en keystores seguros y defensa contra extracción, así como validaciones de certificado/TLS *pinning* donde sea razonable [100]. Asimismo, la rotación de claves (apoyada en `kid`) y la segmentación por audiencias limitan el impacto de una eventual filtración [93], [94].

6.9.3. Autenticación mediante códigos QR

El inicio de sesión asistido por código QR se enmarca en experiencias *passwordless* y *out-of-band*: el punto de acceso (p. ej., un navegador) muestra un QR que el dispositivo confiable del usuario (típicamente su móvil) escanea para autorizar la sesión. Conceptualmente, este patrón puede modelarse como una variante visual del *Device Authorization Grant*, donde un dispositivo "limitado" inicia la transacción y otro, plenamente autenticado, la aprueba [101]. Para robustecer el esquema, el QR no transporta credenciales sino un desafío efímero que vincula de manera fuerte la solicitud con el servidor de autenticación. En términos prácticos, el contenido codifica: (i) un identificador de sesión impredecible (`sid`) asociado a un estado en servidor (*pending* $\rightarrow$ *claimed*/*expired*); (ii) un *nonce* de un solo uso para prevenir reprocesamientos; y (iii) una firma de integridad `sig` computada sobre campos críticos (p. ej., `sid.nonce`) empleando HMAC conforme a lo normado por FIPS 198-1, de modo que el servidor pueda detectar cualquier alteración o falsificación [102]. Adicionalmente, se impone un tiempo de vida estricto (TTL reducido) tras el cual la sesión caduca y cualquier

intento posterior es rechazado. El dispositivo autenticador, tras escanear, verifica localmente la legitimidad del dominio y confirma la operación (PIN/biometría según el nivel de aseguramiento), comunicándose directamente con el servidor por canal TLS para canjear el desafío por un *Bearer token* (p. ej., un JWT firmado) que materializa la sesión [93], [101].

6.9.4. Amenazas y controles en flujos QR

Los esquemas de *QR login* introducen amenazas específicas. Análisis recientes documentan vectores como secuestro de sesión por presentación maliciosa del QR, ventanas de validez excesivas, validaciones de dominio laxas o ausencia de confirmación explícita en el dispositivo [103]. OWASP describe además *QRLJacking*, donde un atacante induce al usuario a escanear un QR de sesión ajeno (o sobrepuesto) para capturar el acceso [104]. Entre las mitigaciones recomendadas se cuentan: (i) atar criptográficamente el QR al contexto (HMAC sobre `sid`/`nonce` y metadatos de emisión), (ii) TTL de minutos y uso estrictamente *single-use*, (iii) confirmación consciente en el dispositivo (biometría/PIN) y notificación fuera de banda, (iv) validación del origen y canal TLS, (v) estado transaccional en servidor que invalide de inmediato el desafío tras su canje, y (vi) registros/auditoría para trazabilidad y detección de anomalías [102]-[104]. Una vez emitido el token, aplican las mismas salvaguardas del ciclo de vida JWT: expiraciones cortas, verificación de audiencia, almacenamiento seguro del *access token* y rotación controlada de *refresh tokens* [91], [93], [98], [100].

6.10. Pruebas de desempeño en APIs, métricas y control de colas

Las pruebas de desempeño en APIs caracterizan capacidad, tiempos de respuesta y resiliencia bajo diferentes regímenes de carga. En ingeniería de confiabilidad (SRE), un marco clásico para observar la salud de un servicio son las *cuatro señales doradas*: **latencia**, **tráfico**, **errores** y **saturación**, que se emplean para definir indicadores (SLI) y objetivos (SLO) alineados a expectativas de usuario y negocio [105]-[107]. En este contexto, los percentiles de latencia (p95, p99) son preferibles a promedios para detectar degradación y crecimiento de colas; la literatura resalta que la *latencia de cola* (*tail latency*) suele dominar la experiencia agregada [108].

6.10.1. Métricas fundamentales e instrumentación

- **Latencia** (p50/p95/p99): distribución de tiempos por endpoint; los percentiles altos revelan colas y saturación con mayor sensibilidad que el promedio [105], [108].
- **Rendimiento/Throughput** (Solicitudes por Segundo (RPS)): solicitudes por segundo o por unidad de tiempo, idealmente a una tasa objetivo conocida [107].
- **Tasa de error**: fracción de respuestas fallidas (p. ej., HTTP 5xx) y errores semánticos relevantes [105].

- **Saturación**: utilización de CPU/memoria, conexiones activas y *queue depth*, como señales de cercanía a límites físicos o lógicos [105].

- **Histogramas de latencia** (Prometheus/OpenTelemetry): recomendados para cuantiles agregables en clúster; seguir convenciones semánticas HTTP [109], [110].

- **Índice Apdex**: métrica sintética (0–1) basada en umbrales de tiempo de respuesta (T, $4T$) para resumir satisfacción de usuario [111].

- **Sesgo *coordinated omission***: si el generador de carga espera la respuesta para emitir la siguiente, los periodos de pausa quedan submuestreados y la cola "desaparece" del muestreo; se mitiga usando modelos/tasas de llegada abiertas [108].

6.10.2. Modelos de carga: cerrado vs. abierto

Un **modelo cerrado** fija usuarios concurrentes y *think time*; cada usuario espera su respuesta antes de la siguiente solicitud. Un **modelo abierto** controla la *tasa de llegadas* (λ), emitiendo nuevas solicitudes a razón fija e independiente de los tiempos de servicio. El modelo abierto evita gran parte del sesgo por *coordinated omission* y es útil cuando se comparan versiones a *igual tasa* [112], [113]. Herramientas modernas permiten ambos enfoques: p. ej., ejecutores de *arrival-rate* (modelo abierto) o temporizadores de *throughput* constante (modelo cerrado) [113], [114].

6.10.3. Herramientas para carga/estrés en APIs

Locust modela comportamientos en Python, soporta ejecución distribuida (maestro–trabajadores) y exporta métricas (RPS, percentiles, errores) en tiempo real [115]. Ofrece *pacing*/*constant_throughput* para aproximar tasas objetivo, y *custom load shapes* para rampas, picos (*spikes*) o pruebas de resistencia (*soak*) [116], [117]. Otros ecosistemas incorporan ejecutores de tasa de llegada fija (p. ej., *constant-arrival-rate*) [113] o temporizadores de *throughput* constante [114]. Para *benchmarks* a throughput constante y registro de latencia corregida, existen generadores específicos [118].

6.10.4. Control de colas, *backpressure* y *rate limiting*

La **Ley de Little** establece que, en régimen estable, el número promedio de solicitudes en el sistema L es igual a la tasa de llegadas λ por el tiempo promedio en el sistema W: $L = \lambda W$. Es independiente de las distribuciones de llegada y servicio y resulta muy útil para estimar concurrencia y tamaños de cola [119]. En arquitecturas reactivas, el *backpressure* permite que consumidores más lentos regulen a productores más rápidos para acotar colas/buffers; la especificación *Reactive Streams* estandariza este mecanismo [120]. A nivel de *gateway*, el *rate limiting* suele implementarse con *token bucket* y marcadores tricolores (IETF RFC 2697/2698) para regular ráfagas y tasas sostenidas por flujo/tenant [121], [122].

6.10.5. Tipos de prueba de desempeño

Además de la **carga** (volumen esperado), conviene orquestar: **estrés** (por encima de lo nominal para ubicar el punto de quiebre), **spike** (picos súbitos), **soak/endurance** (carga sostenida para detectar fugas o degradación) y **breakpoint** (incremental hasta fallo). La documentación actual de herramientas generaliza estos perfiles y su configuración [123].

6.10.6. Calidad funcional previa: Jest (unitarias/integración/E2E)

Antes de ejercer carga, es recomendable asegurar corrección funcional con pruebas unitarias, de integración y E2E en CI/CD . En entornos TypeScript/NestJS, **Jest** ofrece guía y soporte integrados para unitarias y E2E (comúnmente con *Supertest*) [124]-[126]. Esto reduce ruido en las mediciones de desempeño al probar sobre un sistema estable.

6.10.7. Diseño de experimentos e interpretación

Se proponen las siguientes pautas: (i) definir SLI/SLO (p. ej., p95 $<$ 200 ms, error $<$ 0.5 %); (ii) elegir modelo de carga (cerrado vs. abierto) y perfilar rampas/picos/soak; (iii) instrumentar servidor con histogramas y convenciones HTTP [109], [110]; (iv) estimar concurrencia y colas con la Ley de Little; y (v) leer reportes priorizando p95/p99 por endpoint , RPS objetivo vs. alcanzado, tasa de error y señales de saturación [105], [107].

CAPÍTULO 7

Metodología

Esta sección describe el enfoque técnico y organizativo seguido para diseñar, implementar y validar una solución de captura de datos en campo para un ingenio azucarero, caracterizado por *operación distribuida, conectividad irregular y restricciones de dispositivo*. Partimos de los principios establecidos en el marco teórico —en particular, *diseño* offline-first , *arquitecturas modulares* y *calidad de servicio basada en idempotencia y validaciones tempranas*— para justificar cada decisión tecnológica y de proceso. De ese modo, la solución no sólo atiende los requisitos funcionales (formularios adaptables, catálogos, sesiones de llenado y envío), sino que también evidencia *robustez operacional*, *mantenibilidad* y *trazabilidad* desde la captura hasta el almacenamiento definitivo.

7.1. Enfoque metodológico y alineación con el marco teórico

El enfoque se fundamenta en cuatro directrices, derivadas del marco teórico y de buenas prácticas de ingeniería de software para sistemas móviles en campo:

1. **Priorizar el offline-first** : la aplicación debe ser plenamente utilizable sin conectividad, con consistencia local y sincronización diferida. Esto reduce la dependencia de la red y mejora la resiliencia operacional.

2. **Aislar dominios y contratos**: separar responsabilidades (autenticación, catálogo de formularios, grupos y entradas) favorece la prueba unitaria, la extensibilidad y la gobernanza del cambio.

3. **Minimizar el acoplamiento cliente–servidor**: los clientes envían *datos mínimos suficientes* (tipos simples) y el servidor preserva la foto de estructura para auditoría sin forzar migraciones frecuentes.

4. **Observabilidad y control del riesgo**: idempotencia , límites de tamaño, tiempos de espera acotados, *correlation-id* y métricas operativas habilitan diagnóstico en terreno y prevención de fallos repetidos.

Estas directrices guían la selección tecnológica y la arquitectura descrita a continuación, y explican por qué el enfoque propuesto es adecuado para el **sector agrícola** (con jornadas en campo, múltiples cuadrillas y heterogeneidad de dispositivos).

7.2. Panorama y arquitectura de referencia

La solución se apoya en una arquitectura *práctica* para capturar datos en campo con conectividad irregular (Figura 4). El cliente utiliza **React Native con TypeScript** para compartir base de interfaz y lógica en Android/iOS, manteniendo un *tipado estático* que reduce errores al renderizar formularios y transformar valores. Ese tipado acompaña a un *motor de formularios* que interpreta la estructura enviada por el backend (categorías, formularios, páginas y campos), normaliza y valida por tipo, y admite *campos calculados* y *grupos repetibles*. La orquestación se concentra en **Redux**: cada sesión conserva estado (página actual, valores por página/campo, errores, grupos cacheados y el `status`) de forma predecible y depurable. La persistencia local se realiza en **SQLite** por estabilidad y rendimiento en dispositivos móviles; allí se cachea el árbol de formularios y catálogo/dataset s, además de las sesiones locales en curso.

