Table of Contents

INDUXA Engineering Manual

Guía técnica para integradores y diseñadores de aplicación

INDUXA SCADA · v1.0 · 2026

Powered by Dynux Technologies EAS

Preámbulo

Sobre este manual

Este manual está dirigido a **ingenieros y diseñadores de aplicación** que construyen, mantienen u operan sistemas SCADA basados en **INDUXA SCADA**. Cubre la creación de la base de datos de tags, la programación de scripts, la composición de pantallas con widgets, la configuración de alarmas, recetas, máquinas de estado y la integración con dispositivos de campo vía Modbus, MQTT, OPC-UA y S7.

No reemplaza la documentación de operador (que se centra en el uso del Viewer en planta) ni el manual de administración del Gateway (que cubre despliegue, licencias y operación de la pasarela).

Convenciones

- `Código en línea` para identificadores, nombres de tags, comandos y rutas.
- Bloques de código indican lenguaje:

```
// JavaScript de ejemplo
Induxa.tag.write('PUMP_RUN', true);

IEL: {TEMP} > 50 ? '#FF0000' : '#00FF00'
```

- Las llamadas de atención sobre seguridad o riesgos se marcan con una cita destacada como esta.
- Los **casos de uso** están numerados (UC-01, UC-02 …) para poderlos referenciar en formación o en revisiones técnicas.

Versiones cubiertas

Componente	Versión documentada
INDUXA Editor	1.x
INDUXA Viewer	1.x
INDUXA Gateway	1.x
Esquema de proyecto	V4
API REST	v1

Cuando una característica esté disponible solo a partir de una versión posterior, se indicará explícitamente.

Soporte

Reportes de errores, mejoras y consultas técnicas: a través de los canales internos de soporte de Induxa.

Introducción y arquitectura

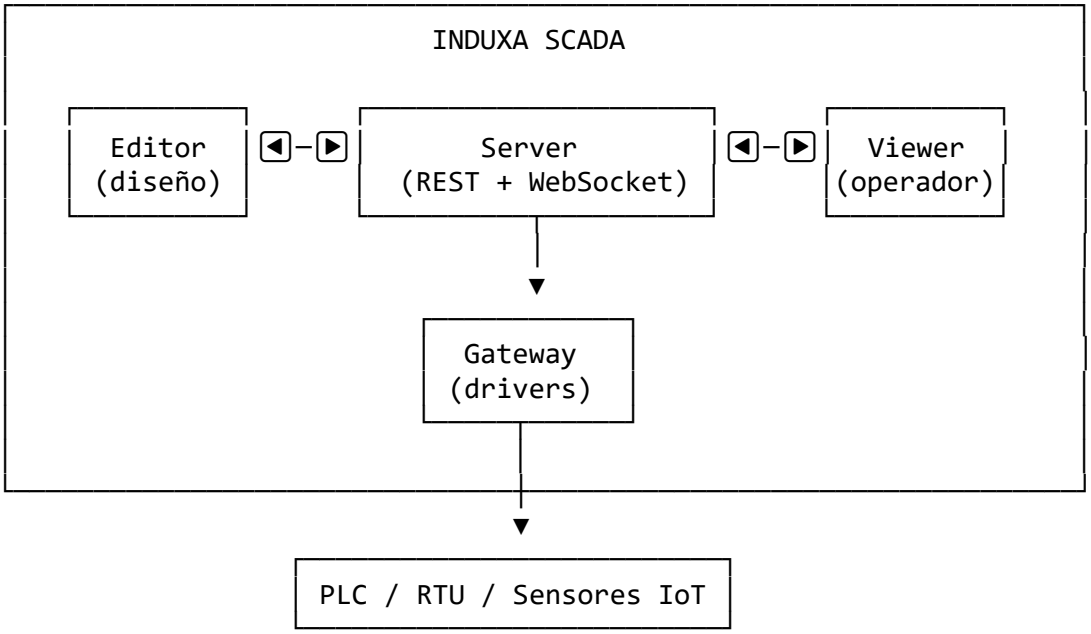
Qué es INDUXA SCADA

INDUXA SCADA es una plataforma de supervisión, control y adquisición de datos para entornos industriales. Permite:

- Adquirir variables de proceso desde PLCs, RTUs y dispositivos IIoT vía Modbus TCP/RTU, MQTT (incluido Sparkplug B), OPC-UA y Siemens S7.
- Modelar la planta con **Tags** y **UDTs** (User-Defined Types) que encapsulan equipos completos (bombas, válvulas, lazos PID).
- Diseñar interfaces operativas con un **Editor visual** y un catálogo de **widgets** vinculables a tags mediante **IEL** (INDUXA Expression Language) o JavaScript completo.
- Detectar y gestionar alarmas según ISA-18.2.
- Ejecutar lógica de negocio en tres ámbitos: en el Gateway (event-driven), programada (Scheduler) o en el navegador del operador.
- Auditar cada escritura de operador y cada acción de seguridad.

Componentes

INDUXA SCADA se compone de tres procesos cooperantes y un almacén de proyecto compartido:



Componente	Rol	Audiencia
Editor	Diseño del proyecto: tags, pantallas, widgets, alarmas, scripts	Ingeniero
Viewer	Visualización en producción y operación de la planta	Operador
Gateway	Conecta con dispositivos de campo, ejecuta drivers y motores	Ingeniero / Mantenimiento
Server	API REST, WebSocket de actualización, autenticación, persistencia	(servicio interno)

Flujo de datos

1. Un **driver** del Gateway lee un dispositivo de campo en su scan rate.
2. El valor se escribe en el **Tag** correspondiente (cambia su `value`, `quality` y `timestamp`).
3. El motor de tags **publica** el cambio:
    a. A los clientes Viewer activos vía WebSocket → los **widgets** enlazados al tag se re-renderizan.
    b. A los **scripts de tag** (`onValueChange` / `onQualityChange`) si los tiene definidos.

   c. A los **motores** de alarma, evento, máquina de estado, log histórico.
4. Los **scheduler jobs** independientes pueden disparar trabajos periódicos (cron, intervalo, change, startup) que también leen y escriben tags.
5. El operador, desde el **Viewer**, puede iniciar escrituras: éstas pasan por la API REST, son auditadas y enviadas al Gateway, que las propaga al dispositivo.

Roles y permisos

INDUXA SCADA define cuatro roles built-in con permisos jerárquicos:

Permiso	admin	engineer	operator	viewer
`viewSynoptic` (ver pantallas)	✓	✓	✓	✓
`operate` (escribir tags, ack alarmas)	✓	✓	✓	—
`editSynoptic` (editar pantallas)	✓	✓	—	—
`manageTags` (crear/editar tags y scripts)	✓	✓	—	—
`manageConnections` (drivers y conexiones)	✓	—	—	—
`manageUsers` (crear/borrar usuarios)	✓	—	—	—
`viewReports`	✓	✓	✓	✓
`viewAuditLog`	✓	✓	—	—
`ackAlarms`	✓	✓	✓	—

> **Importante.** El rol con el que se evalúa cada petición **no es el que envía el navegador**: lo determina el JWT firmado por el Server al iniciar sesión. Cualquier intento del Viewer de afirmar un rol superior será ignorado por la API.

Estructura de un proyecto

Un proyecto de INDUXA SCADA es un único archivo JSON (`.syk`) que contiene la totalidad de la definición:

```
proyecto.syk
├── meta            (nombre, versión, autor, dimensiones del lienzo)
├── tags[]          (variables de proceso)
├── udt_definitions[] / udt_instances[]
├── screens[]       (pantallas del Viewer)
├── alarms[]        (definiciones de alarma)
├── events[]        (reglas declarativas de evento)
├── state_machines[]
├── recipes[]
├── scheduler.jobs[]
├── globalScripts.{gateway,client}[]
├── connections[]   (modbus, mqtt, opcua, s7)
└── security        (políticas, usuarios)
```

Para producción se genera una **instantánea de despliegue** (`.svk`) que el Viewer descarga atómicamente y mantiene en su cache local.

Caso de uso introductorio (UC-01)

> *Quiero mostrar la temperatura del reactor en pantalla y permitir al operador iniciar una bomba con un botón.*

1. **Conexión.** En el Editor, sección *Connections*, doy de alta una conexión Modbus al PLC.
2. **Tags.** Creo dos tags:
   - `REACTOR_TEMP` (`float32`, RO, scan `normal`, dirección `40001`, unidades `°C`).
   - `PUMP_RUN` (`bool`, RW, dirección `00033`).
3. **Pantalla.** Creo una pantalla `Main`. Arrastro un widget *numeric-display* y lo enlazo al tag `REACTOR_TEMP`. Arrastro un *button-momentary* y le configuro:
   - `clickAction = write_tag`
   - `targetTag = PUMP_RUN`
   - `valueOn = true`
4. **Alarma.** En *Alarms* defino `REACTOR_OVERTEMP` con condición `{REACTOR_TEMP} > 80`, prioridad `High`.
5. **Despliegue.** Pulso *Deploy* en el Editor. El Viewer recibe la nueva instantánea y muestra la pantalla `Main`.

Este flujo cubre las cuatro disciplinas que vamos a desarrollar en los siguientes capítulos: tags, pantallas, alarmas y operación. Más adelante veremos cómo añadir lógica con scripts (Capítulo 3) y expresiones IEL (Capítulo 4).

