LUBECATECH

WorkDone Sanitarios

Documentación técnica y de onboarding

Producto	**WorkDone Sanitarios — control de limpieza por NFC**
Ecosistema	**Backend (Spring Boot) · App móvil (Android) · Kiosk Smiley**
Stack	**Java 25 · Spring Boot 4 · JHipster · Kotlin · Room · MSSQL**
Cliente piloto	**Corpal Servicios de Limpieza (Córdoba, AR)**
Fecha	**Junio 2026**
Audiencia	**Desarrollo y onboarding de nuevos integrantes**

shoppings: el personal acerca el teléfono a una placa NFC pegada en el baño, y eso queda asentado como un turno de limpieza con su hora exacta. El cliente piloto es **Corpal Servicios de Limpieza (Córdoba, AR)**. Antes de leer una línea de código, conviene entender qué problema resuelve el sistema, quiénes lo usan y cómo encajan las piezas.

Ver toda la documentación en PDF

Qué problema resuelve y para quién

En un sanitario de aeropuerto o shopping, la pregunta operativa de todos los días es simple y difícil a la vez: **¿se limpió este baño, cuándo, y se está cumpliendo la frecuencia comprometida?** Hasta ahora eso se resolvía con planillas en papel o con la confianza de que "alguien pasó". WorkDone reemplaza esa incertidumbre por un registro confiable y trazable:

- El **personal de limpieza** marca con un toque (tap NFC) el inicio y fin de cada limpieza, sin papeleo.
- La **supervisión** ve en tiempo real qué sanitario está libre, cuál se está limpiando y cuál tiene su **SLA vencido** (se pasó de la frecuencia comprometida).
- La **administración** del cliente recibe alertas, reportes y, a futuro, un portal de transparencia.
- El **visitante** del baño puede dejar su opinión con tres caritas en una terminal a la salida.

Está pensado para un escenario concreto: sanitarios con **mala conectividad** (subsuelos, terminales, zonas sin WiFi), muchos operarios rotando, y la necesidad de que **ninguna limpieza real se pierda** por un problema de red, de placa o de teléfono.

> **Cliente piloto**
>
> El piloto corre en **Corpal (Córdoba)**. Dimensiones de referencia: ~30 sanitarios, ~25 operarios, ~240 trabajos por día. Cargas chicas para la infraestructura — el foco está en la confiabilidad, no en la escala.

Actores y roles del sistema

WorkDone distingue claramente quién hace qué. Los tres roles operativos viven en la entidad `operario` y definen qué puede hacer cada persona en la app móvil:

Rol	Quién es	Qué hace
OPERADOR_LIMPIEZA	Personal de limpieza	Hace tap NFC y registra trabajos de tipo **LIMPIEZA**.
SUPERVISOR	Supervisor de Corpal	Hace tap NFC y registra trabajos de tipo **SUPERVISION** (control de calidad, no limpieza).
ADMIN	Encargado de equipamiento	No registra limpiezas: su app es **Gestión NFC** (alta, asignación y baja de placas).

A estos se suman dos actores que no usan la app móvil:

- **Supervisor / administración en el BackOffice web** — autenticación web propia (`jhi_user`), separada de la del móvil. Hace ABM de catálogos, mira el dashboard, gestiona alertas y dispositivos.
- **Visitante del sanitario** — anónimo, deja su opinión en la **terminal Smiley** (3 caritas) o mediante un **QR público**. No tiene login.

Dos mundos de autenticación

Las **personas con login** (operarios, web) se autentican con credenciales y reciben un **JWT**. Los **dispositivos** (teléfonos, terminales Smiley) se identifican con una **api_key** propia. Son dos planos distintos: `admin` (web) no sirve para la app móvil, y un operario no entra al portal del cliente. El detalle vive en Arquitectura.

Los componentes del ecosistema

WorkDone no es una sola aplicación, sino **un backend más tres clientes**, cada uno en su propio repositorio:

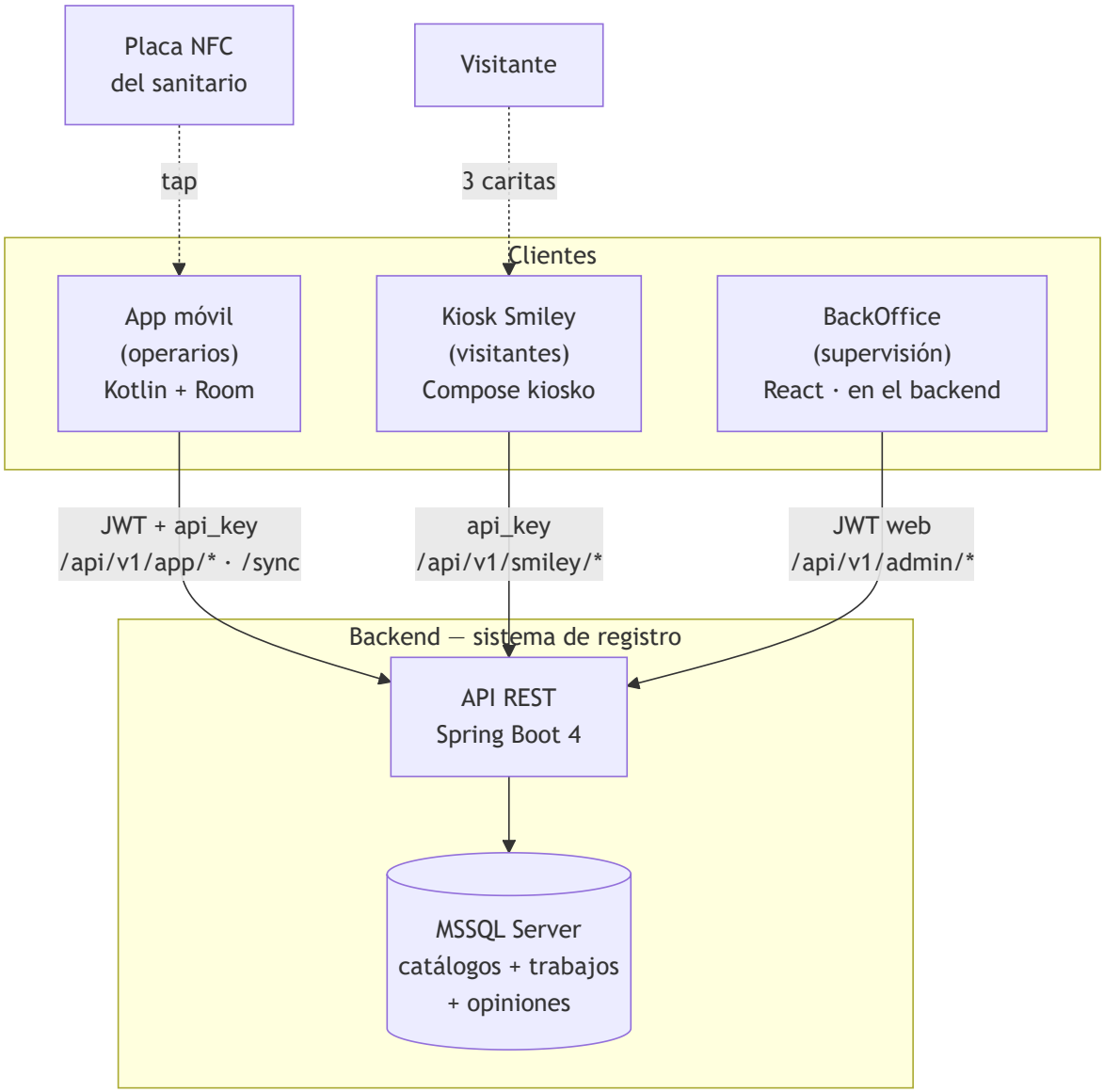
Componente	Repo	Stack	Qué hace

Backend	`workdone-backend`	Spring Boot 4 + JHipster + Java 25 + MSSQL	Sistema de registro central: API REST, autenticación, motor de trabajos (taps), sincronización offline, scheduler de SLA, exportes. La única fuente de verdad.
App móvil	`workdone-mobile`	Kotlin + Jetpack Compose + Room	App Android para operarios. Lee el tag NFC del sanitario, registra trabajos, opera 100% offline y sincroniza cuando hay red.
Kiosk Smiley	`workdone-smiley-kiosk`	Kotlin + Compose (modo kiosko)	Tablet montada a la salida del baño. El visitante opina con 3 caritas; muestra "Último servicio: hace X min" en reposo. Sin login.
BackOffice	dentro de `workdone-backend` (`src/main/webapp`)	Vite + React + shadcn/ui + Tanstack Query	Panel web de supervisión y administración: ABM (Estructura, Operarios, Rutas…), dashboard en tiempo real, reportes, alertas, y el portal del cliente. Ya construido y servido por el backend.

Por qué repos separados

Cada cliente tiene su propio ciclo de release, su stack y su equipo. El backend es la **fuente de verdad de los contratos**: los clientes no inventan endpoints, los consumen. Lo desarrollamos en detalle en Arquitectura.

Diagrama de alto nivel del ecosistema



Cómo leer esta documentación

El onboarding está pensado como un recorrido en tres tramos. No saltees el primero por más que quieras tocar código ya.

1. **Entender el sistema** *(estás acá)*. Empezá por esta página y seguí con Arquitectura para captar el porqué de la separación en repos y del enfoque offline-first. Cuando aparezca un término que no conozcas — *tap*, *sector*, *cierre automático*, *SLA* — está definido en el Glosario.
2. **Referencia técnica**. El modelo de datos, los contratos de API y el protocolo de sincronización son la letra chica del sistema. El protocolo de sync y el motor de taps son el **corazón del**

negocio: leelos enteros antes de tocar nada que los roce.

3. **Empezar a programar**. Recién entonces levantás el backend en local, corrés la app móvil contra él y empezás a contribuir. Cada repo tiene su `CLAUDE.md` con comandos, convenciones y reglas de oro.

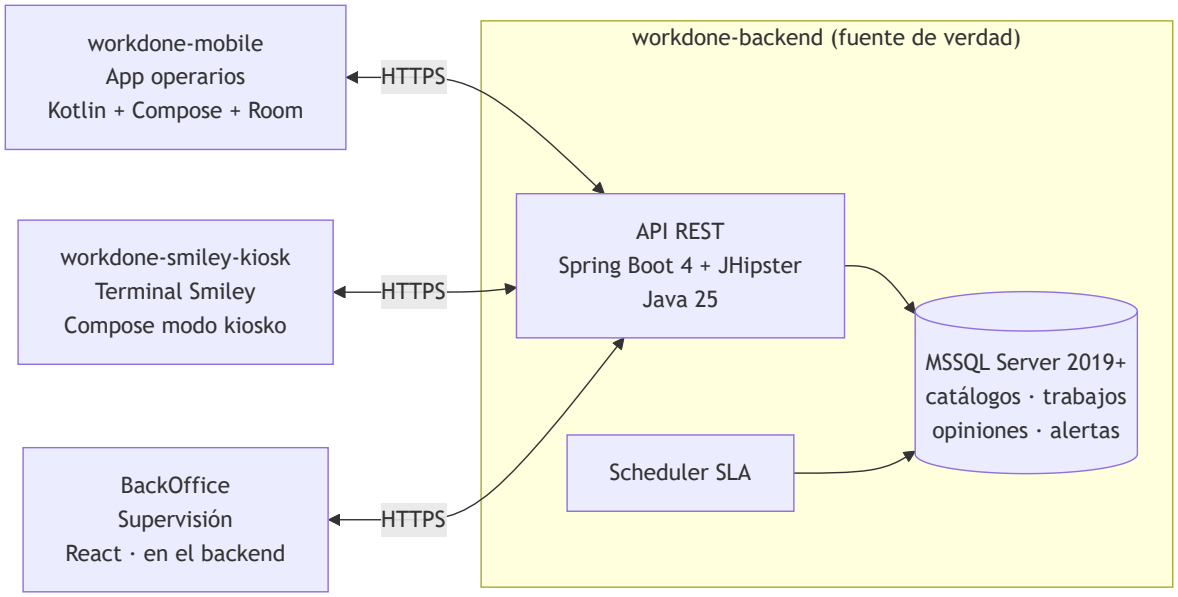
> ⚠️ **Regla de oro del ecosistema**
>
> El **backend es la fuente de verdad de los contratos**. Si un cliente (móvil o kiosk) tiene una copia de un contrato y no coincide con el backend real, **gana el backend**: se reporta, no se "arregla" en el cliente.

Arquitectura del ecosistema WorkDone

WorkDone no es una aplicación monolítica: es **un backend que oficia de sistema de registro y tres clientes que lo consumen**, cada uno en su propio repositorio. Esta página explica por qué está dividido así, por qué cada cliente trabaja *offline-first*, cómo se comunican las piezas y cómo se autentica cada actor. La idea no es que memorices endpoints — eso vive en la referencia de API — sino que entiendas el modelo mental: dónde está la verdad, quién manda y por qué.

La arquitectura 3+1

El ecosistema tiene **cuatro componentes en cuatro repos separados**: un backend central y tres clientes.



Componente	Repo	Rol
Backend	`workdone-backend`	Sistema de registro: la única fuente de verdad. Expone la API, procesa los taps, sincroniza, corre el scheduler de SLA.

App móvil	`workdone-mobile`	Cliente de operarios. Lee NFC, registra trabajos, sincroniza.
Kiosk Smiley	`workdone-smiley-kiosk`	Cliente de opinión pública. Registra caritas, muestra frescura del último servicio.
BackOffice	dentro de `workdone-backend` (`src/main/webapp`, Vite + React)	Cliente de supervisión y administración: ABM, dashboard, reportes, alertas y portal del cliente. Servido por el backend en `:8080`.

Por qué repos distintos

No es una decisión cosmética. Cada cliente tiene un stack, un ciclo de vida y restricciones propias:

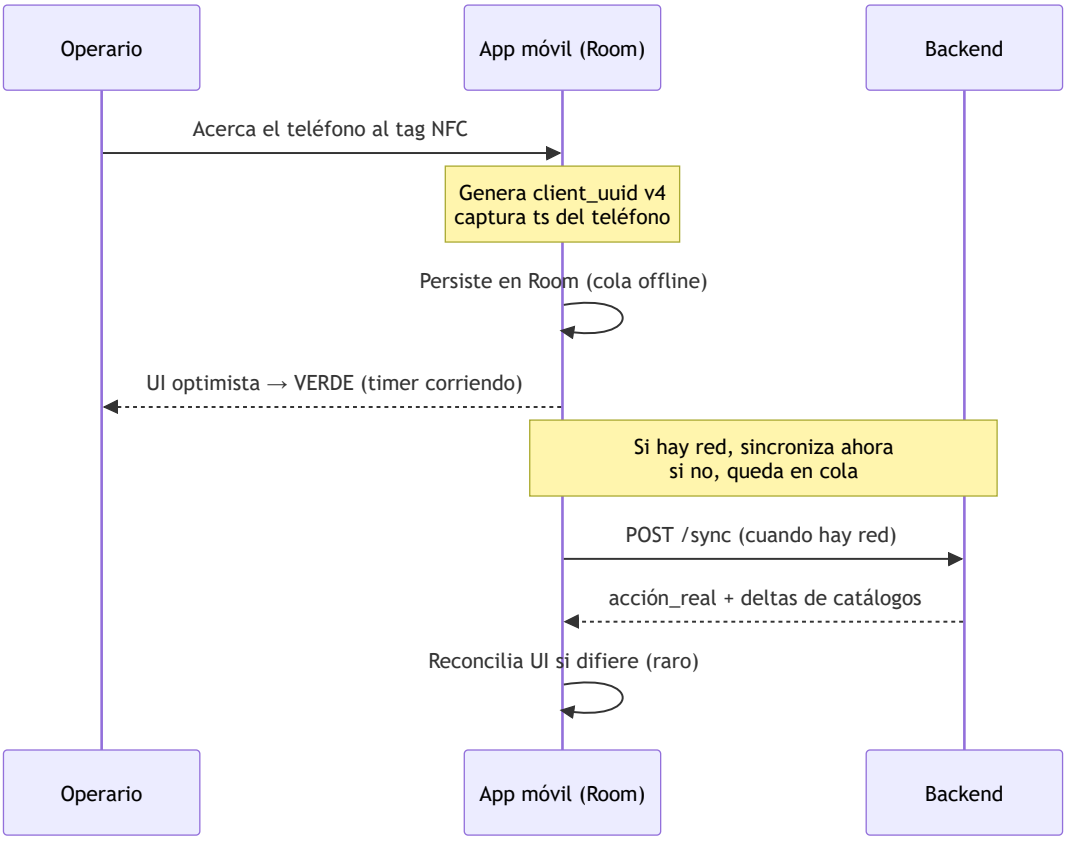
- **Stacks incompatibles entre sí.** El backend es Java/Spring, el móvil y el kiosk son Kotlin/Android. Mezclar esos builds en un repo único acoplaría cosas que no tienen nada que ver.
- **Releases independientes.** Una corrección en la app móvil no debería forzar un redeploy del backend, ni viceversa.
- **El BackOffice es la excepción: vive con el backend.** Es un frontend Vite + React + shadcn/ui + Tanstack Query que reside en `workdone-backend/src/main/webapp`: se construye y se sirve junto con el backend (en `:8080`), pero deliberadamente **no** usa el scaffolding de JHipster.

> **El backend manda sobre los contratos**
>
> Los clientes **no inventan endpoints ni DTOs**: los consumen. La fuente de verdad de cada contrato REST vive en `workdone-backend/docs/`. El kiosk y el móvil mantienen *copias gobernadas* del protocolo de sync, generadas automáticamente desde el canónico del backend (`scripts/contracts.ps1`), y un hook de git **aborta el push si la copia divergió**. Si documento y realidad no coinciden, **gana el backend**: se reporta, no se "arregla" en el cliente.

El patrón offline-first y por qué

Acá está la decisión arquitectónica más importante del ecosistema. Los clientes **no asumen que hay red**: asumen que **no la hay** y tratan la conexión como un lujo eventual.

El contexto lo explica todo: los sanitarios están en **subsuelos, terminales y zonas de aeropuertos/shoppings con conectividad pésima o nula**. Si el registro de una limpieza dependiera de tener señal en el momento del tap, se perderían limpiezas reales todos los días. Eso es inaceptable: la regla operativa de Corpal es que **ninguna limpieza real se pierde**.



Las reglas que hacen que esto funcione sin perder ni duplicar nada:

- `client_uuid` **se genera AL toque, antes de cualquier I/O.** Es la identidad del evento. Persiste en disco local antes de cualquier intento de red, así que sobrevive a crashes, reinicios y a que el sync falle 100 veces.
- **Idempotencia por** `client_uuid`**.** El mismo evento reenviado N veces produce el mismo resultado: el backend deduplica. Reenviar es seguro.
- **El timestamp del teléfono al momento del tap es la verdad** (`inicio_ts` / `fin_ts`). El backend solo registra aparte cuándo recibió el evento (`server_received_ts`). Esto vale aunque el evento llegue horas después.
- **UI optimista.** Al tapear, la app asume éxito y pinta verde/gris al instante. Si el backend después rechaza, se reconcilia. El operario nunca espera a la red.

- **La cola sobrevive a todo.** Reinicios de app, de device y actualizaciones: los trabajos pendientes en Room quedan intactos.

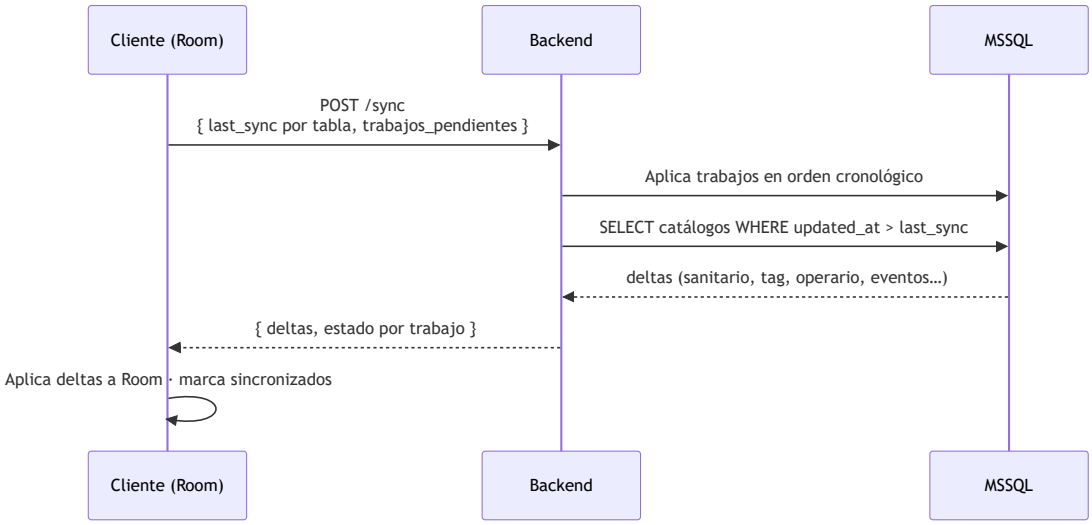
> **Mismo patrón en el kiosk**
>
> La terminal Smiley aplica las mismas reglas: opera 100% sin red (solo el "hace X min" del idle pierde frescura), genera `client_uuid` al toque y deduplica por idempotencia. El cliente cambia, el principio no.

Cómo se comunican: backend como sistema de registro

El backend es el **sistema de registro** (*system of record*): el lugar donde la verdad se consolida. Los clientes son **fuentes de eventos** que empujan lo que pasó y reciben de vuelta el estado consolidado.

El mecanismo central es la **sincronización** (`/sync`), que dispara periódicamente (cada ~15 min vía WorkManager), al recuperar red, al abrir la app o al loguearse:

1. El cliente empaqueta su cola de **trabajos pendientes** más un *watermark* (`last_sync`) por cada catálogo.
2. `POST /sync` con ese payload.
3. El backend procesa los trabajos **en orden cronológico** (`ts_local_device ASC` — el cliente garantiza el orden) y calcula los **deltas** de catálogos: las filas con `updated_at > last_sync` .
4. La respuesta trae los deltas a aplicar más el estado de cada trabajo procesado.
5. El cliente aplica los deltas a su cache local (Room) y marca los trabajos como sincronizados.

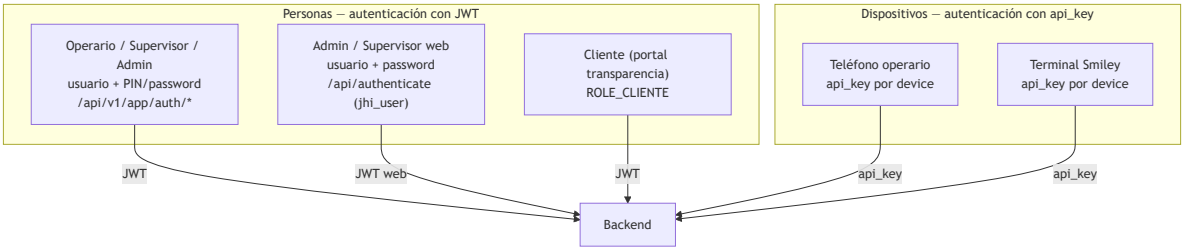


Las consultas en tiempo real (dashboard del BackOffice, frescura del kiosk) son lecturas directas contra el backend; el dashboard hace *polling* periódico. Pero el flujo que define la arquitectura es el de sync: **los clientes acumulan eventos localmente y el backend los consolida**.

> **El corazón del negocio**
>
> El motor de taps y el protocolo de sync (`/sync` , `/tap`) son la pieza crítica del sistema: ahí viven la idempotencia, el cierre automático de trabajos colgados, el anti doble-tap y los casos especiales del tap. En el backend, **leer `docs/SYNC_PROTOCOL.md` entero es obligatorio** antes de tocar nada que los roce. Esta página da el modelo mental; el contrato exacto vive ahí.

Autenticación a alto nivel

WorkDone separa tajantemente **personas** de **dispositivos**. Son dos planos de identidad distintos.



- **Personas → JWT.** Los operarios se loguean con `usuario + PIN` (o password) y reciben un **JWT corto** más un **refresh token** ligado al `device_uuid` . El BackOffice web tiene su propio plano de login (`jhi_user`), separado del móvil: el rol web (`ROLE_USER` / `ROLE_ADMIN`) no es el

rol móvil (`ROLE_OPERARIO`). Hay además un `ROLE_CLIENTE` de solo lectura para el portal de transparencia.

- **Dispositivos → api_key.** Cada teléfono y cada terminal Smiley se identifica con una **api_key propia**, usada para rate limiting, trazabilidad y para poder **deshabilitar un device** (si se pierde o roban un teléfono, administración lo deshabilita y el siguiente sync lo rechaza).
- **Endpoints sin auth.** El QR público (`/api/v1/publico/*`) es anónimo, *rate-limited* por IP — la cara pública de transparencia para el visitante.

> **Separación de prefijos**
>
> Cada audiencia tiene su prefijo de endpoints: `/api/v1/app/*` (móvil), `/api/v1/admin/*` (backoffice), `/api/v1/cliente/*` (portal cliente) y `/api/v1/publico/*` (QR público, sin auth). Los detalles de cada contrato — payloads, códigos, errores — están en la referencia de API del backend, no acá.

Deploy

El backend corre como un JAR de Spring Boot (servicio Windows vía NSSM) detrás de un reverse proxy (IIS ARR / nginx, `443 → 8080`), sobre una **EC2 Windows compartida con otros productos de LubecaTech**, con **MSSQL Server** en la misma instancia. HTTPS obligatorio (TLS 1.2 mínimo). Es una topología deliberadamente simple: las cargas son chicas (~240 trabajos/día, ~5 devices sincronizando cada 15 min) y no justifican infraestructura distribuida ni cache compartido.

Para entender las entidades que viajan en estos flujos — `sanitario`, `trabajo`, `tag_nfc`, `operario`, `dispositivo`, `alerta` — pasá por el Glosario.

Glosario del dominio

WorkDone tiene un vocabulario propio, en español, que aparece por todos lados: en el modelo de datos, en los endpoints, en las conversaciones con el cliente. **Los nombres de dominio se mantienen en español** (`sanitario`, `trabajo`, `operario`) — no se renombran a inglés. Esta página define los términos reales del sistema, agrupados por categoría, para que cuando los leas en el código o en una reunión sepas exactamente de qué se habla. Si un término te lleva a otro, seguilo: el dominio es una red, no una lista.

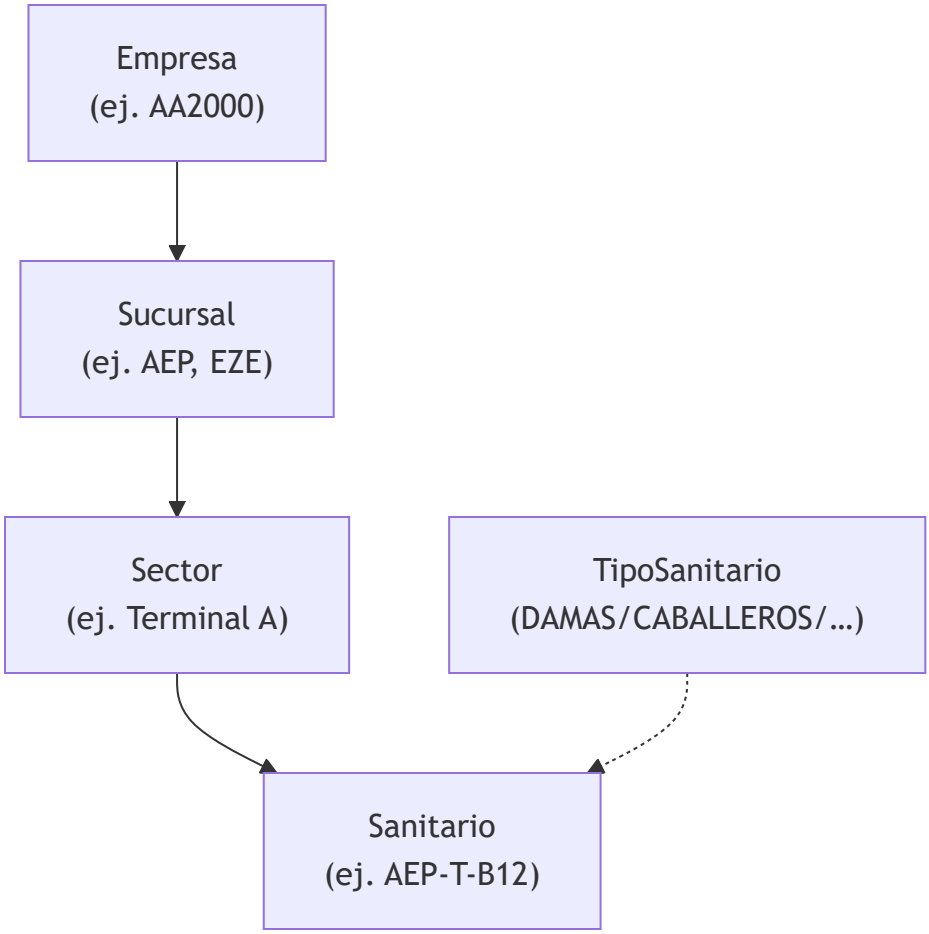
> **Fuente**
>
> Todas las definiciones salen del modelo de datos del backend y de las convenciones del ecosistema. Cuando necesites el detalle de columnas, índices o estados, la referencia es el modelo de datos del backend.

Jerarquía física

Cómo se organiza el espacio, de lo más grande a lo más chico. Es una cadena de pertenencia estricta: cada nivel cuelga del anterior.

Término	Definición
Empresa	El cliente del servicio (por ejemplo AA2000). Es el nivel más alto de la jerarquía. **No es multi-tenant**: una instalación de WorkDone sirve a una operación, la empresa es un dato de catálogo, no un aislamiento de inquilinos.
Sucursal	Una sede física de la empresa (por ejemplo AEP, EZE). Cuelga de una empresa.
Sector	Una zona dentro de la sucursal (por ejemplo "Terminal A", "Patio de comidas"). Cuelga de una sucursal. Su código es único dentro de la sucursal.
Sanitario	El baño concreto que se limpia y controla. La unidad operativa central. Cuelga de un sector, tiene un código único global (ej. `AEP-T-B12`) y opcionalmente un tipo. La historia de limpiezas vive acá, no en la placa.

TipoSanitario	Catálogo chico que clasifica al sanitario: `DAMAS`, `CABALLEROS`, `DISCAPACITADOS`, `MIXTO`.



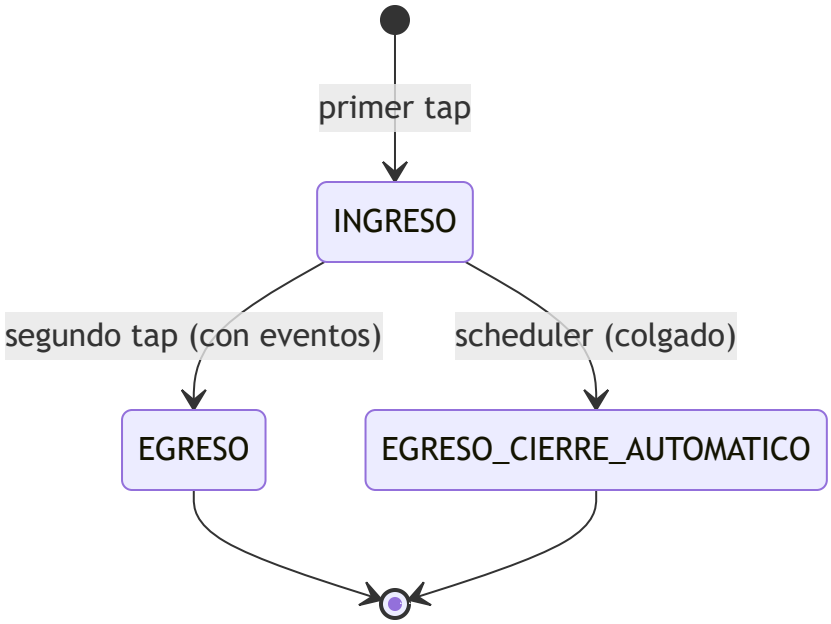
Operación: trabajos, taps y SLA

El núcleo de lo que el sistema registra. Acá vive el "qué pasó y cuándo".

Término	Definición
Tap	El gesto de acercar el teléfono a la placa NFC. Funciona como un **toggle**: el primer tap abre el trabajo (ingreso), el segundo lo cierra (egreso).

Trabajo	La entidad central: un turno de limpieza **o** de supervisión. Mismo motor, distinto `tipo`. Tiene su `client_uuid` (idempotencia), inicio, fin, duración, estado y modo de registro.
Ingreso	El estado del trabajo apenas se abre (primer tap). El operario "entró" al sanitario.
Egreso	El estado del trabajo cuando se cierra (segundo tap). Queda registrada la duración.
Cierre automático	Cierre de un trabajo que quedó "colgado" (sin egreso). Lo cierra el scheduler con la duración truncada. Lleva un *warning*; el cierre automático **sí** resetea el SLA, pero **no** cuenta como limpieza válida medida.
Tipo de trabajo	`LIMPIEZA` (lo crea el OPERADOR_LIMPIEZA) o `SUPERVISION` (lo crea el SUPERVISOR). El tipo lo define el rol del operario. La supervisión nunca resetea el SLA.
Modo de registro	Cómo se registró el trabajo: `ONLINE`, `OFFLINE_SYNC`, `OFFLINE_PENDIENTE_RESOLUCION` (tag desconocido) o `MANUAL` (registro sin placa).
Evento de limpieza	Lo que el operario reporta al cerrar el trabajo ("¿cómo quedó el sanitario?"). Catálogo configurable, no enum. Seed: `SIN_NOVEDAD`, `ROTURAS`, `REPONER_INSUMOS`, `NO_SE_PUDO_LIMPIAR`, `NO_ESTA_LIMPIO`. Algunos generan alerta; algunos **invalidan** la limpieza (no resetean el SLA).
SLA	El compromiso de frecuencia de limpieza de un sanitario (`SlaLimpieza`): cada cuántos minutos debe limpiarse, en qué ventana horaria y qué días de la semana.
SLA vencido	Estado en el que un sanitario pasó su frecuencia comprometida sin una limpieza válida. Dispara la alerta `SLA_VENCIDO`.
Limpieza válida	La definición clave para SLA y dashboard: un trabajo `tipo=LIMPIEZA` cerrado, **sin** eventos que invaliden la limpieza y con duración no anómala. **Solo las limpiezas válidas resetean el reloj del SLA.**

Duración anómala	Un egreso con duración por debajo del piso de validez configurado. Se marca y no cuenta como limpieza válida (un tap-tap demasiado corto no es una limpieza real).
Cuenta para SLA	Bandera por trabajo que materializa si esa fila cuenta para el SLA. NFC siempre cuenta; el registro manual depende de configuración; el cierre por scheduler no cuenta.
Registro sin placa / manual	Registro de limpieza cuando la placa no se puede leer (rota, ausente, no lee). El operario elige el sanitario y un **motivo de contingencia**. Cuenta como válida (no perjudica el cumplimiento) y avisa a administración.
Cierre con eventos	La pantalla del egreso donde el operario selecciona los eventos de limpieza antes de confirmar.

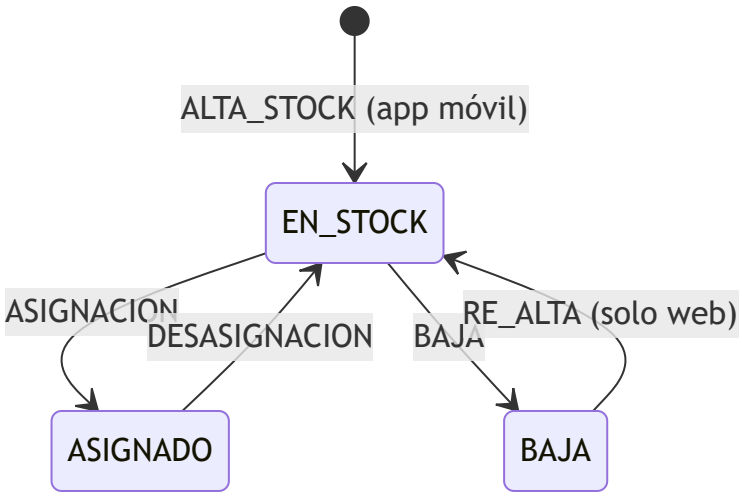


Tecnología NFC y dispositivos

Las piezas físicas y de identidad del ecosistema.

Término	Definición

NFC	*Near Field Communication*. La tecnología de proximidad con la que el teléfono lee la placa del sanitario.
Tag NFC / Placa	El chip pegado en el sanitario. Contiene una URL con un UUID (`https://workdone.lubeca.tech/t/{uuid}`). El móvil lee la URL y extrae el UUID. En el modelo es `TagNfc`.
UUID del tag	El identificador único que viaja en la placa. Se **normaliza** (trim/mayúsculas/sin espacios) en todo punto de entrada antes de usarlo.
Whitelist estricta	Regla de seguridad: un tag no registrado es inutilizable. Un tap con un UUID desconocido genera un trabajo `OFFLINE_PENDIENTE_RESOLUCION` y la alerta `TAG_DESCONOCIDO`.
Máquina de estados del tag	El ciclo de vida de una placa: `EN_STOCK` → `ASIGNADO` (a un sanitario) → `BAJA`, con desasignación y re-alta. Máximo **un tag ASIGNADO por sanitario**.
Alta de stock	Dar de alta una placa nueva. **Solo desde la app móvil** (rol ADMIN), porque hay que leer el chip onsite.
Gestión NFC	La app del rol ADMIN: en vez de registrar limpiezas, lee tags y los administra (alta, asignar, desasignar, reasignar, baja).
Dispositivo	Un equipo registrado: un teléfono de operario (`APP_OPERARIO`) o una terminal Smiley (`TERMINAL_SMILEY`). Tiene `device_uuid`, api_key, número de terminal y telemetría. Se puede **deshabilitar** (teléfono perdido/robado).
api_key	La credencial del dispositivo (no de la persona). Identifica al equipo para rate limiting y trazabilidad.
device_uuid	El identificador del dispositivo, al que se liga el refresh token de la sesión.
Drift de reloj	La diferencia entre el reloj del dispositivo y el del servidor, medida en el sync. Un drift grande dispara la alerta `RELOJ_DESVIADO`.
Doble-tap / rebote	Dos taps físicos muy seguidos. El sistema descarta el ingreso que rebota justo después de un egreso (`REBOTE_POST_EGRESO`) para no crear un trabajo espurio.



Sincronización e idempotencia

Los conceptos que sostienen el offline-first. Profundizados en Arquitectura.

Término	Definición
Offline-first	El cliente opera 100% sin red y trata la conexión como eventual. Nada se pierde, nada se duplica.
`client_uuid`	El identificador del evento, generado en el dispositivo **al toque, antes de cualquier I/O**. Es la clave de idempotencia.
Idempotencia	Garantía de que reenviar el mismo evento (mismo `client_uuid`) produce el mismo resultado. Reintentar es seguro.
Sync	El intercambio periódico (cada ~15 min, al recuperar red, al abrir/loguear) entre cliente y backend: el cliente empuja su cola, el backend responde con deltas y estados.
Cola offline	Los trabajos pendientes guardados localmente (Room) que sobreviven a crashes, reinicios y actualizaciones hasta que se sincronizan.
Delta	Las filas de catálogo cambiadas desde el último sync (`updated_at > last_sync`) que el backend devuelve para que el cliente actualice su cache.

`ts_local_device`	El timestamp del dispositivo al momento del evento. **Es la verdad temporal**; el cliente garantiza el orden cronológico de la cola por este campo.
`server_received_ts`	Cuándo el backend recibió el evento. Convive con `ts_local_device` pero no lo reemplaza.
`updated_at`	Marca de tiempo de catálogo que habilita los deltas de sync.
Soft delete	Borrado lógico (`deleted_at`) en catálogos: la fila no se elimina, así el sync puede propagar la baja a los clientes. En el BackOffice la convención es desactivar (`activo=false`), no borrar.

Opinión del público (Smiley)

La cara pública del sistema: lo que ve y toca el visitante del baño.

Término	Definición
Smiley / Terminal de opinión	La tablet en modo kiosko montada a la salida del sanitario. El visitante opina con un toque (3 caritas) y, si fue negativa, puede indicar un motivo. En reposo muestra "Último servicio: hace X min". Sin login.
Opinión	El registro inmutable de una carita: `POSITIVA` , `NEUTRA` o `NEGATIVA` , con su motivo opcional, su `client_uuid` y su origen.
Origen de la opinión	`TERMINAL_SMILEY` (la tablet) o `QR_PUBLICO` (escaneo de QR). Las de QR son "de segunda clase": no alimentan el motor de tendencias ni las stats de confianza.
Motivo de opinión	El detalle de una opinión negativa. Seed: `SUCIO` , `SIN_INSUMOS` , `MAL_OLOR` , `ALGO_ROTO` , `OTRO` .
Tendencia negativa	Una racha de opiniones negativas que supera un umbral en una ventana de tiempo. Dispara la alerta `TENDENCIA_NEGATIVA` .