En el backend se eligió **NestJS** por su modularidad y su modelo de inyección de dependencias, lo cual facilita separar dominios: *Auth*, *Forms* y *Groups*. Esta segmentación clarifica contratos y permite pruebas y despliegues graduales. *Swagger* documenta automáticamente las rutas y DTO s, ofreciendo una referencia viva para el cliente móvil y QA. El control de acceso se aplica con **JWT** y *guards* por ruta; asimismo, se contempla el caso de formularios públicos donde ciertas lecturas pueden ser anónimas. La comunicación es deliberadamente simple (HTTP+JSON), lo que favorece el *debug* en terreno y reduce superficie de falla.

Criterios de elección tecnológica Con base en los principios de *portabilidad*, *consistencia local ACID* y *bajo acoplamiento* presentados en el marco teórico, se priorizaron herramientas con comunidad amplia, documentación vigente y ecosistemas maduros. **React Native** reduce el costo de mantenimiento al evitar dos bases de código; **TypeScript** agrega contratos estáticos que mitigan errores en transformaciones y normalizadores; **Redux** aporta trazabilidad y reproducibilidad de estados; **SQLite** ofrece baja latencia, transacciones locales confiables y cero dependencias externas para offline-first . En el servidor, **NestJS** ordena el código por dominios y promueve inyección controlada de dependencias; **PostgreSQL** aporta transacciones sólidas y `JSONB` como contenedor de capturas heterogéneas. Se descartó *sincronización automática de fondo* en el cliente (por consumo de batería y riesgos de carrera) y se descartó un esquema *estrictamente relacional* para las capturas (demandaría migraciones invasivas ante cada cambio menor en formularios).

Supuestos y restricciones operativas. Se asume pérdida de conectividad, cierres inesperados de la app y reintentos múltiples. Por ello cada sesión se guarda inmediatamente, el

cursor de página se conserva y los envíos son idempotentes. En el canal móvil se fija un tamaño razonable para `fill_json` y para la *firma* en Base64; el cliente no adjunta metadatos de catálogos en las capturas para mantener paquetes livianos. En el backend se establece un tiempo de espera corto en controladores y límites explícitos de carga, con mensajes de error concisos y accionables.

Flujo extremo a extremo. El flujo inicia con autenticación, continúa con la descarga del árbol y catálogo/dataset s, y luego se abre una sesión de formulario (Figura 5). Cada cambio se valida y normaliza en el cliente, se recalculan dependencias y se persiste en SQLite (incluido el cursor). Cuando el formulario está listo, la app envía un payload idempotente con `form_id`, `index_version_id`, `filled_at_local`, `status`, `fill_json` y `form_json`. El backend valida, inserta/actualiza en `formularios_entry` y responde con un acuse para marcar `synced`.

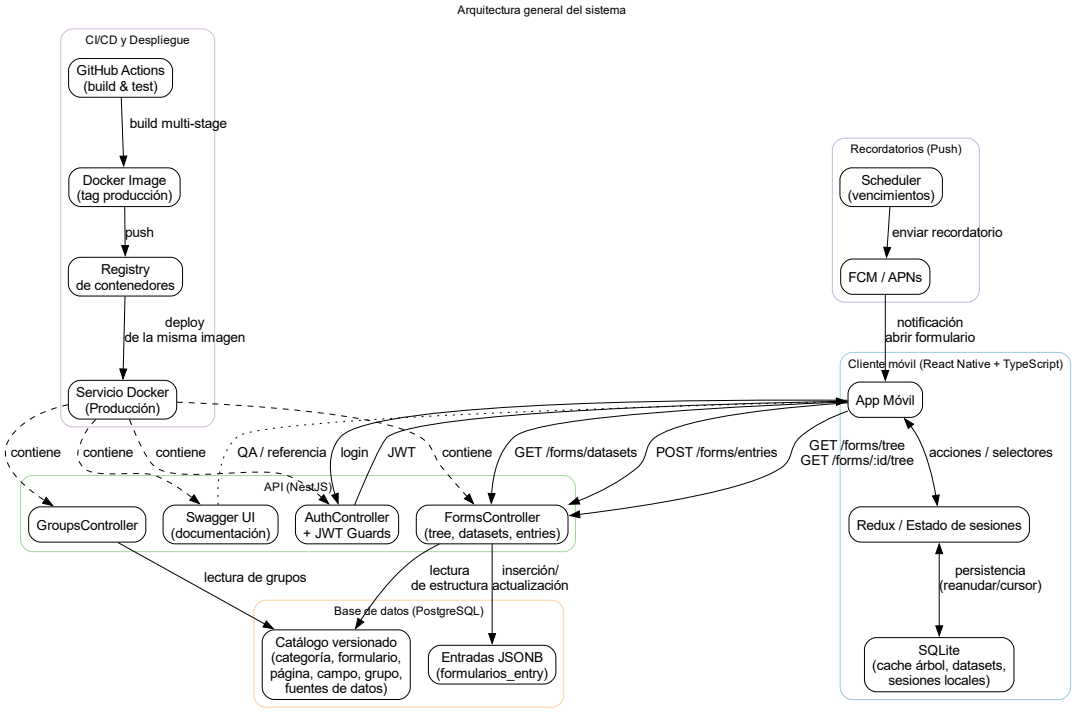


Figura 4: Arquitectura general del sistema.

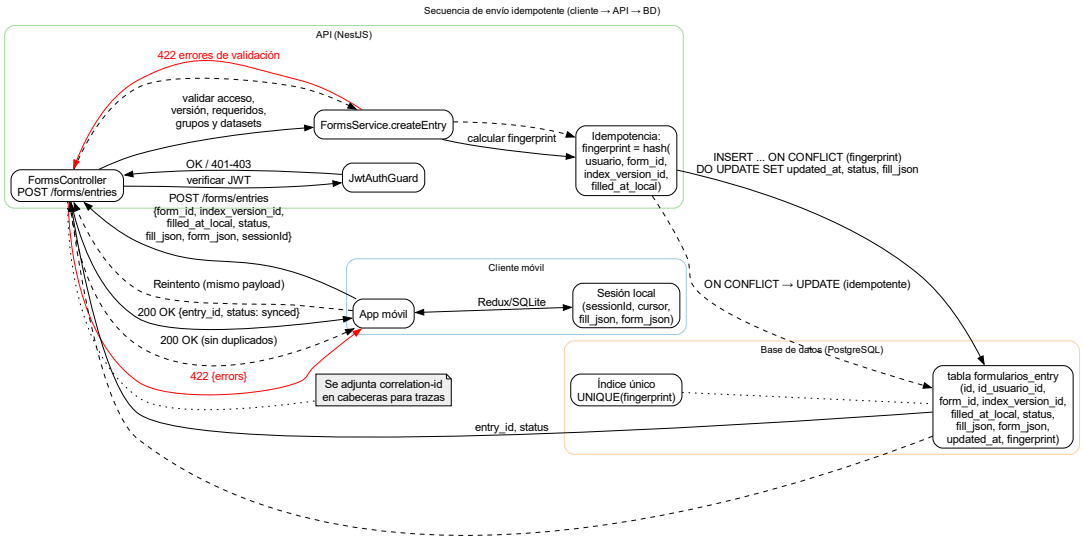


Figura 5: Secuencia de envío idempotente (cliente → API → BD).

7.3. Backend y contratos de API

El backend expone contratos concisos: *(i)* `POST /auth/login` para autenticación (JWT); *(ii)* `GET /forms/tree` y `GET /forms/:id/tree` para el árbol filtrado por rol (categoría → formulario → páginas → campos); *(iii)* `GET /forms/datasets` y `GET /forms/:id/datasets` para datasets visibles (con un *switch* de diagnóstico `?notFoundWhenEmpty=true`); y *(iv)* `POST /forms/entries` para la creación idempotente de entradas. Los DTOs emplean `snake_case` coherente con la base de datos y evitan fugas de implementación hacia el cliente.

Idempotencia y validaciones mínimas. En `POST /forms/entries` el cliente envía `form_id`, `index_version_id`, `filled_at_local`, `status`, `fill_json` y `form_json`. El servidor autoriza por JWT y revisa acceso, confirma que `index_version_id` pertenece al formulario, calcula un *fingerprint* estable (usuario, formulario, versión, tiempo local) para insertar o actualizar sin duplicar, y aplica validaciones mínimas: requeridos, grupos con al menos una fila y, cuando corresponda, que las claves provengan de datasets visibles.[1] Esta estrategia, apoyada en el marco teórico de *operaciones idempotentes en interfaces REST*, permite tolerar reintentos y cortes de red sin efectos adversos.

Errores, diagnósticos y observabilidad. Se devuelve 401 (token inválido), 403 (sin acceso al formulario), 404 (diagnóstico con `?notFoundWhenEmpty=true`) y 422 (requeridos o formatos mínimos). Se registra un *correlation-id* en cabeceras y *logs* para rastreo en soporte. La instrumentación mínima incluye métricas de tasa de envíos, errores por tipo y latencia del endpoint de entradas; estas métricas respaldan decisiones operativas y SLA internos.

Notificaciones *push* (recordatorios). El backend mantiene un *scheduler* que calcula *próximos vencimientos* por formulario recurrente y envía *push sólo* como recordatorio de llenado. El flujo es deliberadamente simple: el móvil registra su *device token* (FCM/APNs), el backend almacena el vínculo usuario↔dispositivo, y al aproximarse la fecha se emite una notificación con el identificador del formulario. La apertura profundiza en la pantalla correspondiente, sin afectar la lógica de captura ni el modelo de datos (cero efectos colaterales).

[1] Los campos se almacenan como tipos simples; la firma como Base64; los grupos como objetos de primitivos.

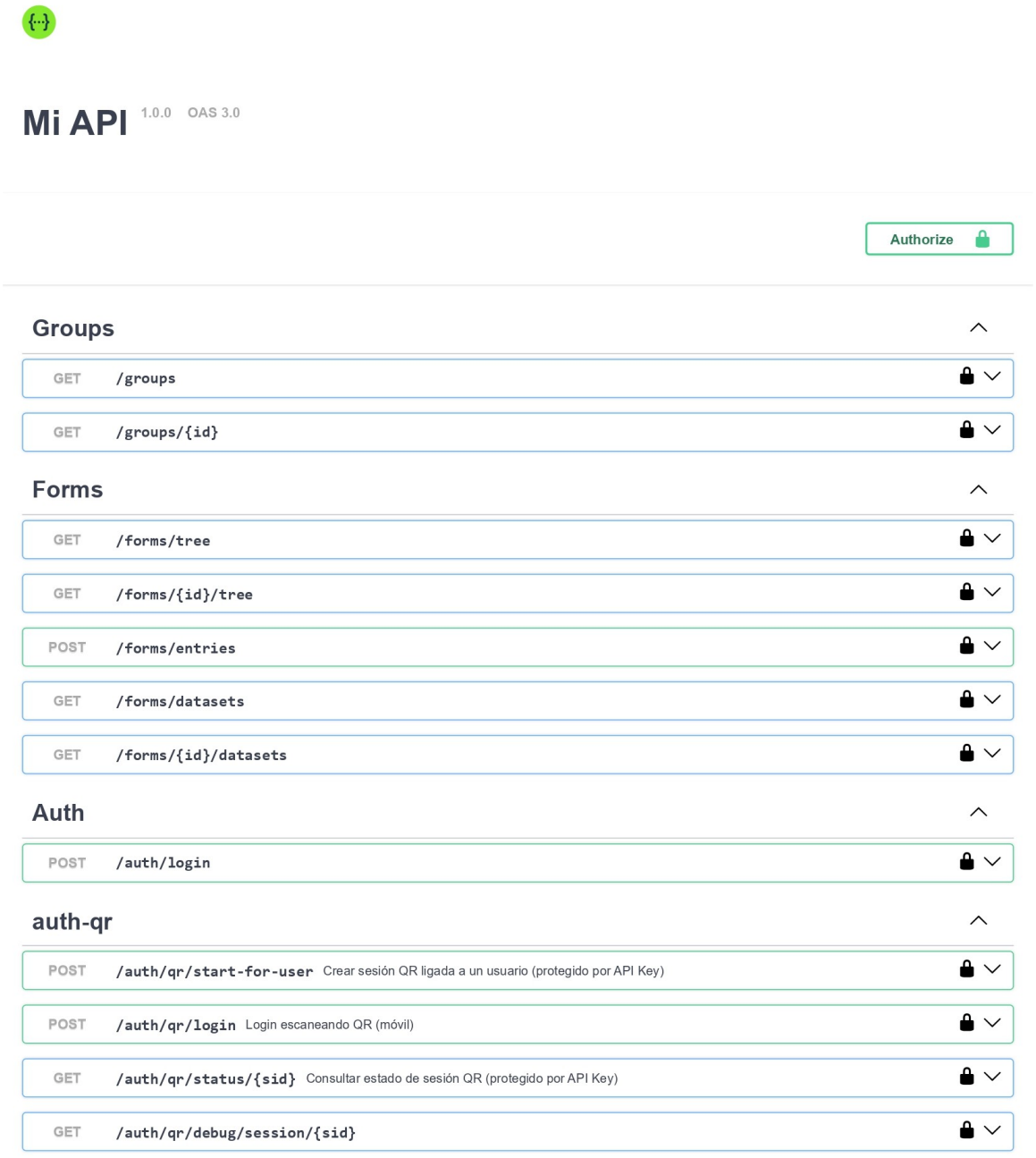


Figura 6: Contratos publicados en Swagger.