Tags

Concepto

Un **Tag** es la unidad atómica de información en INDUXA SCADA: una variable nombrada con un valor, una calidad y una marca temporal. Cada widget de pantalla, cada regla de alarma, cada script y cada job del scheduler se enlaza al sistema mediante tags.

Cada tag pertenece a un **protocolo** (Modbus, MQTT, OPC-UA, S7, internal o calculated) y a una **clase de scan** (fast, normal, slow u onchange). El motor de tags muestrea cada clase a una frecuencia distinta, optimizando ancho de banda y CPU.

Tipos de dato

Tipo	Rango	Uso típico
`bool`	`false` / `true`	señales digitales, flags
`int16`	-32768 … 32767	enteros de PLC
`int32`	$\pm 2.1 \times 10^9$	contadores

Tipo	Rango	Uso típico
`uint16`, `uint32`	sin signo	direcciones, máscaras
`float32`	IEEE 754 simple	medidas físicas
`float64`	IEEE 754 doble	acumulados de alta precisión
`string`	texto UTF-8	descripciones, recetas

Clases de scan

Clase	Periodo típico	Cuándo usarla
`fast`	250 ms	alarmas críticas, lazos rápidos
`normal`	1 s	proceso general (por defecto)
`slow`	5–10 s	analógicas estables (temperaturas ambientales)
`onchange`	dirigido por evento	MQTT, OPC-UA con suscripción

> **Regla de bolsillo.** Cuanto más rápido el scan, más carga sobre el Gateway. No subas un tag a `fast` solo porque "se ve más fluido": 1 s ya es imperceptible para un operador humano.

Protocolos

Modbus

Campos relevantes: `device_id`, `address` (entero o `40001`, `00033` con prefijo de área), `modbus_type` (`holding`, `coil`, `input`, `discrete`), `data_type`.

```
address   : 40001        # holding register
data_type: float32      # 32 bits → 2 registros consecutivos
scale     : 0.1          # ingeniería = raw × 0.1
offset    : -50          # valor final = (raw × 0.1) - 50
```

MQTT

Campos: `mqtt_topic_sub` para lectura (con wildcards +/#), `mqtt_topic_pub` para publicación, `mqtt_qos`, `mqtt_json_path` para extraer un campo de un payload JSON, `mqtt_filter_field` / `mqtt_filter_value` para filtrar mensajes a este tag.

OPC-UA

Campo único `opcua_node_id` (`ns=2;s=Reactor.Temp`). El Gateway abre una sesión por conexión, descubre tipos de dato y ofrece soporte de suscripción nativa (clase `onchange`).

Siemens S7

`device_id` apunta a un PLC S7-300/400/1200/1500 configurado. `address` usa la sintaxis nativa Siemens (`DB10.DBD20`, `M5.0`, `I0.3`).

Internal

Tag virtual residente solo en el Server (no se conecta a hardware). Útil para variables de receta, contadores, banderas de SCADA. Se inicializa con `default` y persiste en el `.syk`.

Calculated

Su valor se computa a partir de otros tags vía **IEL** (Capítulo 4) o una **pipeline** declarativa de pasos (`source`, `convert_unit`, `math_op`, `round`, `clamp`, `scale`, `json_field`, …). No tiene hooks `onValueChange` / `onQualityChange`; su única vía de mutación es la fórmula.

Calidad

Cada tag mantiene un campo `quality` con valores normalizados:

Calidad	Significado
`GOOD`	dato válido y reciente
`UNCERTAIN`	dato disponible pero con baja confianza (timeout parcial, fuera de rango razonable)
`BAD`	comunicación caída, dato no disponible
`STALE`	última lectura demasiado antigua

Los widgets respetan la calidad: por defecto un tag `BAD` se renderiza con tonos grises y un indicador de fallo. La lógica de scripts puede reaccionar al cambio con `onQualityChange`.

Tags calculados

Un tag calculado (`protocol = 'calculated'`) tiene una de estas formas:

Forma fórmula (IEL)

```
formula: '{T1} + {T2} / 2'
```

Se reevalúa cuando cualquiera de los tags referenciados cambia. Soporta operadores `+ - * / %`, comparaciones, ternario, y funciones built-in (`abs`, `min`, `max`, `clamp`, `round`, `floor`, `ceil`, `format`, `concat`, `if`).

Forma pipeline

```yaml
pipeline:
  - { op: source, tag: RAW_FLOW }
  - { op: convert_unit, from: 'L/min', to: 'm3/h' }
  - { op: clamp, min: 0, max: 1000 }
  - { op: round, decimals: 1 }
```

Útil cuando la transformación implica varias etapas y se quiere que sea legible y trazable paso a paso.

Eventos del tag (scripts)

Si la versión del proyecto lo permite, cada tag puede llevar dos hooks de evento:

```
// onValueChange — se ejecuta cuando 'value' cambia
if (current.value > 80 && prev.value <= 80) {
  Induxa.tag.write('FAULT_OVERTEMP', true);
}

// onQualityChange — se ejecuta cuando 'quality' cambia
if (current.quality === 'BAD') {
  Induxa.log('Lost link to ' + tagPath);
}
```

Estos scripts se ejecutan en el Server dentro de un sandbox aislado. La API completa, los límites y los casos de uso se cubren en el Capítulo 3.

> Cada script se limita a **10 KB**. Los scripts largos suelen ser síntoma de lógica que pertenece a un *Scheduler job* o a un *Global script*.

Persistencia y limitaciones

Aspecto	Valor
Tags por proyecto (free)	hasta el límite de licencia
Identificador (`id`)	inmutable; se preserva al renombrar y al regenerar UDTs
Acceso	`ro` (solo lectura) o `rw` (lectura/escritura)
Historiado	configurable por tag (intervalo, deadband, valueMode)
Direcciones duplicadas	permitidas (alias) — útil para vistas alternativas
Tags `internal` con `default`	el valor se hidrata al cargar el proyecto

Casos de uso

UC-02 — Acumular minutos de trabajo de una bomba

1. Tag `PUMP_RUN` (`bool`, RW) — comando de marcha.
2. Tag `PUMP_RUN_MIN` (`internal`, `float32`, default 0).
3. Scheduler job `pump-runtime`, tipo `clock`, intervalo `60000` ms:

   ```
   const r = Induxa.tag.read('PUMP_RUN');
   if (r && r.value === true) {
     const m = Induxa.tag.read('PUMP_RUN_MIN');
     Induxa.tag.write('PUMP_RUN_MIN', (m?.value || 0) + 1);
   }
   ```

4. Widget *numeric-display* enlazado a `PUMP_RUN_MIN` con sufijo `min`.

UC-03 — Validación de rango antes de aceptar set-point

1. Tag `SETPOINT_REQ` (`internal`, escrito por el botón del operador).
2. Tag `SETPOINT_OK` (`bool`, internal, RO).
3. Script onValueChange en `SETPOINT_REQ`:

```
const v = current.value;
const ok = v >= 0 && v <= 100;
Induxa.tag.write('SETPOINT_OK', ok);
if (ok) Induxa.tag.write('SETPOINT_PV', v);
else    Induxa.log('SETPOINT fuera de rango: ' + v);
```

4. Widget de alerta en pantalla mostrando `SETPOINT_OK = false`.

Scripts

INDUXA SCADA permite escribir lógica en JavaScript en seis ubicaciones distintas. Cada una tiene su motor, su sandbox y sus garantías. Saber elegir bien la ubicación es la diferencia entre una solución elegante y una pesadilla de mantenimiento.

Mapa de superficies

Superficie	Persistencia	Motor	Hooks
Eventos de tag	`tag.scripts.{onValueChange, onQualityChange}`	Server vm	onValueChange, onQualityChange
Scheduler	`project.scheduler.jobs[].script`	Server vm	clock / calendar / change / startup
Globales gateway	`project.globalScripts.gateway[]`	Server vm aislado	invocables como `Induxa.global.<fn>(...)` desde tag/scheduler
Globales cliente	`project.globalScripts.client[]`	Navegador	invocables como `Induxa.global.<fn>(...)` desde screen/widget
Pantalla	`screen.scripts.{onOpen, onClose, onRefresh, onIdle}`	Navegador	navegación + ciclo de vida
Widget	`widget.events.{onClick, onDoubleClick, onMouseEnter,`	Navegador	DOM events

Superficie	Persistencia	Motor	Hooks
	`onMouseLeave`, `onRightClick`, `onChange}`		

Reglas de oro

1. Si la lógica reacciona a un cambio de valor → script de tag.
2. Si la lógica es periódica → scheduler.
3. Si la lógica está vinculada a una pantalla concreta → screen script.
4. Si la lógica está en un botón o input → widget event.
5. Si la lógica se reutiliza en varios sitios → global (gateway o cliente).

API INDUXA

INDUXA SCADA expone una API estable bajo el namespace global **`INDUXA`**. Su superficie cambia según el contexto: el server tiene acceso síncrono al store de tags, el navegador hace todo vía REST y añade utilidades de UI.