QR público	Cara de transparencia para el visitante vía QR (`/p/{slug}`): muestra el estado público del sanitario (ej. "Limpiado hace X min") y permite opinar de forma anónima.
Provisioning / vinculación	El proceso por el que una terminal Smiley se asocia a un sanitario y recibe su identidad (número de terminal, alias) desde el backend. El device **no define su identidad legible**: la asigna el backend.
Modo kiosko	El confinamiento de la tablet (Lock Task, launcher dedicado, sin barras de sistema, autoarranque) para que el visitante no salga de la app Smiley.

Planificación y supervisión

Capas que se montan por encima de la operación.

Término	Definición
Ruta / Recorrido	Plantilla reusable de paradas (sanitarios) en un orden. Es una capa de **planificación arriba del tap, nunca un bloqueo**: planifica, no impide registrar.
Parada de ruta	Un sanitario dentro de una ruta, con su orden.
Asignación de ruta	La recurrencia semanal de una ruta a un operario (qué días, en qué ventana horaria, desde/hasta qué fecha).
Ejecución de ruta	La instancia de una ruta en un día concreto (`PENDIENTE` → `EN_CURSO` → `COMPLETADA` / `INCOMPLETA`). Una limpieza válida marca su parada como `HECHA` .
Alerta	Una señal que el sistema levanta para administración: `SLA_VENCIDO` , `DURACION_ANOMALA` , `OPERARIO_INACTIVO_OFFLINE` , `TAG_DESCONOCIDO` , `EVENTO_REPORTADO` , `RELOJ_DESVIADO` , `TAG_DEFECTUOSO` , `TENDENCIA_NEGATIVA` . Puede atenderse manualmente o, en algunos casos, auto-atenderse.
Calendario de excepciones	Fechas (feriados, paradas) por sucursal en las que no se exige limpieza: el SLA se pausa y no saltan alertas.

Dashboard	La grilla en tiempo real del BackOffice: cada sanitario con su estado (`LIBRE` / `LIMPIANDO` / `SLA_VENCIDO`), tiempo desde la última limpieza y operarios activos.
Portal del cliente	Acceso de solo lectura (`ROLE_CLIENTE`) a la transparencia del servicio: un cliente ve únicamente las empresas a las que tiene acceso, validado server-side.

Roles del sistema

Quién es quién. Los tres roles operativos viven en el `operario`; los demás son del plano web/público.

Término	Definición
Operario	La persona que usa la app móvil. Tiene `usuario`, PIN/password (hasheados, nunca viajan al móvil) y un **rol**.
OPERADOR_LIMPIEZA	El rol que limpia: crea trabajos de tipo `LIMPIEZA`.
SUPERVISOR	El rol que controla: crea trabajos de tipo `SUPERVISION`.
ADMIN	El rol que gestiona placas: su app es **Gestión NFC**, no registra limpiezas por tap.
ROLE_CLIENTE	Authority web de solo lectura para el portal de transparencia. No accede a endpoints operativos.
Usuario web (`jhi_user`)	La cuenta del BackOffice (supervisión/administración), en un plano de autenticación **separado** del móvil.
Visitante	El público anónimo que opina en la terminal Smiley o por QR. No tiene cuenta.

Para ver cómo estos términos se conectan en flujos concretos, volvé a Arquitectura; para el panorama general, a la página de inicio.

Sincronización offline-first

El protocolo de sincronización es el corazón de WorkDone Sanitarios. Los dispositivos —la app móvil de los operarios y el kiosk Smiley de opinión del público— operan en lugares donde la red no es confiable (sótanos de aeropuertos, sanitarios de shoppings). El sistema está diseñado para que **nada de eso importe**: el dispositivo trabaja 100% offline, acumula eventos localmente y los sube cuando puede.

Esta página explica por qué offline-first, el ciclo de sync paso a paso, las garantías (idempotencia, deduplicación, resolución de conflictos) y las diferencias entre el sync del mobile y el del kiosk Smiley.

> **Contratos de referencia**
>
> El detalle de request/response está en el API. Las entidades involucradas, en el modelo de datos. El vocabulario, en el glosario.

Por qué offline-first

La realidad física manda: un operario tapea su tag NFC en un sanitario donde no hay señal. Si el sistema dependiera de la red en ese instante, perdería el dato. En cambio:

- El dispositivo genera el `client_uuid` **al momento de la acción (el tap), ANTES de cualquier I/O**. Esa es la identidad del evento y la única garantía de idempotencia: PK local (Room) + UK server.
- El **timestamp del teléfono es la verdad** temporal (`ts_local_device` → `inicio_ts` / `fin_ts`). El server registra cuándo lo recibió, pero no recalcula cuándo ocurrió.
- La cola local se drena cuando hay red, en orden cronológico.

Lo que esto garantiza:

- **Nada se pierde**: cada acción queda persistida localmente antes de cualquier intento de red.
- **Nada se duplica**: el `client_uuid` deduplica tanto en el cliente (`OnConflictStrategy.IGNORE` en Room) como en el server (UK).

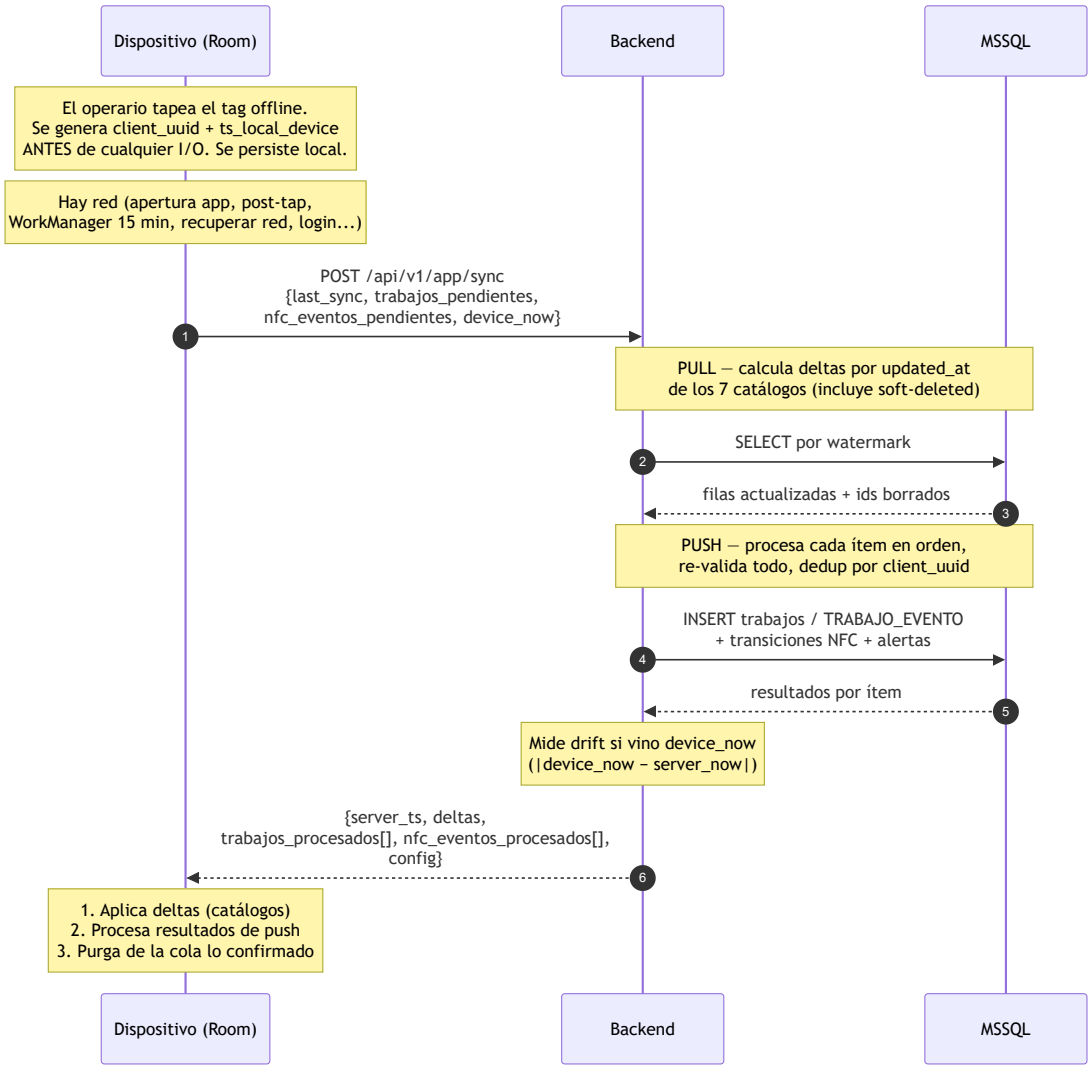
- **El server siempre re-valida**: roles, exclusividad de eventos, máquina de estados del tag. Nunca se confía en la validación del cliente.

> **Lo que el protocolo NO hace (a propósito)**
>
> No hace merge de conflictos en catálogos (en catálogos el server siempre gana), no usa websockets ni push (es **pull-based**), y no implementa compresión propia (la maneja HTTP).

El ciclo de sync paso a paso

El sync es un único request transaccional que combina **pull** (bajar deltas de catálogos) y **push** (subir la cola local). Dos reglas de orden son intocables:

1. **Push en orden cronológico** (`ts_local_device ASC`) — ambas colas (trabajos y NFC). El server confía en ese orden.
2. **Los deltas se aplican ANTES que los resultados de push** en el cliente.



Disparadores del sync (mobile)

Abrir la app (`last_sync` > 5 min) · tras cada tap/acción NFC · WorkManager cada 15 min · al recuperar red · al loguearse · pull-to-refresh · "Sincronizar ahora".

Procesamiento server por trabajo (el orden importa)

El server procesa cada trabajo de la cola en este orden:

1. **Dedup**: si el `client_uuid` ya está en `tap_descartado` → `RECHAZADO_REBOTE` (idempotente, se consulta **antes** que `trabajo`). Si ya está en `trabajo` → `DUPLICADO_IGNORADO` (devuelve el existente, no duplica eventos ni alertas).

2. **Resolver tag** (uuid normalizado): si no existe / EN_STOCK / BAJA → trabajo con `sanitario_id=NULL`, `modo=OFFLINE_PENDIENTE_RESOLUCION` + alerta `TAG_DESCONOCIDO` → estado `TAG_DESCONOCIDO`.
3. **Sanitario inactivo/eliminado** → `SANITARIO_INACTIVO`, no se persiste (el cliente purga sin reintentar).
4. **Operario inactivo** → se acepta + warning + alerta `OPERARIO_INACTIVO_OFFLINE`.
5. **Toggle según rol** (LIMPIEZA si OPERADOR, SUPERVISION si SUPERVISOR; ADMIN → `ROL_NO_PERMITIDO`, no persiste):
   - sin trabajo abierto → **INGRESO**.
   - abierto en el mismo sanitario → **EGRESO** (+ procesar eventos).
   - abierto en otro sanitario → cerrar el anterior con **EGRESO_CIERRE_AUTOMATICO** (duración truncada, sin eventos, warning) + INGRESO nuevo.
6. **Eventos del EGRESO**: validar existencia/activo, `roles_permitidos`, exclusividad (es_exclusivo ⇒ único), mínimo 1; materializar `TRABAJO_EVENTO` con snapshots; cada `genera_alerta` produce una alerta `EVENTO_REPORTADO`.
7. **Drift > 5 min** → warning.
8. **Race de `client_uuid` en el INSERT** → devolver el existente.

> **Limpieza válida y SLA**
>
> Solo una **limpieza válida** resetea el SLA: tipo `LIMPIEZA` cerrada, sin eventos que invaliden la limpieza y **sin `duracion_anomala`**. `SUPERVISION` nunca resetea. El cierre automático sí resetea, con warning.

Idempotencia, deduplicación y conflictos

Idempotencia y deduplicación

La idempotencia se ancla por completo en el `client_uuid`:

- Generado en el dispositivo **antes de cualquier I/O**, es PK local y UK server.
- Un reintento del mismo `client_uuid` devuelve siempre el mismo resultado: si ya está procesado, `DUPLICADO_IGNORADO` con el trabajo existente; si fue un rebote, `RECHAZADO_REBOTE` desde `tap_descartado`.

- Esto hace que reintentar la cola sea seguro: la red puede cortar a mitad de un sync sin generar duplicados.

Resolución de conflictos NFC

Para la cola de eventos NFC (`nfc_eventos_pendientes`), el server procesa cada uno:

1. `client_uuid` duplicado → `DUPLICADO_IGNORADO` .
2. Normalizar el `uuid_tag` y aplicar la acción vía `TagNfcLifecycleService` :
   - **ALTA_STOCK**: idempotente por uuid (existente → `OK` con el actual).
   - **ASIGNACION**: el tag debe estar EN_STOCK y el sanitario sin tag ASIGNADO; si no → `CONFLICTO_ASIGNACION` (**el primero en sincronizar gana**; el delta de `tag_nfc` trae la verdad).
   - **DESASIGNACION / REASIGNACION**: validación de máquina de estados → `TRANSICION_INVALIDA` si no aplica.
   - **Rol del operario insuficiente** para la gestión NFC → `ROL_INSUFICIENTE` .
   - **Fallo inesperado** al procesar el evento → `ERROR` .
3. Toda transición OK persiste su `TAG_NFC_EVENTO` .

Reacción del cliente ante la respuesta: `OK` / `DUPLICADO` → purgar de la cola. `CONFLICTO` / `TRANSICION_INVALIDA` → purgar, revertir el optimismo local con el delta y notificar al usuario. Error transitorio → backoff (mismo esquema de 8 intentos que los trabajos).

Casos especiales destacados

- **Tag reasignado entre tap y sync**: el trabajo se asocia al sanitario **actual** del tag (la realidad física manda) + warning.
- **Olvido de salida**: cierre automático truncado (por el server o por el scheduler de trabajos colgados).
- **Conflicto de asignación NFC offline**: dos devices asignan tags al mismo sanitario; el primero en sincronizar gana, el segundo recibe `CONFLICTO_ASIGNACION` y revierte.
- **Egreso corto accidental (doble-tap al salir)**: el server cierra el trabajo con `duracion_anomala=1` (no resetea SLA) + alerta `DURACION_ANOMALA` ; el registro queda auditable.
- **Rebote de ingreso**: un segundo tap a menos de `rebote-ingreso-seg` del EGRESO (mismo operario+sanitario+device) se interpreta como doble-tap, no como INGRESO nuevo → `RECHAZADO_REBOTE` , se registra en `tap_descartado` , no crea trabajo fantasma.

- **Registro MANUAL (placa rota)**: el ítem trae `manual=true` + `sanitario_id` + `motivo_contingencia_codigo` sin tag; el server resuelve por id, exige motivo válido (si falta → `MANUAL_INVALIDO`), crea el trabajo con `modo_registro=MANUAL` + alerta `TAG_DEFECTUOSO`.

Mobile vs. kiosk Smiley

El kiosk Smiley tiene **su propio endpoint de sync** (`POST /api/v1/smiley/sync`), separado del de operarios. Comparte las reglas de oro (idempotencia por `client_uuid`, `ts_local_device` como verdad temporal, push en orden cronológico, deltas antes que push, el server re-valida todo), pero es mucho más simple.

Aspecto	**Mobile (`/api/v1/app/sync`)**	**Kiosk Smiley (`/api/v1/smiley/sync`)**
Autenticación	JWT del operario (`ROLE_OPERARIO`)	api_key del device, headers `X-Device-Uuid` + `X-Api-Key` (`ROLE_TERMINAL`)
Qué sube (push)	`trabajos_pendientes` + `nfc_eventos_pendientes`	`opiniones_pendientes`
Qué baja (pull)	7 catálogos (deltas)	solo el catálogo `motivo_opinion`
Identidad del sanitario	va en el request (vía tag/sanitario)	la determina el server por el vínculo del device; la terminal **no puede** opinar por otro sanitario
Estados de push	8 estados (OK, TAG_DESCONOCIDO, RECHAZADO_REBOTE, …)	solo `OK` y `DUPLICADO_IGNORADO`
Anti-abuso	rebote/drift/duración anómala	debounce (3 s) + colapso de ráfaga (5 / 30 s)

El sync de opiniones

La terminal envía opiniones con `valor` ∈ **POSITIVA** | **NEUTRA** | **NEGATIVA** (enum `ValorOpinion`), `motivo_codigo?` opcional, más telemetría opcional (`app_version`, `os_version`, `bateria_pct`,

`red_tipo`, `lock_task_activo`). El server responde con `opiniones_procesadas` (estado `OK` o `DUPLICADO_IGNORADO`, los **únicos dos** que emite), el delta de `motivo_opinion`, y un bloque `config` (último servicio, textos, logo, `pin_tecnico_hash`).

> **Anti-abuso silencioso en el kiosk**
>
> Los descartes por debounce o ráfaga también devuelven `OK` a propósito, y se persisten con `descartada=true` (auditables, no se borran). El cliente drena de la cola toda opinión cuyo estado sea `OK` o `DUPLICADO_IGNORADO`; si una opinión enviada no vuelve en `opiniones_procesadas`, queda en cola para reintentar.

Disparadores del sync Smiley: tras cada opinión (inmediato) · WorkManager cada 15 min · al recuperar red. El idle ("último servicio") se refresca aparte con `GET /api/v1/smiley/estado` cada 5 min mientras la terminal está visible.

Detalles internos del sync

Hasta acá vimos el sync desde el contrato (qué entra, qué sale, qué garantías da). Esta sección baja un nivel: **cómo lo implementa el server por dentro**. Es material para debuggear —entender por qué una query es cara, por qué un trabajo abortó el batch pero un evento NFC no, o por qué rotar el device no resetea el throttling—. La fuente es el código de `SyncService`, `TrabajoTapService`, `LoginThrottlingService` y `AppAuthService`. Si algo de acá contradice al contrato del endpoint, gana el código.

Cálculo de deltas: no hay un watermark único

El pull **no** usa un único `last_sync`. El cliente manda un `last_sync` con un `since` **por cada catálogo** y el server dispara **7 queries separadas**, una por catálogo (`SyncService.calcularDeltas`, `SyncService.java:490-544`):

```
sanitarioRepository.findByUpdatedAtGreaterThan(sinceSanitario)
tagNfcRepository.findByUpdatedAtGreaterThan(sinceTagNfc)
operarioRepository.findByUpdatedAtGreaterThan(sinceOperario)
// ...sucursal, sector, tipo_sanitario, evento_limpieza
```

Cada query trae **todas las filas con `updated_at > since`, incluyendo las soft-deleted** (las que tienen `deleted_at != null`). El server **no** filtra los borrados en SQL: trae todo y particiona en memoria. El helper `partition` (`SyncService.java:656-672`) recorre las filas y las reparte:

Condición de la fila	Va a
`deleted_at == null`	`updated[]` (DTO completo)
`deleted_at != null`	`deleted_ids[]` (solo el id)

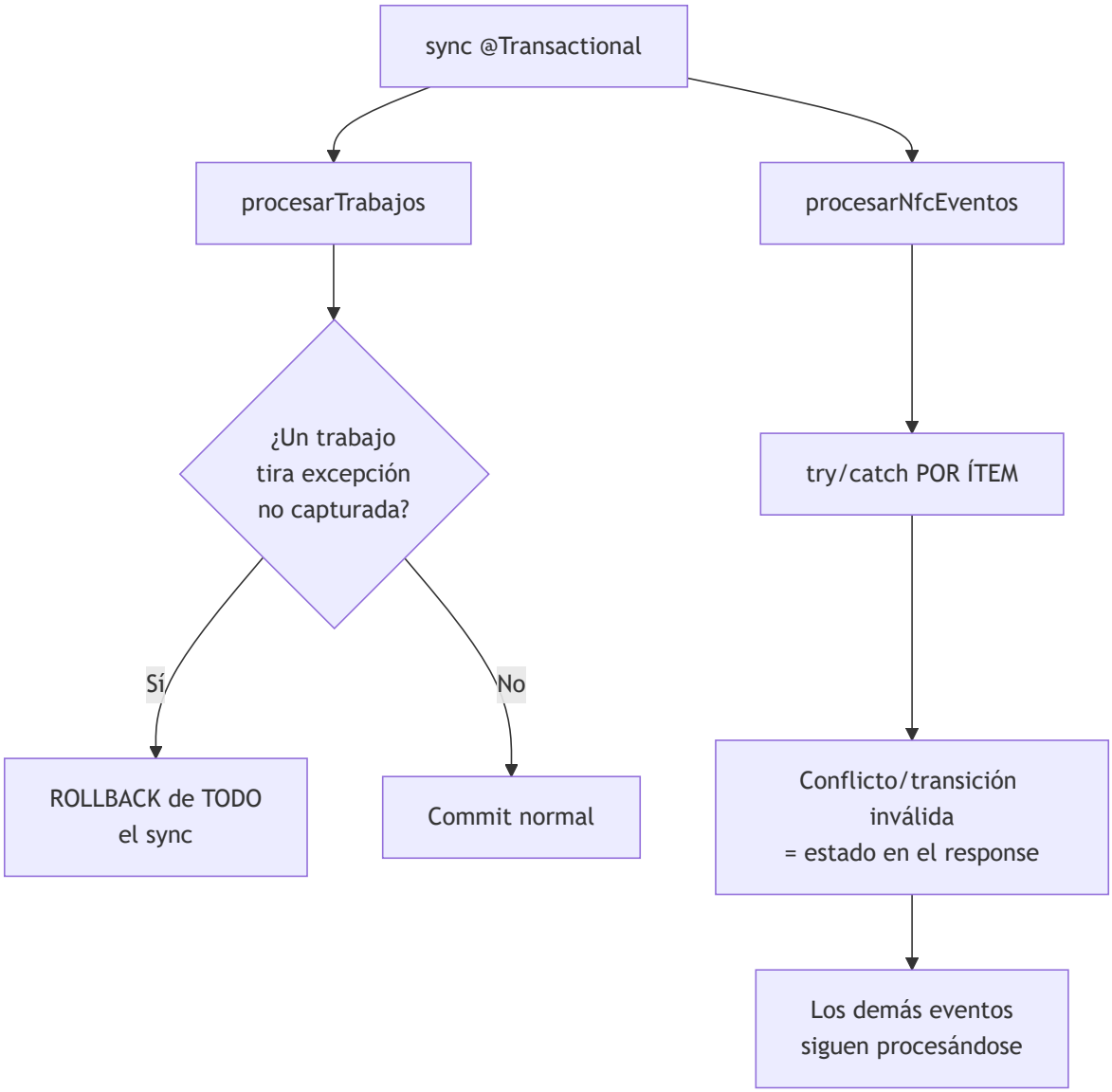
Por eso un catálogo borrado viaja como id en `deleted_ids` y el cliente lo elimina de su Room local. Si `last_sync` es `null` para un catálogo, el server usa `Instant.EPOCH` como `since` (`SyncService.java:491`) → trae **todo el catálogo**.

El riesgo a evaluar: queries sin LIMIT

> ⚠ **Riesgo real: pull sin paginación → posible OOM**
>
> Las 7 queries de deltas **no tienen `LIMIT` ni paginación**. Con un `last_sync` nulo (primer sync de un device) o muy viejo, `findByUpdatedAtGreaterThan(EPOCH)` materializa **el catálogo entero en memoria** del server: primero la `List<Entity>` de Hibernate, después la `List<DTO>` que arma `partition`. Con catálogos chicos no se nota; con catálogos grandes (miles de filas, p. ej. `tag_nfc` o `sanitario` en un despliegue grande) hay **riesgo de OOM o pausa de GC** que degrada todo el nodo. Para el piloto Corpal los volúmenes son chicos y no muerde, pero **antes de escalar a catálogos grandes hay que paginar o acotar el delta** (o cachear/streamear). Marcado como riesgo a evaluar, no como bug confirmado.

Asimetría transaccional: trabajos vs. eventos NFC

El método `sync()` está anotado `@Transactional` a nivel de clase (`SyncService.java:90`), así que **todo el batch corre en UNA transacción**. La consecuencia es una asimetría deliberada en el manejo de errores:



- **Trabajos** (`procesarTrabajos`, `SyncService.java:249-261`): no hay try/catch por ítem. Si `tapService.procesar(...)` revienta con una excepción no capturada, propaga y **hace rollback de todo el sync** (trabajos previos, eventos NFC, drift, `ultimo_sync`).
- **Eventos NFC** (`procesarUnNfcEvento`, `SyncService.java:451-486`): cada ítem va envuelto en try/catch. Un `ConflictoAsignacionException` o `TransicionInvalidaException` se traduce a un **estado en el response** (`CONFLICTO_ASIGNACION`, `TRANSICION_INVALIDA`) y **no aborta** los demás ítems ni la transacción.

Para el detalle de la máquina de estados NFC que dispara esas excepciones, ver la resolución de conflictos NFC más arriba en esta misma página.

Orden defensivo: el server no confía en el cliente

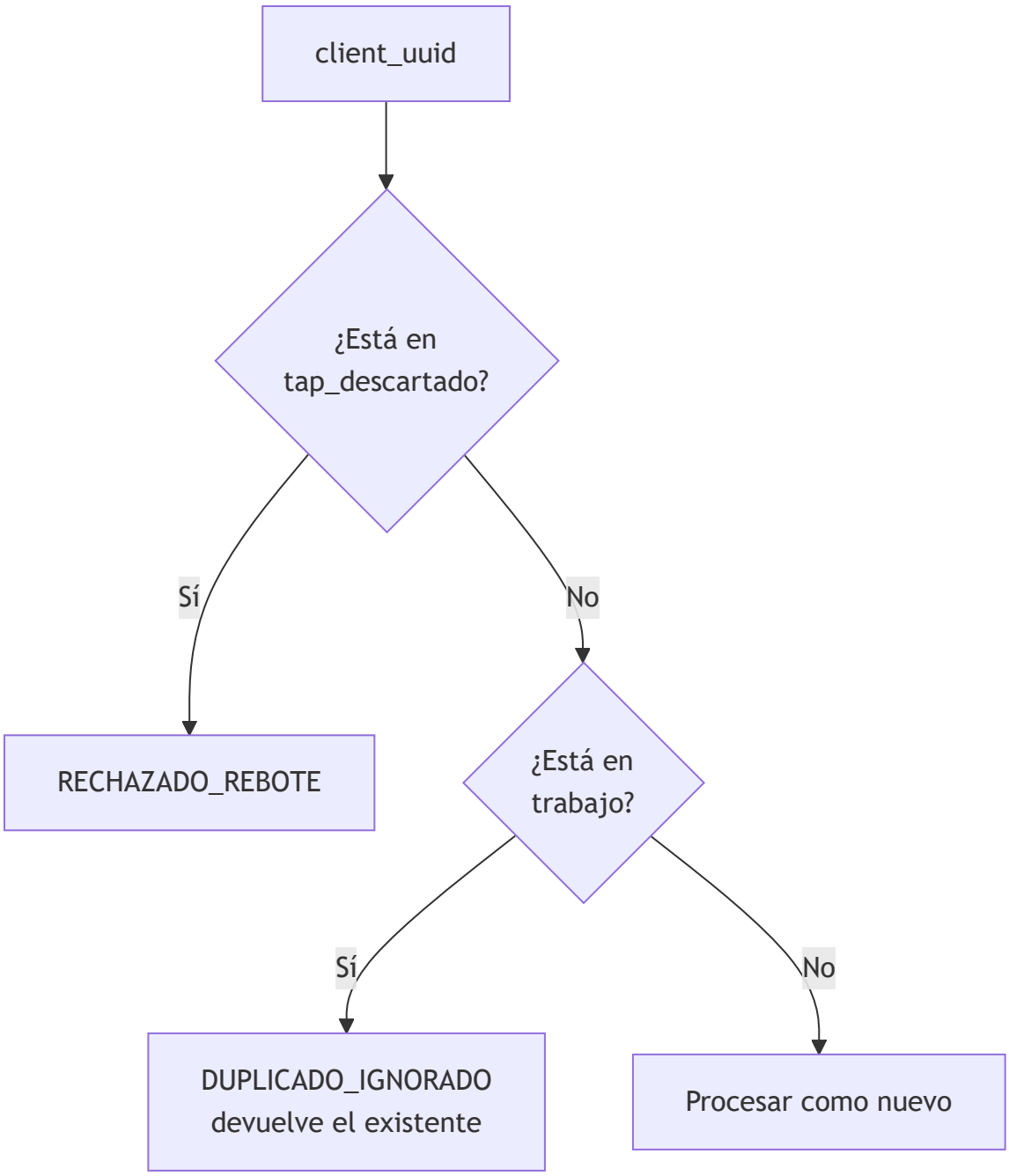
El protocolo exige que el cliente mande la cola en orden cronológico (`ts_local_device ASC`), pero el server **no se confía**: re-ordena la cola en memoria antes de procesar (`SyncService.java:252-253`):

```
List<TrabajoPendiente> ordenados = pendientes.stream()
    .sorted(Comparator.comparing(TrabajoPendiente::tsLocalDevice)).toList();
```

La cola NFC hace lo mismo (`SyncService.java:431-435` , con `Instant.EPOCH` como fallback si falta el timestamp). Esto importa para los casos donde el **orden decide el resultado**: el primer INGRESO/EGRESO en el tiempo gana, y en NFC el primero que asigna un sanitario gana (los demás reciben `CONFLICTO_ASIGNACION`).

Idempotencia de trabajos en 3 pasos (el orden importa)

`TrabajoTapService.procesar` (`TrabajoTapService.java:174-184`) chequea idempotencia en **tres consultas, en este orden exacto** —y el orden no es casual—:



1. `tap_descartado` **primero** (`findByClientUuid` → `RECHAZADO_REBOTE`): un reintento de un INGRESO ya rechazado por rebote debe responder `RECHAZADO_REBOTE` igual que la primera vez. Si chequeáramos `trabajo` primero, no lo encontraríamos y procesaríamos un INGRESO fantasma.
2. `trabajo` **después** (`findByClientUuid` → `DUPLICADO_IGNORADO`): devuelve el trabajo existente, no duplica eventos ni alertas.

3. **Recién si no existe en ninguna**, procesa como nuevo.

Throttling de login con dos buckets

`LoginThrottlingService` (`LoginThrottlingService.java:42-72`) mantiene **dos buckets** en una cache Caffeine in-memory:

Bucket	Clave	Para qué
Por usuario	`usuario`	Sobrevive a la rotación del device
Por par	`(usuario, device_uuid)`	Aísla el throttling de un device específico

El `device_uuid` lo manda el cliente **sin autenticar** (header `X-Device-UUID`). Si el contador fuera solo por par, un atacante rotaría el header en cada intento y **resetearía el contador** para bruteforcear el usuario. El bucket **por usuario** sobrevive esa rotación (ADR-039). Ambos buckets se incrementan en cada fallo (`recordFailure`) y se invalidan en un login exitoso (`recordSuccess`). El límite es 5 intentos → bloqueo de 10 min, y la entrada **expira ~1h sin nuevos fallos** (`expireAfterWrite(ATTEMPT_WINDOW_HOURS=1)`, `LoginThrottlingService.java:34`).

Rotación de refresh tokens

En cada login (`AppAuthService.doLogin`, `AppAuthService.java:145-151`) el server **revoca los refresh tokens anteriores** del mismo par antes de emitir uno nuevo:

```
refreshTokenRepository.revokeAllByOperarioAndDevice(operario, deviceUuid, now);
```

Tres decisiones de diseño:

- **Ligado al `device_uuid`**: el refresh trae el `device_uuid` y al refrescar se valida que coincida (`AppAuthService.java:118-126`). Usarlo en otro device es ilegítimo → `InvalidCredentialsException`. En la práctica: **cambiar de device = re-login**.
- **Rotación en cada refresh**: el `refresh()` revoca el token usado (`setRevokedAt`) y emite uno nuevo (`AppAuthService.java:128-131`). Un token de refresh es de un solo uso.
- **Hash SHA-256, no BCrypt** (`sha256Hex`, `AppAuthService.java:205-214`): el token se guarda hasheado para poder **buscarlo por igualdad** (`findByTokenHash`). SHA-256 es determinístico y rápido; BCrypt (con salt por fila) obligaría a un scan + verify y no permite lookup directo. El

token plano tiene suficiente entropía (`UUID.randomUUID()`), así que no necesita el work-factor de BCrypt.

Revocación de dispositivo inmediata

Cada `/sync` valida `dispositivo.activo` **por request, sin cache** (`SyncService.java:158-164`). Si un admin deshabilita un device (`activo=false`), el **próximo** sync de ese device se rechaza con `401 DEVICE_REVOCADO` —no hay ventana de cache que demore la revocación—. El cliente borra credenciales y se bloquea; su cola offline **no se acepta** hasta re-habilitarlo. Ver la nota sobre revocación en el modelo de datos.

Eventos de limpieza: CSV, array y exclusividad todo-o-nada

Dos detalles de los eventos de limpieza que viajan en el delta y se validan en el EGRESO:

- **`roles_permitidos`: CSV en la base, array en el wire.** La entidad guarda los roles como un CSV (`"OPERADOR,SUPERVISOR"`), pero la app espera `roles_permitidos` como **array JSON**. `splitRolesPermitidos` (`SyncService.java:644-650`) parte el CSV antes de serializar; emitir el CSV crudo rompería la deserialización Moshi del cliente. Null/blank → array vacío.
- **Exclusividad todo-o-nada.** Al validar los eventos de un EGRESO (`TrabajoTapService.java:413-419`), si **algún** evento es `es_exclusivo` y vino acompañado de otros, se rechaza **la combinación entera** y **no se persiste ningún** `trabajo_evento` (solo queda un warning). No es "ignoro el exclusivo y proceso el resto" ni al revés: o el exclusivo viene solo, o no entra nada.

Reglas operativas (parámetros y SLA)

Estos son los números y reglas de negocio que gobiernan el comportamiento del sistema; cambiarlos cambia la operación. Todo lo que sigue está verificado contra el código del backend (no contra documentos previos): cada fila apunta al archivo y la línea que la define. Si mañana alguien edita un default, este documento queda desactualizado salvo que se actualice junto al código.

> **Cómo leer esta página**
>
> Los **parámetros** son configurables vía `application.yml` (prefijos `app.tap`, `app.smiley`, `app.monitor`, `app.scheduler`). Los valores de la tabla son los **defaults de seguridad** que aplica el constructor de cada `record` cuando el valor inyectado es `<= 0` (o vacío). Es decir: aunque no configures nada, el sistema arranca con estos números.

---

1. Parámetros configurables de negocio

1.1. Algoritmo del tap — `app.tap`

Definidos en `TapProperties.java` (`@ConfigurationProperties(prefix = "app.tap")`). El constructor compacto del `record` fija el default si el valor es `<= 0`.

Parámetro	Default	Qué controla
`cierreAutomaticoMaxMinutes`	**30 min**	Si el operario olvida marcar salida, la duración del trabajo cerrado automáticamente se trunca a este tope (ADR-003).
`clockDriftToleranceMinutes`	**5 min**	Tolerancia entre `ts_local_device` y `server_received_ts`. Si se excede, agrega un warning pero **NO** rechaza el trabajo.

`durac ionMi nimaS eg`	**30 seg**	Piso de validez de una limpieza (ANEXO DT, DT-D2). Un EGRESO con duración menor cierra el trabajo con `duracion_anomala=1`: no cuenta como limpieza válida y genera Alerta `DURACION_ANOMALA`. Es un detector de tap accidental, no una vara de calidad fina.
`rebot eIngr esoSe g`	**8 seg**	Ventana anti-rebote server-side (DT-D3 / DT-B3). Un INGRESO a menos de este margen de un EGRESO del mismo operario+sanitario+device se descarta como doble-tap accidental.
`drift MaxSe g`	**300 seg (5 min)**	Umbral de drift de reloj del device (DT-D5). Si `|device_now - server_now|` lo excede, se estampa `drift_seg` y se genera Alerta `RELOJ_DESVIADO`.
`silen cioCl iente Seg`	**8 seg**	Config remota que viaja al cliente en el sync (DT-D1): tras procesar un tap, la app ignora lecturas del **mismo tag** por estos segundos.
`confi rmarE greso Min`	**2 min**	Config remota que viaja al cliente (DT-D4): si un EGRESO resuelve con duración menor a estos minutos, la app pide confirmación al operario.

Dos parámetros distintos para el reloj desviado

`clockDriftToleranceMinutes` (5 min) y `driftMaxSeg` (300 seg) coinciden en valor pero son cosas distintas: el primero decide el **warning** `clock_drift_warning` en el trabajo; el segundo decide la **Alerta** `RELOJ_DESVIADO`. No los unifiques pensando que es redundancia.

1.2. Anti-abuso del Smiley — `app.smiley`

Definidos en `SmileyProperties.java` (`prefix = "app.smiley"`).

		temporal).
`rafagaMax`	**5 opiniones**	Máximo de opiniones del **mismo valor** permitidas dentro de la ventana de ráfaga. Si hay más, se conserva 1 y el resto se marca `descartada=true, motivoDescarte="rafaga"`.
`rafagaVentanaSeg`	**30 seg**	Ventana de tiempo para el colapso de ráfaga.
`pinTecnicoDefault`	`2468`	PIN técnico GLOBAL del menú oculto de la terminal (S13/B3a). Es un secreto de deploy: el backend lo hashea con BCrypt al arrancar y solo entrega el hash en `/sync`. El override por empresa (`empresa.pin_tecnico_hash`) lo pisa si existe.

Colapso de ráfaga, en una frase

Hasta **5 opiniones del mismo valor en 30 segundos** se cuentan como **una sola**. Es la regla que evita que alguien martille el botón "feliz" 50 veces.

1.3. Estado de conexión de dispositivos — `app.monitor`

Definidos en `MonitorDispositivoProperties.java` (`prefix = "app.monitor"`). Derivan el estado de conexión a partir de `ultimoContactoTs`.

Parámetro	Default	Qué controla
`operarioEnLineaMin`	**30 min**	Minutos desde el último contacto por debajo de los cuales un `APP_OPERARIO` se considera `EN_LINEA`.
`operarioConRetrasoMin`	**120 min**	Umbral superior: entre 30 y 120 min un `APP_OPERARIO` está `CON_RETRASO`; por encima, offline.
`terminalEnLineaMin`	**15 min**	Minutos por debajo de los cuales una `TERMINAL_SMILEY` se considera `EN_LINEA`.

`terminalSinReportarMin`	**30 min**	Minutos a partir de los cuales una `TERMINAL_SMILEY` se considera `SIN_REPORTAR` (genera alerta G3).
`monitorCheckFixedRateMin`	**5 min**	Intervalo del scheduler G3 que detecta tablets caídas y auto-atiende las que reconectaron.

> **Terminal vs app operario tienen umbrales distintos**
>
> Una **terminal** se considera caída a los **30 min** (umbral único `SIN_REPORTAR`). Una **app de operario** tiene dos escalones: `EN_LINEA` (< 30 min), `CON_RETRASO` (30–120 min) y offline (> 120 min). No es el mismo número porque la terminal reporta de forma continua y el operario no.

---

2. "Limpieza válida" y `cuentaParaSla` — la regla más importante

Una limpieza **resetea el SLA del sanitario** (es decir, vuelve a poner el reloj de "última limpieza" en cero) **solo si cumple TODO** lo siguiente. Si falla **cualquiera** de los criterios, no cuenta y el sanitario sigue acumulando tiempo sin limpiar.

Evidencia: `TrabajoRepository.findUltimaLimpiezaValida` (`TrabajoRepository.java:95-108`) y su gemela `existsLimpiezaValidaById` (líneas 115-127), que aplican los mismos criterios.

#	Criterio	Campo / condición
1	Es una limpieza (no una supervisión)	`tipo = LIMPIEZA`
2	Está cerrada	`finTs IS NOT NULL`
3	No es de duración anómala (egreso corto, < `duracionMinimaSeg`)	`duracionAnomala = false`
4	Cuenta para el SLA	`cuentaParaSla = true`

5	Ningún evento la invalida	no existe `TrabajoEvento` con `eventoLimpieza.invalidaLimpieza = true`

> ⚠ **Los criterios son AND, no OR**
>
> Basta con que **uno** falle para que la limpieza no resetee el SLA. Una limpieza puede estar perfectamente cerrada y registrada (`finTs != null` , sin eventos) y aun así **no contar** si `cuentaParaSla = false` . Ese es exactamente el caso del trabajo colgado que se ve abajo.

