AutoCompraMod

Documentación técnica y de onboarding

Producto	**AutoCompraMod — terminal de autoservicio para retail**
Ecosistema	**Backend Spring Modulith · POS Flutter (TiprePOS) · Cockpit**
Stack	**Java 21 · Spring Boot 3.4 · Spring Modulith · Flutter · Dart · Isar**
Fecha	**Junio 2026**

Bienvenido a AutoCompraMod

Si recién entrás al equipo, esta es tu puerta de entrada. **AutoCompraMod** es el ecosistema de **autoservicio (autopago)** de Tipre para retail: una **terminal** donde el cliente del supermercado escanea sus productos, gestiona envases y vales, paga y se lleva su ticket, sin pasar por una caja con cajero. Detrás de esa terminal hay **un único backend** que orquesta catálogo, pagos, promociones y envases, y un **panel de operación** para monitorear el parque de terminales. 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 supermercado, cada caja con cajero es costo y es cola. El **autoservicio** mueve parte de ese flujo a una terminal donde el propio cliente arma su compra y paga solo. Pero eso solo funciona si la terminal es **rápida, confiable y no se cuelga**: si el catálogo no responde, si el pago queda en un estado indeterminado o si la impresora falla en silencio, el cliente abandona la compra y se genera un problema operativo.

AutoCompraMod ataca eso en dos frentes:

- **La terminal (TiprePOS)** corre el flujo completo de autopago — escaneo, envases/vales, medios de pago (QR, tarjeta, Point Smart), impresión — y está pensada para operar en **desktop** (la terminal física) con reconexión automática ante caídas de red.
- **El backend (AutoCompraMod)** consolida en **un solo deployable** lo que antes eran **5 microservicios** separados. Una terminal ya no tiene que hablar con cinco servicios distintos: habla con **uno**.

> **El cambio de fondo: de 5 microservicios a un monolito modular**
>
> Históricamente el autoservicio eran cinco servicios independientes (`TsTicket`, `TsArticulos`, `TsPagos`, `TsPromos`, `TsEnvases`). AutoCompraMod los unifica en un **monolito modular** con Spring Modulith: un único proceso, pero con **fronteras internas verificadas por el build**. Se gana simplicidad de deploy sin perder el desacople. El detalle está en El monolito modular.

Los componentes del ecosistema

AutoCompraMod no es una sola aplicación, sino **un backend, su panel y una terminal cliente**:

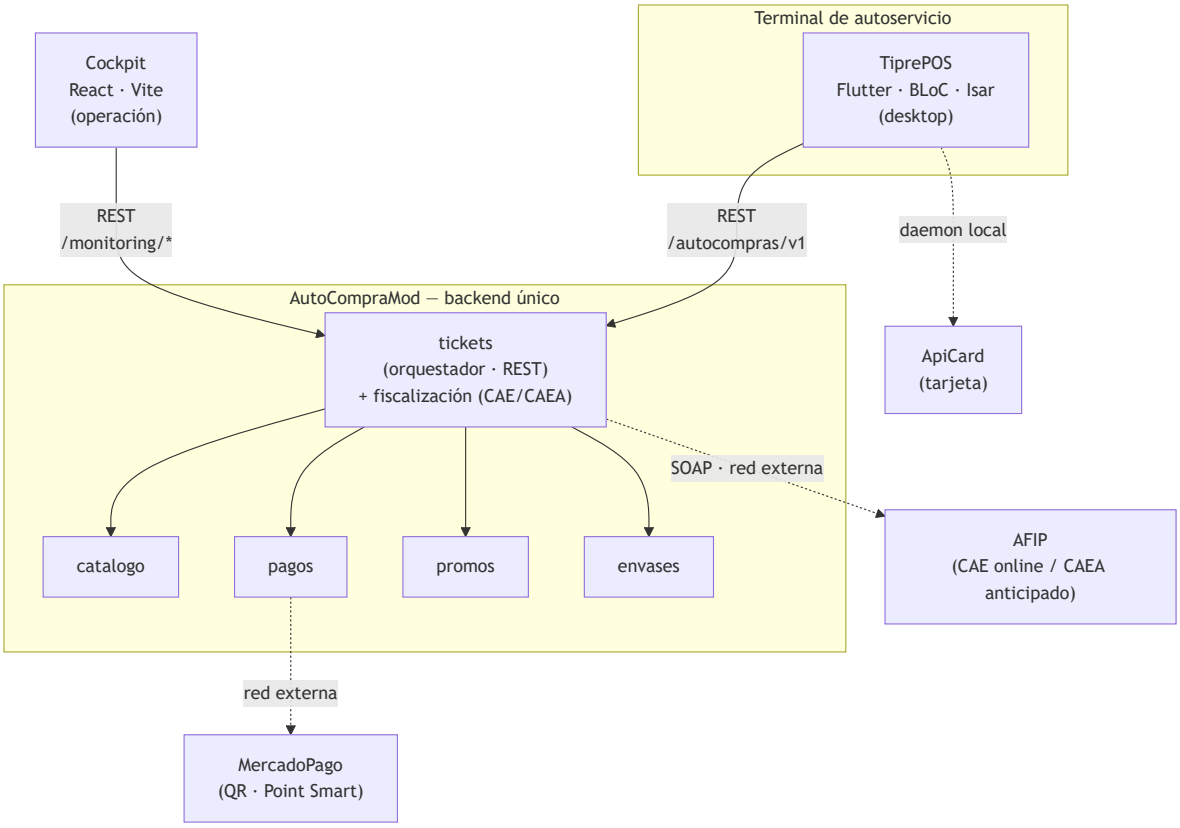
Componente	Repo	Stack	Qué hace
Backend	`AutoCompraMod`	Java 21 · Spring Boot 3.4 · Spring Modulith	El backend único del autoservicio. Orquesta tickets, catálogo, pagos, promociones y envases. Expone una API REST bajo `/autocompras/v1`.
Cockpit	`AutoCompraMod/cockpit-ui`	React · Vite · TypeScript	Panel de operación: monitoreo del parque de terminales, salud de devices, pagos y panel fiscal. Servido por el backend.
POS	`TiprePOS`	Flutter · Dart · BLoC · Isar	La terminal de autoservicio. Escaneo, compra, envases/vales, pagos, impresión. Habla con el backend por **REST** (antes STOMP — ver migración).
StressBench	`pos-swarm`	Node · TypeScript	Herramienta de **load-test E2E**: simula ~200 terminales vendiendo a la vez contra el backend

real, con un gateway de pago falso y un cockpit propio. Ver StressBench.

Por qué repos separados

El backend y la terminal tienen stacks (Java vs. Flutter), ciclos de release y equipos distintos. El backend es la **fuente de verdad de los contratos**: la terminal no inventa endpoints, los consume. El Cockpit, en cambio, vive *dentro* del repo del backend porque comparte su ciclo de vida.

Diagrama de alto nivel del ecosistema



Los bordes externos del sistema

El backend es un solo proceso, pero tiene **tres dependencias externas de red**: `pagos → MercadoPago` (QR y Point Smart), la terminal → **ApiCard** (daemon local de tarjeta) y `tickets → AFIP` para la **fiscalización** (CAE por SOAP cuando AFIP está online; CAEA anticipado/local cuando no). Esos bordes son los que pueden fallar por red y por eso tienen manejo durable. El detalle está en Facturación fiscal y El ciclo de pago y fiscalización.

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 por qué hay un backend único y una terminal cliente. El Monolito modular explica el corazón del backend; la Migración strangler, cómo se llegó hasta acá desde los 5 microservicios. Cuando aparezca un término que no conozcas — *ticket*, *envase*, *promo*, *intent de pago*, *Point Smart* — está definido en el Glosario.
2. **Referencia técnica**. Acá vive el corazón del negocio: cómo se computa un ticket (El agregado Ticket), el núcleo impositivo, las promociones, los envases y vales y la facturación fiscal AFIP. Más los módulos del backend, la API REST y la terminal Flutter. El cómputo del ticket y la facturación fiscal son lo más delicado: leelos enteros antes de tocar nada que los roce.
3. **Empezar a programar**. Recién entonces levantás el backend en local, corrés la terminal contra él y empezás a contribuir. Cada repo tiene su material de setup en Setup del entorno.

Regla de oro del ecosistema

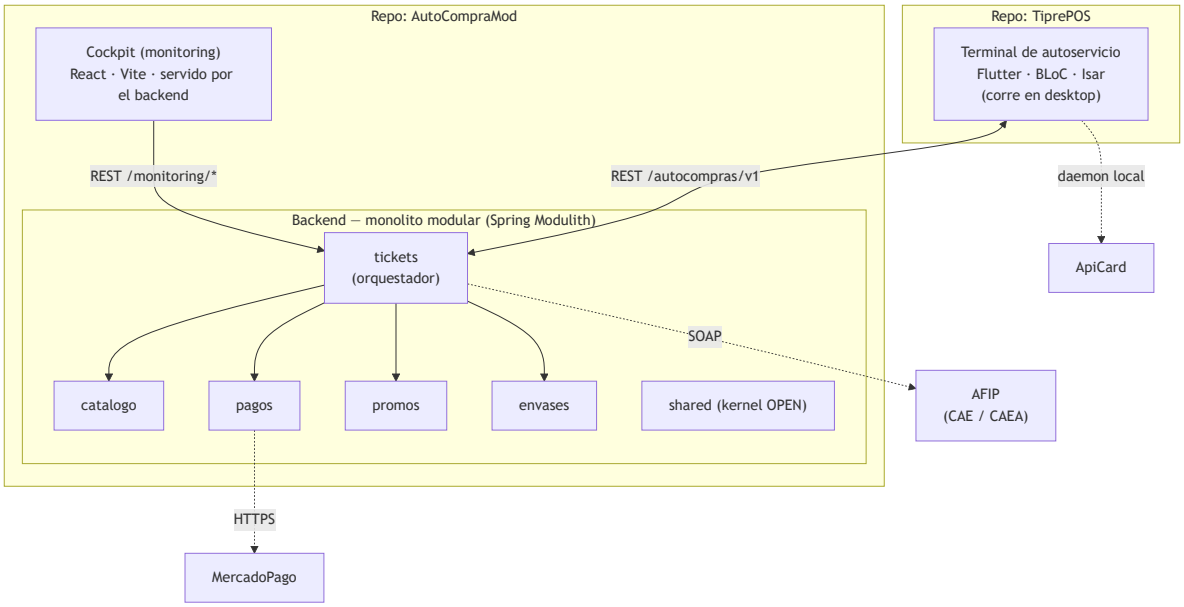
El **backend es la fuente de verdad de los contratos**, y dentro del backend **el guardrail manda**: `ModularityTests.verify()` rompe el build si se viola una frontera entre módulos o aparece un ciclo. Si tu cambio cruza un boundary que no debería, no lo "destrabás" tocando el test: revisás el diseño.

Arquitectura del ecosistema AutoCompraMod

AutoCompraMod no es una aplicación monolítica clásica ni un enjambre de microservicios: es **un monolito modular que oficia de backend único, y una terminal cliente que lo consume**. Esta página explica por qué está dividido así, cómo se comunican las piezas, cómo se autentica cada actor y dónde está la verdad. La idea no es que memorices endpoints — eso vive en la referencia — sino que entiendas el modelo mental: quién manda, qué frontera no se cruza y por qué.

La forma del sistema

El ecosistema tiene **tres piezas**: el backend, su panel de operación (Cockpit, que vive dentro del backend) y la terminal de autoservicio (TiprePOS, en su propio repo).



Componente	Repo	Rol
Backend	`AutoCompraMod`	El sistema único del autoservicio: orquesta tickets, catálogo, pagos, promos y envases. Fuente de verdad de los contratos.

Cockpit	`AutoCompraMod/cockpit-ui`	Panel de operación (observabilidad): parque de terminales, salud, tickets, errores, pagos y fiscal. Servido por el propio backend.
POS	`TiprePOS`	La terminal. Corre el flujo de autopago contra el backend.

Por qué un monolito modular y no microservicios

Acá está la decisión arquitectónica más importante. El autoservicio **eran cinco microservicios** (`TsTicket`, `TsArticulos`, `TsPagos`, `TsPromos`, `TsEnvases`). AutoCompraMod los consolida en **un solo deployable** con Spring Modulith.

¿Por qué? Porque cinco servicios traían el costo de los microservicios (cinco deploys, cinco conexiones que la terminal tenía que orquestar, latencia de red entre servicios, fallos parciales) **sin el beneficio**: las cargas son chicas y los cinco servicios escalaban juntos de todos modos. Un monolito modular da lo mejor de los dos mundos:

- **Un solo proceso, un solo deploy.** La terminal habla con **un backend**, no con cinco.
- **Fronteras internas reales, verificadas por el build.** Cada módulo (`tickets`, `catalogo`, …) tiene una API pública y todo lo demás es interno. Spring Modulith **rompe el build** si un módulo accede a los internals de otro o si aparece un ciclo de dependencias.

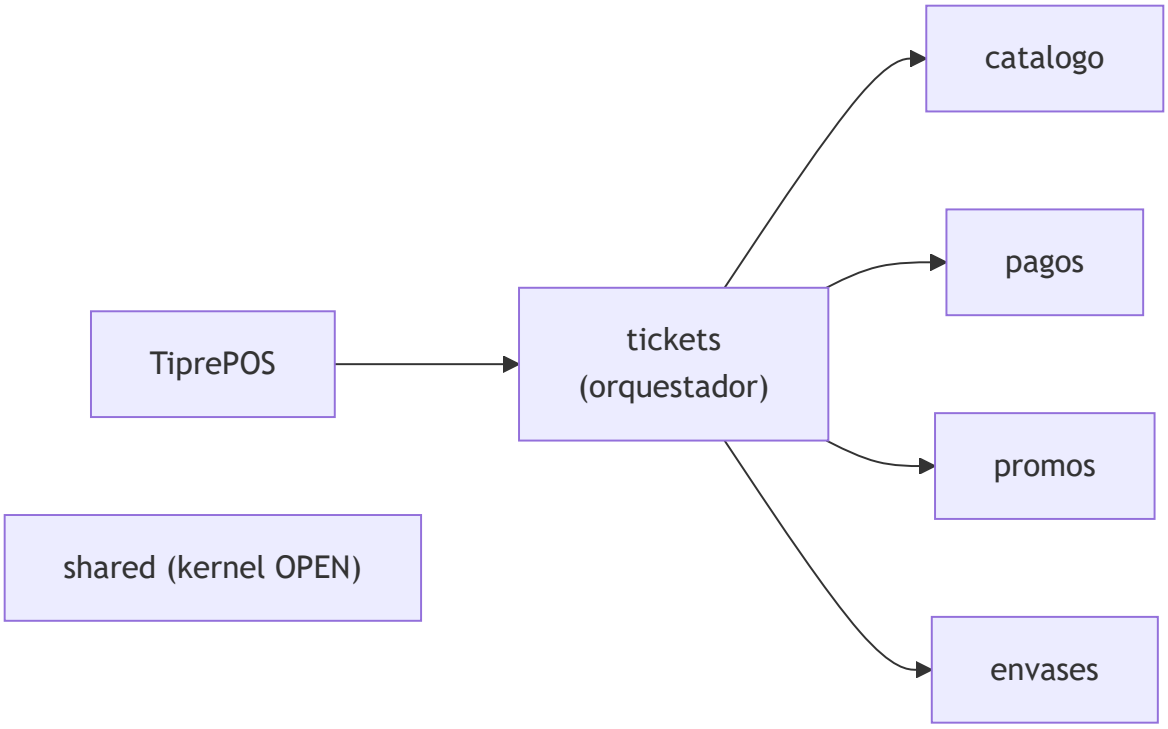
El detalle de cómo funciona ese guardrail está en El monolito modular. Cómo se llegó hasta acá desde los cinco servicios, en Migración strangler.

> **El backend manda sobre los contratos**
>
> La terminal **no inventa endpoints ni DTOs**: los consume. Si la terminal tiene una copia de un contrato y no coincide con el backend real, **gana el backend**: se reporta, no se "arregla" en el cliente.

Topología interna: estrella acíclica

Dentro del backend, los módulos no se llaman entre sí libremente. La topología es una **estrella acíclica**: sólo `tickets` (el orquestador, la API que ve el POS) llama a los demás; `catalogo`,

`pagos`, `promos` y `envases` son **hojas** que no llaman a nadie. Cero ciclos.



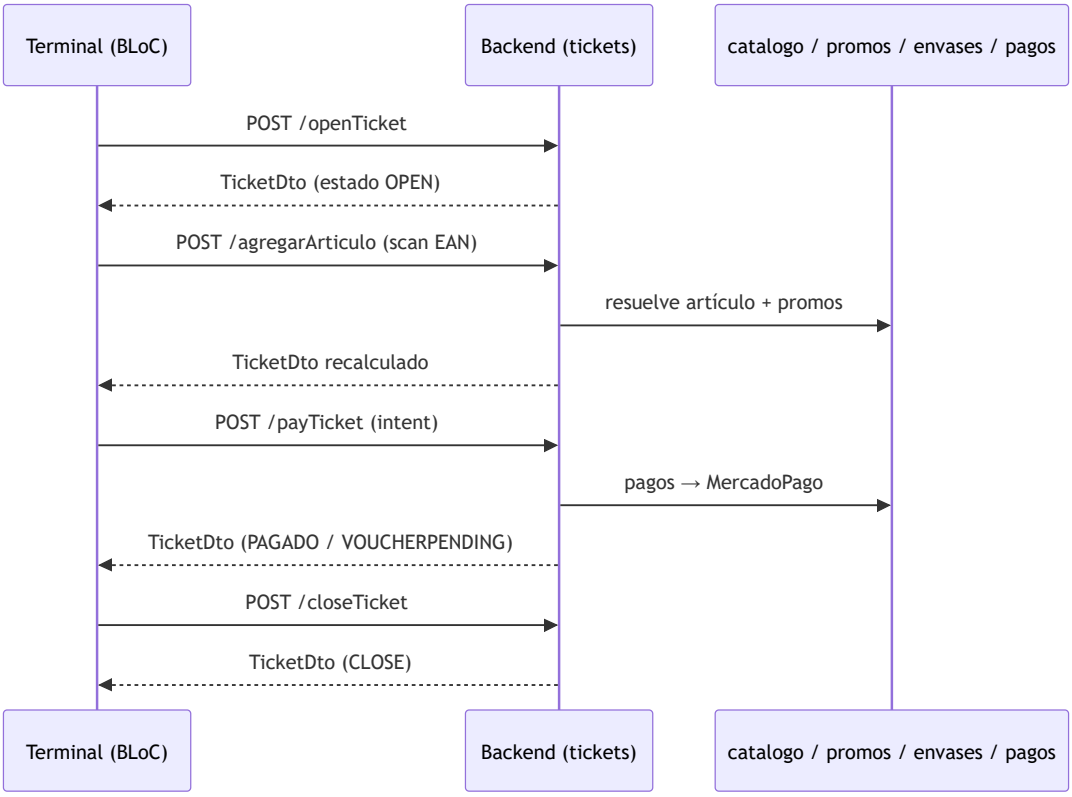
`shared` es un **kernel abierto** (`ResponseMessage`, enums de pago, `NucleoImpositivoDto`) que cualquier módulo puede usar: es la lengua común, no un módulo de negocio.

> ⚠ **Reglas no negociables**
>
> 1. **Comunicación = sólo la API pública del otro módulo.** Nunca los internals. Para desacoplar más → eventos de aplicación (`@ApplicationModuleListener`).
> 2. **Datos: schema-por-módulo, CERO foreign keys cruzadas.** Si `pagos` necesita un ticket, lo pide por la API de `tickets`, no con un JOIN.
> 3. **Los bordes externos siguen siendo red.** `pagos → MercadoPago` y `tickets → AFIP` (CAE por SOAP) se mantienen async/durable porque son red de verdad: ahí viven la reconciliación de pagos y el `VOUCHERPENDING` fiscal.
> 4. **El guardrail manda.** `ModularityTests.verify()` rompe el build si se viola un boundary.

Cómo se comunican: la terminal y el backend

El backend expone una **API REST** bajo el context-path `/autocompras/v1`. La terminal hace mutaciones de ticket (`openTicket`, `agregarArticulo`, `removeItem`, `closeTicket`,

`changeStatus`, `validateTicket`, `payTicket`) por POST, y consulta el estado de los servicios (`/status`) con un **polling adaptativo**: cada 60 s cuando el backend está sano, cada 15 s cuando está degradado.



Antes era STOMP, hoy es REST

La terminal **se comunicaba por STOMP/WebSocket** y migró a REST puro (cutover de full-REST, documentado en Comunicación POS ↔ Backend). En el backend actual **no hay `@MessageMapping` ni broker STOMP**; quedan referencias muertas como `TerminalSessionService` que son sólo histórico. Si ves STOMP mencionado en el README viejo del POS, está desactualizado: la realidad del código es REST.

Autenticación a alto nivel

La seguridad es **app-wide**: un solo `SecurityFilterChain` y un `JwtAuthConverter` (paquete `security`), no uno por módulo. El modelo es deliberadamente liviano: se gatean **sólo las mutaciones de administración**, mientras que el runtime del POS queda accesible para no romper el checkout.

- `app.security.enabled` (default **`false`**): con `false` es `anyRequest().permitAll()`; con `true` aplica la política de roles (resource server JWT / Keycloak).
- `app.security.admin-role` (default **`ADMIN`**): rol exigido para `POST /promociones/update` y `POST /cache/refresh` (espera la authority `ROLE_<admin-role>`).

> ⚠ **El riesgo del toggle**
>
> Con `app.security.enabled=false`, **TODOS** los endpoints quedan `permitAll`, incluidas las mutaciones admin (precios/promos, refresh de cache). Es sólo para dev / parallel-run, y al armar el filter chain con el toggle en `false` se loguea un `WARN` recordándolo. En el ambiente con Keycloak: `enabled=true` y `admin-role` al rol real del realm.

Identidad de la terminal

Cada terminal se identifica en cada request con tres headers que inyecta su cliente HTTP:

- `X-Cod-Terminal` — el código de la terminal.
- `X-Terminal-Uuid` — un UUID por device.
- `X-Device-Health` — la última salud conocida de los dispositivos de pago (pinpad / Point), que alimenta la observabilidad por terminal del Cockpit.

El backend valida ese UUID contra la tabla `pos` en las mutaciones. Así el Cockpit puede mostrar el **parque de terminales** reconciliando las configuradas con las que efectivamente pingearon.

Deploy

El backend corre como un JAR de Spring Boot. El Cockpit (React/Vite) **no es un deploy aparte**: se construye con el `frontend-maven-plugin` durante el build de Maven y se copia a `src/main/resources/static/cockpit`, de modo que el propio backend lo sirve como SPA. Es una topología deliberadamente simple: las cargas del autoservicio son chicas y no justifican infraestructura distribuida.

Para entender las entidades que viajan en estos flujos — `ticket` / `Trx`, `articulo`, `vale`, `envase`, `pos` — pasá por el Glosario.

Glosario del dominio

Los términos que aparecen una y otra vez en el código y en esta documentación. Si entrás nuevo, leé esto antes de la referencia técnica: muchas confusiones son sólo vocabulario.

Conceptos del negocio

Término	Qué es
Autoservicio / autopago	El modelo donde el propio cliente del comercio escanea, paga y se lleva su ticket, sin cajero. Es lo que opera la terminal.
Terminal	La máquina física de autoservicio (corre TiprePOS en desktop). Tiene un `codTerminal` y un `uuid` propios. También se le dice **POS**.
Ticket	La compra en curso: el carrito con sus ítems, totales, impuestos y estado. En el backend se persiste como `Trx`.
Artículo	Un producto del catálogo, identificado por **EAN** (código de barras) y sucursal. Tiene precio, impuesto, si es pesable, si es envase, etc.
EAN	El código de barras estándar del producto. La terminal escanea un EAN y el backend resuelve el artículo.
Promo / promoción	Un descuento o beneficio que el módulo `promos` calcula sobre el ticket. Ver Promociones.

Envase	Producto retornable (botella, cajón) con depósito. Se gestiona en el módulo `envases`. Ver Envases y vales.
Vale	Comprobante de un envase devuelto (o crédito retornable), con `nroVale` y `codVale`. Auditado con Hibernate Envers.
Itemización	La separación entre **ítems** (lo que el cliente ve) y **movimientos** (las anotaciones contables que componen el total). Ver El agregado Ticket.
Núcleo impositivo	Un monto **con su descomposición fiscal adentro** (`NucleoImpositivoDto`, en `shared`): neto, IVA, impuestos internos. No es un número suelto. Ver Núcleo impositivo.
EAN pesable / random-price	Código de barras que trae el **peso** o el **precio** embebidos (productos de balanza). Ver Decodificación de EAN.
DUN	Código de venta por **bulto** (más de 13 dígitos).

Estados del ticket (`Trx`)

El backend modela el ciclo de vida del ticket con estos estados:

Estado	Significa
`OPEN`	Ticket abierto, el cliente está agregando ítems.
`PAGADO`	El pago se aprobó.

`VOUCHERPENDING`	Pagado pero el comprobante fiscal todavía no se emitió/confirmó. Estado de borde con la fiscalización.
`CLOSE`	Ticket cerrado y completo.
`CANCELED_USER`	Cancelado por el cliente.
`CANCELED_INACTIVITY`	Cancelado automáticamente por inactividad (la terminal tiene timers).
`ERROR`	Terminó en error.

Medios y procesadores de pago

Término	Qué es
QR (MercadoPago)	Pago donde la terminal muestra un QR que el cliente escanea con su app. Timeout en la terminal: ~210 s.
Point Smart	Terminal física de MercadoPago (pinpad) donde el cliente inserta/apoya la tarjeta. Timeout: ~90 s. Tiene su propia máquina de estados de orden.
ApiCard	Integración de pago con tarjeta vía un **daemon local** (por defecto `localhost:50001`), no por el backend.
Intent (intención de pago)	El primer paso del pago: se crea la intención (`payment-intent` / `createIntent`) antes de ejecutar/cobrar.
PaymentAttempt	Un intento de cobro. El `paymentAttemptId` se genera una vez por intención y se reusa en

	reintentos para **evitar el doble cobro** (idempotencia).

Estados de orden de Point Smart

Estado	Significa
`created`	Orden creada, la terminal física todavía no la tomó.
`at_terminal`	La terminal tiene la orden, esperando que el comprador interactúe.
`processed`	Aprobada y acreditada.
`failed`	Rechazada.
`action_required`	Incierto (~40 s): hay que revisar la terminal.
`expired`	Pasaron más de ~15 min sin completarse.
`canceled`	Cancelada por la API o la terminal.
`refunded`	Reembolsada.

Tipos de promoción

Término	Qué es
COMBO	Promo que exige comprar productos de varios grupos (ej. "2 de A + 1 de B").
CANTIDAD	Promo que se activa por cantidad mínima de unidades (ej. "llevando 3").

MAYORISTA	Escala de precio por volumen de un EAN.
BULTO	Venta especial por bulto (identificado por DUN).
Acumulativa	Si una promo puede apilarse con otras sobre la misma línea (`SI`) o si gana solo la mejor (`NO`).
MODO ITEMS / MODO PAGO (MDP)	Si la promo se calcula al armar el carrito (ITEMS) o al elegir medio de pago (MDP).

Facturación fiscal (AFIP)

Término	Qué es
Comprobante	El documento fiscal: Factura A/B/C, nota de crédito o débito. Ver Facturación fiscal.
CAE	Código de Autorización Electrónico que AFIP otorga **por comprobante**, online, por SOAP.
CAEA	Código de Autorización Electrónico **Anticipado**: AFIP lo otorga por adelantado para una quincena. Permite facturar con AFIP caído.
Condición IVA	La categoría fiscal del comprador (Responsable Inscripto, Consumidor Final, Monotributo…). Determina la letra del comprobante.
Punto de venta (`nroPVFiscal`)	El número de punto de venta fiscal del POS ante AFIP.
Reconciliador	El job (`PaymentReconciliador`) que resuelve los pagos `INDETERMINADO` sin depender del

	POS. Ver El ciclo de pago y fiscalización.
Indeterminado	El estado de un pago cuando la red cortó entre "cobré" y "recibí la confirmación": no se sabe si se cobró.
REQUIERE_REVISION	Un pago que agotó los reintentos del reconciliador y necesita intervención humana.

Conceptos técnicos

Término	Qué es
Monolito modular	Un solo deployable dividido internamente en módulos con fronteras verificadas. Acá: Spring Modulith.
Spring Modulith	El framework que define los módulos (`@ApplicationModule`) y verifica los boundaries en el build.
Módulo	Una unidad del backend (`tickets`, `catalogo`, `pagos`, `promos`, `envases`, `shared`) con API pública e internals privados.
Guardrail	El test `ModularityTests.verify()` que rompe el build si se viola una frontera o aparece un ciclo.
Kernel OPEN	Un módulo (`shared`) marcado como abierto: cualquiera puede depender de él.
Estrella acíclica	La topología interna: sólo `tickets` llama, el resto son hojas, sin ciclos.
Strangler (migración)	Estrategia de migrar de a un módulo por vez, manteniendo lo viejo hasta el corte.

Schema-por-módulo	Cada módulo tiene su propio schema/DataSource; nada de foreign keys cruzadas.
BLoC	El patrón de manejo de estado de la terminal Flutter (`flutter_bloc`): eventos → estados.
Isar	La base de datos local de la terminal (`auto_compra_db`), donde se persiste la configuración.
Cockpit / monitoring	El panel de operación y el módulo backend (`monitoring`) que lo alimenta.
Heartbeat	El ping periódico de cada terminal que el backend usa para saber qué POS están online (ventana ~90 s).
Parque de POS	El conjunto de terminales: las configuradas en la tabla `pos` ∪ las que pingearon.

El monolito modular (Spring Modulith)

Esta es la decisión de diseño que define al backend. Vale la pena entenderla bien, porque condiciona **dónde escribís cada cosa** y **qué rompe el build**.

El problema que resuelve

Un monolito clásico tiende al "gran bollo de barro": todo accede a todo, y con el tiempo nadie sabe qué depende de qué. Los microservicios resuelven eso con fronteras de red duras, pero **a un costo alto**: múltiples deploys, latencia, fallos parciales, y la complejidad de coordinar contratos entre procesos.

AutoCompraMod elige un punto intermedio: **un solo proceso, pero con fronteras internas tan reales como las de un microservicio** — sólo que verificadas por el compilador y los tests, no por la red. Eso es Spring Modulith.

Cómo se define un módulo

Cada módulo es un subpaquete directo de `com.tipre.autocompras` con un `package-info.java` que lo declara:

```
com.tipre.autocompras
├── tickets/      ← package-info.java con @ApplicationModule
├── catalogo/     ← package-info.java
├── pagos/
├── promos/
├── envases/
├── shared/       ← @ApplicationModule(type = OPEN)  (kernel)
├── security/
└── monitoring/   ← el Cockpit
```

La regla de oro de Spring Modulith: **lo que está en la raíz del paquete del módulo es API pública; lo que está en subpaquetes es interno**. Un módulo puede llamar la API pública de otro, pero **no** sus internals.

Módulo	Rol	API pública (ejemplos)	Llama a

`tickets`	Orquestador. La API que ve el POS.	`TicketRestController`, `TicketService`, `VoucherService`	catalogo, pagos, promos, envases
`catalogo`	Artículos por EAN/sucursal + búsqueda. Cache Caffeine (~100k).	`ArticuloRestController`, `ArticuloService`	— (hoja)
`pagos`	intent/pay/check/cancel. Stateless.	`PagoRestController`, `PagoService`	— (hoja, sale a gateways)
`promos`	Calcula promociones del ticket.	`PromoController`, `PromoService`	— (hoja)
`envases`	Vales y depósitos retornables.	`ValeController`, `ValeService`, `EnvaseService`	— (hoja)
`shared`	Kernel **OPEN**: lengua común.	`ResponseMessage`, enums de pago, `NucleoImpositivoDto`	—
`monitoring`	Cockpit / observabilidad. Lee las APIs públicas.	`MetricsController`, `TerminalsController`, …	— (hoja)

Las reglas no negociables

1. **Comunicación = sólo la API pública del otro módulo.** Nunca internals. Si necesitás más desacople, usá **eventos de aplicación** (`@ApplicationModuleListener`) en lugar de una llamada directa.
2. **Datos: schema-por-módulo, CERO foreign keys cruzadas.** Cada módulo con DataSource propio tiene su schema. Si `pagos` necesita un ticket, lo pide por la API de `tickets`, no con un

JOIN entre schemas.

3. **Los bordes externos siguen siendo red.** `pagos → MercadoPago` es red de verdad: ahí se mantienen la reconciliación durable y el manejo de estados indeterminados. No todo es una llamada en proceso.
4. **El guardrail manda.** Lo que decide si una frontera está bien no es el code review: es el test.