Server-side (tag y scheduler)

```
Induxa.tag.read(path)              → { value, quality, timestamp } | null
Induxa.tag.write(path, value)      → boolean   (encolado, async)
Induxa.tag.writeMultiple(map)      → boolean   (solo scheduler)
Induxa.log(message)                → void      (escribe en log del server)
Induxa.global.<fn>(...)            → función global definida por usuario
console.log(message)               → alias de Induxa.log

# Reservados para futuras versiones (lanzan excepción si se invocan):
Induxa.db.query(...)
Induxa.http.post(...)
```

Variables disponibles en el sandbox del script:

Variable	Tipo	Significado
`tagPath`	string	nombre del tag que disparó el script
`current`	`{ value, quality, timestamp }`	estado **después** del cambio
`prev`	`{ value, quality, timestamp }`	estado **antes** del cambio
`isInitial`	bool	`true` solo en la primera ejecución del proceso
`INDUXA`	objeto API	ver tabla anterior
`console`	`{ log }`	alias de `Induxa.log`

Browser-side (pantalla y widget)

```
Induxa.tag.read(name)              → { value, quality, timestamp } | null
(sync, desde caché)
Induxa.tag.write(name, value)      → Promise<{ success, error? }>
Induxa.tag.writeWithContext(...)   → Promise<{ success, writeId? }>
 (audited)
Induxa.tag.writeBatch(writes, ctx) → Promise<{ success, results }>
Induxa.tag.readMultiple(names)     → { name: { value, quality, timestamp } }
Induxa.navigate(screenName)        → void
Induxa.openPopup(viewId, params)   → void
Induxa.notify(msg, type)           → void          ('info'|'success'|'warn'|
'error')
Induxa.confirm(msg)                → Promise<bool>
Induxa.user.{name, role, hasRole(r), getCurrentUser(), getCurrentRole()}
Induxa.screen.{name, previous, next, setData(k,v), getData(k)}   (solo screen
 scripts)
Induxa.popup.{params, id, close}                                  (solo popup
scripts)
Induxa.global.<fn>(...)            → función global cliente
```

> **Seguridad.** El rol expuesto en `Induxa.user.role` proviene del token de sesión del navegador y se usa **solo para afordancia visual** (deshabilitar botones, mostrar avisos). La autorización real la hace el Server contra el JWT firmado: nunca confíes en que *limitar el botón en la UI* sea suficiente para evitar la escritura.

Modelo de ejecución

Server (tag y scheduler)

Cada invocación corre en un sandbox aislado (`vm.runInNewContext`) con un timeout duro:

Superficie	Timeout
Script de tag	500 ms
Scheduler	configurable, por defecto 5 s, máximo 30 s

El sandbox **no tiene acceso** a `require`, `process`, `fs`, `module`, `__dirname`, `globalThis`, ni a ningún módulo nativo. Los errores se capturan, se registran y se exponen vía `GET /api/v1/scripts/errors`. Un script no puede tirar el server.

Globales del Gateway

Se compilan en un contexto vm aislado propio. Cuando un script de tag o de scheduler invoca `Induxa.global.miFn()`, esa función ejecuta en el realm de los globales, **no en el del invocador y nunca en el host**.

Browser

Cada invocación se envuelve en una IIFE asíncrona (`new Function`), de modo que `await` y promesas funcionan. Errores se capturan y se loguean en la consola del navegador; nunca rompen el viewer.

Restricciones importantes

`async` / `await` y temporizadores en server-side

No se soportan. Los scripts de tag y scheduler corren síncronamente. El timeout solo cubre la ejecución síncrona. Esto:

```
// ❌ NO FUNCIONA en scripts server-side
await new Promise(r => setTimeout(r, 100));
fetch('https://api.empresa.com/...');
```

falla con `setTimeout is not defined` o se ignora silenciosamente porque el motor sale del script antes de que la promesa se resuelva.

Soluciones idiomáticas:

- ¿Necesitas espera periódica? → un *Scheduler job* con intervalo.
- ¿Necesitas llamar a un HTTP externo? → un *Scheduler job* con la llamada HTTP envuelta en `Induxa.http.post()` (cuando esté disponible) o publicar a un tópico MQTT al que un servicio externo responda.
- ¿Necesitas leer/escribir en BD? → tag de tipo *SQL Query* (driver built-in) o un servicio externo con MQTT bidireccional.

Loop guard

Si un script escribe a un mismo tag más de **50 veces por segundo**, INDUXA SCADA pausa esas escrituras durante **5 segundos** y registra una entrada `loop` en el log de errores. Esto protege contra oscilaciones del tipo *A escribe B → B escribe A → A escribe B …*.

La cola interna de escrituras tiene **profundidad 1** y se drena al final del ciclo de scan, así que las escrituras encadenadas se procesan en pasos discretos, no recursivamente.

Lo que NO se soporta en server-side

- Cargar módulos externos (`require`, `import`)
- Buffers binarios, criptografía nativa
- Programar callbacks (`setTimeout`, `setInterval`, `setImmediate`)
- Hacer peticiones HTTP fuera de `Induxa.http`
- Compartir estado mutable entre scripts (los tags **son** ese estado compartido — usa tags `internal` para coordinación)
- Persistir estado fuera del proyecto

Límites de tamaño

Superficie	Límite
Script de tag (por hook)	10 KB
Código de función global	64 KB
Args en `/global-scripts/:id/test`	8 args, 16 KB JSON total

Casos de uso

UC-04 — Latch de fallo

Detectar el flanco de subida de una señal de fallo y dejarlo enclavado hasta acuse del operador.

- `FAULT_RAW` (`bool`, lectura del PLC)
- `FAULT_LATCH` (`bool`, internal, RW)
- `FAULT_ACK` (`bool`, internal, RW, lo escribe el botón del operador)

Script onValueChange en `FAULT_RAW`:

```
if (current.value === true && prev.value === false) {
  Induxa.tag.write('FAULT_LATCH', true);
}
```

Script onValueChange en `FAULT_ACK`:

```
if (current.value === true) {
  Induxa.tag.write('FAULT_LATCH', false);
  Induxa.tag.write('FAULT_ACK',   false);  // auto-reset
}
```

UC-05 — Watchdog de comunicaciones (Scheduler)

Si la marca temporal del último valor del PLC tiene más de 30 s, forzar bandera de pérdida de comunicación.

```
// scheduler job, type=clock, interval=5000 ms
const t = Induxa.tag.read('PLC_HEARTBEAT');
const stale = !t || (Date.now() - new Date(t.timestamp).getTime()) > 30000;
Induxa.tag.write('PLC_LINK_LOST', stale);
```

UC-06 — Backup nocturno de recetas

```
// scheduler job, type=calendar, cron='0 2 * * *'  (todos los días a las 02:0
0)
const ts = new Date().toISOString().slice(0,10);
Induxa.tag.write('LAST_RECIPE_BACKUP', ts);
Induxa.log('Recipe backup tag stamped: ' + ts);
// La copia física la hace un global gateway que llama a Induxa.http
// (cuando esté disponible) o publica a MQTT.
```

UC-07 — Función global reutilizable

Convertir un identificador de lote en formato compactado.

```
// global gateway, name = 'formatBatchId'
function (year, lineNo, seq) {
  const yy   = String(year).slice(-2);
  const line = String(lineNo).padStart(2, '0');
  const s    = String(seq).padStart(4, '0');
  return `B${yy}${line}-${s}`;
}
```

Uso desde un script de tag:

```
const id = Induxa.global.formatBatchId(2026, 3, 472);
Induxa.tag.write('CURRENT_BATCH_ID', id);
```

UC-08 — Botón con confirmación (Widget)

Pedir confirmación antes de detener una bomba en marcha.

```
// widget.events.onClick (botón "STOP")
const ok = await Induxa.confirm('¿Detener bomba P-101?');
if (!ok) return;
const r = await Induxa.tag.writeWithContext('PUMP_RUN', false, {
  widgetType: 'button',
  confirmed:  true,
});
if (!r.success) Induxa.notify('No se pudo detener: ' + r.error, 'error');
```

UC-09 — Pantalla con auto-cierre por inactividad

Una pantalla con set-points sensibles vuelve al menú principal tras 5 min sin actividad.

```
// screen.scripts.onIdle  (idle_timeout = 300000)
Induxa.notify('Cerrando por inactividad', 'warn');
Induxa.navigate('Main');
```

UC-10 — Refresh periódico de un KPI

Recalcular y mostrar un KPI cada 30 s mientras la pantalla está visible.

```
// screen.scripts.onRefresh  (refresh_interval = 30000)
const a = Induxa.tag.read('KPI_NUM')?.value || 0;
const b = Induxa.tag.read('KPI_DEN')?.value || 1;
Induxa.tag.write('KPI_RATIO', Math.round((a/b)*100)/100);
```

> Los hooks `onRefresh` y `onIdle` se pausan cuando la pestaña del navegador no está visible y se rearman al volver al foco.

Diagnóstico

Pregunta	Dónde mirar
¿Cuándo falló mi script?	`GET /api/v1/scripts/errors?limit=100` (manageTags)
¿Cuál fue la última ejecución del scheduler job X?	`GET /api/v1/scheduler/jobs/:id/status`
¿Por qué el operador ve la pantalla congelada?	DevTools del navegador, filtro `[HMI]`, `[Screen]`, `[ScreenScript]`
¿El tag se escribió de verdad?	`GET /api/v1/audit/writes` (admin)

Anti-patrones frecuentes

1. **Bucles `while` con condiciones de tag.** El sandbox los matará al timeout, pero se pierden 500 ms de CPU del servidor por nada. Usa un scheduler job con `type=change`.
2. **Scripts gigantes en `onValueChange`.** Refactoriza a un global y llámalo: `Induxa.global.processRecipeStep()`.
3. **Confiar en `Induxa.user.role` para autorizar escrituras.** No es una garantía; el server lo decide. Úsalo solo para deshabilitar UI.
4. **Acumular contadores en `onValueChange`.** Si el PLC oscila, pierdes la cuenta. Usa un scheduler periódico que muestree.
5. **Usar `await fetch()` en un script de tag.** No funciona; el `fetch` no existe en el sandbox del server.