2.1. El default y quién lo cambia

`Trabajo.cuentaParaSla` arranca en `Boolean.TRUE` por construcción de la entidad (`Trabajo.java:78-79`). Casi nadie lo baja a `false` . Solo dos caminos lo tocan:

Camino	Efecto sobre `cuentaParaSla`	Evidencia
Registro **MANUAL** (placa rota + motivo de contingencia)	`true` si NFC; si es MANUAL depende de `contingenciaProps.cuentaComoValida( )` (RB-D3, default `true`)	`TrabajoTapService.java:523`
Cierre por **scheduler** de trabajo colgado	**`false`** — forzado	`SlaSchedulerService.java:150`

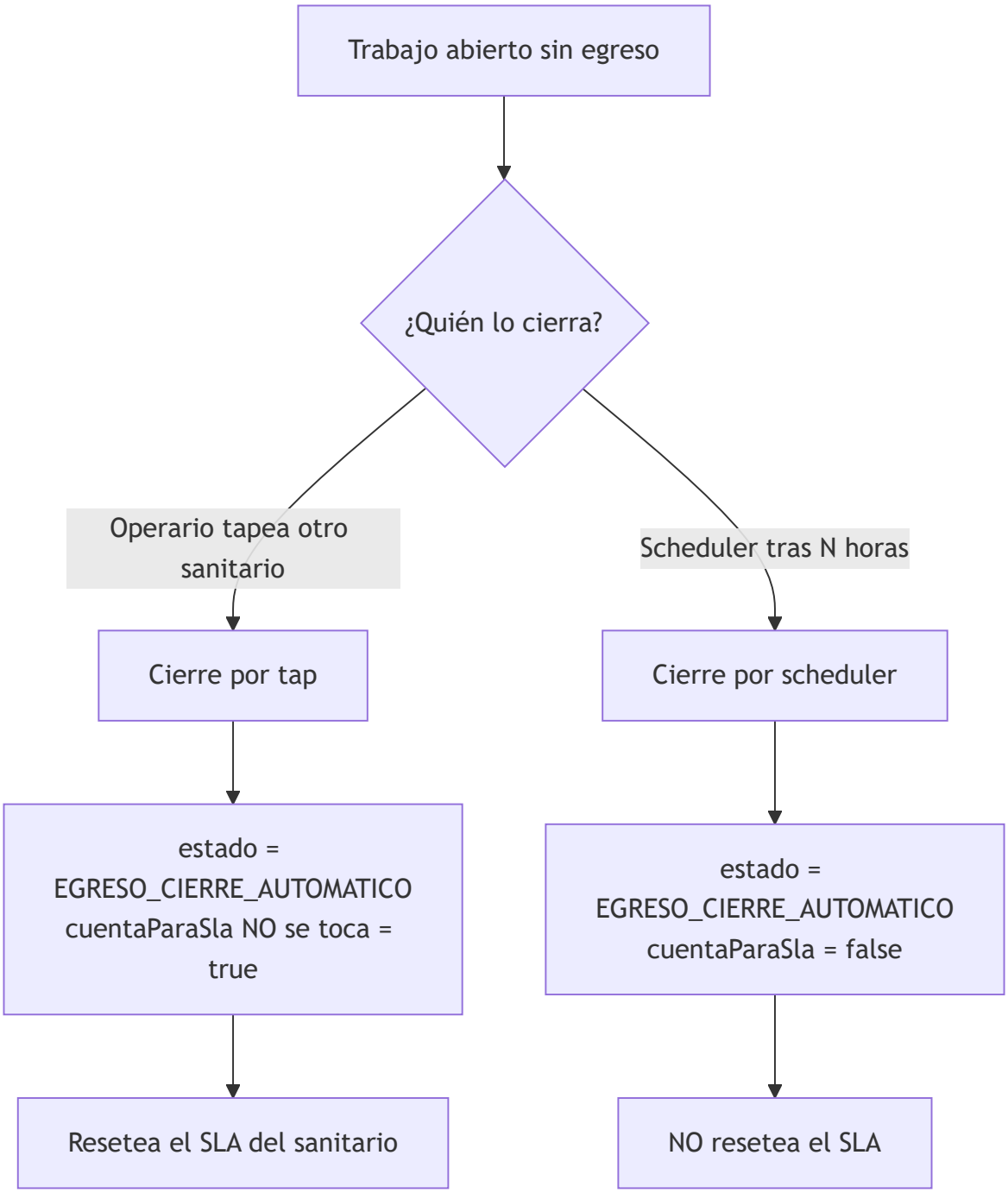
2.2. Los dos cierres automáticos que se comportan distinto

Esta es la causa de la confusión clásica: "hay dos cierres automáticos y se comportan distinto". Y es cierto. Ambos dejan el trabajo en estado `EGRESO_CIERRE_AUTOMATICO` y truncan la duración a `cierreAutomaticoMaxMinutes` (30 min), **pero solo uno toca `cuentaParaSla`** .

	Cierre automático por tap	Cierre por scheduler (colgado)
Qué lo dispara	El operario abre en sanitario A y **tapea B** sin cerrar A	El trabajo quedó abierto más de `app.scheduler.trabajos-colgados-max-hours` (horas) y nadie lo cerró

Quién lo ejecuta	`TrabajoTapService` (en el sync)	`SlaSchedulerService.cerrarTrabajosColgados()` (job `@Scheduled`, cada 1 hora)
Estado resultante	`EGRESO_CIERRE_AUTOMATICO`	`EGRESO_CIERRE_AUTOMATICO`
Duración	truncada a 30 min (o al tap de B si es antes)	truncada a 30 min (o a `now` si es antes)
`cuentaParaSla`	**NO se toca → queda `true`**	**se fuerza a `false`**
¿Resetea el SLA?	**SÍ** (es una limpieza real que el operario olvidó cerrar al instante)	**NO** (un colgado de horas no es una limpieza real medida)
Warning	`cierre_automatico`	`scheduler_cierre`
Evidencia	`TrabajoTapService.java:334-340`	`SlaSchedulerService.java:141-155`

> ⚠ **La distinción es deliberada (ADR-024)**
>
> El cierre por tap **sí** cuenta porque el operario efectivamente limpió A y se fue a B: la limpieza ocurrió, solo faltó el egreso explícito. El cierre por scheduler **no** cuenta porque un trabajo abierto durante horas no representa una limpieza medible — pudo haber sido un tap accidental al inicio del turno que quedó huérfano. El comentario en `SlaSchedulerService.java:147-150` lo dice textual: *"el cierre automático por siguiente ingreso (path del tap, ADR-024) sí cuenta: aquel no toca `cuentaParaSla`"*.



---

3. Cierre automático y truncado de duración

Cuando el operario abre en el sanitario A y **tapea B sin cerrar A**, el sistema cierra A automáticamente. El `fin_ts` no es simplemente "ahora": se calcula como el **mínimo** entre el tope

de 30 min y el momento del ingreso a B.

```
fin_ts(A) = min( inicio_ts(A) + cierreAutomaticoMaxMinutes , ts_ingreso(B) )
```

Evidencia: `calcularCierreAutomatico` (`TrabajoTapService.java:672-675`), invocado desde el flujo de cierre+ingreso (`TrabajoTapService.java:334-356`):

```
private Instant calcularCierreAutomatico(Instant inicioAbierto, Instant
tapActual) {
    Instant maxFin =
inicioAbierto.plus(Duration.ofMinutes(props.cierreAutomaticoMaxMinutes()));
    return tapActual.isBefore(maxFin) ? tapActual : maxFin;
}
```

Por qué el mínimo y no el tap directo

Si el operario abrió A a las 09:00 y recién tapea B a las 11:30, sería absurdo registrar 2h30 de limpieza en A. El tope de 30 min corta esa inflación. Pero si tapeó B a las 09:10, el cierre usa las 09:10 reales (10 min), no los 30 de tope. El tope es un **techo**, no un valor fijo.

---

4. Reglas de aceptación / rechazo del tap

El orden de validación importa: el rol se chequea **antes** de resolver el tag, y los operarios inactivos **no** se rechazan.

4.1. Operario con rol ADMIN → `ROL_NO_PERMITIDO` (no persiste nada)

Un operario con `rol = ADMIN` que tapea devuelve `RolNoPermitido` y **no persiste absolutamente nada**: ni el trabajo, ni un `TAG_DESCONOCIDO`, ni su alerta. El cliente ADMIN purga el tap sin reintentar.

Evidencia: `TrabajoTapService.java:200-203`.

```
RolOperario rol = operario.getRol() != null ? operario.getRol() :
RolOperario.OPERADOR_LIMPIEZA;
if (rol == RolOperario.ADMIN) {
    return new RolNoPermitido(rol);
}
```

> ⚠️ **Regla de seguridad de auditoría (2026-06-11)**
>
> El chequeo de rol va **antes** de resolver el tag justamente para que un ADMIN no deje rastro alguno. Si lo movieras después de la resolución del tag, un ADMIN tapeando una placa desconocida generaría un `TAG_DESCONOCIDO` persistido y su alerta — exactamente lo que esta regla evita. No reordenes este chequeo.

4.2. Operario inactivo → se ACEPTA (con warning + alerta)

Contraintuitivamente, un operario **inactivo** que tapea **no se rechaza**: el trabajo se crea igual, se le agrega el warning `operario_inactivo_offline` y se genera la Alerta `OPERARIO_INACTIVO_OFFLINE`.

Evidencia: el flag se calcula sin cortar el flujo en `TrabajoTapService.java:190`, se agrega el warning en la línea 194, y la alerta se emite tras el cierre/ingreso (líneas 257-261, 328, 355).

```
boolean operarioInactivo = !Boolean.TRUE.equals(operario.getActivo());
// ...
if (operarioInactivo) warnings.add(WARN_OPERARIO_INACTIVO);
```

> 🔥 **Por qué no se rechaza**
>
> El operario ya está físicamente en el sanitario y ya limpió — rechazar el dato perdería trabajo real hecho en campo. El sistema prefiere **registrar y alertar** antes que descartar. La inactividad es un problema administrativo (alguien lo dio de baja en el sistema pero sigue trabajando), no una razón para tirar evidencia.

4.3. Resumen de resoluciones del tap

Situación	Resultado (`TapResultado`)	¿Persiste trabajo?
Mismo `client_uuid` ya procesado	`Duplicado`	No (devuelve el existente)
Rol ADMIN	`RolNoPermitido`	**No (nada)**

Operario inactivo	sigue el flujo normal (`Ingreso` / `Egreso` /...)	**Sí** + warning + alerta
Tag no `ASIGNADO` (stock/baja/inexistente)	`TagDesconocido`	Sí (con tag nulo) + alerta
Sanitario inactivo o borrado	`SanitarioInactivo`	No
Rebote post-egreso (< 8 seg)	`RechazadoRebote`	No (registra `tap_descartado`)
MANUAL sin sanitario/motivo válido	`ManualInvalido`	No

---

5. Acumulación de warnings

El trabajo guarda sus warnings en una sola columna `warnings NVARCHAR(500)`, concatenados con `|`, y **truncados a 500 caracteres**. No es una tabla de eventos: es un string plano.

Constantes definidas en `TrabajoTapService.java:78-86`; concatenación y truncado en `appendWarnings` (`TrabajoTapService.java:680-694`):

```
private static void appendWarnings(Trabajo t, List<String> warnings) {
    if (warnings == null || warnings.isEmpty()) return;
    String joined = String.join("|", warnings);
    if (t.getWarnings() == null || t.getWarnings().isBlank()) {
        t.setWarnings(joined);
    } else {
        t.setWarnings(t.getWarnings() + "|" + joined);
    }
    // Mantener dentro del límite de 500 chars
    if (t.getWarnings().length() > 500)
t.setWarnings(t.getWarnings().substring(0, 500));
}
```

Warnings posibles:

Constante	String almacenado	Cuándo se agrega

`WARN_OPERARIO_INACTIVO`	`operario_inactivo_offline`	Operario dado de baja que tapea
`WARN_CLOCK_DRIFT`	`clock_drift_warning`	Drift entre device y server > 5 min
`WARN_CIERRE_AUTOMATICO`	`cierre_automatico`	Cierre por tap en otro sanitario
`WARN_TAG_DESCONOCIDO`	`tag_desconocido`	Tag no `ASIGNADO`
`WARN_SIN_EVENTOS`	`sin_eventos`	EGRESO sin eventos (B2)
`WARN_EVENTOS_EXCLUSIVO_COMBINADO`	`eventos_exclusivo_combinado`	Evento exclusivo combinado con otros
`WARN_EVENTO_RECHAZADO_PREFIX`	`evento_rechazado:<ref>`	Evento inexistente/inactivo/borrado
`WARN_EVENTO_NO_PERMITIDO_PREFIX`	`evento_no_permitido:<codigo>`	Evento no permitido para el rol
`scheduler_cierre` (`WARN_SCHEDULER_CIERRE`)	`scheduler_cierre`	Cierre por scheduler de colgado (`SlaSchedulerService`)

> ⚠ **El truncado a 500 puede perder warnings**
>
> Si un trabajo acumula muchos warnings (varios `evento_rechazado:` con refs largas, por ejemplo), el string se corta a 500 caracteres y los últimos warnings **se pierden silenciosamente**. No confíes en la columna `warnings` como auditoría completa cuando hay alta cardinalidad de eventos — para eso están las alertas y la tabla de eventos.

Ver también

- Sincronización — el protocolo de sync donde se procesan los taps y se aplican estas reglas.

- Jobs y alertas — los schedulers (`SlaSchedulerService` , monitor G3) que cierran colgados y derivan estados de conexión.
- Modelo de datos — los campos `cuenta_para_sla` , `duracion_anomala` , `warnings` y la tabla de eventos.

Jobs de fondo y alertas

WorkDone Sanitarios no es un sistema puramente reactivo: no todo nace de un tap sobre una placa NFC. Hay un conjunto de **procesos periódicos** (`@Scheduled`) que corren en el backend cada cierto tiempo, sin que nadie los dispare, y que se encargan de **generar alertas**, **cerrar estado colgado** y **limpiar datos**.

Esta página explica esos jobs, el catálogo real de alertas, cómo viven y mueren las alertas (incluida la deduplicación y la auto-atención), y los estados de conexión que el backend deriva de cada dispositivo.

> **Fuente de verdad**
>
> Todo lo que sigue está verificado contra el código del backend (`ar.com.lubeca.workdone`). Las rutas de archivo y líneas citadas son las reales al momento de escribir esta página.

1. Schedulers `@Scheduled`

Cada job es un método `@Scheduled` + `@Transactional` **independiente**: si uno falla, su transacción hace rollback sola y no afecta a los demás. La cadencia es configurable por properties (se muestra el default).

Job	Clase	Cadencia (default)	Qué hace
`revisarSlaVencido`	`SlaScheduleService`	cada **5 min** (`app.scheduler.sla-check-fixed-rate-min`)	Recorre los SLA activos. Si el último EGRESO válido de un sanitario excede `frecuencia_min` , genera una alerta `SLA_VENCIDO` (con dedup).
`cerrarTrabajosColgados`	`SlaScheduleService`	cada **1 hora** (fijo)	Cierra trabajos abiertos hace demasiado tiempo, trunca la duración y los marca como no

			contables para SLA. Ver detalle abajo.
`limpiarRefreshTokensExpirados`	`SlaSchedulerService`	**cron 3 AM** (`app.scheduler.refresh-token-cleanup-cron`)	Borra físicamente los refresh tokens cuyo `expires_at` quedó fuera de la ventana de retención.
`verificarTerminalesSinReportar`	`DispositivoMonitorService`	cada **5 min** (`app.monitor.monitor-check-fixed-rate-min`)	Recorre las terminales Smiley activas; genera o auto-atiende `DISPOSITIVO_SIN_REPORTAR`.
`evaluarTendencias`	`SmileyTendenciaService`	cada **5 min** (`app.scheduler.tendencias-check-fixed-rate-min`)	Evalúa cada terminal Smiley contra su regla de tendencia; genera o auto-atiende `TENDENCIA_NEGATIVA`.
`generarEjecucionesDelDia`	`RutaEjecucionGeneratorService`	**cron 4 AM** (`app.scheduler.rutas-generar-ejecuciones-cron`)	Cierra como `INCOMPLETA` las ejecuciones de ruta de días anteriores y genera las del día (idempotente).
`removeNotActivatedUsers`	`UserService`	**cron 1 AM** (fijo)	Limpieza JHipster estándar: borra usuarios no activados.

Single-instance asumido

El scheduler de Spring corre single-thread, así que un mismo job nunca se solapa consigo mismo. El deployment hoy es **una sola instancia EC2** — por eso el check-then-insert anti-duplicado basta. Con escalado horizontal el check sería *race-prone* (TOCTOU); el backstop a nivel datos es el índice único filtrado `UX_alerta_sla_vencido_activa` (migration 029, ADR-035).

`cerrarTrabajosColgados` en detalle

Busca trabajos abiertos cuyo `inicio_ts` es anterior a `now - app.scheduler.trabajos-colgados-max-hours` y los cierra con estado `EGRESO_CIERRE_AUTOMATICO`. Sobre el cierre:

- **Trunca** el `fin_ts` al máximo de `app.tap.cierre-automatico-max-minutes` (no inventa una duración gigante).
- Setea `cuenta_para_sla = false`: un trabajo colgado durante horas **no es una limpieza real medida**, así que no debe contar para el SLA.
- Agrega el warning `scheduler_cierre` al trabajo.

> ⚠ **No confundir con el cierre automático por tap**
>
> Hay **dos** cierres automáticos distintos:
>
> - **Por scheduler** (este job): el trabajo quedó colgado y el sistema lo cierra de oficio → `cuenta_para_sla = false`.
> - **Por siguiente ingreso (tap)**: cuando llega un nuevo INGRESO en el mismo sanitario y había uno abierto, se cierra el anterior. Ese **sí** cuenta para el SLA (ADR-024) y NO toca `cuenta_para_sla`.
>
> El detalle de ambos vive en Reglas operativas.

2. Generación de alertas

El catálogo real es el enum `TipoAlerta` (`domain/enumeration/TipoAlerta.java`). Estas son las **9** clases de alerta y quién las dispara:

`TipoAlerta`	Condición que la dispara	Quién la genera
`SLA_VENCIDO`	El último EGRESO válido del sanitario excede `frecuencia_min` (y hoy es día con exigencia).	`SlaSchedulerService.revisarSlaVencido` (job 5 min)
`DURACION_ANOMALA`	Un EGRESO con duración menor a `duracion-minima-seg` — posible doble tap.	`TrabajoTapService` (al procesar el tap)
`OPERARIO_INACTIVO_OFFLINE`	Se acepta un trabajo sincronizado de un operario marcado como inactivo.	`TrabajoTapService` (al procesar el tap)
`TAG_DESCONOCIDO`	Tap sobre un tag NFC no habilitado; el trabajo queda sin sanitario, pendiente de	

	resolución manual.	`TrabajoTapService` (al procesar el tap)
`EVENTO_REPORTADO`	El operario reporta un evento al cerrar un trabajo (ej. falta de insumos).	`TrabajoTapService` (al procesar el tap)
`RELOJ_DESVIADO`	El `device_now` del dispositivo difiere del server más de `drift-max-seg`.	`SyncService.procesarDriftReloj` (durante el sync)
`TAG_DEFECTUOSO`	Se registra un trabajo en modo **MANUAL** porque la placa NFC del sanitario no funciona.	`TrabajoTapService` (al procesar el tap)
`TENDENCIA_NEGATIVA`	El conteo/proporción de opiniones NEGATIVA supera el umbral de la regla en la ventana.	`SmileyTendenciaService.evaluarTendencias` (job 5 min)
`DISPOSITIVO_SIN_REPORTAR`	Una terminal Smiley activa pasa al estado `SIN_REPORTAR`.	`DispositivoMonitorService.verificarTerminalesSinReportar` (job 5 min)

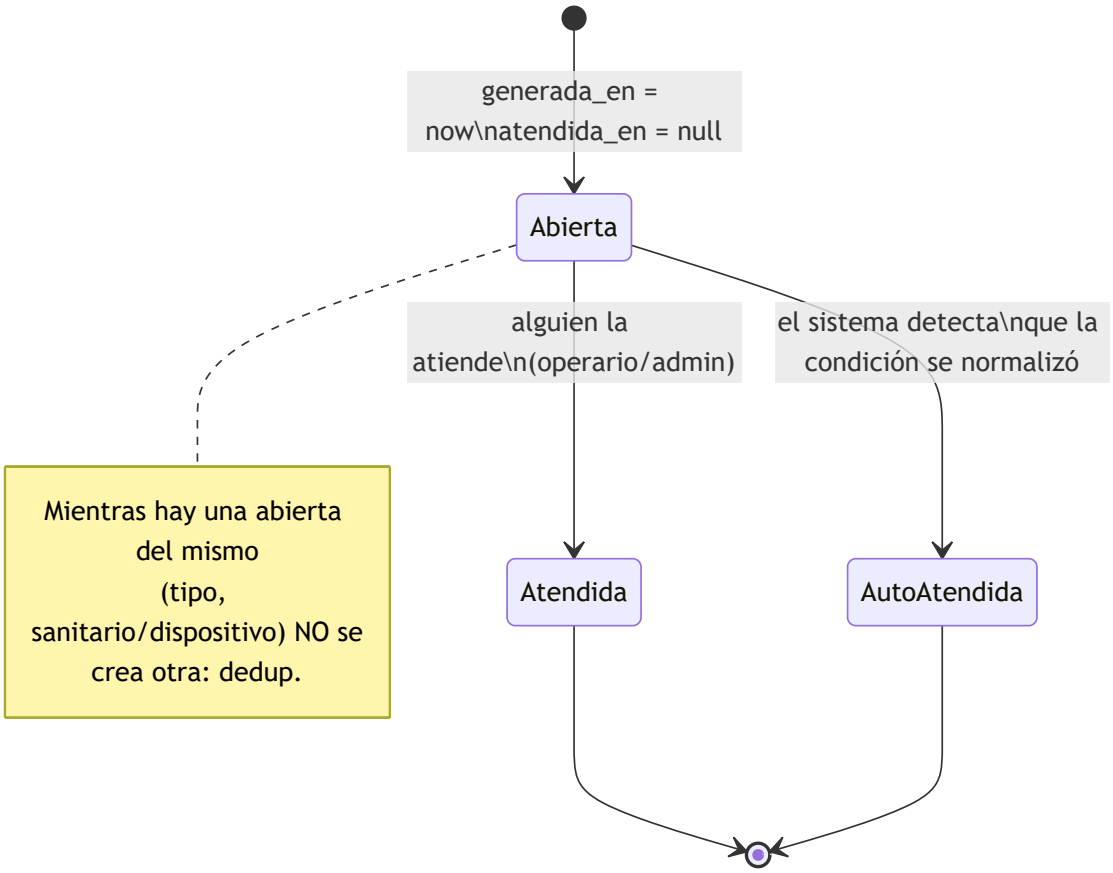
Reactivas vs. periódicas

Las que genera `TrabajoTapService` / `SyncService` nacen **reactivamente** mientras se procesa lo que mandó un dispositivo. Las tres restantes (`SLA_VENCIDO`, `TENDENCIA_NEGATIVA`, `DISPOSITIVO_SIN_REPORTAR`) nacen **periódicamente** desde un job, sin tap de por medio.

3. Ciclo de vida de una Alerta

Una alerta tiene dos timestamps clave:

- `generada_en` — cuándo se creó.
- `atendida_en` — cuándo se cerró. **`null` significa que está abierta.**



Deduplicación

Antes de crear una alerta, el backend chequea si ya existe una **abierta equivalente** y, si la hay, no crea una nueva. La clave de equivalencia es `(tipo, sujeto, atendida_en IS NULL)`, donde el sujeto es el sanitario o el dispositivo según el tipo:

```
// SlaSchedulerService.java:112
if
(alertaRepository.existsByTipoAndSanitario_IdAndAtendidaEnIsNull(TipoAlerta.SLA_V
ENCIDO, s.getId())) continue;
```

Este patrón se repite igual en `TENDENCIA_NEGATIVA`, `DISPOSITIVO_SIN_REPORTAR`, `RELOJ_DESVIADO` y `TAG_DEFECTUOSO`. Resultado: **una sola alerta abierta por sujeto y tipo**.

Auto-atención

Algunas alertas las cierra el **propio sistema** (`atendida_por_usuario = "sistema"`) cuando detecta que la condición que las generó se normalizó. No requieren intervención humana.

`TipoAlerta`	Se auto-atiende cuando…	Evidencia
`RELOJ_DESVIADO`	el drift del reloj del dispositivo vuelve por debajo del umbral.	`SyncService.java:228-244`
`DISPOSITIVO_SIN_REPORTAR`	la terminal vuelve a reportar (estado ≠ `SIN_REPORTAR`).	`DispositivoMonitorService.java:163-183`
`TENDENCIA_NEGATIVA`	hay una limpieza válida **posterior** a `generada_en` de la alerta.	`SmileyTendenciaService.java:129-148`
`TAG_DEFECTUOSO`	llega un trabajo NFC posterior en ese sanitario (la placa volvió a leerse).	`TrabajoTapService.java:602-613`

La auto-atención corre ANTES de evaluar el umbral

En `evaluarTendencias` la auto-atención va deliberadamente **antes** del chequeo de umbral. Si hubo una limpieza correctiva pero siguen llegando negativas en la nueva ventana, en la misma ejecución se cierra la alerta vieja y se abre una nueva. Es decir: la limpieza no silencia indefinidamente.

4. Estados derivados de conexión del dispositivo

El backend **no persiste** el estado de conexión: lo **deriva en memoria** a partir de `ultimo_contacto_ts` y `now`, sin un hit extra a la base. La lógica vive en `DispositivoMonitorService.derivarEstado` (`DispositivoMonitorService.java:90-108`).

Si `ultimo_contacto_ts` es `null` → `NUNCA_CONECTADO`. En el resto, los umbrales dependen del **tipo** de dispositivo:

Tipo	`EN_LINEA`	`CON_RETRASO`	`SIN_REPORTAR`
`APP_OPERARIO` (celular)	< 30 min	< 120 min	≥ 120 min
`TERMINAL_SMILEY` (tablet)	< 15 min	< 30 min	≥ 30 min

Los valores son los defaults de `MonitorDispositivoProperties` y son configurables (`app.monitor.*`).

> ⚠ **Solo las terminales Smiley generan alerta por estar caídas**
>
> El job `verificarTerminalesSinReportar` **solo itera** `TERMINAL_SMILEY`. Un celular `APP_OPERARIO` en `SIN_REPORTAR` **no** genera `DISPOSITIVO_SIN_REPORTAR`: el operario puede estar offline a propósito entre turnos, y eso se gestiona por inactividad de SLA, no por conexión (decisión G-D4). Más sobre las terminales en Kiosk Smiley.

5. Tendencia negativa (Smiley)

La detección de tendencias vive en `SmileyTendenciaService`. Lo no obvio es **cómo se delimita la ventana de análisis** y **qué opiniones cuentan**.

Ventana de análisis

La ventana arranca en el **más reciente** entre:

1. el `fin_ts` de la última limpieza válida del sanitario, y
2. `now - ventanaMin` (el inicio "natural" de la ventana configurada).

```
// SmileyTendenciaService.java:191-195
Instant calcularVentanaInicio(Instant now, int ventanaMin, Instant
ultimaLimpieza) {
    Instant porVentana = now.minusSeconds(ventanaMin * 60L);
    if (ultimaLimpieza == null) return porVentana;
    return ultimaLimpieza.isAfter(porVentana) ? ultimaLimpieza : porVentana;
}
```

La consecuencia: una limpieza reciente **reinicia el conteo**. Las negativas previas a esa limpieza dejan de pesar, porque ya fueron "respondidas" por la limpieza correctiva.

Qué opiniones cuentan

Las opiniones con `origen = QR_PUBLICO` se **EXCLUYEN** de las tendencias. Solo cuentan las de origen `TERMINAL_SMILEY`. Las consultas de `OpinionRepository` filtran explícitamente:

```
// OpinionRepository.java
"  AND o.origen =
ar.com.lubeca.workdone.domain.enumeration.OrigenOpinion.TERMINAL_SMILEY "
```

ADR-041

Las opiniones por QR público son "de segunda clase" y nunca alimentan el motor de tendencias (regla de oro V22_QR_PUBLICO). Sí se contabilizan aparte para estadística, pero jamás disparan una `TENDENCIA_NEGATIVA`.

Para seguir

- Reglas operativas — cierres automáticos, SLA y semántica de los taps.
- Modelo de datos — entidad `Alerta`, enums y relaciones.
- Kiosk Smiley — terminales, opiniones y orígenes.

Resumen. WorkDone corre 7 jobs `@Scheduled` independientes que generan alertas y mantienen el estado sin depender de taps. El catálogo real son los 9 valores de `TipoAlerta`; cada alerta vive con `generada_en` / `atendida_en` (null = abierta), se deduplica por `(tipo, sujeto, atendida_en IS NULL)` y algunas el sistema las **auto-atiende** al normalizarse la condición. El estado de conexión se **deriva** de `ultimo_contacto_ts` con umbrales distintos para celular (30/120) y terminal (15/30). Para tendencias, la ventana arranca en la última limpieza válida y **solo cuentan opiniones** `TERMINAL_SMILEY` (ADR-041).

Architecture Decision Records (ADRs)

Registro de decisiones de arquitectura del ecosistema WorkDone. Cada ADR documenta **una** decisión: el contexto, las opciones, lo que se eligió y por qué.

Una decisión registrada no se borra ni se edita: si cambia, se crea un ADR nuevo que supersede al anterior. Así queda el rastro del razonamiento.

Formato

Un archivo por decisión: `NNNN-titulo-corto.md` (ej. `0001-repos-separados.md`).

Plantilla mínima:

```
# ADR-NNNN — Título de la decisión

- **Estado:** propuesto | aceptado | supersedido por ADR-XXXX
- **Fecha:** YYYY-MM-DD

## Contexto
Qué problema o fuerza nos llevó a decidir.

## Decisión
Qué elegimos.

## Consecuencias
Qué gana y qué pierde el proyecto por esta decisión.
```

De dónde sale este registro

Las decisiones viven en el `DECISIONS.md` de cada repo (`workdone_backend` y `workdone_mobile`), consolidado como **v2**. Esta página sintetiza ese registro para onboarding: el índice completo abajo y, separadas, las decisiones más formativas para entender CÓMO se programa el sistema. Para el texto íntegro y la historia de cada ADR, la fuente de verdad es el `DECISIONS.md` del repo correspondiente.

Cómo leer los estados

`VIGENTE` / `ACCEPTED` rigen el código hoy. `PROPUESTO` está sobre la mesa pero no implementado. Cuando un ADR dice "supersede" a otro, el viejo se conserva como historia pero ya no manda. Los ADR `001-017` se consolidaron en v1; sus textos completos están en el historial de git, acá quedan en la tabla por referencia.

Las cuatro reglas de oro

Antes de los ADRs conviene tener presentes los cuatro invariantes que atraviesan casi todas las decisiones. Aparecen citados por número a lo largo del registro:

#	Regla de oro	Dónde pega
1	El **tap** (NFC) es el único gesto de registro: toggle inicio/fin	ADR-001/002, motor de Trabajo
2	En el device solo se persiste el **refresh_token, idealmente cifrado** (matizada por ADR-028 mobile)	Auth móvil
3	El **timestamp del teléfono es la verdad**	Sync, SLA, drift
4	Los operarios son **soft-delete**: nunca se borran físicamente	Catálogos, FKs

Índice de decisiones — Backend

Repo `workdone_backend`. Núcleo del dominio, sync, seguridad y los cuatro módulos de v2.1/v2.2/v2.3.

ADR	Título	Estado	Qué decide en una línea
001	NFC sobre QR	VIGENTE	El registro se hace por chip NFC, no por QR escaneado

002	Tap = toggle, tag único por sanitario	VIGENTE (ref. 019)	Un tap abre o cierra el Trabajo; un tag por sanitario
003	Cierre automático con duración truncada	VIGENTE (ampl. 022)	Trabajo olvidado abierto se cierra solo
004	Concurrencia de operarios permitida	VIGENTE	Varios operarios pueden trabajar en paralelo
005	**Offline-first, sync bidireccional**	VIGENTE	La app opera sin red; sincroniza en ambos sentidos
006	Device compartido, usuario+PIN, sesión 30 min	VIGENTE	Un teléfono lo comparten operarios del turno
007	Tag como URL opaca	VIGENTE	El tag codifica una URL no adivinable
008	**Single-tenant**	VIGENTE (prec. 018)	Una sola instancia, de Corpal
009	**Stack** (Java 25 · SB 4.0.6 · JHipster 9.1 · MSSQL · Kotlin/Compose/Room)	VIGENTE	Elección de tecnologías base
010	Soft delete en catálogos	VIGENTE	Las bajas no borran filas
011	Auditoría JHipster en entidades	VIGENTE	`created_by` / `last_modified_by` automáticos
012	Timestamp del teléfono = verdad	VIGENTE	El reloj del device manda sobre el del server
013	Idempotencia por `client_uuid`	VIGENTE (ext. 020)	Reintentos no duplican registros

014	Operario separado de `jhi_user`	VIGENTE (ext. 021)	El operario de campo no es el usuario web
015	Ícono NFC en placa física	VIGENTE	Señalización física del punto de tap
016	Testing NFC sin emulador	VIGENTE	Mocks + panel dev + device físico
017	**Virtual threads para `/sync`**	PROPUESTO	Aún no implementado
018	Jerarquía Empresa › Sucursal › Sector › Sanitario	ACCEPTED	Empresa = cliente del servicio, no la limpieza
019	Máximo 1 tag NFC activo por sanitario	ACCEPTED	Constraint DB `UNIQUE WHERE estado='ASIGNADO'`
020	Ciclo de vida del tag (stock, whitelist, auditoría)	ACCEPTED	Máquina de estados + alta solo desde la app
021	**Tres roles de campo en Operario**	ACCEPTED	ADMIN / SUPERVISOR / OPERADOR_LIMPIEZA
022	**Supervisión = mismo motor que limpieza**	ACCEPTED	`Trabajo.tipo`, sin entidad separada
023	Catálogo de eventos configurable con flags	ACCEPTED	Eventos como filas, no enum
024	**"Limpieza válida" como única fuente del SLA**	ACCEPTED	Define qué resetea el reloj de vencimiento
025	Deudas v1 saldadas en la migración	ACCEPTED	Normalización de UUID, alertas, auditoría
026	Endurecimiento de seguridad post-auditoría	ACCEPTED	CRUD scaffold → `ROLE_ADMIN`, 409 en conflicto

027	Duración mínima como property global	ACCEPTED	`app.tap.duracion-minima-seg`, detector de doble-tap
028	Portal de transparencia del cliente	ACCEPTED	Solo lectura, `ROLE_CLIENTE`, scoping server-side
029	Robustez operativa (V21_ROBUSTEZ)	ACCEPTED	Días sin exigencia, contingencia de placa, mail, revocación
030	Smiley — opinión del público	ACCEPTED	Tablet kiosko, terminal como `Dispositivo` tipado
031	Planificación por rutas (V21_RUTAS)	ACCEPTED	La ruta planifica, nunca bloquea el tap
032	Flecos del code review v21 (SLA y rate-limit)	ACCEPTED	Decisiones de owner sobre semántica del SLA
033	`updatedAt` opcional en entrada de catálogos	ACCEPTED	Lo completa el server si viene null
034	Relaciones anidadas de DTOs incluyen `nombre`	ACCEPTED	El BackOffice muestra nombre, no id
035	Backstop de unicidad para alerta SLA_VENCIDO	ACCEPTED	Índice único filtrado contra race del scheduler
036	**Parity dev↔MSSQL + gate de CI**	ACCEPTED	Migrations validadas sobre SQL Server real
037	Sacar credenciales scaffold de JHipster en prod	ACCEPTED	Elimina `user`, resetea `admin`, fail-closed
038	`/sync` valida `device_uuid` del body contra el JWT	ACCEPTED	No se puede sincronizar como otro device

039	Throttling de login también por usuario	ACCEPTED	Cierra el bypass rotando `device_uuid`
040	**Authority separada del operario móvil** (`ROLE_OPERARIO`)	ACCEPTED	Separa la identidad móvil de la web
041	Opinión por QR público NO alimenta tendencias	ACCEPTED	El QR mide, no dispara correctivas
042	Estado público con umbral, anonimato, slug opaco	ACCEPTED	Página `/p/{slug}` sin tracking
043	Planos y mapeo: capa georreferenciada opcional	ACCEPTED	Módulo desprendible, plano → pin → sanitario
R1	Anti-fraude de ubicación	RIESGO ASUMIDO	Sin GPS en el tap para el piloto
R2	PIN prestado entre operarios	RIESGO ASUMIDO	Problema organizacional, no técnico
R3	Pérdida de cola offline	RIESGO ASUMIDO	Inherente al offline-first; sync inmediato lo acota

Backlog del backend (disparador definido)

Quedan registrados con su gatillo: **B1** SLA por franja horaria · **B2** Push FCM · **B3** Multi-idioma del kiosk · **B4** Auditoría de acciones admin · **B5** Retención y archivado · **B6** QR público (ya promovido a V22) · **B7** Subdominio/theming del portal · **B8** Visual-regression del BackOffice · **B9** Contraste a11y (WCAG AA) · **B10** N+1 en dashboard/portal.

Índice de decisiones — Mobile

Repo `workdone_mobile` (Kotlin/Compose/Room). Su `DECISIONS.md` comparte los ADR `001-025` con el backend, pero a partir del **026** numera decisiones PROPIAS del cliente Android (telemetría, purga, tokens, backup). Cuidado: los números 026-029 NO son los mismos que en el backend.

ADR	Título	Estado	Qué decide en una línea
001-025	(compartidos con backend)	VIGENTE / ACCEPTED	Ver índice del backend
026	Telemetría del device en el sync	ACCEPTED	`red_tipo` WIFI/CELULAR/..., sin distinguir 4G/5G
027	Cliente Android NO purga por `deleted_ids`	ACCEPTED	Traduce la baja a `activo=false`
028	**Tokens en claro en el device** (riesgo aceptado)	ACCEPTED	DataStore sin cifrar durante el piloto
029	Backup de Android deshabilitado	ACCEPTED	`allowBackup=false` para evitar restores corruptos
R1-R3	(riesgos compartidos)	RIESGO ASUMIDO	Mismos que el backend

Backlog del mobile

B1 SLA por franja horaria · **B2** Push FCM · **B3** Multi-idioma del kiosk · **B4** Auditoría admin · **B5** Retención · **B6** QR como segunda vía de opinión · **B7** Subdominio/theming del portal.

A continuación, las decisiones que más afectan CÓMO se programa en este ecosistema. Si sos dev nuevo, leé estas primero: explican por qué el código tiene la forma que tiene.

ADR-005 (backend) · Offline-first con sync bidireccional

Contexto. El personal de campo limpia baños en lugares con conectividad inestable (subsuelos, depósitos, shoppings). No se puede depender de la red en el momento del tap.

Qué se decidió. La app opera 100% offline y sincroniza en ambos sentidos cuando hay red. Lo que el operario registra (taps, eventos, gestión NFC del admin) se encola localmente y se envía después; el server devuelve catálogos, rutas y deltas hacia el device.

Consecuencias. Esta es la decisión más estructurante de todo el sistema. Arrastra el resto:

- Toda escritura del cliente necesita **idempotencia por** `client_uuid` (ADR-013): un reintento no puede duplicar un Trabajo.
- El **timestamp del teléfono es la verdad** (ADR-012/regla de oro #3), porque el evento ocurrió offline y el server lo recibe minutos después.
- Hay **colas locales** por dominio (`trabajo_pendiente` , `nfc_evento_pendiente`) y **resolución de conflictos** al sincronizar (primero gana, `CONFLICTO_ASIGNACION`).
- El **riesgo R3** (pérdida de cola si el device se destruye antes de sincronizar) se asume conscientemente, acotado por el sync inmediato tras cada tap.

> ⚠ **Implicancia para el dev**
>
> Nunca asumas que el server vio un registro en el momento en que ocurrió. Programá pensando en "esto llega tarde, puede reintentarse, y el reloj de origen es el del device".

ADR-009 (backend) · Stack tecnológico

Contexto. Sistema nuevo que necesita backend robusto con auditoría y un cliente móvil de campo.

Qué se decidió. Backend **Java 25 · Spring Boot 4.0.6 · JHipster 9.1 · MSSQL (SQL Server)**. Cliente móvil **Kotlin · Jetpack Compose · Room**.

Consecuencias. Varias decisiones posteriores son consecuencia directa del stack:

- JHipster genera **CRUD scaffold** y entidades con auditoría (ADR-011) — pero también arrastra deuda: credenciales `admin/admin` por defecto (ADR-037), DTOs con `@NotNull updatedAt` (ADR-033), endpoints abiertos (ADR-026).
- **MSSQL** tiene semánticas propias (un solo NULL en UNIQUE, cascade paths, ALTER con índice) que el dev H2 oculta — ver ADR-036.
- **Room** en el cliente impone que las colas offline sean device-local.

ADR-014 + ADR-021 + ADR-040 · Identidad: operario de campo ≠ usuario web

Contexto. JHipster trae `jhi_user` para el BackOffice. Pero el operario de campo es otra cosa: comparte device, se loguea con PIN, tiene rol operativo.

Qué se decidió (evolución en tres pasos).

1. **ADR-014:** `Operario` es una tabla separada de `jhi_user` (link `user_id` opcional). El usuario web no es el operario.
2. **ADR-021:** el operario tiene un campo `rol` (`ADMIN | SUPERVISOR | OPERADOR_LIMPIEZA`, uno solo). El rol define la experiencia en la app: operador limpia, supervisor controla, admin gestiona NFC.
3. **ADR-040:** se crea la authority `ROLE_OPERARIO` para el JWT móvil, distinta de `ROLE_USER` (web). Antes `ROLE_USER` hacía dos trabajos y un token web podía colarse en endpoints de app interpretando su `jhi_user.id` como `operario.id`.

Consecuencias. La autorización es **siempre server-side y por rol**. Los endpoints `/api/v1/app/**` exigen `ROLE_OPERARIO`; el CRUD `/api/*` exige `ROLE_ADMIN` (ADR-026). Si tocás un endpoint, sabé qué authority lo protege y desde qué cliente viene.