Listing 7.1: Evidencia: fragmento del controlador de formularios

```
// POST /forms/entries
@Post('entries')
async createEntry(
  @Body() dto: CreateFormEntryDto,
  @AuthUser() user: TypeAuthUser,
) {
  return this.service.createEntry(dto, user);
}

// ——— DATASETS: TODOS los datasets visibles para el usuario
// GET /forms/datasets
@Get('datasets')
async getUserDatasets(@AuthUser() user: TypeAuthUser) {
  return this.service.getUserDatasetsAsTables(user);
}
```

7.4. Cliente móvil y operación offline-first

El cliente organiza el llenado en **sesiones de formulario**. Al iniciar, se crea un `sessionId`, se pre-inicializa cada campo con su valor vacío correcto (según tipo/clase) y se recalculan los campos dependientes antes de mostrar la primera página. Ese estado inicial se persiste en **SQLite**, junto con metadatos operativos (`form_id`, `index_version_id`, `filled_at_local`, cursor de página).

Cada asignación de valor pasa por *normalizadores* y *validadores* por tipo/clase; se recalculan los campos dependientes y se marca la página como válida sólo si todos los requeridos pasan sus reglas. La sesión mantiene un `status` global (`pending`/`ready_to_submit`) coherente con la UI para no inducir envíos anticipados. Los **grupos repetibles** se representan como arreglos de filas con identificadores estables; al guardar se eliminan filas vacías. Los **datasets** se mantienen en caché local con búsqueda *case-insensitive*.

Experiencia de usuario en formularios extensos. Se incorpora un indicador de progreso por páginas, mensajes de error junto al campo y persistencia del cursor para retomar exactamente donde se quedó, incluso si el SO cerró la app. Estas prácticas, recomendadas en el marco teórico de *diseño centrado en el usuario* para tareas prolongadas, reducen la fricción durante el llenado.

Manejo de medios y tamaños. La **firma** se captura, se convierte a Base64 y se incluye como texto. Se definen tamaños máximos razonables para prevenir bloqueos y para garantizar tiempos de sincronización adecuados cuando la conectividad está disponible. Los demás campos sólo admiten *tipos simples*.

Accesibilidad y localización. Se utilizan etiquetas accesibles, tamaños de fuente escalables y formatos locales cuando son pertinentes. Esto es especialmente relevante en operaciones de campo, donde las condiciones de luz, el uso de guantes o la intermitencia de la atención del usuario exigen interfaces robustas.

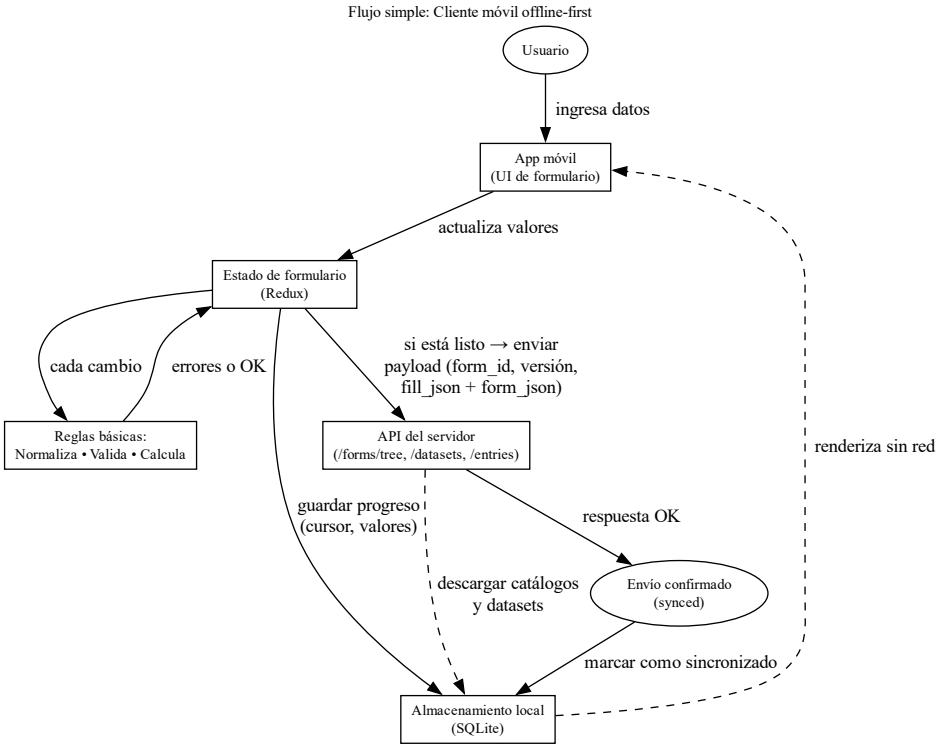


Figura 7: Flujo del cliente móvil offline-first .

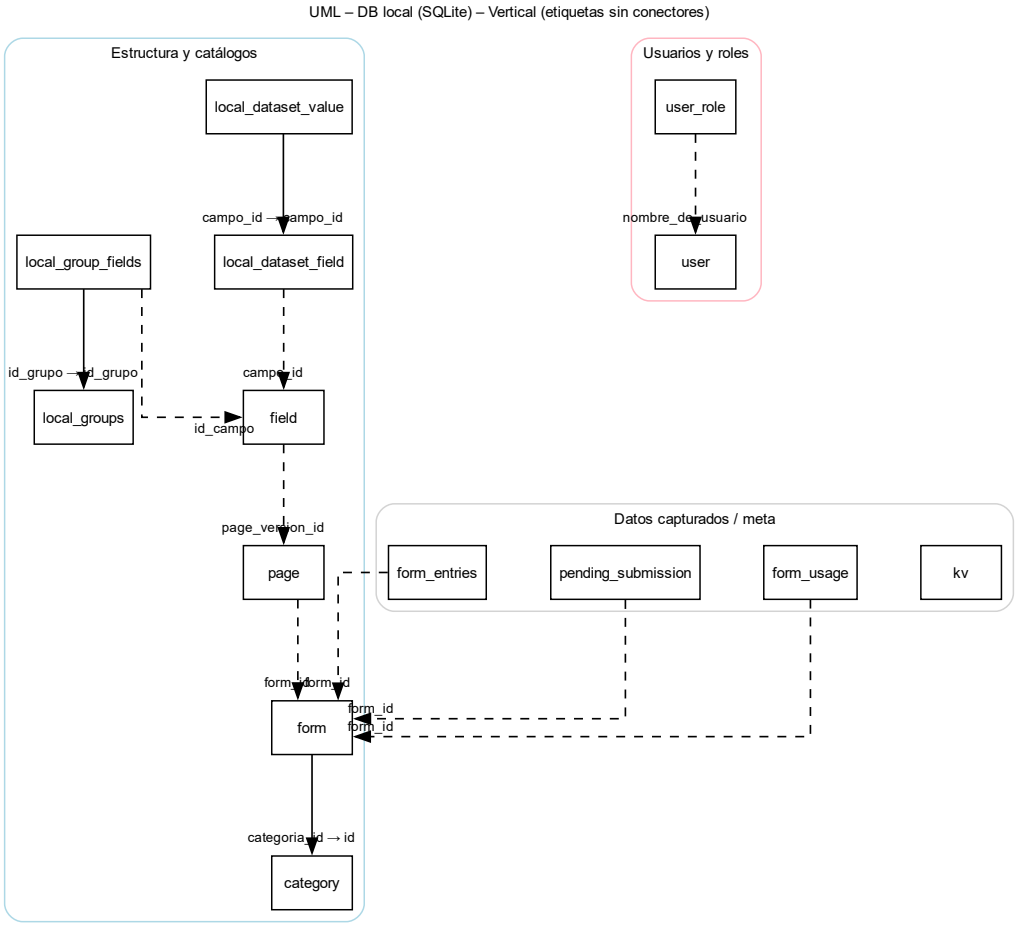


Figura 8: Esquema local SQLite y relaciones mínimas.

7.5. Almacenamiento con *tipos simples* y versionado operativo

En `formularios_entry` sólo se guardan **tipos simples**: texto, numéricos y booleanos. La **firma** es la única excepción y se almacena como *string Base64*. Si un campo es de *catálogo/dataset* , en `fill_json` queda únicamente el *valor primitivo* elegido (id/código), sin metadatos ni rótulos. Los **grupos** se representan como objetos (o arreglos de objetos) con atributos primitivos; funcionan como sub-páginas autocontenidas.

Cada entrada incluye `fill_json` (valor mínimo) y `form_json` (la "foto" de la estructura y su `index_version_id`). La idempotencia se implementa con un identificador estable (usuario, formulario, versión, tiempo local), de modo que reintentos por conectividad deficiente *actualizan* la fila en lugar de duplicarla. El `status` acompaña el ciclo del cliente (`pending`,

`ready_to_submit`, `synced`).

Convenciones y compatibilidad. El catálogo se mantiene **relacional y versionado** (formularios, páginas, campos, grupos y fuentes de datos), mientras que `formularios_entry` centraliza capturas heterogéneas sin exigir migraciones ante nuevos campos. Se emplean convenciones (`snake_case` y prefijos de dominio) y UUID donde conviene. Cambios en la naturaleza de un campo se resuelven con un `nombre_interno` nuevo; la versión controla la convivencia.

Integridad y límites. El servidor verifica que `fill_json` sólo contenga primitivos (y grupos como objetos de primitivos) y que la *firma* sea Base64 válida. Se fijan límites de tamaño y tiempos de espera. El contenedor JSONB transporta la captura y permite validar invariantes sin adjuntar metadatos de catálogos. El *formateo y exportación relacional* se delega a un módulo especializado de otro backend, fuera del alcance de esta metodología, alineado con el principio de *separación de intereses*.

Despliegue y operaciones (Docker + GitHub Actions). El despliegue se realiza mediante un **servicio basado en imagen de Docker** . El repositorio incluye un flujo de **GitHub Actions** que, ante un *tag* de versión para producción, construye la imagen de la API (con *multi-stage build*), ejecuta pruebas, adjunta metadatos de versión (`LABEL`, `BUILD_DATE`, `GIT_SHA`) y publica la imagen en el registro. La *misma* imagen es la que se **despliega en producción**, garantizando reproducibilidad y control de configuración (*twelve-factor*). La configuración (credenciales, URLs, CORS, claves JWT) se inyecta por variables en tiempo de ejecución. Las migraciones del catálogo son *forward-only* y se ejecutan al iniciar el contenedor (o mediante un *job* previo). Se mantienen entornos de *staging* y *producción* con separación clara de credenciales y orígenes permitidos.