El guardrail: `ModularityTests.verify()`

Spring Modulith ofrece un test que **analiza el grafo de dependencias entre módulos** y falla si:

- un módulo accede a los internals de otro,
- aparece un **ciclo** de dependencias entre módulos,
- un módulo depende de otro que no declaró.

```
// El guardrail, conceptualmente
ApplicationModules.of(AutoComprasApplication.class).verify();
```

Se corre con la suite normal:

```
mvn test
```

> ⚠ **Si el guardrail falla, NO lo destrabás tocando el test**
>
> Un fallo de `verify()` significa que tu cambio **cruzó una frontera que no debía** o **introdujo un ciclo**. La respuesta correcta es revisar el diseño: ¿de verdad `pagos` tiene que conocer a `tickets`? Casi siempre la solución es invertir la dependencia (que `tickets` orqueste) o comunicar por evento. Cambiar el test para que pase es esconder el problema, no resolverlo.

Por qué esta forma y no otra

- **Un solo deploy.** La terminal habla con un backend. Operar uno es más simple que operar cinco.
- **Refactor seguro.** Como las fronteras son explícitas y verificadas, mover lógica entre módulos es un cambio acotado y el build te avisa si rompés algo.
- **Camino abierto a extraer un servicio si algún día hace falta.** Si un módulo necesitara escalar aparte, ya tiene su API y su schema: extraerlo es mucho más barato que partir un bollo de

barro.

Para entender cómo se llegó a esta estructura desde los cinco microservicios originales, seguí con Migración strangler.

Migración strangler

AutoCompraMod no nació de cero: es el resultado de **consolidar cinco microservicios** del autoservicio en un solo backend modular. Esta página cuenta cómo se hizo esa migración sin un "big bang", y por qué importa para entender el estado del repo.

De dónde venimos

El autoservicio eran cinco servicios independientes:

Microservicio original	Se convirtió en el módulo	Rol
`TsTicket`	`tickets`	Orquestador: la API que ve el POS.
`TsArticulos`	`catalogo`	Artículos por EAN/sucursal.
`TsPagos`	`pagos`	intent/pay/check/cancel.
`TsPromos`	`promos`	Cálculo de promociones.
`TsEnvases`	`envases`	Vales y depósitos retornables.

Cada uno era un deploy, una base, una conexión que la terminal tenía que orquestar.

El patrón strangler

La migración siguió el patrón **strangler fig** (la "higuera estranguladora"): en vez de reescribir todo de golpe, se va **migrando de a un módulo por vez**, manteniendo lo viejo en producción hasta que lo nuevo lo reemplaza por completo. Lo nuevo "estrangula" gradualmente a lo viejo.

La secuencia fue:

1. `promos` **como piloto.** El más simple y autocontenido: se migró primero para validar el enfoque (estructura de módulo + guardrail) con bajo riesgo.
2. `catalogo`, `envases`, `pagos`: uno por uno.
3. `tickets` **como host final**: el orquestador, lo último, porque es el que ata todo.

En cada paso se corría `verify()`**.** Si la migración de un módulo introducía un acceso indebido o un ciclo, el build lo cazaba en el acto. Eso es lo que hizo segura la migración incremental: la red de seguridad no era la confianza, era el guardrail.

> **Los repos viejos quedaron como referencia**
>
> Los cinco repositorios originales (`TsTicket`, `TsArticulos`, …) se mantuvieron intactos como referencia hasta el corte. La idea es poder comparar comportamiento contra el canónico mientras la migración se estabiliza.

La otra migración: STOMP → REST en la terminal

En paralelo a la consolidación del backend, **la terminal cambió cómo habla con él**. Originalmente TiprePOS se comunicaba por **STOMP/WebSocket**; migró a **REST puro** bajo `/autocompras/v1`.

Ese cutover está documentado en el propio repo de la terminal (`MIGRACION_BLOC_REST.md`) y se hizo por fases:

Operación (antes, destino STOMP)	Ahora (REST)
`openTicket`	`POST /openTicket`
`agregarArticulo`	`POST /agregarArticulo`
`removeItem`	`POST /removeItem`
`closeTicket`	`POST /closeTicket`

`changeStatusTicket`	`POST /changeStatus`
`validateTicket`	`POST /validateTicket`
`payTicket`	`POST /payTicket`
Suscripción a `/topic/status`	Polling adaptativo de `/status` (60 s / 15 s)

> ⚠ **Consecuencia: STOMP está muerto en el código actual**
>
> En el backend de hoy **no hay `@MessageMapping` ni broker STOMP configurado**. Quedan vestigios como `TerminalSessionService` que son sólo histórico y no se usan. En la terminal, las referencias a STOMP en enums y comentarios están en deprecación (la Fase 3 del plan es eliminar `StompService` y la dependencia `stomp_dart_client`). **Si leés STOMP en el README viejo del POS, ignoralo: la realidad del código es REST.** El detalle vive en Comunicación POS ↔ Backend.

Estado actual

- Backend consolidado en el monolito modular, con los cinco módulos migrados y el kernel `shared`.
- Guardrail (`ModularityTests`) activo y en CI (GitHub Actions corre `mvn test` con JDK 21 en cada push/PR).
- Terminal migrada a REST; limpieza final de STOMP pendiente.
- Cockpit (`monitoring`) construido sobre las APIs públicas de los módulos, sin acoplarse a sus internals.

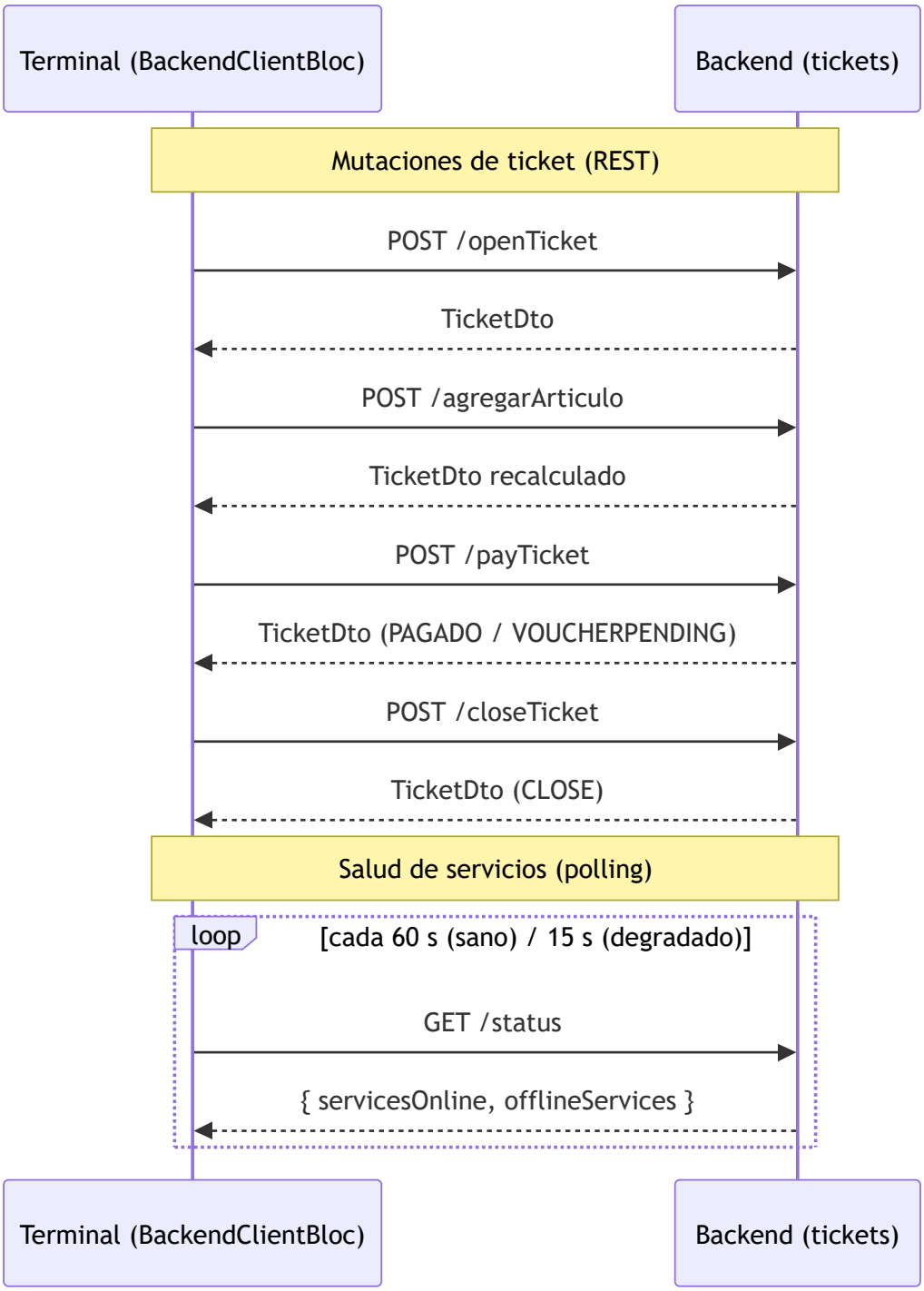
Para el detalle técnico de cada módulo migrado, seguí en la Referencia técnica → Los módulos.

Comunicación POS ↔ Backend

Cómo se hablan la terminal y el backend hoy, qué cambió respecto del pasado, y qué patrones de resiliencia hacen que el checkout no se rompa cuando la red tiembla.

El canal: REST sobre `/autocompras/v1`

La terminal habla con el backend por **HTTP REST**. Todas las mutaciones del ticket son `POST` bajo el context-path `/autocompras/v1`. No hay WebSocket ni STOMP en el flujo actual.



Quién orquesta esto en la terminal

El `BackendClientBloc` (en `lib/viewmodels/backend_client/`) es el centro de la comunicación:

- **Mutaciones**: el evento `SendMessage(destination, request, …)` despacha cada operación de ticket al `TicketService`, que hace el `POST` correspondiente.
- **Salud**: un `Timer` adaptativo dispara `CheckStatusServices`, que consulta `/status` vía `StatusRepository`. Si los servicios están sanos, el timer se reprograma a **60 s**; si están degradados, baja a **15 s** para detectar la recuperación más rápido.

Identidad de la terminal en cada request

El `HttpCliente` (en `lib/services/http/http_cliente.dart`) inyecta automáticamente tres headers en cada llamada:

Header	Para qué
`X-Cod-Terminal`	Identifica la terminal.
`X-Terminal-Uuid`	UUID del device; el backend lo valida contra la tabla `pos`.
`X-Device-Health`	Última salud de los dispositivos de pago (pinpad/Point), insumo del Cockpit.

Resiliencia: por qué no se rompe el checkout

La terminal opera en un comercio real, con red que se cae. Los mecanismos que la sostienen:

- **Polling adaptativo de salud.** La terminal siempre sabe si el backend está sano o degradado, y reacciona (60 s ↔ 15 s).
- **`ConnectionRecoveryMixin`** (en las pantallas de pago QR y Point Smart): maneja timeout + reintento automático de la consulta de transacción, y navega al resultado correcto sin dejar la pantalla colgada.
- **Idempotencia de pago (`paymentAttemptId`).** Se genera **una vez** por intención de pago y se reusa en los reintentos del mismo intento. Así, si la red corta entre "cobré" y "recibí la confirmación", reintentar **no genera un doble cobro**: el backend deduplica.
- **Recuperación de pagos in-flight.** Al arrancar, la terminal revisa la entidad `IsarPendingPayment` (persistida en Isar) para recuperar pagos que quedaron sin resolver en la sesión anterior. Un corte de luz en medio de un cobro no deja la plata en el limbo.

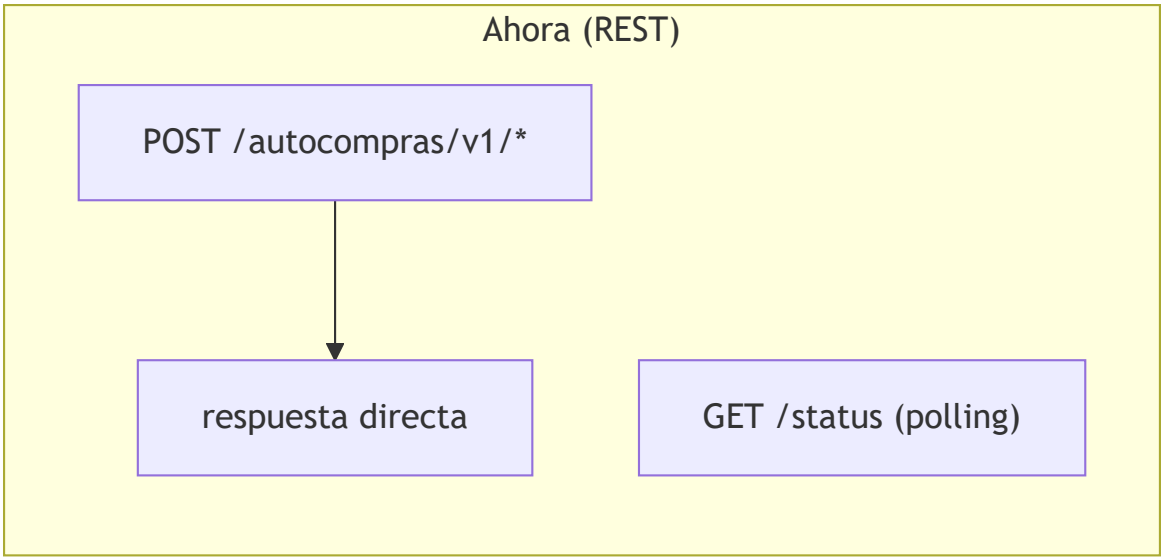
- **Manejo de errores de transporte.** El `HttpCliente` traduce `SocketException` (sin conexión), `TimeoutException` y `HandshakeException` (mismatch http/https) a fallas de dominio claras, en vez de propagar excepciones crudas.

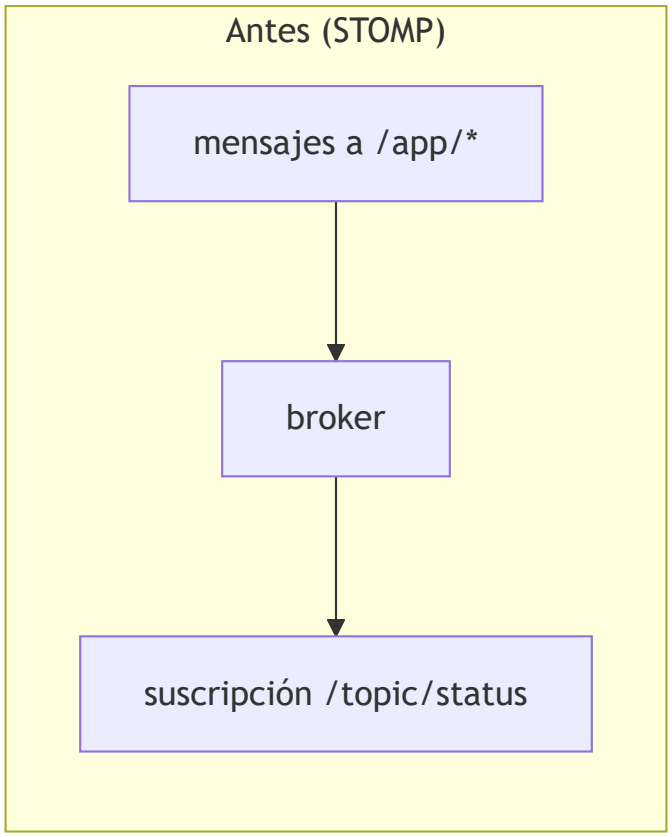
> **El detalle más delicado está en los pagos**
>
> La idempotencia y la máquina de estados de Point Smart (`created → at_terminal → processed/failed/action_required/expired/...`) son lo más fácil de romper. Antes de tocar cualquier flujo de cobro, leé Referencia → Pagos entero.

Lo que cambió: de STOMP a REST

Históricamente la terminal usaba **STOMP/WebSocket** (un broker, suscripciones a tópicos como `/topic/status`, mutaciones por mensajes). Ese modelo se reemplazó por REST por la fase del cutover descrita en `MIGRACION_BLOC_REST.md`:

- Las mutaciones pasaron de mensajes STOMP a `POST` REST (uno por operación).
- El estado de servicios pasó de una **suscripción** push a un **polling** adaptativo.


Ahora (REST)
POST /autocompras/v1/*
respuesta directa
GET /status (polling)


Antes (STOMP)
mensajes a /app/*
broker
suscripción /topic/status

Vestigios que vas a encontrar

- **Backend**: `TerminalSessionService` existe pero está muerto; no hay `@MessageMapping` ni config de broker.
- **Terminal**: quedan enums y comentarios que nombran STOMP, y la dependencia `stomp_dart_client` todavía está en `pubspec.yaml`. La Fase 3 de la migración es eliminarlos.

Ninguno de esos vestigios participa del flujo real. La verdad operativa de hoy es **REST**.

El ciclo de pago y fiscalización

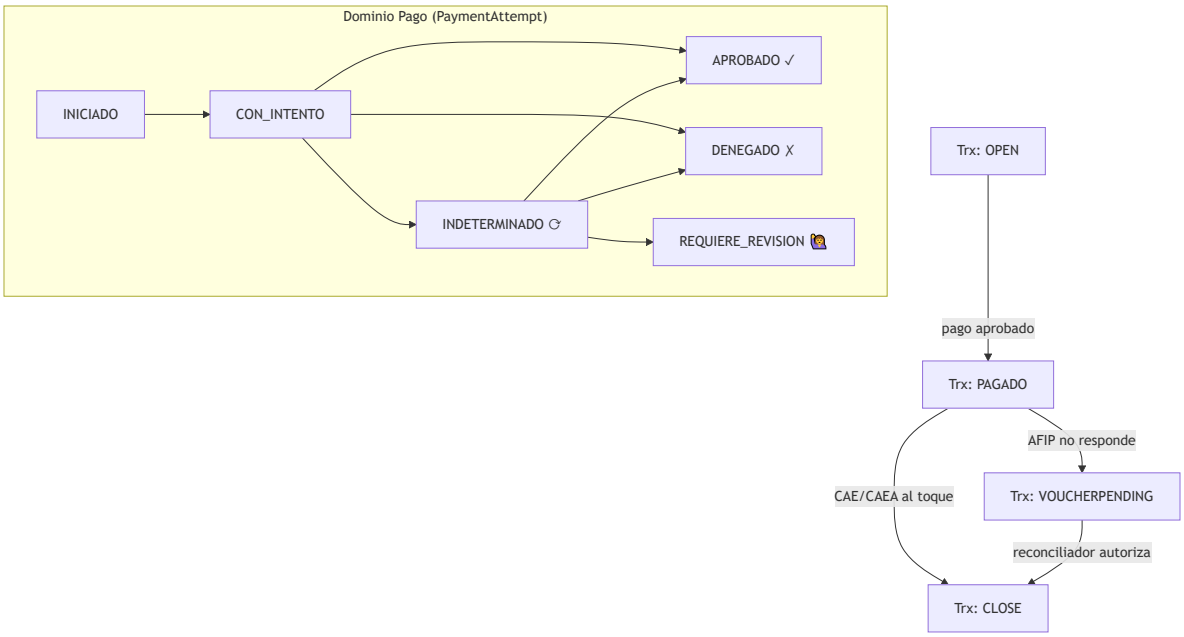
Esta es la conversación más difícil del sistema, y la que más plata pone en juego. Cuando un cliente paga en la terminal, pasan dos cosas que **pueden fallar por separado**: el **cobro** (¿el procesador aprobó?) y la **fiscalización** (¿AFIP autorizó el comprobante?). La red se cae justo en el medio más seguido de lo que uno quisiera. Esta página explica cómo el sistema atraviesa esa incertidumbre **sin perder plata ni emitir comprobantes truchos**. Para el detalle de cada pieza, está la referencia de facturación fiscal y la de pagos.

Dos dominios, dos verdades

Hay que separar tajantemente dos cosas que la intuición tiende a mezclar:

Dominio	Pregunta	Estado	Dueño
Pago	¿Se cobró la plata?	`EstadoPagoIntento` (`PaymentAttempt`)	el procesador (MercadoPago / ApiCard)
Fiscalización	¿AFIP autorizó el comprobante?	`TrxEstado` (`Trx`)	AFIP

Son independientes. Podés tener la plata cobrada y el comprobante sin autorizar (eso es `VOUCHERPENDING`). Podés tener un pago indeterminado mientras el ticket sigue esperando. **La regla que ordena todo**: el ticket **no se cierra** hasta que el pago esté `APROBADO`. El dominio de pago manda sobre el de ticket.



El enemigo: el estado indeterminado

El caso que rompe los sistemas ingenuos es este: la terminal le pide al procesador que cobre, el procesador **cobra**, y justo ahí se corta la red **antes** de que la terminal reciba el "aprobado". ¿Qué sabe la terminal? Nada. ¿Se cobró? No sabe. Si asume que no y reintenta, **cobra dos veces**. Si asume que sí y no se cobró, **regala mercadería**.

La respuesta del sistema tiene tres patas:

1. Idempotencia: reintentar es seguro

Cada intento de pago tiene un `paymentAttemptId` que la terminal genera **una sola vez** por intención y **reusa** en cada reintento. El backend deduplica por ese id. Así, "cobrar" el mismo intento N veces produce **un solo cobro**. Esto convierte el reintento de algo peligroso a algo seguro: ante la duda, se reintenta el **mismo** intento, no uno nuevo.

> **El error que nunca hay que cometer**
>
> Generar un `paymentAttemptId` nuevo en un reintento. Eso rompe la dedup y habilita el doble cobro. El id es la identidad del **intento**, no del request.

2. Durabilidad: el pago in-flight sobrevive

La terminal persiste el intento en Isar (`IsarPendingPayment`) **antes** de cualquier I/O de red. Si la terminal se apaga (corte de luz, crash) en medio del cobro, al arrancar lee esos pendientes y los reconcilia. La plata no queda en el limbo del lado del cliente.

3. Reconciliación: el backend resuelve solo

Del lado del backend, el `PaymentReconciliador` (un job `@Scheduled`) levanta los `PaymentAttempt` que quedaron `INDETERMINADO` y le pregunta al procesador "¿este pago, al final, se cobró?" (`CHECK_STATUS`). Según la respuesta:

- **Aprobado** → finaliza el ticket (cierre fiscal, `CLOSE`).
- **Denegado** → marca el intento terminal.
- **Sigue sin saberse** → reintenta, hasta un máximo; si se agota, escala a `REQUIERE_REVISION`.

Lo importante: **esto pasa aunque la terminal esté apagada**. El backend no depende de que el POS esté poleando para cerrar la incertidumbre. Esa es la diferencia entre un pago que se resuelve solo y uno que queda colgado para siempre.

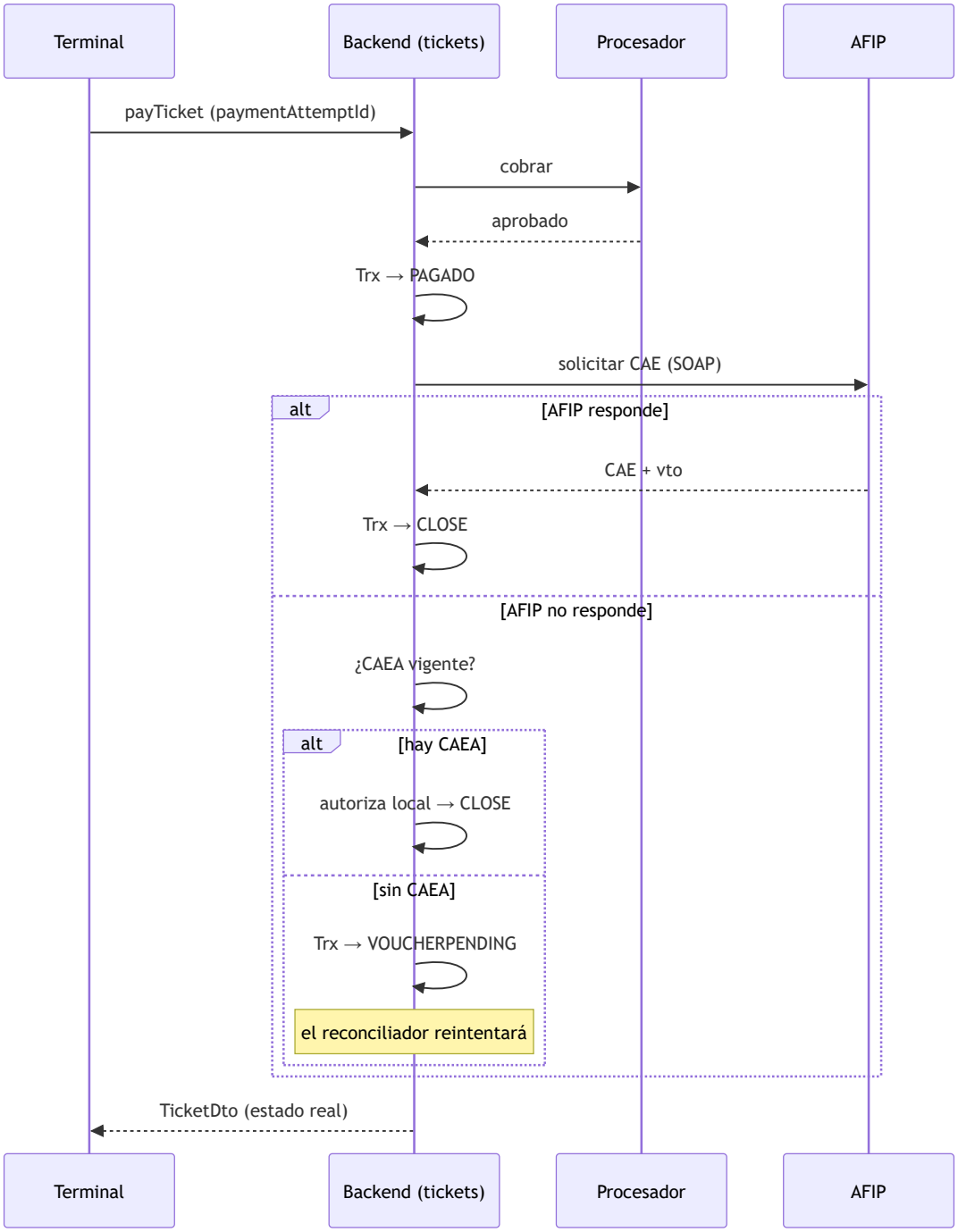
> ⚠ **REQUIERE_REVISION: cuando la máquina se rinde**
>
> Si tras todos los reintentos el sistema todavía no puede decidir si se cobró, **no inventa una respuesta**: escala a una persona. El Cockpit lo muestra en el panel fiscal. Es la admisión honesta de que algunos casos necesitan ojos humanos — y es mucho mejor que adivinar con la plata de por medio.

La segunda incertidumbre: AFIP

Resuelto el cobro, falta el comprobante. Y AFIP también se cae. Por eso hay **dos vías** de autorización:

- **CAE** (online): se pide un código por comprobante, por SOAP, en el momento. Es lo normal cuando AFIP responde.
- **CAEA** (anticipado/local): un código que AFIP otorga **por adelantado** para toda la quincena. Si AFIP está caído justo en la venta, el comercio igual puede facturar con el CAEA que ya tenía guardado.

Si ninguna vía funciona en el momento (AFIP caído **y** sin CAEA vigente), el ticket queda en `VOUCHERPENDING`: cobrado, pero sin papel fiscal. Es un estado **durable** y esperado, no un error. El reconciliador (o un reintento posterior) lo resuelve cuando AFIP vuelve.



Por qué está diseñado así

Podría parecer sobre-ingeniería. No lo es: es la diferencia entre un autoservicio que se puede dejar operando solo y uno que necesita un humano vigilando cada pago. Las propiedades que el diseño garantiza:

- **Nunca se cobra dos veces** (idempotencia por `paymentAttemptId`).
- **Ningún pago queda en el limbo** (durabilidad en Isar + reconciliador en el backend).
- **Nunca se emite un comprobante sin respaldo de cobro** (el ticket no cierra hasta `APROBADO`).
- **Se puede facturar con AFIP caído** (CAEA anticipado).
- **Cuando la máquina no puede decidir, lo admite** (`REQUIERE_REVISION`), en vez de adivinar.

> **Si vas a tocar pagos o fiscalización**
>
> Leé esta página y las dos referencias (pagos, facturación fiscal) **enteras** antes. Es el código donde un bug no se ve en una demo pero aparece como un descuadre de caja a fin de mes. Acá la prudencia no es opcional.

Decisiones de arquitectura (ADRs)

Un **ADR** (Architecture Decision Record) es un registro corto de una decisión de diseño importante: qué se decidió, en qué contexto y con qué consecuencias. Sirve para que quien llega nuevo entienda **por qué** las cosas son como son, sin tener que arqueologizar el git.

En AutoCompraMod, las decisiones de mayor peso ya están narradas en la sección **Entender el sistema**; esta página las resume como índice y deja lugar para futuros ADRs formales.

Decisiones tomadas

ADR-001 — Monolito modular en lugar de microservicios

Contexto. El autoservicio eran cinco microservicios (`TsTicket`, `TsArticulos`, `TsPagos`, `TsPromos`, `TsEnvases`) con cargas chicas que escalaban juntos.

Decisión. Consolidarlos en un **monolito modular** con Spring Modulith: un solo deployable con fronteras internas verificadas por el build.

Consecuencias. Un solo deploy y una sola API para la terminal; el guardrail (`ModularityTests.verify()`) protege las fronteras; queda abierto el camino a extraer un módulo a servicio si algún día hiciera falta. → El monolito modular

ADR-002 — Migración strangler, no big-bang

Contexto. Reescribir los cinco servicios de golpe era riesgoso.

Decisión. Migrar **de a un módulo por vez** (`promos` piloto → `catalogo` → `envases` → `pagos` → `tickets` host), corriendo `verify()` en cada paso, dejando los repos viejos como referencia.

Consecuencias. Riesgo acotado en cada paso; el guardrail caza regresiones de frontera en el acto. → Migración strangler

ADR-003 — Topología en estrella acíclica

Contexto. Sin reglas, los módulos terminarían llamándose entre sí en cualquier dirección.

Decisión. Sólo `tickets` (orquestador) llama; el resto son **hojas**. Cero ciclos. `shared` es kernel OPEN. Comunicación sólo por API pública o eventos; datos schema-por-módulo sin foreign keys cruzadas.

Consecuencias. El grafo de dependencias es trivial de razonar y el guardrail lo hace cumplir. → Arquitectura

ADR-004 — La terminal habla REST, no STOMP

Contexto. TiprePOS usaba STOMP/WebSocket, con un broker y suscripciones.

Decisión. Cutover a **REST puro** bajo `/autocompras/v1` : mutaciones por `POST` , salud por polling adaptativo (60 s / 15 s).

Consecuencias. Modelo de comunicación más simple y depurable; STOMP queda como vestigio en deprecación. → Comunicación POS ↔ Backend

ADR-005 — Seguridad app-wide con toggle

Contexto. Cinco servicios tenían cada uno su seguridad; al consolidar hacía falta un único modelo, y durante el parallel-run no se podía romper el runtime del POS.

Decisión. Un solo `SecurityFilterChain` + `JwtAuthConverter` . Toggle `app.security.enabled` (default `false`) y `app.security.admin-role` (default `ADMIN`); con seguridad activa se gatean sólo las mutaciones admin.

Consecuencias. Flexibilidad para dev/parallel-run, con el riesgo conocido de que `enabled=false` deja todo `permitAll` (se loguea un `WARN`). → Arquitectura → Autenticación

ADR-006 — Cockpit in-process, sin STOMP, snapshots @Scheduled

Contexto. Hacía falta observabilidad del parque de terminales y del backend sin montar un stack de monitoreo aparte.

Decisión. Un módulo `monitoring` que expone `/monitoring/*` , calcula **snapshots periódicos** (`@Scheduled`) en vez de consultar por request, persiste errores 30 días en un DataSource separado, y se sirve como SPA React desde el propio backend.

Consecuencias. Observabilidad barata y autocontenida; el Cockpit lee las APIs públicas sin acoplarse a internals. → Referencia → Cockpit

Cómo agregar un ADR nuevo

Cuando tomes una decisión de arquitectura con consecuencias duraderas, agregá una entrada acá con el mismo formato: **Contexto** → **Decisión** → **Consecuencias**, y enlazá a la página de explicación que la desarrolla. Un ADR es corto a propósito: si necesita tres páginas, esas tres páginas van en `explanation/` y el ADR las referencia.