Material técnico de referencia

Referencia técnica de scripts (extendida)

This document is the canonical reference for every place where a user can write JavaScript and have it executed by the platform. It complements `SEL-EXPRESSION.md` (the lightweight expression language used in widget bindings) by covering the *full-JavaScript* surfaces.

Surfaces

Surface	Storage	Runtime	Hooks
Tag event scripts	`tag.scripts.{onValueChange, onQualityChange}`	Server-side `vm` sandbox (`server/script-engine.js`)	`onValueChange`, `onQualityChange`
Scheduler jobs	`project.scheduler.jobs[].script`	Server-side `vm` sandbox (`server/scheduler-engine.js`)	clock / calendar / change / startup
Gateway global	`project.globalScri`	Server-side isolated	callable as

Surface	Storage	Runtime	Hooks
functions	`pts.gateway[]`	`vm` context (`server/global-scripts.js`)	`Induxa.global.<name>(...)` from tag/scheduler scripts
Client global functions	`project.globalScripts.client[]`	Browser `new Function()` (`app/js/global-script-library.js`)	callable as `Induxa.global.<name>(...)` from screen/widget scripts
Screen lifecycle	`screen.scripts.{onOpen, onClose, onRefresh, onIdle}`	Browser (`app/js/screen-script-engine.js`)	`onOpen`, `onClose`, `onRefresh`, `onIdle`
Widget events	`widget.events.{onClick, onDoubleClick, onMouseEnter, onMouseLeave, onRightClick, onChange}`	Browser (`app/js/hmi-script-engine.js`)	DOM-style callbacks

INDUXA API

Server-side (Tag & Scheduler scripts)

```
Induxa.tag.read(path)              → { value, quality, timestamp } | null
Induxa.tag.write(path, value)      → boolean
Induxa.tag.writeMultiple(map)      → boolean      (scheduler only)
Induxa.log(message)                → void         (writes to server log)
Induxa.global.<fn>(...)            → user-defined gateway global
console.log(message)               → alias of Induxa.log
```

`Induxa.db` and `Induxa.http` exist as stubs and currently throw. Do not rely on them.

Browser (Screen & Widget scripts)

```
Induxa.tag.read(name)              → { value, quality, timestamp } | null
Induxa.tag.write(name, value)      → Promise<{ success, error? }>
Induxa.tag.writeWithContext(...)   → Promise<{ success, writeId? }>
Induxa.tag.writeBatch(writes, ctx) → Promise<{ success, results? }>
Induxa.tag.readMultiple(names)     → { name: { value, quality, timestamp } }
Induxa.navigate(screenName)        → void
Induxa.openPopup(viewId, params)   → void
Induxa.notify(message, type)       → void         (toast)
Induxa.confirm(message)            → Promise<boolean>
Induxa.user.{name, role, hasRole, getCurrentUser, getCurrentRole}
Induxa.screen.{name, previous, next, setData(k,v), getData(k)}   (screen scripts only)
Induxa.popup.{params, id, close}                                  (popup scripts only)
Induxa.global.<fn>(...)            → user-defined client global
```

Server-side authorisation is performed against the JWT, not the value of `Induxa.user.role` claimed by the browser. The role exposed in the API is for UI affordance only; do not use it as a security boundary.

Execution model

Server-side (`vm.runInNewContext` / `vm.runInContext`)

- Each invocation runs in a fresh sandbox with a hard timeout (tag scripts: 500 ms, scheduler: configurable up to 30 s).
- The sandbox has **no access** to `require`, `process`, `fs`, `module`, `__dirname`, the host `globalThis`, or any Node built-in.
- Errors are caught, logged, and surfaced via `GET /api/v1/scripts/errors`. Scripts never crash the server.
- Gateway globals are compiled inside their own isolated `vm` context. When called from a tag/scheduler script, they execute in the globals' realm, never in the host realm.

Browser

- Each invocation is wrapped in an async IIFE built with `new Function`.
- Script errors are caught and logged to the browser console; they never break the viewer.
- The browser sandbox has access to whatever the page runtime exposes — be aware that scripts can read tag values, write tags, and call browser APIs (within CSP). Restrict global script CRUD to engineering roles.

Restrictions and gotchas

`async` / `await` and Promises (server-side)

Tag and scheduler scripts run synchronously. The `vm` timeout only covers synchronous execution. If you write:

```
// ✗ DOES NOT WORK on server-side scripts
await new Promise(resolve => setTimeout(resolve, 100));
```

You will get a reference error (`setTimeout` is not in the sandbox) or the returned `Promise` will be ignored — the script returns immediately and the async work is never observed. The same applies to:

- `setTimeout`, `setInterval`, `setImmediate` (not exposed)
- `fetch`, `XMLHttpRequest` (not exposed)
- top-level `await` (parser allows it, but the timeout still only covers the synchronous prelude — callbacks scheduled afterwards are dropped)
- `Promise.then` callbacks scheduled on user-supplied promises — they fire outside the timeout window and may be silently lost when the engine moves on

Stick to synchronous code in tag and scheduler scripts. For long-running work (HTTP calls, DB queries), use a dedicated scheduler job at low frequency or wait for `Induxa.http`/`Induxa.db` to land in a future release.

Browser scripts (screen, widget, client globals) **are** wrapped in an async IIFE so `await` and Promises work as expected — but errors in async branches that escape the IIFE catch are only logged, not surfaced to the operator.

Loop guard

Tag scripts that write to a tag more than 50 times in 1 second are automatically paused for 5 seconds and an entry is written to the script error log. This protects against `A → B → A` oscillations.

Frozen `Induxa.global`

The server-side `Induxa.global` object is deep-frozen — scripts cannot add, remove or replace globals at runtime. Restart the server (or call `rebuild` through the API) to pick up changes. The browser-side equivalent is re-fetched live when the editor saves a change (WebSocket `global_scripts_changed` event).

Size limits

Surface	Hard limit
Tag script (per hook)	10 KB
Global function code	64 KB
Test-call args (`/global-scripts/:id/test`)	8 args, 16 KB JSON

What is NOT supported

- Loading external modules (`require`, `import`)
- Native binary access (`Buffer`, `crypto.randomBytes`, etc.)
- Inter-script messaging (no shared mutable state besides tag values)
- Persistent state across script invocations beyond the project file
- `eval`, `new Function`, `WebAssembly.compile` from inside server scripts *(these still work but compile inside the same isolated `vm` context)*

Diagnostics

- `GET /api/v1/scripts/errors?limit=100` — last N script execution failures (tag, scheduler). Requires `manageTags` permission.
- `DELETE /api/v1/scripts/errors` — clear the buffer.
- `GET /api/v1/scheduler/jobs/:id/status` — last result, duration, error for a scheduler job.
- Server log lines prefixed with `[SCRIPT][...]` or `[Scheduler][...]`.

IEL — INDUXA Expression Language

IEL (INDUXA Expression Language) es un lenguaje de expresiones ligero, sin efectos colaterales, diseñado para incrustarse en *bindings* de widgets, condiciones de transición de máquina de estado, fórmulas de tags calculados, condiciones de alarma y triggers de evento. No es un

sustituto de JavaScript: es una herramienta para expresar reglas declarativas de forma segura, predecible y rápida.

Cuándo usar IEL y cuándo escribir un script

Situación	Lenguaje
Color o visibilidad dinámica de un widget	**IEL**
Cálculo derivado en un tag calculated	**IEL**
Condición de alarma	**IEL**
Lógica con efectos secundarios (escribir tags, navegar, notificar)	Script
Iteración, recursión, estructuras de datos	Script
Comunicación con servicios externos	Script

> IEL **no escribe** tags. Si una expresión necesita "hacer algo", conviértela en un script. IEL es seguro precisamente porque solo *lee* y *transforma*.

Sintaxis

Referencias a tags

Dentro de llaves:

```
{TEMP}              # valor actual (number, string o bool según el tag)
{TEMP.quality}      # 'GOOD' | 'UNCERTAIN' | 'BAD' | 'STALE'
```

Por compatibilidad, un identificador suelto sin operadores se interpreta como `{Tag}`:

```
TEMP                # equivalente a {TEMP}
```

Operadores

Precedencia y asociatividad similares a JavaScript.

Clase	Operadores
Aritméticos	`+ - * / %`
Comparación	`< <= > >= == !=`
Lógicos	`&& || !`
Ternario	`cond ? a : b`
Agrupación	`( … )`

`&&` y `||` son de cortocircuito; `!` es la negación lógica.

