Whitepaper Técnico: TilyShop Inventory

Sistema offline-first para la gestión de inventario y ventas de uniformes escolares

Campo	Detalle
Proyecto	TilyShop Inventory
Tipo de documento	Whitepaper técnico
Versión	1.0
Fecha	9 de noviembre de 2026
Ámbito	Arquitectura, seguridad, consistencia de datos y operación
Repositorio	Monorepo: `Inventory-Frontend/` + `Inventory-Backend/`

Resumen ejecutivo

TilyShop Inventory es un sistema de gestión de inventario diseñado para puntos de venta de uniformes escolares que operan en condiciones de conectividad inestable. El problema central que aborda no es únicamente registrar existencias, sino **garantizar la consistencia operativa entre ventas, ajustes administrativos y auditoría cuando la red no está disponible**.

La solución combina una **PWA offline-first** (Vue 3, TypeScript, Vite, Pinia, Dexie/IndexedDB) con una **API REST transaccional** (Node.js, Express 5, TypeScript, Prisma ORM 7, PostgreSQL/Supabase). Su propuesta técnica se apoya en tres decisiones estructurales: (1) cálculo de *stock efectivo* en el cliente sin modificar la “verdad” del servidor, (2) sincronización idempotente mediante `client_txn_id` (UUID generado por el cliente) y (3) operaciones críticas ejecutadas en transacciones atómicas con validación estricta, control de acceso por rol y trazabilidad.

Este documento describe el contexto del problema, la arquitectura, el modelo de datos, los mecanismos de seguridad, las garantías de consistencia y las consideraciones operativas del sistema, con el objetivo de servir como base técnica para evaluación, implementación, auditoría o evolución del producto.

1. Contexto y problema

La operación de una tienda de uniformes escolares presenta restricciones que dificultan el uso de sistemas puramente en línea:

- **Conectividad intermitente o nula**: cortes de internet, baja cobertura o fallas de infraestructura impiden registrar ventas en tiempo real.
- **Riesgo de sobreventa**: si múltiples terminales o reintentos duplican una operación, el stock puede descuadrarse.

- **Necesidad de auditoría**: los ajustes de inventario deben ser atribuibles a un usuario autorizado y dejar rastro verificable.
- **Protección de datos operativos y credenciales**: el sistema maneja información de usuarios, historial de ventas y movimientos sensibles.
- **Usabilidad en mostrador**: la interfaz debe permitir búsquedas rápidas, filtros por institución/prenda/color/talla y operación desde escritorio o dispositivos móviles.

Un enfoque tradicional “cliente delgado + servidor siempre disponible” traslada la disponibilidad del negocio a la disponibilidad de la red. TilyShop Inventory invierte esa dependencia: el cliente mantiene capacidad operativa local y el servidor actúa como **fuente autoritativa de reconciliación**, aplicando reglas de negocio y consistencia cuando la sincronización ocurre.