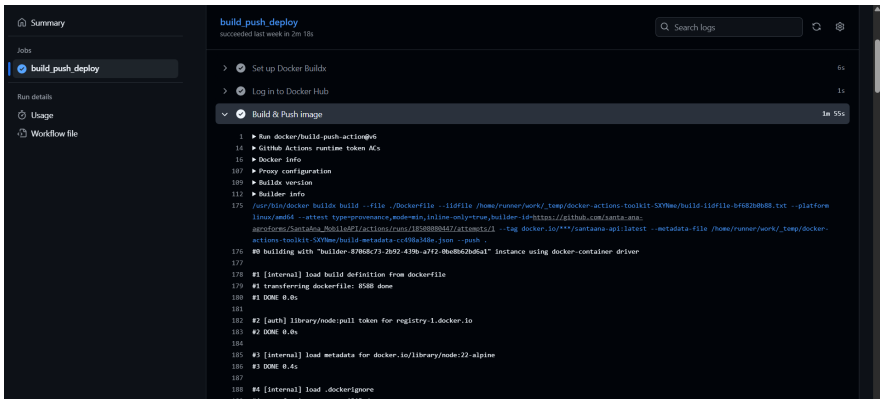


Figura 9: Pipeline de CI/CD en GitHub Actions

Evidencia de la ejecución del pipeline de CI/CD y despliegue de la imagen Docker .

Listing 7.2: Evidencia: Dockerfile multi-stage

```
# ---------- Stage 1: build ----------
FROM node:22-alpine AS builder
WORKDIR /app
RUN apk add --no-cache python3 make g++ curl libc6-compat
# libc6-compat ayuda a sharp
COPY package*.json ./
RUN npm install
COPY tsconfig*.json nest-cli.json ./
COPY src ./src
COPY assets ./assets
COPY test/fixtures ./test/fixtures
# Para que exista en build
RUN npm run build
RUN npm prune --omit=dev

# ---------- Stage 2: runtime ----------
FROM node:22-alpine AS runner
WORKDIR /app
ENV NODE_ENV=production
ENV FIXTURES_DIR=/app/dist/fixtures
COPY package*.json ./
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/assets ./assets
COPY --from=builder /app/test/fixtures ./dist/fixtures

# Copia los fixtures al runtime
COPY --from=builder /app/test/fixtures ./test/fixtures
COPY --from=builder /app/test/fixtures ./fixtures
```

```
# Llevar assets al runtime
RUN addgroup -S app && adduser -S app -G app
USER app
ARG PORT=3000
ENV PORT=${PORT}
EXPOSE ${PORT}
CMD ["node", "dist/main.js"]
```

7.6. Pruebas del API

El diseño metodológico de las pruebas para la capa de API está alineado con los principios del marco teórico: *contratos bien definidos*, *separación de intereses* y *resiliencia ante fallos*. El objetivo es verificar de forma progresiva (de *adentro hacia afuera*) que las reglas de negocio, los contratos HTTP y la integración con la base de datos se comporten de manera consistente, determinista y verificable bajo condiciones realistas de operación en campo.

7.6.1. Enfoque por niveles.

Las pruebas se estructuran en tres niveles complementarios, con una cuarta línea de prueba de carga automatizada:

1. **Unidad (servicios de dominio)**: validan funciones puras y métodos de servicios que transforman y agrupan datos, aplican reglas de ordenamiento y enriquecen estructuras para el cliente. En este nivel *no* se toca infraestructura real; se sustituyen dependencias por dobles de prueba (p. ej., `DataSource.query` simulado).
2. **Unidad ligera de controladores**: aseguran que los controladores delegan correctamente en sus servicios, respetan el contrato de datos y no introducen lógica de negocio.
3. **Extremo a extremo (E2E)**: ejercitan la API completa con base de datos real en entorno efímero (contenedores), autenticación JWT y *fixture* s semilla; verifican contratos, visibilidad por rol y coherencia de estructuras entregadas al cliente.
4. **Carga (headless)**: escenarios reproducibles con Locust para someter los endpoint s críticos a concurrencia sostenida; su objetivo es metodológico (dimensionamiento y estabilidad), sin presentar aquí resultados.

7.6.2. Alcance y mapeo de riesgos.

El diseño de casos responde a riesgos específicos del dominio:

- **Estructuras de formularios inconsistentes** → *Unidad (FormsService)*: se comprueba el agrupamiento *flat→tree*, el ordenamiento por página/secuencia y la post-compilación de campos calculados. Para evitar costos de compilación real, se *mockea* el compilador TS subyacente, garantizando determinismo.

- **Normalización de catálogos/catálogo/dataset s** → *Unidad (FormsService)*: se verifica la deserialización de filas (`JSON` en columnas), el conteo total y la forma tabular consumible por el cliente.
- **Autenticación y autorización** → *Unidad (AuthService)*: se ejercitan rutas felices y de error (usuario inexistente, inactivo, credenciales inválidas), con utilidades criptográficas y firma JWT *mockeadas* para aislar la lógica.
- **Ordenamiento y composición de grupos de campos** → *Unidad (GroupsService)*: se valida el orden por secuencia de página y de campo, además del parseo de configuraciones y requeridos.
- **Deriva de contrato en controladores** → *Unidad ligera (Controllers)*: se comprueba *delegación pura* y forma del retorno esperado.
- **Integración completa y visibilidad por rol** → *E2E* : se levanta la pila (PostgreSQL +API), se cargan *fixture* s, se autentica un usuario de prueba y se valida que el árbol de formularios responda a las reglas de visibilidad y que los campos calculados lleguen *postprocesados* para el cliente.

7.6.3. Aislamiento y dobles de prueba.

Las pruebas de unidad sustituyen todas las dependencias externas para asegurar ejecución rápida y determinista:

- **Base de datos**: el `DataSource` de TypeORM se sustituye por un objeto con `query` simulado, controlando las filas devueltas en cada escenario.
- **Compilación de cálculos**: se define un *mock* de `@swc/core` para que las operaciones declarativas de campos `calc` queden como código compilado canónico, sin dependencias de compilación nativa.
- **Criptografía y tokens**: el verificador Argon2 y la emisión JWT se reemplazan por implementaciones simuladas; se fijan variables de entorno (`JWT_EXPIRES_IN_SECONDS`) para reproducibilidad y se restauran al finalizar cada caso.

7.6.4. Datos de prueba y *fixture* s.

En E2E se emplean *fixture* s versionados que pueblan catálogos, usuarios y formularios de ejemplo. La siembra se ejecuta tras un *healthcheck* de la API para evitar condiciones de carrera. Se mantiene un usuario de prueba con credenciales conocidas para automatizar el inicio de sesión y la obtención del token.

7.6.5. Configuración de los *runners*.

Las suites usan Jest con TypeScript (`ts-jest`) y entorno `node`. Para E2E , la configuración delimita los archivos `*.e2e-spec.ts`, establece inicialización común (sembrado/cierre)

y un `timeout` elevado para sincronizar con la creación de tablas y migraciones. En unidad, los módulos de prueba de Nest (`TestingModule`) inyectan servicios y repositorios simulados, con `clearAllMocks()` entre casos.

7.6.6. Orquestación efímera para E2E .

La pila de E2E se levanta con `docker-compose`: un PostgreSQL con *healthcheck*, la API en modo `test` que aplica migraciones al iniciar y un contenedor de siembra que carga *fixture s* una vez que la API está saludable. Los *healthchecks* incluyen una ruta `/health` con tiempo de gracia (`start_period`) para evitar falsos negativos. Las variables de entorno delimitan el `DATABASE_URL`, el secreto JWT y claves de prueba, acotando cualquier efecto colateral a la red de contenedores.

7.6.7. Criterios de verificación

Cada caso define condiciones de éxito observables y estables, por ejemplo: ordenamiento por secuencia, presencia de campos calculados compilados, forma de los DTO s expuestos por controladores, códigos de estado bajo autenticación válida y estructura de catálogos normalizada. En carga, los escenarios definen usuarios concurrentes, rampa, duración e informes (`HTML/CSV`) como evidencia de ejecución, pospuestos para la sección de resultados.

7.6.8. Razonamiento metodológico.

Esta estratificación minimiza el costo de diagnóstico: los defectos lógicos se capturan en *unidad*, los desacoples de contrato en *controladores* y las divergencias de infraestructura en *E2E* . La incorporación de una línea de *carga* orienta el dimensionamiento inicial y otorga trazabilidad de la estabilidad temporal de los endpoint s más críticos, sin anticipar conclusiones cuantitativas en esta etapa.

7.7. Pruebas en el dispositivo móvil

Con el propósito de validar la robustez operativa del cliente bajo el enfoque offline-first , se implementó una pantalla de diagnóstico accesible exclusivamente en *modo QA* dentro de la aplicación. Esta pantalla orquesta pruebas *in situ* que ejercitan tanto la capa de persistencia local (SQLite) como la integridad del estado de la interfaz (Redux), sin librerías nativas adicionales ni tráfico de red. El diseño responde a tres principios del marco teórico: (*i*) **aislamiento** de riesgos en el dispositivo, (*ii*) **mediciones reproducibles** y exportables, y (*iii*) **no intrusión** sobre datos de producción.

7.7.1. Instrumentación y alcance.

La pantalla de QA expone una *suite* de pasos ejecutables con un toque, presenta progreso y resultados detallados por caso, y permite *exportar* evidencia como JSON. Las pruebas abarcan: (a) **SQLite**, para verificar ACID local, sensibilidad al modo de diario y efectos de transacciones sobre el rendimiento; y (b) **Redux**, para asegurar propiedades de pureza y estabilidad del *reducer* de sesiones de formulario. Todas las operaciones son locales y efímeras; al finalizar, el ambiente vuelve a un estado limpio.

7.7.2. Pruebas sobre SQLite (persistencia local).

Las pruebas usan `expo-sqlite` sobre la base `forms.db`, habilitando `PRAGMA foreign_keys=ON` y `journal_mode=WAL` para simular el modo operativo real. Se emplean tablas diagnósticas (`diag_bench`, `diag_meta`) que se crean si no existen y se vacían al inicio/final. Los casos son:

- **Preparación de esquema**: garantiza la existencia de tablas diagnósticas y su vaciado inicial.
- **Atomicidad de transacciones**: inserta una serie de filas y fuerza un error a mitad de una `BEGIN;...;COMMIT`. El criterio de éxito es *rollback íntegro*: el conteo posterior debe ser cero (consistencia local).
- **Rendimiento por lotes**: compara inserciones *sin transacción* vs *dentro de una transacción* para un mismo *batch* (N filas). Se espera menor latencia con transacción al reducir *fsync*/I/O por operación.
- **Concurrencia en WAL**: mantiene un escritor secuencial y un lector que realiza *polling* concurrente. Se registran lecturas totales y eventos `SQLITE_BUSY`. Bajo WAL, los lectores no suelen bloquear al escritor, por lo que el criterio de éxito es alcanzar el número de filas previstas y contabilizar, si los hay, bloqueos transitorios.
- **Limpieza final**: vaciado de `diag_*` y `wal_checkpoint(TRUNCATE)` para liberar archivos auxiliares.

Cada caso captura *duración* (ms) y métricas asociadas (p. ej., `sin_txn_ms`/`con_txn_ms`, `SQLITE_BUSY`, total de lecturas). El *payload* exportado incluye zona horaria del dispositivo y marca de tiempo ISO, almacenándose también en `diag_meta` para auditoría local.

7.7.3. Pruebas sobre Redux (estado de sesión de formularios).

Se valida el *reducer* de sesiones (`@/forms/state/formSessionSlice`) sin dependencias externas. Los casos comprueban *propiedades algebraicas* esperables en un *store* bien diseñado:

- **Estado inicial definido y serializable**: el *reducer* debe devolver un estado inicial válido y serializable a JSON.