Funciones built-in

Función	Resultado
`abs(x)`	valor absoluto
`round(x [, d])`	redondeo con `d` decimales (def. 0)
`floor(x)` / `ceil(x)`	redondeo dirigido
`min(a, b, …)`	mínimo de N args
`max(a, b, …)`	máximo de N args
`clamp(x, lo, hi)`	acotar entre `lo` y `hi`
`format(x, …)`	formato numérico con localización
`concat(a, b, …)`	concatenación de cadenas
`if(cond, a, b)`	equivalente al ternario (legado)

Tipos

IEL es dinámico. Las coerciones siguen las reglas habituales de JavaScript con dos cuidados:

- Comparación con `==` aplica coerción; usa `===` si quieres estrictez, **pero ten en cuenta que IEL no admite `===` ni `!==`**. Para comparar tipos exactos, normaliza con un tag calculated antes.
- Si un tag tiene `quality !== 'GOOD'`, su valor puede ser `null` o un dato obsoleto. Compáralo explícitamente cuando sea relevante:

  ```
  {TEMP.quality} == 'GOOD' && {TEMP} > 50
  ```

Errores y calidad

`IEL.evaluate(...)` **nunca lanza**. Si la expresión falla en tiempo de ejecución (división por cero, función no definida, tag desconocido), devuelve `{ value: null, error: '<mensaje>' }`. Los consumidores internos (gestor de bindings, motor FSM, motor de alarmas) interpretan ese error como una bajada de calidad del downstream: el widget se renderiza en gris, la transición no se cumple, la alarma no dispara.

Casos de uso

UC-11 — Color por umbral en un widget

Binding de la propiedad `fillColor` de un *circle-indicator*:

```
{TEMP} > 80  ? '#CF222E'        // rojo
             : {TEMP} > 50
               ? '#9A6700'      // ámbar
               : '#1A7F37'      // verde
```

UC-12 — Visibilidad condicional

Binding de la propiedad `visible` de un widget *alert-banner*:

```
{FAULT_LATCH} && !{FAULT_ACK}
```

UC-13 — Tag calculated: caudal másico

Si tienes caudal volumétrico en `m3/h` y densidad en `kg/m3`:

```
formula: '{FLOW_VOL} * {DENSITY}'
```

El tag `MASS_FLOW` (también `float32`) se recalcula cada vez que cambia `FLOW_VOL` o `DENSITY`, sin scripts ni jobs.

UC-14 — Condición de alarma combinada

```
{TEMP} > {SET_POINT} + 5 && {ENABLE} && !{MAINT_MODE}
```

(la condición se evalúa en cada cambio de los tags referenciados; el motor de alarmas se ocupa del debounce y de la confirmación según ISA-18.2).

UC-15 — Transición FSM con guarda

Una transición del estado `RUNNING` al estado `STOP` está habilitada solo si:

```
{STOP_REQ} || {FAULT} || ({TEMP.quality} == 'BAD')
```

UC-16 — Etiqueta dinámica de un widget de texto

```
concat('Lote ', {BATCH_ID}, ' — ', round({YIELD}, 1), ' %')
```

renderiza algo como `Lote B26-03-0472 — 87.4 %`.

Limitaciones explícitas

IEL **no soporta** ninguno de los siguientes — son por diseño, para mantener el lenguaje seguro y rápido:

- Asignación (`=`, `+=`, …)
- Bucles (`for`, `while`)
- Definición de funciones o variables
- `===` / `!==`
- Plantillas literales (backticks)
- Acceso a propiedades de objetos arbitrarios (solo `tag.quality` está permitido sobre referencias de tag)
- Llamadas a métodos prototípicos (`'X'.toUpperCase()`)

Si una expresión empieza a sentirse limitada, eso suele ser señal de que la lógica pertenece a un script, a un global o a una pipeline de tag calculated.

Material técnico de referencia

Referencia técnica del lenguaje SEL/IEL

> Compact expression language for widget bindings, FSM conditions, calculated tags, alarm logic and event triggers. Source: `app/js/sel-engine.js`. Tests: `test/sel-engine.test.js`.

Where SEL is used

Surface	Example
Widget bindings (expr)	`{TEMP} > 50 ? '#FF0000' : '#00FF00'`
FSM transitions	`{ENABLE} && !{FAULT}`
Calculated tags	`({A} + {B}) / 2`
Alarm conditions	`{TEMP} > {SET_POINT}`
Event triggers	`{LEVEL} < 20 && {PUMP_RUN}`

Tag references

Inside `{ }`. The unwrapped value is whatever the tag engine has cached:

```
{TEMP}              # current value (number / string / bool)
{TEMP.quality}      # 'GOOD' | 'BAD' | 'UNCERTAIN'
```

For backward compatibility, a **bare identifier with no operators** is treated as `{TagName}`:

```
TEMP                # equivalent to {TEMP}
```

Operators

Standard JavaScript-ish precedence and associativity:

Class	Operators
Arithmetic	`+ - * / %`
Comparison	`< <= > >= == !=`
Logical	`&& || !`
Ternary	`cond ? a : b`
Grouping	`( … )`

`&&` and `||` short-circuit. `!` is logical not.

Built-in functions

Function	Result
`abs(x)`	Absolute value
`round(x [, d])`	Round to `d` decimals (default 0)

Function	Result
`floor(x)`	`Math.floor`
`ceil(x)`	`Math.ceil`
`min(a, b, …)`	Minimum of N args
`max(a, b, …)`	Maximum of N args
`clamp(x, lo, hi)`	`min(max(x, lo), hi)`
`format(x, …)`	Locale-aware number formatting
`concat(a, b, …)`	String concatenation
`if(cond, a, b)`	Same as `cond ? a : b` (legacy)

Compile / evaluate

```
const c = SEL.compile('{TEMP} > 50');
const result = SEL.evaluate(c, { TEMP: 60 });
// result === { value: true, error: null }

const deps = SEL.extractDeps('{TEMP} > {SET_POINT} && {ENABLE}');
// deps === ['TEMP', 'SET_POINT', 'ENABLE']
```

`evaluate()` is **total**: it never throws. On runtime error it returns `{ value: null, error: '<message>' }`. Callers (BindingManager, FSM engine, alarm engine) are expected to handle the `.error` field gracefully — most surface it as a quality=BAD condition on the downstream tag.

`extractDeps()` is used by `BindingManager` to build an O(1) reverse index of `tag → widgets that depend on it`.

Examples by use case

Widget colour binding

```
{TEMP} > 80  ? '#CF222E'          // red
             : {TEMP} > 50
               ? '#9A6700'        // amber
               : '#1A7F37'        // green
```

FSM transition with debounce hint

```
{ENABLE} && !{FAULT} && {TIMER_ELAPSED} > 5
```

(SEL has no native debounce — use the FSM's `tag_duration` for that.)

Calculated average tag

```
( {SENSOR_A} + {SENSOR_B} + {SENSOR_C} ) / 3
```

Alarm with deadband

```
{LEVEL} > {ALARM_HI} || {LEVEL} < {ALARM_LO}
```

Conditional unit conversion

```
{TEMP_UNIT} == 'F'
  ? round({TEMP_C} * 1.8 + 32, 1)
  : {TEMP_C}
```

Error handling

Compile error	Symptom
Empty string	`compile('').error === 'Empty expression'`
Syntax error	`compile('a + + b').fn === null`

Runtime error	Result
Tag missing in context	`value === undefined` (caller decides)
Division by zero	`Infinity` (JS native)
`null + 5`	`5` (JS coercion)
Function arity mismatch	`error: '...'`, `value: null`

The FSM engine treats any `error` as a transition that does NOT fire (the FSM stays in its current state and emits an `_ERROR` pulse).

Limitations

- **No assignment**, no statements — only expressions.
- **No loops, no functions, no variables.**
- **No regex** (use a calculated tag if you really need one — fence it off in JS code).
- **Strings:** literal `"..."` or `'...'`. No template strings, no interpolation. Use `concat()`.
- **Booleans:** `true` / `false` are reserved names — they evaluate to the bool primitives, not tag refs.

Performance notes

- A compiled expression is a function reference; reuse it. The Editor caches one per binding.
- `extractDeps` is the gating cost for binding wiring on screen load. Both `compile` and `extractDeps` are O(n) over expression length and fast (sub-millisecond for typical bindings).
- Avoid putting heavy work into a calculated tag formula that runs every scan-class tick — prefer a debounced HMI script.

Widgets

Un **widget** es un componente visual reutilizable del Editor que renderiza datos en pantalla, recibe interacción del operador o ambos. Cada widget tiene una caja de propiedades, un sistema de bindings que lo conecta a tags y un conjunto de eventos a los que se les puede adjuntar lógica (Capítulo 3).

Catálogo

Categoría	Widgets
Indicadores numéricos	`numeric-display`, `counter-odometer`, `gauge`, `bar-progress`
Indicadores discretos	`led-indicator`, `toggle-switch`, `state-timeline`
Entradas de operador	`button-momentary`, `button-latching`, `button-multistate`, `numeric-entry`, `text-input`, `slider`, `dropdown`, `checkbox`, `radio-group`, `segmented`, `datetime-picker`, `nav-button`
Gráficas	`bar-chart`, `pie-chart`, `xy-chart`, `data-logger`
Equipos	`equipment-pump`, `equipment-valve-control`, `equipment-flow-meter`, `tank-level`
Alarma	`alarm-banner`, `alarm-list`
Forma	`rect-shape`, `ellipse-shape`, `triangle-shape`, `diamond-shape`, `hexagon-shape`, `star-shape`, `arrow-block`, `cloud-shape`, `cylinder-shape`, `line-shape`, `label-widget`, `text-display`
Estado / sistema	`status-history`, `status-table`, `clock-display`, `recipe-frame`, `geo-map`, `spinner`

Cada widget se describe individualmente en § Referencia de widgets al final del capítulo. Sus tamaños por defecto y la lista completa de propiedades también están disponibles en el panel *Widgets* del Editor (a la izquierda del lienzo).