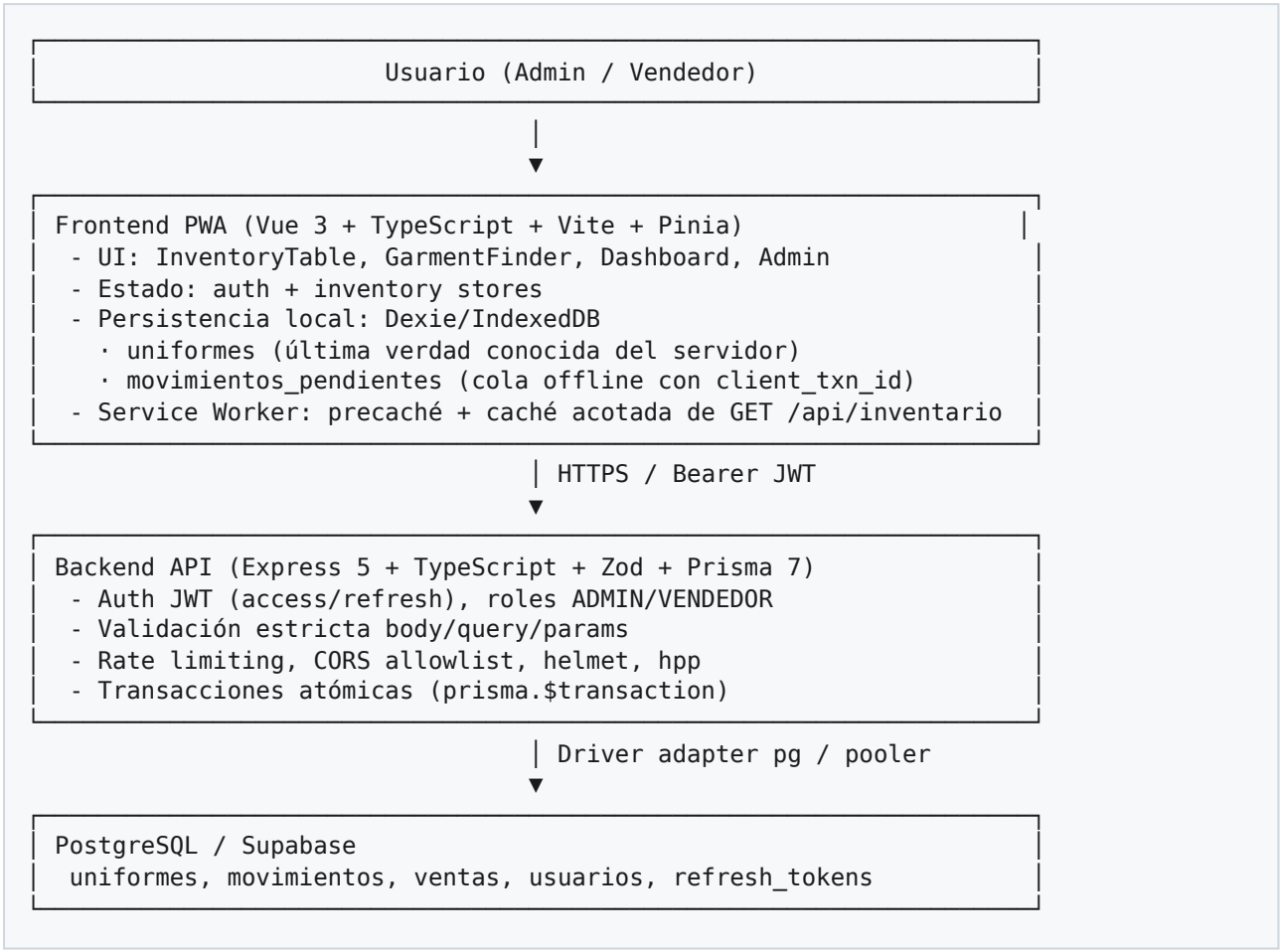
2. Objetivos de diseño

Los objetivos técnicos del sistema son:

1. **Continuidad operativa offline**: permitir consulta del catálogo conocido y registro local de operaciones sin conexión.
2. **Consistencia eventual con garantías fuertes**: evitar duplicados por reintentos y prevenir descuentos inválidos de stock.
3. **Trazabilidad**: conservar historial de movimientos y ventas con contexto suficiente para auditoría.
4. **Seguridad por capas**: autenticación, autorización, validación de entrada, endurecimiento HTTP, manejo seguro de secretos y mitigación de enumeración.
5. **Mantenibilidad**: separación clara entre frontend y backend, tipado estricto, esquemas explícitos y contratos REST estables.
6. **Experiencia de uso**: interfaz reactiva, instalable como PWA, con soporte de tema claro/oscuro y componentes adaptados a escritorio y móvil.

3. Arquitectura general

3.1 Visión de componentes



3.2 Responsabilidades

- **Frontend**: captura de intención del usuario, persistencia local, visualización de stock efectivo, generación de identificadores idempotentes y reintentos de sincronización.
- **Backend**: validación, autorización, ejecución de reglas de negocio, protección contra duplicados, mutación transaccional del estado y exposición de historial.
- **Base de datos**: integridad referencial, restricciones de unicidad (`client_txn_id`), índices de consulta y persistencia durable.

4. Modelo de datos y dominio

El modelo separa el catálogo, los eventos operativos y la auditoría financiera:

Entidad	Propósito	Aspectos relevantes
`uniformes`	Catálogo de prendas	Institución, prenda, color, talla, precio, stock, borrado lógico (`activo`).
`movimientos`	Evento de inventario	Tipo (`VENTA` , `INGRESO` , `AJUSTE`), cantidad (con signo en ajustes), `client_txn_id` único, `usuario_id` para trazabilidad.
`ventas`	Registro financiero inmutable	Cantidad, total congelado, precio unitario, snapshots de prenda/talla/color, usuario y enlace opcional al movimiento.
`usuarios`	Identidad y acceso	Email único, hash de contraseña, rol (`ADMIN` / `VENDEDOR`), estado activo, credenciales de recuperación hasheadas.
`refresh_tokens`	Sesiones renovables	`jti` , hash de token, expiración, revocación, rotación (`replaced_by_jti`).