ADR-022 + ADR-023 + ADR-024 · El motor de Trabajo: un solo flujo, eventos como datos, SLA derivado

Estas tres juntas explican el corazón del dominio.

ADR-022 — Supervisión usa el mismo motor que la limpieza. No hay entidad `Supervision`. Es un `Trabajo` con `tipo = LIMPIEZA | SUPERVISION` (lo define el rol del autenticado). Mismo flujo de tap inicio/fin, timer y eventos. Se descartó un checklist con severidad: el multiselect de eventos es más operable en campo y reutiliza todo el motor.

ADR-023 — Los eventos al cierre son datos, no código. Son filas de `evento_limpieza` (catálogo sincronizado) con cuatro flags de comportamiento: `roles_permitidos`, `es_exclusivo`, `genera_alerta`, `invalida_limpieza`. Al cerrar, la selección se persiste con **snapshot** de código+nombre, así el histórico es inmune a cambios futuros del catálogo. Corpal agrega eventos sin release de la app.

ADR-024 — El SLA tiene una única fuente: la "limpieza válida". El reloj de vencimiento se resetea **solo** con un `Trabajo tipo=LIMPIEZA` cerrado sin eventos `invalida_limpieza`. Por lo tanto: "no se pudo limpiar" NO resetea y alerta; la supervisión nunca resetea; el cierre automático sí.

Por qué importa para el dev

Si vas a tocar cierre de trabajos, eventos o el SLA, partí de estas tres. La definición de "limpieza válida" es transversal: la usan el scheduler, el dashboard, el portal cliente y el módulo de rutas. Cambiarla sin entenderla rompe métricas en cascada.

ADR-026 (backend) · Endurecimiento de seguridad post-auditoría

Contexto. Una auditoría adversarial sobre el diff de v2 encontró agujeros reales de autorización antes del piloto.

Qué se decidió. El más importante: **todo CRUD scaffold `/api/*` pasa a `ROLE_ADMIN`**. Antes, el JWT del operario (`USER`) atravesaba el `.authenticated()` genérico y leía datos de todos los operarios y hacía DELETE físico. El operario móvil usa SOLO `/api/v1/app/*`. Además: tap de ADMIN no persiste nada (check de rol antes de resolver el tag), conflicto NFC concurrente devuelve 409, y varias defensas en profundidad.

Regla operativa derivada. Todo endpoint nuevo bajo `/api/*` que no sea `/api/v1/app/*` **nace cerrado a ADMIN** salvo decisión explícita. Internalizá esto antes de agregar un controller.

ADR-031 (backend) · Rutas: planifican, nunca bloquean

Contexto. Se suma planificación de trabajo por rutas (qué recorrer hoy, en qué orden).

Qué se decidió (principio rector que gobierna todo el módulo). La ruta es una capa de planificación **ARRIBA** del sistema de taps, nunca un bloqueo. El tap libre sigue idéntico: cualquier operario registra cualquier sanitario. Un trabajo fuera de ruta no es un error, es información. **Ninguna validación de ruta puede impedir un tap.**

Cómo se implementó sin tocar el tap. Los cierres de Trabajo estaban dispersos en 3 puntos. En vez de tocarlos, los 3 publican `TrabajoCerradoEvent` y un `@TransactionalEventListener(AFTER_COMMIT)` aparte hace la vinculación trabajo↔parada. Un fallo de vinculación **nunca** hace rollback del cierre.

Patrón recurrente: efectos secundarios post-commit

Este mismo patrón (publicar evento + listener `AFTER_COMMIT` `@Async`, fallo aislado del flujo principal) aparece en el mail de alertas (ADR-029) y en la auto-atención de Smiley (ADR-030). Es la forma estándar de colgar comportamiento sin contaminar el motor de taps.

ADR-036 (backend) · Parity dev↔MSSQL + gate de CI

Contexto. Correr dev contra un SQL Server real destapó que **ninguna migration se había validado nunca sobre MSSQL**: el job "Testcontainers MSSQL" en realidad corría H2. Tres divergencias MSSQL-vs-H2 quedaron como deuda latente de prod.

Qué se decidió. Se corrigieron las tres divergencias (UNIQUE sobre columna nullable, ALTER con índice dependiente, múltiples cascade paths) y se agregó un **gate permanente de CI** (`schema-mssql`) que aplica el changelog entero sobre SQL Server real en cada push.

Acción concreta si trabajás con migrations

El dev usa H2 file-based (`target/h2db/`). Editar una migration cambia su checksum y rompe el arranque con `ValidationFailedException`. **Antes de actualizar tras un cambio de migration:** `mvn clean` **(o borrar** `target/h2db`**).** Y recordá: si tu migration usa una operación MSSQL-incompatible, H2 no la atrapa — el gate de CI sí, antes del merge.

ADR-028 (mobile) · Tokens en claro en el device durante el piloto

Contexto. El `SessionStore` del cliente persiste `access_token`, `refresh_token` y `device_uuid` en DataStore Preferences **sin cifrar**, contradiciendo la regla de oro #2 ("solo refresh_token, cifrado"). `androidx.security.crypto` está declarada pero sigue en **alpha** con problemas en Android 14+.

Qué se decidió. Para el piloto, los tokens quedan en claro. NO se adopta `androidx.security.crypto` mientras esté en alpha: cambiaría un riesgo bajo y entendido (texto plano en device compartido) por uno mayor (fallas de descifrado de una lib alpha que dejarían la sesión irrecuperable). El threat model lo habilita: el device es **compartido entre operarios del turno**, no es target de robo, y nunca se persiste password ni PIN.

Consecuencias. El vector por backup ya está cerrado (ADR-029, `allowBackup=false`). Disparadores de revisión: hardening pre-release, exigencia del cliente, o versión estable de la lib.

Nota de evolución (2026-06-15)

El disparador (c) se cumplió: ya hay `security-crypto:1.1.0` estable. Pero trae señales de deprecación de Google, así que queda pendiente un spike (¿Keystore directo? ¿Tink?) y una decisión explícita. No bloquea el piloto. Es un buen ejemplo de cómo el registro de ADRs mantiene honesta la deuda: el código y la regla de oro dejaron de contradecirse en silencio.

Modelo de datos

Si sos nuevo en **WorkDone Sanitarios**, este es el mapa que tenés que tener a mano. Todo el sistema gira alrededor de una idea simple: el personal de limpieza acerca su teléfono (o una terminal) a una placa **NFC** pegada en un sanitario y eso registra un **trabajo**. El resto del modelo existe para darle contexto a ese tap: dónde está el sanitario, quién lo hizo, con qué placa, qué pasó adentro y si cuenta para el SLA.

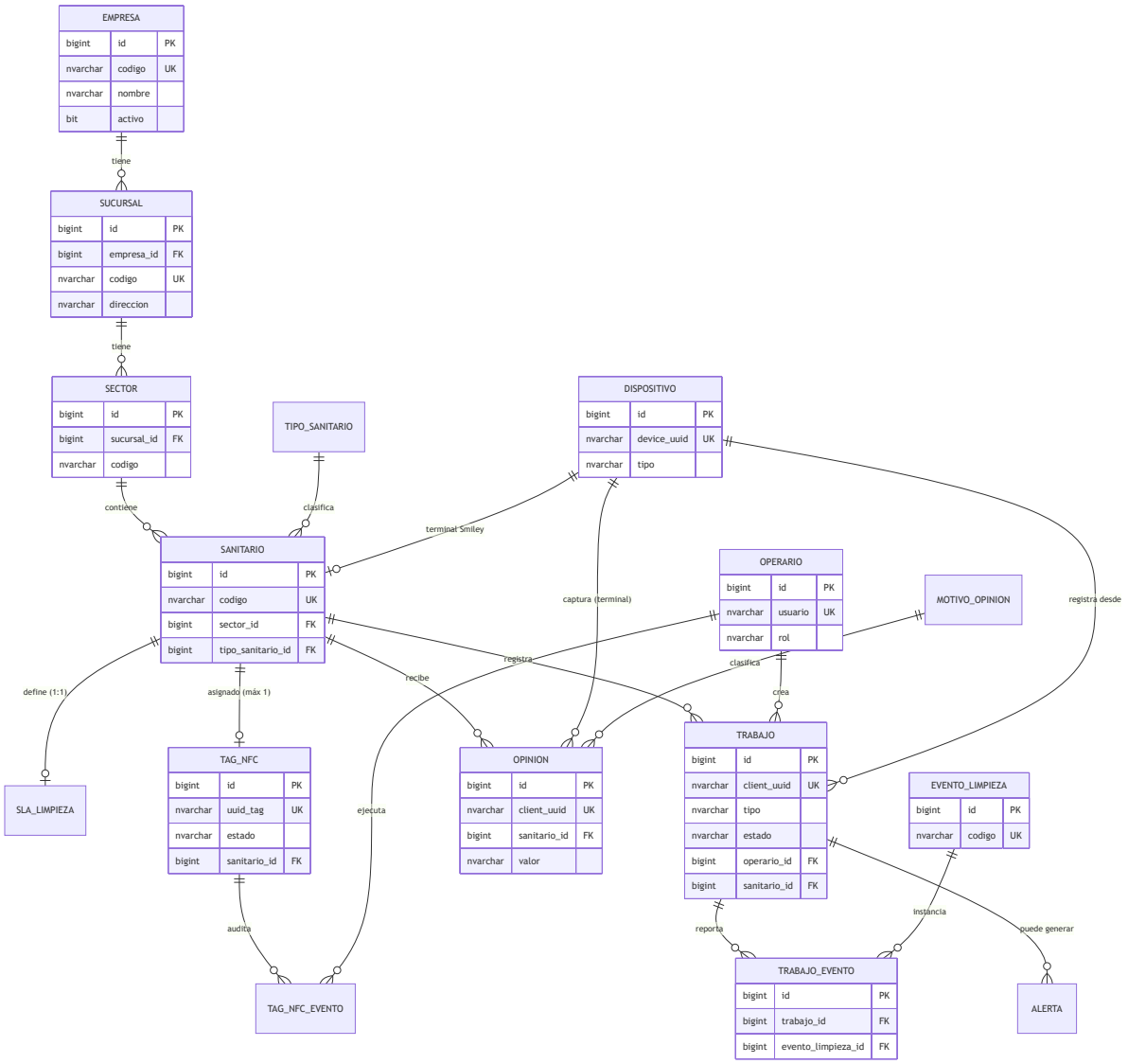
Esta página describe el modelo de datos **v2**, que reemplaza al v1. La **fuente de verdad** son las tablas MSSQL del backend (Spring Boot / JHipster). El nombre de dominio va siempre **en español** (`sanitario`, `trabajo`, `operario`) — nunca se renombra a inglés. La app móvil Android mantiene un **espejo local** (Room) de un subconjunto de estas tablas para funcionar offline; las diferencias están en la sección Espejo móvil.

Convenciones globales

- Las tablas de **catálogo** (Empresa, Sucursal, Sector, TipoSanitario, Sanitario, TagNfc, Operario, Dispositivo, EventoLimpieza, SlaLimpieza) tienen `id BIGINT IDENTITY PK`, `updated_at DATETIME2 NOT NULL` (deltas de sync) y `deleted_at DATETIME2 NULL` (soft delete). Las queries normales filtran el soft delete; **el sync NO** (propaga las bajas).
- Llevan auditoría JHipster (`AbstractAuditingEntity`): `created_by`, `created_date`, `last_modified_by`, `last_modified_date`.
- Las tablas **transaccionales / de eventos** (Trabajo, TrabajoEvento, TagNfcEvento, Alerta, Opinion) son **inmutables** y **sin soft delete**.
- Todo el texto va en `NVARCHAR`. Para los timestamps del dominio, **el del teléfono al momento del tap es la verdad** (`inicio_ts` / `fin_ts`); el server guarda aparte `server_received_ts`.

Diagrama de entidades

El siguiente diagrama muestra las entidades principales y sus relaciones. La jerarquía física baja por la izquierda (Empresa → Sucursal → Sector → Sanitario) y la operación NFC cuelga del Sanitario.



> **El diagrama es un resumen**
>
> Muestra solo las claves y FKs más relevantes para entender las relaciones. Las tablas de cada entidad, más abajo, listan los campos completos que aparecen en las fuentes. Entidades de soporte (TagNfcEvento, Alerta, MotivoOpinion, ReglaTendencia, rutas, planos) se documentan en texto, no todas en el ER.

La jerarquía física

Toda la infraestructura cuelga de una cadena de cuatro niveles. **No es multi-tenant**: la Empresa es el cliente del servicio, no un aislamiento de datos.

```
Empresa (cliente del servicio, ej. AA2000 — NO multi-tenant)
  └── Sucursal (ej. AEP, EZE)
        └── Sector (ej. Terminal A, Patio de comidas)
              └── Sanitario (+ TipoSanitario)
```

- **Empresa** — el cliente contratante (ej. AA2000). Tiene un `codigo` único global.
- **Sucursal** — una sede del cliente (ej. AEP, EZE). Pertenece a una Empresa.
- **Sector** — una zona dentro de la sucursal (ej. Terminal A, Patio de comidas). Su `codigo` es único **dentro de la sucursal** (UK compuesta `sucursal_id, codigo`).
- **Sanitario** — el baño concreto donde va la placa NFC. Su `codigo` es único **global** (ej. `"AEP-T-B12"`). Se clasifica opcionalmente con un **TipoSanitario**.

> **El Sanitario es el centro de gravedad**
>
> La historia de limpiezas, las opiniones y el SLA viven en el **Sanitario**, no en la placa. Una placa se puede desasignar, dar de baja o reemplazar, y el sanitario conserva todo su histórico. Por eso, cuando reemplazás un tag, no se pierde nada.

Empresa

Columna	Tipo	Descripción
codigo	NVARCHAR(20) UK	Código único global de la empresa
nombre	NVARCHAR(100) NN	Nombre del cliente
activo	BIT NN	Si está operativa
visible_opiniones	BIT NN default 0	Si el portal cliente muestra opiniones Smiley de esta empresa
logo_cliente_url / logo_cliente_hash		Branding de la terminal Smiley por empresa

texto_pregunta / texto_gracias		Textos de la terminal Smiley por empresa
pin_tecnico_hash	NVARCHAR (BCrypt)	Override del PIN técnico del menú oculto del kiosk; nunca en texto plano
umbral_publico_min	INT NN default 90	Minutos máx. para mostrar "Limpiado hace X min" al público (QR)
mostrar_estado_publico	BIT default 1	Kill-switch del estado público por empresa

Sucursal

Columna	Tipo	Descripción
empresa_id	FK NN → Empresa	Empresa a la que pertenece
codigo	NVARCHAR(20) UK	Código de la sucursal
nombre	NVARCHAR(100) NN	Nombre de la sede
direccion	NVARCHAR(255)	Dirección física
activo	BIT NN	Si está operativa

Sector

Columna	Tipo	Descripción
sucursal_id	FK NN → Sucursal	Sucursal contenedora · UK compuesta `(sucursal_id, codigo)`
codigo	NVARCHAR(20) NN	Código del sector (único dentro de la sucursal)

nombre	NVARCHAR(100) NN	Nombre del sector
activo	BIT NN	Si está operativo

TipoSanitario

Catálogo chico. Seed: `DAMAS`, `CABALLEROS`, `DISCAPACITADOS`, `MIXTO`.

Sanitario

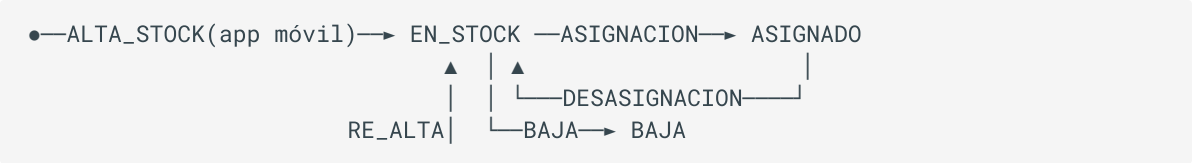
Columna	Tipo	Descripción
codigo	NVARCHAR(30) UK	Código único global, ej. `"AEP-T-B12"`
nombre	NVARCHAR(100) NN	Nombre del sanitario
descripcion	NVARCHAR(255)	Descripción libre
sector_id	FK NN → Sector	Sector contenedor (reemplaza al `ubicacion_id` v1)
tipo_sanitario_id	FK NULL → TipoSanitario	Clasificación opcional
activo	BIT NN	Si está operativo
slug_publico	VARCHAR(64) UK NN	Identificador opaco para la URL pública del QR (`/p/{slug}`), generado con SecureRandom

Cambio v1 → v2

En v1 existía una tabla `ubicacion`. En v2 desaparece: se descompone en **Empresa + Sucursal + Sector**, y el Sanitario ahora apunta a `sector_id`.

La placa NFC y su máquina de estados

Cada placa física es un registro **TagNfc** identificado por su `uuid_tag` (normalizado: trim/upper/sin espacios en todo punto de entrada). El sistema usa **whitelist estricta**: un tag no registrado es inutilizable. Un tap con UUID desconocido genera un trabajo `OFFLINE_PENDIENTE_RESOLUCION` y una alerta `TAG_DESCONOCIDO`.



- **Alta de stock SOLO desde la app móvil** (rol ADMIN — hay que leer el chip onsite).
- Asignar / desasignar / reasignar / baja: app o BackOffice web. **Re-alta: solo web.**
- **Máximo 1 tag ASIGNADO por sanitario.** Para reemplazar: desasignar → asignar.
- **BAJA solo desde EN_STOCK** (desasignar primero).

TagNfc

Columna	Tipo	Descripción
uuid_tag	NVARCHAR(64) UK	UUID normalizado del chip; UNIQUE global
alias	NVARCHAR(100) NN	Descriptivo, ej. "tag #14 lote mayo"
estado	NVARCHAR(20) NN	`EN_STOCK` | `ASIGNADO` | `BAJA`
sanitario_id	FK NULL → Sanitario	Solo cuando ASIGNADO; al desasignar queda NULL (la historia vive en TagNfcEvento)
asignado_en / dado_baja_en	DATETIME2 NULL	Marcas temporales de transición
activo	BIT NN	Legado v1, derivado (`= estado=='ASIGNADO'`); se mantiene por compat de DTO

Constraints MSSQL: `UNIQUE(uuid_tag)` y `UNIQUE(sanitario_id) WHERE estado='ASIGNADO'` (índice filtrado que enforza "máx. 1 tag asignado por sanitario").

TagNfcEvento (auditoría dedicada, inmutable)

Registra cada transición de la máquina de estados.

Columna	Descripción
tag_nfc_id FK NN · uuid_tag snapshot	Tag afectado
accion	`ALTA_STOCK` | `ASIGNACION` | `DESASIGNACION` | `REASIGNACION`[1] | `BAJA` | `RE_ALTA`
sanitario_anterior_id / sanitario_nuevo_id	FK NULL — origen/destino de la asignación
operario_id	FK NULL (null solo en backfill) — quién lo hizo
dispositivo_id	FK NULL — si fue desde móvil
fecha · notas	Cuándo y observaciones

Diferencia backend vs. mobile

[1] El **backend** define la acción `REASIGNACION` (1 evento con anterior + nuevo). El **MODEL del mobile** lista solo `ALTA_STOCK | ASIGNACION | DESASIGNACION | BAJA | RE_ALTA` (sin `REASIGNACION`). Tomá el backend como fuente de verdad.

Usuarios, dispositivos y roles

Operario

El **Operario** es la identidad del personal en la app móvil. Su `rol` decide qué tipo de trabajo crea.

Columna	Tipo	Descripción
usuario	NVARCHAR(50) UK	Login móvil
nombre / apellido	NVARCHAR(100) NN	Nombre del operario
rol	NVARCHAR(30) NN	`ADMIN` | `SUPERVISOR` | `OPERADOR_LIMPIEZA` (nuevo v2)
pin_hash / password_hash	NVARCHAR(128) NN	BCrypt cost 12 · `@JsonIgnore` · **NUNCA viajan al móvil**
telefono / email / ultimo_login		Datos de contacto y último acceso
user_id	FK NULL → jhi_user	Link opcional; `jhi_user` sigue siendo SOLO BackOffice web
activo	BIT NN	Si está habilitado

Semántica de roles en la app:

- `OPERADOR_LIMPIEZA` crea trabajos `LIMPIEZA`.
- `SUPERVISOR` crea trabajos `SUPERVISION`.
- `ADMIN` **no** crea trabajos por tap — su app es la de Gestión NFC.

Dispositivo

El **Dispositivo** es el equipo físico: un celular con la app del operario o una **terminal Smiley** (kiosk). Una parte de sus columnas son de telemetría y provisioning.

Columna	Tipo	Descripción
device_uuid	UK	Identificador del equipo
nombre / alias		Nombre amigable

api_key_hash		Credencial del equipo
tipo		`APP_OPERARIO` | `TERMINAL_SMILEY` (default `APP_OPERARIO`)
sanitario_id	FK NULL	Solo terminales — sanitario al que está pegada la terminal
sonido_activo / linea_diagnostico		Config de terminal Smiley
codigo_vinculacion / codigo_expira_en / codigo_usado_en		Provisioning de terminal (`codigo_vinculacion` con `@JsonIgnore`)
numero_terminal	INT UNIQUE	Correlativo global asignado al alta
ultimo_sync / ultima_ip / ultimo_contacto_ts		Telemetría de conexión
app_version / os_version / bateria_pct / red_tipo		Telemetría opcional recibida en el sync
drift_seg	INT NULL	Último drift de reloj (seg) medido en el sync cuando el device manda `device_now`
lock_task_activo	BIT NULL	Confinamiento Lock Task del kiosk: `true` confinado, `false` no, `null` no reportado
activo	BIT NN	Si está habilitado

Constraint: `UNIQUE(sanitario_id) WHERE tipo='TERMINAL_SMILEY' AND activo=1` — máx. 1 terminal activa por sanitario (solo prod; H2 no lo enforza).

RefreshToken acompaña al login: `token_hash`, `operario_id`, `device_uuid`, `issued_at`, `expires_at`, `revoked_at`, con rotation.

Trabajo: la entidad central

El **Trabajo** es el corazón del sistema: un registro por turno de **limpieza o supervisión** (mismo motor, distinto `tipo`). Se crea por **tap NFC** (el tap funciona como toggle: el primero abre `INGRESO`, el segundo cierra `EGRESO`).

Columna	Tipo	Descripción
client_uuid	NVARCHAR(64) UK	Idempotencia; generado en el device ANTES de cualquier I/O
tipo	NVARCHAR(20) NN	`LIMPIEZA` | `SUPERVISION` (lo define el rol del operario)
uuid_tag_leido	NVARCHAR(64) NN	Snapshot crudo del tap
operario_id	FK NN	Quién lo hizo
dispositivo_id	FK NN	Desde qué equipo
sanitario_id / tag_id	FK NULL	NULL si el tag es desconocido (pendiente de resolución)
inicio_ts	NN	Timestamp del teléfono al abrir (= la verdad)
fin_ts / duracion_seg	NULL	Cierre y duración (NULL mientras abierto)
estado		`INGRESO` | `EGRESO` | `EGRESO_CIERRE_AUTOMATICO`
modo_registro		`ONLINE` | `OFFLINE_SYNC` | `OFFLINE_PENDIENTE_RESOLUCION` | `MANUAL` [2]

ts_local_device / server_received_ts	NN	Reloj del device vs. reloj del server (drift > 5 min → warning)
warnings	NVARCHAR(500)	Acumulativo
duracion_anomala	BIT NN default 0	`true` si el EGRESO tuvo duración < `app.tap.duracion-minima-seg` (piso de validez)
cuenta_para_sla	BIT NN default 1	Si la fila cuenta para el SLA. NFC siempre `true`; el colgado cerrado por scheduler lo pone en `false`
motivo_contingencia_id / motivo_contingencia_codigo	FK NULL + snapshot	Solo en registro `MANUAL` (sin placa)

El registro MANUAL es parte de V21_ROBUSTEZ

[2] El valor `MANUAL` de `modo_registro` (registro sin placa) y los campos `motivo_contingencia_*` aparecen **solo en el MODEL del backend** (sección Robustez operativa). El catálogo `motivo_contingencia` siembra `PLACA_ROTA`, `PLACA_AUSENTE`, `NO_LEE`, `OTRO`. El registro manual cuenta como limpieza válida según `app.contingencia.cuenta-como-valida` (default true).

TrabajoEvento (puente con snapshot, inmutable)

Cuando se cierra un trabajo, el operario reporta uno o más **eventos**. La relación N:N entre Trabajo y EventoLimpieza se materializa acá, con snapshot del catálogo para preservar el histórico:

```
trabajo_id FK · evento_limpieza_id FK · codigo_snapshot · nombre_snapshot · observacion NULL · created
```

Eventos de limpieza

EventoLimpieza es un **catálogo configurable** (no un enum) de qué puede reportar el operario al cerrar.

Columna	Descripción
codigo UK · nombre · orden · activo	Identidad y orden del evento
roles_permitidos	NVARCHAR(100) NN — CSV de roles que pueden usarlo, ej. `"OPERADOR_LIMPIEZA,SUPERVISOR"` [3]
es_exclusivo	BIT — si se selecciona, debe ser el único (ej. `SIN_NOVEDAD`)
genera_alerta	BIT — al cerrar el trabajo crea `Alerta(EVENTO_REPORTADO)`
invalida_limpieza	BIT — la LIMPIEZA que lo incluye NO resetea el SLA
visible_cliente	BIT NN default 0 — si el evento es visible en el portal del cliente

Seed:

codigo	roles	exclusivo	alerta	invalida
SIN_NOVEDAD	ambos	✓	—	—
ROTURAS	ambos	—	✓	—
REPONER_INSUMOS	ambos	—	✓	—
NO_SE_PUDO_LIMPIAR	solo OPERADOR_LIMPIEZA	—	✓	✓
NO_ESTA_LIMPIO	solo SUPERVISOR	—	✓	—

Reglas: mínimo 1 evento en un EGRESO normal · `EGRESO_CIERRE_AUTOMATICO` va SIN eventos · la validación de rol y exclusividad es **server-side** (no se confía en la app).

Diferencia backend vs. mobile en roles_permitidos

[3] El **backend** define `roles_permitidos` como CSV concreto (ADR-023) y agrega `visible_cliente`. El **MODEL del mobile** describe `roles_permitidos` de forma genérica ("qué roles lo ven/usan, representación elegida en implementación") y **no** menciona `visible_cliente`.

SLA y alertas

SlaLimpieza

Define la exigencia de limpieza por sanitario (relación 1:1, `sanitario_id` es UK).

```
sanitario_id FK UK · frecuencia_min · ventana_desde/hasta · duracion_minima_seg · activo
```

V21_ROBUSTEZ agrega `dias_semana CHAR(7) NN default 'LMXJVSD'` (patrón Lun..Dom): un día sin su letra queda en estado `SIN_EXIGENCIA` (sin alertas SLA). Default = todos los días (comportamiento v1 intacto).

Qué es una limpieza válida

Solo una **limpieza válida** resetea el reloj del SLA. La definición v2 (backend) es: Trabajo `tipo=LIMPIEZA` cerrado (`EGRESO` o `EGRESO_CIERRE_AUTOMATICO`), **sin** eventos con `invalida_limpieza=true` **y con** **`duracion_anomala=false`**. Las `SUPERVISION` nunca resetean. El cierre automático SÍ resetea (lleva warning).

El **MODEL del mobile** describe la misma definición **sin** la condición `duracion_anomala=false` (esa robustez del ANEXO DT vive solo del lado backend).

Alerta (inmutable)

Columna	Descripción
sanitario_id	FK NULL (v2: habilita `TAG_DESCONOCIDO` sin sanitario)

dispositivo_id	FK NULL — alertas de equipo (ej. `RELOJ_DESVIADO`), no de sanitario
tipo	`SLA_VENCIDO` | `DURACION_ANOMALA` | `OPERARIO_INACTIVO_OFFLINE` | `TAG_DESCONOCIDO` | `EVENTO_REPORTADO` | `RELOJ_DESVIADO`[4] | `DISPOSITIVO_SIN_REPORTAR`[4]
mensaje · generada_en · atendida_en · atendida_por_usuario	Contenido y atención (`'sistema'` en auto-atención)

Tipos de alerta solo-backend

[4] Los tipos `RELOJ_DESVIADO` (ANEXO DT), `DISPOSITIVO_SIN_REPORTAR` (terminal Smiley sin contacto > umbral, V21_SMILEY/G3) y los de robustez `TAG_DEFECTUOSO` / `TENDENCIA_NEGATIVA` aparecen **solo en el MODEL del backend**. El MODEL del mobile lista hasta `EVENTO_REPORTADO`.

Smiley y opiniones del público

Las terminales **Smiley** (kiosk) y los **QR públicos** capturan la opinión del usuario del baño. Esta capa vive **solo en el MODEL del backend** (V21_SMILEY / V22_QR_PUBLICO); el MODEL del mobile no la documenta.

Opinion (transaccional, inmutable)

Columna	Tipo	Descripción
client_uuid	UK	Idempotencia de la opinión
sanitario_id	NN	Sanitario opinado
dispositivo_id	FK NULL	Nullable: se setea para opiniones de terminal, queda NULL para QR público

origen	NN default TERMINAL_SMILEY	`TERMINAL_SMILEY` | `QR_PUBLICO`
valor		`POSITIVA` | `NEUTRA` | `NEGATIVA`
motivo_id	FK NULL + `motivo_codigo_snapshot`	Motivo de la opinión
ts_local_device / server_received_ts		Reloj del device / del server
descartada / motivo_descarte		Colapso anti-abuso: las ráfagas se marcan, no se borran

> ⚠ **Regla de oro de las opiniones QR (ADR-041)**
>
> Las opiniones con `origen=QR_PUBLICO` son de **segunda clase**: NUNCA alimentan el motor de tendencias ni las stats de confianza. El motor filtra `TERMINAL_SMILEY`.

Entidades de soporte del módulo:

- **MotivoOpinion** — catálogo (`codigo` UK, nombre, orden, activo, soft delete, auditoría). Seed: `SUCIO`, `SIN_INSUMOS`, `MAL_OLOR`, `ALGO_ROTO`, `OTRO`.
- **ReglaTendencia** — define cuándo una racha de negativas dispara `TipoAlerta.TENDENCIA_NEGATIVA`. `sanitario_id` FK NULL (null = regla global; la específica pisa la global), `ventana_min`, `modo` (`CONTEO` | `PROPORCION`), `umbral_negativas`, `umbral_pct` + `min_opiniones`. Seed: 1 regla global CONTEO 3/30.

Anti doble-tap y robustez (solo backend)

El **ANEXO DT** agrega la tabla **tap_descartado** (inmutable): un INGRESO que rebota a < `rebote-ingreso-seg` de un EGRESO del mismo operario+sanitario+device se registra acá con motivo `REBOTE_POST_EGRESO` y **no** crea Trabajo (el sync responde `RECHAZADO_REBOTE`, idempotente). Campos: `client_uuid` UK, `operario_id` / `dispositivo_id` FK NN, `sanitario_id` FK NULL, `motivo` NVARCHAR(40) NN, `ts_local_device` / `server_received_ts`.

Otras capas que viven solo del lado backend: **calendario_excepcion** (feriados por sucursal), **motivo_contingencia** (registro manual), **alerta_notificacion** (suscripciones a mail), **portal del cliente** (`cliente_empresa_acceso` , authority `ROLE_CLIENTE`), **rutas** (`ruta` , `ruta_parada` , `ruta_asignacion` , `ruta_ejecucion` , `ruta_ejecucion_parada`) y **planos** (`plano` , `plano_pin`). No se detallan acá; consultá el MODEL del backend.

Espejo móvil (Room)

La app Android **no** habla con MSSQL: mantiene un **espejo local en Room** poblado por sync, para funcionar **offline-first**. El espejo es un **subconjunto** del modelo backend.

> **Qué espeja y qué es solo-local**
>
> **Espejos por sync:** `sucursal_local` , `sector_local` , `tipo_sanitario_local` , `sanitario_local` (con `sector_id`), `operario_local` (con `rol` , **sin hashes**), `tag_local` (con `estado` / `alias` , `sanitario_id` nullable), `evento_limpieza_local` , `trabajo_sincronizado` (con `tipo` , `eventos_csv`), `ruta_ejecucion_local` + `ruta_parada_local` (snapshot de "mi ruta de hoy", reemplazable cada sync).
>
> **Solo-local (no viajan al server como tales):** `sync_state` (watermarks), `trabajo_pendiente` (con `tipo` y `manual` / `motivo_contingencia_codigo`) + `trabajo_pendiente_evento` , `nfc_evento_pendiente` (cola de gestión NFC del admin).

Diferencias clave a tener en cuenta:

- **No hay** `ubicacion_local` (igual que en backend: descompuesto en sucursal/sector).
- **El soft delete no se persiste local**: se traduce a `activo=false` .
- **Las credenciales NUNCA bajan**: `operario_local` no tiene `pin_hash` ni `password_hash` .
- Las migraciones de Room están **versionadas** y **nunca** usan `fallbackToDestructiveMigration` — la cola offline (`trabajo_pendiente`) debe sobrevivir a cada upgrade de schema.

Schema Room: v5 (validado contra el código)

El MODEL del **backend** menciona, en su resumen del espejo, Room **schema 2** sin `ruta_ejecucion_local` / `ruta_parada_local` — está **desactualizado**. La verdad está en el código: `WorkDoneDatabase.kt:50` declara `@Database(..., version = 5)` con migraciones 1→2→3→4→5; las tablas de ruta se crearon en la migración 3→4. Usá **v5** como referencia del espejo local.

Backend WorkDone Sanitarios

El backend es un servicio **Spring Boot** que concentra el dominio de **WorkDone Sanitarios**, el sistema de control de limpieza por NFC para sanitarios de aeropuertos y shoppings (producto de LubecaTech, cliente piloto Corpal en Córdoba, AR). Acá vive el corazón del negocio: el tap NFC, el endpoint de sincronización offline-first, la máquina de estados de los tags, la gestión de alertas y SLA, y el BackOffice React embebido.

Esta página es la referencia del backend **como componente**: stack, estructura, autenticación, base de datos y convenciones que un dev nuevo tiene que conocer antes de tocar nada.

> **Repos relacionados**
>
> Este repo es solo el backend Spring Boot. La app móvil Android y el kiosk Smiley viven en repos separados. Para entender el modelo de datos del dominio mirá el modelo de datos; para el vocabulario del proyecto, el glosario.

Stack

Tecnología	Versión	Para qué
Java	25 (LTS-track; mínimo Java 21)	Lenguaje base — aprovecha pattern matching for switch, sealed classes, records con compact constructors y virtual threads
Spring Boot	4.0.6	Framework de aplicación
JHipster	9.1.0	Scaffolder inicial (`jhipster jdl workdone.jdl`)
Node	24+ (25 también funciona)	Requerido para correr el JHipster CLI
Maven	3.9.16+	Build (no Gradle)

JWT	—	Auth con refresh tokens
Liquibase	—	Migraciones de schema
MapStruct	—	Mapeo de DTOs
Caffeine	—	Cache + Hibernate 2nd level
mssql-jdbc	12.6+ (classifier jre21+)	Driver de SQL Server (prod)
H2	—	DB de desarrollo, file-based en `./target/h2db/`
MSSQL Server	2019+	DB de producción, en EC2 Windows

Features de Java 25 que se aprovechan

- **Pattern matching for switch** (final, ya no preview) — usado en el `switch` sobre `EstadoTrabajo` en `TrabajoTapService`.
- **Sealed classes** — para los tipos cerrados de resultado del tap (`Ingreso`, `Egreso`, `CierreAutomatico`, `Rechazo`).
- **Records con compact constructors** — para los DTOs del API.
- **Virtual threads** — habilitados vía `spring.threads.virtual.enabled=true` para el endpoint `/sync`.

> ⚠ **Spring Boot 4 dropea APIs deprecados de Spring 5**
>
> Si copiás snippets de blogs viejos pueden no compilar. Otros cambios a tener presentes: Testcontainers usa `DynamicPropertySource`; Hibernate 6.6+ cambió el API de `Session.find()` con tipos genéricos.

Estructura del proyecto

El proyecto se genera con JHipster y luego se extiende con paquetes custom (marcados con ⚠ en el árbol original). El paquete base es `ar.com.lubeca.workdone`.

```
src/main/java/ar/com/lubeca/workdone/
├── WorkdoneApp.java
├── config/                      # Spring config (security, cache, jpa, jackson)
├── domain/                      # Entities JPA (generadas)
├── repository/                  # Repositories Spring Data (generadas)
├── service/                     # Services + impls (generadas)
│   ├── dto/                     # DTOs (generadas + sync/ nfc/ admin/ auth/
custom)
│   ├── mapper/                  # Mappers MapStruct (generadas)
│   ├── sync/                    # ⚠ CUSTOM — SyncService + TrabajoTapService
(corazón del negocio)
│   ├── nfc/                     # ⚠ CUSTOM — TagNfcLifecycleService (máquina de
estados del tag)
│   ├── admin/                   # ⚠ CUSTOM — Dashboard, Reportes,
TrabajoConsulta, Alertas
│   ├── auth/                    # ⚠ CUSTOM — AppAuthService (JWT móvil +
throttling)
│   ├── scheduler/               # ⚠ CUSTOM — SlaSchedulerService (3 tareas)
│   └── util/                    # ⚠ CUSTOM — UuidTagNormalizer (todo uuid pasa
por acá)
├── web/rest/                    # Controllers REST (generadas)
│   ├── app/                     # ⚠ CUSTOM — app móvil (Auth, Me, Sync, Trabajo,
Nfc, EventoLimpieza)
│   └── admin/                   # ⚠ CUSTOM — backoffice (Dashboard, Reporte,
Alerta, Operario, Trabajo, Nfc)
└── security/                    # JWT, filtros (modificada)

src/main/webapp/                  # ⚠ CUSTOM — BackOffice React (Vite+TS+shadcn),
build → target/classes/static/

src/main/resources/
├── config/
│   ├── application.yml
│   ├── application-dev.yml
│   ├── application-prod.yml
│   └── liquibase/changelog/     # migraciones NNN_descripcion.xml
└── i18n/                        # JHipster genera, no se usa
```

Capas

- **`domain/`** + **`repository/`** — entidades JPA y repositorios Spring Data, en su mayoría generados por JHipster.
- **`service/`** — lógica de negocio. Los subpaquetes custom (`sync`, `nfc`, `admin`, `auth`, `scheduler`, `util`) son donde vive el dominio real. **`service/sync/`** **es el corazón**: `SyncService` y `TrabajoTapService`.
- **`web/rest/`** — controllers REST, separados en `app/` (móvil) y `admin/` (backoffice), más los controllers del módulo Smiley en `web/rest/smiley/`.

- `security/` — configuración JWT y filtros, incluido el filtro de api_key del device.
- `src/main/webapp/` — el BackOffice React, que en build prod se embebe en el JAR (`target/classes/static/`).

> **Servicios que un dev nuevo debe ubicar primero**
>
> `TrabajoTapService` (toggle ingreso/egreso del tap), `SyncService` (orquesta el endpoint `/sync`), `TagNfcLifecycleService` (máquina de estados del tag + auditoría) y `UuidTagNormalizer` (toda normalización de UUID de tag pasa por acá, sin excepción).

Autenticación y autorización

El sistema tiene **dos mundos de autenticación distintos**: personas (JWT) y dispositivos (api_key). No se mezclan.

Personas — JWT con refresh tokens

Actor	Endpoint de login	Tabla	Authority
BackOffice web (admin)	`POST /api/authenticate` (JHipster estándar)	`jhi_user`	`ROLE_ADMIN`
Operario móvil	`POST /api/v1/app/auth/login-password` · `login-pin` · `refresh` · `logout`	`operario`	`ROLE_OPERARIO`
Cliente (portal de transparencia)	flujo de activación por mail	`jhi_user`	`ROLE_CLIENTE`

Detalles del login móvil:

- **JWT de 15 minutos**, con `refresh` por rotación de token.
- **Throttling**: 5 fallos por usuario+device devuelven `429` por 10 minutos.
- Header `X-Device-UUID` **obligatorio** en login y refresh (si falta → `400`).
- `logout` es **público**: valida por el hash del refresh token, así que el access token puede haber expirado.

- Las responses de login y `GET /api/v1/app/me` incluyen el `rol` del operario: `ADMIN | SUPERVISOR | OPERADOR_LIMPIEZA`.

> ⚠ **El operario móvil NO es lo mismo que el usuario web**
>
> El operario móvil usa la authority `ROLE_OPERARIO` (**no** `ROLE_USER`, que es el rol web) y accede **exclusivamente** a `/api/v1/app/*`. El admin web usa `ROLE_ADMIN` y accede al CRUD scaffold (`/api/*`) y a `/api/v1/admin/*`. Un operario que pegue a `/api/<entidad>` recibe **403**, y un cliente que pegue a `/api/v1/app/*` también recibe **403**. La separación de authorities está hecha a propósito (ADR-040) y está validada en el código: `SecurityConfiguration.java:106` (`/api/v1/app/**` → `hasAuthority(OPERARIO)`) y `AuthoritiesConstants.java:18` (`OPERARIO = "ROLE_OPERARIO"`).