- **Acción desconocida no muta**: ante un tipo de acción no reconocido, el estado debe permanecer *referencialmente* idéntico ($next \equiv prev$) o, en su defecto, estructuralmente invariante (para evitar recomputaciones innecesarias).
- **Determinismo**: misma entrada produce misma salida; se evalúa con acciones *noop* repetidas.
- **Serializabilidad del árbol de estado**: se inspecciona la presencia de estructuras no serializables (p. ej., `Map`/`Set`/funciones); `Date`/`RegExp` se aceptan si el contrato de la aplicación lo define explícitamente.
- **Integración con `configureStore`**: se crea un *store* real y se verifica estabilidad ante acciones desconocidas.
- **Rendimiento *noop***: se ejecuta un número configurable de acciones triviales para estimar latencia promedio ($\mu s/accin$) de la ruta de *renderizado* en reposo.

Cuando el módulo del *reducer* no está presente, la pantalla reporta el motivo y omite los casos, preservando la *observabilidad* sin falsos fallos.

7.7.4. Exportación de evidencia y no intrusión.

Los resultados se pueden exportar vía el mecanismo estándar de `Share` de React Native (sin dependencias nativas adicionales) y se guardan en `diag_meta` (`key`=`last_results`). La pantalla no toca tablas de catálogo ni sesiones reales de usuario; su alcance está acotado a `diag_*` y a introspección de estado en memoria, en línea con el principio de *no intrusión* y con el carácter *QA-only*.

7.7.5. Supuestos, límites y reproducibilidad.

Las mediciones dependen del modelo de dispositivo, temperatura de la CPU y carga del sistema operativo; por tanto, se reportan *como evidencia de ejecución*, no como garantías de *SLO*. Las pruebas de concurrencia en SQLite pueden exhibir `SQLITE_BUSY` esporádicos bajo presión de I/O; el criterio es que el total de filas insertadas coincida con el esperado y que los bloqueos no impidan completar la secuencia. La suite fija `journal_mode=WAL` y usa conexiones independientes por lector/escritor; las tablas diagnósticas se limpian antes y después, asegurando **reproducibilidad** y ausencia de efectos colaterales.

7.7.6. Criterio metodológico.

Las pruebas en dispositivo complementan las de API validando *propiedades locales* cruciales para offline-first : transacciones ACID, rendimiento por lotes, no bloqueo lector/escritor en WAL y estabilidad del estado de interfaz. Al concentrar la ejecución en una pantalla QA, el equipo operacional dispone de una herramienta auditable para diagnóstico en campo y para generar evidencia comparable entre dispositivos, sin abrir la superficie de riesgo del entorno productivo.

CAPÍTULO 8

Resultados

Este capítulo presenta y discute la evidencia empírica del backend y del cliente móvil en relación con los objetivos planteados en la Sección 4.1–4.2: (i) garantizar integridad, consistencia y disponibilidad de los datos en entornos con conectividad intermitente; (ii) ofrecer formularios dinámicos y configurables desde el servidor; y (iii) establecer autenticación y sincronización seguras en modo offline-first . Para mantener trazabilidad, se reportan: (a) resultados de una prueba de **carga** reproducible (Locust) *del API*, (b) métricas agregadas y por contrato, (c) cobertura de **pruebas unitarias/funcionales del API**, y (d) resultados **en dispositivo** (SQLite y Redux) *del cliente móvil*.

8.1. Resultados del backend (API)

8.1.1. Contexto y alcance *sólo* para la prueba de carga

El panel del cliente móvil utilizado en operación reporta una población de **32** usuarios. Para dimensionar holgura operativa, el escenario de **carga** modeló **96** usuarios virtuales (VUs) en Locust , con esperas entre transacciones (*closed model*) para emular uso real. *Nota*: este contexto aplica únicamente a la **prueba de carga**; las pruebas **unitarias** y **E2E** no dependen de la población objetivo, pues validan correctitud y contratos bajo condiciones controladas.

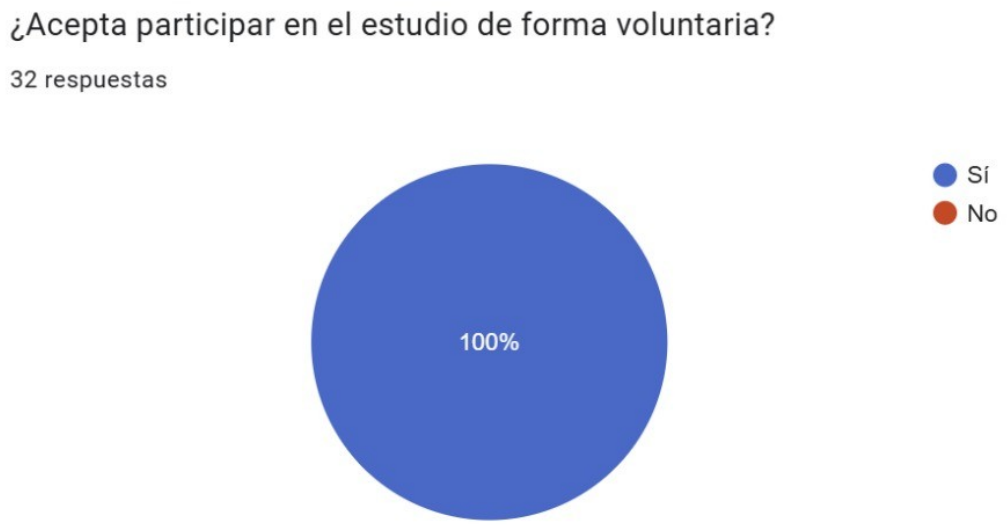


Figura 10: Evidencia de usuarios activos en el cliente móvil

Fuente: imagen provista por el frontend de la aplicación (Hernández, D.).

8.1.2. Prueba de carga

Diseño del experimento. Se desplegó la API en contenedores (Docker) e incluimos en el plan: autenticación (`/auth/login`), descubrimiento del árbol de formularios (`/forms/tree`), consulta de catálogos (`/forms/catálogo/dataset s`), grupos (`/groups`) y creación idempotente de entradas (`/forms/entries`). La *misma* imagen de producción se utilizó para la corrida, con semillas y JWT de prueba.

Métricas y discusión

Ámbito	Reqs	Fails	RPS	Avg (ms)	p95	p99	Máx	Concurr.	Err %
Total	37964	0	45.37	262.79	560	1000	2327	11.92	0.00 %

Cuadro 4: Métricas agregadas de la prueba de carga con 96 VU s (Locust).

Método	Endpoint	Reqs	RPS	Avg (ms)	p95	p99	Máx	Concurr.
POST	/auth/login	96	0.11	899.26	1900	1900	1904	0.10
GET	/catálogo/dataset s	6931	8.28	268.89	610	1100	2327	2.23
POST	/forms/entries	3472	4.15	134.95	480	1100	2017	0.56
GET	/forms/tree	17190	20.54	265.38	520	940	2058	5.45
GET	/groups	10275	12.28	291.59	590	1100	2255	3.58

Cuadro 5: Rendimiento por endpoint en la prueba de carga con 96 VU s (Locust).

Aumento de rendimiento con latencia controlada. Con 96 VU s se alcanzó una tasa agregada de **45.37** RPS y una concurrencia efectiva estimada de **11.92** ($N \approx \lambda \cdot W$), con latencia promedio de **262.79** ms. Pese al incremento de carga, los recursos de *lectura* (`/forms/tree`, `/datasets`, `/groups`) mantuvieron **p95** sub-segundo ($\leq$ 520–610 ms), favorable a la percepción de fluidez en móviles.

Login como ruta más costosa (esperable). El punto de *login* exhibió *p95* ∼1900 ms y *p99* ∼1900 ms, coherente con operaciones criptográficas y lectura de usuario. En la práctica, el inicio de sesión ocurre esporádicamente y la app reutiliza el JWT, por lo que su impacto sobre UX sostenida es bajo.

Inserciones idempotentes estables. El `POST /forms/entries` mantuvo promedio de **135** ms y *p95* de **480** ms, con **0 %** de errores, confirmando el correcto funcionamiento del mecanismo idempotente bajo reintentos.

Ausencia de fallos y saturación. No se registraron fallos ni excepciones durante la corrida (`0` fallos en los reportes de Locust). La distribución de percentiles y la baja concurrencia efectiva relativa sugieren que aún existe margen antes de alcanzar saturación del servicio.

8.2. Pruebas unitarias y de controladores (API)

8.2.1. Cobertura global

Se ejecutaron **7** suites y **26** pruebas, todas con estado *passing*. La cobertura global fue de **52.99** % (sentencias), **40.66** % (ramas), **34.8** % (funciones) y **53.61** % (líneas).

8.2.2. Cobertura en módulos críticos

La Tabla 6 concentra módulos con mayor impacto funcional: servicios de dominio y controladores de contratos.

Módulo	Sentencias	Ramas	Funciones	Líneas
AuthService	96.07 %	79.41 %	100.00 %	97.67 %
FormsService	64.31 %	52.07 %	64.28 %	67.10 %
GroupsService	91.37 %	68.29 %	100.00 %	93.75 %
AuthQrService	74.19 %	49.20 %	90.00 %	74.71 %
FormsController	100.00 %	75.00 %	100.00 %	100.00 %
GroupsController	100.00 %	75.00 %	100.00 %	100.00 %
AppController	90.00 %	75.00 %	66.66 %	87.50 %

Cuadro 6: Cobertura focalizada (valores del reporte de `jest -coverage`).

8.2.3. Lectura y riesgos mitigados

Las cifras altas en `AuthService` y `GroupsService` validan autenticación, emisión de token y visibilidad basada en grupos. `FormsService` cubre ensamblado de árbol, ordenamientos y postproceso de campos calculados; `AuthQrService` prueba éxito y casos de firma inválida/expirada. Quedan fuera de foco *guards, modules, strategies* y archivos de arranque, por su naturaleza infraestructural, sin afectar la verificación de *lógica de dominio* ni la correcta exposición de *contratos*.

8.2.4. Implicaciones y discusión por nivel (API)

- **Unitarias (servicios).** Capturan defectos lógicos con costo mínimo de diagnóstico. Al *mockear* DB, cifrado y compilación, las fallas se localizan en reglas y transformaciones (p. ej., *flat→tree*, secuencias, postcálculo). Esto eleva la *mantenibilidad* y reduce el *MTTR* ante cambios en catálogos y formularios.
- **Controladores.** Verifican delegación pura y forma de DTO s, evitando "deriva" de contrato. Su cobertura alta asegura que cambios en rutas no rompan a clientes existentes.
- **E2E (API+DB real).** Confirma que el contrato publicado funciona con migraciones, JWT y visibilidad por rol sobre datos reales. Su valor está en validar *integración* e *invariantes de dominio* con *fixture s* versionados, dando trazabilidad reproducible.

- **Carga.** Con 96 VU s, los p95 subsegundo en lecturas y el 0 % de errores evidencian margen operativo frente a los ∼32 usuarios reales. La concurrencia efectiva baja (Ley de Little) es coherente con el patrón móvil: no todo usuario interactúa a la vez. Implica *holgura* para picos por turnos en campo y que la idempotencia en `/forms/entries` evita duplicaciones bajo reintentos.

8.3. Resultados del cliente móvil

Esta sección sintetiza los resultados de las pruebas ejecutadas directamente en el dispositivo móvil, mediante pantallas de diagnóstico dedicadas. Se evaluaron dos ejes: (i) almacenamiento local en SQLite para operación *offline-first* y (ii) correctitud y desempeño del estado de interfaz gestionado con Redux. Los resultados reportan tiempos, propiedades de seguridad (atomicidad) y comportamiento bajo concurrencia lectora, así como determinismo y serializabilidad del estado.