> El gráfico de tendencia (`trend`) no aparece en la tabla porque no se publica como HTML — es un widget de **canvas** implementado en `app/js/trend-chart.js`. Su comportamiento y propiedades se cubren en § Gráfico de tendencia (`trend`).

Anatomía de un widget en el proyecto

Un widget se serializa así dentro del `.syk`:

```
{
  "id":      "w_abc123",
  "type":    "numeric-display",
  "x": 120, "y": 80, "w": 180, "h": 56,
  "props": {
    "label":     "Temperatura",
    "unit":      "°C",
    "decimals":  1,
    "fillColor": "#FFFFFF"
  },
  "bindings": {
    "value":     { "mode": "static", "tag": "REACTOR_TEMP" },
    "fillColor": { "mode": "expr",   "expr": "{REACTOR_TEMP} > 80 ? '#FCE7E7'
 : '#FFFFFF'" }
  },
  "events": {
    "onClick": "Induxa.navigate('ReactorDetail');"
  }
}
```

Bindings

Los bindings conectan una propiedad del widget a una fuente de datos. Hay **tres modos** disponibles:

Modo	Cuándo usarlo	Ejemplo
`static`	enlazar 1:1 a un tag	`{ mode:'static', tag:'TEMP' }`
`expr`	derivar del valor de uno o más tags	`{ mode:'expr', expr:'{TEMP} > 50 ? "RED" : "GREEN"' }`
`state`	mapear el valor del tag a un valor literal por estado	`{ mode:'state', tag:'PUMP_STATE', map:{ 0:'Stopped', 1:'Running', 2:'Fault' } }`

> El modo `expr` usa **IEL** (Capítulo 4). No puede tener efectos secundarios; si necesitas lógica con escritura o navegación, usa un script de evento del widget.

Calidad y bindings

Si el tag enlazado está en calidad `BAD` o `STALE`, el widget se renderiza en estado degradado:

- `numeric-display` muestra `--` y un punto rojo en la esquina.
- `gauge` y `bar-progress` se renderizan con la barra gris.
- `led-indicator` muestra el color “off” con borde rojo.

Este comportamiento se puede sobrescribir por widget en sus propiedades `qualityPolicy` (`hide`, `placeholder`, `lastKnown`).

Eventos del widget

Cada widget puede declarar scripts en uno o varios de estos eventos:

Evento	DOM equivalente	Cuándo se dispara
`onClick`	`click`	clic / toque sobre el widget
`onDoubleClick`	`dblclick`	doble clic
`onMouseEnter`	`mouseenter`	el cursor entra (no aplica en touch)
`onMouseLeave`	`mouseleave`	el cursor sale
`onRightClick`	`contextmenu`	clic derecho (se suprime el menú nativo)
`onChange`	`change`	el valor del control cambia (toggles, inputs, sliders, dropdowns)

El script tiene acceso a la API completa de `Induxa.*` en el navegador (Capítulo 3) más el `eventData` específico del evento:

```
// onClick — eventData = { x, y } (coordenadas locales)
Induxa.notify('Click en (' + eventData.x + ',' + eventData.y + ')', 'info');

// onChange — eventData = { value, checked }
if (eventData.checked) Induxa.tag.write('AUTO_MODE', true);
```

> En widgets compuestos (los que tienen `<input>` interno), el evento onChange se engancha al control de formulario, no al contenedor del widget. La de-duplicación impide doble disparo si el control hace bubble del evento.

Acciones predefinidas (sin escribir script)

Los widgets de tipo botón ofrecen acciones predefinidas vía la propiedad `clickAction`. Son la forma rápida de cubrir el 80 % de los casos sin tocar JavaScript:

`clickAction`	Comportamiento
`none`	no hace nada (útil para widgets puramente decorativos)
`write_tag`	escribe `valueOn` (o `valueOff`) en `targetTag`
`toggle_tag`	invierte el valor booleano del tag
`momentary`	escribe `true` mientras se mantiene presionado, `false` al soltar

clickAction	Comportamiento
navigate	abre la pantalla targetScreen
open_popup	abre el popup targetPopup con popupParams
run_script	invoca el evento onClick declarado por el usuario
acknowledge_alarm	confirma la alarma targetAlarm

Cuando clickAction = run_script, el onClick del usuario tiene control total. Para todas las demás acciones, el motor del Viewer ya respeta auditoría (writeWithContext), permisos y notificaciones de error.

Caso de uso

UC-17 — Pulsador con confirmación, auditoría y feedback

Un botón que detiene una bomba pidiendo confirmación, registrando la acción con widgetType para que aparezca correctamente en el log de auditoría, y mostrando un toast con el resultado.

Propiedades del widget (button-momentary):

Propiedad	Valor
label	STOP
clickAction	run_script
confirmRequired	true
confirmMessage	¿Detener bomba P-101?
minRole	operator

Binding de enabled:

```
{PUMP_RUN} == true && {MAINT_MODE} == false
```

Script onClick:

```
const ok = await Induxa.confirm('¿Detener bomba P-101?');
if (!ok) return;

const r = await Induxa.tag.writeWithContext('PUMP_RUN', false, {
  widgetType: 'button-momentary',
  confirmed:  true,
});

if (r.success) Induxa.notify('Bomba detenida (#' + r.writeId + ')', 'success
');
else           Induxa.notify('Error: ' + r.error, 'error');
```

> **Permisos.** `minRole` solo controla la afordancia visual. La autorización real la hace el server: si el usuario no tiene el permiso `operate`, la llamada a `writeWithContext` devolverá `{ success: false, error: 'Insufficient permissions' }` y el botón no podrá saltarse esa barrera ni aunque el binding de `enabled` se manipule en DevTools.

Referencia de widgets

Cada entrada describe **qué hace** el widget, un **caso típico** y las **propiedades clave** que el ingeniero suele configurar. Las propiedades universales (`x`, `y`, `w`, `h`, `rotation`, `visible`, `enabled`, `minRole`, `tooltip`) se omiten en cada ficha — están disponibles para todos los widgets.

Indicadores numéricos

`numeric-display`

Lectura numérica con etiqueta, unidad y rangos de color condicionales. Estado degradado automático ante calidad `BAD` / `STALE`. - *Caso típico*: temperatura, presión o caudal en una pantalla de proceso. - *Clave*: `value` (binding), `unit`, `decimals`, `colorRanges`, `borderRadius`, `shadow`.

`counter-odometer`

Contador / odómetro con formato (separadores, dígitos fijos, ceros a la izquierda) y opción de mostrar la tasa de cambio. - *Caso típico*: producción acumulada, horas de funcionamiento, consumo total. - *Clave*: `value`, `digits`, `unit`, `showRate`, `rateUnit`.

`gauge`

Indicador de aguja en arco 3/4 con rangos de color y marcador opcional de setpoint. - *Caso típico*: presión de caldera, RPM de motor, nivel de tanque. - *Clave*: `value`, `min`, `max`, `colorRanges`, `setpoint`.

`bar-progress`

Barra de progreso horizontal o vertical con rangos y marcador de setpoint. - *Caso típico*: porcentaje de llenado, avance de receta, eficiencia OEE. - *Clave*: `value`, `min`, `max`, `orientation`, `colorRanges`, `setpoint`.

Indicadores discretos

`led-indicator`

LED ON/OFF con formas (círculo, cuadrado, anillo) y colores por estado. Soporta parpadeo. - *Caso típico*: marcha de un equipo, alarma activa, comunicación OK. - *Clave*: `value`, `shape`, `colorOn`, `colorOff`, `blink`.

toggle-switch

Interruptor binario tipo switch deslizante con confirmación opcional antes de escribir. - *Caso típico*: activar/desactivar luces, ventiladores o bypass. - *Clave*: `tagWrite`, `confirmRequired`, `labelOn`, `labelOff`.

state-timeline

Línea de tiempo tipo Gantt con cambios de estado de uno o varios tags en una ventana móvil. - *Caso típico*: registro de modos de operación, OEE por turno, MTBF. - *Clave*: lista `tracks` (cada una con tag y mapa de estados a colores), `timeWindow`.

Entradas de operador

button-momentary

Botón que escribe `true` al pulsar y `false` al soltar (hombre-muerto). - *Caso típico*: jog manual de un motor, parada momentánea. - *Clave*: `clickAction = momentary` (o script `onClick`), `targetTag`, `valueOn`, `valueOff`, `label`, `color`.

button-latching

Conmutador ON/OFF. Cada pulsación invierte el estado del tag. - *Caso típico*: habilitar/deshabilitar un sistema, fijar modo automático. - *Clave*: `clickAction = toggle_tag`, `targetTag`, `confirmOn`, `confirmOff`, `colorOn`, `colorOff`.

button-multistate

Botón que cicla por N estados definidos por el operador. Permite estados sólo lectura que se saltan al ciclar. - *Caso típico*: selector de modo (`Manual` / `Auto` / `Remoto` / `Mantenimiento`). - *Clave*: `tagRead`, `tagWrite`, tabla `states` con `value`, `label`, `color`, `readonly`.

numeric-entry

Campo numérico con validación de rango y confirmación opcional al escribir. - *Caso típico*: introducir setpoint, dosis de receta. - *Clave*: `tagWrite`, `min`, `max`, `decimals`, `unit`, `confirmRequired`.

text-input

Entrada de texto con prefijo (label) e icono. Escribe al perder el foco o al pulsar Enter. - *Caso típico*: notas de operador, código de lote, comentario en evento. - *Clave*: `tagWrite`, `placeholder`, `maxLength`, `prefix`.