Dispositivos — api_key (kiosk Smiley)

La terminal Smiley **no tiene operario**, así que no usa JWT. Se autentica por la api_key del dispositivo:

- Headers **`X-Device-Uuid` + `X-Api-Key`** (filtro `SmileyApiKeyAuthFilter`).
- El backend valida que el dispositivo exista, esté activo, no esté borrado y sea `tipo=TERMINAL_SMILEY`, y verifica la api_key con **BCrypt**.
- Headers faltantes o api_key inválida → `401`.
- La authority resultante es **`ROLE_TERMINAL`**.
- El `sanitario` lo determina el server por el vínculo del dispositivo — la terminal **no puede** opinar por otro sanitario.

La api_key se entrega en **texto plano una sola vez**, al vincular la terminal (`POST /api/v1/smiley/vincular`); el backend solo persiste su hash BCrypt.

Matriz de authorities

Prefijo de ruta	Authority requerida	Quién
`/api/v1/app/*`	`ROLE_OPERARIO`	operario móvil
`/api/v1/admin/*`	`ROLE_ADMIN`	BackOffice web

`/api/v1/cliente/*`	`ROLE_CLIENTE`	portal de transparencia (solo lectura)
`/api/v1/smiley/*` (salvo `vincular` / `ping`)	`ROLE_TERMINAL`	terminal Smiley (api_key)
`/api/v1/publico/*`	sin auth (anónimo, rate-limited por IP)	QR público
`/api/*` (CRUD scaffold)	`ROLE_ADMIN`	BackOffice web

> **Revocación de dispositivos**
>
> Un dispositivo con `activo=false` que llama a `POST /api/v1/app/sync` recibe **`401` con cuerpo `DEVICE_REVOCADO`** (validado en cada sync, sin cache). El cliente móvil ante ese 401 debe detener el sync, borrar credenciales del device y bloquear la app.

Base de datos por ambiente

Ambiente	Motor	Detalle
Desarrollo	H2	File-based en `./target/h2db/` (incluye DevTools)
Producción	MSSQL Server 2019+	En EC2 Windows, compartida con otros productos de LubecaTech

También se puede levantar dev contra un MSSQL local nativo en Windows (sin Docker) con el perfil `dev,mssql`. La suite completa con Testcontainers MSSQL sí requiere Docker.

Migraciones — Liquibase

El schema y los seeds se manejan con **Liquibase**, una migration por feature.

Reglas (en convenciones):

- Numerar `NNN_descripcion.xml` con N ≥ 002 (la `001` la genera JHipster).

- **Nunca editar una migration anterior.**
- Para SQL específico de MSSQL (ROWVERSION, índices filtrados), usar `<sql dbms="mssql">`.
- Si excepcionalmente se edita una migration ya aplicada (cambia su checksum), hay que recrear el H2 local antes de levantar (`mvn clean` o borrar `target/h2db`). CI y prod no se ven afectados.

> **Seeds por contexto**
>
> Los seeds de catálogo usan `context="dev,prod"`; los seeds de datos de prueba usan `context="dev"`. Los seeds de dev incluyen los operarios de prueba (ver más abajo).

Credenciales de desarrollo

Hay dos mundos de auth con credenciales distintas en dev:

- **BackOffice web** (`/api/authenticate`, tabla `jhi_user`): `admin` / `admin` **(solo dev)**. En prod, la migration 031 (`context=prod`) borra el `user` scaffold y resetea el password de `admin` al bcrypt de la env var `PROD_ADMIN_PASSWORD_HASH` — el deploy DEBE exportarla, si no `admin` no autentica (fail-closed).
- **App móvil** (`/api/v1/app/auth/*`, tabla `operario`), seedeados en dev:

usuario	PIN	password	rol
jperez	1234	`Pa$$w0rd!`	ADMIN
msuarez	5678	`Pa$$w0rd!`	OPERADOR_LIMPIEZA
rgomez	1234	`Pa$$w0rd!`	SUPERVISOR

> **`admin` no sirve para la app móvil**
>
> `admin` no es un operario. En prod, el supervisor/admin operario se crean por ABM o por `UPDATE operario SET rol=...`.

Convenciones clave

Estas son las convenciones que un dev nuevo **tiene que** conocer antes de escribir código. Vienen del `CLAUDE.md` del backend.

Nombres

- **Entidades**: PascalCase singular (`Sanitario`, `TagNfc`, `Trabajo`).
- **Tablas**: snake_case singular (`sanitario`, `tag_nfc`, `trabajo`).
- **DTOs**: `XxxDTO` para entrada, `XxxResponseDTO` para salida custom.
- **Services**: interface `XxxService` + impl `XxxServiceImpl`.
- **Endpoints custom**: `/api/v1/app/*` (móvil), `/api/v1/admin/*` (backoffice), `/api/v1/cliente/*` (portal), `/api/v1/publico/*` (QR público). Los endpoints generados por JHipster quedan en `/api/*` (sin `/v1`) por compatibilidad.

Reglas de oro del negocio

1. **Antes de tocar `/sync` o `/tap`, leer `docs/SYNC_PROTOCOL.md` entero.** No es opcional. (Ver también Sincronización.)
2. **Idempotencia siempre.** Cualquier endpoint que mute datos desde la app móvil debe ser idempotente vía `client_uuid` o equivalente.
3. **El timestamp del teléfono es la verdad** para `inicio_ts` / `fin_ts`. El server solo registra `server_received_ts`.
4. **Nunca borrar físicamente catálogos.** Soft delete con `deleted_at`.
5. **Hashes con BCrypt cost 12** para password y PIN. Usar `BCryptPasswordEncoder` de Spring Security.
6. **No exponer hashes en responses.** `@JsonIgnore` en los DTOs de salida.
7. **Concurrencia de operarios permitida.** Dos operarios pueden tener trabajos abiertos sobre el mismo sanitario.
8. **El backend confía en el orden cronológico** de `trabajos_pendientes`. El cliente garantiza orden por `ts_local_device ASC`.
9. **Docs antes del commit.** Siempre actualizar los `.md` que correspondan al cambio (API, modelo, decisiones, sync, estado) y commitear código y docs juntos.

Soft delete

Las entidades de catálogo tienen `deleted_at` con **manejo manual** (campo `deletedAt` en la entity; los services/queries filtran donde corresponde). **No se usan** `@SQLDelete` / `@Where`: el sync necesita ver los eliminados para propagar las bajas, y esas anotaciones lo harían imposible.

> ⚠ **El soft-delete es frágil ante regeneración con JHipster**
>
> El `delete(id)` de los 7 service impls de catálogos sincronizados (Sanitario, TagNfc, Operario, Sucursal, Sector, TipoSanitario, EventoLimpieza) hace soft-delete (`deletedAt` + `updatedAt` + `activo=false`, nunca `deleteById`). Sus tests generados usan `assertSameRepositoryCount`, no `assertDecrementedRepositoryCount`. Si regenerás esas entidades con JHipster, hay que re-aplicar ambos cambios.

Auditoría

Todas las entidades de catálogo extienden `AbstractAuditingEntity` de JHipster, que auto-popula `created_by` / `last_modified_by` desde el SecurityContext.

Tests

- Unit tests con **JUnit 5 + Mockito**.
- Integration tests con `@SpringBootTest` + **Testcontainers** para MSSQL. Los `*IT` corren con failsafe, **no** con `mvn test`.
- Mínimo 1 test por endpoint custom (happy path + 2 edge cases).
- Los tests de `TrabajoTapService` son **obligatorios**: deben cubrir los casos especiales del protocolo de sync.
- **Property-based tests** con jqwik para invariantes de funciones puras críticas (ej. `UuidTagNormalizerPropertyTest`).

Comandos comunes

```
# Levantar en dev (H2 + DevTools)
./mvnw

# Tests unitarios (surefire — excluye los *IT)
./mvnw test

# Tests de integración (failsafe — los *IT corren acá)
./mvnw test-compile failsafe:integration-test failsafe:verify -Dit.test='*IT' -
DfailIfNoTests=false

# Suite completa con Testcontainers MSSQL (requiere Docker)
./mvnw verify

# Build prod (JAR con el BackOffice embebido)
./mvnw -Pprod clean package -DskipTests
```

Swagger UI en dev: `http://localhost:8080/swagger-ui/index.html` · OpenAPI JSON: `http://localhost:8080/v3/api-docs`.

API HTTP

El backend de WorkDone Sanitarios expone una API REST versionada. Esta página cubre las convenciones generales, los endpoints principales agrupados por consumidor, y en profundidad el endpoint `/sync` —el transaccional que mueve toda la operación offline-first.

> **Fuente de verdad**
>
> Los contratos de `/api/v1/app/sync` y `/api/v1/smiley/sync` son contratos compartidos: la fuente de verdad vive en `docs/SYNC_PROTOCOL.md` y `docs/SYNC_PROTOCOL_SMILEY.md` del backend, y las copias en los repos consumidores se generan por script. Para el detalle conceptual del flujo, ver Sincronización. Para las entidades, ver el modelo de datos.

Convenciones generales

- **Versión 2** del contrato. Los endpoints custom van bajo `/api/v1/app/*`, `/api/v1/admin/*`, `/api/v1/cliente/*`, `/api/v1/publico/*` y `/api/v1/smiley/*`. El CRUD scaffold de JHipster queda en `/api/*` (sin `/v1`) por compatibilidad.
- **Errores RFC 7807** (problem+json).
- **Timestamps ISO 8601 con offset** (ej. `2026-06-10T10:15:32.123-03:00`).
- **Autenticación** según consumidor (ver Autenticación y autorización):
  - Operario móvil: JWT (`Authorization: Bearer …`), header `X-Device-UUID` obligatorio en login/refresh.
  - Terminal Smiley: headers `X-Device-Uuid` + `X-Api-Key`.
  - BackOffice / portal cliente: JWT.
  - QR público: sin auth, rate-limited por IP.

Capitalización del header de device (validado contra código)

El operario móvil y su filtro usan `X-Device-UUID`; la terminal Smiley usa `X-Device-Uuid`. Es una inconsistencia **real** en el código (`AppAuthController.java` / `AuthInterceptor.kt` vs `SmileyApiKeyAuthFilter.java` / `SmileyAuthInterceptor.kt`), pero **inocua**: los headers HTTP son case-insensitive por RFC 7230 y cada subsistema es internamente consistente.

Endpoints principales

La tabla resume los endpoints documentados, agrupados por consumidor. No es exhaustiva del CRUD scaffold.

Autenticación

Método	Ruta	Para qué	Auth
POST	`/api/authenticate`	Login BackOffice web (JHipster)	—
POST	`/api/v1/app/auth/login-password`	Login operario por password	— (header `X-Device-UUID`)
POST	`/api/v1/app/auth/login-pin`	Login operario por PIN	— (header `X-Device-UUID`)
POST	`/api/v1/app/auth/refresh`	Rotación de refresh token	— (header `X-Device-UUID`)
POST	`/api/v1/app/auth/logout`	Logout (valida por hash del refresh)	público

App móvil — operación

Método	Ruta	Para qué	Auth

POST	`/api/v1/app/sync` ★	Sync transaccional (pull deltas + push trabajos/NFC)	`ROLE_OPERARIO`
GET	`/api/v1/app/me`	Perfil del operario (incluye `rol`)	`ROLE_OPERARIO`
GET	`/api/v1/app/trabajos/actual`	Trabajo abierto actual (expone `tipo`)	`ROLE_OPERARIO`
GET	`/api/v1/app/trabajos/historial?limit=50`	Historial filtrado por operario + rol (incluye `eventos[]`)	`ROLE_OPERARIO`
GET	`/api/v1/app/eventos-limpieza`	Catálogo de eventos activo, filtrado por rol	`ROLE_OPERARIO`

App móvil — gestión NFC

Método	Ruta	Para qué	Auth
GET	`/api/v1/app/nfc/resolver/{uuid}`	Resuelve estado del tag (`NO_REGISTRADO|EN_STOCK|ASIGNADO|BAJA`)	cualquier operario
POST	`/api/v1/app/nfc/alta-stock`	Alta de tag en stock (`{uuid_tag, alias}`), idempotente	solo ADMIN
POST	`/api/v1/app/nfc/asignar`	Asigna tag a sanitario (`{uuid_tag, sanitario_id}`)	solo ADMIN
POST	`/api/v1/app/nfc/desasignar`	Desasigna tag (`{uuid_tag}`)	solo ADMIN
POST	`/api/v1/app/nfc/reasignar`	Reasigna tag, transaccional (`{uuid_tag, sanitario_nuevo_id, notas?}`)	solo ADMIN

GET	`/api/v1/app/sanitarios/disponibles`	Árbol Sucursal›Sector›Sanitario con `tiene_tag`	solo ADMIN

Toda mutación NFC pasa por `TagNfcLifecycleService`

El servicio audita cada cambio (`TagNfcEvento`) y normaliza el UUID. Rol insuficiente → `403` .

Admin (BackOffice) — selección

Método	Ruta	Para qué	Auth
GET	`/api/v1/admin/trabajos?desde&hasta&tipo&rol&operario_id&…`	Consulta de trabajos con filtros asociativos (AND)	`ROLE_ADMIN`
GET	`/api/v1/admin/dashboard/estado?sucursal_id&sector_id`	Grilla de estado en tiempo real + alertas activas	`ROLE_ADMIN`
GET	`/api/v1/admin/alertas/activas`	Alertas activas	`ROLE_ADMIN`
POST	`/api/v1/admin/alertas/{id}/atender`	Atender una alerta	`ROLE_ADMIN`
GET	`/api/v1/admin/reportes/eventos?desde&hasta&codigo&…`	Ranking evento × sanitario/sector (con export)	`ROLE_ADMIN`
POST	`/api/v1/admin/dispositivo/{id}/deshabilitar` · `/habilitar`	Revocación / rehabilitación de dispositivos	`ROLE_ADMIN`

Terminal Smiley (kiosk)

Método	Ruta	Para qué	Auth
GET	`/api/v1/smiley/ping`	"Probar conexión" del setup (`{version, server_time}`)	público
POST	`/api/v1/smiley/vincular`	Canjea código de un solo uso → credenciales del device	público (rate-limited)
POST	`/api/v1/smiley/sync`	Sync de opiniones (pull motivos + push opiniones)	`ROLE_TERMINAL`
GET	`/api/v1/smiley/estado`	Refresca el idle (`{ultima_limpieza_hace_min}`)	`ROLE_TERMINAL`

Público (QR)

Método	Ruta	Para qué	Auth
GET	`/api/v1/publico/sanitario/{slug}`	Estado público del sanitario	sin auth (rate-limited por IP)
POST	`/api/v1/publico/opinion`	Opinión anónima (`{slug, valor, motivo_codigo?, client_uuid}`)	sin auth (rate-limited por IP)
GET	`/api/v1/publico/motivos`	Catálogo de motivos activos	sin auth (rate-limited por IP)

CRUD scaffold (JHipster, sin `/v1`)

Todo el CRUD scaffold `/api/*` exige `ROLE_ADMIN`. Endpoints verificados: `/api/sanitarios`, `/api/tag-nfcs`, `/api/operarios`, `/api/sla-limpiezas`, `/api/dispositivos`, `/api/trabajos`, `/api/alertas`, `/api/empresas`, `/api/sucursals`, `/api/sectors`, `/api/tipo-sanitarios`, `/api/evento-limpiezas`.

> ⚠ **Pluralización a la inglesa**
>
> Se acepta la pluralización que genera JHipster (`/api/sucursals` , `/api/sectors`). El dominio en español vive en las entidades, no en los paths. `/api/ubicacions` dejó de existir en v2.

El endpoint `/sync` en profundidad

`POST /api/v1/app/sync` es el **endpoint único transaccional** del sync de operarios: en una sola llamada el cliente baja deltas de catálogos (pull) y sube su cola de trabajos y eventos NFC (push). Es **retrocompatible con clientes v1**.

Request

```
{
  "device_uuid": "...",
  "operario_id": 5,
  "last_sync": {              // null por tabla = pull completo de esa tabla
    "sanitario": "...", "tag_nfc": "...", "operario": "...",
    "sucursal": "...", "sector": "...", "tipo_sanitario": "...",
"evento_limpieza": "..."
  },
  "trabajos_pendientes": [{
    "client_uuid": "...", "uuid_tag_leido": "...", "operario_id": 5,
    "ts_local_device": "2026-06-10T10:42:18.456-03:00",
    "tipo_tentativo": "EGRESO",
    "tipo": "LIMPIEZA",                       // v2 · LIMPIEZA | SUPERVISION ·
default LIMPIEZA
    "eventos": [{ "evento_id": 3, "codigo": "REPONER_INSUMOS", "observacion":
"..." }]   // v2 · solo EGRESO
  }],
  "nfc_eventos_pendientes": [{                 // v2 · cola de gestión NFC (rol
ADMIN)
    "client_uuid": "...", "accion":
"ALTA_STOCK|ASIGNACION|DESASIGNACION|REASIGNACION",
    "uuid_tag": "...", "alias": "...", "sanitario_id": 12, "sanitario_nuevo_id":
null,
    "ts_local_device": "..."
  }],
  "device_now": "2026-06-10T10:42:20.000-03:00",  // ANEXO DT/DT-B4 · hora del
device al enviar · OPCIONAL
  "app_version": "0.1.0",                          // G1 · telemetría · OPCIONAL
  "os_version": "Android 15",
  "bateria_pct": 87,
  "red_tipo": "WIFI"                               // WIFI | CELULAR | ETHERNET |
OTRA | SIN_RED
}
```

Compatibilidad v1

`tipo`, `eventos`, `nfc_eventos_pendientes`, `device_now` y los 4 campos de telemetría son **opcionales con defaults seguros** (`LIMPIEZA`, `[]`, `[]`, sin medición de drift). Un cliente viejo que no los manda no rompe nada.

Response

```
{
  "server_ts": "...",                          // nuevo watermark
  "deltas": { /* updated[] + deleted_ids[] por cada uno de los 7 catálogos */ },
  "trabajos_procesados": [{
    "client_uuid": "...", "estado": "OK", "accion_real": "EGRESO",
    "trabajo_id": 1235, "sanitario_id": 5,
    "inicio_ts": "2026-06-10T10:15:32.000-03:00",   // verdad del server (clave
en cierre automático)
    "fin_ts": "2026-06-10T10:42:18.000-03:00",      // null si quedó abierto
(INGRESO) o no se persistió trabajo
    "duracion_seg": 1606,
    "tipo": "LIMPIEZA", "eventos": [{ "codigo": "...", "nombre": "...",
"observacion": "..." }],
    "warnings": []
  }],
  "nfc_eventos_procesados": [{ "client_uuid": "...", "estado": "OK" }],
  "mi_ruta_hoy": { /* ... */ },                 // V21_RUTAS · ruta asignada al
operario hoy · OPCIONAL (NON_NULL)
  "config": { "silencio_cliente_seg": 8, "confirmar_egreso_min": 2 }   // ANEXO
DT/DT-B4 · config remota anti doble-tap
}
```

Notas de los `deltas` (7 catálogos): `tag_nfc` expone `estado` + `alias` (la app resuelve taps **solo** contra `estado='ASIGNADO'`); `operario` expone `rol`; `sanitario` expone `sector_id` + `tipo_sanitario_id`; `evento_limpieza.roles_permitidos` viaja como **array JSON** (`["OPERADOR_LIMPIEZA","SUPERVISOR"]`), no CSV.

Estados por trabajo

El campo `estado` de cada ítem de `trabajos_procesados` puede ser:

Estado	¿Persiste trabajo?	Significado
`OK`	sí	Procesado correctamente
`TAG_DESCONOCIDO`	sí (sanitario nullable)	Tag no resuelto offline; trabajo con `modo=OFFLINE_PENDIENTE_RESOLUCION` + alerta
`OPERARIO_INACTIVO`	sí	Se acepta con warning + alerta

`DUPLICADO_IGNORADO`	sí (devuelve el existente)	`client_uuid` ya procesado
`ROL_NO_PERMITIDO`	no	Tap de un operario rol ADMIN; el cliente purga
`RECHAZADO_REBOTE`	no (queda en `tap_descartado`)	Doble-tap accidental al salir; idempotente
`MANUAL_INVALIDO`	no	Registro manual sin sanitario/motivo válido

Drift de reloj

Si el request trae `device_now`, el server estampa `dispositivo.drift_seg = |device_now - server_now|`. Si excede `app.tap.drift-max-seg` (default 300 s), genera **una** alerta `RELOJ_DESVIADO` por device (dedup por `dispositivo_id`), auto-atendida cuando un sync posterior reporta drift normal.

> ⚠ **El drift nunca se mide con los `ts_local_device` de los ítems**
>
> La cola offline trae timestamps viejos legítimos. El drift solo se calcula con `device_now` (la hora del device **al momento de enviar**).

Códigos de error y conflictos

Errores RFC 7807. Códigos custom v2:

Code	HTTP	Cuándo
`CONFLICTO_ASIGNACION`	409	El sanitario ya tiene tag activo, el tag no está EN_STOCK, o conflicto concurrente
`TRANSICION_INVALIDA`	409	Transición inválida en la máquina de estados del tag (ej. BAJA sobre ASIGNADO)

`ROL_INSUFICIENTE`	403	Mutación NFC móvil con operario no-ADMIN
`EVENTO_NO_PERMITIDO`	403	Evento fuera de los `roles_permitidos` del rol del operario
`EVENTO_EXCLUSIVO_COMBINADO`	400	Evento exclusivo combinado con otros en el mismo cierre
`EVENTOS_REQUERIDOS`	400	EGRESO normal sin eventos
`DEVICE_REVOCADO`	401	Dispositivo con `activo=false` llamando a `/sync`
(sin code)	400	Header `X-Device-UUID` ausente/vacío en login/refresh

La resolución de conflictos en el push (asignación NFC, máquina de estados del tag) se explica en detalle en Sincronización.

Ciclo de vida de los tags NFC

Si recién entrás al proyecto, esta es la página que explica **cómo nace, se asigna, se reasigna y se da de baja una placa NFC** en **WorkDone Sanitarios**, y qué transiciones de estado son válidas (y cuáles no). Una placa (`tag_nfc`) es el chip físico que el personal acerca con el teléfono para registrar un trabajo; antes de poder usarse, tiene que recorrer un ciclo de vida controlado.

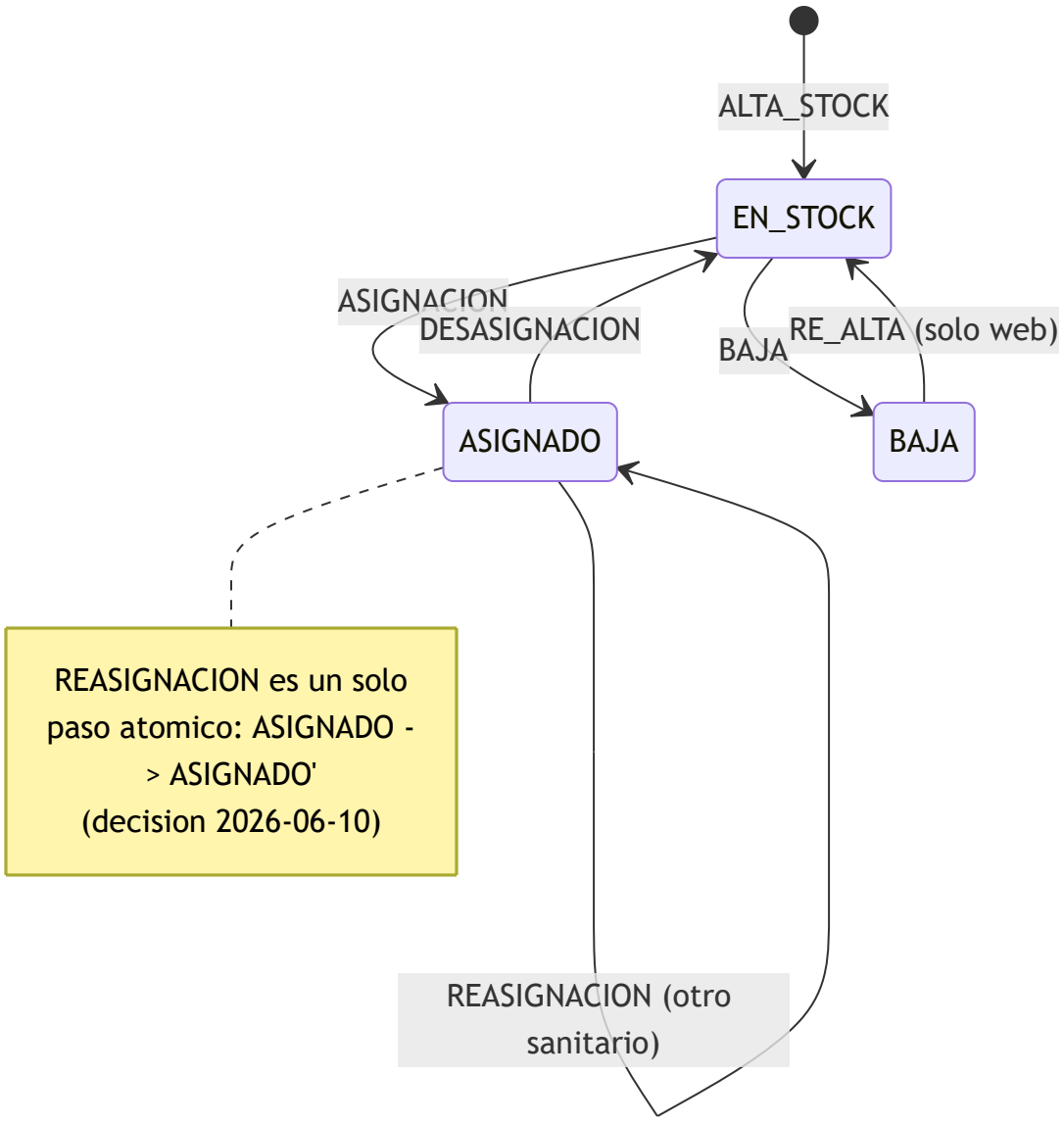
La **fuente de verdad** de este documento es el backend (Spring Boot / JHipster), concretamente `TagNfcLifecycleService`: el servicio que concentra **TODAS** las transiciones de estado del tag. Ningún otro componente cambia el estado de un tag por su cuenta.

> **Punto de partida**
>
> Cada operación recibe un `uuid` crudo (leído del chip o recibido por API), un contexto de auditoría (`AccionNfcContexto`: operario, dispositivo, notas — **todos nullable**) y devuelve el `TagNfc` resultante. Las transiciones inválidas lanzan `TransicionInvalidaException`; los conflictos de asignación, `ConflictoAsignacionException`. Cada transición exitosa persiste un registro inmutable en `tag_nfc_evento`.

Para el contexto del modelo de tablas, ver Modelo de datos. Para cómo estas operaciones viajan offline desde la app, ver Sincronización.

Máquina de estados

Un tag tiene tres estados (`EstadoTag`): `EN_STOCK`, `ASIGNADO` y `BAJA`. El alta lo crea directamente en `EN_STOCK`. Las transiciones entre estados están todas mediadas por una acción concreta del `TagNfcLifecycleService`.



Lectura del diagrama

- `ALTA_STOCK` es idempotente: si el tag ya existe, devuelve el existente **sin** generar evento.
- `REASIGNACION` es un **bucle** sobre `ASIGNADO`: el tag no pasa por `EN_STOCK`; cambia de sanitario en una sola operación atómica.
- `BAJA` sale **solo** desde `EN_STOCK`. Un tag `ASIGNADO` debe desasignarse antes de darse de baja.
- `RE_ALTA` (`BAJA → EN_STOCK`) está disponible **solo desde web (BackOffice)**, nunca por `/sync`.

Acciones

Cada acción valida el estado de origen esperado, aplica el cambio y persiste un `TagNfcEvento` con la `AccionTagNfc` correspondiente. La siguiente tabla resume qué hace cada una.

Acción	De → A	Quién puede ejecutarla	Evento que genera
`ALTA_STOCK`	`[*]` → `EN_STOCK`	Web (BackOffice) y mobile vía `/sync`	`ALTA_STOCK` — **salvo** que el tag ya exista (idempotente, no genera evento)
`ASIGNACION`	`EN_STOCK` → `ASIGNADO`	Web y mobile vía `/sync`	`ASIGNACION` (con `sanitarioNuevo`)
`DESASIGNACION`	`ASIGNADO` → `EN_STOCK`	Web y mobile vía `/sync`	`DESASIGNACION` (con `sanitarioAnterior`)
`REASIGNACION`	`ASIGNADO` → `ASIGNADO'`	Web y mobile vía `/sync`	`REASIGNACION` (un solo evento con `sanitarioAnterior` + `sanitarioNuevo`)
`BAJA`	`EN_STOCK` → `BAJA`	Web y mobile vía `/sync`	`BAJA`
`RE_ALTA`	`BAJA` → `EN_STOCK`	**Solo web (BackOffice)**	`RE_ALTA`

> ⚠ **Mobile no envía todas las acciones**
>
> El procesador de la cola NFC del sync (`SyncService.procesarUnNfcEvento`) sólo despacha `ALTA_STOCK`, `ASIGNACION`, `DESASIGNACION` y `REASIGNACION`. **`RE_ALTA` y `BAJA` no están en ese `switch`**: `RE_ALTA` es exclusiva de web por diseño. Cualquier otra acción recibida por sync cae en el `default` y se marca como `ERROR`.

`ALTA_STOCK` — alta idempotente

Crea el tag en `EN_STOCK` con `activo = false`. **Es idempotente por uuid**: si el uuid normalizado ya existe en `tag_nfc`, devuelve el tag existente **sin crear evento ni modificar nada**. Esto es clave para la idempotencia best-effort del sync (ver más abajo).

`ASIGNACION` — vincular a un sanitario

Requiere que el tag esté `EN_STOCK` y que el sanitario destino **no tenga ya un tag `ASIGNADO`**. Pasa el tag a `ASIGNADO`, le setea el sanitario, `asignadoEn` y `activo = true`.

> **Conflicto: el primero en sincronizar gana**
>
> Si el tag no está `EN_STOCK`, o si el sanitario ya tiene un tag `ASIGNADO`, se lanza `ConflictoAsignacionException`. En el sync esto se traduce a `CONFLICTO_ASIGNACION`. Cuando dos dispositivos asignan offline al mismo sanitario, **el primero en el orden de sincronización gana** y el segundo recibe el conflicto. El orden lo define `ts_local_device ASC`.

`DESASIGNACION` — devolver a stock

Sólo válida desde `ASIGNADO`. Limpia el sanitario, `asignadoEn`, pone `activo = false` y devuelve el tag a `EN_STOCK`. Guarda el `sanitarioAnterior` en el evento.

`REASIGNACION` — mover entre sanitarios (atómica)

Mueve un tag de un sanitario a otro **en un solo paso atómico**, sin pasar por `EN_STOCK`. El tag permanece `ASIGNADO`; sólo cambian `sanitario` y `asignadoEn`.

> **Decisión 2026-06-10: un solo evento**
>
> `REASIGNACION` **no** es "desasignar + asignar". Genera **un único** `TagNfcEvento` con `sanitarioAnterior` **y** `sanitarioNuevo` poblados. Esto preserva la trazabilidad del movimiento como una sola operación en la auditoría.
>
> Caso borde: si el sanitario nuevo es **el mismo** que el anterior, **no se valida el conflicto** (no tendría sentido chequear que el sanitario "ya tiene un tag" cuando ese tag es justamente el que estamos moviendo). En cualquier otro caso, el destino no puede tener un tag `ASIGNADO`, o se lanza `ConflictoAsignacionException`.

`BAJA` — desactivar definitivamente

Sólo válida desde `EN_STOCK`. Pone el tag en `BAJA`, setea `dadoBajaEn` y `activo = false`. Un tag `ASIGNADO` **no** puede darse de baja directamente: hay que desasignarlo primero.

`RE_ALTA` — reactivar desde baja (solo web)

Devuelve un tag desde `BAJA` a `EN_STOCK` (limpia `dadoBajaEn`, `activo = false`). **Disponible únicamente desde web (BackOffice)**; no se procesa por `/sync`.

Transiciones inválidas

Cada método valida el estado de origen esperado antes de aplicar el cambio. Si el tag está en un estado distinto del esperado, se lanza `TransicionInvalidaException`, que en el sync se reporta como `TRANSICION_INVALIDA`. Además, si el uuid no existe en `tag_nfc`, `resolverTag` lanza la misma excepción (`tag no registrado`).

Estas son las combinaciones que **rechazan** la operación:

Acción intentada	Estado del tag	Resultado
`ASIGNACION`	`ASIGNADO` o `BAJA`	`ConflictoAsignacionException` (debe estar `EN_STOCK`)
`DESASIGNACION`	`EN_STOCK` o `BAJA`	`TransicionInvalidaException` (solo desde `ASIGNADO`)
`REASIGNACION`	`EN_STOCK` o `BAJA`	`TransicionInvalidaException` (solo desde `ASIGNADO`)
`BAJA`	`ASIGNADO`	`TransicionInvalidaException` (desasignar primero)
`BAJA`	`BAJA`	`TransicionInvalidaException` (ya está en `BAJA`)
`RE_ALTA`	`EN_STOCK` o `ASIGNADO`	`TransicionInvalidaException` (solo desde `BAJA`)

Cualquiera	uuid inexistente	`TransicionInvalidaException` (`tag no registrado`)

> **Conflicto vs. transición inválida**
>
> `ASIGNACION` sobre un tag que no está `EN_STOCK` lanza `ConflictoAsignacionException` (no `TransicionInvalidaException`), porque el caso típico es una carrera de asignación, no un estado imposible. La distinción importa: en el sync producen estados distintos (`CONFLICTO_ASIGNACION` vs `TRANSICION_INVALIDA`).

Normalización de UUID

Todo punto de entrada de uuid pasa por `UuidTagNormalizer.normalize()` antes de cualquier búsqueda o persistencia. La forma canónica es: null-safe, sin espacios (internos y de borde) y en **mayúsculas** (`raw.replaceAll("\\s+", "").toUpperCase()`).

Cada método del `TagNfcLifecycleService` normaliza el uuid en su **primera línea**. `TrabajoTapService` también normaliza para resolver el tag al procesar un tap (el `uuid_tag_leido` del Trabajo se persiste **crudo**, como snapshot del tap).

> **El backend es defensivo a propósito**
>
> Aunque la app móvil ya normalice el uuid antes de enviarlo, el backend **vuelve a normalizar** en cada entrada. No se confía en que el cliente haya hecho el trabajo: dos lecturas del mismo chip con distinto formato (con/sin guiones, mayúsculas/minúsculas) deben resolver al **mismo** registro `tag_nfc`. Esto sostiene la whitelist por uuid normalizado del ADR de sync.

Auditoría

Cada transición exitosa persiste un `TagNfcEvento` (tabla **inmutable** `tag_nfc_evento`). El evento captura: el tag, el `uuidTag` (ya normalizado), la `AccionTagNfc`, `sanitarioAnterior`, `sanitarioNuevo`, `operario`, `dispositivo`, `fecha` y `notas`.

> ⚠ **Operario y dispositivo son nullable**
>
> Los campos `operario` y `dispositivo` del contexto de auditoría son **nullable**. `AccionNfcContexto.vacio()` los deja en `null` para operaciones **web** o de **sistema** que no tienen un actor humano / dispositivo asociado (por ejemplo, backfill o tareas programáticas). No asumas que todo evento NFC tiene operario.

Campo	Poblado en
`sanitarioAnterior`	`DESASIGNACION`, `REASIGNACION`
`sanitarioNuevo`	`ASIGNACION`, `REASIGNACION`
`operario` / `dispositivo`	Operaciones con actor humano (nullable en web/sistema)

Estados de procesamiento NFC en el sync

Cuando los eventos NFC viajan en la cola del sync, cada ítem se procesa de forma aislada (un ítem que falla **no aborta** el sync completo) y se devuelve con un `EstadoNfcEventoProcesado`. La autorización es estricta: **solo el operario con rol `ADMIN`** puede enviar ítems NFC; si no lo es y la cola no está vacía, todos los ítems se rechazan con `ROL_INSUFICIENTE` sin abortar.

Estado	Significado
`OK`	Transición ejecutada con éxito.
`CONFLICTO_ASIGNACION`	El sanitario ya tiene un tag `ASIGNADO`, o el tag no está `EN_STOCK` (`ConflictoAsignacionException`).
`DUPLICADO_IGNORADO`	`client_uuid` ya visto (best-effort, según el estado del tag).
`TRANSICION_INVALIDA`	La máquina de estados rechaza la operación (`TransicionInvalidaException`).

`ROL_INSUFICIENTE`	El operario del sync no es `ADMIN` — ítem ignorado sin abortar.
`ERROR`	Excepción inesperada o acción desconocida — no aborta el sync completo.

Más sobre el protocolo de sincronización, orden de procesamiento y casos offline en Sincronización.

Limitación real: idempotencia best-effort (riesgo a evaluar)

La idempotencia de los eventos NFC en el sync es **best-effort**, no estricta. **No existe** una tabla `nfc_evento_procesado` que persista los `client_uuid` ya vistos. La idempotencia depende enteramente del comportamiento idempotente de cada acción — principalmente que `ALTA_STOCK` devuelve el tag existente sin tocar nada.

Consecuencia: el sistema **no puede distinguir** un reintento genuino de una operación nueva con el mismo uuid. Para idempotencia estricta haría falta persistir el `client_uuid` en una tabla auxiliar. El código lo marca explícitamente:

```
// TODO v2.1: tabla nfc_evento_procesado para idempotencia estricta por
client_uuid.
```

Tratá esto como un **riesgo a evaluar** antes de cualquier escenario con reintentos agresivos o redelivery de la cola NFC.

App móvil Android

La **app móvil WorkDone** es la herramienta de campo del personal de limpieza. Corre en celulares Android, lee tags NFC pegados en cada sanitario y registra los trabajos que se hacen. Está diseñada **offline-first**: opera al 100% sin conexión y sincroniza con el backend cuando hay red. Es parte del producto WorkDone de LubecaTech (cliente piloto: Corpal, Córdoba).

Esta página describe para qué sirve, su arquitectura, los flujos de UI principales y cómo registra un trabajo de punta a punta. Para entender cómo viajan los datos al backend ver Sincronización; para las entidades, Modelo de datos.

Para qué sirve y quién la usa

Desde la **v2**, la app maneja **3 roles** y cada uno ve una variante distinta de la misma app:

Rol	Qué hace	Pantalla principal	Historial
`OPERADOR_LIMPIEZA`	Limpia sanitarios	Main de limpieza (verde/gris)	"Mis limpiezas"
`SUPERVISOR`	Controla (trabajo tipo `SUPERVISION`)	Main de control (textos "Controlando:")	"Mis controles"
`ADMIN`	Gestiona tags NFC (alta/asignación)	Gestión NFC	—

El rol llega en la respuesta de login y en `GET /api/v1/app/me`. Si viene `null`, el default seguro es `OPERADOR_LIMPIEZA`.

> **La operación cotidiana es un tap**
>
> El gesto central de toda la app es **apoyar el celular sobre la placa NFC** del sanitario. Un tap abre el trabajo (INGRESO), otro tap lo cierra (EGRESO). Todo lo demás es soporte de ese gesto.

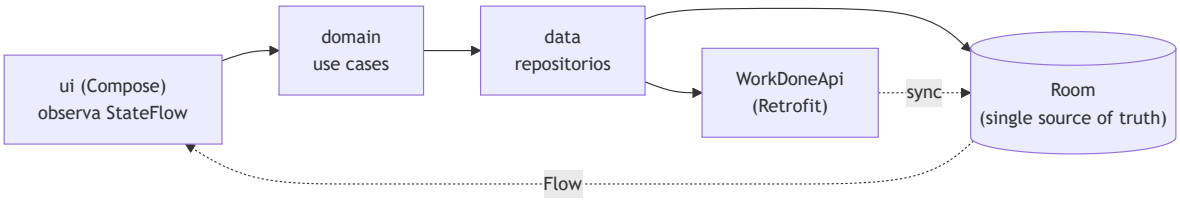
Stack y arquitectura

La app es 100% nativa Android, Kotlin + Jetpack Compose.

Capa	Tecnología
Lenguaje	Kotlin 2.2.10+
UI	Jetpack Compose (Material3)
Persistencia local	Room (SQLite, schema 5, migraciones SQL explícitas)
HTTP/JSON	Retrofit + OkHttp + Moshi
Sync en background	WorkManager (periódico)
Sesión/config	DataStore (Preferences)
Inyección de dependencias	Hilt
Asíncrono	Coroutines + Flow/StateFlow
Min SDK	26 (Android 8.0, por estabilidad NFC)

Arquitectura (Clean lite + MVVM)

El flujo de dependencias es `ui` → `domain` (use cases) → `data` (repositorios).