8.3.1. SQLite local (persistencia offline)

Las pruebas cubrieron la creación de esquema de diagnóstico, la atomicidad de transacciones, el rendimiento por lotes con y sin transacción, y la concurrencia con `journal_mode=WAL` (lectores simultáneos contra un escritor). Las Tablas 7 y 8 consolidan los resultados observados.

Prueba	**Estado**	**Resumen**
Preparación de esquema	OK	Tablas `diag_*` creadas y vaciadas.
Atomicidad (rollback íntegro)	OK	Fallo inyectado a mitad de transacción; conteo final de filas = 0.
Rendimiento por lotes (sin vs con transacción)	OK	Inserciones masivas más veloces dentro de transacción por menor I/O/*fsync*.
Concurrencia WAL (lectores vs escritor)	OK	Lectores no bloquearon al escritor; sin eventos `SQLITE_BUSY`.
Limpieza	OK	Vaciado de `diag_*` y `wal_checkpoint(TRUNCATE)`.

Cuadro 7: SQLite en dispositivo: estado y resumen cualitativo.

Prueba	Métrica	Valor
Preparación	dur_ms	55
Atomicidad	dur_ms	2270
Batch	N	3000
Batch	sin_txn_ms	28603
Batch	con_txn_ms	15614
Batch	mejora_x	1.83
WAL	writes	1200
WAL	reads	689
WAL	SQLITE_BUSY	0
WAL	dur_ms	3936
Limpieza	dur_ms	20

Cuadro 8: SQLite en dispositivo: métricas cuantitativas.

Hallazgos principales. (a) La atomicidad se verificó: tras un fallo inyectado a mitad de transacción, la tabla de diagnóstico quedó sin inserciones. (b) El insert masivo *dentro* de transacción mostró una mejora de **1.83x** frente a inserciones *sin* transacción (por reducción de I/O y *fsync*), lo que favorece sincronizaciones puntuales de alto volumen. (c) En modo `WAL`, los lectores no bloquearon al escritor y no se observaron eventos `SQLITE_BUSY`; el conteo final de filas coincidió con el total de escrituras.

8.3.2. Redux (correctitud y desempeño del estado)

Las pruebas se enfocaron en propiedades esperadas para reducers puros: estado inicial definido, identidad ante acciones desconocidas, determinismo, serializabilidad del estado, integración con `configureStore` y un micro-benchmark de acciones *noop*. Las Tablas 9 y 10 resumen lo observado.

Prueba	Estado	Resumen
Estado inicial + serializable	OK	Estado inicial definido y serializable.
Acción desconocida	OK	Identidad: `next === prev`.
Determinismo	OK	Misma entrada → misma salida.
Serializabilidad	OK	Sin valores no serializables.
Integración con store	OK	Estable ante acción desconocida.
Performance (acciones *noop*)	OK	10000 acciones; costo por acción muy bajo.

Cuadro 9: Redux en dispositivo: estado y resumen cualitativo.

Prueba	Métrica	Valor
Estado inicial	dur_ms	0
Acción desconocida	dur_ms	0
Determinismo	dur_ms	0
Serializabilidad	dur_ms	0
Integración store	dur_ms	3
Performance (noop)	iterations	10000
Performance (noop)	dur_ms	17
Performance (noop)	avg_us	2

Cuadro 10: Redux en dispositivo: métricas cuantitativas.

Hallazgos principales. (a) El reducer de sesión de formulario presentó estado inicial definido y serializable; (b) acciones desconocidas no mutaron el estado (identidad), y el cálculo fue determinista; (c) la integración con `configureStore` se comportó estable; (d) el micro-benchmark de 10 000 acciones *noop* registró aproximadamente 2 µs por acción, lo que da margen suficiente para formularios largos sin penalización perceptible en la interfaz.

8.3.3. Implicaciones y discusión (móvil)

- **SQLite** (***offline-first***)**.** La verificación de atomicidad y la mejora ∼1.8x de inserciones dentro de transacción respaldan sincronizaciones puntuales robustas en conectividad irregular. El modo `WAL` sin `SQLITE_BUSY` observables indica que el patrón lector/escritor concurrente no penaliza la captura, preservando *fluidez* de la UI y evitando pérdida de trabajo.
- **Redux (estado de sesión).** Las invariantes (estado inicial definido, identidad ante acciones desconocidas, determinismo, serializabilidad) confirman que el *store* es seguro para formularios extensos. El micro-benchmark (≈2 µs/acción) sugiere que el costo de "*no-op*" es despreciable, dejando el cuello de botella en E/S y red, no en el *reducer*.
- **Complementariedad API–móvil.** Las pruebas móviles validan *propiedades locales* (ACID, estado puro) que las pruebas del API no pueden observar; a la inversa, E2E y carga validan *propiedades remotas* (contratos, latencias, p95) fuera del alcance del dispositivo. En conjunto, reducen el riesgo sistémico: consistencia local ∧ contratos estables ∧ holgura de rendimiento.

CAPÍTULO 9

Conclusiones

1. Se construyó un backend móvil de formularios adaptables que preserva integridad, consistencia y disponibilidad aun con conectividad intermitente, mediante idempotencia de envíos, almacenamiento local en SQLite, sincronización explícita y versionado de estructuras.

2. Al diseñar un contrato claro entre servidor y cliente para el árbol de formularios y conjuntos de datos; el cliente interpretó categorías, formularios, páginas y campos (calculados y repetibles) con validación por tipo, lo que habilitó personalización sin migraciones invasivas y mejoró la captura en campo.

3. Se implementó la consistencia y la seguridad mediante validaciones mínimas con identificador idempotente, autenticación y autorización con tokens firmados, cifrado en tránsito y respaldos automáticos; quedó pendiente estandarizar cifrado en reposo y la verificación periódica de restauraciones, para preservar integridad.

4. La operación *offline-first* se sincronizó con sesión y cursor persistidos, avance en SQLite, sincronización diferida y reintentos sin duplicación; en pruebas, el estado Redux se mantuvo puro, determinista y serializable, asegurando continuidad y fluidez en formularios extensos.

5. Incluso modelando más usuarios de los previstos, las latencias permanecieron en rango subsegundo y la concurrencia efectiva fue baja, coherente con ráfagas de uso móvil; esto sugiere holgura operativa suficiente hoy y en la puesta en producción.

6. Las pruebas unitarias y de controladores cubren rutas y servicios críticos; las de extremo a extremo validan el trayecto cliente→API→BD; y las pantallas de diagnóstico en móvil permiten repetir y exportar evidencia, aportando trazabilidad y base reproducible para iterar.

CAPÍTULO 10

Recomendaciones

Con el sistema operativo y validado, se proponen acciones concretas, simples y orientadas a cerrar brechas y facilitar la adopción en producción.

1. **Seguridad y resguardo de la información.** Mantener todo el tráfico bajo HTTPS, activar cifrado en reposo donde sea necesario y programar respaldos automáticos *con pruebas periódicas de restauración*. Dejar documentada la ubicación de llaves y credenciales y su proceso de rotación.

2. **Monitoreo práctico.** Definir objetivos de servicio por los endpoints principales (tiempos p95/p99 y tasa de errores) y crear alertas sencillas cuando se superen. Agregar un *correlation-id* en los registros para acelerar diagnósticos.

3. **Robustez offline.** Priorizar sincronización diferencial (enviar solo cambios) y definir reglas simples ante conflictos (por ejemplo, último cambio gana con registro mínimo de auditoría). Usar reintentos con espera creciente para mejorar estabilidad en redes irregulares.

4. **Calidad y despliegue.** Ampliar la cobertura de pruebas en piezas estructurales, incorporar escenarios de extremo a extremo representativos y utilizar despliegues graduales con verificación de salud antes de liberar a todos los usuarios.

CAPÍTULO 11

Bibliografía

[1] CGIAR, "Perfil de Agricultura Digital — Guatemala," CGIAR, 2024. dirección: https://santalucialareformasimsan.siinsan.gob.gt/wp-content/uploads/2024/11/CGIAR-Perfil-de-la-agricultura-digital-en-Guatemala.pdf (visitado 27-10-2025).

[2] Food and Agriculture Organization of the United Nations, "The future of food and agriculture: Trends and challenges," FAO, Rome, 2017. dirección: https://openknowledge.fao.org/server/api/core/bitstreams/2e90c833-8e84-46f2-a675-ea2d7afa4e24/content (visitado 27-10-2025).

[3] Asociación de Azucareros de Guatemala (Asazgua), "Agroindustria Azucarera de Guatemala | Industria, Innovación e Infraestructura (ODS 9)," Asazgua, Estudio de caso, 2024. dirección: https://guatecana.com/wp-content/uploads/2024/10/Estudio-de-caso-AIA-ODS09.pdf (visitado 27-10-2025).

[4] Android Developers. "Build an offline-first app: Data layer." (feb. de 2025), dirección: https://developer.android.com/topic/architecture/data-layer/offline-first (visitado 27-10-2025).

[5] Android Developers. "Data layer | App architecture." (feb. de 2025), dirección: https://developer.android.com/topic/architecture/data-layer (visitado 27-10-2025).

[6] C. Hartung, Y. Anokwa, W. Brunette, A. Lerer, C. Tseng y G. Borriello, "Open Data Kit: Tools to Build Information Services for Developing Regions," en *Proceedings of the 4th ACM/IEEE International Conference on Information and Communication Technologies and Development (ICTD '10)*, ACM/IEEE, 2010. DOI: 10.1145/2369220.2369236. dirección: https://odk-x.org/assets/files/ODK-Paper-ICTD-2010.pdf (visitado 27-10-2025).

[7] M. Bartling, S. Sotelo, A. Eitzinger y K. Atzmanstorfer, "Press the Button: Online/Offline Mobile Applications in an Agricultural Context," *GIForum*, vol. 4, n.º 1, págs. 106-116, 2016. DOI: 10.1553/giscience2016_01_s106. dirección: https://austriaca.at/rootcollection?arp=0x0033fe93 (visitado 27-10-2025).

[8] SQLite. "Write-Ahead Logging." (n.d.), dirección: https://sqlite.org/wal.html (visitado 27-10-2025).

[9] SQLite. "SQLite is Transactional (ACID)." (n.d.), dirección: https://www.sqlite.org/transactional.html (visitado 27-10-2025).

[10] L. Klerkx, F. Bert, A. Gardeazabal et al. "Estado de la digitalización del sector agropecuario en América Latina y el Caribe: Perspectivas y propuestas para optimizar el ecosistema Agtech," Instituto Interamericano de Cooperación para la Agricultura (IICA). (2025), dirección: https://blog.iica.int/blog/estado-digitalizacion-del-sector-agropecuario-en-america-latina-caribe-perspectivas-propuestas (visitado 26-10-2025).

[11] Ó. San José Herrero. "Formularios digitales y su utilidad en la cadena del agronegocio," AgtechApps. (31 de ene. de 2022), dirección: https://agtechapps.com/formularios-digitales-agronegocio/ (visitado 26-10-2025).

[12] Thunderbit. "Las 6 mejores herramientas de software para la recolección de datos que debes probar." (14 de ago. de 2025), dirección: https://thunderbit.com/es/blog/best-data-collection-software-tools (visitado 26-10-2025).

[13] F. Jeffrey-Coker, M. Basinger y V. Modi, "Open Data Kit: Implications for the Use of Smartphone Software Technology for Questionnaire Studies in International Development," Columbia University, inf. téc., 2010. dirección: https://mobileactive.org/research/open-data-kit-implications-use-smartphone-software-technology-questionnaire-studies-international-development/ (visitado 26-10-2025).

[14] Glance. "Offline-first: Why Your Mobile App Must Work Without Internet." (2023), dirección: https://thisisglance.com/blog/offline-first-mobile-apps/ (visitado 26-10-2025).