slider

Slider analógico horizontal o vertical. Sólo escribe al soltar el pulgar — nunca durante el arrastre — para no saturar la red. - *Caso típico*: setpoint de velocidad, apertura manual de válvula. - *Clave*: `tagWrite`, `min`, `max`, `step`, `orientation`.

dropdown

Lista desplegable con búsqueda, agrupación de opciones y puntos de color por entrada. - *Caso típico*: seleccionar receta, operario activo, turno de producción. - *Clave*: `tagWrite`, lista `options` (`value`, `label`, `group`, `color`).

checkbox

Casilla binaria con soporte de estado indeterminado. - *Caso típico*: lista de permisos, opciones de receta. - *Clave*: `tagWrite`, `label`.

radio-group

Selección única de N opciones, layout horizontal o vertical. - *Caso típico*: tipo de producto, perfil de proceso. - *Clave*: `tagWrite`, lista `options`, `orientation`.

segmented

Selector tipo pestañas horizontales — todas las opciones siempre visibles. - *Caso típico*: alternar vistas, rangos de tiempo (1h / 8h / 24h). - *Clave*: `tagWrite`, lista `options`.

datetime-picker

Selector temporal en modos `date`, `time`, `datetime` o `range`. - *Caso típico*: filtros de históricos, programación de tareas. - *Clave*: `tagWrite`, `mode`, `format`.

nav-button

Botón de navegación a otra pantalla, vista o URL externa. - *Caso típico*: menú principal, retroceso, abrir popup de detalle. - *Clave*: `clickAction = navigate` o open_popup, `targetScreen`, `targetPopup`, `popupParams`, `icon`, `label`.

Gráficas

bar-chart

Barras verticales para comparar varios tags simultáneamente. - *Caso típico*: producción por línea, consumo por zona. - *Clave*: lista `series` (cada una con `tag`, `label`, `color`), `min`, `max`.

pie-chart

Tarta o donut con valor central opcional. - *Caso típico*: distribución de tiempos (run / stop / fault), reparto de producción. - *Clave*: `series`, `donut`, `centerValue`.

xy-chart

Dispersión XY con línea de tendencia, curvas operativas y regiones opcionales. - *Caso típico*: curva característica de bomba, scatter de calidad, mapa de operación. - *Clave*: `xTag`, `yTag`, `regressionLine`, `referenceCurves`, `regions`.

data-logger

Tabla en tiempo real con buffer circular, exportación CSV y columnas configurables. - *Caso típico*: registro de eventos de proceso, log de operación. - *Clave*: lista `columns` (con tag/expresión, header, ancho), `bufferSize`, `exportEnabled`.

Equipos

equipment-pump

Bomba animada que refleja en tiempo real estado de marcha, fallo y velocidad. - *Caso típico*: bombas centrífugas en P&ID animado. - *Clave*: `tagRunning`, `tagFault`, `tagSpeed`, `orientation`.

equipment-valve-control

Válvula de control con animación de apertura, indicación de marcha y fallo. - *Caso típico*: válvulas modulantes en lazo de control. - *Clave*: `tagOpening` (0–100%), `tagFault`, `tagRunning`, bodyType.

equipment-flow-meter

Caudalímetro con lectura de caudal y totalizador integrado. - *Caso típico*: medidores en línea, balances de masa. - *Clave*: `tagFlow`, `tagTotalizer`, `unitFlow`, `unitTotal`.

tank-level

Visualización de nivel en tanque vertical u horizontal con animación de líquido. - *Caso típico*: depósitos, silos, reactores. - *Clave*: `value`, `min`, `max`, `liquidColor`, `orientation`.

Alarma

alarm-banner

Banner que muestra el estado de una alarma única (activa, reconocida, normal). - *Caso típico*: alarma crítica destacada en cabecera, parada de emergencia. - *Clave*: `alarmTag`, `severity`, `ackButton`.

alarm-list

Lista de eventos de alarma con histórico, *shelving* (silenciar) y exportación. - *Caso típico*: pantalla principal de alarmas, histórico para análisis post-incidente. - *Clave*: filtros por área y severidad, columnas, `enableAck`, `enableShelve`.

Forma

Todas las formas soportan **rotación**, **trazo configurable** y **color por tag** (modos `static`, `expr` y `state`).

rect-shape

Rectángulo o cuadrado con relleno, trazo, radio de esquinas y sombra. - *Caso típico*: marcos de zona, fondos, paneles, equipos genéricos. - *Clave*: `fill`, `stroke`, `strokeWidth`, `radius`, `shadow`.

ellipse-shape

Elipse o círculo con relleno (sólido o gradiente), trazo y sombra. - *Caso típico*: nodos de proceso, indicadores circulares, decoración. - *Clave*: `fill`, `gradient`, `stroke`, `strokeWidth`.

triangle-shape

Triángulo con orientación (arriba/abajo/izq/der) y colores por estado. - *Caso típico*: indicadores de dirección, símbolos de advertencia. - *Clave*: `fill`, `stroke`, `orientation`.

diamond-shape

Rombo / diamante. Habitual en nodos de decisión y leyendas P&ID. - *Caso típico*: punto de decisión en flowchart, símbolo de leyenda. - *Clave*: `fill`, `stroke`.

hexagon-shape

Hexágono *flat-top* o *pointy-top*. - *Caso típico*: nodos de red, celdas de panel, símbolos custom. - *Clave*: `fill`, `stroke`, `orientation`.

star-shape

Estrella de N puntas con ratio interno configurable. - *Caso típico*: marca de favorito, indicador de calidad, decoración. - *Clave*: `points`, `innerRatio`, `fill`, `stroke`.

arrow-block

Flecha en bloque con varios estilos: recta, doble, en codo o en U. - *Caso típico*: flujo de proceso destacado, dirección de transporte. - *Clave*: `style` (`straight` / `double` / `corner` / `u-turn`), `direction`, `fill`.

cloud-shape

Nube — anotaciones, agrupaciones, representación de red/internet. - *Caso típico*: marcar zona de comentario, símbolo de WAN o cloud. - *Clave*: `fill`, `stroke`.

cylinder-shape

Cilindro vertical u horizontal (tanque o vasija de pared simple, sin animación de nivel — para eso usar `tank-level`). - *Caso típico*: representación esquemática de un tanque en P&ID. - *Clave*: `fill`, `stroke`, `orientation`.

line-shape

Línea con trazo configurable, flechas en extremos y animación de flujo opcional. - *Caso típico*: tuberías, conexiones lógicas, flechas de proceso. - *Clave*: `stroke`, `strokeWidth`, `arrowStart`, `arrowEnd`, `flowAnimation`.

label-widget

Etiqueta de texto estática o dinámica. Soporta fondo, borde lateral y alineación. - *Caso típico*: títulos de sección, cabeceras, anotaciones de pantalla. - *Clave*: `text` (estático o expresión), `fontSize`, `bgColor`, `borderLeftColor`, `align`.

text-display

Texto multi-estado: por cada valor del tag muestra texto + icono + color. - *Caso típico*: estado textual de un equipo (`STOPPED` / `RUNNING` / `FAULT`). - *Clave*: `tagRead`, tabla `states` con `value`, `text`, `icon`, `color`.

Estado / sistema

status-history

Grid histórico de estados de tag con cálculo automático de uptime. - *Caso típico*: disponibilidad de equipos día a día, % uptime mensual. - *Clave*: lista `tags`, `period` (día/semana/mes), `colorMap`.

status-table

Tabla de tags con columnas: valor, unidad, estado, timestamp. - *Caso típico*: resumen de variables clave, panel de instrumentos. - *Clave*: lista `tags`, columnas visibles, orden.

clock-display

Reloj en tiempo real con formato y zona horaria configurables. - *Caso típico*: cabecera de pantalla, sello horario en partes de producción. - *Clave*: `format` (`HH:mm:ss`, `dd/MM/yyyy HH:mm`, …), `timezone`.

recipe-frame

Visor embebido de una *recipe* o frame externo dentro de la pantalla. - *Caso típico*: mostrar tabla de receta activa, embeber un mini-dashboard. - *Clave*: `recipeId` o `frameUrl`, `refreshInterval`.

geo-map

Mapa geoespacial con marcadores agrupados (clustering) y filtros por estado. - *Caso típico*: flotas distribuidas, plantas multi-sitio, redes hídricas. - *Clave*: lista `sites` (lat, lon, estado, tags), `tileLayer`, `clusterRadius`.

spinner

Numérico con botones +/− y paso configurable. Soporta flechas de teclado. - *Caso típico*: ajuste fino de un parámetro, contadores manuales. - *Clave*: `tagWrite`, `min`, `max`, `step`, `unit`.

Gráfico de tendencia (`trend`)

A diferencia del resto del catálogo, el widget `trend` está **renderizado por canvas** y vive en `app/js/trend-chart.js`. No tiene manifest HTML; sus propiedades se declaran en `app/js/widget-prop-map.js` y `app/js/widget-prop-schema.js`. Aparece en la paleta del Editor como un widget más, pero su *engine* de redibujado es independiente del `UIWidgetEngine`.

Es la herramienta principal de análisis de proceso del Viewer y la única que se invoca con frecuencia desde el operador. Soporta:

- Múltiples plumas (`traces`), cada una con tag, color y escala Y propia.
- Ventana de tiempo móvil (live) o fija (rango histórico).
- Zoom, *pan*, reset, pausa, exportación a CSV/PNG.
- Cursor con valores instantáneos por pluma.
- Toolbar configurable (color de fondo por defecto `#F6F8FA`), tipografía propia.