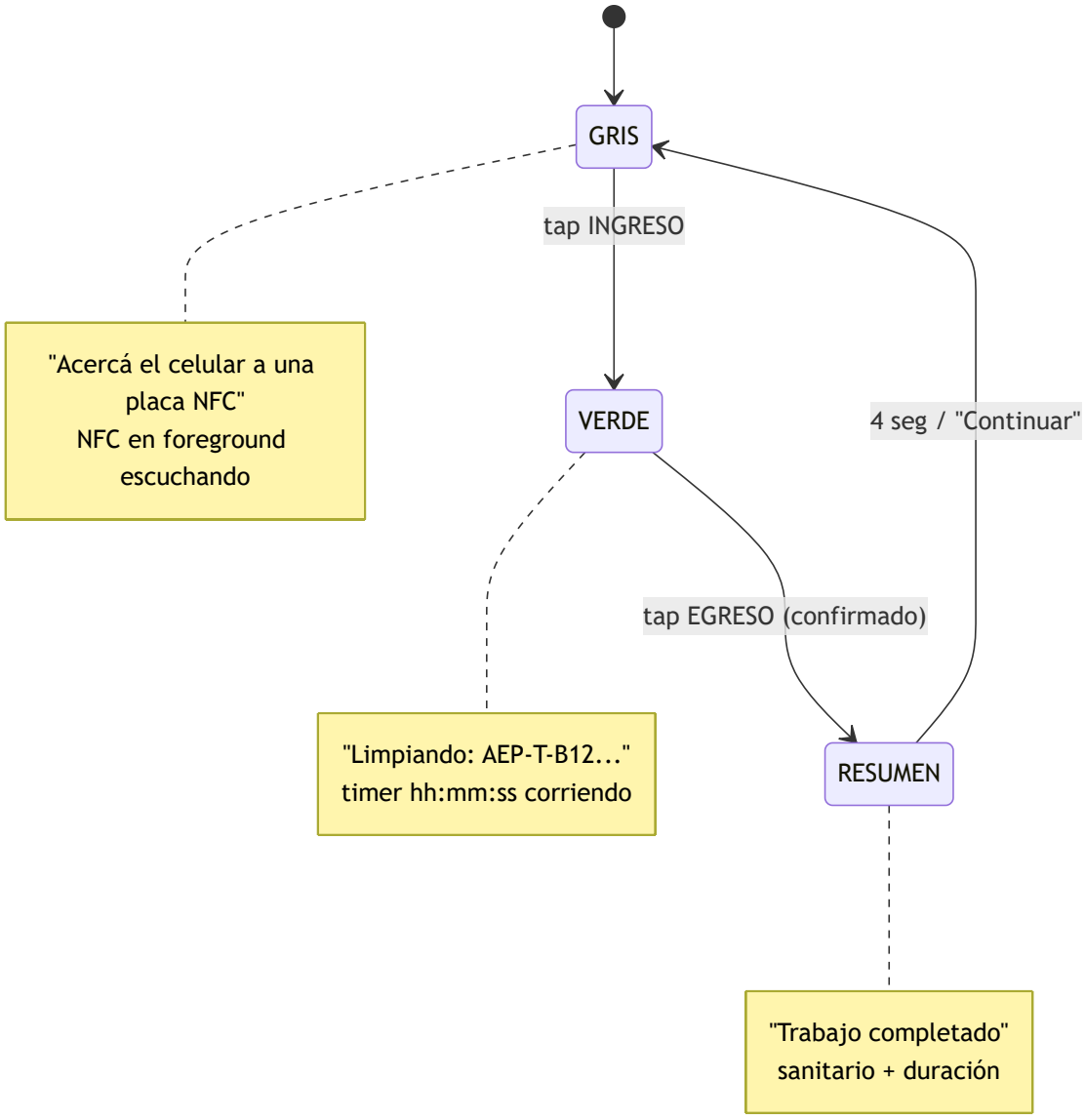


Reglas de arquitectura que importan para entender los flujos:

- **Single source of truth: Room.** Los Composables observan `Flow` del DAO, **nunca** del API directamente.
- **MVVM con StateFlow** desde el ViewModel hacia la UI. No se usa LiveData.
- **La UI es optimista**: al tapear, la app asume éxito y pinta verde/gris al instante. Si el server después rechaza, se reconcilia (ver Flujo B).
- **El timestamp del teléfono al momento del tap es la verdad.** Se captura `Instant.now()` antes de cualquier procesamiento.

Estados de UI principales

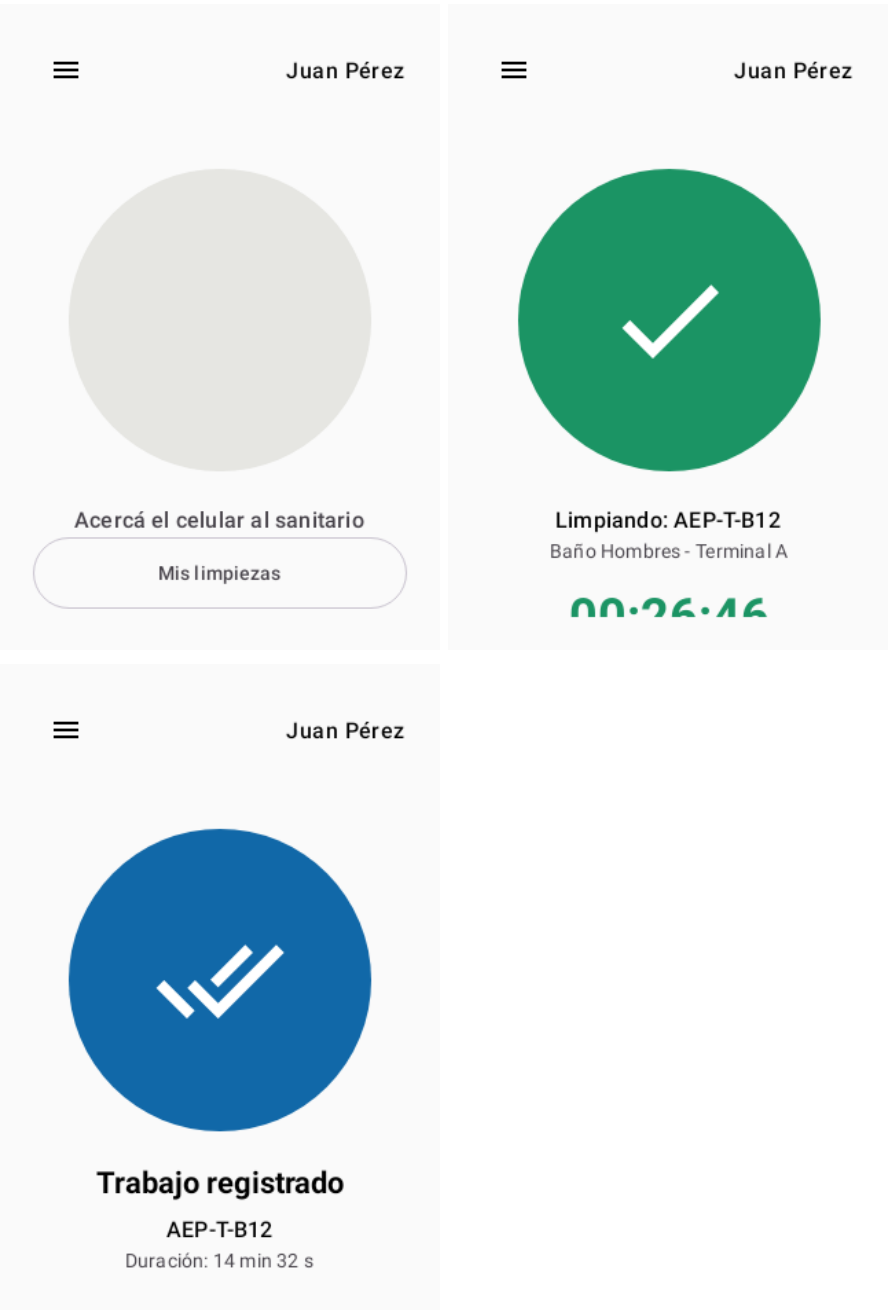
La pantalla principal (Main) es una máquina de estados visual. El componente central es un círculo de color que comunica el estado de un vistazo.



Estado	Color	Significado	Texto
GRIS	`#9C9C99` (⊙)	No estoy limpiando	"Acercá el celular a una placa NFC"
VERDE	`#1B9464` (✓, con pulso)	Limpiando	"Limpiando: AEP-T-B12 - Baño Hombres Terminal A" + timer

AZUL/RESUMEN	`#1168A8` (✓✓)	Post-egreso transitorio (4 seg)	"Trabajo completado" + duración

Los tres estados, tal como los ve el operario en el celular:



Izquierda a derecha: GRIS ("Acercá el celular al sanitario", esperando un tap) · VERDE ("Limpiando: AEP-T-B12 - Baño Hombres - Terminal A" con timer corriendo) · RESUMEN/AZUL ("Trabajo registrado",

AEP-T-B12, "Duración: 14 min 32 s", transitorio 4 seg antes de volver a GRIS). Arriba a la izquierda el menú (drawer); arriba a la derecha el operario logueado.

> **Variante SUPERVISOR**
>
> Para el rol `SUPERVISOR` los estados son idénticos pero los textos dicen **"Controlando:"** en vez de "Limpiando:", el trabajo es tipo `SUPERVISION` y el historial es "Mis controles".

Indicador de red

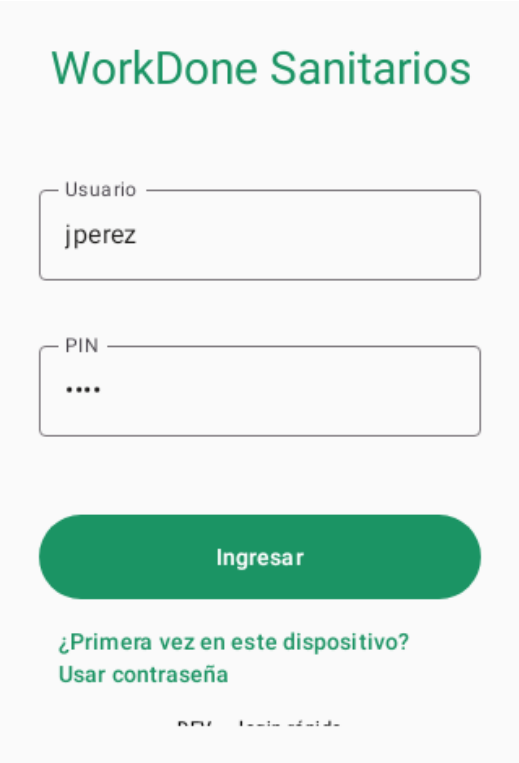
Un chip discreto abajo de la pantalla muestra el estado de conexión **sin bloquear nunca la operación**:

- 🟢 `Conectado` + hora del último sync (ej. "sync: 14:32")
- 🟡 `Sincronizando...` + spinner
- 🔴 `Sin conexión` + cantidad de pendientes (ej. "🔴 Sin conexión · 3 pendientes")

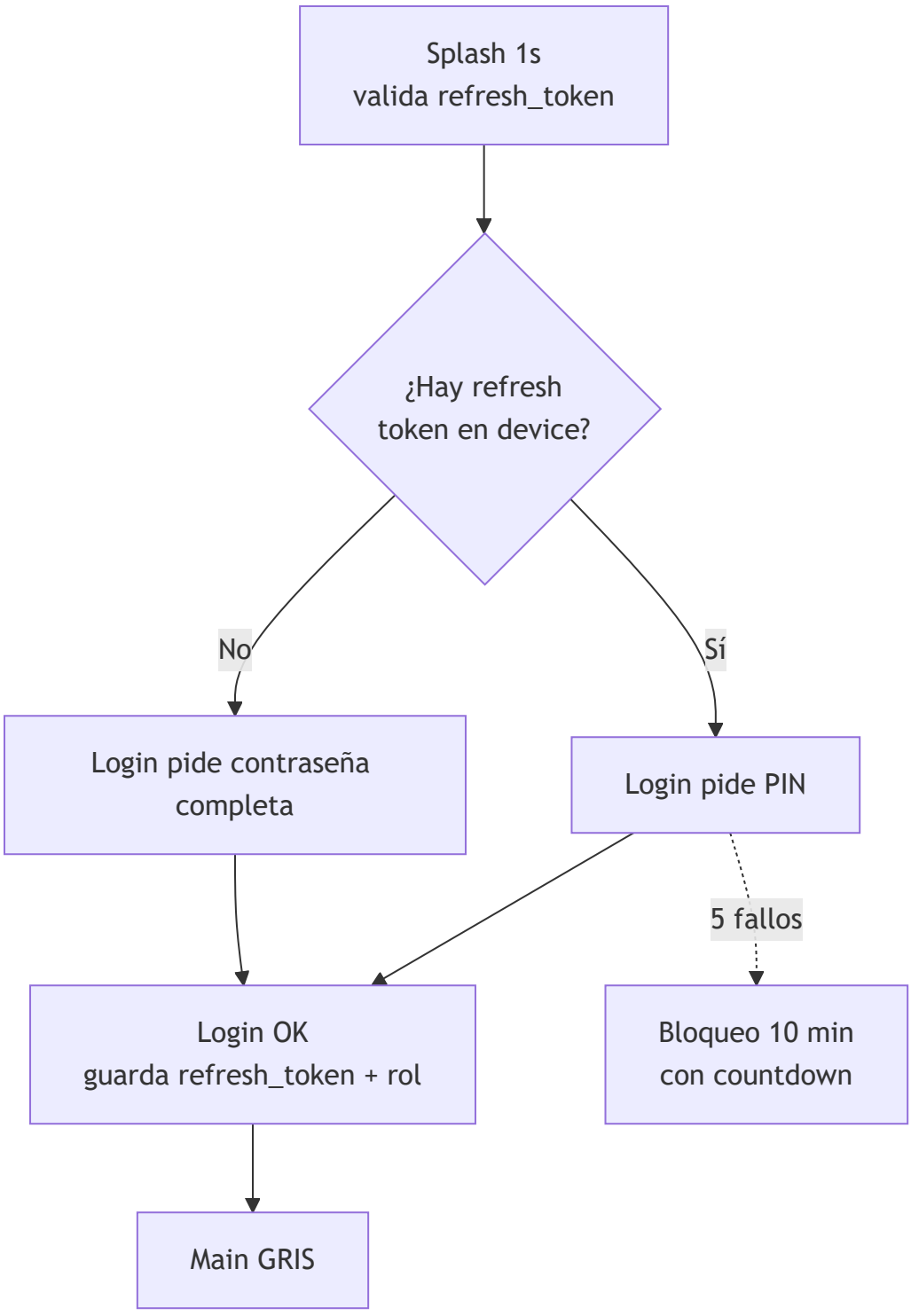
Flujos de UI

Login

La app abre con un **Splash** (1 seg) que valida el `refresh_token` y decide si va a Login o directo a Main.



Pantalla de login. Por defecto pide usuario + PIN (4 dígitos); el link "¿Primera vez en este dispositivo? Usar contraseña" cambia a password completa.



- Por defecto pide **PIN** (más rápido). Si el operario nunca se logueó en ese device, pide **password completa**; después, el siguiente login alcanza con PIN.

- **5 intentos fallidos** → bloqueo de 10 min con mensaje claro y countdown (el backend throttlea 5 fallos por usuario+device → 429 por 10 min).

Flujo A: Tap exitoso INGRESO → EGRESO

Este es el camino feliz: el operario llega al sanitario, tapea para entrar, limpia, y tapea de nuevo para salir.







Kiosk Smiley

El **kiosk Smiley** es una tablet Android en **modo kiosko** montada a la salida de cada sanitario. Es la terminal de opinión del público: el visitante opina con un toque (3 caritas) y, si la opinión fue negativa, opcionalmente indica un motivo. Cierra el ciclo de calidad del ecosistema WorkDone: el equipo registra lo que **hace** (app móvil), la supervisión lo que **controla**, y Smiley lo que el público **percibe**.

Vive en el repo `workdone-smiley-kiosk`, que **no tiene backend propio**: es otro cliente del backend WorkDone, con los mismos patrones que la app móvil. Se autentica como **Dispositivo** (api_key), sin login de usuario. Para el protocolo de sync ver Sincronización; para las entidades, Modelo de datos.

Para qué sirve y la experiencia del visitante

El loop completo del módulo:

> el visitante opina con un toque → el motor de tendencias detecta negativas acumuladas → acción correctiva (alerta + semáforo rojo) → el operario limpia (flujo NFC de siempre) → una limpieza válida resetea la tendencia → el sistema mide si las opiniones mejoraron.

La experiencia para el visitante es deliberadamente mínima: sale del sanitario, ve tres caritas grandes, toca una y listo. Sin texto que leer, sin login, sin pasos obligatorios.

Decisión	Valor
Escala	**3 caritas**: POSITIVA / NEUTRA / NEGATIVA. Un toque.
Motivos	Solo en negativas, segunda pantalla, opcional (catálogo: SUCIO, SIN_INSUMOS, MAL_OLOR, ALGO_ROTO, OTRO)
Hardware	Tablet Android en modo kiosko + soporte antivandálico
Idle	"Último servicio: hace X min" — **sin nombre del operario**

Targets táctiles	≥ 20 mm (público general, de apuro, a veces con manos mojadas)

Reglas de oro heredadas del ecosistema

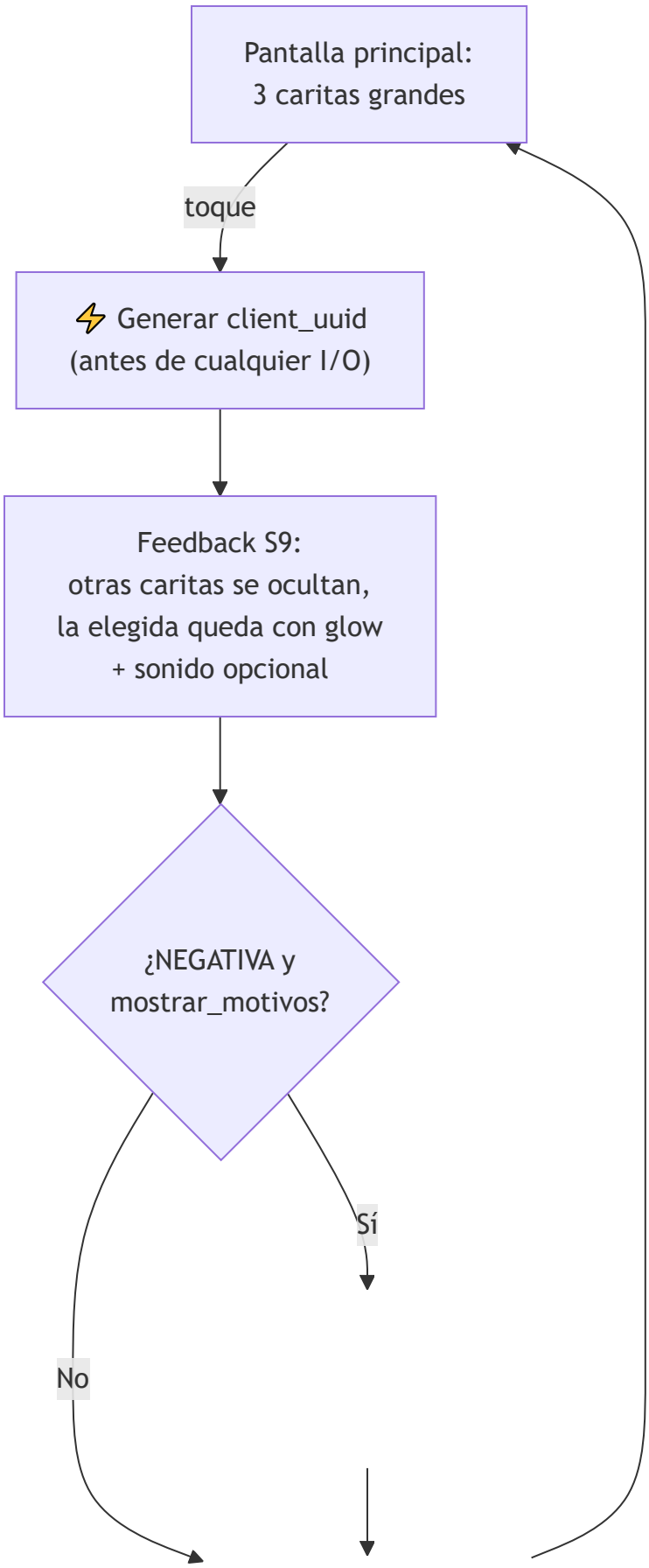
- `client_uuid` se genera **al toque**, antes de cualquier I/O. Es la identidad del evento.
- **Offline-first real**: la terminal opera 100% sin red (solo el idle pierde frescura). Nada se pierde, nada se duplica (idempotencia por `client_uuid`).
- `ts_local_device` es la verdad temporal del evento.
- El device **no define su identidad legible**: `numero_terminal` y `alias` los asigna el backend al vincular; la app los muestra, no los edita.

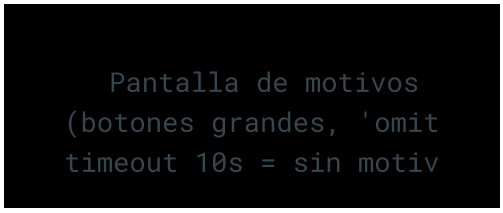
Stack

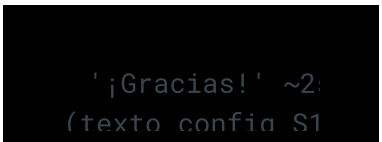
- **Kotlin + Jetpack Compose** (mismas versiones/convenciones que la app móvil).
- **Room mínimo**: `opinion_pendiente` (PK `client_uuid`, `OnConflictStrategy.IGNORE`) + `config_local`.
- **WorkManager**: sync periódico (15 min) + sync inmediato tras cada opinión + al recuperar red.
- **Retrofit/Moshi** contra `/api/v1/smiley/*`.
- **DataStore** para `api_key`, `device_uuid` y datos de vinculación (api_key cifrada en reposo).
- **i18n desde el día 1**: todo texto visible en `strings.xml` (es-AR default). La terminal va a aeropuertos: EN/PT se agregan después sin tocar código.

El flujo de opinión paso a paso

La pantalla principal son las **3 caritas grandes**. El visitante toca una y la terminal lo guía según el valor elegido.


Pantalla principal:
3 caritas grandes
toque
⚡ Generar client_uuid
(antes de cualquier I/O)
Feedback S9:
otras caritas se ocultan,
la elegida queda con glow
+ sonido opcional
¿NEGATIVA y
mostrar_motivos?
Sí
No


Pantalla de motivos
(botones grandes, 'omit
timeout 10s = sin motiv


'¡Gracias!' ~2
(texto config S1

Setup del entorno de desarrollo

Bienvenido al ecosistema **WorkDone Sanitarios** (LubecaTech). Esta guía te lleva, paso a paso, desde un Windows limpio hasta tener los tres proyectos corriendo en tu máquina:

- **Backend** Spring Boot (la fuente de verdad de todo el ecosistema).
- **App móvil Android** para operarios de limpieza.
- **Terminal Smiley (kiosk)** Android de opinión del público.

Al terminar vas a poder: levantar el backend en local y entrar a Swagger, compilar e instalar la app móvil en un device/emulador, y compilar el kiosk corriendo su suite de tests.

> **Orden recomendado**
>
> Arrancá **siempre por el backend**. Tanto la app móvil como el kiosk son *clientes* del backend: consumen sus endpoints REST y comparten sus contratos (`SYNC_PROTOCOL.md` , `API.md`). Sin el backend levantado no vas a poder probar el flujo de punta a punta.

Prerequisitos

Estas son las versiones exactas que el ecosistema espera. Las sacamos de los `CLAUDE.md` de cada repo: respetalas, sobre todo las del backend (Java 25) y las de los proyectos Android (JDK 24 para compilar).

Backend

Herramienta	Versión	Notas
Java (JDK)	**25**	LTS-track. Mínimo Java 21 si bajás por algún motivo.
Spring Boot	4.0.6	Lo trae el proyecto, no lo instalás aparte.

Maven	**3.9.16+**	No Gradle. El repo trae wrapper (`./mvnw`).
Node	**24+**	Requerido para el CLI de JHipster (Node 25 también funciona).
JHipster CLI	9.1.0	Scaffolder inicial.
Base de datos (dev)	H2 con disco	En `./target/h2db/`, sin instalar nada.
Base de datos (prod/local opcional)	MSSQL Server 2019+	SQL Server nativo en Windows, **no necesita Docker**.

> **JDK 21 también instalado**
>
> El backend documenta correr PIT (mutation testing) con un **JDK ≤ 24** porque PIT 1.19.1 no corre sobre JVM Java 25. Si vas a hacer mutation testing, tené un JDK 21 a mano. Para el día a día alcanza con Java 25.

App móvil Android y Kiosk Smiley

Ambos proyectos Android comparten stack y convenciones (el kiosk usa "las mismas versiones/convenciones que workdone-mobile salvo justificación").

Herramienta	Versión	Notas
JDK (toolchain)	**24**	Compila con bytecode target Java 21. JDK 25 dispara un warning falso de Kotlin 2.2.10.
Kotlin	2.2.10+	Compose Compiler Plugin nativo, sin kapt.
Android Gradle Plugin (AGP)	9.2.1+	
Gradle	9.4.1+	Via wrapper (`./gradlew`).

Android Studio	**Quail 1** (stable, junio 2026) o superior	Panda 4 también sirve, sin diferencia funcional.
Min SDK	26 (Android 8.0)	Por estabilidad NFC.
Target / Compile SDK	36 (Android 16)	Compile SDK con `minorApiLevel = 1`.

> **Una sola instalación de Android Studio para los dos repos Android**
>
> Tanto la app móvil como el kiosk son proyectos Gradle estándar de Android. El SDK que instala Android Studio (Compile SDK 36) y el JDK 24 te sirven para los dos.

---

1. Backend (Spring Boot)

Empezá por acá. Es la fuente de verdad del ecosistema.

Clonar e instalar dependencias

```
git clone <url-del-repo> workdone_backend
cd workdone_backend
```

El proyecto trae el Maven Wrapper, así que no necesitás instalar Maven globalmente: `./mvnw` resuelve y baja las dependencias en el primer build.

Levantar en dev (H2, sin base de datos externa)

La forma más rápida de arrancar es con el perfil dev, que usa H2 en disco y DevTools:

```
./mvnw
```

Deberías ver...

El arranque de Spring Boot en consola y, una vez levantado, podés entrar a **Swagger UI** en:

- Swagger UI: `http://localhost:8080/swagger-ui/index.html`
- OpenAPI JSON: `http://localhost:8080/v3/api-docs`

El perfil dev incluye los api-docs, así que si Swagger carga, el backend está sano.

(Opcional) Levantar contra MSSQL local

Si querés probar contra SQL Server nativo en Windows (no necesita Docker):

```
# Prerequisito, 1 sola vez: CREATE DATABASE workdone; + un login SQL con
db_owner.
# Liquibase crea el schema y los seeds al levantar.

# Credenciales por variables de entorno:
$env:MSSQL_USER='sa'
$env:MSSQL_PASSWORD='...'   # ver application-mssql.yml

./mvnw -Pdev,-webapp -Dspring.profiles.active=dev,mssql
```

Verificar

```
# Tests unitarios (surefire — EXCLUYE los *IT)
./mvnw test
```

- [ ] El backend levanta con `./mvnw` y Swagger UI responde en `http://localhost:8080/swagger-ui/index.html`
- [ ] `./mvnw test` pasa en verde

> **Credenciales de desarrollo (seed dev)**
>
> Hay **dos mundos de auth** distintos en dev:
>
> - **BackOffice web** (`/api/authenticate`): `admin` / `admin` (SOLO dev).
> - **App móvil** (`/api/v1/app/auth/*` , tabla `operario`): por ejemplo `jperez` / PIN `1234` / password `Pa$$w0rd!` (rol ADMIN). El usuario `admin` NO sirve para la app móvil.
>
> Guardá estas credenciales: las vas a usar cuando pruebes la app móvil contra tu backend local.

---

2. App móvil Android (operarios)

Es un cliente del backend: lee tags NFC, registra trabajos, opera 100% offline-first y sincroniza. Repo separado.

Clonar y abrir

```
git clone <url-del-repo> workdone_mobile
cd workdone_mobile
```

Abrí el proyecto con **Android Studio Quail 1** (o superior). El proyecto trae el Gradle Wrapper, así que las dependencias se resuelven en el primer build/sync de Gradle. Asegurate de que el **JDK toolchain sea 24** (la config del proyecto ya lo fija; Android Studio lo respeta).

> **A qué backend apunta**
>
> El build `debug` apunta por defecto a `https://workdone-dev.lubeca.tech/api/v1/` (definido en `buildConfigField API_BASE_URL`). Si querés que pegue contra **tu backend local**, vas a tener que ajustar esa URL — pero eso ya es trabajo de desarrollo, no de setup inicial.

Compilar e instalar en device/emulador

```
# Build debug + instalar en el device conectado
./gradlew installDebug
```

Verificar

```
# Tests unitarios
./gradlew test

# Lint
./gradlew lint

# Ver logs filtrados de la app
adb logcat -s WorkDone:*
```

- [ ] `./gradlew installDebug` instala la app en un device/emulador
- [ ] `./gradlew test` y `./gradlew lint` pasan en verde

> ⚠ **Problemas comunes (documentados en el repo)**
>
> - **El emulador de Android Studio NO soporta NFC nativamente.** Es una limitación de larga data. Para el desarrollo cotidiano usá el **modo dev de la app**: en build `debug`, tap 5 veces sobre el logo de la pantalla principal abre un panel oculto que simula taps con UUIDs de prueba. Es la herramienta de uso diario (10x más rápido que cualquier emulación NFC).
> - **Para validar el flujo NFC real**, hace falta un device físico (el del proyecto es un Ulefone RugKing 2 Pro, Android 15) y tags **NTAG215/213 grabables**. Una tarjeta contactless de pago/acceso NO sirve: no tiene la URL WorkDone en NDEF, el reader la ignora.
> - **Room 2.6.x falla** con KSP2 + Kotlin 2.2 ("unexpected jvm signature V"): usá Room **2.8.4+**.
> - **Hilt < 2.59 NO soporta AGP 9**: usá Hilt 2.59.2+.

---

3. Terminal Smiley (kiosk)

App Android en modo kiosko para la terminal de opinión del público. **No tiene backend propio**: es otro cliente del `workdone-backend`, con los mismos patrones que la app móvil.

Clonar y abrir

```
git clone <url-del-repo> workdone-smiley-kiosk
cd workdone-smiley-kiosk
```

Es un proyecto Gradle de Android igual que la app móvil; abrilo con Android Studio y dejá que Gradle sincronice. Usa **Kotlin + Jetpack Compose con las mismas versiones/convenciones que workdone-mobile** (mismo JDK toolchain 24, mismo Compile SDK 36).

> ⚠ **Este repo NO documenta un comando de build/run de la app**
>
> El `CLAUDE.md` del kiosk **solo documenta la suite de tests JVM** (ver abajo). No hay un comando de "levantar/instalar" la app documentado en las fuentes consultadas, ni un equivalente a `installDebug`. Para correr la terminal en una tablet real interviene el **modo kiosko** (Lock Task / dedicated launcher, autoarranque al boot) y un **flujo de vinculación** que se hace con un código generado en el BackOffice — eso es instalación de campo, no setup de dev. Si necesitás levantar la app más allá de los tests, confirmá el procedimiento con AndresM en vez de asumirlo.

Verificar

```
# Tests JVM
./gradlew test
```

- [ ] `./gradlew test` pasa en verde

> 🔥 **Hook pre-push (red de seguridad local)**
>
> El repo trae un hook `pre-push` que corre los tests JVM antes de dejarte pushear. Activalo **una vez por clon**:
>
> ```
> git config core.hooksPath scripts/git-hooks
> ```
>
> Para saltearlo puntualmente: `git push --no-verify`.

> ✏ **Cómo se instala una terminal en el lugar (contexto, no es setup de dev)**
>
> La puesta en marcha de una terminal física (montaje, "Probar conexión" contra `GET /api/v1/smiley/ping`, ingreso del código de vinculación, menú técnico con 10 toques sobre el logo) está documentada en la guía de instalación de terminal Smiley. La terminal **no se configura en el lugar**: textos, sonido, logo y reglas se administran remoto desde el BackOffice y bajan en el próximo sync.

Checklist final

Cuando los tres tildes estén verdes, tenés el entorno listo:

- [ ] **Backend** levanta con `./mvnw` y Swagger responde en `http://localhost:8080/swagger-ui/index.html`
- [ ] **App móvil** instala con `./gradlew installDebug` y sus tests/lint pasan
- [ ] **Kiosk** compila y `./gradlew test` pasa en verde

A partir de acá, para entender los contratos entre backend y clientes, seguí por la referencia del backend.

How-to guides

Documentación **orientada a tareas**. Recetas para resolver un problema concreto, asumiendo que el lector ya sabe lo básico.

A diferencia de un tutorial, una how-to no enseña: da los pasos para lograr un objetivo específico.

Guías disponibles

Guía	Para qué
Provisionar una tablet (kiosk Smiley)	Dejar una tablet nueva vinculada a un sanitario y opinando.
Registrar y asignar un tag NFC	Dar de alta una placa NFC y asociarla a un sanitario.
Montar un entorno desde cero	Crear todo el entorno en orden: empresa → sucursal → sector → sanitario → tags → operarios → rutas.
El portal del cliente	Darle acceso a un cliente y entender qué ve en su panel.

Por hacer

- Cómo correr las migraciones de Liquibase.
- Cómo desplegar a producción.

Provisionar una tablet nueva para el kiosk Smiley

Esta guía te lleva paso a paso por la puesta en marcha de una tablet nueva como terminal de opinión Smiley. **Al terminar, la tablet queda vinculada a un sanitario y opinando**: muestra el idle con "Último servicio: hace X min" y encola/sincroniza opiniones contra el backend WorkDone.

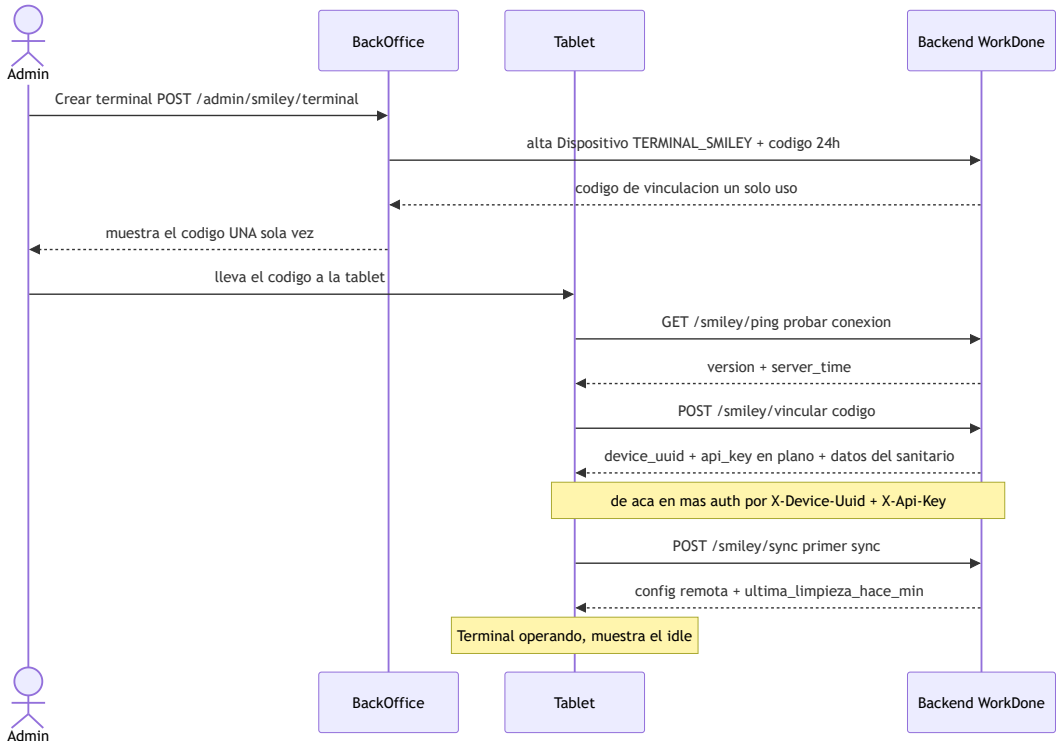
La terminal no tiene login de usuario: se autentica como **Dispositivo** de tipo `TERMINAL_SMILEY`, vinculado a **un único sanitario**. La identidad y las credenciales las asigna el backend al vincular. Para el contexto del módulo ver Kiosk Smiley; para los contratos REST, la Referencia de API.

Prerequisitos

Necesitás	Por qué	Cómo lo conseguís
Acceso **ADMIN** al BackOffice	Crear la terminal y generar el código de vinculación son operaciones solo-ADMIN (`/api/v1/admin/smiley/terminal`)	Tu usuario tiene que tener `ROLE_ADMIN`
El **sanitario ya creado** en el sistema	La terminal se vincula a un sanitario existente; sin él, el alta devuelve `404`	Verificá el sanitario en el BackOffice antes de empezar
Una **tablet Android** con la app Smiley instalada	Es la terminal física que vas a vincular	Idealmente con **device-owner** para Lock Task silencioso (ver más abajo)
Conectividad para el primer sync	El código de vinculación se canjea online y el primer sync baja la config remota (textos, logo, PIN-hash)	WiFi o red de datos que alcance el backend

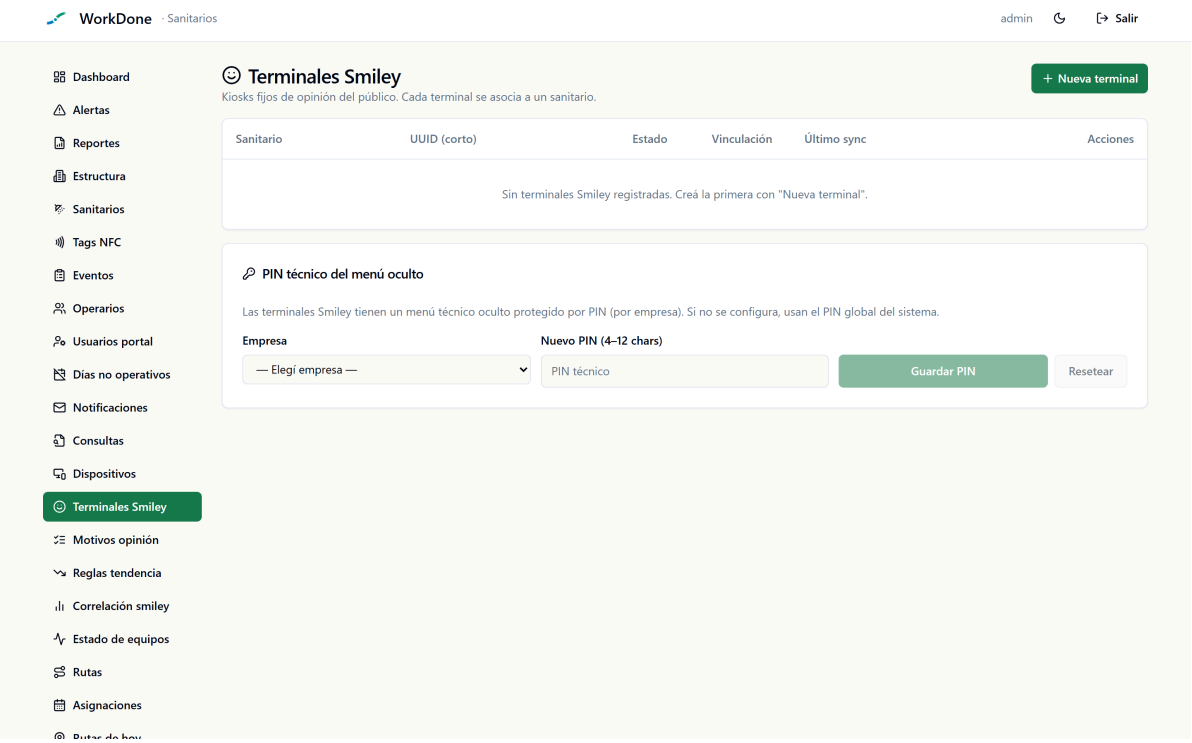
> **Device-owner para un kiosko real**
>
> Una tablet **sin device-owner** funciona, pero NO queda confinada con Lock Task: el visitante puede salir de la app. En ese caso la terminal muestra un banner **"Terminal no asegurada"**. Para un kiosko de producción, provisioná la tablet como device-owner antes de empezar (ver Modo kiosko).

El flujo de vinculación de un vistazo



Paso 1 — Crear la terminal en el BackOffice

Entrá al BackOffice como ADMIN y abrí la pantalla **Terminales Smiley** desde el menú lateral. Vas a ver la lista de terminales (cada una asociada a un sanitario) con su estado de vinculación y último sync. Si todavía no hay ninguna, la pantalla muestra el estado vacío con el aviso *"Sin terminales Smiley registradas. Creá la primera con 'Nueva terminal'."*.



Pantalla Terminales Smiley: la lista de terminales (acá vacía), el botón + Nueva terminal arriba a la derecha y, abajo, la tarjeta para fijar el PIN técnico del menú oculto por empresa.

Para crear la terminal:

1. Hacé clic en **+ Nueva terminal** (arriba a la derecha).
2. Elegí el **sanitario** donde va a ir montada la tablet.
3. Confirmá. La terminal aparece en la lista con su **número/alias** asignados y el estado **vinculación pendiente** (código pendiente, todavía sin canjear).