Backend AutoCompraMod

Referencia técnica del backend: stack, layout, build, datos y seguridad. Para el *porqué* del diseño, mirá El monolito modular; acá están los datos exactos.

Stack

Pieza	Versión / valor
Lenguaje	Java 21 (enforcer fuerza `[21,22)`)
Framework	Spring Boot 3.4.0
Modularidad	Spring Modulith 1.3.0
Persistencia	Spring Data JPA / Hibernate, SQL Server (`mssql-jdbc`)
Auditoría	Hibernate Envers (entidad `Vale`)
Cache	Caffeine (catálogo ~100k)
Async externo	Spring WebFlux `WebClient` + Reactor Netty (gateways de pago)
Seguridad	OAuth2 Resource Server (JWT / Keycloak)
QR	ZXing
Voucher / impresión	escpos-coffee (ESC/POS) + OpenPDF (reemplazó iText 5 AGPL)

Frontend Cockpit	React 18 + Vite + TypeScript (build vía `frontend-maven-plugin`)
Context-path	`/autocompras/v1`

Layout de paquetes

```
com.tipre.autocompras
├── tickets/        orquestador (API del POS)
├── catalogo/       artículos por EAN/sucursal
├── pagos/          intent/pay/check/cancel (stateless)
├── promos/         cálculo de promociones
├── envases/        vales y depósitos retornables
├── shared/         kernel OPEN (ResponseMessage, enums, NucleoImpositivoDto)
├── security/       SecurityConfig + JwtAuthConverter (app-wide)
└── monitoring/     Cockpit (observabilidad)
```

Cada módulo se declara con un `package-info.java` anotado `@ApplicationModule` (`shared` es `type = OPEN`). El detalle de roles y reglas está en Los módulos.

Datos: schema-por-módulo

No hay una base única: cada módulo con persistencia define su propio `DataSource` (`hbm2ddl.auto=none` — el schema no lo maneja la app). **Cero foreign keys cruzadas entre módulos.**

Módulo	Config	Base	Notas
`tickets`	`TicketsDataSourceConfig` (`@Primary`)	`TsAutoCompra` (SQL Server)	Entidades `Trx`, `PaymentAttempt`, `PosConfig`, `DeviceConfig`.
`catalogo`	`CatalogoDataSourceConfig`	`TipreRetail`	Read-only, cache Caffeine ~100k.
`promos`	`PromosDataSourceConfig`	`TsPromos`	Promos + auditoría de cálculos.

`envases`	`EnvasesDataSourceConfig`	`TsEnvases`	`Vale` (`@Audited`, Envers → tablas `_AUD`), `Envase`.
`pagos`	—	—	Stateless: no persiste, usa `WebClient` a gateways.
`monitoring`	`MonitoringDataSourceConfig`	DB de observabilidad separada	`ErrorEvent`, retención 30 días.

Entidades principales

- **`Trx`** (tickets) — el ticket persistido. Estados: `OPEN`, `PAGADO`, `VOUCHERPENDING`, `CLOSE`, `CANCELED_USER`, `CANCELED_INACTIVITY`, `ERROR`.
- **`Articulo`** (catalogo) — `@Table(name = "dm_Artic")`. Campos: `cEan`, `cDescripcion`, `fPrecio`, `iTax`, `bPeso`, `bEnvase`, `iNroSuc`, etc.
- **`Vale`** (envases) — `@Entity @Audited`. Campos: `nroVale`, `codVale`, `nroSucursal`, `fechaCreacion`, `fechaVto`, `terminalOrigen`, estado. Genera `Vales_AUD`.
- **`Envase`** (envases) — `ean`, `plu`, `descripcion`, `nroSucursal`, `imagen` (`@Lob`), `habilitado`, `orden`.

Seguridad

App-wide: un único `SecurityFilterChain` y `JwtAuthConverter` en el paquete `security` (`SecurityConfig.java`).

Propiedad	Default	Efecto
`app.security.enabled`	`false`	`false` → `anyRequest().permitAll()`. `true` → política de roles (resource server JWT/Keycloak).

`app.security.admin-role`	`ADMIN`	Rol exigido en mutaciones admin (espera authority `ROLE_<admin-role>`).

Con seguridad activa:

- `permitAll`: `/actuator/health`, `/actuator/info`, `/*/dummy`, swagger.
- `hasRole(adminRole)`: `POST /promociones/update`, `POST /cache/refresh`, `POST /catalogo/cache/refresh`.
- `authenticated()`: el resto (incluido el runtime del POS, para no romper el checkout).

> ⚠ **Toggle en `false` = todo abierto**
>
> Con `app.security.enabled=false` **todos** los endpoints quedan `permitAll`, incluidas las mutaciones admin. Es sólo para dev / parallel-run; se loguea un `WARN` al armar el filter chain. En producción con Keycloak: `enabled=true` y `admin-role` al rol real del realm.

Build y guardrail

```
# Build + verificación de boundaries de módulos. Requiere JDK 21.
mvn test

# Levantar (con los DataSource configurados)
mvn spring-boot:run
```

- **`ModularityTests`** corre `ApplicationModules.of(...).verify()` y **rompe el build** si se viola una frontera o aparece un ciclo. Ver El monolito modular.
- **CI**: GitHub Actions corre `mvn test` con JDK 21 en cada push/PR (~411 tests; 22 deshabilitados por requerir infra de DB).
- **Cockpit**: el profile `frontend` de Maven instala Node local (`frontend-maven-plugin`), corre `npm ci && npm run build` y copia `dist/` a `src/main/resources/static/cockpit`, servido por Spring como SPA.

Por dónde seguir

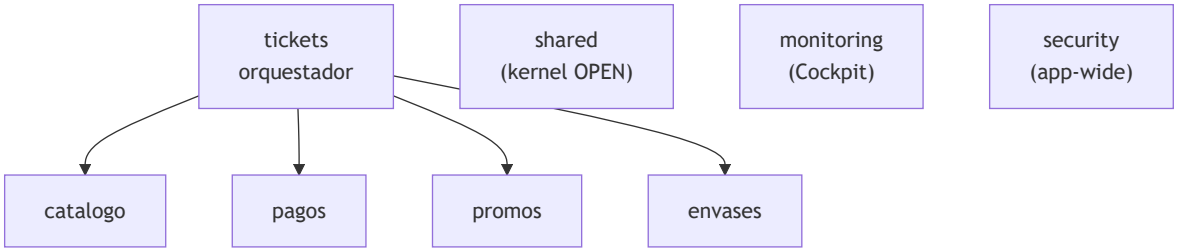
- Los módulos — qué hace cada uno y su API pública.

- API REST — endpoints concretos.
- Cockpit — el panel de operación y `/monitoring/*`.

Los módulos

El backend está dividido en módulos Spring Modulith. Cada uno tiene una **API pública** (lo que está en la raíz de su paquete) e **internals** privados. Acá: qué hace cada uno, su API pública y a quién llama. El *porqué* de esta estructura está en El monolito modular.

Mapa rápido



Módulo	Rol	Llama a
`tickets`	Orquestador. La API que ve el POS.	catalogo, pagos, promos, envases
`catalogo`	Artículos por EAN/sucursal + búsqueda.	— (hoja)
`pagos`	intent/pay/check/cancel. Stateless.	— (hoja, sale a gateways)
`promos`	Calcula promociones del ticket.	— (hoja)
`envases`	Vales y depósitos retornables.	— (hoja)
`shared`	Kernel OPEN: lengua común.	—

`security`	Seguridad app-wide.	—
`monitoring`	Cockpit / observabilidad.	— (lee APIs públicas)

`tickets` — el orquestador

El corazón del backend: la API que consume la terminal y la que coordina a los demás módulos para armar, calcular y cerrar un ticket.

- **Servicios**: `TicketService` (ciclo de vida del ticket), `VoucherService` (generación e impresión del comprobante), `EanDecodeService` (decodifica el EAN escaneado), `TerminalSessionService` (vestigio STOMP, **muerto**).
- **Controllers**: `TicketRestController` (open/close/validate), `PosRestController`, `TerminalSessionRestController`, `CacheRestController`, `EnvasesRestController`, `TemplateController`, `TestToolController`.
- **Datos**: `Trx` (el ticket), `PaymentAttempt`, `PosConfig`, `DeviceConfig` en la base `TsAutoCompra`.

> **Dos issues conocidos en `VoucherService`**
>
> - **Thread-safety** (plan 007): un campo de config mutable compartido entre requests puede hacer que un thread pise la config de otro mientras imprime. La solución es aislar por thread.
> - **`getFirst()` sin guarda** (plan 008): un `.getFirst()` sobre lista vacía tira `NoSuchElementException` opaca si falta config; conviene fallar explícito con mensaje.

`catalogo` — artículos

Resuelve artículos por EAN y sucursal, y ofrece búsqueda paginada. Read-only sobre la base `TipreRetail`, con **cache Caffeine** de ~100k artículos.

- **Servicios**: `ArticuloService`, `ArticuloCacheService`.
- **Controller**: `ArticuloRestController` — `GET /searchBy` (por `id` / `codigoInterno` / `ean` / `descripcion` / `sucursal`, paginado, **máx. 200 ítems** por página).
- **Admin**: `ArticuloCacheAdminController` — `POST /catalogo/cache/refresh` (rol admin si la seguridad está activa).

- **Datos**: `Articulo` (`dm_Artic`), `Envase` (projection).

`pagos` — cobros

Maneja el ciclo de pago contra los gateways. **Stateless**: no persiste, sale por `WebClient` a MercadoPago / RouterQR.

- **Servicios**: `PagoService`, `RealPaymentProcessorService`, `RouterQRBuilder`.
- **Controller**: `PagoRestController` — `POST /payment-intent` (async, devuelve `Mono<ResponseEntity>`), pay, check-status.
- **Borde externo**: la salida a MercadoPago es red real; ahí viven la reconciliación durable y el manejo del estado indeterminado. Ver Pagos.

`promos` — promociones

Calcula las promociones aplicables a un ticket. Hoja, con cache propia.

- **Servicios**: `PromoService`, `PromosCacheService`.
- **Controller**: `PromoController` — `GET /promociones`, `POST /promociones/update` (rol admin), `POST /cache/refresh` (rol admin).
- **Datos**: promociones + auditoría de cálculos en `TsPromos`.

`envases` — vales y retornables

Gestiona envases retornables y los vales asociados.

- **Servicios**: `ValeService`, `EnvaseService`.
- **Controllers**: `ValeController` (`@RequestMapping("/vales")` — `GET /vales?codVale=`, alta, update), `EnvaseController`.
- **Datos**: `Vale` (`@Audited` con Envers → `Vales_AUD`), `Envase` en `TsEnvases`.

`shared` — kernel OPEN

La lengua común que cualquier módulo puede usar sin violar fronteras: `ResponseMessage` (la envoltura unificada de respuestas), los enums de pago y `NucleoImpositivoDto` (info impositiva). No tiene lógica de negocio.

`security` — app-wide

Un único `SecurityFilterChain` + `JwtAuthConverter`. Detalle en Backend → Seguridad.

`monitoring` — el Cockpit

Observabilidad del backend y del parque de terminales. Lee las APIs públicas de los demás módulos (no sus internals) y expone `/monitoring/*`. Ver Cockpit.

El agregado Ticket y su cómputo

El **ticket** es el agregado raíz del sistema: todo —itemización, promociones, envases, impuestos, pago y fiscalización— gira alrededor de él. Esta página describe su estructura, su ciclo de vida y, sobre todo, **el orden exacto en que se computa**. Es la página más importante de la referencia: si vas a tocar la lógica de negocio, leela entera.

> ⚠ **El código es la fuente de verdad**
>
> Reproducimos fórmulas, nombres de clase y fixes tal como están en el código a la fecha. Algunas firmas exactas de método pueden cambiar; ante cualquier discrepancia, mandan las clases en `src/main/java/com/tipre/autocompras/tickets/`.

El agregado: `Trx`

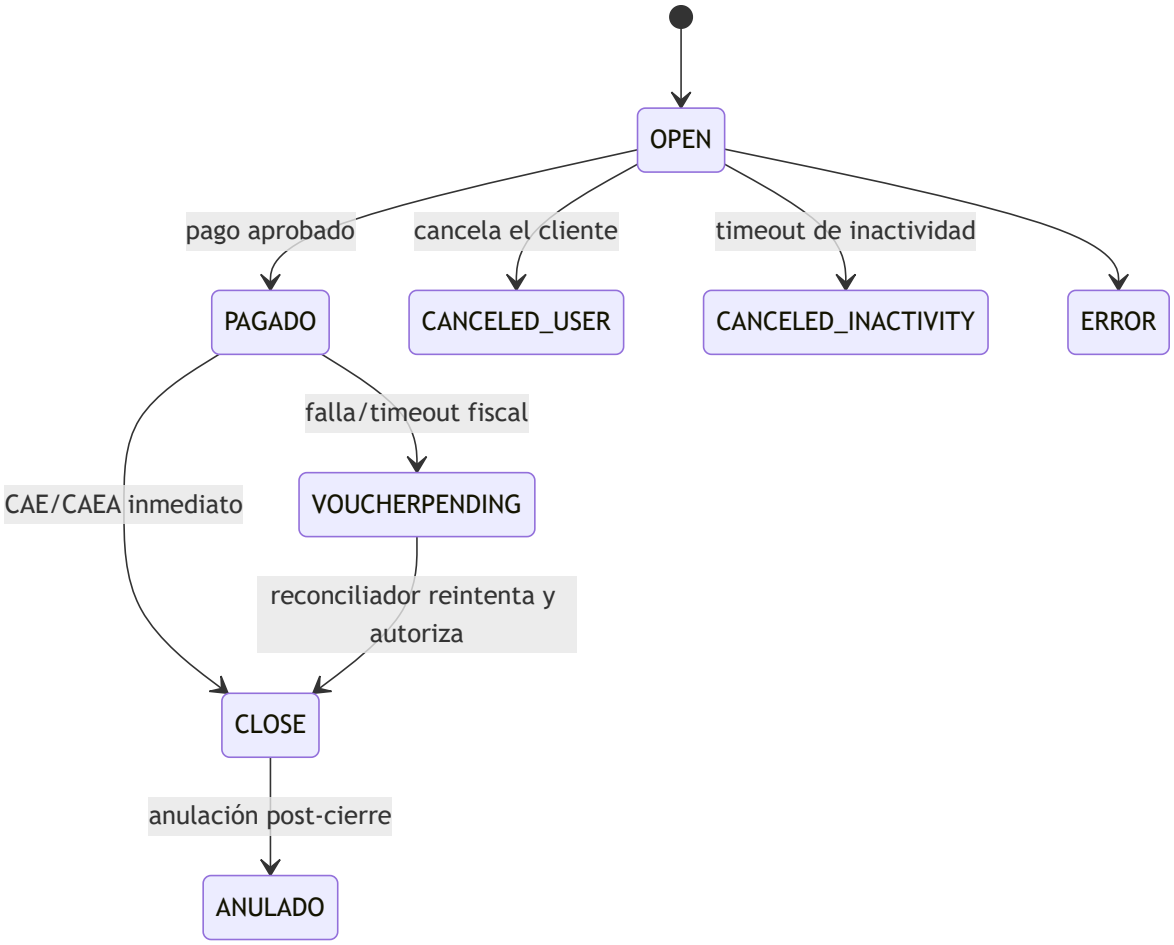
El ticket se persiste como la entidad `Trx` (`tickets/domain/Trx.java`) en la base `TsAutoCompra`. Campos principales:

Campo	Tipo	Rol
`id`	`Long`	PK autogenerada.
`fechaCreacion` / `fechaModificacion`	`LocalDateTime`	Apertura y última modificación.
`codComercio`, `nroSucursal`, `nroPos`, `codTerminal`	—	Identidad de origen.
`nroTicketPos`	`int`	Número secuencial del ticket en la terminal.
`total`	`BigDecimal`	Monto total.
`estado`	`TrxEstado`	Estado actual (ver abajo).

`jsonticket`	`TEXT`	**El `TicketDTO` completo serializado a JSON.** Acá vive el detalle (ítems, movimientos, promos, núcleo impositivo).
`idempotencyKey`	`String(64)`	Clave de deduplicación (fix TSTK-006).
`errorMessage`	`String`	Mensaje si terminó en error.

> **El detalle vive en el JSON, no en columnas**
>
> `Trx` guarda los campos de cabecera en columnas y **el cuerpo completo del ticket en `jsonticket`** como un `TicketDTO` serializado. Esto mantiene el modelo relacional simple y deja el agregado rico en el JSON. Cuando el POS pide un ticket, recibe ese `TicketDTO`.

Estados del ticket: `TrxEstado`



Estado	Significa
`OPEN`	Abierto; se agregan/quitan ítems.
`PAGADO`	Pago aprobado, en tránsito a cierre fiscal.
`VOUCHERPENDING`	Pagado pero el comprobante aún no fue autorizado por AFIP. Durable; lo resuelve el reconciliador. Ver Facturación fiscal.
`CLOSE`	Cerrado y autorizado fiscalmente (comprobante emitido).
`CANCELED_USER`	Cancelado por el cliente (desde `OPEN`).

`CANCELED_INACTIVITY`	Cancelado automáticamente por inactividad (desde `OPEN`).
`ERROR`	Terminó en error.
`ANULADO`	Anulado después del cierre.
`TEST`	Ticket de prueba.

> **Regla de oro: el ticket no se cierra hasta que el pago esté APROBADO**
>
> `Trx.estado` **nunca** avanza a `PAGADO` / `CLOSE` mientras el `PaymentAttempt` no esté `APROBADO`. El dominio de pago manda sobre el de ticket. El detalle está en El ciclo de pago y fiscalización.

Itemización

El `TicketDTO` no es una lista plana de líneas: separa **ítems** (lo que el cliente ve) de **movimientos** (las anotaciones contables que componen el total). Esta separación es la que permite que una misma línea reciba una venta, un descuento de promo y un crédito de envase sin perder trazabilidad.

- **Ítem** (`ItemDTO`): una línea del ticket. Tiene su artículo, cantidad, tipo (`ItemTicketTipo`: venta normal, `ENVASE`, etc.) y estado (`ItemTicketEstado`: `VENTA`, `SALDADO`, `ANULADO`).
- **Movimiento** (`MovimientoDTO`): una anotación sobre un ítem. Una venta es un movimiento; un descuento de promo es otro movimiento ligado al mismo ítem; un crédito de envase, otro. El total del ticket es la suma de los movimientos.

Operaciones (todas mutaciones REST, ver API REST):

Operación	Qué hace
`openTicket`	Crea el `Trx` en `OPEN`.
`agregarArticulo`	Resuelve el EAN en `catalogo`, arma el ítem y su movimiento de venta, recalcula promos y total.

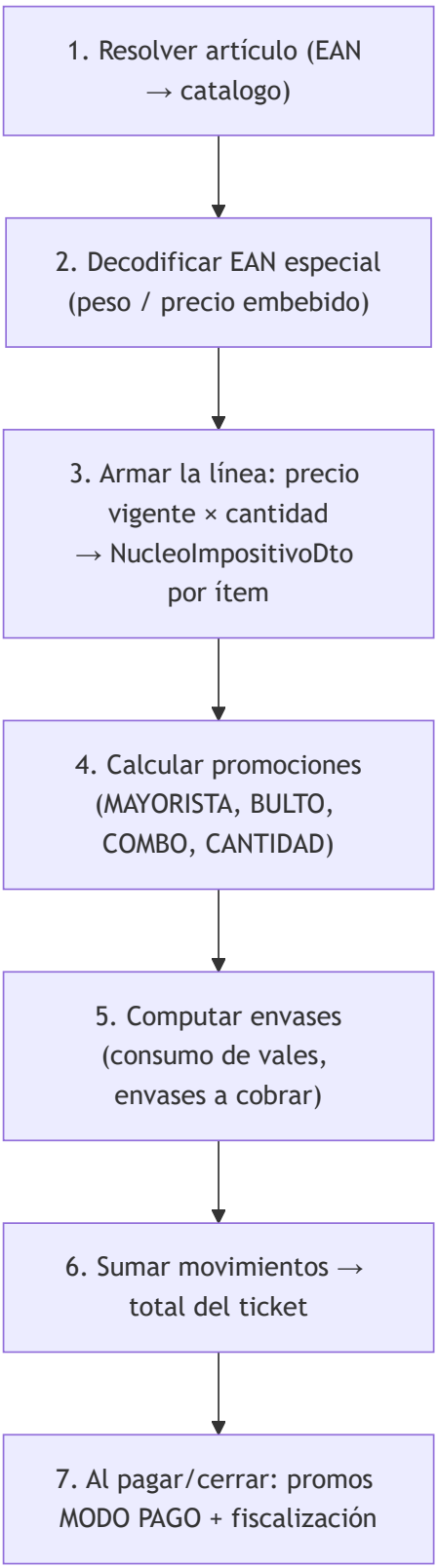
`removeItem`	Quita un ítem (o lo marca `ANULADO`) y recalcula.
`changeStatus`	Cambia el estado del ticket.
`validateTicket`	Valida el ticket antes de pagar.
`closeTicket`	Cierra (dispara fiscalización).

Solo los ítems en estado VENTA se facturan

Al armar la solicitud fiscal, `CaeService.resolveFacturableItems` toma **solo los ítems en `VENTA`**. Los `SALDADO` (saldados por envase/crédito) y `ANULADO` no van al comprobante.

El orden de cálculo

Este es el corazón. Cuando se agrega o quita un artículo, el total se recompone siguiendo un **orden fijo**. Saltearse un paso o cambiarlo de lugar produce totales mal calculados.


1. Resolver artículo (EAN → catalogo)
2. Decodificar EAN especial (peso / precio embebido)
3. Armar la línea: precio vigente × cantidad → NucleoImpositivoDto por ítem
4. Calcular promociones (MAYORISTA, BULTO, COMBO, CANTIDAD)
5. Computar envases (consumo de vales, envases a cobrar)
6. Sumar movimientos → total del ticket
7. Al pagar/cerrar: promos MODO PAGO + fiscalización

1. **Resolución del artículo**: el EAN escaneado se busca en `catalogo` (cache Caffeine ~100k). Devuelve precio, alícuota (`iTax`), flags (`bPeso`, `bRandomPrice`, `bEnvase`), impuesto interno (`fImpinterno`).
2. **Decode de EAN especial**: si el EAN codifica peso o precio (productos pesables, random price, DUN de bulto), se extrae. Ver Decodificación de EAN.
3. **Armado de la línea**: se calcula el `NucleoImpositivoDto` del ítem (neto + IVA + impuestos internos) a partir del precio vigente y la cantidad. Ver Núcleo impositivo.
4. **Promociones**: se invoca el módulo `promos`. Primero precios MAYORISTA y BULTO (por ítem), luego COMBO/CANTIDAD/CUPONES. El beneficio se distribuye proporcionalmente entre las líneas. Ver Promociones.
5. **Envases**: se computa el consumo de vales contra los ítems de tipo `ENVASE` y se determina cuántos envases hay que **cobrar** (`cantEnvasesCobrar = max(0, cantEnvases - consumidos)`). Ver Envases y vales.
6. **Total**: la suma de todos los movimientos da el `total` del ticket.
7. **Al pagar/cerrar**: se recalculan las promos que dependen del **medio de pago** (MODO PAGO, ver Promociones) y se dispara la fiscalización.

Redondeo y escala

Todo el dinero usa `BigDecimal` con reglas consistentes (definidas en `NucleoImpositivoDto`):

- **Escala de moneda**: 2 decimales, `RoundingMode.HALF_UP`.
- **Escala de cálculo intermedio**: mayor (≈10 decimales) para no acumular error antes del redondeo final.
- En la **distribución proporcional** de descuentos, la **última línea absorbe el residuo** para que la suma de las porciones sea exactamente el total (evita el centavo perdido por truncamiento).

> ⚠ **Nunca redondees a mano en pasos intermedios**
>
> El error clásico es redondear a 2 decimales en cada paso. Eso descuadra el ticket. Se calcula con escala alta y se redondea **una sola vez** al final. La aritmética vive en `NucleoImpositivoDto` justamente para centralizar esto.

El comprobante (voucher)

Cuando el ticket cierra y tiene CAE/CAEA, `VoucherService` genera el comprobante imprimible:

- **Impresión térmica**: ESC/POS vía `escpos-coffee`.
- **PDF**: vía OpenPDF (reemplazó iText 5 AGPL — plan 011).
- **QR fiscal de AFIP**: generado con ZXing a partir de la URL fiscal (`AfipUtil.getQrAfip`). Ver Facturación fiscal.
- **Templates**: gestionados por `TemplateController`.

`VoucherType` distingue: `TICKET`, `TICKET_PDF`, `ENVASE`, `ENVASE_TOMRA`, `CUPON`, `ANULACION_PAGO`.

> **Dos issues conocidos en `VoucherService`**
>
> - **Thread-safety** (plan 007): un campo de config mutable compartido entre requests puede hacer que un thread pise la config de otro mientras imprime.
> - **`getFirst()` sin guarda** (plan 008): un `.getFirst()` sobre lista vacía tira `NoSuchElementException` opaca si falta config.

Idempotencia y casos especiales

- **`idempotencyKey`** (TSTK-006): deduplica mutaciones reenviadas. La misma operación repetida no duplica el efecto.
- **Cierre por inactividad**: la terminal tiene timers (`InactividadBloc`); un ticket abandonado pasa a `CANCELED_INACTIVITY`.
- **Zona horaria**: las fechas con efecto fiscal se calculan en `America/Argentina/Buenos_Aires`, no en la zona de la JVM (fix TSTK-024).
- **Logging por ítem** (plan 010): `calcularTotalTicket` logueaba 2× INFO por ítem (1800 líneas en un carrito de 30); se baja a DEBUG.

Por dónde seguir

- Decodificación de EAN — el paso 2.
- Núcleo impositivo — el paso 3, el modelo fiscal.
- Promociones — el paso 4.
- Envases y vales — el paso 5.

- Facturación fiscal — el paso 7.

Decodificación de EAN

No todos los códigos de barras son iguales. Un yogur tiene un EAN fijo que mapea a un precio; pero un trozo de queso cortado en balanza trae el **peso o el precio embebidos en el propio código**. La terminal escanea el mismo tipo de lector para todos, así que el backend tiene que **decodificar** qué clase de EAN es y extraer la información correcta. Ese trabajo lo hace `EanDecodeService` (`tickets`), y es el **paso 2** del cómputo del ticket.

> ⚠ **El código es la fuente de verdad**
>
> Los prefijos y longitudes exactos dependen de la configuración de balanzas del comercio. Esta página describe el modelo; los valores concretos viven en `EanDecodeService` y en la config.

Por qué hace falta

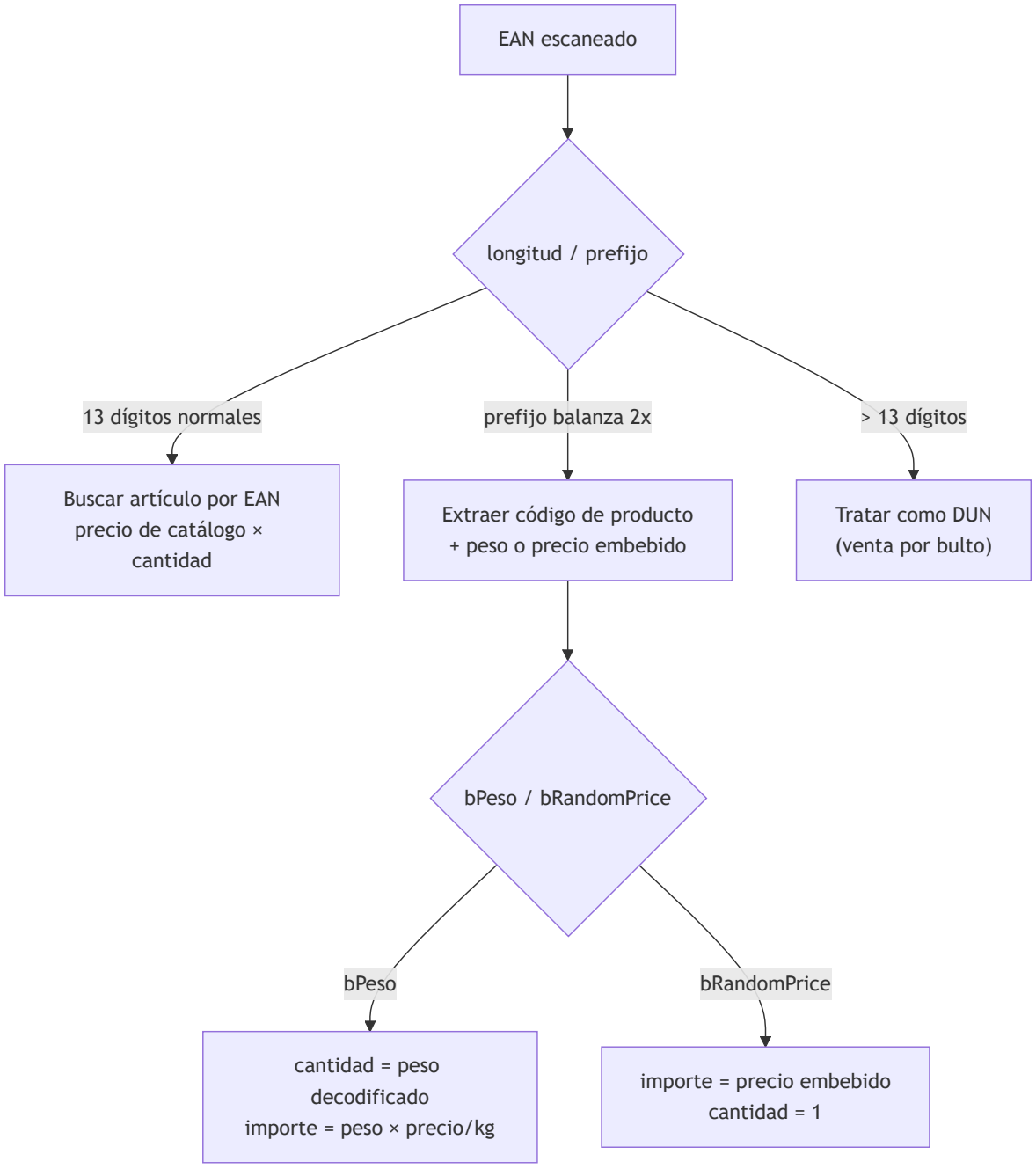
Un artículo del catálogo (`Articulo`, en `catalogo`) trae flags que indican cómo interpretarlo:

Flag	Significa
`bPeso`	El artículo es **pesable**: se vende por kg/g, no por unidad.
`bRandomPrice`	El **precio** viene embebido en el código (random price / price-embedded).
`bEnvase`	Es un **envase retornable** (ver Envases y vales).

Y el EAN escaneado puede ser:

- **EAN normal** (13 dígitos): mapea 1:1 a un artículo con precio fijo.
- **EAN pesable / random-price** (típicamente con prefijo `2x`): los primeros dígitos identifican el producto, y un tramo posterior codifica **peso** o **precio**, más un dígito verificador.
- **DUN** (más de 13 dígitos): identifica una **venta por bulto**. Ver Promociones → BULTO.

Cómo se decodifica



1. **Identificar el tipo** por longitud y prefijo del código.
2. **Extraer el código de producto** (el tramo que identifica el artículo) y buscarlo en `catalogo`.
3. **Extraer el valor embebido** (peso o precio) del tramo correspondiente.

4. **Armar la línea** según el flag del artículo:
5. **Pesable (`bPeso`)**: la *cantidad* del ítem es el **peso** decodificado (ej. 0,750 kg) y el importe = peso × precio por kg.
6. **Random price (`bRandomPrice`)**: el importe es el **precio embebido** directamente; la cantidad es 1.
7. **Normal**: cantidad entera × precio de catálogo.

Por qué importa para el cómputo

El resultado del decode alimenta el **paso 3** (armado de la línea y su núcleo impositivo): un pesable de 0,750 kg a \$2.000/kg produce una línea de \$1.500 con su IVA correspondiente, exactamente como si fuera una unidad de \$1.500. A partir de ahí, el resto del cómputo (promos, envases, total) no necesita saber que era pesable: ya quedó normalizado a una línea con su importe y su núcleo impositivo.

> **Pesables y random-price son donde más se rompe la itemización**
>
> Un dígito verificador mal interpretado, un prefijo de balanza no contemplado o un redondeo del peso producen importes incorrectos. Si vas a tocar `EanDecodeService`, probá con EANs reales de balanza del comercio (no inventados): los formatos varían por configuración.