[15] DigiFormsApp. "DigiFormsApp (Página oficial)." (2023), dirección: https://www.facebook.com/digiformsapp/ (visitado 26-10-2025).

[16] E. Reale. "Climatempo lanza el app Agroclima PRO," Revista Cultivar. (25 de feb. de 2019), dirección: https://revistacultivar.com/noticias/climatempo-lanca-o-app-do-agroclima-pro (visitado 26-10-2025).

[17] N. Eichorn. "How CommCare Provider Oikoi is Digitizing Data Collection in Agriculture," Dimagi. (30 de oct. de 2024), dirección: https://dimagi.com/blog/commcare-provider-profile-how-oikoi-is-digitizing-data-collection-for-agriculture/ (visitado 26-10-2025).

[18] SmartProtect H2020. "AgroBase – agronomic pest and disease database (app)." (), dirección: https://platform.smartprotect-h2020.eu/en/view/ipm/71 (visitado 26-10-2025).

[19] Food and Agriculture Organization of the United Nations (FAO), "The future of food and agriculture: Trends and challenges," FAO, 2017, Recuperado de http://www.fao.org/3/a-i6583e.pdf y https://openknowledge.fao.org/server/api/core/bitstreams/f7ecc7f0-8b58-48b5-ae57-290d5f8808c1/content.

[20] World Business Council for Sustainable Development (WBCSD), *Exploring the business case for Digital Climate Advisory Services (DCAS): India case study*, World Business Council for Sustainable Development, Recuperado de https://www.wbcsd.org/wp-content/uploads/2023/10/WBCSD-Digital-Climate-Advisory-Services-for-Sustainable-and-resilient-agriculture-in-India-Case-study.pdf, 2020.

[21] L. Ruiz-García y L. Lunadei, "The role of RFID in agriculture: Applications, limitations and challenges," *Computers and Electronics in Agriculture*, vol. 79, n.º 1, págs. 42-50, 2011. DOI: 10.1016/j.compag.2011.07.009.

[22] A. Matese y S. F. Di Gennaro, "Technical aspects of unmanned aerial vehicle for precision agriculture: a review," *International Journal of Remote Sensing*, vol. 36, n.º 8, págs. 2101-2132, 2015. DOI: 10.1080/01431161.2015.1022671.

[23] H. A. Beshir, Z. Abdel-Aty y S. Abdel-Hafeez, "Mobile applications for precision agriculture: A review," *Computers and Electronics in Agriculture*, vol. 168, pág. 105 091, 2020. DOI: 10.1016/j.compag.2019.105091.

[24] A. Kamilaris, A. Fonts y F. X. Prenafeta-Boldú, "The rise of blockchain technology in agriculture and food supply chains," *ArXiv*, 2017. DOI: 10.5281/zenodo.657207.

[25] M. Satyanarayanan, "Fundamental challenges in mobile computing," en *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC 1996)*, ACM, 1996, págs. 1-7. DOI: 10.1145/248052.248054.

[26] G. Lawrence, C. Marais, A. Herbert y B. Irwin, "Offline-First Design for Fault Tolerant Applications," en *SATNAC 2018 Conference Proceedings*, Recuperado de https://www.researchgate.net/publication/327624337_Offline-First_Design_for_Fault_Tolerant_Applications, 2018.

[27] C. Pahl y P. Jamshidi, "Microservices: A systematic mapping study," en *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016)*, ACM, 2016, págs. 137-144. DOI: 10.5220/0005785501370146.

[28] MoCloudCom, *Addressing Serverless Computing Vendor Lock-In through Cloud Service Abstraction*, ArXiv, Recuperado de https://faculty.washington.edu/wlloyd/papers/MoCloudCom2023proof.pdf, 2023.

[29] "About the New Architecture," React Native. (2025), dirección: https://reactnative.dev/architecture/landing-page (visitado 19-10-2025).

[30] "Native Modules: Introduction (TurboModules)," React Native. (2025), dirección: https://reactnative.dev/docs/turbo-native-modules-introduction (visitado 19-10-2025).

[31] "The New Architecture is Here," React Native. (2024), dirección: https://reactnative.dev/blog/2024/10/23/the-new-architecture-is-here (visitado 19-10-2025).

[32] "Impeller rendering engine," Flutter. (2025), dirección: https://docs.flutter.dev/perf/impeller (visitado 19-10-2025).

[33] "Flutter architectural overview," Flutter. (2025), dirección: https://docs.flutter.dev/resources/architectural-overview (visitado 19-10-2025).

[34] "Technology | 2025 Stack Overflow Developer Survey," Stack Overflow. (2025), dirección: https://survey.stackoverflow.co/2025/technology/ (visitado 19-10-2025).

[35] "About npm," npm, Inc. (2023), dirección: https://docs.npmjs.com/about-npm/ (visitado 19-10-2025).

[36] "npm | Home," npm, Inc. (2025), dirección: https://www.npmjs.com/ (visitado 19-10-2025).

[37] "Using Codegen (New Architecture)," React Native. (2025), dirección: https://reactnative.dev/docs/the-new-architecture/using-codegen (visitado 19-10-2025).

[38] "Technology | 2024 Stack Overflow Developer Survey," Stack Overflow. (2024), dirección: https://survey.stackoverflow.co/2024/technology (visitado 19-10-2025).

[39] "Performance profiling," Flutter. (2025), dirección: https://docs.flutter.dev/perf/ui-performance (visitado 19-10-2025).

[40] NestJS. "OpenAPI (Swagger) — NestJS." (2025), dirección: https://docs.nestjs.com/openapi/introduction (visitado 20-10-2025).

[41] NestJS. "GraphQL — Quick Start." (2025), dirección: https://docs.nestjs.com/graphql/quick-start (visitado 20-10-2025).

[42] TypeScript Team. "TypeScript Handbook — Intro to Static Types." (2025), dirección: https://www.typescriptlang.org/docs/handbook/2/everyday-types.html (visitado 20-10-2025).

[43] Express.js. "Express — Node.js web application framework." (2025), dirección: https://expressjs.com/ (visitado 20-10-2025).

[44] Express.js. "Express routing." (2025), dirección: https://expressjs.com/en/guide/routing.html (visitado 20-10-2025).

[45] Express.js. "Using middleware." (2025), dirección: https://expressjs.com/en/guide/using-middleware.html (visitado 20-10-2025).

[46] Express.js. "Hello World example." (2025), dirección: https://expressjs.com/en/starter/hello-world.html (visitado 20-10-2025).

[47] Spring Team. "Spring Boot — Reference Documentation." (2025), dirección: https://docs.spring.io/spring-boot/index.html (visitado 20-10-2025).

[48] Spring Team. "Spring Boot — Core Features." (2025), dirección: https://docs.spring.io/spring-boot/reference/features/index.html (visitado 20-10-2025).

[49] springdoc-openapi. "springdoc-openapi — Documentation." (2025), dirección: https://springdoc.org/ (visitado 20-10-2025).

[50] S. Ramírez. "FastAPI — Features." (2025), dirección: https://fastapi.tiangolo.com/features/ (visitado 20-10-2025).

[51] S. Ramírez. "FastAPI — OpenAPI and API docs." (2025), dirección: https://fastapi.tiangolo.com/reference/openapi/ (visitado 20-10-2025).

[52] Pallets Projects. "Flask — Official Documentation." (2025), dirección: https://flask.palletsprojects.com/ (visitado 20-10-2025).

[53] Microsoft Docs. "APIs overview — ASP.NET Core." (2025), dirección: https://learn.microsoft.com/aspnet/core/fundamentals/apis (visitado 20-10-2025).

[54] Microsoft Docs. "Tutorial: Create a minimal API with ASP.NET Core." (2024), dirección: https://learn.microsoft.com/aspnet/core/tutorials/min-web-api (visitado 20-10-2025).

[55] Microsoft Docs. "Create web APIs with ASP.NET Core — Generate web API help pages with Swagger/OpenAPI." (2024), dirección: https://learn.microsoft.com/aspnet/core/web-api/ (visitado 20-10-2025).

[56] Microsoft Docs. "Authentication and authorization in minimal APIs." (2024), dirección: https://learn.microsoft.com/aspnet/core/fundamentals/minimal-apis/security (visitado 20-10-2025).

[57] gin-gonic. "Gin Web Framework — Documentation." (2025), dirección: https://gin-gonic.com/docs/ (visitado 20-10-2025).

[58] Go Developers. "Gin — pkg.go.dev." (2025), dirección: https://pkg.go.dev/github.com/gin-gonic/gin (visitado 20-10-2025).

[59] TechEmpower. "Framework Benchmarks." (2025), dirección: https://www.techempower.com/benchmarks/ (visitado 20-10-2025).

[60] Stack Overflow. "2025 Stack Overflow Developer Survey — Technology (Databases)." Sección "Databases": porcentajes de uso por base de datos. (2025), dirección: https://survey.stackoverflow.co/2025/technology/.

[61] SQLite. "Most Widely Deployed SQL Database Engine." Página oficial: "billions and billions of copies" y presencia en móviles/computadoras. (s.f.), dirección: https://www.sqlite.org/mostdeployed.html.

[62] D. R. Hipp, *Appropriate Uses For SQLite*, Documentación oficial de SQLite, Recuperado de https://www.sqlite.org/whentouse.html, 2025.

[63] PostgreSQL Global Development Group. "About PostgreSQL." Documentación oficial de PostgreSQL. (2025), dirección: https://www.postgresql.org/about/.

[64] M. Bartling, S. Sotelo, A. Eitzinger y K. Atzmanstorfer, "Press the Button: Online/Offline Mobile Applications in an Agricultural Context," *GI_Forum*, vol. 1, n.º 1, págs. 106-116, 2016.

[65] Android Developers. "Save data in a local database using Room." (2023), dirección: https://developer.android.com/training/data-storage/room.

[66] G. Firebase, *Habilita funciones sin conexión en Android – Realtime Database*, Documentación de Firebase, n.d.

[67] A. Developers, *Cómo compilar una app que prioriza el uso sin conexión*, https://developer.android.com/topic/libraries/architecture/coding-for-offline, Recuperado de https://developer.android.com/topic/libraries/architecture/coding-for-offline, 2023.

[68] M. Palani y K. Thangaraj, *Offline-First Application – Design and Architecture*, BigThinkCode, recuperado de https://www.bigthinkcode.com/insights/offline-first-application-design-and-architecture, 2022.

[69] PouchDB, *Guide: Conflicts*, PouchDB Official Documentation, 2021.

[70] Couchbase, *Cloud to Edge Data Sync with Couchbase Sync Gateway*, The Couchbase Blog, 2023.

[71] Redux. "Normalizing State Shape." Documentación oficial de Redux. Consultado en 2025-10-19. (2021), dirección: https://redux.js.org/usage/structuring-reducers/normalizing-state-shape.

[72] Redux. "Redux Style Guide." Guía oficial de estilo de Redux. Consultado en 2025-10-19. (2023), dirección: https://redux.js.org/style-guide/.

[73] rt2zz. "redux-persist: persist and rehydrate a Redux store." Repositorio oficial en GitHub. Consultado en 2025-10-19. (2021), dirección: https://github.com/rt2zz/redux-persist.

[74] Redux Toolkit. “Persistence and Rehydration (RTK Query).” Documentación oficial de RTK Query. Consultado en 2025-10-19. (2024), dirección: https://redux-toolkit.js.org/rtk-query/usage/persistence-and-rehydration.

[75] Redux Toolkit. “Manual Cache Updates (RTK Query).” Actualizaciones de caché optimistas/pesimistas. Consultado en 2025-10-19. (2024), dirección: https://redux-toolkit.js.org/rtk-query/usage/manual-cache-updates.