Al crearla, el backend genera un **código de vinculación de un solo uso con vencimiento a 24 h** y te lo muestra.

Bajo el capó

Al confirmar, el BackOffice dispara:

```
POST /api/v1/admin/smiley/terminal
Content-Type: application/json

{ "sanitario_id": 42, "nombre": "Smiley baño hombres - PB" }
```

El campo `nombre` es opcional. El backend da de alta un `Dispositivo` de tipo `TERMINAL_SMILEY` vinculado a ese sanitario y devuelve el código de vinculación:

```
{
  "id": 17,
  "sanitario_id": 42,
  "numero_terminal": 5,
  "alias": "Smiley · S-042",
  "codigo_vinculacion": "K7P2M9QX",
  "codigo_expira_en": "2026-06-21T15:30:00-03:00",
  "vinculada": false,
  "codigo_pendiente": true
}
```

El código se muestra una sola vez — copialo ahora

Copiá el código apenas aparece (la UI lo muestra en un modal de un solo uso). Si lo perdés, no lo recuperás: tenés que regenerarlo (ver Problemas comunes).

Deberías ver la terminal nueva en la lista de **Terminales Smiley** con estado de vinculación pendiente (código pendiente) y su número/alias asignados.

Errores posibles en este paso

- `404` — el `sanitario_id` no existe. Verificá el sanitario antes de reintentar.
- `409` — ya hay una terminal **activa** para ese sanitario. El sistema permite **una sola** terminal activa por sanitario; desvinculá la vieja o elegí otro sanitario.

Paso 2 — Preparar la tablet

Antes de tocar la app, dejá la tablet lista como kiosko físico:

- **Montaje y alimentación**: fijá el soporte antivandálico a la salida del sanitario, a altura de mano, con la pantalla bien visible. La tablet queda **siempre enchufada**.
- **Modo kiosko / device-owner**: para un kiosko real, la tablet tiene que estar como **device-owner** (Lock Task confina la app, sin barras de sistema). Sin device-owner aparece el banner "Terminal no asegurada".
- **Orientación horizontal**: la app bloquea la orientación en **horizontal** (`sensorLandscape`) — va montada en pared y no debe rotar a vertical.
- **Autoarranque**: la app arranca **sola al boot** y tiene watchdog de reinicio; no hace falta abrirla a mano tras un corte de luz.

Deberías ver la app Smiley arrancar en horizontal al encender la tablet. En el **primer arranque** (sin vincular) aparece directo la **pantalla de setup** — sin gesto ni PIN, porque todavía no hay nada que proteger.

Paso 3 — Setup en la tablet: URL del server y probar conexión

En la pantalla de setup:

1. **Verificá la URL del server**. Viene precargada con el default del build; confirmá que apunta al backend correcto.
2. Tocá **"Probar conexión"**. Por debajo, la app llama al endpoint público (sin credenciales):

```
GET /api/v1/smiley/ping
```

que responde:

```
{ "version": "...", "server_time": "2026-06-20T15:31:02-03:00" }
```

Deberías ver una confirmación de conexión OK. Si falla, **frená acá** y revisá red/URL antes de seguir — el código no se va a poder canjear sin conexión.

> **La URL bien formada importa**
>
> La URL tiene que ser `http(s)://host[:puerto]` **sin** ruta ni query. Una URL con path la rechaza la app antes de mutar nada.

Paso 4 — Vincular: canjear el código

Ingresá el **código de vinculación** del Paso 1 en el campo correspondiente y confirmá. La app llama a:

```
POST /api/v1/smiley/vincular
Content-Type: application/json

{ "codigo": "K7P2M9QX" }
```

El backend valida el código (un solo uso, no expirado), lo consume y responde con las credenciales y la identidad de la terminal:

```
{
  "device_uuid": "8f3a1c20-...-9e21",
  "api_key": "sk_live_3f9a...PLANO_UNA_SOLA_VEZ",
  "sanitario_id": 42,
  "numero_terminal": 5,
  "alias": "Smiley · S-042",
  "sanitario_codigo": "S-042",
  "sanitario_nombre": "Baño hombres - Planta Baja"
}
```

> ⚠ **La `api_key` viaja en texto plano UNA sola vez**
>
> El backend solo persiste el **hash BCrypt** de la api_key — el texto plano viaja únicamente en esta respuesta y no se puede volver a pedir. La app la guarda cifrada en reposo (DataStore). Si se pierde, hay que **regenerar el código y re-vincular**.

De acá en más la terminal se autentica en todos los endpoints `/api/v1/smiley/*` (salvo `/ping` y `/vincular`) con los headers:

```
X-Device-Uuid: 8f3a1c20-...-9e21
X-Api-Key: sk_live_3f9a...
```

El backend valida que el dispositivo exista, esté activo, sea `TERMINAL_SMILEY` y que el hash BCrypt coincida; falla **cerrado** si no. El `sanitario_id` **nunca viaja en el body del sync**: lo determina el server por el vínculo del dispositivo, así una terminal no puede opinar por otro sanitario.

Deberías ver la app pasar del setup a la **pantalla operativa** (idle / caritas). Los campos `numero_terminal`, `alias`, `sanitario_codigo` y `sanitario_nombre` alimentan la línea de

diagnóstico y el menú técnico.

Paso 5 — Verificar que quedó andando

Cerrá el lazo con tres chequeos:

1. **Pantalla idle**: en reposo, la terminal muestra **"Último servicio: hace X min"** (sin nombre del operario). Si no hay limpieza registrada o el dato es viejo, la línea **se oculta** — es el comportamiento correcto, no un error.
2. **Menú técnico → identidad**: hacé **10 toques sobre el logo Lubeca** (máx. ~2,5 s entre toques) → ingresá el **PIN técnico** (el de la empresa o el default global; verifica offline). En la sección **Info/diagnóstico** deberías ver: Terminal Nº, alias, sanitario, uuid corto, versión, última sync y **opiniones pendientes en cola**. Tienen que coincidir con lo que muestra el BackOffice para ese device.
3. **Primer sync OK**: desde el menú técnico tocá **"sincronizar ahora"** (o esperá el ciclo automático). El sync (`POST /api/v1/smiley/sync`) baja la config remota (textos, sonido, logo, `pin_tecnico_hash`, `ultima_limpieza_hace_min`).

> **Confirmá contra el tablero de equipos**
>
> En el BackOffice, **Estado de equipos** (`GET /api/v1/admin/dispositivo/estado`) la terminal debería aparecer **EN LÍNEA**, con su número/alias y el sanitario correcto. Una tablet con device-owner aparece como confinada (`lock_task_activo: true`).

Problemas comunes

El código venció o ya se usó → `410`

El código de vinculación es de **un solo uso** y vence a las **24 h**. Si está expirado o ya consumido, `POST /api/v1/smiley/vincular` devuelve **`410`** . Regenerá uno nuevo desde el BackOffice:

```
POST /api/v1/admin/smiley/terminal/{id}/regenerar-codigo
```

y repetí el Paso 4 con el código nuevo.

El código es inválido / mal formado → `400`

Un código mal tipeado o con formato incorrecto devuelve **`400`**. Volvé a copiarlo del BackOffice (cuidado con confundir caracteres) y reintentá.

> **Cómo colapsa la app los errores de vinculación**
>
> La app agrupa los rechazos del backend (`400` mal formado, `404` inexistente, `409` ya usado, `410` expirado) en un único estado "código inválido/expirado/usado" para mostrarte un mensaje claro. Si ves ese error, lo más probable es código expirado o ya canjeado → regeneralo.

Banner "Terminal no asegurada"

Aparece cuando la tablet **no tiene device-owner**: la app no puede confinarse con Lock Task y el visitante podría salir. La terminal igual opina, pero para producción provisioná la tablet como device-owner y reabrí la app. En builds debug/DEMO_MODE el banner nunca aparece (no es un kiosko real).

Necesito mover la terminal o anular la vinculación

Para **desvincular** una terminal (la deja inactiva, liberando el sanitario para una terminal nueva):

```
POST /api/v1/admin/smiley/terminal/{id}/desvincular
```

Para **mover la terminal a otro sanitario**: creá una terminal nueva en el otro sanitario (Paso 1) y desvinculá esta. Recordá que solo puede haber **una terminal activa por sanitario** (`409` si intentás duplicar).

Cambié la URL del server y se desvinculó sola

Es por diseño: editar la URL del server desde el menú técnico **borra las credenciales** y vuelve la terminal al estado **no-vinculado**. La `api_key` jamás viaja a un server distinto del que la emitió, así que un cambio de URL exige **un código nuevo** del BackOffice y volver a vincular (Paso 4).

La tablet se quedó sin conexión

Opera igual: la terminal es **offline-first**. Las opiniones se encolan localmente (idempotentes por `client_uuid`) y suben cuando vuelve la red. Lo único que pierde frescura es el "Último servicio:

hace X min" del idle.

> **Regla de oro: la terminal NO se configura en el lugar**
>
> Textos, sonido, logo y reglas se administran **remoto** desde el BackOffice y bajan en el próximo sync. El menú técnico es solo para diagnóstico, conexión y re-vinculación — nunca configuración funcional.

Registrar y asignar un tag NFC

Esta guía te lleva paso a paso por la tarea concreta de **dejar operativo un tag NFC en blanco**: darlo de alta en stock y asignarlo a un sanitario, para que cuando el personal apoye el celular sobre la placa, el tap registre trabajos contra el sanitario correcto.

> **Alcance: qué es 'registrar un dispositivo NFC' acá**
>
> En WorkDone Sanitarios, **registrar un dispositivo NFC** significa dar de alta una **placa/tag NFC** — la entidad `TagNfc`, el chip físico que se pega al sanitario — y asignarla a un sanitario. **NO** se trata de registrar un teléfono ni un dispositivo de operario (eso es otra cosa: el `Dispositivo` que se vincula en el login). Acá hablamos del **chip que el personal acerca con el celular** para registrar una limpieza.

El ciclo de vida completo del tag (estados, transiciones válidas y auditoría) vive en la referencia Ciclo de vida de los tags NFC. Esta página es la receta operativa; aquella, el porqué.

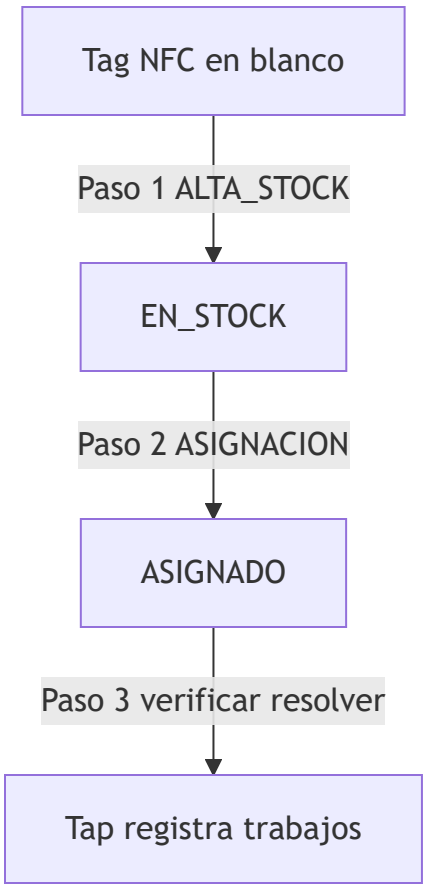
Prerequisitos

Necesitás	Detalle	Cómo verificarlo
Rol **ADMIN**	Las mutaciones NFC desde la app móvil (`alta-stock`, `asignar`, `desasignar`, `reasignar`) son **solo ADMIN**. Un operador de limpieza o supervisor no puede ejecutarlas.	Drawer → Configuración → "Rol del operario actual" debe decir **ADMIN**. La pantalla **Gestión NFC** solo aparece para ADMIN.
Un **tag NFC físico en blanco**	NTAG215 o NTAG213. En producción se usan los antimetal (Atlantico3D) sobre superficie metálica; para pruebas cualquier NTAG común alcanza.	Un chip que al resolverlo dé `NO_REGISTRADO`.

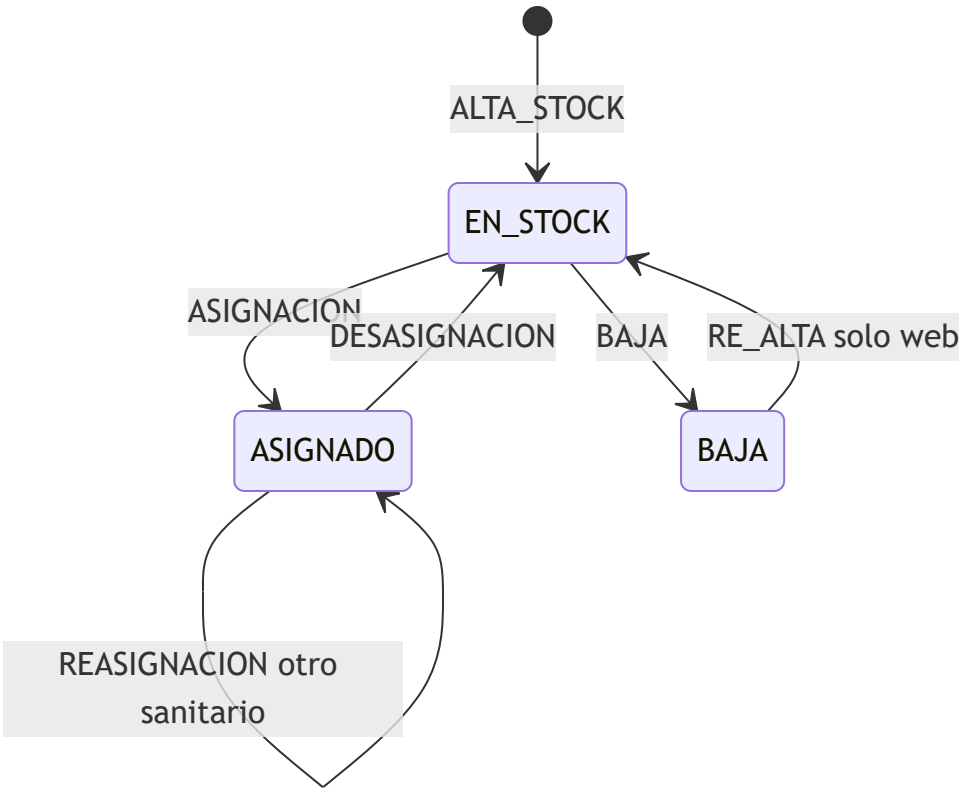
El **sanitario destino** ya creado	El tag se asigna a un sanitario existente (ej. `AEP-T-B12`).	Aparece en el árbol Sucursal›Sector›Sanitario de `GET /api/v1/app/sanitarios/disponibles` (con `tiene_tag` por cada uno).
La **app móvil** logueada como ADMIN	El alta de stock **solo se hace desde la app**, porque hay que leer el chip onsite. No existe alta de stock en el BackOffice web.	Que la app esté sincronizando (chip 🟢 Conectado abajo).

> ⚠ **El alta de stock NO se puede hacer desde la web**
>
> `ALTA_STOCK` requiere leer el chip físico en el lugar, así que **solo existe en la app móvil** (rol ADMIN). El BackOffice web puede desasignar, reasignar, dar de baja y re-dar de alta, pero **no** dar de alta en stock.

El recorrido en una imagen



Y los tres estados posibles del tag, con las transiciones que vas a usar:



Cómo leer el diagrama

- `ALTA_STOCK` es **idempotente**: si el tag ya existe, devuelve el existente sin generar evento.
- `REASIGNACION` es un **bucle** sobre `ASIGNADO`: el tag cambia de sanitario en una sola operación atómica, sin pasar por `EN_STOCK`.
- `BAJA` sale **solo** desde `EN_STOCK`. Un tag `ASIGNADO` hay que desasignarlo primero.
- `RE_ALTA` (`BAJA → EN_STOCK`) está disponible **solo desde el BackOffice web**, nunca por la app/sync.

Paso 1 — Alta en stock

Con la app en la pantalla **Gestión NFC**, apoyá el celular sobre el tag en blanco. La app resuelve el estado (`NO_REGISTRADO`) y te ofrece "**Dar de alta en stock**". Confirmá con un alias descriptivo (ej. `tag #14 lote mayo`).

La app móvil esperando el tap: "Acercá el celular a un tag". Apoyás el celular sobre el chip para que la app lo lea y resuelva su estado. El campo "DEV — simular tag (UUID)" solo aparece en builds de desarrollo, para simular el tap sin un chip físico.

Esto **graba/registra el tag como** `EN_STOCK` en el backend. El endpoint que la app encola y envía por sync es:

```
POST /api/v1/app/nfc/alta-stock
Authorization: Bearer <jwt-operario-ADMIN>
Content-Type: application/json
```

```
{
  "uuid_tag": "aaaa1111-1111-1111-1111-111111111111",
  "alias": "tag #14 lote mayo"
}
```

Es idempotente: podés re-leer sin miedo

`alta-stock` es **idempotente por uuid normalizado**: si el tag ya existe, el backend **devuelve el tag actual sin crear evento ni modificar nada**. Volver a apoyar el celular sobre un tag ya dado de alta no rompe nada ni duplica registros.

Grabar el tag físico (NDEF)

Si además vas a **grabar el chip** (con una app tipo "NFC Tools" / "NFC TagWriter"), escribí en el NDEF la URL `https://workdone.lubeca.tech/t/{uuid}`, donde `{uuid}` es el mismo `uuid_tag` que diste de alta. Ese UUID es la **fuente de verdad**: el backend normaliza el UUID (trim, mayúsculas, sin espacios) en cada lectura, así que el formato no importa, pero el valor sí tiene que coincidir.

Resultado verificable: al resolver el tag, su estado pasa a `EN_STOCK`.

Paso 2 — Asignar a un sanitario

Con el tag ya `EN_STOCK`, la app te ofrece **"Asignar a sanitario"**. Elegí el sanitario destino del árbol Sucursal›Sector›Sanitario y confirmá. La app encola y sincroniza:

```
POST /api/v1/app/nfc/asignar
Authorization: Bearer <jwt-operario-ADMIN>
Content-Type: application/json
```

```
{
  "uuid_tag": "aaaa1111-1111-1111-1111-111111111111",
  "sanitario_id": 12
}
```

Esto ejecuta la transición `EN_STOCK → ASIGNADO`: el tag queda vinculado al sanitario, con `asignado_en` seteado y `activo = true`.

> ⚠ **Conflicto de asignación: el primero en sincronizar gana**
>
> Solo puede haber **un tag `ASIGNADO` por sanitario** (constraint en la base). Si el sanitario destino **ya tiene** un tag asignado, o si el tag no está `EN_STOCK`, la operación devuelve `CONFLICTO_ASIGNACION` (HTTP 409). Cuando dos dispositivos asignan offline al mismo sanitario, **el primero en el orden de sincronización gana** y el segundo recibe el conflicto (el orden lo define `ts_local_device ASC`). Si te pasa, la app revierte el optimismo local y te avisa.

Resultado verificable: al resolver el tag, su estado pasa a `ASIGNADO` y trae los datos del sanitario.

Paso 3 — Verificar

Confirmá que el tag quedó operativo resolviéndolo:

```
GET /api/v1/app/nfc/resolver/aaaa1111-1111-1111-1111-111111111111
Authorization: Bearer <jwt-operario>
```

Respuesta esperada:

```
{
  "estado": "ASIGNADO",
  "alias": "tag #14 lote mayo",
  "sanitario": {
    "id": 12,
    "codigo": "AEP-T-B12",
    "nombre": "Baño Hombres Terminal A",
    "sector": "Terminal A",
    "sucursal": "AEP"
  }
}
```

Prueba final (la que vale): que un operario de limpieza apoye el celular sobre el tag. Debería pasar a estado **VERDE** y arrancar el timer del trabajo. La app resuelve taps **solo** contra tags en estado `ASIGNADO`, así que si llegaste hasta acá, el tag ya está operativo.

Operaciones adicionales

Una vez asignado, el tag puede moverse o retirarse. Todas estas mutaciones pasan por el mismo servicio (`TagNfcLifecycleService`) y dejan auditoría.

Reasignar a otro sanitario

Mueve el tag de un sanitario a otro en **un solo paso atómico**, sin pasar por `EN_STOCK`:

```
POST /api/v1/app/nfc/reasignar
```

```
{
  "uuid_tag": "aaaa1111-1111-1111-1111-111111111111",
  "sanitario_nuevo_id": 20,
  "notas": "se movió la placa al baño contiguo"
}
```

> **Reasignar NO es 'desasignar + asignar'**
>
> `REASIGNACION` genera **un único** evento de auditoría con `sanitario_anterior` **y** `sanitario_nuevo` poblados, para preservar la trazabilidad del movimiento como una sola operación. Caso borde: si el sanitario nuevo es el mismo que el anterior, no se valida el conflicto.

Desasignar (devolver a stock)

```
POST /api/v1/app/nfc/desasignar
```

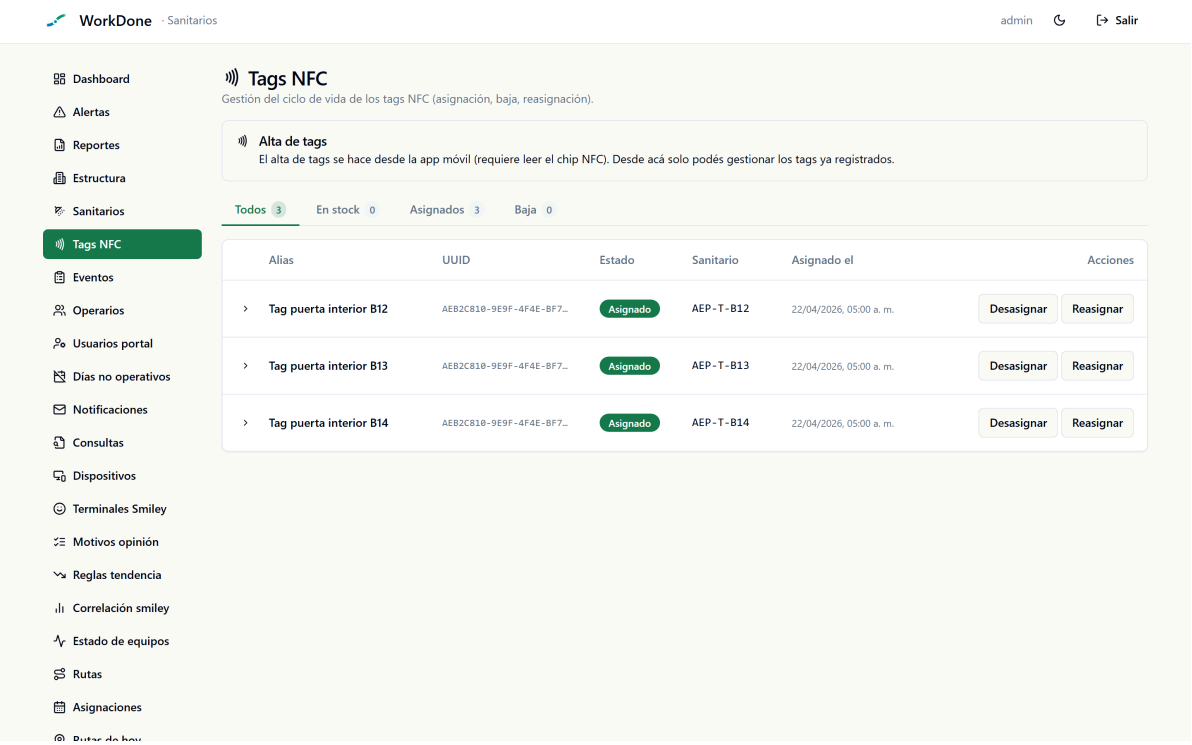
```
{ "uuid_tag": "aaaa1111-1111-1111-1111-111111111111" }
```

Limpia el sanitario y devuelve el tag a `EN_STOCK`.

Dar de baja

La **baja** (desactivar definitivamente) y la **re-alta** se hacen desde el **BackOffice web**, no por la app móvil. Recordá: `BAJA` solo es posible desde `EN_STOCK` — un tag asignado hay que desasignarlo primero.

Desde el BackOffice, la pantalla **Tags NFC** lista todos los tags ya registrados y te deja gestionarlos: filtrarlos por estado (Todos / En stock / Asignados / Baja), y desasignar o reasignar cada uno con los botones de la columna **Acciones**.



La pantalla Tags NFC del BackOffice: tabla con Alias, UUID, Estado, Sanitario y fecha de asignación, más las acciones Desasignar / Reasignar por fila. El aviso "Alta de tags" recuerda que el alta se hace desde la app móvil (requiere leer el chip onsite); desde acá solo se gestionan los tags ya registrados.

Tabla de transiciones válidas

Acción	De → A	Dónde
`ALTA_STOCK`	`[*]` → `EN_STOCK`	**Solo app móvil** (leer el chip onsite)
`ASIGNACION`	`EN_STOCK` → `ASIGNADO`	App o web
`DESASIGNACION`	`ASIGNADO` → `EN_STOCK`	App o web
`REASIGNACION`	`ASIGNADO` → `ASIGNADO` (otro sanitario, atómico)	App o web

`BAJA`	`EN_STOCK` → `BAJA`	App o web — **solo desde** `EN_STOCK`
`RE_ALTA`	`BAJA` → `EN_STOCK`	**Solo web (BackOffice)**

Problemas comunes

403 / `ROL_INSUFICIENTE` — 'no me deja dar de alta ni asignar'

Las mutaciones NFC son **solo ADMIN**. Si tu operario no es ADMIN, el backend rechaza la operación con HTTP 403 y código `ROL_INSUFICIENTE`. Verificá tu rol en Configuración. En la cola de sync, los ítems NFC de un operario no-ADMIN se ignoran sin abortar el sync.

`TRANSICION_INVALIDA` — 'la operación no aplica al estado del tag'

La máquina de estados rechazó la operación porque el tag no está en el estado de origen esperado (HTTP 409). Ejemplos: dar de **baja** un tag `ASIGNADO` (hay que desasignar primero), o **desasignar/reasignar** un tag que no está `ASIGNADO`. Resolvé el tag primero (`GET /nfc/resolver/{uuid}`) para ver en qué estado está y qué transición corresponde.

`CONFLICTO_ASIGNACION` — 'el sanitario ya tiene tag'

El sanitario destino ya tiene un tag `ASIGNADO`, o el tag no está `EN_STOCK` (HTTP 409). Solo puede haber un tag asignado por sanitario: desasigná el actual o reasignalo, según corresponda.

`TAG_DESCONOCIDO` al tapear — 'el tag no registra nada'

Si un operario apoya el celular sobre un tag que **no fue dado de alta** (whitelist estricta: tag no registrado = inutilizable), el tap genera un trabajo en modo `OFFLINE_PENDIENTE_RESOLUCION` y dispara una alerta `TAG_DESCONOCIDO`. La solución es volver al **Paso 1** y dar de alta ese UUID.

Auditoría: todo cambio queda registrado

Cada transición exitosa persiste un `TagNfcEvento` (tabla inmutable). El evento captura el tag, el UUID normalizado, la acción, los sanitarios anterior/nuevo, el operario, el dispositivo, la fecha y las notas. **Nada cambia el estado de un tag sin dejar rastro.** El detalle completo del modelo de auditoría está en Ciclo de vida de los tags NFC.

Para el contrato completo de los endpoints NFC, ver API HTTP — App móvil · gestión NFC. Para entender la máquina de estados en profundidad, Ciclo de vida de los tags NFC. Si todavía no tenés el backend corriendo para probar esto, empezá por Montar el entorno del backend.

Montar un entorno desde cero (alta y asociación de entidades)

Esta guía te lleva, paso a paso y **en el orden correcto**, a dar de alta un entorno operativo completo de **WorkDone Sanitarios** desde una base vacía. Al terminar tenés un entorno operativo: una empresa con sucursales, sectores, sanitarios con sus tags, operarios, y rutas asignadas — listo para que el personal de campo empiece a tapear.

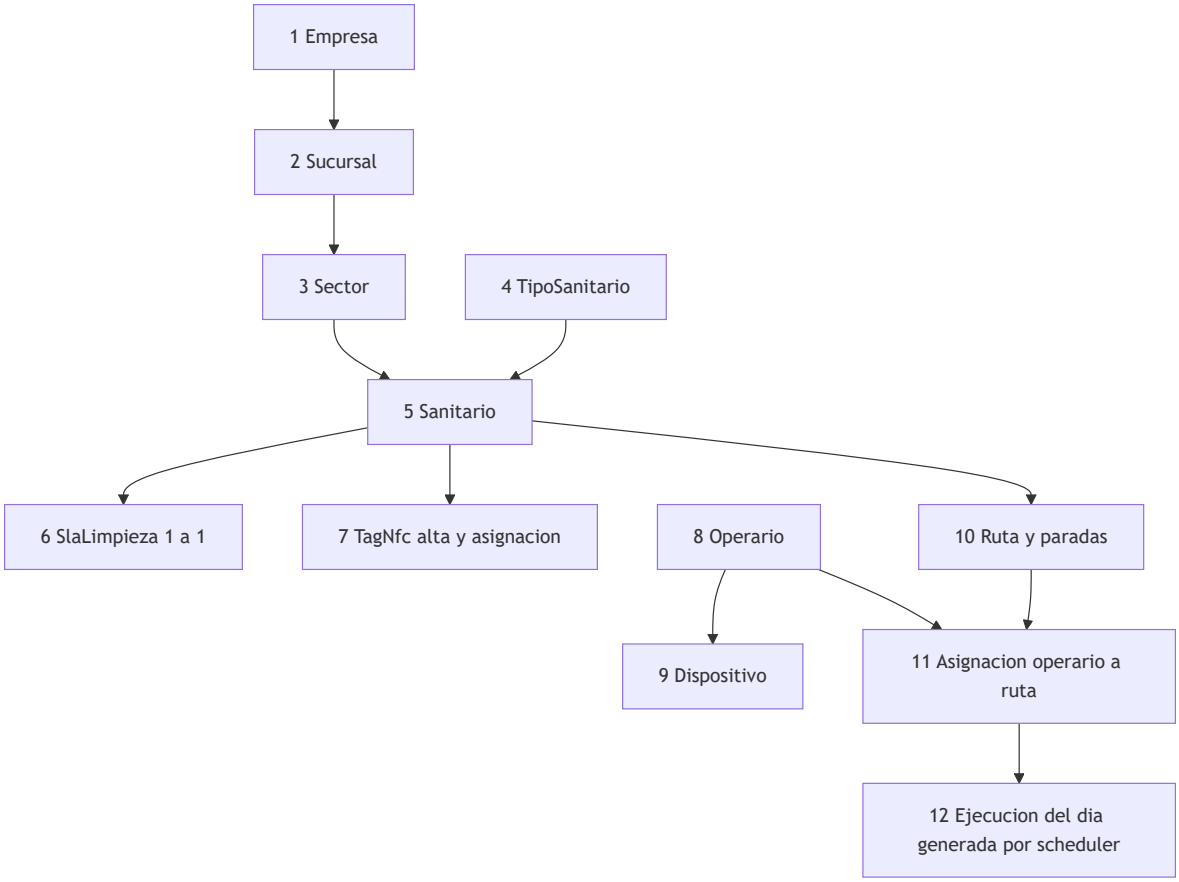
Todo lo que hacés acá se hace en el **BackOffice web**: la aplicación React que el propio backend sirve en `http://localhost:8080` (en desarrollo entrás con `admin` / `admin`). Cada paso de esta guía es **una pantalla real** del menú lateral. Para los integradores que prefieren el API, debajo de cada paso dejamos el endpoint y el JSON exactos en un bloque "**Bajo el capó**".

El **orden importa**: cada entidad depende de la anterior por una FK (foreign key). No podés crear un Sector sin su Sucursal, ni un Sanitario sin su Sector. Seguí la secuencia y no vas a chocar contra un `404` de "padre inexistente".

> **A quién apunta esta guía**
>
> A un **administrador** (rol `ADMIN` / `ROLE_ADMIN`) que opera el BackOffice. La sesión web de admin y todos los endpoints `/api/*` del CRUD scaffold y los `/api/v1/admin/*` exigen `ROLE_ADMIN`; un operario de campo recibe `403`. Si todavía no sabés cómo se autentica el BackOffice, mirá Sincronización y autenticación.

El orden de dependencias

Este diagrama es el mapa de toda la guía. Cada flecha es una dependencia: la entidad de destino necesita que la de origen exista primero.



Cómo se expresan las FKs bajo el capó (CRUD scaffold JHipster)

Cuando creás una entidad desde la pantalla, el BackOffice arma el body por vos. Si vas por API directo, recordá: en el CRUD scaffold (`/api/*`) una FK **no** se manda como `empresa_id`, se manda como un **objeto anidado** con el `id` del padre. Por ejemplo, una Sucursal referencia a su empresa así:

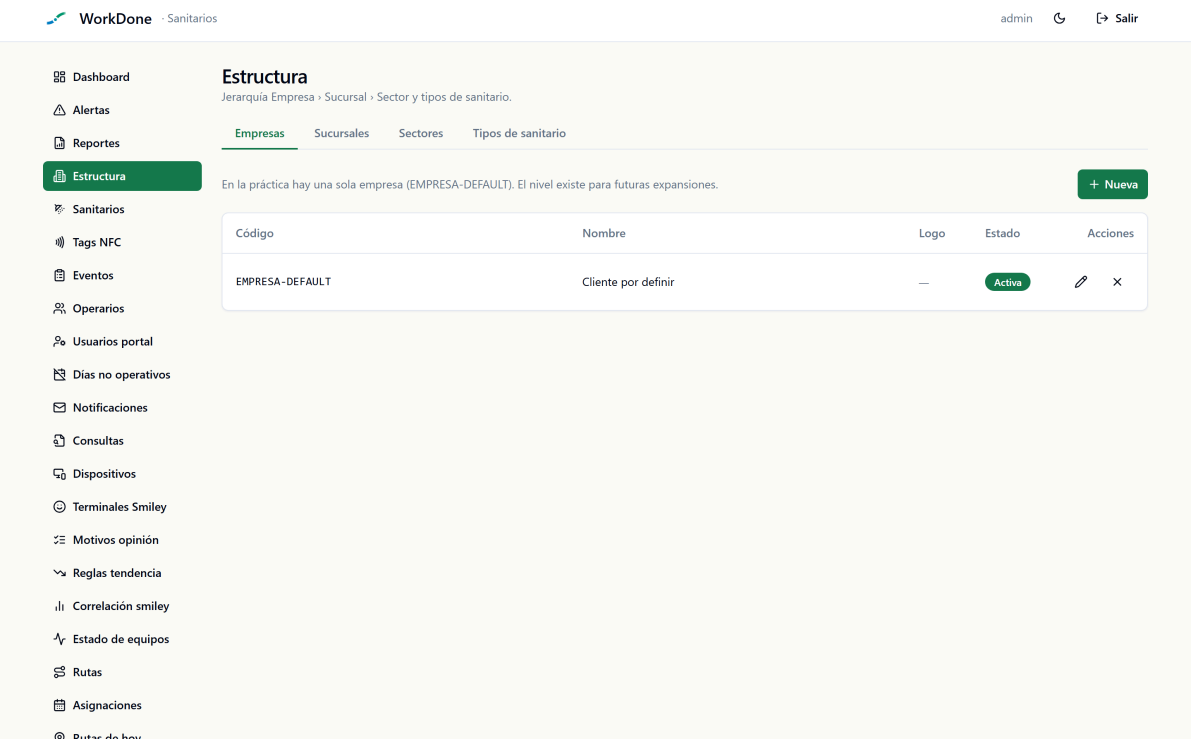
```
{ "empresa": { "id": 1 } }
```

En cambio, los endpoints de **rutas** (`/api/v1/admin/ruta*`) usan un contrato propio con FKs planas en **snake_case** (`ruta_id`, `operario_id`). Cada paso de abajo muestra el formato exacto que corresponde — no los mezcles.

Paso 1 — Empresa (la raíz)

Dónde: menú lateral **Estructura** → pestaña **Empresas** → botón **+ Nueva**.

La **Empresa** es el cliente del servicio (ej. AA2000). No tiene dependencias: es la raíz de toda la jerarquía. **No es multi-tenant**; es solo el primer nivel de agrupación. En la práctica suele haber una sola (`EMPRESA-DEFAULT`); el nivel existe para futuras expansiones.



La pantalla Estructura concentra los cuatro primeros niveles de la jerarquía en pestañas: Empresas, Sucursales, Sectores y Tipos de sanitario. Cada pestaña tiene su propio botón + Nueva y su lista.

Bajo el capó (API)

Qué	Detalle
Endpoint	`POST /api/empresas`
FK que necesita	Ninguna
Campos obligatorios	`codigo`, `nombre`, `activo`

```
{
  "codigo": "AA2000",
  "nombre": "Aeropuertos Argentina 2000",
  "activo": true,
  "visibleOpiniones": false
}
```

El `201 Created` devuelve el `EmpresaDTO` con su `id`. Lo necesitás en el paso 2.

Resultado verificable: la nueva empresa aparece en la lista de la pestaña **Empresas** con estado **Active**.

Paso 2 — Sucursal (FK → Empresa)

Dónde: Estructura → pestaña **Sucursales** → **+ Nueva**. Al crearla, elegís la empresa del paso 1 como padre.

Una **Sucursal** es una sede del cliente (ej. AEP, EZE). Pertenece a una Empresa.

Plural a la inglesa (solo en el API)

El path es `/api/sucursals` (así pluralizó JHipster), no `/api/sucursales`. Es una decisión deliberada del proyecto: el dominio en español vive en las entidades, no en las rutas. La UI siempre dice "Sucursales".

Bajo el capó (API)

Qué	Detalle
Endpoint	`POST /api/sucursals`
FK que necesita	`empresa` → del paso 1
Campos obligatorios	`codigo`, `nombre`, `activo`, `empresa`

```
{
  "codigo": "AEP",
  "nombre": "Aeroparque Jorge Newbery",
  "direccion": "Av. Costanera Rafael Obligado s/n",
  "activo": true,
  "empresa": { "id": 1 }
}
```

Resultado verificable: la sucursal aparece en la pestaña **Sucursales**. Su `id` lo usás en el paso 3.

Paso 3 — Sector (FK → Sucursal)

Dónde: Estructura → pestaña **Sectores** → **+ Nueva**, eligiendo la sucursal del paso 2.

Un **Sector** es una zona dentro de la sucursal (ej. Terminal A, Patio de comidas).

El código de Sector es único DENTRO de la sucursal

Existe una UK compuesta `(sucursal_id, codigo)`. Podés tener un sector `"T-A"` en AEP y otro `"T-A"` en EZE sin conflicto, pero **no** dos `"T-A"` en la misma sucursal.

Bajo el capó (API)

Qué	Detalle
Endpoint	`POST /api/sectors`
FK que necesita	`sucursal` → del paso 2
Campos obligatorios	`codigo`, `nombre`, `activo`, `sucursal`

```
{
  "codigo": "T-A",
  "nombre": "Terminal A",
  "activo": true,
  "sucursal": { "id": 1 }
}
```

Resultado verificable: el sector aparece en la pestaña **Sectores**. Su `id` lo usás en el paso 5.

Paso 4 — TipoSanitario (catálogo, sin FK)

Dónde: Estructura → pestaña **Tipos de sanitario**. Suele venir ya **sembrado** (seed: `DAMAS`, `CABALLEROS`, `DISCAPACITADOS`, `MIXTO`); en ese caso solo revisás la lista y saltás al paso 5. Si falta alguno, **+ Nueva**.

TipoSanitario es un catálogo chico que clasifica los sanitarios.

Chequeá el seed antes de crear

Mirá primero qué hay en la pestaña. Si los tipos ya están sembrados, no los dupliques: los reusás al crear el sanitario en el paso 5.