Núcleo impositivo

El **núcleo impositivo** (`NucleoImpositivoDto`, en el kernel `shared`) es la pieza más transversal del sistema. Cada importe que viaja por el ticket —el precio de una línea, un descuento de promo, un crédito de envase, el total— **no es un `BigDecimal` suelto**: es un núcleo impositivo, es decir, un monto **con su descomposición fiscal adentro**. Entender esto es entender cómo el sistema mantiene el IVA y los impuestos cuadrados a lo largo de todo el cómputo.

> **Por qué un objeto y no un número**
>
> Si un descuento fuera solo "−\$100", al aplicarlo perderías sobre qué neto y qué IVA impactó. El núcleo impositivo lleva el monto **y** sus componentes (neto gravado, IVA, impuestos internos…), de modo que sumar, restar o prorratear importes **preserva la composición fiscal**. Esto es lo que permite emitir un comprobante con el IVA discriminado correcto al final.

Estructura

`shared/NucleoImpositivoDto.java`:

```
public class NucleoImpositivoDto {
    private BigDecimal monto;     // Total final (neto + impuestos)
    private BigDecimal neto;      // Base gravable
    private List<NucleoComponenteDto> componentes;  // Desglose por tipo

    public static final int SCALE = 8;          // precisión interna de cálculo
    public static final int DISPLAY_SCALE = 2;  // escala de moneda (mostrar)
}
```

Cada componente:

```
public static class NucleoComponenteDto {
    private NucleoComponenteTipo tipo;
    private BigDecimal base;      // importe base sobre el que se calcula
    private BigDecimal alicuota;  // % (ej. 21.00, 10.5)
    private BigDecimal monto;     // base × alícuota / 100
}
```

Tipos de componente

`NucleoComponenteTipo` modela el sistema impositivo argentino de retail:

Grupo	Componentes	Qué es
Neto (base imponible)	`NETO_IVA_21`, `NETO_IVA_10_5`, `NETO_IVAEXENTO`	La mercadería antes de impuestos, separada por la alícuota de IVA que le corresponde.
IVA	`IVA_21`, `IVA_10_5`	El IVA calculado sobre cada neto.
Impuestos internos	`IMPINTERNO`	Impuesto interno (tabaco, bebidas, etc.), del campo `fImpinterno` del artículo.
Percepciones	`PERCEPCION_IIBB`, `PERCEPCION_COMIND`, `PERCEPCION_IVA_21`, `PERCEPCION_IVA_10_5`	Percepciones impositivas (Ingresos Brutos, etc.).
Retenciones	`RETENCION_GANANCIAS`, `RETENCION_IVA`	Retenciones.

Qué está implementado y qué está modelado

Los componentes de **neto, IVA e impuestos internos** se computan y viajan en cada ticket. Los de **percepción y retención** existen en el enum (el modelo los contempla) pero, según el código actual, `CaeService` todavía **no arma tributos de percepción** en la solicitud a AFIP. Documentamos el modelo completo y marcamos lo que aún no se ejercita.

Aritmética que preserva la composición

`NucleoImpositivoDto` ofrece operaciones que operan sobre el monto **y** propagan a los componentes:

Operación	Qué hace
`add(otro)`	Suma montos y componentes, normaliza.
`subtract(otro)`	Resta (ej. aplicar un descuento), normaliza.
`multiply(factor)` / `divide(factor)`	Escala (ej. precio × cantidad).
`recomponer(nuevoMonto)`	Recalcula los componentes para un monto nuevo, **manteniendo las proporciones** (ej. fijar un precio de promo y que el IVA se reparta solo).
`normalized()`	Recompone `monto` / `neto` desde los componentes.
`zero()` / `isZero()`	Cero impositivo.

`recomponer()`: el método clave de las promos

Cuando una promo fija un precio nuevo (un combo a \$1.000, un precio mayorista), no se puede simplemente cambiar el monto: hay que **recalcular el neto y el IVA** para que sigan cuadrando. `recomponer(nuevoMonto)` hace eso, prorrateando el nuevo monto entre los componentes según su peso original.

`recomponer()` y la base cero

Si el monto base es cero, `recomponer()` dividiría por cero (`ArithmeticException`). El fix **TSPM-016** lo evita: un ítem con base cero no recibe descuento (su porción es cero). Si tocás aritmética de núcleo, respetá ese guard.

Normalización: el fix TSTK-025

Cuando un `NucleoImpositivoDto` se deserializa desde JSON, a veces `monto` / `neto` llegan en cero aunque los componentes tengan valores. `normalized()` recompone los totales desde los componentes para no perder precisión en las sumas/restas posteriores (fix **TSTK-025**). Por eso casi todas las operaciones llaman a `normalized()` al final.

Escala y redondeo

- **Cálculo interno**: `SCALE = 8` decimales — se calcula con holgura para no acumular error.
- **Moneda**: `DISPLAY_SCALE = 2`, `HALF_UP` — se redondea al mostrar/persistir el importe final.
- El módulo `promos` usa la misma escala (`MONEY_SCALE` + 8 para intermedios) justamente para ser consistente con el núcleo.

> **Regla práctica**
>
> Trabajá siempre con `NucleoImpositivoDto`, no con su `monto` pelado. Si extraés el `BigDecimal` y operás aparte, perdés la composición fiscal y vas a descuadrar el IVA del comprobante. La aritmética del dinero **vive acá** a propósito.

Dónde se usa

- En cada **línea** del ticket (paso 3 del cómputo).
- En cada **beneficio** de promoción (el descuento es un núcleo, no un número).
- En el **crédito de envases** (envases y vales).
- En el **comprobante fiscal**, donde el desglose por alícuota se vuelca al CAE/CAEA y al QR de AFIP (facturación fiscal).

Promociones

El módulo `promos` es un **motor completo de cálculo de promociones**: evalúa condiciones, resuelve conflictos entre promos, calcula el beneficio y lo **distribuye proporcionalmente** entre las líneas del ticket. Es el **paso 4** del cómputo del ticket y una de las piezas con más reglas de negocio del sistema.

> ⚠ **El código es la fuente de verdad**
>
> `PromoService` tiene ~2650 líneas y decenas de casos de borde. Esta página documenta el modelo, los tipos, el algoritmo y los fixes conocidos; el detalle fino vive en `src/main/java/com/tipre/autocompras/promos/`.

Los tres tipos de promoción

El enum `PromocionTipo` (`promos/model/enums/PromocionTipo.java`) distingue:

Tipo	Qué es	Resuelto por
`PROMOCION`	Promociones genéricas: se aplican con un **método** (`COMBO` o `CANTIDAD`).	`PromoService` (COMBO/CANTIDAD)
`MAYORISTA`	Escalas de precio por **volumen** de un EAN.	`PrecioMayoristaService`
`BULTO`	Venta especial por **bulto** (identificado por DUN).	`PrecioBultoService`

Además existen las **CUPONES**: promos cuyo `beneficio.destino = "CUPONES"`, que no dan descuento monetario directo sino que acumulan cupones (acción tipo `IMPRESION`).

Métodos de `PROMOCION`

- **COMBO**: requiere comprar productos de varios grupos (ej. "2 del grupo A + 1 del grupo B"). Agrupa por `nro`, cada grupo con su `cantidadminima`. Soporta múltiples ocurrencias por ticket.
- **CANTIDAD**: se activa por cantidad mínima de unidades elegibles (ej. "llevando 3, descuento"). Puede topar repeticiones con `maxOcurrenciasPorTicket`.

Tipos de beneficio

`BeneficioDto`:

```
private BigDecimal valor;   // monto o porcentaje
private String tipo;        // PORCENTAJE | MONTO | PRECIO_VENTA_FIJO
private String accion;      // DESCUENTO (default) | RECARGO
private String destino;     // ITEMS (default) | CUPONES | MDEP
```

`tipo`	**Cálculo**
`PORCENTAJE`	`base × (valor / 100)`.
`MONTO`	Descuento fijo en dinero.
`PRECIO_VENTA_FIJO`	Precio final fijo; beneficio = `base - valor`.

Modelo de datos

Entidad `Promociones` (base `TsPromos`):

```
@Entity @Table(name = "Promociones")
public class Promociones {
    @Id @GeneratedValue Long id;
    int version;
    @Column(columnDefinition = "NVARCHAR(MAX)") String jsondata;  // payload JSON
    String dataType;                  // PROMOCION | MAYORISTA | BULTO
    LocalDateTime fechaActualizacion;
}
```

El detalle de cada promo se deserializa del `jsondata` a `PromocionesDto`, que incluye vigencia (`fechadesde` / `fechahasta`, `horaEjecucionDesde/Hasta`, `diassemana`), `metodo`, `beneficio`, `condiciones`, `sucursales`, `mediodepago`, `convenios` y listas de inclusión/exclusión de artículos.

Las **condiciones** (`CondicionesDto`) son ricas: `acumulativa` (SI/NO), `usaPrecioLista`, `maxOcurrenciasPorTicket`, `montominimo`, `cantidadminima`, y exclusiones (`excluyeVentaMayorista`, `excluyeVentaXBulto`, `excluyeArticulosOferta`).

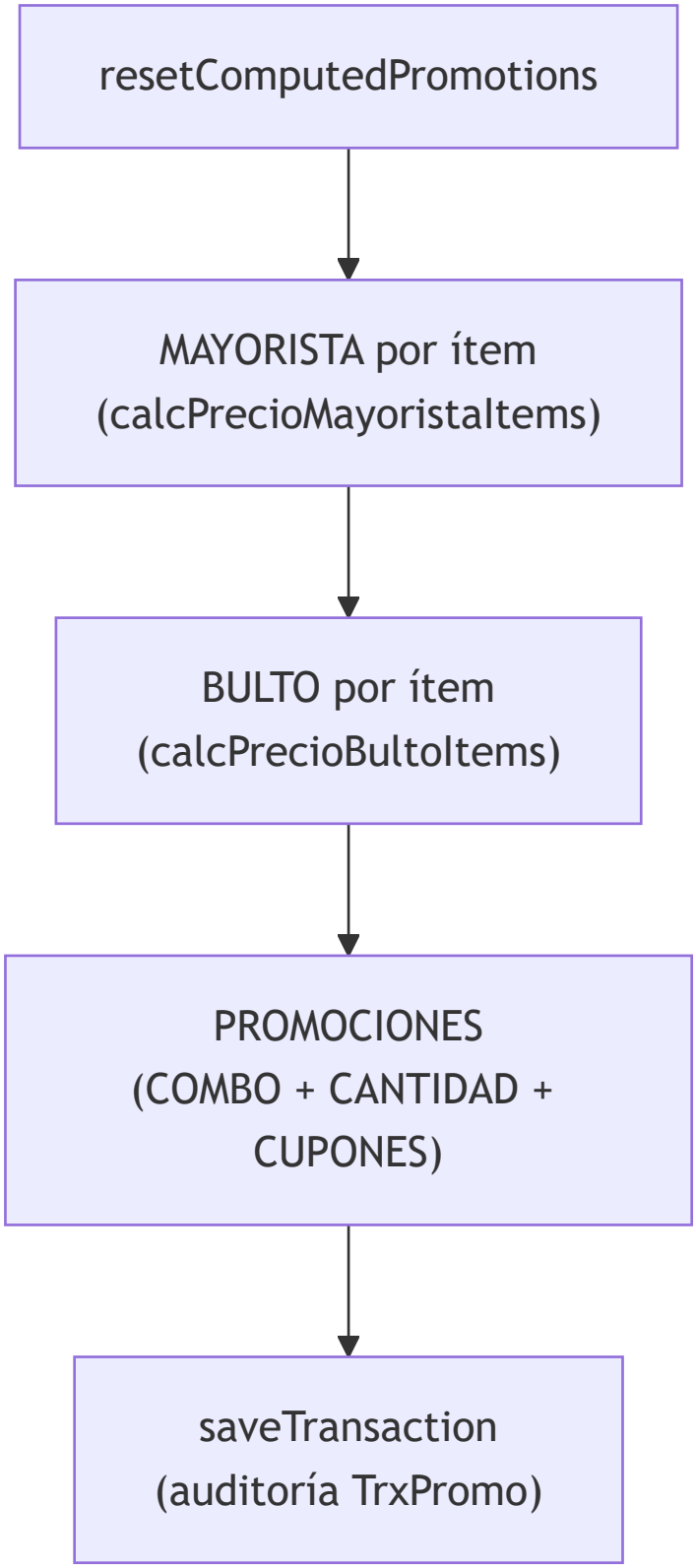
Las **listas** (`ListaDto`) definen qué artículos alcanza la promo por `tipoElemento`: `EAN`, `PLU`, `FAMILIA`, `AREA`, `RUBRO`, `MARCA`, `NEGOCIO`, `SECTOR`, `PROVEEDOR`, `CUPONES_TEXTO`, con `tipo` `INCLUSION` o `EXCLUSION`.

El motor: `calculatePromoTicket`

Punto de entrada (consumido por `tickets`):

```
public TicketPromoDto calculatePromoTicket(TicketPromoDto ticket, String
modoComputo)
```

Flujo:


resetComputedPromotions
MAYORISTA por ítem
(calcPrecioMayoristaItems)
BULTO por ítem
(calcPrecioBultoItems)
PROMOCIONES
(COMBO + CANTIDAD +
CUPONES)
saveTransaction
(auditoría TrxPromo)

Modos de cómputo: ITEMS vs MDP

Modo	Cuándo	Qué aplica
`ITEMS`	Mientras se arma el carrito.	Todas las promos **excepto** las que dependen del medio de pago.
`MDP`	Al elegir/abrir un medio de pago.	**Solo** las promos con `mediodepago` que coincida.

`tickets` resuelve el modo: si hay una orden de pago en estado `OPEN`, usa `MDP`; si no, `ITEMS`. Por eso una promo "pagando con tarjeta X, 10% off" recién aparece en el paso de pago.

MAYORISTA (escalas por volumen)

Para cada EAN, suma las unidades del ticket y busca el **nivel** mayorista con la mayor `cantidadminima` que esa cantidad alcanza. Si el precio mayorista mejora al vigente, el beneficio = `precioMayorista - precioVigente` (negativo), distribuido entre los movimientos. Motivos de no-aplicación auditados: sin mayorista para el EAN, no aplica a la sucursal, cantidad no alcanza, el precio no mejora.

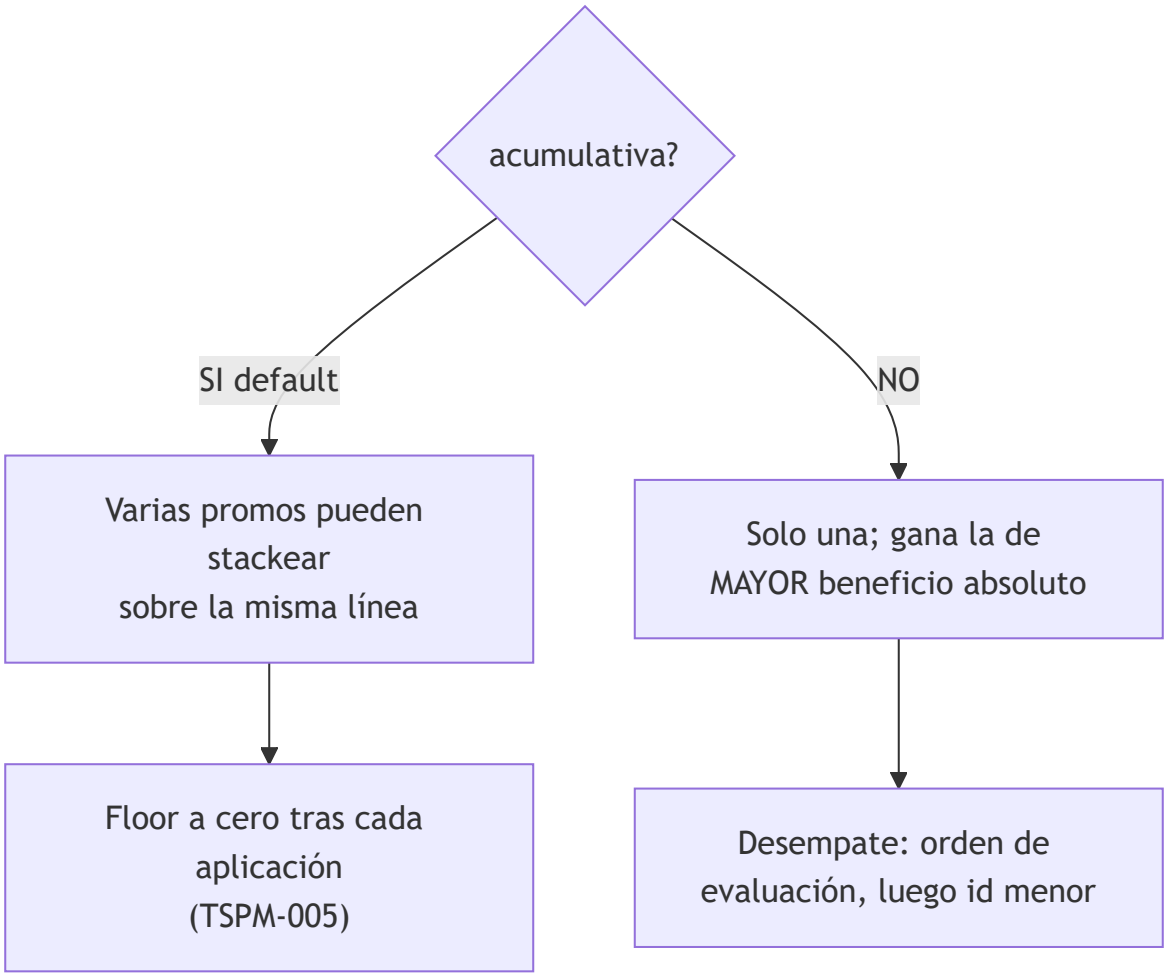
BULTO

Para artículos de bulto (DUN > 13 caracteres, `unidadesPorBulto > 0`), busca el bulto por DUN y elige el de **menor** `precioXBulto` que mejore el vigente.

COMBO y CANTIDAD

1. **Filtra promos elegibles**: beneficio no nulo, dentro de vigencia (fechas, horas, días de semana en zona Buenos Aires), sucursal/convenio coincidentes, y según el modo (ITEMS excluye MDP, y viceversa).
2. **Arma ocurrencias**: agrupa las líneas elegibles según las listas de inclusión (para COMBO, por grupo `nro`; para CANTIDAD, en bloques de `cantidadminima`), validando `montominimo`.
3. **Resuelve conflictos** (ver abajo).
4. **Distribuye el beneficio** proporcionalmente.

Acumulatividad: cómo se resuelven los conflictos



- `acumulativa = SI` (default): los descuentos se apilan; tras cada aplicación, el precio vigente **no puede bajar de cero** (fix TSPM-005).
- `acumulativa = NO`: compite con otras no-acumulativas; **gana la de mayor beneficio absoluto**; desempata por orden de evaluación y luego por id. Excluye las líneas que ya tienen una promo acumulativa.
- En COMBO acumulativo, se aplica iterativamente: en cada ronda gana el combo de mayor beneficio y se excluyen los que se solapan.

Distribución proporcional del beneficio

El beneficio total se reparte entre las líneas según su base:

```
ratio_línea = base_línea / Σ bases
porción_línea = beneficio_total × ratio_línea
última línea: absorbe (beneficio_total - Σ porciones)   // evita el centavo
perdido
```

Cada porción se aplica con `NucleoImpositivoDto.recomponer()` para preservar la composición fiscal (ver Núcleo impositivo).

Auditoría: `TrxPromo`

Cada cálculo persiste una fila en `TrxPromo` con el ticket **antes** (`jsonRequest`), **después** (`jsonResponse`), las **observaciones** de evaluación (por qué cada promo aplicó o se descartó) y el `modoComputo`. Es la caja negra para entender por qué un ticket recibió (o no) una promo.

Observaciones típicas: *"descartada por mayor beneficio"*, *"cantidadminima no alcanzada"*, *"lista.montominimo no alcanzado"*, *"exclusión por lista"*, *"precio mayorista no mejora el precio vigente"*.

Cache

`PromosCacheService` mantiene en memoria: promociones, mayoristas (y un índice por EAN), bultos (y un índice por DUN). Se carga al startup (`@PostConstruct`) y se refresca async por triggers: `STARTUP`, `MANUAL` (`POST /cache/refresh`), `SCHEDULED`, `UPDATE_COMPLETED`. Hay un endpoint de diagnóstico (`GET /promociones/diagnostico`) que compara DB vs cache y reporta promos inválidas.

Casos de borde y fixes conocidos

Fix	Problema → solución
TSPM-001	Descuento mayor que la base → se acota a la base (evita precio negativo).
TSPM-004	Acción nula se interpretaba como recargo → acción desconocida = DESCUENTO (seguro).

TSPM-005	Promos acumulativas llevaban el precio bajo cero → floor a cero tras cada aplicación.
TSPM-006	Beneficio `MONTO` negativo inflaba el descuento → clamp a cero.
TSPM-016	`recomponer()` dividía por cero con base cero → ítem con base cero no recibe descuento.
TSPM-018	Stack traces en la respuesta HTTP → mensaje genérico, detalle solo en logs.
TSPM-019	Colisión de ids de movimiento (`size()+1`) → `max(id)+1` .

Si una promo 'no aplica' y no entendés por qué

Mirá la fila de `TrxPromo` de ese ticket: las **observaciones** dicen exactamente qué condición falló. No adivines — el motor ya te dejó la traza.

Envases y vales

El cliente devuelve **envases** retornables (botellas, cajones) y a cambio recibe un **vale** (un documento de crédito con código y QR) que puede redimir en una compra. Este dominio vive en el módulo `envases` del backend y en las pantallas de envases del POS. Es el **paso 5** del cómputo del ticket.

> ⚠ **El código es la fuente de verdad**
>
> Algunos detalles (nombres de campo, longitudes) se reproducen del código a la fecha. Ante discrepancias, mandan `envases/` (backend) y `lib/.../envase`, `lib/.../vale` (POS).

Modelo de datos

`Vale` (auditado con Envers)

`envases/model/Vale.java`, tabla `Vales` en `TsEnvases`, anotada `@Audited`:

Campo	Rol
`id`	PK.
`nroVale`	Número correlativo **único**, generado por una **sequence atómica** (`dbo.seq_vales_nrovale`).
`codVale`	Código del vale; viaja **encriptado AES** al cliente.
`nroSucursal`	Sucursal origen.
`fechaCreacion` / `fechaVto`	Creación y vencimiento (`fechaVto = fechaCreacion + diasCaducidad`, default 365 días).

`terminalOrigen` / `userOrigen`	Quién lo emitió.
`estado`	`ValeEstado` (ver abajo).
`envases`	JSON serializado de la lista de envases que contiene.
`cantEnvases`	Total de envases en el vale.
`referencia`, `trxOrigen`, `trxModificacion`, `fechaModificacion`, `terminalModificacion`	Trazabilidad.

Por qué Envers

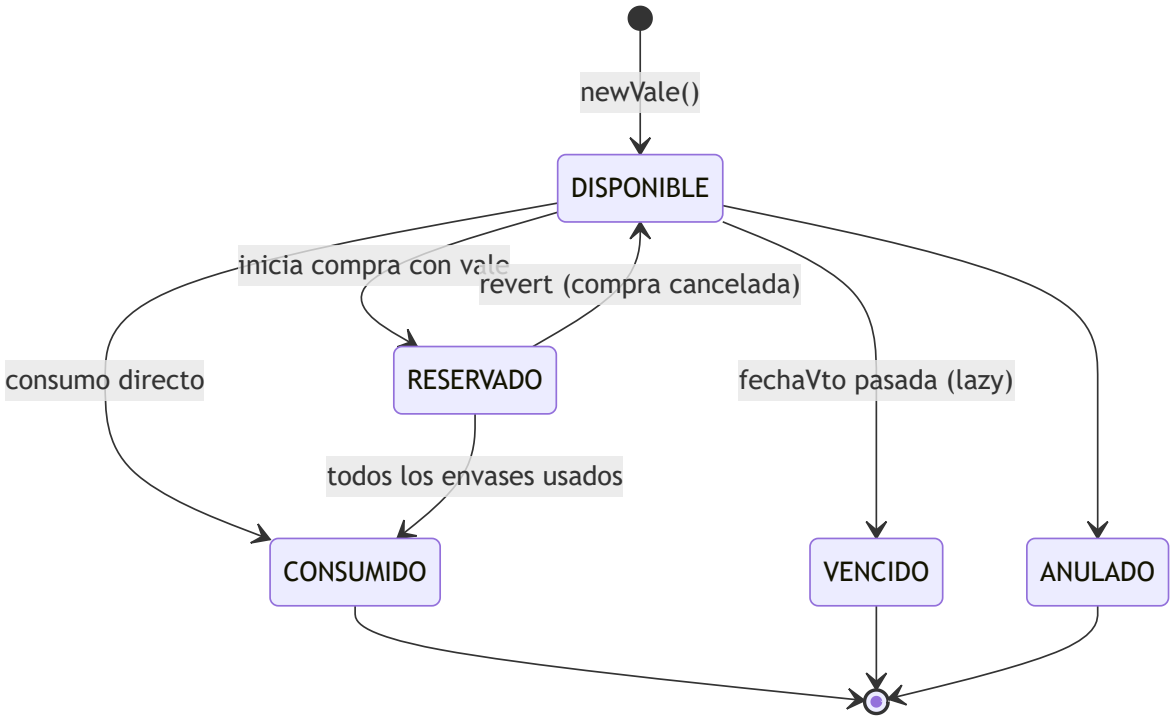
El vale es dinero. Cada cambio de estado genera una fila en `Vales_AUD` + `REVINFO`: se puede reconstruir el ciclo de vida completo (creación, reserva, consumo) para auditoría. No se borra ni se pisa historia.

`Envase`

`envases/model/Envase.java`, tabla `Envases`:

`ean` (clave de búsqueda), `plu`, `descripcion`, `nroSucursal`, `imagen` (`@Lob`, se sirve como Base64), `habilitado`, `orden`. Se asocia a un artículo del catálogo por EAN cuando el artículo tiene `bEnvase = true`.

Ciclo de vida del vale



`ValeEstado`: `DISPONIBLE`, `RESERVADO`, `CONSUMIDO`, `VENCIDO`, `ANULADO`. Las transiciones las valida `updateValeSafty` (en `tickets`):

- **`CONSUMIDO` y `VENCIDO` son terminales.**
- Persistir el **mismo** estado es OK (idempotencia, fix TSTK-009).
- En un **revert**, no se puede resucitar un vale `ANULADO` / `VENCIDO` (fix TSTK-010).

Control de vencimiento (lazy)

El vencimiento se evalúa **al leer** el vale: si está `DISPONIBLE` y `fechaVto` ya pasó, se marca `VENCIDO` y se persiste en el acto (`isValeVigente` → `getVale`). No hay un job que recorra todos; se resuelve en el momento del uso.

Numeración y código

`nroVale` atómico

Antes se hacía `findMaxNroVale() + 1`, con **race condition**: dos threads leían el mismo máximo y generaban el mismo número. El fix (`docs/db/manual/fix_vale_nrovale_dup.sql`) introdujo:

1. Una **sequence** `dbo.seq_vales_nrovale` (`SELECT NEXT VALUE FOR …` es atómico en la DB).
2. Un **unique index** `UX_Vales_nrovale` como defensa en runtime.

`codVale` encriptado (AES)

El `codVale` se guarda en claro en la DB pero viaja **encriptado AES** (`AES/ECB/PKCS5Padding`, Base64) al cliente. Cuando el cliente lo redime, lo manda encriptado y el backend lo desencripta para buscarlo (`GET /vales?codVale=…`). El motivo es ofuscación: que nadie lea ni adivine los códigos de vale de otros.

> **Detalle de implementación sensible**
>
> La clave AES vive en configuración (`app.security.aes-key`). Es un dato sensible: no lo hardcodees ni lo loguees. La desencriptación hace URL-decode primero (para `%2F`, `%2B`) antes de descifrar.

Cómputo de envases en el ticket

`EnvaseService.computTicketEnvase` (en `tickets`) determina, para cada ítem de tipo `ENVASE`, cuántos se **cobran** y cuántos se **consumen** contra vales:

```
cantEnvases          = Σ ítems tipo ENVASE (no anulados)
cantEnvasesConsumidos = envases emparejados con vales DISPONIBLE/RESERVADO
cantEnvasesCobrar     = max(0, cantEnvases - cantEnvasesConsumidos)   // TSTK-
011: clamp a cero
```

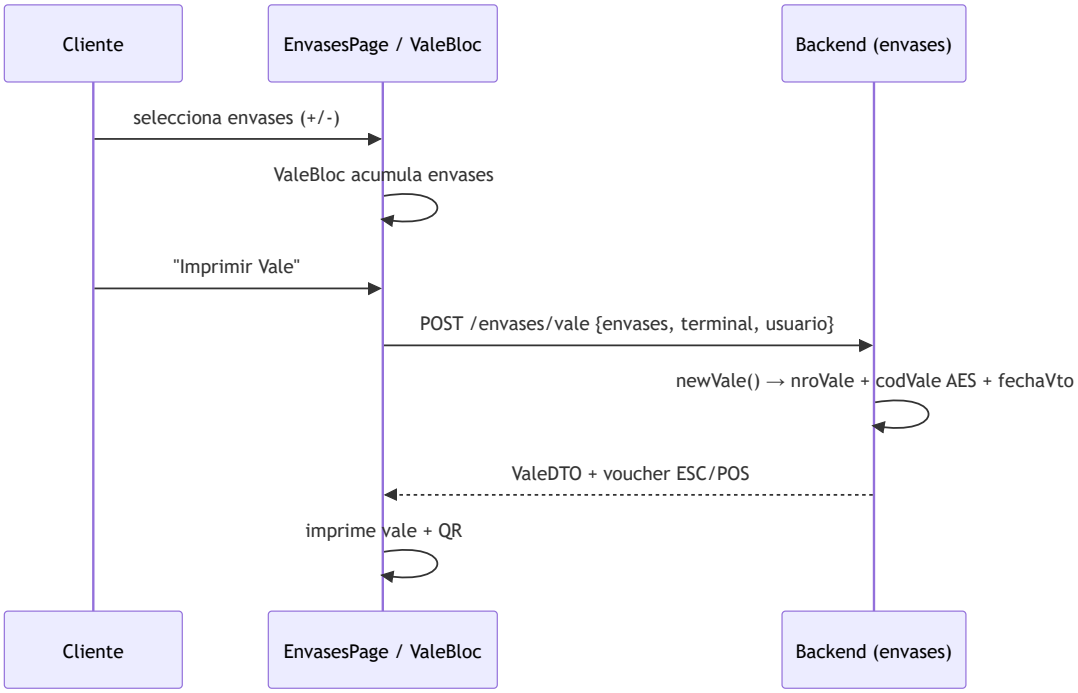
- Cada envase del ticket que matchea (por EAN) un envase de un vale lo **consume**; cuando un vale agota todos sus envases, pasa a `CONSUMIDO`.
- Si quedan envases del vale **sin usar** (saldo), se crea un **vale nuevo** con ese saldo (`createValeDiff`), que queda `DISPONIBLE` para la próxima compra.
- El crédito por envases impacta el núcleo impositivo del ticket como un movimiento de descuento.

Vales TOMRA (máquinas de terceros)

Los vales TOMRA (de máquinas recicladoras de terceros) son de **un solo uso**. `registrarRedencionTomra` inserta una fila en `tomra_redimido` con PK por operación; un segundo

intento viola la PK y se rechaza (`ValeEnvaseException: Vale TOMRA ya redimido`). Es la defensa anti-doble-redención.

Flujo en la terminal



- **Pantalla**: `EnvasesPage` (`lib/views/envases/`). `EnvaseBloc` carga los envases disponibles de la sucursal; `ValeBloc` acumula la selección y dispara `CreateVale` .
- **Redención**: en una compra, el cliente presenta el vale (QR/código); el ticket lo consume según el cómputo de arriba.
- **Casos de uso**: `TiprePOS/doc/documentation/CasosUsos/Autocompra_Modo_Envase.md` y `Autocompra_Modo_Autoservicio.md` .

Concurrencia y validaciones

- **Doble consumo**: `updateVale` usa lock pesimista (`findByCodValeForUpdate` , `SELECT … FOR UPDATE`). El segundo thread espera, lee el estado ya actualizado (`CONSUMIDO`) y se rechaza.
- **Transaccionalidad**: las operaciones de vale corren bajo el TransactionManager del datasource de envases, con `rollbackFor = Exception` .

Endpoints

`ValeController` (`/vales`): `GET /vales?codVale=` (desencripta y busca), `GET /vales/search` (filtros), `POST /vales` (alta), `PUT /vales` (cambio de estado), `GET /vales/last`. `EnvaseController` (`/envases`): `GET /envases?nroSucursal=`, `POST /envases`. Desde `tickets`, `EnvasesRestController` orquesta estas operaciones para el flujo de compra.