[76] Redux. “Redux Essentials, Part 8: RTK Query Advanced Patterns.” Documentación oficial de Redux. Consultado en 2025-10-19. (2025), dirección: https://redux.js.org/tutorials/essentials/part-8-rtk-query-advanced.

[77] Amazon Web Services. “Transactional outbox pattern — AWS Prescriptive Guidance.” Guía prescriptiva de AWS. Consultado en 2025-10-19. (2023), dirección: https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/transactional-outbox.html.

[78] C. Richardson. “Pattern: Transactional outbox.” microservices.io: patrón de datos (Transactional Outbox). Consultado en 2025-10-19. (s.f.), dirección: https://microservices.io/patterns/data/transactional-outbox.html.

[79] Android Developers. “Task scheduling | Background work (WorkManager).” Documentación oficial de WorkManager. Consultado en 2025-10-19. (2025), dirección: https://developer.android.com/develop/background-work/background-tasks/persistent.

[80] Android Developers. “Define work requests | Background work (Retry and backoff policy).” Política de reintentos y *backoff* en WorkManager. Consultado en 2025-10-19. (2025), dirección: https://developer.android.com/develop/background-work/background-tasks/persistent/getting-started/define-work.

[81] SQLite. “Write-Ahead Logging (WAL).” Documentación oficial de SQLite. Consultado en 2025-10-19. (2025), dirección: https://www.sqlite.org/wal.html.

[82] WSO2/Ballerina, *Publish-Subscribe Channel – Enterprise Integration Patterns*, Ballerina EIP Documentation, “Publish-Subscribe Channel delivers a copy of a particular event to each receiver.”, 2021.

[83] M. Mak, *Monitor data changes with SQL Server query notifications*, Makolyte Tech Blog, Recuperado de https://makolyte.com/event-driven-dotnet-how-to-use-query-notifications-in-sql-server-to-monitor-database-changes (consulta 2025), 2020.

[84] Microsoft, *SignalR – Real-time web functionality for .NET*, Microsoft Docs (ASP.NET SignalR), “Microsoft ASP.NET SignalR is a library for ASP.NET developers that simplifies the process of adding real-time web functionality to your applications.”, 2012.

[85] Google, *Firebase Cloud Messaging – Introduction*, Firebase Documentation, Recuperado de https://firebase.google.com/docs/cloud-messaging (fragmento: “Firebase Cloud Messaging (FCM) is a cross-platform messaging solution that lets you reliably send messages at no cost.”), 2023.

[86] TypeScript Team. “The TypeScript Handbook: Introduction.” (2025), dirección: https://www.typescriptlang.org/docs/handbook/intro.html.

[87] WHATWG. “The iframe element.” (2025), dirección: https://html.spec.whatwg.org/multipage/iframe-embed-object.html.

[88] TypeScript Team. "Basic Types." (2025), dirección: https://www.typescriptlang.org/docs/handbook/2/basic-types.html.

[89] Mozilla Developer Network. "Web Workers API." (2025), dirección: https://developer.mozilla.org/docs/Web/API/Web_Workers_API.

[90] R. Roberts, S. Marr, M. Homer y J. Noble, "Transient Typechecks are (Almost) Free," *arXiv preprint arXiv:1807.00661*, 2018. dirección: https://arxiv.org/abs/1807.00661.

[91] M. Jones y D. Hardt, *The OAuth 2.0 Authorization Framework: Bearer Token Usage*, IETF RFC 6750, https://www.rfc-editor.org/rfc/rfc6750, 2012.

[92] D. Hardt, *The OAuth 2.0 Authorization Framework*, IETF RFC 6749, https://www.rfc-editor.org/rfc/rfc6749, 2012.

[93] M. Jones, J. Bradley y N. Sakimura, *JSON Web Token (JWT)*, IETF RFC 7519, https://www.rfc-editor.org/rfc/rfc7519, 2015.

[94] M. Jones, J. Bradley y N. Sakimura, *JSON Web Signature (JWS)*, IETF RFC 7515, https://www.rfc-editor.org/rfc/rfc7515, 2015.

[95] M. Jones, *JSON Web Algorithms (JWA)*, IETF RFC 7518, https://www.rfc-editor.org/rfc/rfc7518, 2015.

[96] N. Sakimura, J. Bradley, M. Jones y B. de Medeiros, *Proof Key for Code Exchange by OAuth Public Clients (PKCE)*, IETF RFC 7636, https://www.rfc-editor.org/rfc/rfc7636, 2015.

[97] NIST, "Digital Identity Guidelines: Authentication and Lifecycle Management," National Institute of Standards y Technology, inf. téc. SP 800-63B, 2023, Recuperado de https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-63b.pdf.

[98] OWASP. "REST Security Cheat Sheet." (2023), dirección: https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html.

[99] OWASP. "Testing JSON Web Tokens & JSON Web Token Cheat Sheet." (2022), dirección: https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/06-Session_Management_Testing/10-Testing_JSON_Web_Tokens.

[100] OWASP. "Mobile Application Security Verification Standard (MASVS)." (2024), dirección: https://owasp.org/www-project-mobile-app-security/.

[101] W. Denniss, J. Bradley, M. Jones y D. Hardt, *OAuth 2.0 Device Authorization Grant*, IETF RFC 8628, https://www.rfc-editor.org/rfc/rfc8628, 2019.

[102] NIST, "The Keyed-Hash Message Authentication Code (HMAC)," National Institute of Standards y Technology, inf. téc. FIPS PUB 198-1, 2008, Recuperado de https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.198-1.pdf.

[103] X. Zhang et al., "Demystifying the (In)Security of QR Code-based Login in Real-World Deployments," en *Proceedings of the 34th USENIX Security Symposium*, Recuperado de https://www.usenix.org/conference/usenixsecurity25/presentation/zhang-xin, 2025.

[104] OWASP. "QRLJacking." (2021), dirección: https://owasp.org/www-community/attacks/Qrljacking.

[105] Google SRE. “Monitoring Distributed Systems.” (2025), dirección: https://sre.google/sre-book/monitoring-distributed-systems/.

[106] Google SRE. “Service Level Objectives.” (2025), dirección: https://sre.google/sre-book/service-level-objectives/.

[107] Google SRE Workbook. “Chapter 2 — Implementing SLOs.” (2025), dirección: https://sre.google/workbook/implementing-slos/.

[108] G. Tene. “How NOT to Measure Latency.” (2016), dirección: https://www.infoq.com/presentations/latency-response-time/.

[109] Prometheus Authors. “Histograms and summaries.” (2025), dirección: https://prometheus.io/docs/practices/histograms/.

[110] OpenTelemetry Authors. “Semantic conventions for HTTP metrics.” (2025), dirección: https://opentelemetry.io/docs/specs/semconv/http/http-metrics/.

[111] Apdex Alliance. “Apdex Technical Specification v1.1.” (2007), dirección: https://www.apdex.org/wp-content/uploads/2020/09/ApdexTechnicalSpecificationV11_000.pdf.

[112] Locust Cloud. “Closed vs Open Workload Models in Load Testing.” (2024), dirección: https://www.locust.cloud/blog/closed-vs-open-workload-models/.

[113] Grafana Labs. “Constant arrival rate executor.” (2025), dirección: https://grafana.com/docs/k6/latest/using-k6/scenarios/executors/constant-arrival-rate/.

[114] Apache JMeter Project. “User’s Manual: Constant Throughput Timer.” (2025), dirección: https://jmeter.apache.org/usermanual/component_reference.html.

[115] Locust Contributors. “Distributed load generation.” (2025), dirección: https://docs.locust.io/en/stable/running-distributed.html.

[116] Locust Contributors. “API: constant_throughput.” (2025), dirección: https://docs.locust.io/en/stable/api.html.

[117] Locust Contributors. “Custom load shapes.” (2025), dirección: https://docs.locust.io/en/stable/custom-load-shape.html.

[118] G. Tene. “wrk2: A constant throughput, correct latency recording variant of wrk.” (2025), dirección: https://github.com/giltene/wrk2.

[119] MIT OpenCourseWare. “Lecture 22: Sliding Window Analysis, Little’s Law.” (2012), dirección: https://ocw.mit.edu/courses/6-02-introduction-to-eecs-ii-digital-communication-systems-fall-2012/resources/lecture-22-sliding-window-analysis-littles-law/.

[120] Reactive Streams Initiative. “Reactive Streams: Standard for asynchronous stream processing with non-blocking back pressure.” (2025), dirección: https://www.reactive-streams.org/.

[121] J. Heinanen y R. Guerin. “RFC 2697: A Single Rate Three Color Marker.” (1999), dirección: https://www.rfc-editor.org/info/rfc2697.

[122] J. Heinanen, R. Guerin, J. Boyle y M. Kurosawa. “RFC 2698: A Two Rate Three Color Marker.” (1999), dirección: https://www.rfc-editor.org/info/rfc2698.

[123] Grafana Labs. “Types of load testing: smoke, stress, spike, breakpoint, soak.” (2025), dirección: https://grafana.com/load-testing/types-of-load-testing/.

[124] NestJS Team. “Testing (unit y E2E).” (2025), dirección: https://docs.nestjs.com/fundamentals/testing.

[125] Jest Authors. “Getting Started.” (2025), dirección: https://jestjs.io/docs/getting-started.

[126] Jest Authors. “Testing Asynchronous Code.” (2025), dirección: https://jestjs.io/docs/asynchronous.

Glosario

offline-first Estrategia de diseño en la que la aplicación debe funcionar primero sin conexión, sincronizando datos cuando haya red. IX, XI, 21–23, 25, 28, 35, 36, 41, 42, 48, 50, 51, 55, 57

backend Capa del sistema que implementa la lógica de negocio y expone servicios o APIs, típicamente conectada a una base de datos. VII, 12, 15, 19, 21–28

catálogo/dataset Conjunto de datos referenciales que alimenta listas, opciones o validaciones en los formularios. 36, 37, 43, 47, 53

Docker Tecnología de contenedores para empaquetar y ejecutar aplicaciones de forma reproducible. 12, 44, 45, 53

endpoint Ruta o recurso específico de una API (p. ej., `/forms/tree`) que admite operaciones HTTP. XIII, 32, 34, 39, 46, 48, 53

fixture Conjunto de datos de prueba controlados que se cargan para ejecutar pruebas reproducibles. IX, 46–48, 54

frontend Capa de presentación que interactúa con el usuario (aplicación móvil o web) y consume los servicios del backend. 12, 52

idempotencia Propiedad por la cual ejecutar la misma operación múltiples veces produce el mismo efecto que ejecutarla una sola vez (sin duplicados). 24, 25, 28, 35, 36, 43, 55

JSON Formato de intercambio de datos basado en texto (*JavaScript Object Notation*). 12, 17, 21, 30

JSONB Representación binaria de `JSON` en PostgreSQL con indexación y operaciones eficientes sobre documentos. 21

Locust Herramienta de pruebas de carga para aplicaciones web/APIs que modela usuarios virtuales y registra métricas de rendimiento. XIII, 51–53

OpenAPI/Swagger Especificación y herramientas para describir y documentar APIs REST de forma estandarizada e interactiva. 16, 17

p95 Percentil 95 de latencia; el 95 % de las solicitudes se resuelven en un tiempo menor o igual a este valor. 53, 55, 57

p99 Percentil 99 de latencia; el 99 % de las solicitudes se resuelven en un tiempo menor o igual a este valor. 32, 34

payload Contenido del mensaje enviado o recibido en una petición/respuesta HTTP, usualmente en `JSON`. 31

PostgreSQL Sistema de gestión de bases de datos relacional, con `MVCC`, extensiones y tipos avanzados (p. ej., `JSONB`). VIII, XI, 19–22, 36, 47, 48

SQLite Motor de base de datos embebido, transaccional (`ACID`), ideal para persistencia local en dispositivos. 12, 15, 19, 36, 37, 55, 57