4.1 Decisiones de modelado

- **Borrado lógico en uniformes**: `DELETE` archiva (`activo=false`) para preservar historial.
- **Snapshots en ventas**: `prenda_snapshot` , `talla_snapshot` , `color_snapshot` , `precio_unitario` y `total_venta` protegen el historial ante cambios posteriores del catálogo.
- **Idempotencia en el dominio**: `client_txn_id` existe en movimientos y ventas para tolerar reintentos offline.
- **Auditoría de ajustes**: los ajustes directos de stock realizados por administradores generan un `Movimiento` de tipo `AJUSTE` con diferencia positiva o negativa y `usuario_id` .

5. Estrategia offline-first y consistencia

5.1 Stock efectivo en el cliente

El cliente no sobreescribe localmente la verdad del servidor con cada operación optimista. En su lugar:

- `uniformes` almacena la **última versión conocida** proveniente del backend.
- `movimientos_pendientes` almacena operaciones aún no confirmadas.
- El stock visible se calcula como: **stock del servidor ± deltas pendientes**.

Este patrón evita dobles descuentos y mantiene una representación coherente tras sincronizar: cuando el backend aplica el delta, el movimiento sale de la cola y el stock efectivo converge con el nuevo estado del servidor.

5.2 Cola de sincronización

Cada venta o ingreso local se encola con un `client_txn_id` UUID. La sincronización se dispara:

- al recuperar conectividad,
- manualmente desde la interfaz,
- periódicamente (intervalo programado).

Las respuestas del servidor se interpretan de forma explícita:

- **Éxito o duplicado idempotente**: la operación se considera aplicada y se retira de pendientes.
- **Conflicto de negocio** (por ejemplo, `INSUFFICIENT_STOCK`): se marca como conflicto para revisión manual, evitando bucles de reintento infinito.
- **Falla transitoria** (red/servidor): permanece en cola para reintento.

5.3 Transacciones atómicas en el backend

Las mutaciones críticas se ejecutan dentro de `prisma.$transaction`:

- **Registro de movimiento**: verifica idempotencia, aplica decremento condicional en ventas (`stock >= cantidad`) o incremento en ingresos, inserta movimiento y, cuando aplica, inserta venta.
- **Ajuste directo de inventario**: obtiene stock actual, calcula diferencia, actualiza uniforme y crea movimiento `AJUSTE` con `usuario_id`; si la diferencia es cero, no genera movimiento.

La atomicidad garantiza que no existan estados intermedios persistidos (por ejemplo, stock actualizado sin registro de auditoría, o auditoría sin cambio real confirmado).

6. Seguridad

La seguridad se implementa como defensa en profundidad en frontend, backend y datos.

6.1 Autenticación y autorización

- **JWT propio** con access token de corta duración y refresh token de mayor duración.
- **Rotación atómica de refresh tokens**, detección de reúso y revocación.
- Algoritmo fijado a **HS256** en firma y verificación.
- Control de acceso basado en roles: operaciones administrativas (catálogo, usuarios, historial financiero, acciones sensibles) restringidas a `ADMIN`; operaciones de venta disponibles según política del negocio.

6.2 Protección de credenciales y recuperación

- Contraseñas almacenadas con hash (bcrypt).
- **Login de tiempo constante**: se ejecuta verificación de hash incluso ante usuario inexistente/inactivo para reducir señales de enumeración.
- Recuperación de contraseña con dos mecanismos: token de enlace (SHA-256) y código de 6 dígitos (HMAC-SHA256 + pepper), ambos con expiración, un solo uso y límites de intento. La respuesta de solicitud es genérica para evitar enumeración de correos.

6.3 Endurecimiento de API y cliente

- **CORS con allowlist estricta** (sin comodines).
- **Rate limiting** global, estricto en login/refresh y en escrituras.
- **Helmet**, `hpp`, límite de payload JSON y desactivación de `x-powered-by`.
- **Validación Zod** estricta en body/query/params antes de acceder a Prisma.
- **CSP y cabeceras** en frontend (`script-src 'self'`, `frame-ancestors 'none'`, HSTS, `nosniff`, `Referrer-Policy`, `Permissions-Policy`).
- **Service worker acotado**: solo cachea `GET /api/inventario`; no cachea respuestas sensibles (auth, usuarios, ventas). Limpieza de caché al iniciar/cerrar sesión o perder sesión.
- **Navegación segura**: redirecciones internas validadas; token de recuperación fuera del historial (`router.replace`).