Fix	Qué resuelve
TSTK-009	Persistir el mismo estado del vale es idempotente, no error.
TSTK-010	No revertir a `DISPONIBLE` un vale `ANULADO` / `VENCIDO`.
TSTK-011	`cantEnvasesCobrar` con clamp a cero (no reembolsos espurios).
nroVale dup	Sequence atómica + unique index.

Facturación fiscal (AFIP)

El dominio más crítico y subdocumentado del sistema. Acá se decide cómo un ticket cobrado se convierte en un **comprobante fiscal autorizado por AFIP**, qué pasa cuando AFIP no responde, y cómo el sistema garantiza que **ningún pago quede en el limbo**. Vive en el módulo `tickets`. Es el **paso 7** del cómputo del ticket.

> ⚠ **El código es la fuente de verdad — y algunas cosas están planeadas**
>
> Reproducimos el flujo real a la fecha, incluidos fixes (TSTK-016, TSTK-024, P0-1, P0-3, INT-002). Donde algo está modelado pero **no implementado todavía**, lo marcamos explícitamente.

Tipos de comprobante

`TipoComprobante` modela 9 comprobantes (3 letras × factura / nota de débito / nota de crédito), cada uno con su código AFIP:

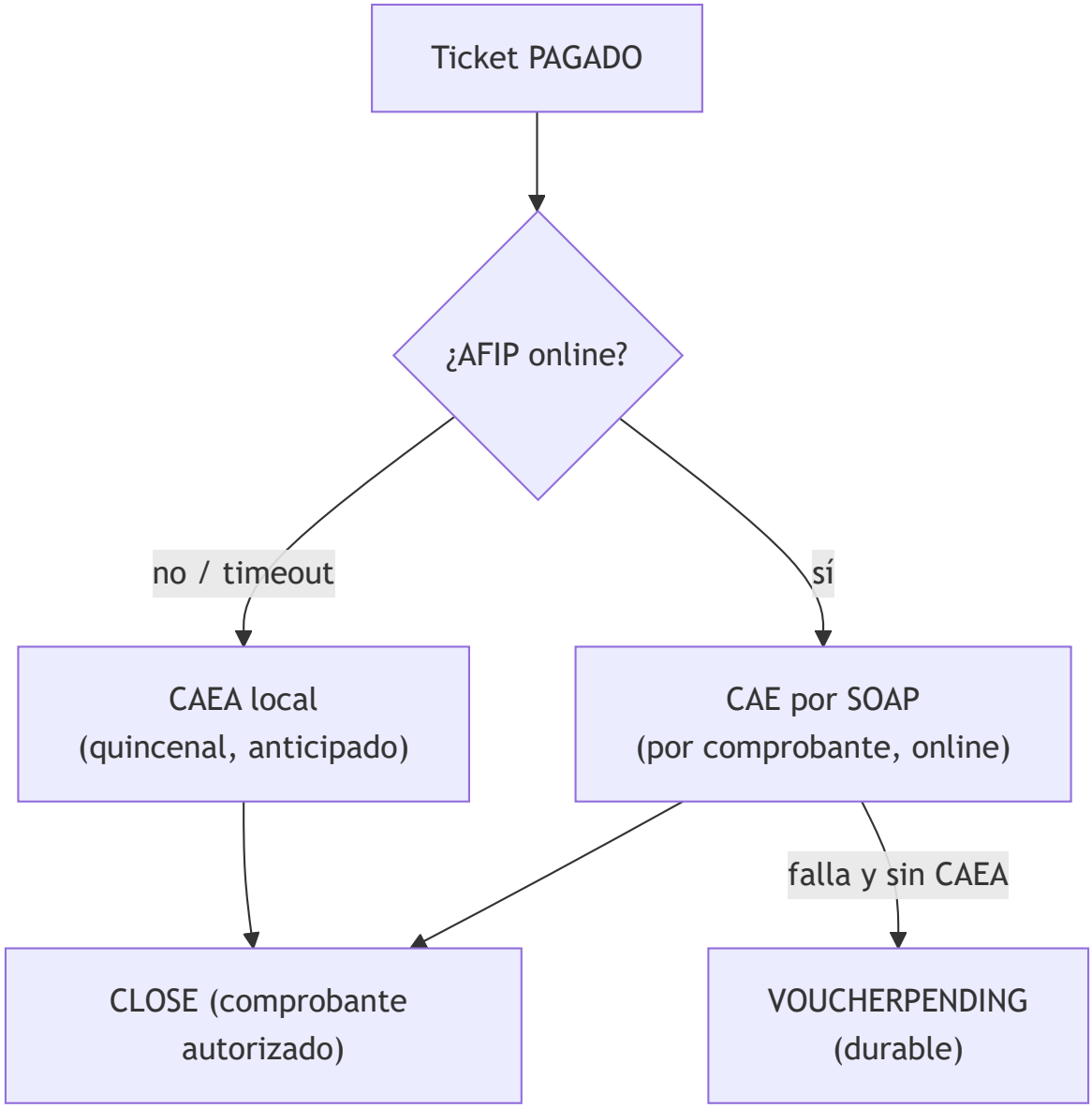
Letra	Factura	Nota de débito	Nota de crédito	Para
A	`FACA` (001)	`NDBA` (003)	`NCRA` (002)	Comprador Responsable Inscripto.
B	`FACB` (006)	`NDBB` (008)	`NCRB` (007)	Consumidor final, monotributo, exento.
C	`FACC` (011)	`NDBC` (013)	`NCRC` (012)	Operaciones no gravadas / sujeto monotributo emisor.

La **numeración fiscal** es por POS y por tipo: la entidad `Pos` guarda contadores separados (`nroFactA`, `nroFactB`, `nroFactC`, `nroNcr*`, `nroNdb*`) más el punto de venta AFIP (`ptoVta`, y opcionalmente `ptoVtaCae` para CAEA). El número se obtiene bajo **lock pesimista** (`SELECT … FOR UPDATE`) e incremento transaccional (fix P0-1).

Condición IVA del comprador

`TipoResponsableIva` modela 11 categorías AFIP (con su id): `RI` (Responsable Inscripto), `CF` (Consumidor Final), `MN` (Monotributo), `EX` (Exento), `MS`, `NC`, `PE`, `CE`, `LI`, `NA`, `MP`. La condición del comprador determina **la letra** del comprobante (A requiere RI), **si se calcula IVA** y el id de receptor que se manda a AFIP.

CAE vs CAEA: las dos vías de autorización



CAE — Código de Autorización Electrónico (online, SOAP)

`CaeService` + `CaeSoapClient` autorizan **un comprobante a la vez** contra AFIP por **SOAP 1.2**:

1. **Consulta el último autorizado** (`feCompUltimoAutorizado`) del punto de venta + tipo.
2. **Reuso (TSTK-016)**: si el último autorizado coincide con **este** ticket (importe + fecha), reutiliza su CAE en vez de pedir uno nuevo. Evita comprobantes huérfanos cuando una respuesta SOAP previa se perdió por timeout.

3. **Solicita el CAE** (`fecaeSolicitar`) con el XML del comprobante: ente facturador (CUIT, razón social, condición IVA del vendedor), cliente, **detalles de cada línea** (cantidad, EAN, neto, impuesto interno, tipo IVA), resumen de **alícuotas IVA** y tributos.
4. **AFIP devuelve** `cae` (14 dígitos) y `caeFchVto` (vencimiento del CAE).

Cada llamada SOAP se persiste **completa** (request + response) en `TrxDetalle` para auditoría.

CAEA — Autorización Electrónica Anticipada (local, quincenal)

El **CAEA** es un código que AFIP otorga **por adelantado** para una **quincena** (días 1–15 → orden 1; 16–fin → orden 2). Se obtiene en la madrugada (fuera de horario de venta) vía `CaeaApiClient` y se guarda en la entidad `Comercio` (`caea1` , `caea1desde` , `caea1hasta` , `caea1topeinfo` , `caea1proceso`).

Ventaja: **no depende de que AFIP esté online en el momento de la venta**. `getValidCAEA` valida que la fecha esté dentro de la vigencia de la quincena. El **tope de informe** (`fchTopeInf`) se chequea (fix TSTK-015): si venció, se alerta por log pero **no se frena la venta**.

> **Atomicidad del CAEA (P0-3)**
>
> La persistencia del CAEA para todos los comercios es **una sola transacción** (`@Transactional(rollbackFor=Exception)` , invocada vía proxy con self-injection). Un fallo a mitad dejaría comercios con CAEA viejo y la cache desincronizada — y el CAEA autoriza **todas** las facturas de la quincena.

VOUCHERPENDING: pagado pero sin comprobante

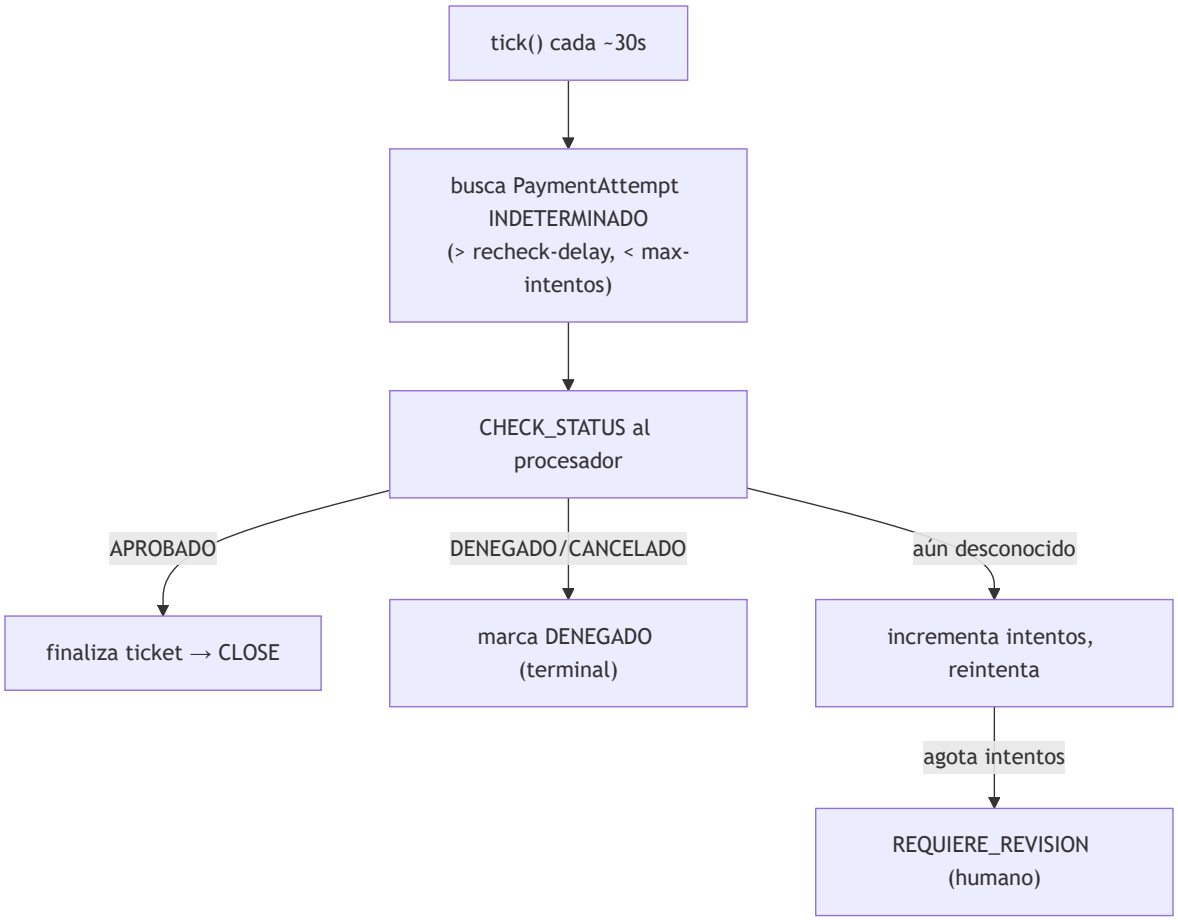
Un ticket entra en `VOUCHERPENDING` cuando el pago **ya fue aprobado** (la plata está cobrada) pero el comprobante **no se pudo autorizar** todavía: timeout SOAP, AFIP caído sin CAEA disponible, o error de datos que requiere revisión.

> Un `VOUCHERPENDING` es una transacción **durable** que espera reconciliación. La plata está; falta el papel fiscal.

Sale de ese estado por dos caminos: el **reconciliador** reintenta y autoriza (→ `CLOSE`), o un operador corrige y reintenta manualmente.

El reconciliador de pagos

`PaymentReconciliador` es un job `@Scheduled` que resuelve los pagos que quedaron **indeterminados** (INT-002) sin depender de que el POS esté poleando.



Estados del intento (`EstadoPagoIntento`): `INICIADO`, `CON_INTENTO`, `APROBADO`, `DENEGADO`, `INDETERMINADO`, `REQUIERE_REVISION`. Solo `APROBADO` y `DENEGADO` son **terminales**.

La identidad del intento es el `paymentAttemptId` (PK, 64 chars), el mismo que genera la terminal para idempotencia de pago. Reintentar el mismo pago **no crea** una fila nueva: actualiza la existente.

Configuración: `app.reconciler.delay-ms` (cada cuánto corre), `recheck-delay-ms` (cuánto espera antes de reconciliar un intento), `max-intentos`, `check-timeout-ms`.

REQUIERE_REVISION es alerta humana

Un intento que agota los reintentos pasa a `REQUIERE_REVISION`: el sistema no pudo decidir solo si se cobró o no. **Eso necesita una persona.** El Cockpit lo muestra en el panel fiscal; no lo ignores.

QR fiscal

`AfipUtil.getQrAfip` arma la URL del QR fiscal de AFIP (escaneable, validable por cualquiera con el teléfono). Codifica en Base64 un JSON con: versión, fecha, CUIT del comercio, punto de venta, tipo y número de comprobante, importe, moneda, documento del receptor, y el código de autorización con su tipo (`A` = CAEA, `E` = CAE). El QR se renderiza con ZXing y va al voucher (ver El agregado Ticket → comprobante).

Panel fiscal del Cockpit

`monitoring` accede a este dominio **solo** vía `FiscalStatusFacade` (modulith-safe, sin tocar internals). `GET /monitoring/fiscal` expone un snapshot:

Campo	Qué muestra	Acción si está mal
`serviceStatuses[AFIP]`	CAE online/offline	Offline → fallback a CAEA.
`serviceStatuses[CAEA]`	CAEA disponible	Offline → **no se pueden autorizar comprobantes**.
`voucherPendingCount`	Tickets cobrados sin comprobante	> 0 → el reconciliador está trabajando.
`reconciliadorBacklog.indeterminado`	Pagos a reconciliar	El reconciliador reintenta.
`reconciliadorBacklog.requiereRevision`	Agotaron reintentos	**Intervención humana.**
`caeaVigente.topeInforme`	Tope de informe de la quincena	Pasado → AFIP no acepta más con ese CAEA.

Patrones y reglas

- **Zona horaria**: la quincena del CAEA y la fecha del comprobante se calculan en `America/Argentina/Buenos_Aires`, no en la zona de la JVM (fix TSTK-024).

- **Transaccionalidad**: numeración fiscal y persistencia de CAEA usan `@Transactional(rollbackFor=Exception)` + locks pesimistas.
- **Modulith**: `monitoring` solo ve `FiscalStatusFacade` y `TicketAggregationService`; nunca `tickets.service` / `tickets.repository`.

Lo que todavía no existe

Según el código actual, está **modelado pero no implementado**:

1. **Notas de crédito/débito de anulación**: `VoucherType.ANULACION_PAGO` existe, pero generar la nota fiscal de anulación no está desarrollado.
2. **Percepciones y retenciones**: los tipos están en `NucleoComponenteTipo`, pero `CaeService` aún no arma esos tributos en la solicitud SOAP.
3. **Multi-comercio en CAEA**: hay TODOs sobre soportar múltiples comercios.
4. **Push al POS desde el reconciliador**: el reconciliador marca el intento `APROBADO`, pero el aviso al POS para que actualice su estado está pendiente.

Para el contexto conceptual de todo este ciclo (pago indeterminado, durabilidad, por qué el ticket no se cierra antes de tiempo), leé El ciclo de pago y fiscalización.

API REST

Referencia de los endpoints del backend. Todo cuelga del context-path `/autocompras/v1`. La terminal consume las rutas de ticket/pago; el Cockpit consume `/monitoring/*` (documentado aparte en Cockpit).

> **No hay STOMP**
>
> El backend actual **no expone WebSocket/STOMP**: no hay `@MessageMapping` ni broker. La terminal habla REST. Si buscás el porqué, está en Comunicación POS ↔ Backend.

Convenciones

- **Base**: `/autocompras/v1` (más la ruta de cada controller).
- **Healthchecks**: cada módulo expone un `GET /*/dummy` (`/tickets/dummy`, `/articulos/dummy`, `/pagos/dummy`, `/promociones/dummy`, `/vales/dummy`), todos `permitAll`.
- **Envoltura**: muchas respuestas usan `ResponseMessage` (del kernel `shared`).
- **Headers de terminal** (los inyecta el `HttpCliente` del POS): `X-Cod-Terminal`, `X-Terminal-Uuid`, `X-Device-Health`.

Tickets (orquestador)

`TicketRestController` y compañía. Es la API que mueve el carrito.

Método	Ruta	Qué hace
`GET`	`/tickets/dummy`	Healthcheck.
`POST`	`/openTicket`	Crea un ticket (`OPEN`).

`POST`	`/agregarArticulo`	Agrega un ítem (resuelve EAN en `catalogo`, recalcula promos).
`POST`	`/removeItem`	Quita un ítem.
`POST`	`/changeStatus`	Cambia el estado del ticket.
`POST`	`/validateTicket`	Valida el ticket.
`POST`	`/closeTicket`	Cierra el ticket (`CLOSE`).
`POST`	`/payTicket`	Dispara el flujo de pago (intent/ejecución). Ver Pagos.

El cliente: `TicketService` en la terminal

En TiprePOS cada una de estas mutaciones tiene su método en `TicketService` (`openTicket`, `agregarArticulo`, `removeItem`, `closeTicket`, `changeStatus`, `validateTicket`), despachado por el `BackendClientBloc` vía el evento `SendMessage`.

Catálogo

`ArticuloRestController`.

Método	Ruta	Qué hace
`GET`	`/articulos/dummy`	Healthcheck.
`GET`	`/searchBy?ean=&sucursal=&page=&limit=`	Busca artículos (también por `id`, `codigoInterno`, `descripcion`). Paginado, **máx. 200 ítems**.
`POST`	`/catalogo/cache/refresh`	Refresca la cache Caffeine. **Rol admin** si la seguridad

	está activa.

Pagos

`PagoRestController`. Stateless; sale a los gateways.

Método	Ruta	Qué hace
`GET`	`/pagos/dummy`	Healthcheck.
`POST`	`/payment-intent`	Crea la intención de pago. **Async** (`Mono<ResponseEntity>`).
`POST`	`/pay`	Procesa el pago.
`GET`	`/check-status/{paymentId}`	Verifica el estado de un pago.

Promos

`PromoController`.

Método	Ruta	Qué hace
`GET`	`/promociones/dummy`	Healthcheck.
`GET`	`/promociones`	Lista promociones.
`POST`	`/promociones/update`	Actualiza promociones. **Rol admin.**
`POST`	`/cache/refresh`	Refresca la cache de promos. **Rol admin.**

Envases / Vales

`ValeController` (`@RequestMapping("/vales")`) y `EnvaseController`.

Método	Ruta	Qué hace
`GET`	`/vales/dummy`	Healthcheck.
`GET`	`/vales?codVale=`	Obtiene un vale por su código (encriptado).
`POST`	`/vales`	Crea un vale.
`PUT`	`/vales/{id}`	Actualiza un vale.

Salud / Actuator

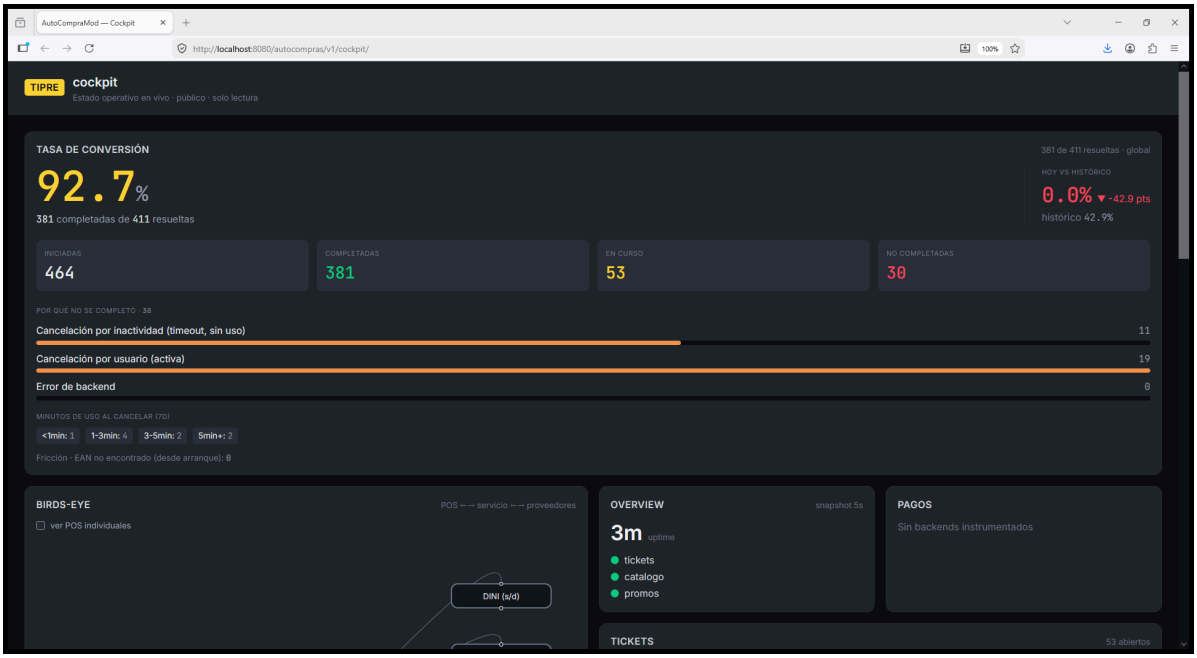
Método	Ruta	Qué hace
`GET`	`/actuator/health`	Health check (Actuator). `permitAll`.
`GET`	`/actuator/info`	Info de build. `permitAll`.
`GET`	`/status`	Estado de servicios que la terminal consulta por polling (60 s / 15 s).

Rutas implícitas

Algunas rutas (`pay`, `check-status`, alta/edición de vales) se infieren del patrón de cada controller y pueden variar en la firma exacta. Ante la duda, **el código del controller es la fuente de verdad**, no esta tabla: las rutas viven en `src/main/java/com/tipre/autocompras/*/controller/`.

Cockpit (panel de operación)

El **Cockpit** es el panel de observabilidad del ecosistema: muestra la **conversión** del autoservicio, el parque de terminales, la salud del backend, los tickets, los errores (por zona), los pagos y el estado fiscal. Es una SPA React que vive en `cockpit-ui/` y la sirve el propio backend; los datos salen del módulo `monitoring`.



El Cockpit actual: el KPI de conversión (92.7% en oro) con las cards iniciadas/completadas/en curso/no completadas, el desglose "por qué no se completó" con sus labels, los "minutos de uso al cancelar" y la fricción EAN. Abajo arrancan el birds-eye, el overview (semáforos verdes) y pagos.

Cómo está montado

- **Frontend**: `cockpit-ui/` — React 18 + TypeScript + Vite + Tailwind, con Recharts (gráficos) y ReactFlow (el "birds-eye").
- **Tema "Binance-dark"** (`cockpit-ui/tailwind.config.js`): fondo near-black (`#0b0e11`), superficies `#1e2329` / `#2b3139` , acento **oro** (`#fcd535`) y semántica de trading **verde** (`#0ecb81` = OK/sube) / **rojo** (`#f6465d` = error/baja). Tipografía Inter (texto) + JetBrains Mono (números).

- **Build integrado**: el profile `frontend` de Maven instala Node local (`frontend-maven-plugin`), corre `npm ci && npm run build` y copia `dist/` a `src/main/resources/static/cockpit`. No es un deploy aparte.
- **Servido en**: `/autocompras/v1/cockpit/` — `CockpitController` hace fallback a `index.html` para el routing client-side.
- **Datos**: el cliente HTTP (`cockpit-ui/src/api/client.ts`) pega contra `/autocompras/v1/monitoring/*`, todo `GET`, sin auth. En dev, Vite proxyea a `http://localhost:8080/autocompras/v1`.

> **Snapshots, no queries por request**
>
> El módulo `monitoring` calcula **snapshots periódicos** con `@Scheduled` (refresco cada ~60 s) en vez de consultar la base en cada request del Cockpit. La instrumentación es **aditiva y no bloqueante**: observar el sistema no lo frena.

El KPI de conversión

Es el indicador estrella (plan 015): **¿qué porcentaje de las sesiones de autoservicio termina en una venta?**

```
conversión = completadas / (completadas + canceladas + con error)
```

- **Completadas** = `CLOSE` + `PAGADO` + `VOUCHERPENDING` (operaciones que llegaron a buen puerto).
- **Resueltas** (denominador) = completadas + canceladas (usuario e inactividad) + errores de backend. Los tickets `OPEN` (en curso) **no** entran; `TEST` / `ANULADO` tampoco.

Lo calcula `MetricsSnapshotService.computeConversion()` y viaja en el bloque `conversion` de `GET /monitoring/tickets` (record `MetricsSnapshot.Conversion`: `rate`, `iniciadas`, `completadas`, `canceladasUsuario`, `canceladasInactividad`, `conError`, `enCurso`, `rateToday`, `rateBaseline`, `direction`, `cancelacionPorMinutos`).

El `ConversionPanel.tsx` lo muestra como **número grande en oro**, con la tendencia de hoy vs. el baseline (verde si sube, rojo si baja), el desglose de **por qué no se completó**, y el histograma de **minutos de uso al cancelar**.

Por qué no se completó (motivos de cancelación)

El Cockpit distingue los motivos y los etiqueta:

Estado	Label en la UI
`CANCELED_INACTIVITY`	"Cancelación por inactividad (timeout, sin uso)"
`CANCELED_USER`	"Cancelación por usuario (activa)"
`ERROR`	"Error de backend"

Minutos de uso al cancelar

Para entender *cuándo* abandona la gente, las cancelaciones se **bucketizan** por cuánto duró la sesión antes de cancelar (`TrxRepository.canceladasDuracionSince` → `bucketizeCancelDurations()` en Java, ventana 7 días):

- `<1 min` — abandono inmediato.
- `1-3 min` — uso muy breve.
- `3-5 min` — uso moderado.
- `5 min+` — uso sostenido (canceló tarde).

Fricción: EAN-no-existe

Aparte de la conversión está la **fricción**: cuando un cliente escanea un EAN que el catálogo no resuelve. **No** baja la conversión (el cliente re-escanea y cierra igual), pero es señal de catálogo incompleto. Se cuenta con un counter Micrometer `selfservice.ean_not_found{sucursal}` (incrementado en `ArticuloService`) y se expone en `GET /monitoring/friction` → `{ eanNotFound: { total, bySucursal } }`.

Endpoints `/monitoring/*`

Endpoint	Controller	Qué devuelve
`GET /monitoring/overview`	`MetricsController`	Uptime, versión, **semáforos de subsistemas con señal**

		real (ver abajo).
`GET /monitoring/tickets?range=1d|7d|30d&codTerminal=`	`MetricsController`	Totales por estado, serie temporal, drill-down por POS, **KPI de conversión** y tendencia.
`GET /monitoring/friction`	`MetricsController`	**(nuevo)** EAN-no-existe por sucursal (acumulado desde el arranque).
`GET /monitoring/caches`	`MetricsController`	Hit rate, estado (`OK` / `EMPTY` / `STALE` / `DEGRADED`) y próximo refresh de cada cache.
`GET /monitoring/api-calls`	`MetricsController`	Top 50 endpoints por volumen — **excluye el tráfico propio del Cockpit** (`/monitoring/**`, `/cockpit/**`).
`GET /monitoring/errors?range=&page=&size=`	`ErrorsController`	Errores paginados + resumen por categoría y **por zona POS** (`byTerminal`).
`GET /monitoring/terminals`	`TerminalsController`	**Parque reconciliado** (configurados ∪ heartbeat) con device-health por terminal.
`GET /monitoring/payments`	`PaymentsFiscalController`	Conteos por backend de pago (DINI / MP), vía Micrometer.
`GET /monitoring/fiscal`	`PaymentsFiscalController`	AFIP, CAEA vigente, `VOUCHERPENDING`, backlog del reconciliador.

`GET /monitoring/jobs`	`JobsController`	Estado de cada `@Scheduled`: última corrida, resultado, duración, próxima, conteos 24 h.
`GET /monitoring/jobs/history?name=&range=`	`JobsController`	**(nuevo)** Historial paginado de un job puntual.

Semáforos del overview: señal real

Antes los semáforos de subsistemas estaban **hardcodeados** a "up" (herencia de los microservicios). Ahora (`MetricsController.overview()`) cada uno refleja una señal real:

Subsistema	Fuente	up	degraded	down
`tickets`	frescura del snapshot	≤ 180 s	> 180 s	—
`catalogo`	estado de la cache `articulos`	`OK`	`STALE` / `DEGRADED`	`EMPTY`
`promos`	estado de la cache `promociones`	`OK`	`STALE` / `DEGRADED`	`EMPTY`

En la UI (`OverviewPanel`): up → dot verde (OK), degraded → amarillo (WARN), down → rojo (ERROR).

El parque de terminales (reconciliado + heartbeat)

`GET /monitoring/terminals` no lista solo lo que pingeó: **reconcilia** las terminales configuradas con las que mandan heartbeat (`TerminalFleetService.snapshot()` = `pos` configuradas ∪ heartbeats vistos). Un POS puede estar en cuatro situaciones:

Estado	Significa
Online	heartbeat fresco (`online=true`).
Offline conocido	se vio antes, ahora sin ping (`lastSeen != null`).
Nunca visto	configurado en `pos` pero sin un solo ping desde el arranque (`lastSeen = null`).
Huérfano	pingea sin estar en la tabla `pos` (`configured=false`).

El parque configurado lo aporta `tickets` vía `PosFleetFacade` (en memoria, sin tocar la DB); si `tickets` no está, degrada con gracia (solo heartbeats). Un registro pasivo capta el ping de cada terminal en cada request; la ventana de "online" es ~90 s.

Device-health por POS (no global)

Cada terminal manda su salud de hardware en el header `X-Device-Health` (ej. `pinpad=OK;point=UNKNOWN`). El `TerminalHeartbeatFilter` lo parsea a un `Map<device,status>` y lo guarda en el heartbeat con **preserve-on-null** (un request sin el header no borra la última salud conocida). Esa salud es **por terminal**, no global.

> **Pinpad/Point salieron del birds-eye global**
>
> Antes el birds-eye mostraba "Pinpad: down" como si fuera un servicio global del sistema — era un falso negativo (el pinpad es hardware de **cada** POS). Ahora el pinpad/Point viven en el tile de cada terminal (chips verde OK / rojo OFFLINE / gris UNKNOWN), no en el diagrama global.

Errores por zona POS

Cada error capturado lleva el `cod_terminal` del request que lo originó: el `TerminalAuthFilter` lo pone en el MDC, el appender de Logback lo persiste en `error_event.cod_terminal`. `GET /monitoring/errors` agrega `byTerminal` (errores de las últimas 24 h agrupados por terminal; los que ocurren fuera de un request POS quedan como `SIN_TERMINAL`). El `ErrorsPanel` muestra las

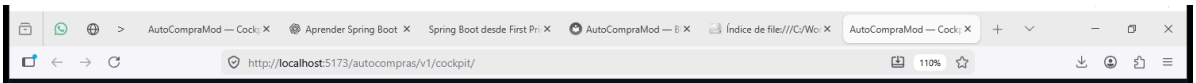
zonas con más errores y la lista filtrable por categoría (`FISCAL` , `PAGO` , `TERMINAL_AUTH` , `VALE_ENVASE` , `DB` , `OTRO`).