Bajo el capó (API)

Qué	Detalle
Endpoint	`POST /api/tipo-sanitarios` (alta) · `GET /api/tipo-sanitarios` (chequear seed)
FK que necesita	Ninguna
Campos obligatorios	`codigo`, `nombre`, `activo`

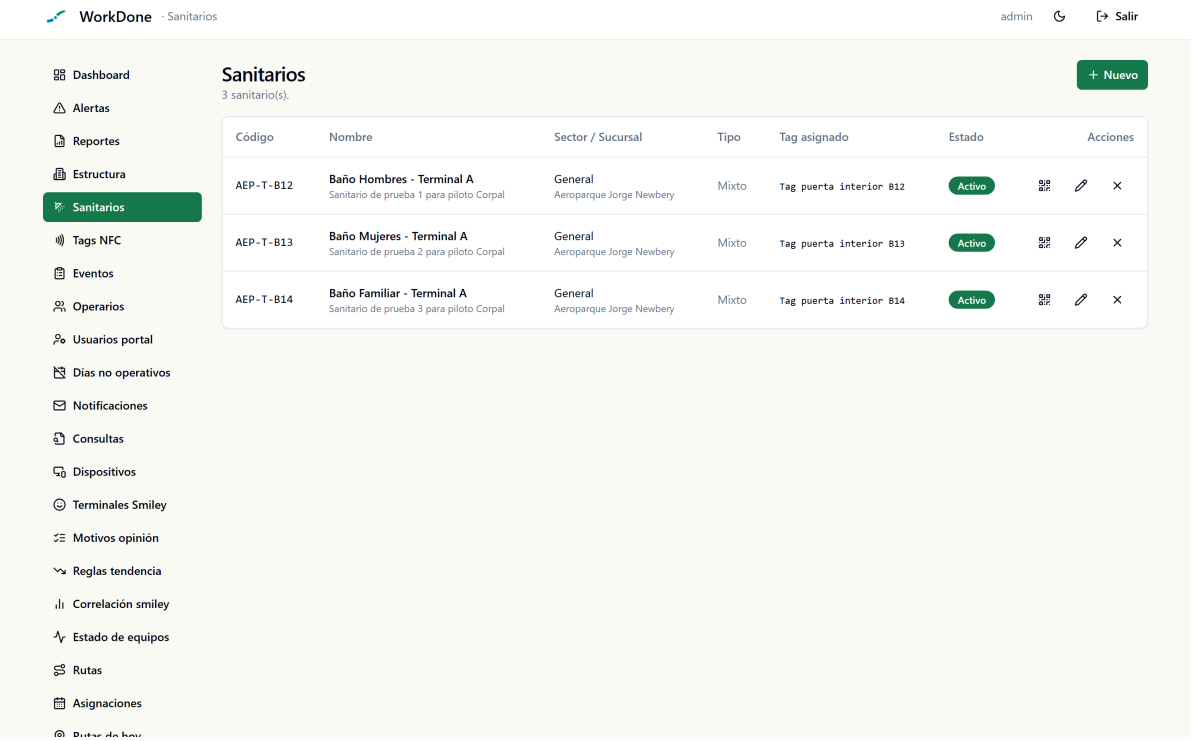
```
{
  "codigo": "DAMAS",
  "nombre": "Damas",
  "activo": true
}
```

Resultado verificable: la pestaña **Tipos de sanitario** lista los tipos disponibles, listos para asignar.

Paso 5 — Sanitario (FK → Sector + TipoSanitario)

Dónde: menú lateral **Sanitarios** → botón **+ Nuevo**. Elegís el sector (paso 3) y, opcionalmente, el tipo de sanitario (paso 4).

El **Sanitario** es el baño concreto donde va pegada la placa NFC. Es el **centro de gravedad** del sistema: la historia de limpiezas, las opiniones y el SLA viven en él.



La pantalla Sanitarios lista cada baño con su código, sector y sucursal, tipo, el tag NFC asignado (paso 7) y su estado. El botón + Nuevo abre el alta.

> **El código de Sanitario es único GLOBAL**
>
> A diferencia del Sector, el `codigo` del Sanitario (ej. `"AEP-T-B12"`) es único en todo el sistema, no por sucursal.

Bajo el capó (API)

Qué	Detalle
Endpoint	`POST /api/sanitarios`
FK que necesita	`sector` (obligatoria, paso 3) + `tipoSanitario` (opcional, paso 4)
Campos obligatorios	`codigo`, `nombre`, `activo`, `sector`

```
{
  "codigo": "AEP-T-B12",
  "nombre": "Baño Damas Terminal A planta 1",
  "descripcion": "Frente a puerta de embarque 12",
  "activo": true,
  "sector": { "id": 1 },
  "tipoSanitario": { "id": 1 }
}
```

Resultado verificable: el sanitario aparece en la lista de **Sanitarios** (todavía sin tag). Su `id` lo usás en los pasos 6, 7 y 10.

Paso 6 — SlaLimpieza (1:1 con el Sanitario)

Dónde: se configura desde la ficha del sanitario (en el detalle/edición de la pantalla **Sanitarios**). Es **1:1**: como máximo un SLA por sanitario (`sanitario` es UK).

La **SlaLimpieza** define la exigencia de limpieza de un sanitario: cada cuánto, en qué ventana horaria y qué días.

`diasSemana` y los días sin exigencia

`diasSemana` es un patrón de **7 caracteres** `Lun..Dom` (`LMXJVSD`). Una letra presente = se exige limpieza ese día; ausente = el sanitario queda en estado `SIN_EXIGENCIA` (gris, sin alertas SLA). El default del schema es `'LMXJVSD'` (todos los días). Más detalle en Reglas operativas.

Bajo el capó (API)

Qué	Detalle
Endpoint	`POST /api/sla-limpiezas`
FK que necesita	`sanitario` → del paso 5
Campos obligatorios	`frecuenciaMin`, `ventanaDesde`, `ventanaHasta`, `activo`, `sanitario`

```
{
  "frecuenciaMin": 120,
  "ventanaDesde": "06:00:00",
  "ventanaHasta": "23:00:00",
  "duracionMinimaSeg": 30,
  "diasSemana": "LMXJVSD",
  "activo": true,
  "sanitario": { "id": 1 }
}
```

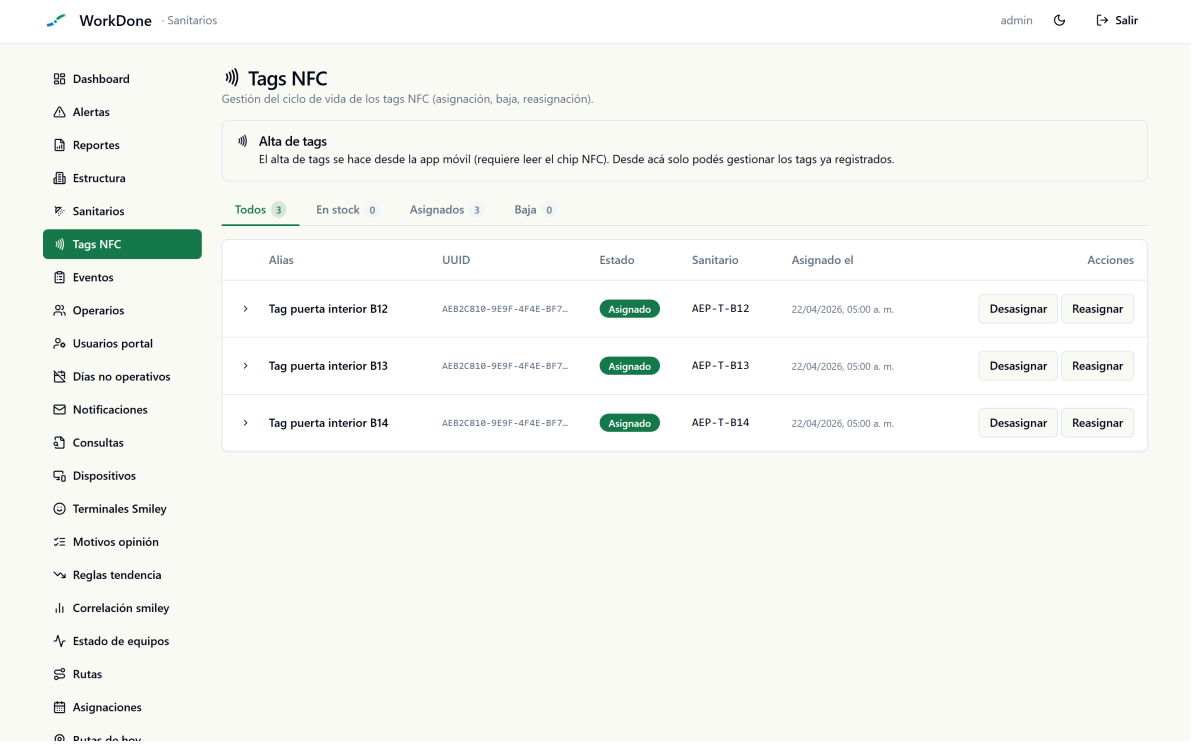
Validaciones: `frecuenciaMin` está acotado (mín. 5, máx. 1440 min). `ventanaDesde` / `ventanaHasta` son horas (`LocalTime` , formato `HH:mm:ss`), no strings libres.

Resultado verificable: el sanitario ya tiene su reloj de SLA; en el **Dashboard** (paso 12) aparece su columna `SLA` con la frecuencia (ej. `c/60m`). Recordá que solo una **limpieza válida** lo resetea.

Paso 7 — TagNfc: alta de stock + asignación al sanitario

Dónde: menú lateral **Tags NFC**. La pantalla lista todos los tags con sus pestañas (Todos / En stock / Asignados / Baja) y permite **Desasignar** / **Reasignar** desde el BackOffice.

Ponerle una placa física a un sanitario son **dos transiciones** de la máquina de estados del tag: primero **alta de stock** (`EN_STOCK`), después **asignación** al sanitario (`ASIGNADO`).



La pantalla Tags NFC muestra el ciclo de vida de cada tag (alias, UUID, estado, sanitario, fecha de asignación). El propio BackOffice avisa: "El alta de tags se hace desde la app móvil (requiere leer el chip NFC). Desde acá solo podés gestionar los tags ya registrados."

> ⚠️ **El alta de stock es SOLO desde la app móvil (rol ADMIN)**
>
> Dar de alta un tag requiere **leer el chip onsite**, así que el alta de stock (`POST /api/v1/app/nfc/alta-stock`) se hace exclusivamente desde la app móvil con un operario `ADMIN`. **No existe alta de stock por web** — la pantalla Tags NFC lo deja claro. La asignación, en cambio, se puede hacer desde la app o desde acá.

Como es un flujo con sus propias reglas (whitelist estricta, normalización de UUID, máximo 1 tag asignado por sanitario, auditoría en `TagNfcEvento`), está documentado aparte:

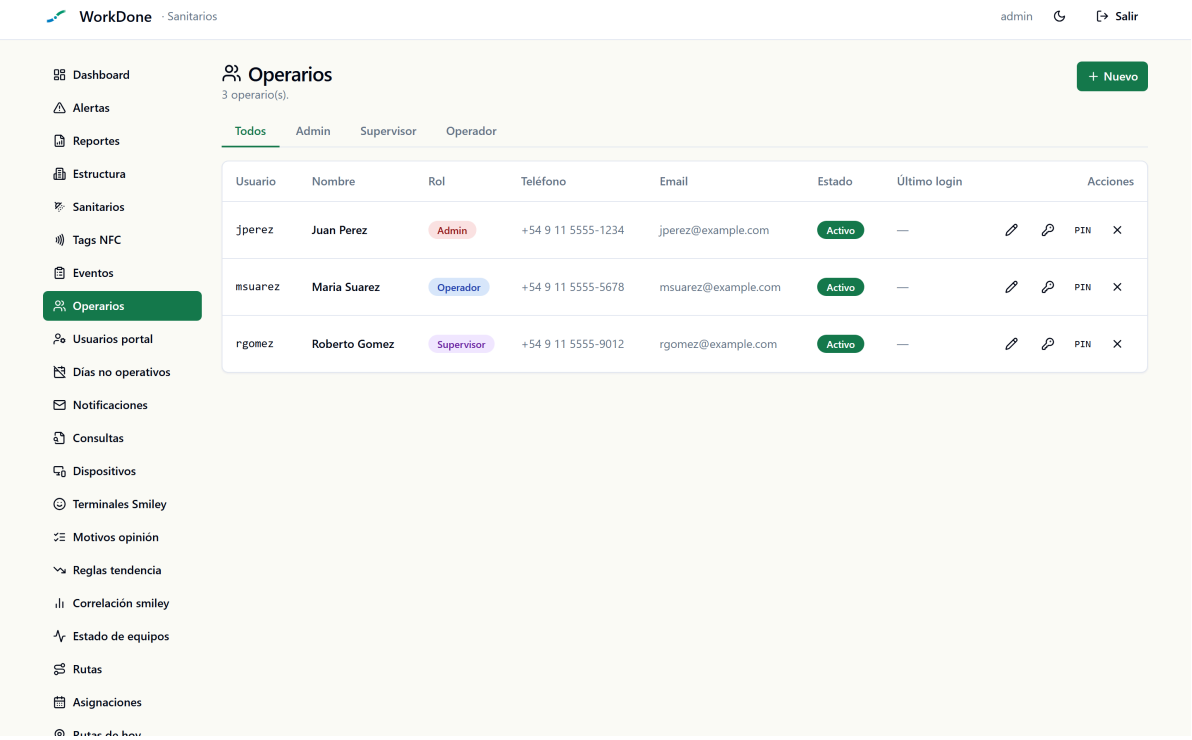
➡️ **Seguí la guía dedicada: Registrar y asignar un tag NFC.**

Resultado verificable: en la pestaña **Asignados** de Tags NFC, tu tag figura ligado al sanitario del paso 5. Bajo el capó, `GET /api/v1/app/nfc/resolver/{uuid}` devuelve `{ "estado": "ASIGNADO", "sanitario": { ... } }`.

Paso 8 — Operario (con su rol)

Dónde: menú lateral **Operarios** → botón **+ Nuevo**.

El **Operario** es la identidad del personal en la app móvil. Su `rol` decide qué hace en el sistema.



La pantalla Operarios lista al personal con su usuario, nombre, rol (chip de color), teléfono, email y último login. Las pestañas superiores filtran por rol; las acciones por fila incluyen editar, resetear password (ícono de llave) y resetear PIN.

Valores de `rol` (enum `RolOperario`):

Rol	Qué hace
`OPERADOR_LIMPIEZA`	Operario de campo; sus taps crean trabajos `LIMPIEZA`.
`SUPERVISOR`	Supervisa; sus taps crean trabajos `SUPERVISION`.
`ADMIN`	No crea trabajos por tap — su app es la de Gestión NFC (alta/asignación de tags).

La password y el PIN son write-only y obligatorios en el alta

`nuevaPassword` (mín. 8, máx. 128 chars) y `nuevoPin` (4 a 8 dígitos) **se cargan en el alta** y el server los hashea con BCrypt. Los hashes (`pinHash`, `passwordHash`) tienen `@JsonIgnore`: **nunca** vuelven en la respuesta ni viajan al móvil. Para cambiarlos después, usá las acciones de la fila (resetear password / resetear PIN), que bajo el capó son `POST /api/admin/operarios/{id}/reset-password` y `/reset-pin`.

Bajo el capó (API)

Qué	Detalle
Endpoint	`POST /api/operarios` (exige `ROLE_ADMIN`)
FK que necesita	Ninguna obligatoria (`user` → `jhi_user` es opcional)
Campos obligatorios	`usuario`, `nombre`, `apellido`, `activo`, `rol`, `nuevaPassword`, `nuevoPin`

```
{
  "usuario": "jperez",
  "nombre": "Juan",
  "apellido": "Pérez",
  "telefono": "+54 11 5555-0000",
  "email": "jperez@corpal.com",
  "activo": true,
  "rol": "OPERADOR_LIMPIEZA",
  "nuevaPassword": "unaClaveSegura123",
  "nuevoPin": "4821"
}
```

Resultado verificable: el operario aparece en la lista de **Operarios** con estado **Activo**. Ya puede loguearse en la app (`POST /api/v1/app/auth/login-password` o `login-pin`). Su `id` lo usás en el paso 11.

Paso 9 — Dispositivo del operario

Dónde: menú lateral **Dispositivos**, donde se listan y se habilitan/deshabilitan los dispositivos registrados.

Para que un teléfono pueda sincronizar, su **Dispositivo** tiene que estar registrado y activo. Un dispositivo no registrado (o deshabilitado) que llama a `POST /api/v1/app/sync` recibe **`401`** (cuerpo `DEVICE_REVOCADO` cuando estaba revocado).

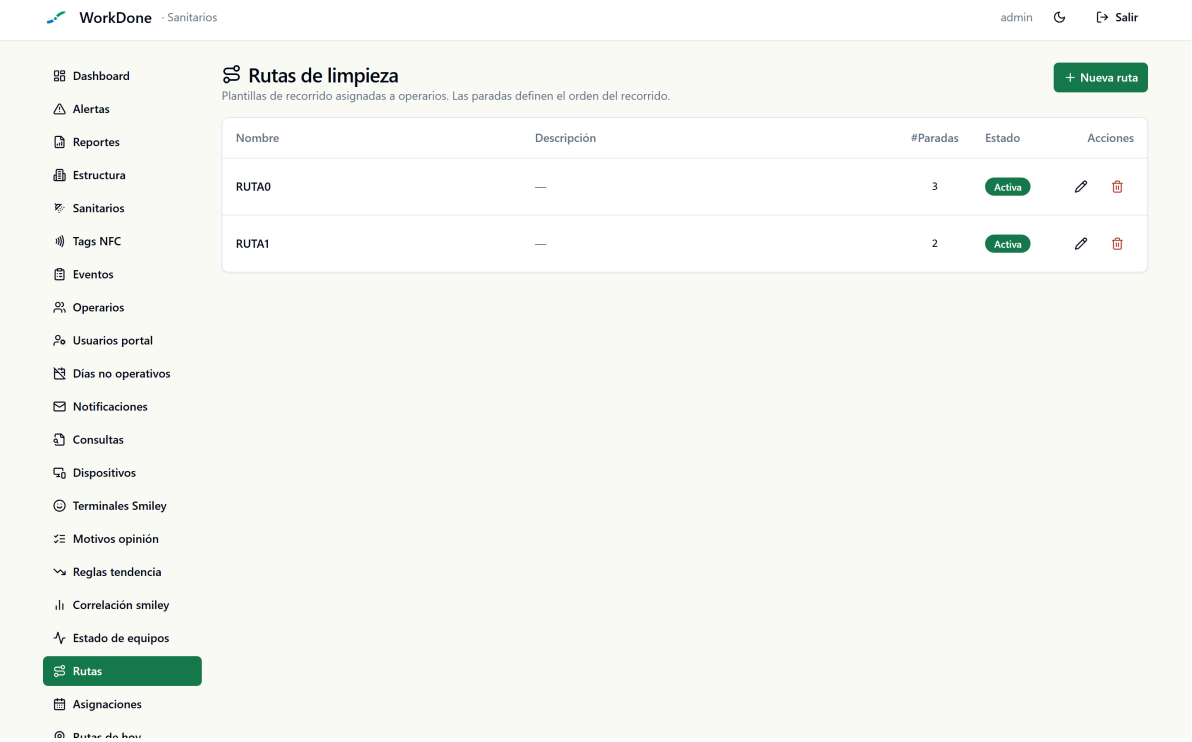
> **Verificación incompleta en las fuentes: el alta de un dispositivo de operario**
>
> Las fuentes de verdad **no documentan un endpoint de alta explícito para el celular de un operador**. Lo que sí está verificado:
>
> - El **CRUD scaffold** `POST /api/dispositivos` existe y acepta `deviceUuid`, `nombre`, `apiKeyHash`, `activo`. **Pero expone/recibe el `apiKeyHash` directamente**, lo cual no parece el camino de provisioning pensado para un celular (habría que precalcular el hash de la api key por fuera). Tratalo con cuidado.
> - El **provisioning verificado** es el de la **terminal Smiley**: `POST /api/v1/smiley/vincular {codigo}` canjea un código de un solo uso y devuelve `device_uuid` + `api_key` en texto plano **una sola vez**. Ese flujo es para **terminales**, no para la app del operario.
> - Para la **operación del BackOffice** sobre dispositivos ya existentes está confirmado: `GET /api/v1/admin/dispositivo` (lista, sin `api_key_hash`), `POST /api/v1/admin/dispositivo/{id}/deshabilitar` y `/habilitar` — que es lo que hacen los botones de la pantalla **Dispositivos**.
>
> **No inventamos un endpoint de alta de dispositivo de operario que no esté en las fuentes.** Si en tu deploy el celular se registra solo en el primer login (auto-provisioning por `X-Device-UUID`) o por otro mecanismo, **confirmalo contra el código del backend** antes de documentarlo como definitivo. Lo que sí podés garantizar como admin desde la pantalla Dispositivos: que el dispositivo termine **activo** para que el sync funcione.

Resultado verificable: en **Dispositivos** el equipo del operario figura como activo y, tras el primer sync, con su `ultimo_sync` poblado.

Paso 10 — Ruta y sus paradas (FK → Sanitario)

Dónde: menú lateral **Rutas** → botón **+ Nueva ruta**.

Una **Ruta** es una plantilla reusable: una lista ordenada de sanitarios (las **paradas**) que define el recorrido. La ruta es una capa de **planificación arriba del tap, nunca un bloqueo**: un tap fuera de ruta no es un error.



La pantalla Rutas de limpieza lista cada plantilla con su nombre, descripción, #Paradas y estado. El botón + Nueva ruta abre el alta donde agregás las paradas (sanitarios) en orden.

Contrato snake_case + paradas anidadas (al ir por API)

Este endpoint **no** es CRUD scaffold: usa un contrato propio con campos en **snake_case** y las paradas embebidas en el body. El `orden` de cada parada es `>= 1` y la combinación `(ruta, sanitario)` es única (no repitas el mismo sanitario dos veces en una ruta).

El orden es sugerido, no obligatorio (R1)

El `orden` marca el "siguiente sugerido" en la app, pero cualquier parada puede hacerse en cualquier orden. La ruta nunca impide un tap.

Bajo el capó (API)

Qué	Detalle
Endpoint	`POST /api/v1/admin/ruta` (exige `ROLE_ADMIN`)
FK que necesita	`sanitario_id` de cada parada → del paso 5
Campos obligatorios	`nombre`, `activo`, `paradas` (array, puede ir vacío)

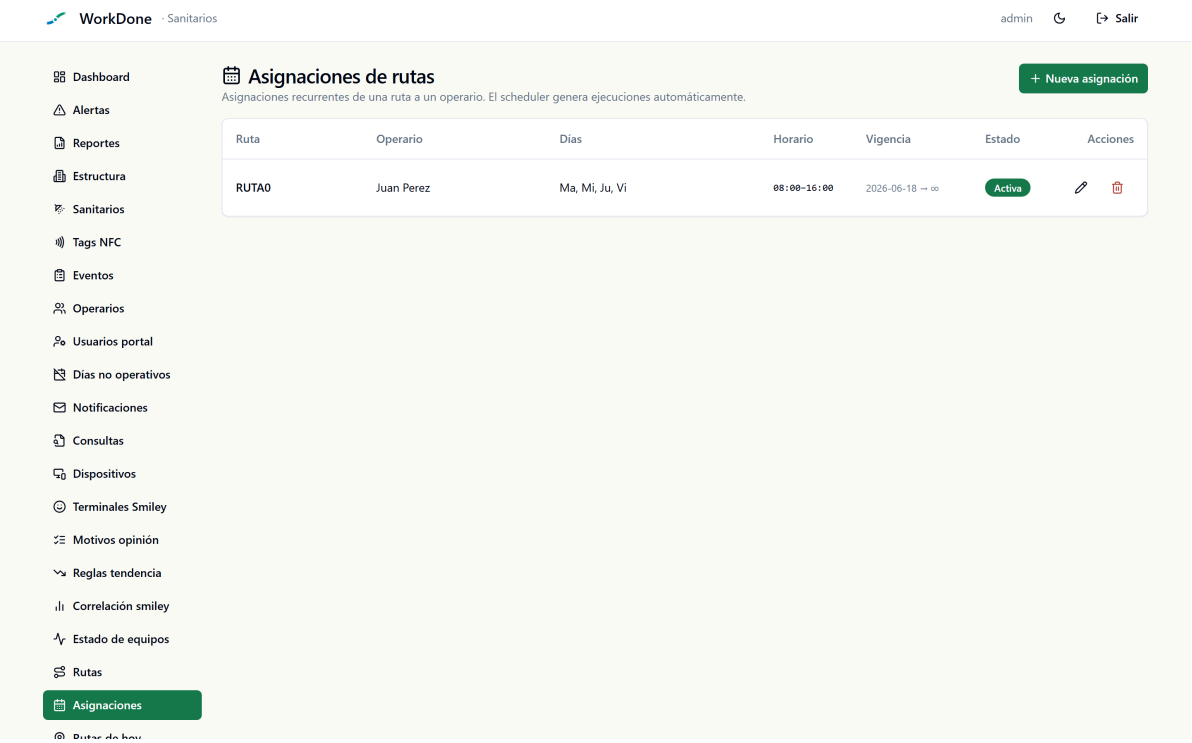
```
{
  "nombre": "Recorrido matutino Terminal A",
  "descripcion": "Limpieza de apertura, planta 1",
  "activo": true,
  "paradas": [
    { "sanitario_id": 1, "orden": 1 },
    { "sanitario_id": 2, "orden": 2 },
    { "sanitario_id": 3, "orden": 3 }
  ]
}
```

Resultado verificable: la ruta aparece en la lista de **Rutas** con su contador de **#Paradas** y estado **Activa**. Su `id` lo usás en el paso 11.

Paso 11 — Asignar el operario a la ruta (FK → Ruta + Operario)

Dónde: menú lateral **Asignaciones** → botón **+ Nueva asignación**. Elegís la ruta (paso 10) y el operario (paso 8).

La **RutaAsignacion** conecta una ruta con un operario de forma **recurrente** (qué días, en qué franja horaria, durante qué vigencia). De acá salen, automáticamente, las ejecuciones de cada día.



La pantalla Asignaciones de rutas muestra cada asignación recurrente: la ruta, el operario, los días activos, el horario (ej. 08:00–16:00), la vigencia y el estado. El subtítulo lo resume: "El scheduler genera ejecuciones automáticamente."

Reglas de la asignación

- `dias_semana` es un patrón de **7 caracteres** `Lun..Dom`; las letras presentes marcan los días activos (ej. `LMXJV--` = lunes a viernes). Patrón mal formado → `400`.
- `vigencia_hasta` puede ser `null` (vigencia indefinida, `∞`); si viene, debe ser `>= vigencia_desde`.
- El **cruce de medianoche** está permitido (`hora_hasta` antes que `hora_desde`).
- Se permiten **N asignaciones por operario** (sin solape exigido). El caso típico es 1.

Bajo el capó (API)

Qué	Detalle
Endpoint	`POST /api/v1/admin/ruta-asignacion` (exige `ROLE_ADMIN`)
FK que necesita	`ruta_id` (paso 10) + `operario_id` (paso 8)
Campos obligatorios	`ruta_id`, `operario_id`, `dias_semana`, `hora_desde`, `hora_hasta`, `vigencia_desde`, `activo`

```
{
  "ruta_id": 1,
  "operario_id": 7,
  "dias_semana": "LMXJV--",
  "hora_desde": "06:00:00",
  "hora_hasta": "14:00:00",
  "vigencia_desde": "2026-06-22",
  "vigencia_hasta": null,
  "activo": true
}
```

Cómo se generan las ejecuciones del día

La asignación es la plantilla recurrente; lo que el operario ve cada día es una **RutaEjecucion** concreta:

- Un **scheduler nocturno** (`@Scheduled`, default `0 0 4 * * *` — las 04:00) recorre las asignaciones activas cuyo día matchea `dias_semana` y que están en vigencia, y por cada una crea la `RutaEjecucion` de hoy (`PENDIENTE`) con sus paradas como **snapshot** de la plantilla. Es idempotente. (Detalle de schedulers en Jobs y alertas.)
- Una **limpieza válida** (operador) o una **supervisión cerrada** (supervisor) sobre un sanitario de la ruta marca esa parada `HECHA` automáticamente — vía evento, el tap no sabe de rutas.
- **Reasignación del día:** si el operario titular falta, movés la ejecución de **hoy** a otro operario desde la pantalla **Rutas de hoy**, sin tocar la asignación recurrente. Bajo el capó:

```
PATCH /api/v1/admin/ruta-ejecucion/{id}/reasignar
Content-Type: application/json

{ "operario_id": 9 }
```

> ⚠️ **La reasignación es solo del día y con estados acotados**
>
> Solo aplica a ejecuciones de **hoy** en estado `PENDIENTE` / `EN_CURSO` (`400` si no es de hoy, `409` si está en otro estado, `404` si el operario o la ejecución no existen). Las paradas ya `HECHA` conservan su trabajo. La gestión de licencias/francos está **fuera de alcance** del módulo.

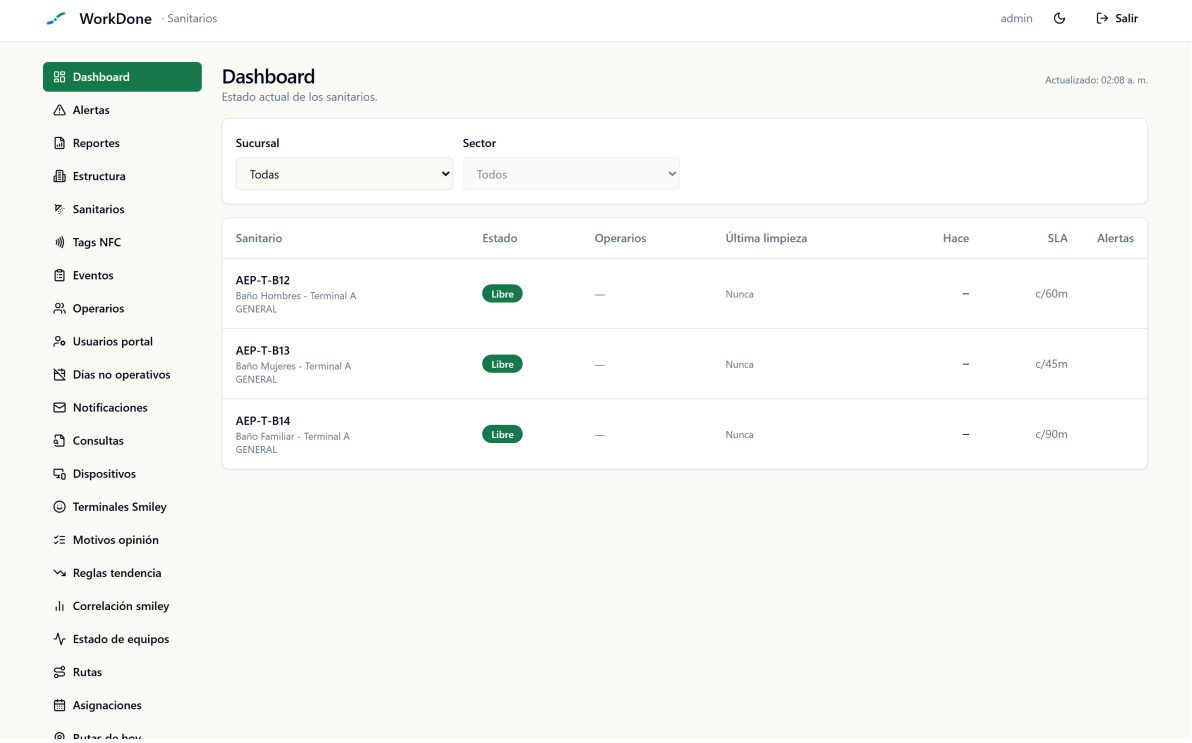
Resultado verificable: la asignación aparece en la lista de **Asignaciones** con sus días, horario y vigencia. Tras correr el scheduler (o el día siguiente), la ejecución del día se ve en **Rutas de hoy** (ver paso 12).

Paso 12 — Verificación final del entorno

Cerrás el círculo confirmando que las dos puntas — operario en campo y administrador en las pantallas — ven lo que tienen que ver.

El Dashboard pinta tus sanitarios

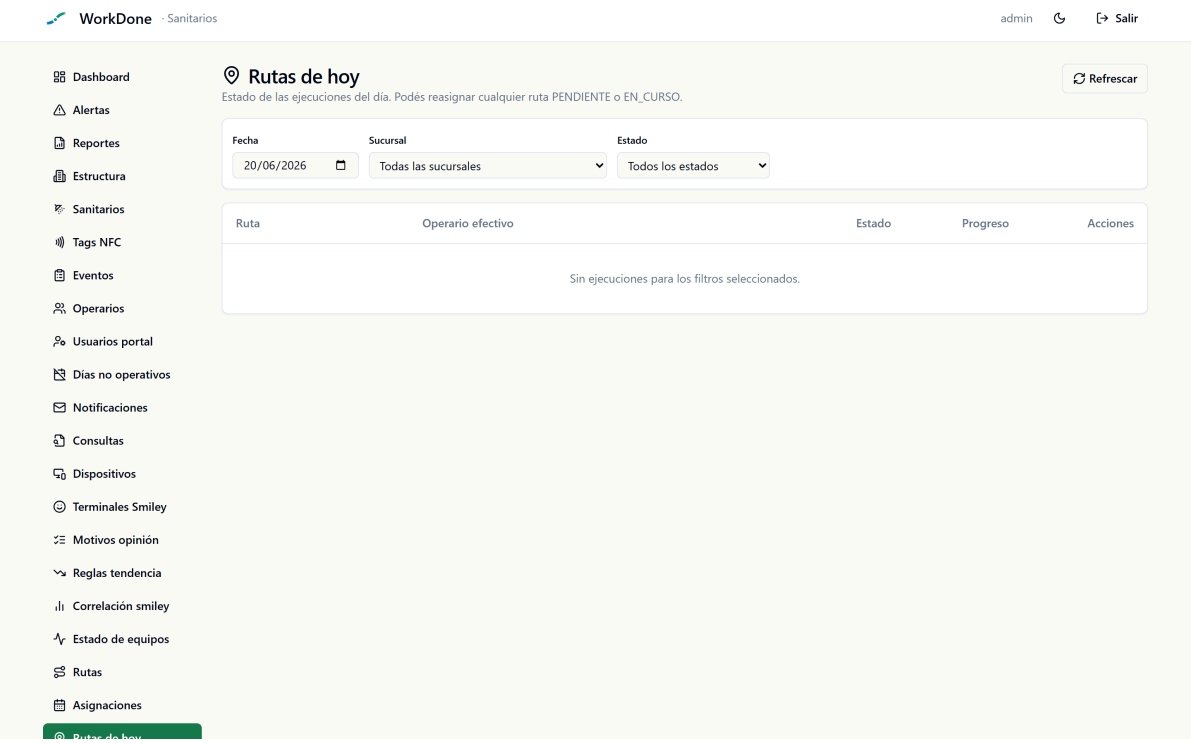
Dónde: menú lateral **Dashboard**. Filtrás por sucursal y sector y ves la grilla en tiempo real.



El Dashboard muestra el estado actual de cada sanitario: estado (`Libre` / `Limpiando` / `SLA vencido` / `Sin datos` / `Sin exigencia`), última limpieza, hace cuánto, el SLA configurado (paso 6) y alertas. Es la vista de "todo en orden".

El tablero de Rutas de hoy

Dónde: menú lateral **Rutas de hoy**. Filtrás por fecha, sucursal y estado, y desde acá reasignás las ejecuciones `PENDIENTE` / `EN_CURSO`.



La pantalla Rutas de hoy lista las ejecuciones generadas por el scheduler para la fecha elegida, con su operario efectivo, estado y progreso (`paradas_hechas/total`). Si todavía no corrió el scheduler para hoy, muestra "Sin ejecuciones para los filtros seleccionados".

Resumen de las verificaciones

Verificación	Cómo
El operario ve "mi ruta de hoy"	En el response de `POST /api/v1/app/sync` aparece el bloque `mi_ruta_hoy` con las ejecuciones de hoy del operario y sus paradas. (Ausente si no tiene ruta hoy.)
El dashboard muestra los sanitarios	Pantalla **Dashboard**. Bajo el capó, `GET /api/v1/admin/dashboard/estado?sucursal_id=&sector_id=`.
El tablero de rutas del día	Pantalla **Rutas de hoy**. Bajo el capó, `GET /api/v1/admin/ruta-ejecucion?fecha=&sucursalId=` lista las ejecuciones con progreso y permite reasignar.

Entorno operativo

Si el sync del operario trae `mi_ruta_hoy` y el Dashboard pinta tus sanitarios, el entorno está montado: jerarquía física, tags, operarios, dispositivo activo y rutas asignadas, todo encadenado por sus FKs.

Ver también

- Registrar y asignar un tag NFC — el detalle del paso 7 (alta de stock + asignación), que se hace por la app móvil.
- Modelo de datos — todas las entidades, columnas y FKs de las que habla esta guía.
- Jobs y alertas — el scheduler que genera las ejecuciones del día y los demás jobs.
- Portal del cliente — cómo el cliente final ve el estado de sus sanitarios una vez que el entorno está operativo.

El portal del cliente: darle acceso y qué ve

Esta guía cubre dos cosas: **(a)** cómo un ADMIN le da acceso a un cliente al portal de transparencia, y **(b)** qué ve ese cliente una vez que entra.

El **cliente** acá no es el operario ni el técnico: es la **empresa contratante** (el aeropuerto, el shopping, el hospital, la planta) que quiere ver, en tiempo real y de forma transparente, el estado del servicio de limpieza sobre sus sanitarios. El portal es **solo lectura** — un ventanal, no una puerta: cero gestión, cero datos internos del personal de la operadora.

> **Fuente de verdad**
>
> El comportamiento descrito acá se valida contra el backend (`V21_PORTAL_CLIENTE.md` , `docs/API.md` y el modelo de datos). Donde el detalle visual del front no esté garantizado por el contrato del API, lo señalamos explícitamente.

El portal NO es el BackOffice

Antes de los pasos, dejemos clara la distinción que gobierna todo el módulo:

	BackOffice (admin)	**Portal del cliente**
Authority	`ROLE_ADMIN`	`ROLE_CLIENTE`
Quién	Personal de la operadora del servicio	La empresa contratante
Prefijo de API	`/api/v1/admin/*` + CRUD `/api/*`	`/api/v1/cliente/*`
Alcance	TODAS las empresas	Solo la(s) empresa(s) con acceso explícito
Capacidades	Lee **y** gestiona (CRUD, alertas, NFC, usuarios)	**Solo lectura**

Datos de personal	Visibles	**Nunca** (nombres de operarios, RRHH)
Layout del front	Sidebar de gestión	`PortalLayout` sobrio, sin sidebar de gestión

> ⚠ **La autorización real es server-side**
>
> El guard por rol en el front React es **UX**, no seguridad. La autorización real vive en `SecurityConfiguration` y en el scoping por empresa de cada endpoint (Fase A/B del backend). Un `ROLE_CLIENTE` recibe **403** en todo `/api/v1/admin/**`, en todo el CRUD scaffold `/api/*` y en `/api/v1/app/**`.

Prerequisitos

- Ser **ADMIN** del BackOffice (`ROLE_ADMIN`). El alta de usuarios cliente **solo** la hace el ADMIN — no hay auto-registro ni invitaciones por mail (decisión P3).
- Tener ya creada(s) la(s) **empresa(s)** que el cliente va a ver (la jerarquía Empresa → Sucursal → Sector → Sanitario debe existir). Para montar todo eso desde cero, ver Montar un entorno desde cero.
- Tener el servicio de mail operativo (en dev sale por log / MailHog): el alta dispara un mail de activación.

Paso 1 — Crear el acceso del cliente

El cliente necesita una cuenta de BackOffice web (`jhi_user`) con la authority `ROLE_CLIENTE` y, sobre todo, el **scoping por empresa**: a qué empresas puede ver.

Desde el BackOffice (ABM)

1. Logueate como ADMIN en el BackOffice (`POST /api/authenticate`, JHipster estándar).
2. Sidebar → **Usuarios portal** (ruta `/usuarios-cliente`).
3. **Crear**: login (ej. `cliente-aa2000`), email, nombre, apellido, y seleccioná **la(s) empresa(s)** con acceso.
4. El usuario recibe un mail con una *reset key* y fija su contraseña por el flujo JHipster estándar. **No se setea password en el alta** (P3).

Activación por mail, no por password

El ABM crea el `User` con `ROLE_CLIENTE` vía `UserService.createUser` (cuenta activada + reset key por mail). El admin **nunca** tipea la contraseña del cliente: el cliente la fija él mismo desde el link del mail.

Equivalente por API

```
POST /api/v1/admin/usuarios-cliente
Authorization: Bearer <JWT de admin>
Content-Type: application/json
```

```
{
  "login": "cliente-aa2000",
  "email": "transparencia@aa2000.com.ar",
  "nombre": "Gerencia",
  "apellido": "Regional",
  "empresaIds": [42]
}
```

El resto del ABM (también `ROLE_ADMIN`):

Método	Ruta	Para qué
`GET`	`/api/v1/admin/usuarios-cliente`	Lista los usuarios cliente con sus empresas
`PUT`	`/api/v1/admin/usuarios-cliente/{login}/empresas`	**Reemplaza** el set de empresas con acceso
`PUT`	`/api/v1/admin/usuarios-cliente/{login}/activado/{bool}`	Habilita / deshabilita la cuenta
`POST`	`/api/v1/admin/usuarios-cliente/{login}/reset-password`	Dispara reset (flujo JHipster, responde 204 siempre)

Multi-empresa: el caso del gerente regional

Un usuario cliente puede tener acceso a **N empresas** (decisión P5). El caso típico es 1, pero un gerente regional con varias plantas se modela como un cliente con varias entradas en `cliente_empresa_acceso`. "Último acceso" se devuelve `null`: JHipster no lo trackea nativo y no se sumó schema en el piloto.

El scoping es la regla de oro

El alcance de datos se deriva y valida **siempre server-side** contra la tabla `cliente_empresa_acceso`. Ningún endpoint del portal confía en un `empresaId` del request sin validar primero que el usuario autenticado tiene acceso a esa empresa.

Cliente de empresa A pidiendo empresa B → 403

Si `cliente-aa2000` solo tiene acceso a la empresa 42 y llama `GET /api/v1/cliente/estado?empresaId=99`, el backend responde **403**. Hay un test de seguridad por endpoint que cubre exactamente este caso (cross-empresa). No es opcional: es criterio de aceptación.