6.4 Consideración sobre tokens en el navegador

El frontend usa Bearer tokens en `localStorage` por compatibilidad con la arquitectura actual. Esta decisión implica un riesgo residual ante XSS, mitigado por CSP estricta, ausencia de scripts inline/de terceros y controles de entrada/salida. Una evolución futura podría considerar refresh token en cookie `HttpOnly` con protección CSRF y ajustes CORS con credenciales.

7. Interfaz, operación y observabilidad funcional

La interfaz prioriza velocidad operativa en mostrador:

- **Búsqueda global normalizada** (sin acentos) y filtros facetados dependientes por institución, nombre, color y talla.
- **Paginación configurable** y tabla reactiva con estados de carga/vacío.
- **Selección múltiple** para acciones masivas administrativas.
- **Dashboard** con métricas calculadas en cliente (stock por institución, distribución por talla, valor y referencias con stock bajo).
- **PWA instalable** con precaché de activos estáticos para resiliencia de carga.

Desde el punto de vista operativo, el sistema distingue claramente entre conectividad, sincronización pendiente y conflictos, permitiendo intervención humana cuando una operación no puede reconciliarse automáticamente.

8. Pruebas y calidad

El proyecto incorpora verificación en múltiples niveles:

- **Backend**: pruebas unitarias (Vitest) para esquemas de validación, utilidades de seguridad, hashing, comportamiento de autenticación/refresh y aspectos críticos de lógica; smoke tests de integración contra base configurada.
- **Frontend**: tipado estricto con `vue-tsc` y pruebas E2E con Playwright. Las pruebas E2E interceptan `/api/**` con respuestas simuladas, lo que las hace deterministas y adecuadas para CI sin depender del backend real.

- **Build de producción**: generación de artefactos PWA y verificación de compilación.

Esta combinación reduce regresiones en contratos, seguridad y flujos de usuario, especialmente en áreas sensibles como filtros, paginación, sincronización y rutas administrativas.

9. Limitaciones actuales y trabajo futuro

Aunque el diseño cubre el caso principal de operación offline con reconciliación, existen líneas de evolución:

1. **Resolución avanzada de conflictos**: políticas más ricas para ventas concurrentes, reservas temporales o negociación de stock.
2. **Telemetría y auditoría centralizada**: métricas de sincronización, tasas de conflicto y alertas operativas.
3. **Pruebas de concurrencia**: escenarios multiusuario/multidispositivo con carga realista.
4. **Gestión de sesiones**: evaluación de cookies `HttpOnly` + CSRF para refresh tokens.
5. **Reportes financieros ampliados**: agregaciones server-side, exportación y conciliación contable.
6. **Observabilidad de datos**: paneles de anomalías (ajustes inusuales, intentos fallidos, reintentos masivos).

10. Conclusiones

TilyShop Inventory aborda un problema operativo concreto —vender y controlar inventario bajo conectividad imperfecta— con una arquitectura que no sacrifica consistencia ni auditoría. La combinación de PWA offline-first, stock efectivo derivado, sincronización idempotente y transacciones atómicas en el servidor permite mantener continuidad de negocio sin perder garantías fundamentales de integridad.

El valor técnico del sistema reside en la **alineación entre modelo de dominio y restricciones reales de operación**: cada venta o ajuste se trata como un evento trazable; los reintentos no duplican efectos; los conflictos de negocio se aíslan para revisión; y la seguridad se aplica desde la validación de entrada hasta el comportamiento del service worker. Esta base lo posiciona como una solución robusta para comercios de uniformes escolares y como una plataforma extensible hacia escenarios de mayor escala o requisitos regulatorios más estrictos.

11. Referencias técnicas del repositorio

- `README.md` — visión general del monorepo.
- `Inventory-Backend/README.md` — API, seguridad, endpoints, migraciones y pruebas.
- `Inventory-Frontend/README.md` — arquitectura offline-first, PWA, componentes y E2E.

- `Inventory-Backend/prisma/schema.prisma` — modelo de datos y restricciones.
- `DEPLOY.md` — guía de despliegue.

Documento generado para uso técnico y de evaluación del proyecto TilyShop Inventory. Debe revisarse junto con el código fuente y la configuración de despliegue vigente antes de decisiones de producción, cumplimiento o auditoría formal.