Paneles de la UI

Bajo `cockpit-ui/src/panels/` :

Panel	Muestra
`ConversionPanel`	**(nuevo)** KPI de conversión (héroe), tendencia, motivos de no-completar, minutos al cancelar, fricción EAN.
`OverviewPanel`	Uptime y semáforos de subsistemas (señal real).
`BirdsEye`	Diagrama interactivo (ReactFlow); con "ver POS individuales" muestra tiles con device-health por terminal.
`TerminalsPanel`	Parque de POS (online/offline/nunca-visto/huérfano).
`TicketsPanel`	Tickets por estado, tendencia y drill-down por POS.
`CachesPanel`	Caches: tamaño, hit rate, estado y próximo refresh.
`ApiCallsPanel`	Top 50 endpoints (sin el tráfico propio del Cockpit).
`ErrorsPanel`	Errores por zona POS + lista filtrable por categoría.
`PaymentsPanel`	Conteos por backend de pago.
`FiscalPanel`	AFIP, CAEA, `VOUCHERPENDING` , reconciliador.

`JobsPanel`	Jobs `@Scheduled` con nombres amigables y estado color-coded (verde OK / rojo error).



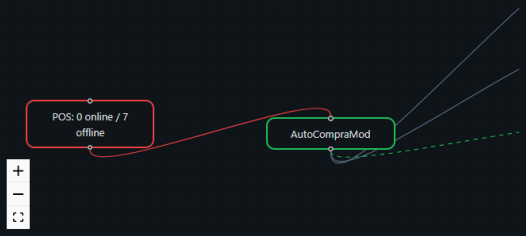
AutoCompraMod · Cockpit

Estado operativo en vivo · público · solo lectura

BIRDS-EYE

POS ←→ servicio ←→ proveedores

ver POS individuales



OVERVIEW

snapshot 52s

8m uptime

- tickets
- promos
- catalogo

PAGOS

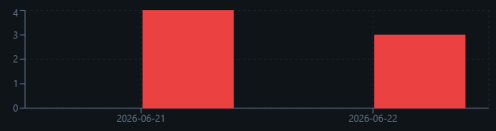
Sin backends instrumentados

TICKETS

53 abiertos

1d | 7d | 30d

53	381	28
Abiertos	Cerrados	Cancelados



Cerrados Cancelados

Cerrados: **3** (prev 378) Cancelados: **7** (prev 21) tendencia: down

POR POS (7)

buscar POS...

POS	Abiertos	Cerrados	Cancelados	Errores	Total ▼
DINO99999	36	211	7	0	254
DINO99888	13	120	10	0	143
DINO99555	3	19	10	0	32
DINO99444	1	18	0	0	19
DINO99666	0	7	0	0	7
PC-SANTI	0	4	1	0	5
AEPTEST01	0	2	0	0	2

CACHES

7 caches

Cache	Entradas	Estado	Últ. refresh	Próx. refresh
articulos	83.901	OK	hace 8m	en 3m
promociones	11.588	OK	hace 8m	en 3m
mayoristas	264	OK	hace 8m	en 3m
mayoristasPorEan	261	OK	hace 8m	en 3m
bultos	15	OK	hace 8m	en 3m
bultosPorDun	14	OK	hace 8m	en 3m
envaseCache	0	OK	—	—

API CALLS

top 8

URI	Método	Status	Count

FISCAL (AFIP)

AFIP	offline
CAEA vigente	202606 Q2
VOUCHERPENDING	1
Reconciliador: indeterminado	0
Reconciliador: requiere revisión	0

JOBS PROGRAMADOS

11 jobs

Job	Cada	Resultado	Última	Próxima	24h
Cache de artículos · refresh	cada 30 min	OK	hace 1h · 1ms	en 3m	6
CAEA (AFIP)	3:00 AM	sin correr	—	en 12h	0
Drainer de errores	cada 5 s	OK	hace 1s · 0ms	—	2171
Retención de errores (30d)	3:00 AM	sin correr	—	en 12h	0
Retención de jobs (30d)	3:00 AM	sin correr	—	en 12h	0
Snapshot del cockpit	cada 1 min	OK	hace 52s · 4ms	—	186
Reconciliador de pagos	cada 30 s	OK	hace 22s · 1ms	—	368
Promos · update (origen→DB)	cada 30 min	OK	hace 1h · 2ms	en 3m	6
Cache de promos · refresh	cada 30 min	OK	hace 1h · 1ms	en 3m	6
Estado de servicios	cada 1 min	sin correr	—	—	0
Limpieza de sesiones POS	cada 15 s	sin correr	—	—	0

ERRORES

1 última hora

1d | 7d | 30d

OTRO: 17

2026-06-22T17:49:14.665Z [OTRO] HHH015007: Illegal argument on static metamodel field injection : org.hibernate.envers.DefaultRevisionEntity_#class_; expected type : org.hibernate.metamodel.model.domain.internal.EntityTypeImpl; encountered type : jakarta.persistence.metamodel.MappedSuperclassType

Los paneles inferiores (esta captura es previa al tema Binance-dark, pero el layout se mantiene): birds-eye, overview, tickets, caches (hit rate por cache), fiscal AFIP (CAEA, VOUCHERPENDING, reconciliador), jobs y el log de errores.

Generar carga para verlo en acción

El Cockpit muestra el estado, pero no genera carga. Para estresar el sistema y ver los paneles moverse (conversión, indeterminados drenando, latencias), está StressBench: 200 POS simulados vendiendo a la vez contra el backend real.

> **Si una terminal aparece offline**
>
> No significa que se borró: no pingeó dentro de la ventana de ~90 s, o está configurada pero nunca pingeó ("nunca visto"). Revisá su conectividad y su último heartbeat antes de asumir que está mal configurada.

POS TiprePOS (Flutter)

Referencia técnica de la terminal de autoservicio: stack, arranque, routing, BLoCs, servicios y persistencia. Vive en el repo `TiprePOS`. Esta es la **vista general**; el detalle de cada subsistema está en sus páginas:

- Flujo de compra (scan → checkout) — escaneo, carrito, validación, DTOs del ticket.
- Subsistema de impresión — 4 tipos de conexión, monitoreo, reconexión, impresión en lote.
- Operación, sesión y configuración — inactividad, registro, modo supervisor, branding, gestión, errores.
- Pagos — QR, Point Smart, tarjeta.

La terminal real (build de Windows) en su pantalla de conexión, con el branding de un comercio (SUPER MAMI / DINO) aplicado por el theming por comercio y los badges de identidad de la terminal (`AUTOCOMPRA`, `DINO99999`, versión de config).

Stack

Pieza	Valor
Lenguaje	Dart `>=3.0.0 <4.0.0`
Framework	Flutter `>=3.35.6`
Estado	`bloc` + `flutter_bloc`
Navegación	`go_router`
Persistencia local	`isar_community` (DB `auto_compra_db`)
Red	`http` (REST a `/autocompras/v1`)
Conectividad	`connectivity_plus`
Desktop	`window_manager` (la terminal corre en escritorio)
Impresión	`ti_printer_plugin` + servicios propios (USB/BT/Red/Serial)

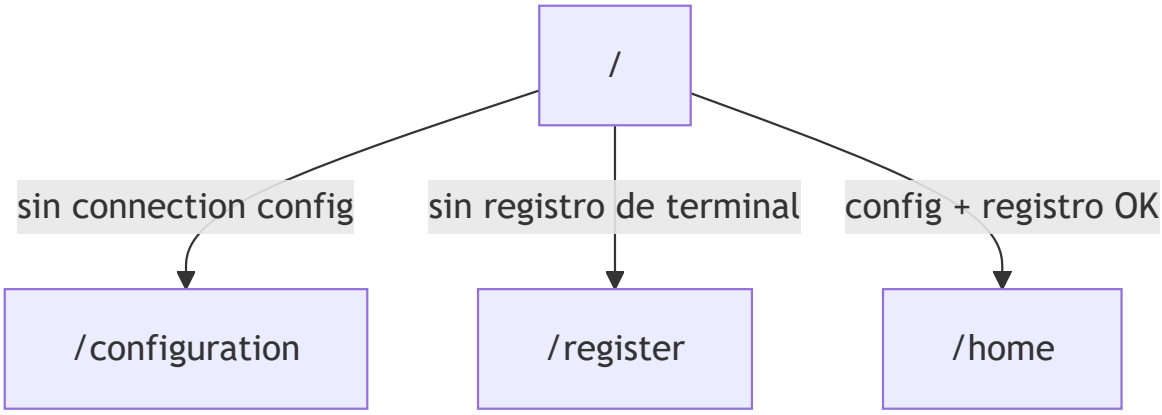
Arranque

La secuencia de init vive en `lib/main.dart` y `lib/auto_compra_app.dart`:

1. **`main()`** corre dentro de `runZonedGuarded()` (captura global de errores). Inicializa `AppLogger`, abre Isar (`auto_compra_db`), recupera pagos in-flight pendientes (`PendingPaymentService`, fix INT-002) y configura `WindowManagerService` para desktop (incluye manejo de sesiones RDP con fullscreen forzado).
2. **`AutoCompraApp`** crea servicios, repositorios y BLoCs, e inyecta todo con un `MultiBlocProvider`.
3. **Init secuencial de BLoCs** (`_initializeBlocsSequentially`): `LoadConnectionConfiguration` → `LoadTerminalConfiguration` → `RegisterTerminal` (si hay conexión) → `StartNetworkMonitoring`.

Routing (`go_router`)

Configurado en `lib/routing/app_router_builder.dart`, con `refreshListenable: AppStateNotifier` (reactivo al estado global) y una **lógica de redirect** que hace de gate:



Rutas principales:

Path	Pantalla	Notas
`/`	Initialize / Splash	Decide el destino.
`/configuration`	`ConfigurationPage`	Server, puerto, SSL, código de terminal.
`/register`	`RegisterPage`	Registra la terminal contra el backend.
`/home`	`HomePage`	Portal central.
`/scan`	`ScanPage`	Escaneo de productos / carrito.
`/envases`	`EnvasesPage`	Envases y vales.
`/mediosPago`	`MediosPagoPage`	Selector de medio de pago.

`/pagoQR`	`PagoQRPage`	Pago QR (timeout ~210 s).
`/pagoPointSmart`	`PagoPointSmartPage`	Point Smart (timeout ~90 s).
`/pagoTarjeta`	`PagoTarjetaPage`	Tarjeta vía ApiCard.
`/planPago`	`PlanPagoPage`	Cuotas.
`/resultPayment`	`ResultPaymentPage`	Resultado del cobro.
`/settings`	`SettingsPage`	Parámetros de UI.
`/managementTicket`	`ManagementTicketPage`	Historial de tickets.
`/managementVale`	`ManagementValePage`	Gestión de vales.
`/managementPrinter`	`ManagementPrinterPage`	Config de impresora.
`/error`	`ErrorPage`	Manejo de errores.

Las rutas protegidas no son accesibles sin completar configuración + registro.

BLoCs principales

Bajo `lib/viewmodels/`:

BLoC	**Responsabilidad**
`ConfigurationBloc`	Carga/persistencia de config de conexión y de terminal.
`BackendClientBloc`	**Centro de la comunicación REST**: despacha mutaciones (`SendMessage`) y hace el polling de `/status` con Timer adaptativo (60 s / 15 s).

`TicketBloc`	Estado del carrito (ítems, selección, scroll, scan pendiente).
`PrinterBloc`	Impresora: conexión (USB/BT/Red/Serial), monitoreo, reintentos con backoff, impresión.
`ApicardBloc`	Pago con tarjeta (daemon ApiCard local).
`NetworkBloc`	Conectividad del device.
`PaymentMeansBloc` / `PaymentMethodBloc` / `PaymentInstallmentBloc`	Medios de pago disponibles, medio elegido y planes de cuotas.
`EnvaseBloc` / `ValeBloc`	Envases y vales.
`TicketManagementBloc` / `ValeManagementBloc`	Historial y administración.
`InactividadBloc`	Timer de inactividad (cancela el ticket por inactividad).
`SupervisorBloc`	Modo supervisor.

Servicios

Bajo `lib/services/`:

Servicio	**Responsabilidad**
`HttpCliente`	Wrapper HTTP que inyecta `X-Cod-Terminal`, `X-Terminal-Uuid`, `X-Device-Health`. Traduce `SocketException` / `TimeoutException` / `HandshakeException` a fallas de dominio.
`TicketService`	Mutaciones REST de ticket (open/agregar/remove/close/changeStatus/v-

	alidate).
`MercadopagoService`	`createIntent` / `executePayment` para QR y Point Smart.
`ApicardService`	Identificación/autorización/anulación contra el daemon ApiCard.
`ConfigurationService`	Config local y remota; genera el UUID del device; propaga identidad al `HttpCliente`.
`IsarServiceImpl`	Persistencia local (`auto_compra_db`).
`PendingPaymentService`	Recupera pagos in-flight al arrancar (fix INT-002).
`PrinterService` (+ implementaciones USB/BT/Network/Serial)	Impresión.

Persistencia (Isar)

Base **`auto_compra_db`**. Ubicación por plataforma: en Linux usa `/opt/autocompra/current/` o el directorio de documentos; en Windows/macOS/Android/iOS, el directorio de documentos de la app.

Entidades (en `lib/models/isar/`):

- `IsarConfig` — contenedor que agrupa config de ticket, POS, impresora, timeouts, devices, bolsas, ingreso manual.
- `IsarConnectionConfig` — `codTerminal`, `server`, `port`, `ssl`, `uuid`, y (fix TPOS-016) `apicardServer/Port/Ssl`.
- `IsarTerminalConfig` — `codSucursal`, `codNegocio`, `codComercio`, `nroPos`, `nroPVFiscal`, etc.
- `IsarPendingPayment` — pagos in-flight (fix INT-002): `paymentAttemptId`, `idProcessorPayment`, `status`, `createdAt`.

Comunicación con el backend

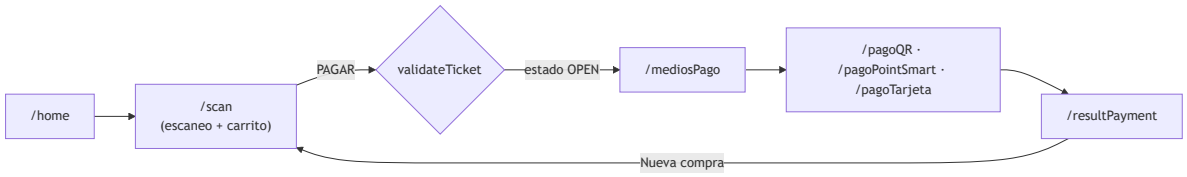
REST sobre `/autocompras/v1` (`lib/core/api/api_paths.dart`). El detalle de mutaciones, polling de salud, resiliencia y la migración desde STOMP está en Comunicación POS ↔ Backend.

Flujo de compra (scan → checkout)

El recorrido central de la terminal: el cliente escanea productos, arma el carrito, valida y pasa a pagar. Esta página sigue ese flujo pantalla por pantalla, con los BLoCs, eventos y DTOs reales. Para los flujos de pago en sí (QR, Point, tarjeta), ver Pagos.

> **El código es la fuente de verdad**
>
> Reproducimos clases, eventos y rutas tal como están en `lib/` a la fecha. Algunos números de línea cambiarán; los archivos no.

El mapa del flujo



El `/home` con la terminal ya conectada al backend: el botón COMENZAR abre un ticket, o el cliente escanea directamente un producto. El timer de inactividad y los badges de identidad acompañan toda la sesión.

La pantalla de escaneo (`ScanPage`)

`lib/views/scan/scan_page.dart` es el centro operativo. En `initState`:

- Lee la config de bolsas del `ConfigurationBloc`.
- Crea un `FocusNode` (`scannerFocusNode`) para **capturar el input del lector de código de barras como si fuera teclado**.
- Dispara `StartInactivityTimer` del `InactividadBloc`.
- Consume cualquier escaneo pendiente y restaura el foco al lector.

Cómo se captura un escaneo

El lector de barras se comporta como un teclado: "tipea" el código y manda un Enter. La terminal lo intercepta con `Focus(onKeyEvent: _handleKeyEvent)`:

1. `_handleKeyEvent` acumula las teclas en un buffer hasta el Enter y produce un `ScanResult { code, type, errorMessage? }`.
2. Si hay error de lectura, muestra un diálogo y no agrega nada.
3. Si es válido, llama a `_processBarcode(code, type)` y **resetea el timer de inactividad** (hubo actividad).

`ScanType` distingue `barcode`, `qr` y `manual` (este último, ingreso manual habilitado por supervisor).

Del escaneo al carrito

`_processBarcode` no agrega el ítem localmente: se lo pide al backend (el backend es la fuente de verdad del ticket). Despacha por el `BackendClientBloc`:

```
context.read<BackendClientBloc>().add(
  SendMessage(
    source: DestinationPage.scanPage,
    destination: BackendDestination.getArticulo,
    topic: BackendTopic.ticketArticulo,
    message: RequestTicketArticuloDto(
      ean: code,
      cantidad: quantity.toDouble(),
      orden: 0,
      tipoScan: type.type,
      ticketDto: ticketBloc.state.rawTicket,  // el ticket actual
    ),
  ),
);
```

El backend resuelve el EAN (ver Decodificación de EAN), recalcula promos y total, y devuelve el `TicketDto` actualizado.

El `TicketBloc` y la respuesta

Un `BlocListener<BackendClientBloc>` procesa la respuesta según el estado:

Estado backend	Acción en la UI
`sending`	`SetScanning(true)` — muestra spinner.
`received`	Parsea el `TicketDto` → `AddTicket(...)` — el carrito se actualiza y hace auto-scroll al ítem nuevo.
`error`	`SetScanning(false)` + diálogo de error.

Eventos del `TicketBloc` (`lib/viewmodels/ticket/`):

Evento	Qué hace
`AddTicket(ticket, rawTicket, selectedItem, moveScroll)`	Agrega/actualiza el ítem en el carrito.
`UpdateTicket(ticket, rawTicket)`	Actualiza el ticket (validación, pago).

`ClearTicket()`	Limpia el carrito (nueva compra).
`SetScanning(bool)`	Muestra/oculta el spinner.
`SetPendingScan(scanResult)` / `ClearPendingScan()`	Guarda/limpia un escaneo para consumir tras navegar.

Por qué se guarda `rawTicket`

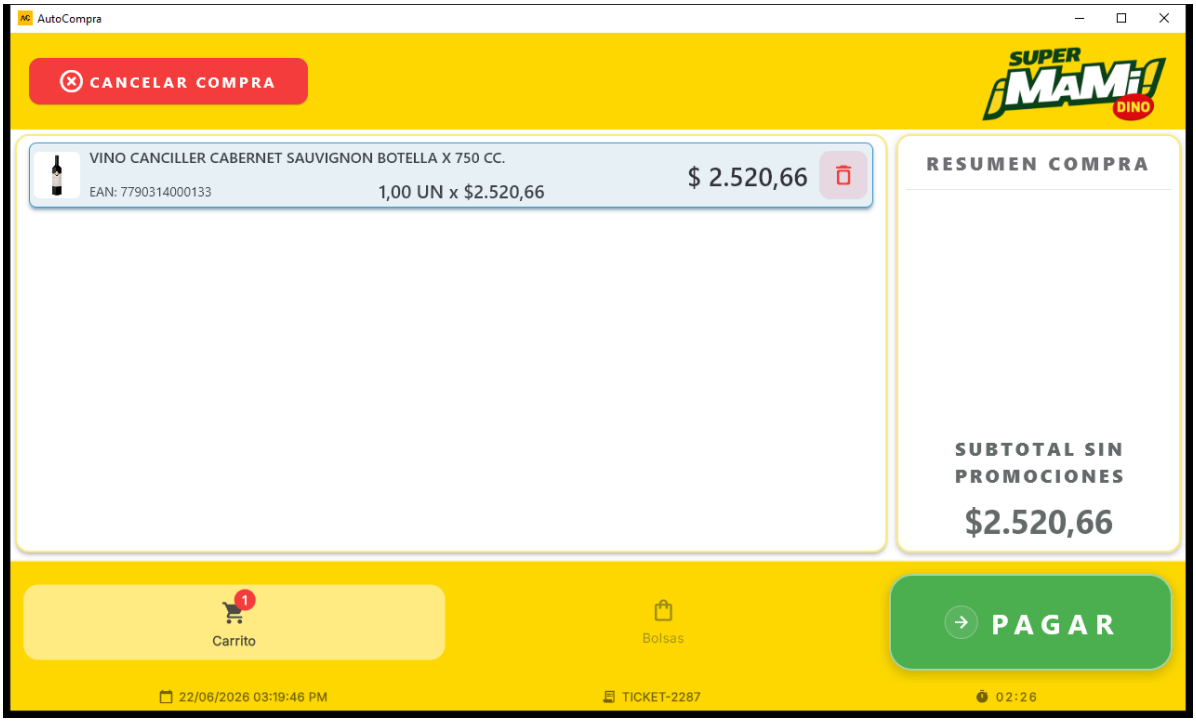
El `TicketState` mantiene dos versiones: el `ticket` (para mostrar) y el `rawTicket` (el crudo del servidor, que se manda de vuelta en el siguiente escaneo). Así el backend siempre recibe el estado exacto que él emitió, sin que el cliente lo "reinterprete".

El carrito en pantalla

Componentes (en `lib/views/scan/`):

Widget	Rol
`ProductView` → `ListProductWidget` → `ItemProducto`	La lista scrollable de ítems (cantidad, precio unitario, total; el ítem nuevo se anima).
`SummaryPurchase`	Totales, impuestos y descuentos.
`MenuItem` (tabs)	Carrito / Bolsas / Fidelización.
`ModernPayButton`	El botón PAGAR, con estado de carga.

El carrito recién abierto: a la izquierda los ítems (acá vacío, "No hay items agregados"), a la derecha el RESUMEN COMPRA con el subtotal, abajo los tabs Carrito/Bolsas y el botón PAGAR (deshabilitado hasta que haya ítems). Arriba, "CANCELAR COMPRA" y el branding del comercio.

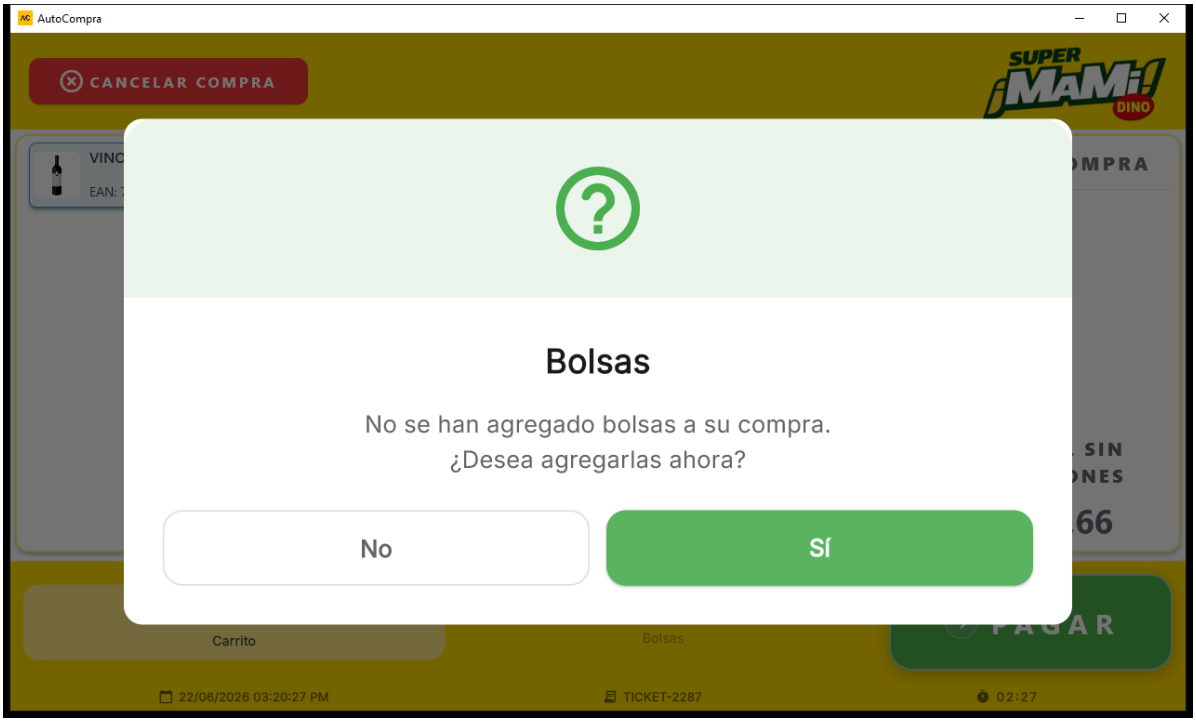


El mismo carrito tras escanear un artículo (EAN `7790314000133`): la línea muestra descripción, EAN, `1,00 UN × $2.520,66` y su total; el RESUMEN COMPRA refleja el subtotal, el badge del carrito marca 1 y el botón PAGAR se habilita (verde). Cada escaneo es un `getArticulo` que el backend resuelve y devuelve como `TicketDto` recalculado — exactamente el orden de cómputo.

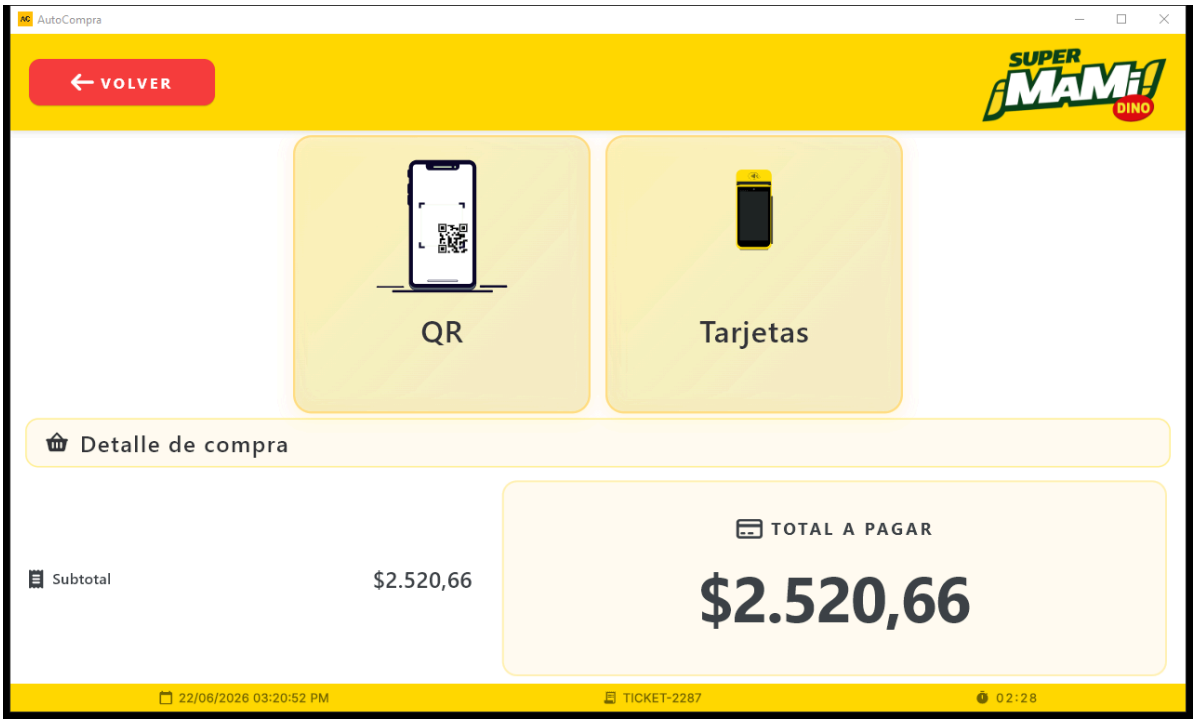
Validación y salida al pago

Cuando el cliente toca **PAGAR** (`_onTapPay`):

1. **Bolsas**: si está configurado preguntar por bolsas y el ticket no tiene, muestra el diálogo "¿Agregar bolsas?".
2. **Validación contra el backend**: despacha `validateTicket` con el `rawTicket` completo.
3. **Resultado**:
4. Si el ticket vuelve en estado `OPEN` (validado, listo para pagar) → navega a `/mediosPago` .
5. Si hay error o **timeout** (la validación tarda > ~30 s) → diálogo con el contexto (`isScanPageValidateTicketTimeoutState` detecta el timeout por `commandReference == 'TIMEOUT'`).



Si el comercio lo configura y el ticket no tiene bolsas, al tocar PAGAR aparece el prompt "¿Desea agregarlas ahora?" (No / Sí) antes de validar.



Tras validar, `/mediosPago`: las opciones QR y Tarjetas (Point Smart), el detalle de compra y el TOTAL A PAGAR. Desde acá cada medio abre su flujo —ver Pagos—.

Los DTOs del ticket

`TicketDto`

`lib/dto/backend/ticket_dto.dart` — el agregado que viaja entre terminal y backend. Campos principales:

- **Identidad/fiscal**: `codComercio`, `cuitComercio`, `codSucursal`, `nroPos`, `codTerminal`, `estado`, `tipoComprobante`, `nroPVFiscal`, `nroComprobanteFiscal`, `nroCAE`, `vtoCAE`.
- **Ítems**: `cantidadItems`, `cantidadItemsAnulados`, `items: List<ItemDto>`.
- **Envases**: `cantEnvases`, `cantEnvasesConsumidos`, `cantEnvasesCobrar`, `valeEnvases`.
- **Totales** (como núcleo impositivo): `totalAPagar`, `totalPromociones`, `total`.
- **Relaciones**: `cliente`, `mediosDePago`, `ordenDePago`, `promociones`, `vouchers`, `movimientos`.

`ItemDto` lleva `ean`, `descripcion`, `cantVenta`, `precioVenta`, `montoVenta`, `estadoItem` (`VENTA` / `ANULADO`), `orden`, impuestos y la info de envase si aplica.

Request DTOs

DTO	Se manda en	Contiene
`RequestTicketArticuloDto`	`getArticulo` / `removeItem`	`ean`, `cantidad`, `orden`, `tipoScan`, `ticketDto`.
`RequestTicketPaymentDto`	`payTicket`	`ticketDto`, `commandReference` (`CONSULTAR` / `EJECUTAR`), `ordenes`.
`RequestTicketStatusDto`	`changeStatusTicket`	`estado` (`CANCELED_USER`, `CANCELED_AUTOMATICALLY`…), `ticketDto`, `motivo?`.

`OrdenPagoDto`

Una orden de pago dentro del ticket: `autorizador` (MERCADOPAGO, etc.), `idGlobalUUID`, `monto`, `customData` (referencia del pago), `orden`, y `estado` (`OPEN`, `PAGADA`, `ERROR`, `RECHAZADA`). El `navigationAction` del `TicketState` decide la ruta posterior según el estado del ticket y de la última orden.

El despacho: `BackendClientBloc`

Todas las mutaciones pasan por el `BackendClientBloc` (`_onSendMessage`), que rutea según `BackendDestination` al método correspondiente del `TicketService`:

```
final result = switch (event.destination) {
  BackendDestination.openTicket        => ticketService.openTicket(...),
  BackendDestination.getArticulo       => ticketService.agregarArticulo(...),
  BackendDestination.removeItem        => ticketService.removeItem(...),
  BackendDestination.closeTicket       => ticketService.closeTicket(...),
  BackendDestination.changeStatusTicket=> ticketService.changeStatus(...),
  BackendDestination.validateTicket    => ticketService.validateTicket(...),
  BackendDestination.payTicket         => _ejecutarPago(...),
};
```

Cada llamada lleva los headers de identidad (`X-Cod-Terminal`, `X-Terminal-Uuid`) que inyecta el `HttpCliente`. El detalle de la comunicación, el polling de salud y la resiliencia está en Comunicación POS ↔ Backend.

Subsistema de impresión

La impresora es el punto donde más cosas pueden salir mal en una terminal desatendida: se queda sin papel, se abre la tapa, se desconecta el Bluetooth, se cae el USB. Por eso la impresión en TiprePOS es un **subsistema completo** —no un `print()` —, con cuatro tipos de conexión, monitoreo de estado, reconexión automática con backoff e impresión en lote resiliente. Todo orquestado por `PrinterBloc` (`lib/viewmodels/printer/printer_bloc.dart` , ~2000 líneas).

> ⚠ **El código es la fuente de verdad**
>
> Reproducimos estados, eventos y el algoritmo de reconexión tal como están a la fecha. Los detalles finos viven en `PrinterBloc` y `lib/services/printer/` .

Los cuatro tipos de conexión

Un `PrinterServiceFactory` crea la implementación según `PrinterConnectionType` :

Tipo	Implementación	Notas
`usb`	`UsbPrinterService`	Puerto USB/COM.
`bluetooth`	`BluetoothPrinterService`	Búsqueda por MAC, **no soporta monitoreo de estado** por comando.
`network`	`NetworkPrinterService`	TCP `IP:puerto` .
`serial`	`SerialPrinterService`	Puerto serie tradicional.

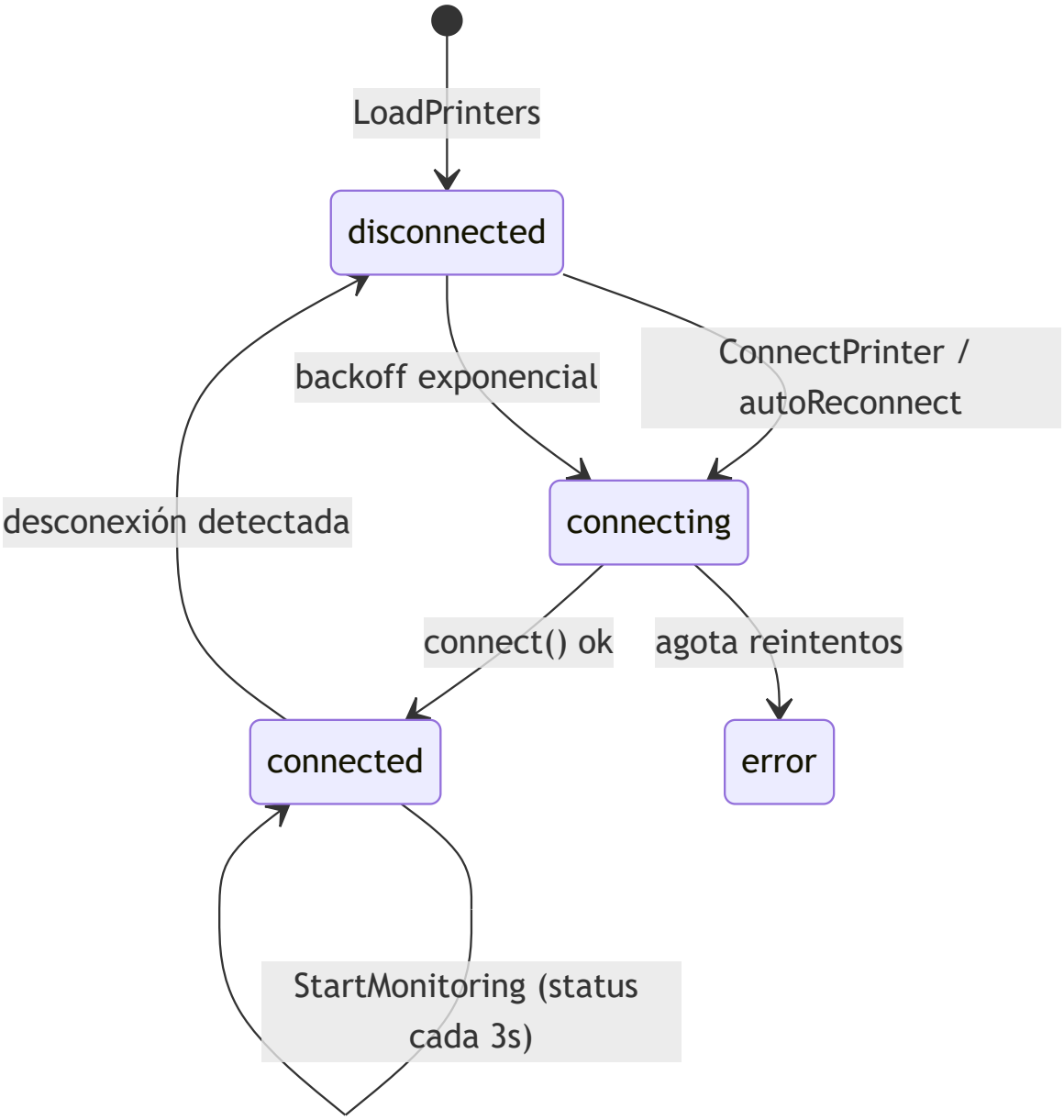
Todas implementan la misma interfaz `PrinterService` : `getAvailablePrinters()` , `connect(config)` , `disconnect()` , `checkStatus()` , `sendRawData(bytes)` , `dispose()` , `isBluetoothEnabled()` . El resto del sistema no sabe qué tipo de conexión hay debajo.

Estados

```
enum PrinterConnectionStatus { notInitialized, disconnected, connecting,
connected, error }
enum PrintStatus { idle, printing, success, error }
enum PrinterErrorType { paperOut, coverOpen, offline, communicationError,
deviceNotFound, unknown }
```

El `PrinterBlocState` lleva, entre otros: la lista de impresoras, la seleccionada, el `PrinterConnectionStatus`, el `PrintStatus`, el tipo de conexión, `usePrinter` (`SI` / `NO` de config), `isMonitoring`, la MAC Bluetooth persistida, y el progreso de impresión en lote (`currentVoucherIndex` / `totalVouchers`).

Ciclo de vida de la conexión



- **Carga** (`LoadPrintersEvent`): si `usePrinter != 'SI'` , no hace nada. Para Bluetooth, recupera la MAC guardada e inicia reconexión; para USB/Serial/Network, busca dispositivos y auto-selecciona el primero.
- **Conexión** (`ConnectPrinterEvent`): arma la config, llama `connect()` , y si tiene éxito, hace `checkStatus()` , persiste la MAC (Bluetooth) y arranca el monitoreo si la conexión lo soporta.

Monitoreo de estado (timer adaptativo)

Si la conexión soporta status (USB/Serial/Network), un `Timer.periodic(3s)` dispara `CheckStatusEvent`, que pregunta a la impresora su estado ESC/POS (sin papel, tapa abierta, etc.). Cada ~15 s, además, chequea el estado del radio Bluetooth.

> **Bluetooth no se puede monitorear por comando**
>
> `BluetoothPrinterService` no soporta el chequeo de estado ESC/POS, así que para Bluetooth el monitoreo se reduce a detectar desconexión del radio. Es una limitación del transporte, no un bug.

Reconexión automática con backoff exponencial

Cuando se detecta una desconexión o falla una conexión, el bloc reintenta solo, con **delay creciente** para no martillar:

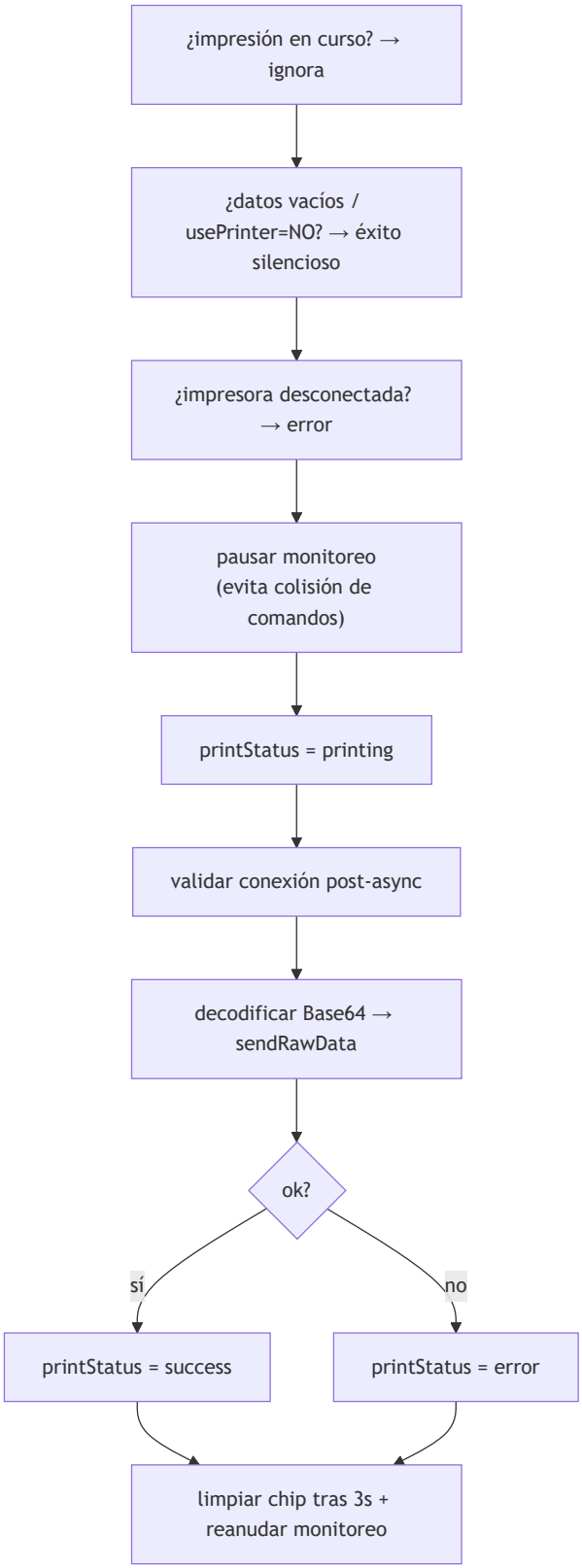
```
delay = min(maxDelay, minDelay × 2^intento)
        = 500 ms → 1 s → 2 s → 4 s → 8 s → (tope 10 s)
```

- **USB/Serial/Network**: busca dispositivos, auto-selecciona, reintenta con el delay exponencial.
- **Bluetooth**: busca el dispositivo por la **MAC guardada**; reintenta hasta **3 veces** (`bluetoothReconnectMaxAttempts`); si no aparece, pide reconfiguración (`requiresPrinterConfiguration`).

La MAC del Bluetooth se persiste localmente (callback `onBluetoothMacChanged`) para poder reconectar sin que el cliente vuelva a parear.

Impresión de un recibo

`PrintReceiptEvent { type, data, currentVoucherIndex }` — `data` es el ticket ya formateado en Base64. El handler es deliberadamente cuidadoso:


¿impresión en curso? → ignora
¿datos vacíos / usePrinter=NO? → éxito silencioso
¿impresora desconectada? → error
pausar monitoreo (evita colisión de comandos)
printStatus = printing
validar conexión post-async
decodificar Base64 → sendRawData
ok?
sí
no
printStatus = success
printStatus = error
limpiar chip tras 3s + reanudar monitoreo

El detalle clave: **se pausa el monitoreo antes de imprimir** (esperando confirmación, hasta 500 ms) para que un `checkStatus` no se cruce con el envío del recibo en el mismo canal, y se reanuda después con un segundo de estabilización.

`ReceiptType` distingue `ticket`, `voucher`, `vale`, `factura`.

Impresión en lote (`StartPrintingVouchers`)

Cuando hay varios comprobantes (ticket + vouchers de pago + vale), `StartPrintingVouchers { vouchers, stopOnError }` los imprime en secuencia:

- Antes de **cada** voucher chequea que la impresora siga conectada (si se perdió, corta y reporta).
- Espera ~300 ms entre comprobantes.
- Acumula fallos: si `stopOnError`, corta en el primero; si no, sigue e informa al final cuántos salieron y cuáles fallaron ("Parcial: 2 exitosos…").

> **El resultado parcial importa**
>
> En autoservicio, que salgan 2 de 3 comprobantes no es lo mismo que fallar todo. El subsistema distingue éxito total, fallo total y parcial — y lo comunica. No asumas binario.

Test y diagnóstico

- `PrintTestPageEvent` → patrón de prueba ESC/POS (`buildTestTicketBytes`).
- "Solo QR" → imprime únicamente el QR (`buildQrTicketBytes`).
- Ambos pasan por `_runManualPrintJob`: mismas guardas (no imprimir si hay un job en curso, pausar/reanudar monitoreo).

La pantalla `management_printer` (`lib/views/management_printer/`) expone todo esto al operador: elegir tipo de conexión, listar/seleccionar dispositivo, conectar/desconectar, ver estado en tiempo real, test page y gestión de la MAC Bluetooth.

Issue gemelo en el backend

Del lado del backend, `VoucherService` tiene un problema de thread-safety conocido (plan 007) en la config de impresora compartida entre requests. Son dos capas distintas (formateo en el backend, envío físico en la terminal); si depurás un voucher mal impreso, fijate en cuál de las dos está el problema.

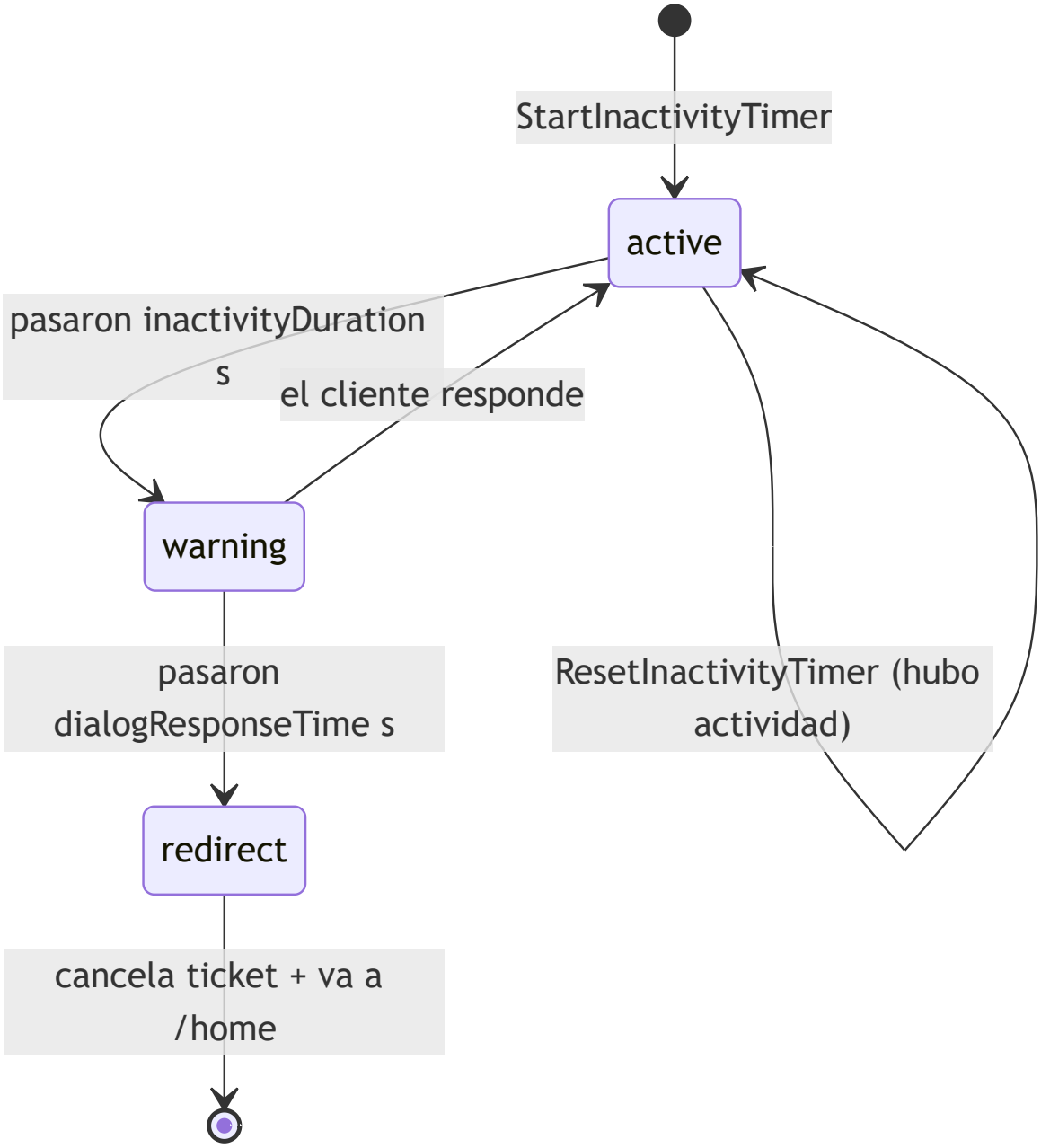
Operación, sesión y configuración

Todo lo que rodea al flujo de compra y lo hace operable en un comercio real, sin nadie atendiendo: el reloj de inactividad, el alta de la terminal, el modo supervisor, el branding por comercio, las pantallas de gestión y el manejo de errores.

> ⚠ **El código es la fuente de verdad**
>
> Reproducimos clases, eventos y estados a la fecha. Donde algo parece **stub** o pendiente de cablear, lo marcamos.

Inactividad: el reloj que protege la terminal

`InactividadBloc` (`lib/viewmodels/inactividad/`) es crítico en autoservicio: si un cliente abandona la terminal a mitad de una compra, **nadie va a venir a cancelarla**. El sistema lo hace solo.



- **Dos timers**: el de inactividad (ej. 300 s) y, cuando ese vence, el del diálogo de advertencia (ej. 30 s para responder). Los valores vienen de la config de la terminal (`LoadTimeoutInactivity`).
- **Reset por actividad**: cada escaneo, tap, scroll o apertura de diálogo en `ScanPage` dispara `ResetInactivityTimer`.
- **Cancelación**: si vence el diálogo, `RedirectToHome` despacha al backend un `changeStatusTicket` con estado `CANCELED_AUTOMATICALLY`, limpia el `TicketBloc`

(`ClearTicket`) y navega a `/home`.

> **Por qué dos timers y no uno**
>
> El primero da tiempo generoso (el cliente puede estar buscando un producto); el segundo es una última chance explícita ("¿seguís ahí?"). Sin el diálogo, cancelarías compras de gente que solo se distrajo un momento.

Configuración y registro de la terminal

La terminal no sirve hasta estar **configurada** (sabe a qué backend hablar) y **registrada** (el backend le dio su `TerminalConfig`). El `ConfigurationBloc` (`lib/viewmodels/configuration/`) maneja ese ciclo.

Evento	Qué hace
`LoadConnectionConfiguration`	Carga server/puerto/SSL/uuid desde Isar.
`LoadTerminalConfiguration`	Carga la `TerminalConfig` local; si no hay, hace falta registrar.
`FetchConfiguration(...)`	Descarga config fresca del backend.
`RegisterTerminal(...)`	Registra la terminal contra el backend.
`RefreshConfiguration(silent?)`	Re-descarga; en modo `silent` no interrumpe la UI si ya hay datos.
`UpdateLocalPrinterMacAddress(mac)`	Persiste la MAC Bluetooth local.

Qué pide el registro

La pantalla `register` (`lib/views/register/`) toma: `codTerminal`, `server`, `port`, `ssl`, y opcionalmente `apicardServer` / `apicardPort` (el daemon de tarjeta). Al registrarse, el backend devuelve la `TerminalConfig` completa.

Qué baja el backend: `TerminalConfig`

`lib/domain/configuration/terminal_config.dart` — la identidad operativa y fiscal de la terminal:

- **Identidad**: `codSucursal`, `codNegocio`, `codComercio`, `nroPos`, `nroPVFiscal` (el punto de venta AFIP).
- **`config`**: bloque detallado con `device` (impresora: modelo, tipo de conexión, path/MAC), `bolsa` (EAN de bolsa, si se pregunta), `timeout` (inactividad, conexión), parámetros fiscales y `theme` (branding).

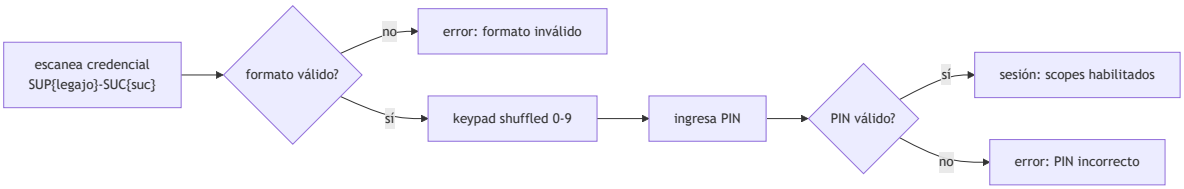
Tras registrar, se persiste todo en Isar y se propaga la identidad al `HttpCliente` (`setTerminalIdentity`), para que cada request lleve los headers `X-Cod-Terminal` / `X-Terminal-Uuid`.

Refresh silencioso

Si cambian precios o promos en el backend, `RefreshConfiguration(silent: true)` re-descarga la config sin spinner ni interrumpir al cliente: si falla, mantiene la UI estable con lo que ya tenía.

Modo supervisor

`SupervisorBloc` (`lib/viewmodels/supervisor/`) habilita funciones restringidas: ingreso manual de productos, anulación de ítems, cambio de precio. El flujo de autenticación:



- **Credencial**: se escanea un código con formato `SUP{legajo}-SUC{sucursalId}` (regex `^SUP(\d{1,8})-?SUC(\d{1,4})$`).
- **PIN con teclado mezclado**: el keypad se baraja (0-9 en orden aleatorio) para que mirar la pantalla no revele el PIN — defensa contra shoulder-surfing.
- **Sesión**: `SupervisorSession { supervisorId, supervisorName, legajo, scopes }`. Los scopes (`manualEntry`, `authorizeVoid`, `modifyPrice`) gobiernan qué puede hacer.

> **La validación de PIN está mockeada (pendiente de backend)**
>
> Según el código actual, la validación de PIN **no consulta al backend**: hay un delay simulado y un PIN fijo, con un `TODO` de re-cablear contra el servidor. Documentamos el flujo real (credencial + keypad shuffled + sesión con scopes), pero **el chequeo de PIN todavía no es real**. Es lo primero a endurecer antes de habilitar funciones sensibles en producción.

Branding por comercio

`AppThemeCubit` (`lib/core/theme/`) cambia colores, logo y marca según el comercio. `AppTheme.resolve(cuit)` busca el tema por CUIT en un mapa predefinido y cae a un tema por defecto si no lo encuentra. Define `primary` / `secondary`, fondos, color del botón PAGAR, colores de estado (error/warning/success), `logoUrl` y datos de marca. Los widgets lo leen con `AppTheme.of(context)`.

El theming en acción: la misma terminal, con el logo, los colores (amarillo SUPER MAMI) y la marca del comercio resueltos a partir de su CUIT. El "Powered by tipre" y los badges de identidad son parte del chrome común.

Pantallas de gestión

Pantalla	Para qué
`management_ticket`	Historial de tickets: búsqueda por fecha/estado/cliente, ver detalle, reimprimir, anular (con autorización de supervisor).
`management_vale`	Vales de envase: búsqueda, movimientos (consumo/devolución), reimpresión. Ver Envases y vales.
`management_printer`	Impresora: estado, tipo de conexión, dispositivo, test page. Ver Subsistema de impresión.

Manejo de errores y resiliencia

- **Tipos de falla**: jerarquía `Failure` (`ServerFailure`, `NetworkFailure`, `TimeoutFailure`, `JsonParseFailure`, `GenericFailure`). El `HttpCliente` traduce excepciones de transporte (`SocketException`, `TimeoutException`, `HandshakeException`) a estas fallas de dominio.
- **Estados de error del backend**: el `BackendClientBloc` distingue timeout (`commandReference == 'TIMEOUT'`, > ~30 s), HTTP 5xx/red caída, JSON inválido, y pago rechazado (`success=false`).
- **`ErrorPage`**: pantalla amigable con ícono, mensaje para el cliente y botón "Reintentar".
- **`ConnectionRecoveryMixin`**: en las pantallas de pago, maneja timeout + reintento de la consulta de transacción sin colgar la UI. Ver Pagos.
- **Logging estructurado**: `AppLogger` + `FlowLogger` registran cada request con contexto (pantalla, acción, EAN, estado del ticket). Hay captura global de errores (`runZonedGuarded` en `main`). Es la primera parada cuando algo falla en una terminal en el campo.

En autoservicio, el error tiene que ser para el cliente Y para soporte

`ErrorPage` muestra algo entendible para quien está comprando; el `AppLogger` deja el detalle técnico para quien después tiene que diagnosticar. Las dos audiencias importan: no mezcles el stack trace en la pantalla ni escondas el detalle del log.

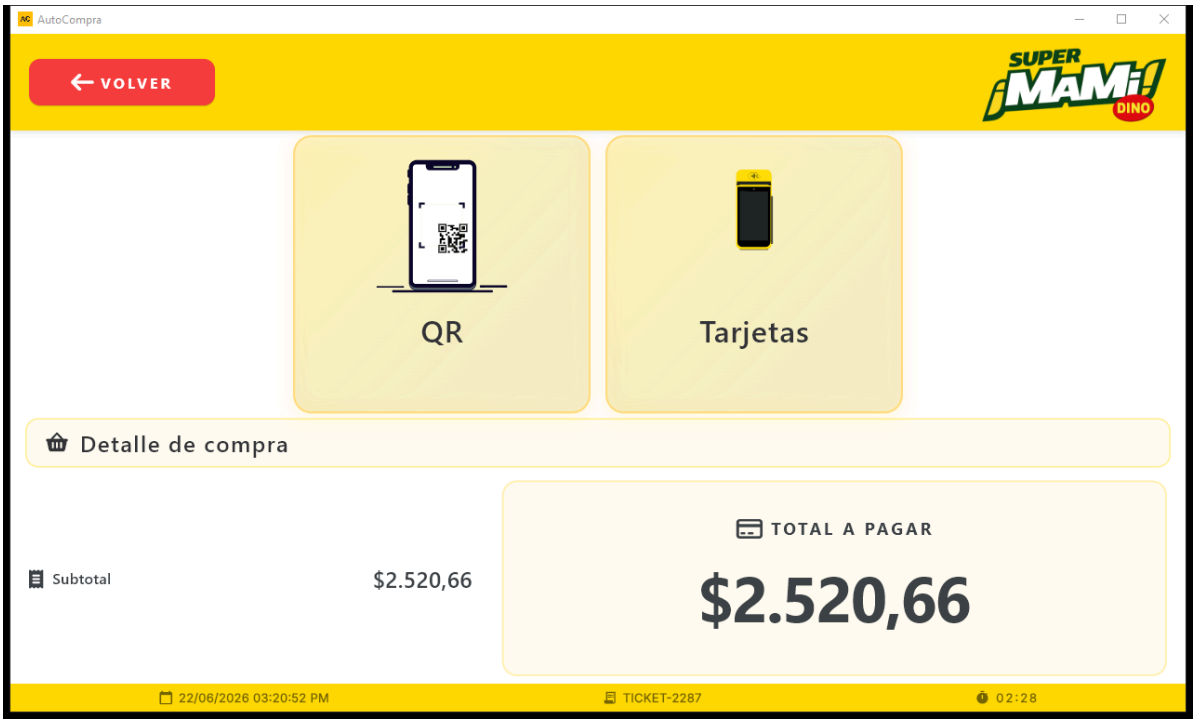
Pagos

El punto más delicado del sistema. Un bug acá no es un pixel mal puesto: es un **doble cobro** o un pago que queda en el limbo. Leé esta página entera antes de tocar cualquier flujo de cobro.

Los tres medios de pago

Medio	Procesador	Dónde corre	Timeout en la terminal
QR	MercadoPago	Backend → MercadoPago; el cliente escanea con su app	~210 s
Point Smart	MercadoPago Point	Terminal física (pinpad) de MercadoPago	~90 s
Tarjeta	ApiCard	**Daemon local** en la terminal (`localhost:50001`)	—

En la terminal, los medios disponibles se cargan con `PaymentMeansBloc` (QR = `9001`, Point Smart = `9002`). El enum de procesadores es `PaymentProcessors { mercadopago, pointsmart, apicard }`.



La pantalla `/mediosPago` real: QR y Tarjetas (Point Smart), con el total a pagar. Cada opción abre el flujo de cobro correspondiente.

La regla de oro: idempotencia

El riesgo central de cualquier pago es **cobrar dos veces** cuando la red corta entre "cobré" y "recibí la confirmación". La defensa es el `paymentAttemptId`:

- Se genera **una sola vez** por intención de pago (`idProcessorPayment`).
- Se **reusa** en todos los reintentos del **mismo** intento.
- El backend **deduplica** por ese id: el mismo intento reenviado N veces produce un solo cobro.

Nunca generes un `paymentAttemptId` nuevo en un reintento

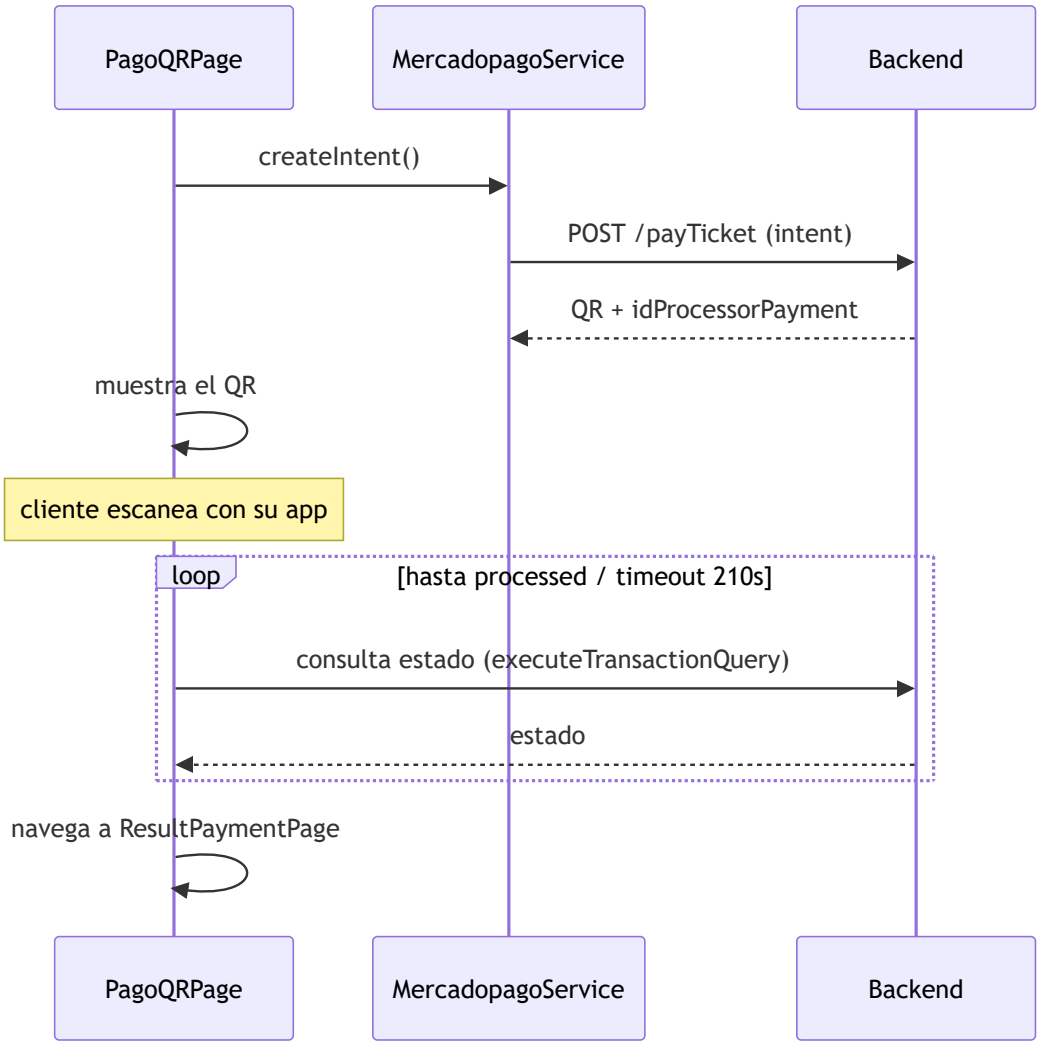
Si en un reintento generás un id nuevo, rompés la dedup y habilitás el doble cobro. El id es la identidad del intento, no del request. Esto es el fix INT-001 y es **no negociable**.

Recuperación de pagos in-flight

Un corte de luz o un crash en medio de un cobro no puede dejar la plata en el limbo. Por eso:

- Al iniciar un pago, la terminal persiste un `IsarPendingPayment` en Isar.
- Al **arrancar**, `PendingPaymentService.pendientes()` lee esos registros y reconcilia los pagos que quedaron sin resolver en la sesión anterior (fix INT-002).
- Cuando el pago se resuelve, el registro se elimina.

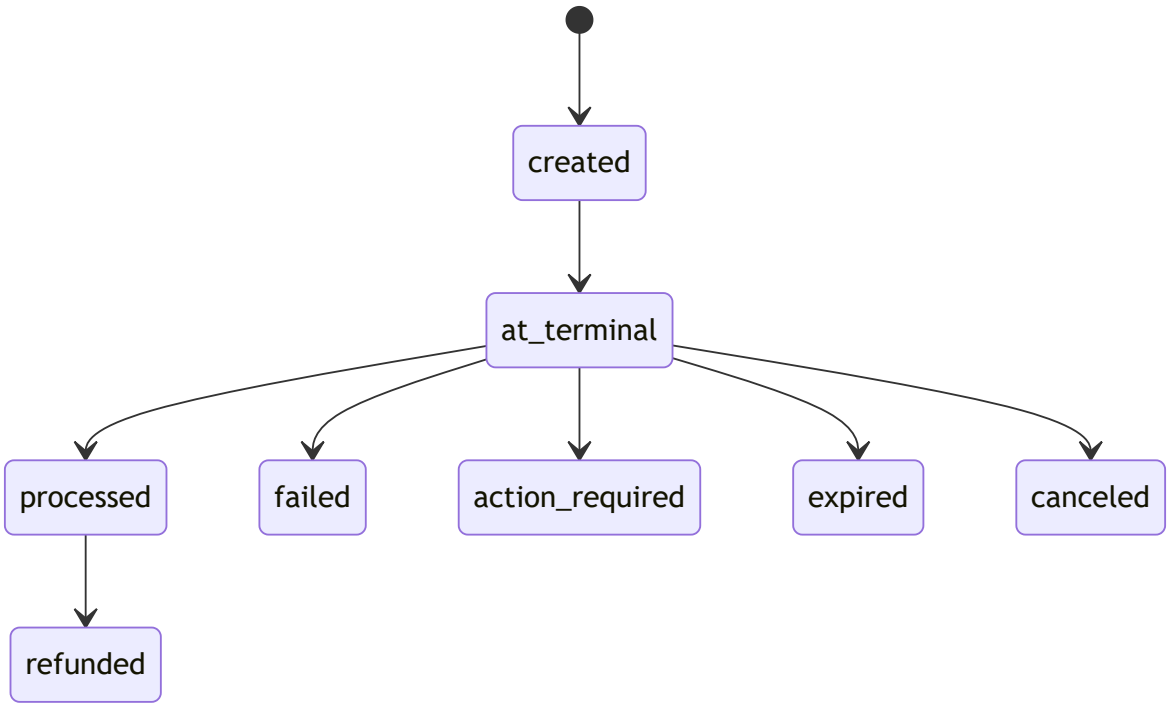
Flujo QR (MercadoPago)



La pantalla usa `ConnectionRecoveryMixin` para manejar timeout + reintento de la consulta sin colgarse.

Flujo Point Smart (MercadoPago Point)

La terminal física tiene su propia **máquina de estados de orden** (documentada en `docs/mp-point_estados_order_validaciones.md`):



Estado	Qué hacer en la terminal
`created`	"Esperando terminal…", permitir cancelar.
`at_terminal`	"Complete en la terminal", bloquear duplicados.
`processed`	Éxito: conciliar monto/moneda, **idempotencia en el cierre** (no reimprimir).
`failed`	Mostrar motivo, ofrecer reintento o cambiar de medio.

`action_required`	Incierto (~40 s): "Revisar terminal", reintentar consulta, protocolo manual.
`expired`	>15 min: informar y, si sigue, generar nueva orden.
`canceled`	Desbloquear la UI.
`refunded`	Guardar referencia y ajustar conciliación.

> ⚠ **`action_required` y `expired` son los traicioneros**
>
> Son los estados donde el dinero puede estar cobrado pero la UI no lo sabe. Ante la duda, **concilia por `payment_id`** contra MercadoPago antes de dar la venta por perdida o por hecha. Nunca asumas el resultado: consultá.

Flujo Tarjeta (ApiCard)

ApiCard **no pasa por el backend**: la terminal habla con un **daemon local**.

1. `PagoTarjetaPage` arranca con `ApicardBloc.add(IdentificarCompra())` → `POST localhost:50001/identificacion_compra_online/`.
2. Se muestran las cuotas (`PlanPagoPage`).
3. El usuario elige cuotas → `ApicardBloc.add(AutorizarCompra())` → `POST .../autorizacion_compra_online/`.
4. La anulación va por `.../autorizacion_anulacion_compra_online/`.

La dirección del daemon es configurable (fix TPOS-016): `apicardServer` / `apicardPort` / `apicardSsl` en `IsarConnectionConfig`, con fallback a `localhost:50001`.

Resultado del cobro

`ResultPaymentPage` cierra el flujo con la animación correspondiente (confetti en éxito) y auto-redirige tras unos segundos. Recibe `title`, `isError`, `errorMessage`, `paymentName`, la ruta de `redirect`, etc.

Antes de tocar pagos, repasá también

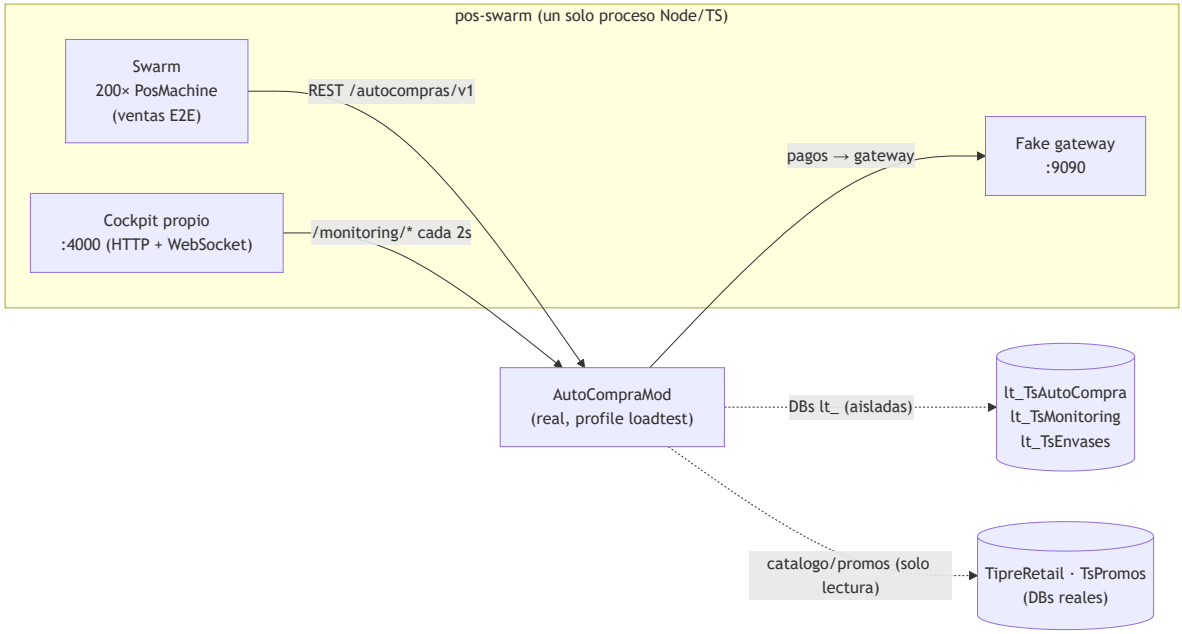
- Comunicación POS ↔ Backend — resiliencia y polling.
- El estado `VOUCHERPENDING` del ticket en el Glosario: pagado pero con el comprobante fiscal aún sin confirmar.

StressBench (pos-swarm) — load-test E2E

StressBench es la herramienta de prueba de carga del ecosistema: simula **~200 terminales de autoservicio** haciendo ventas completas y concurrentes contra el backend **real** (AutoCompraMod), con un **gateway de pago falso** configurable en vivo y un **cockpit web propio** para ver el enjambre y manejarlo. Sirve para asegurar que el sistema aguanta carga realista. Vive en el repo `pos-swarm` (Node/TypeScript).

> **Por qué existe**
>
> El Cockpit te muestra cómo está el backend, pero no genera carga. StressBench **produce** la carga (200 POS vendiendo a la vez) y la cruza con la observabilidad del backend. Es la forma de validar, antes de producción, que el reconciliador drena los pagos indeterminados, que no hay doble cobro, y que las latencias (p50/p95) se mantienen estables bajo presión.

La idea en una imagen

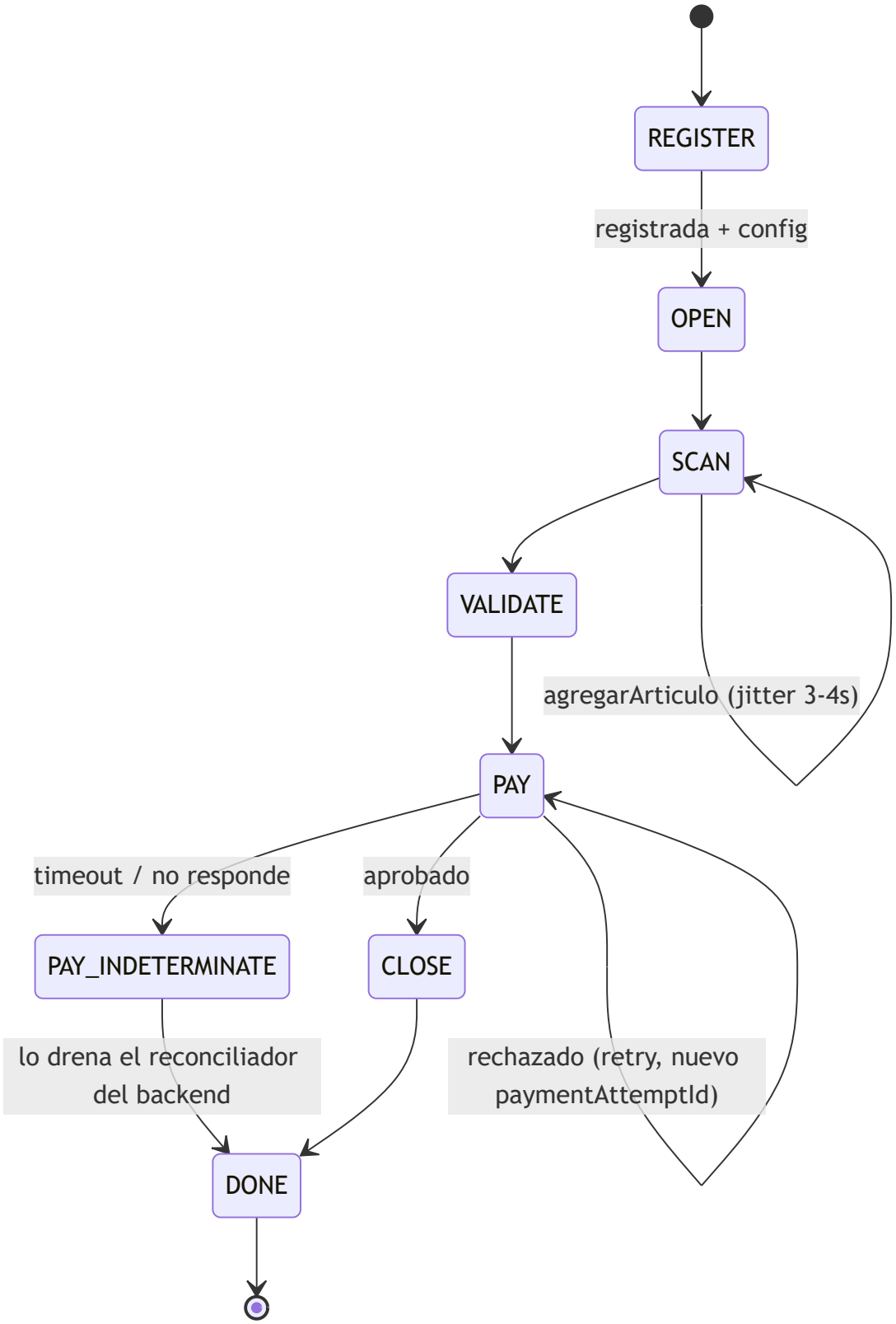


Las tres piezas conviven en **un solo proceso** (`npm start` → `src/index.ts`): el swarm, el fake gateway y el cockpit comparten el event-loop de Node.

El swarm (motor de carga)

`src/engine/swarm.ts` orquesta hasta 200 máquinas virtuales (`PosMachine`), cada una un POS de autoservicio que corre su propio loop async.

- **`Swarm`** : pre-crea las 200 máquinas idle; `setTarget(n)` (el slider del cockpit) define cuántas están activas; `reconcile()` hace **ramp-up escalonado** (arranca una cada ~75 ms) y drena las sobrantes; `snapshot()` / `stateCounts()` alimentan la UI.
- **`PosMachine`** (`src/engine/posMachine.ts`): una máquina de estados async que emula el flujo real de la terminal contra el backend:


REGISTER
registrada + config
OPEN
SCAN
VALIDATE
agregarArticulo (jitter 3-4s)
PAY
timeout / no responde
aprobado
PAY_INDETERMINATE
CLOSE
lo drena el reconciliador del backend
rechazado (retry, nuevo paymentAttemptId)
DONE

Una venta completa (`runOneSale()`) ejecuta los mismos endpoints REST que la terminal Flutter: `openTicket` → `getArticulo` (loop, con think-time 3-4 s) → `validateTicket` → `payTicket` (CREATE_INTENT + EXECUTE_PAYMENT, con retry) → `closeTicket` . Ver Flujo de compra.

Sin doble cobro: el tool NO reintenta el indeterminado

Si el pago queda **indeterminado** (timeout), `PosMachine` **no reintenta** — lo deja para que el reconciliador del backend lo resuelva. En cambio, un pago **rechazado** sí se reintenta, con un `paymentAttemptId` nuevo cada intento. Esto replica exactamente la regla de idempotencia de pago del POS real.

El fake payment gateway (`:9090`)

`src/gateway/fakeGateway.ts` implementa **el mismo contrato** que el gateway de pago real espera (verificado contra `GatewayMPBuilder.java` del backend). Mantiene un ledger en memoria de intents (`estado: OBTENCIONIDTRX | PAGO_APROBADO | PAGO_DENEGADO | PAGO_CANCELADO`).

Endpoint	Paso	Qué hace
`POST /api/trxid`	CREATE_INTENT	Crea el intent (`OBTENCIONIDTRX`), devuelve `id` + `qrData` .
`POST /api/trxpago`	EXECUTE_PAYMENT	Aplica la decisión de pago (aprobar/rechazar). Idempotente.
`POST /api/trxanulacion`	CANCEL	Marca `PAGO_CANCELADO` .
`GET /api/trxes/{id}`	CHECK_STATUS	Lo usa el **reconciliador del backend** para resolver indeterminados.

Escenarios de pago (configurables en vivo desde el cockpit)

`src/gateway/behavior.ts` — el comportamiento es mutable y se cambia desde la UI sin reiniciar:

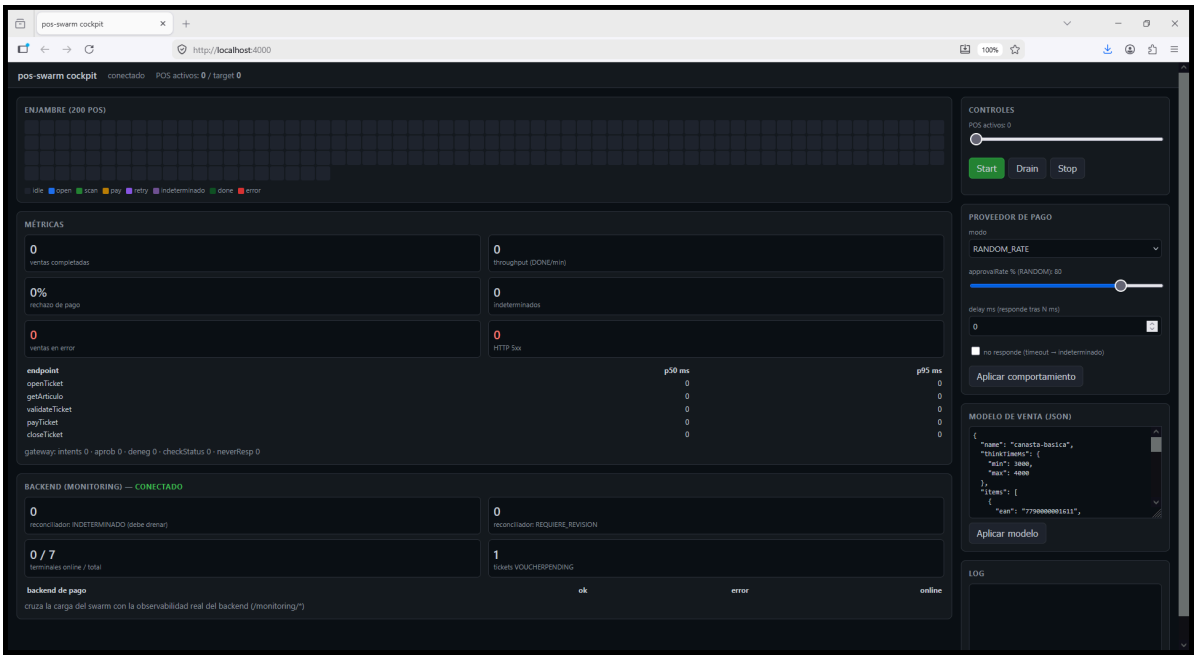
Escenario	Config	Efecto en el sistema
Todo aprueba	`ALWAYS_APPROVE`	Camino feliz.
% de rechazo	`RANDOM_RATE` + `approvalRate` (ej. 80 → ~20% rechazos)	Valida el **retry-on-reject** E2E.
Muy lento	`delayMs` > timeout del backend (~340 s)	Fuerza estado **INDETERMINADO**.
No responde	`neverRespond: true` (cuelga el socket)	Fuerza **INDETERMINADO** por timeout.

Los escenarios "lento" y "no responde" son los valiosos: fuerzan indeterminados para verificar que el reconciliador los drena.

El cockpit propio de StressBench (`:4000`)

`src/cockpit/wsHub.ts` — un server HTTP + WebSocket que sirve una SPA (`src/cockpit/public/index.html`) y empuja estado cada 400 ms.

> ⚠ **No confundir con el Cockpit del backend**
>
> StressBench tiene **su propio** cockpit en `:4000` (controla el enjambre). Es distinto del Cockpit de AutoCompraMod (`/autocompras/v1/cockpit/` , observabilidad del backend). StressBench **además** consume el del backend para cruzar datos (ver abajo).



El cockpit de StressBench (en reposo, antes de arrancar): el enjambre de 200 POS (grilla con leyenda de estados), las métricas (p50/p95 por endpoint, % rechazo, HTTP 5xx, contadores del gateway), los controles (slider de POS, escenario de pago RANDOM_RATE/approvalRate/"no responde", modelo de venta JSON) y el panel "Backend (monitoring) — conectado" que cruza con `/monitoring/*` del backend (reconciliador, terminales, VOUCHERPENDING).

Qué muestra:

- **Grilla de POS** (tiles de colores por estado: OPEN, SCAN, PAY, RETRY, INDETERMINADO, CLOSE, DONE, ERROR), con tooltip de último endpoint y latencia.
- **Métricas del swarm**: ventas completadas, throughput DONE/min, **% de rechazo de pago**, indeterminados pendientes, ventas en ERROR, **HTTP 5xx**, y **p50/p95 por endpoint** (`src/metrics/collector.ts`, ring buffers de 500 muestras).
- **Controles**: slider 0–200 POS (ramp-up automático), Start/Drain/Stop, selector de escenario de pago + `approvalRate` / `delayMs` /"no responde", y un **modelo de venta** editable (JSON, validado con Zod): qué EANs escanear, think-time, retries.

El panel "Backend (monitoring)": el cross-check

`src/backend/monitoring.ts` pollea cada 2 s `/monitoring/{fiscal,terminals,payments}` del backend y muestra un panel con el **reconciliador INDETERMINADO** (debe drenar bajo carga), `REQUIERE_REVISION`, terminales online/total, tickets `VOUCHERPENDING` y el estado de los backends de pago.

Por qué este panel es el corazón de la prueba

El driver solo no puede ver que el reconciliador drena los indeterminados — eso pasa **dentro del backend**. Cruzar la carga del swarm con `/monitoring/*` es lo que valida la pieza más delicada del sistema: el ciclo de pago y fiscalización bajo presión.

Aislamiento de bases: las DBs `lt_`

El backend corre con el profile `loadtest` (`application-loadtest.yml`), que apunta los gateways de pago al fake y separa los datasources:

Módulo	DB en loadtest	Modo
`tickets`	`lt_TsAutoCompra` (vacía, Hibernate crea las tablas)	**escribe** (todas las ventas/pagos/fiscal del test)
`monitoring`	`lt_TsMonitoring`	escribe (error-sink; fail-soft si no está)
`envases`	`lt_TsEnvases` (copia aislada por backup/restore)	escribe (vales)
`catalogo`	`TipreRetail` (**la real**, ~900k filas)	**solo lectura**
`promos`	`TsPromos` (**la real**)	solo lectura

Catálogo es la DB real → usá EANs REALES

Como `catalogo` apunta a la base real (leer no la muta, y da volumen fiel a producción), el modelo de venta **tiene que usar EANs que existan** (`SELECT TOP 5 cEan FROM TipreRetail.dbo.dm_Artic WHERE iNroSuc=1`). Un EAN inventado da "artículo no encontrado". Para probar envases, usá EANs con envase-retornable asociado.

Los scripts en `seed/` arman todo: `00_create_lt_databases.sql` (crea las DBs lt_), `01_create_lt_envases.sql` (copia aislada de envases, opcional),

`02_create_lt_monitoring_tables.sql` (tablas de error/jobs, opcional), y `seed.sql` (copia la config real del comercio + genera los 200 POS `POS-001..200`).

> **VOUCHERPENDING vs CLOSE en el test**
>
> Para que los tickets lleguen a `CLOSE` (y no queden en `VOUCHERPENDING`), el profile loadtest apunta `app.ticket.vouchers.templatepath` a un template real bundleado (`seed/vouchersTemplates`). El voucher se **genera** pero no se imprime (los SelfService no imprimen). La fiscalización usa **CAEA local** (no pega a AFIP); el CAEA viene sembrado en el comercio.

Cómo se corre (resumen)

```
# 1) Crear las DBs lt_ (ajustá -S host / -P password)
sqlcmd -S localhost -U sa -P <pwd> -i seed/00_create_lt_databases.sql

# 2) Backend una vez (crea tablas), Ctrl-C, y sembrar
cd ../AutoCompraMod
mvn spring-boot:run -Dspring-boot.run.arguments="--
spring.profiles.active=loadtest"   # luego Ctrl-C
sqlcmd -S localhost -U sa -P <pwd> -i ../pos-swarm/seed/seed.sql

# 3) El tool (fake gateway :9090 + cockpit :4000)
cd ../pos-swarm && npm install && npm start

# 4) Backend con el profile loadtest
cd ../AutoCompraMod
mvn spring-boot:run -Dspring-boot.run.arguments="--
spring.profiles.active=loadtest"

# 5) Abrir http://localhost:4000 → modelo de venta + escenario de pago → slider a
200 → Start
```

Variables de entorno (opcionales): `POS_SWARM_BACKEND` (default `http://localhost:8080/autocompras/v1`), `POS_SWARM_GATEWAY_PORT` (9090), `POS_SWARM_COCKPIT_PORT` (4000), `POS_SWARM_MAX_POS` (200).

> **Smoke de 1 POS antes del enjambre**
>
> `npx tsx scripts/smoke-one-pos.ts` corre **un** POS por el flujo E2E completo y loguea cada paso (status + latencia). Es la forma rápida de confirmar que el setup (DBs, seed, gateway, EANs) está bien antes de levantar los 200.

Qué mirar para decir "anda ok"

Señal	Bien	Mal
HTTP 5xx	≈ 0 sostenido	picos → el backend no aguanta
Ventas completadas	suben	estancadas → bloqueo/timeout
% rechazo de pago	≈ `approvalRate` configurado	valida el retry-on-reject
Indeterminados	drenan (sube `checkStatus`, los tickets cierran)	quedan colgados → reconciliador roto
p50/p95 por endpoint	estables en el tiempo	crecientes → leak/saturación de pool Hikari/threads

El log del backend confirma la pieza clave: `Reconciliador: ... resuelto a APROBADO`.

Setup del entorno

Guía para levantar el ecosistema en local: primero el **backend**, después la **terminal** apuntándole. Si recién entrás, hacé este recorrido una vez de punta a punta antes de tocar código.

> **Antes de empezar**
>
> Leé Arquitectura y El monolito modular. Vas a entender mucho mejor qué estás levantando y por qué el `mvn test` "raro" que rompe el build en realidad es tu amigo (el guardrail).

1. Backend (AutoCompraMod)

Requisitos

- **JDK 21** (el enforcer fuerza `[21,22)`; con otra versión el build falla a propósito).
- **Maven**.
- Acceso a las bases **SQL Server** que usan los módulos (`TsAutoCompra`, `TipreRetail`, `TsPromos`, `TsEnvases`, y la de observabilidad). Sin ellas, podés correr los tests de modularidad pero no el runtime completo.
- **Node** lo instala solo el build de Maven (no hace falta tenerlo global) cuando construís el Cockpit.

Build y guardrail

```
cd c:/Work/Tipre/AutoCompraMod

# Compila + corre la suite + verifica boundaries de módulos. NO necesita DB.
mvn test
```

`mvn test` corre `ModularityTests.verify()`: si tu cambio cruza una frontera entre módulos o crea un ciclo, **el build se rompe acá**. Eso es deseable. Ver el guardrail.

Levantar el backend

```
# Con los DataSource configurados (application.yml / variables de entorno)
mvn spring-boot:run
```

El backend queda en `:8080` con context-path `/autocompras/v1`. Probá un healthcheck:

```
curl http://localhost:8080/autocompras/v1/tickets/dummy
curl http://localhost:8080/autocompras/v1/actuator/health
```

Seguridad en dev

Por defecto `app.security.enabled=false`: todo `permitAll` (verás un `WARN` en el log recordándolo). Está bien para desarrollar contra la terminal. Para probar la política de roles, poné `enabled=true` y configurá el resource server JWT/Keycloak. Ver Backend → Seguridad.

Cockpit

El panel se construye con el profile `frontend` de Maven y queda servido por el backend en `/autocompras/v1/cockpit/`. Para iterar el front con hot-reload, podés correr Vite aparte dentro de `cockpit-ui/` (proxyea a `localhost:8080`):

```
cd c:/Work/Tipre/AutoCompraMod/cockpit-ui
npm install
npm run dev
```

2. Terminal (TiprePOS)

Requisitos

- **Flutter** `>=3.35.6` y **Dart** `>=3.0.0 <4.0.0`.
- Para tarjeta: el **daemon ApiCard** corriendo (por defecto `localhost:50001`). Sin él, QR y Point Smart funcionan igual.

Instalar y correr

```
cd c:/Work/Tipre/TiprePOS
flutter pub get

# La terminal está pensada para desktop
flutter run -d windows
# o
flutter run -d linux
```

Primer arranque: configuración y registro

La terminal arranca sin config, así que el router te lleva por el gate (ver POS → Routing):

1. **`/configuration`** — cargá `server`, `port`, `ssl` y el código de terminal apuntando a tu backend local (`localhost:8080`, sin SSL).
2. **`/register`** — la terminal se registra contra el backend y baja su `TerminalConfig`.
3. **`/home`** — si todo salió bien, quedás en el portal y podés escanear, armar un ticket y probar un pago.

La config se persiste en Isar (`auto_compra_db`), así que el segundo arranque ya entra derecho a `/home`.

Compilar

```
flutter build windows
flutter build linux
flutter build apk
```

3. Verificar que se hablan

Con backend en `:8080` y terminal configurada apuntándole:

- En el log de la terminal deberías ver el polling de `/status` (cada 60 s sano).
- Un `openTicket` desde la terminal debe devolver un `TicketDto` con estado `OPEN`.
- En el Cockpit (`/autocompras/v1/cockpit/`), tu terminal debería aparecer **online** en el panel de terminales (heartbeat dentro de la ventana de ~90 s).

Si la terminal no conecta

- ¿`ssl` bien? Un `HandshakeException` en el log es mismatch http/https.
- ¿El backend está en `:8080` con context-path `/autocompras/v1`?
- ¿El healthcheck `curl .../tickets/dummy` responde?

Tests

```
# Backend
mvn test

# Terminal
flutter test
flutter test integration_test/app_test.dart
```

Guías de tareas

Recetas concretas para tareas puntuales del ecosistema. A diferencia de la Referencia (que describe qué hay) o de Entender el sistema (que explica por qué), acá el foco es **cómo hago X**, paso a paso.

Backend

Agregar o tocar un módulo sin romper el guardrail

1. Poné el código nuevo en el paquete del módulo correcto (`com.tipre.autocompras.<modulo>`).
2. Lo que sea API pública va en la **raíz** del paquete; lo interno, en **subpaquetes**.
3. Si tu módulo necesita algo de otro, llamá **sólo su API pública** — nunca una clase de un subpaquete ajeno. Si te tienta cruzar la frontera, replanteá: probablemente la dependencia va al revés (que `tickets` orqueste) o por **evento** (`@ApplicationModuleListener`).
4. Corré el guardrail:

   ```
   mvn test
   ```

   Si `ModularityTests.verify()` falla, no toques el test: revisá el diseño. Ver El monolito modular.

Refrescar una cache (catálogo o promos)

Con seguridad activa hace falta rol admin:

```
curl -X POST http://localhost:8080/autocompras/v1/catalogo/cache/refresh \
  -H "Authorization: Bearer <jwt-admin>"

curl -X POST http://localhost:8080/autocompras/v1/cache/refresh \
  -H "Authorization: Bearer <jwt-admin>"
```

Con `app.security.enabled=false` (dev) no hace falta el token, pero recordá el riesgo del toggle.

Diagnosticar el parque de terminales en el Cockpit

1. Abrí `/autocompras/v1/cockpit/`, panel **Terminals**.
2. Una terminal **offline** no se borró: no pingeó dentro de la ventana de ~90 s. Revisá su conectividad y su último heartbeat.
3. Para salud por terminal (pinpad/Point), mirá lo que reporta el header `X-Device-Health`. Ver Cockpit.

Terminal (TiprePOS)

Configurar una terminal nueva contra un backend

1. Arrancá la app: el router te manda a `/configuration` (no hay config).
2. Cargá `server`, `port`, `ssl` y el código de terminal.
3. En `/register` la terminal se registra y baja su `TerminalConfig`.
4. Verificá en el Cockpit que aparezca **online**.

Detalle en Setup del entorno.

Apuntar el pago con tarjeta a otro daemon ApiCard

Por defecto la terminal pega a `localhost:50001`. Es configurable (fix TPOS-016): `apicardServer` / `apicardPort` / `apicardSsl` en `IsarConnectionConfig`. Ver Pagos → Tarjeta.

Entender por qué un pago Point Smart quedó "incierto"

Si ves el estado `action_required` o `expired`, el dinero puede estar cobrado aunque la UI no lo confirme. **No asumas**: conciliá por `payment_id` contra MercadoPago antes de dar la venta por perdida o por hecha. Ver la máquina de estados de Point Smart.

Pruebas de carga (StressBench)

Correr un load-test del backend

Para verificar que el sistema aguanta carga realista (200 terminales vendiendo a la vez), usá StressBench:

1. Creá las DBs `lt_` aisladas y sembrá los 200 POS (`seed/`).
2. Levantá el backend con el profile `loadtest` (apunta los pagos al gateway falso y los datasources a las DBs `lt_`).
3. `cd pos-swarm && npm install && npm start` (gateway falso `:9090` + cockpit `:4000`).
4. Abrí `http://localhost:4000` , cargá el modelo de venta con **EANs reales**, elegí el escenario de pago y subí el slider a 200.

Antes del enjambre, corré el smoke de 1 POS: `npx tsx scripts/smoke-one-pos.ts` . Detalle completo en StressBench.

Forzar (y validar) el drenado de pagos indeterminados

Es la prueba más valiosa. En el cockpit de StressBench, poné el escenario de pago en "**no responde**" o "**muy lento**" (`delayMs` > timeout). Eso fuerza pagos `INDETERMINADO` . Después confirmá en el panel "**Backend (monitoring)**" que `reconciliador.indeterminado` **baja** (el reconciliador del backend los resuelve), y en el log del backend: `Reconciliador: ... resuelto a APROBADO` . Ver El ciclo de pago y fiscalización.

Documentación

Correr esta doc en local

```
python -m venv .venv
.\.venv\Scripts\Activate.ps1
pip install -r requirements.txt
mkdocs serve
```

Abrí http://127.0.0.1:8000. Validá links antes de pushear:

```
mkdocs build --strict
```

Exportar la doc a PDF

```
pip install -r requirements.txt
python -m playwright install chromium
$env:PDF_EXPORT = "true"; mkdocs build
```

El PDF queda en `site/pdf/autocompramod-documentacion.pdf`. Detalle en el README del repo (sección *Exportar a PDF*).

> **Esta sección crece con el uso**
>
> Cuando resuelvas una tarea no trivial y repetible (provisionar una terminal, migrar una dependencia, depurar un flujo de pago), agregá acá la receta. Una buena guía how-to es corta, numerada y va directo al grano.