> **Rendimiento.** El `trend` mantiene su propio *scheduler* de redibujado. Nunca lo invoques desde un evento de tag (`onUpdate` de binding o script en evento) — agendará repintados redundantes y bajará el FPS. Para forzar un refresco usa `Induxa.trend.refresh(widgetId)`.

Propiedades clave

Propiedad	Descripción
`traces`	Lista de plumas. Cada una: `{ tag, label, color, yAxis, lineWidth, smoothing }`
`timeWindow`	Ventana visible (`'live:5m'`, `'live:1h'`, `'fixed'` con `from`/`to`)
`yAxes`	Lista de ejes Y. Cada uno: `{ id, min, max, autoScale, label, side }`
`gridStyle`	`solid` / `dashed` / `none`, color de cuadrícula
`toolbarBg`	Color de fondo de la barra de herramientas (default `#F6F8FA`)
`cursorEnabled`	Habilita el cursor de inspección sobre la gráfica
`exportEnabled`	Habilita los botones de exportación CSV/PNG

Caso de uso

UC-18 — Trend de tres variables con ventana móvil

Una pantalla de control de reactor que necesita mostrar temperatura, presión y caudal de alimentación en una sola gráfica con escala Y independiente para cada una.

```
{
  "id":   "w_trend_reactor",
  "type": "trend",
  "x": 16, "y": 320, "w": 800, "h": 320,
  "props": {
    "timeWindow": "live:30m",
    "traces": [
      { "tag": "REACTOR_TEMP",     "label": "T (°C)",   "color": "#E11", "yAx
is": "y1" },
      { "tag": "REACTOR_PRESSURE", "label": "P (bar)",  "color": "#06C", "yAx
is": "y2" },
      { "tag": "FEED_FLOW",        "label": "Q (m³/h)", "color": "#0A0", "yAx
is": "y3" }
    ],
    "yAxes": [
      { "id": "y1", "min": 0,  "max": 200, "side": "left",  "label": "Tempera
tura" },
      { "id": "y2", "min": 0,  "max": 10,  "side": "right", "label": "Presión
" },
      { "id": "y3", "min": 0,  "max": 50,  "side": "right", "label": "Caudal"
 }
    ],
    "toolbarBg":     "#F6F8FA",
    "cursorEnabled": true,
    "exportEnabled": true
  }
}
```

Material técnico de referencia

Sistema de diseño visual

Status: v1 · 2026-05-05 **Reference set:** `data/projects/DEMO_INDUFAR_FARMA_12_assets/` (5 SVG equipment widgets — mezclador, granulador, recubridor, secador, compresor).

This manual codifies the visual language of the equipment SVGs from the INDUFAR Farma demo so future SVG widgets share the same look. Every section answers a single question: *what choice should I make here?*

---

1. Anatomy of a widget

Every equipment SVG follows the same skeleton:

```
<svg viewBox="0 0 W H"
     data-tag-run="" data-tag-fault="" data-state="running">
  <defs>
    <!-- gradients -->
```

```
    <!-- drop-shadow filter -->
    <!-- arrow marker -->
  </defs>
  <style>
    /* keyframes + state classes */
  </style>
  <g id="<prefix>-root" class="<prefix>-running">
    <!-- visual layers (back→front) -->
  </g>
  <script><![CDATA[ /* state machine */ ]]></script>
</svg>
```

Mandatory contract (the editor / viewer rely on it): - Root `<g>` carries a class `<prefix>-{running|stopped|fault}`. - A `#<prefix>-status-dot` element exists; the inline script paints its `fill`. - Inline script exposes `root.init / root.putValue / root.update`. - A `MutationObserver` watches `data-state` on the SVG root for editor preview.

Prefix: the equipment tag in lowercase (`r101`, `rc101`, `sec101`, `cp101`, `gr101`). Every `id`, `class`, gradient and animation name is prefixed so two instances of the same widget can coexist on a screen.

---

2. Canvas & viewBox

Equipment shape	viewBox	Aspect
Tall vessels (mixers, granulators)	`0 0 260 420`	portrait 1 : 1.6
Square housings (drum coater)	`0 0 340 320`	near-square
Horizontal beds (dryers)	`0 0 420 260`	landscape 1.6 : 1
Mid-form machines (compressor)	`0 0 300 340`	near-square portrait

Rule of thumb: pick the rectangle that fits the equipment's natural silhouette. Don't pad with empty viewBox — the editor scales the SVG to the widget's container, so the viewBox should be a tight frame.

---

3. Color tokens

All values are taken verbatim from the reference SVGs.

3.1 Vessel body & pipes (the "industrial blue-grey" family)

Token	Value	Use
`--vb-stop-1`	`#D8E4F4`	vessel body gradient stop 1

Token	Value	Use
		(left edge / shadow)
`--vb-stop-2`	`#F4F8FF`	vessel body gradient stop 2 (highlight)
`--vb-stop-3`	`#F0F6FF`	vessel body gradient stop 3 (highlight)
`--vb-stop-4`	`#D0DCEC`	vessel body gradient stop 4 (right edge / shadow)
`--shell-stroke`	`#98A8C0`	primary outline (vessel walls, frames)
`--shell-stroke-dark`	`#8898B8`	darker outline (top/bottom ellipses)
`--detail-stroke`	`#7888A8`	inner / fitting details
`--pipe-fill`	`#C0CCDF`	pipe / duct rectangles
`--pipe-fill-light`	`#B8C8DC`	flange faces, support beams
`--pipe-fill-soft`	`#A8B8D0` / `#A8B8CC`	non-load-bearing fittings, bolts

3.2 Process media

Token	Value	Use
`--liquid-1`	`#C8DEFF`	liquid mid-tone
`--liquid-2`	`#D8EAFF`	liquid highlight
`--liquid-3`	`#BCD4F8`	liquid shadow
`--liquid-wave`	`#8ABCF0`	wave stroke (1.2px)
`--bubble`	`#88B8F0`	bubble stroke (1px)
`--powder-1`	`#D8C890`	powder / granule particle (warm)
`--powder-2`	`#C8B880`	powder darker variant
`--tablet-fill`	gradient `#F8FAFF → #E0ECFF`	tablet/pellet body

3.3 Action accent — #3D52D5 (INDUXA signature blue)

Reserved for **anything the operator's eye should track**: - Flow lines (`stroke-dasharray="5,3"`, animated) - Valve diamonds (white fill, 1.8px stroke) - Arrow markers - Drop-shadow tint (`flood-color` on `feDropShadow`) - Outlet pipes (4px stroke, round linecap)

Never use `#3D52D5` for static decoration — it loses meaning.

3.4 Hot / heating (yellow-amber family)

Used for steam jackets, hot-air ducts, heater coils, drying zones.

Token	Value	Use

Token	Value	Use
`--hot-fill-1`	`#E8C060`	jacket gradient stop 1
`--hot-fill-2`	`#F4D888`	jacket gradient stop 2 / highlight
`--hot-fill-soft`	`#EED080` / `#F8D888`	hot-air ducts, heater coils
`--hot-stroke`	`#C89828`	warm element outline
`--hot-detail`	`#B07018` / `#B07820`	coil ribs (0.8px, opacity 0.6-0.7)
`--hot-fastener`	`#D4A030` / `#A07820`	jacket bolts (4px circle)

3.5 Motor (always green, always animated)

Token	Value	Use
`--motor-fill-1`	`#28A055`	motor body gradient top
`--motor-fill-2`	`#1A7840`	motor body gradient bottom
`--motor-stroke`	`#166030`	motor housing outline (1.5px)
`--motor-text`	`#FFFFFF`	“M” letter (size 13, weight 700, monospace)
`--motor-readout`	`#80E8A8`	sub-text (RPM, etc — size 7, monospace)
`--motor-shaft`	`#1A7840`	output shaft stub

The motor block is **always rendered with `feDropShadow`** (token below).

3.6 Status & state colors

State	Status dot fill	Behavior
`running`	`#1A7A40` (deep green)	gentle 2s opacity pulse
`stopped`	`#8898B8` (cool grey)	static
`fault`	`#BE2E18` (alarm red)	0.5s `step-end` blink

Status dot is always `r="7"`, `stroke="#FFFFFF"`, `stroke-width="1.5"`, positioned in a quiet area (top-left or top-right corner of the equipment, never overlapping the process flow).

3.7 Surface decoration

Token	Value	Use
`--shine-white`	`white` at `opacity 0.14 - 0.30`	top-of-cylinder highlight
`--shadow-soft`	`feDropShadow dy=2 std=3-4 flood=#3D52D5 op=.1`	motor, main housings

---

4. Stroke & geometry

Element	stroke-width
Main vessel outline	`1.5`
Inner ellipse / detail outline	`1` to `1.2`
Decorative coil ribs / scale lines	`0.8` (opacity ~`0.6`)
Action accent (outlet pipes, valve frames)	`1.8` to `4`
Status dot border	`1.5`

Border-radius (rx) on rects: | Element | rx | |—|—| | Outer machine frame | `6 - 8` | | Inner frame, ducts | `3 - 4` | | Pipes, supports, flanges | `2 - 3` | | Small fittings (bolts, sensors) | `2` |

Linecaps: `round` for accent pipes (the 4px outlet lines); default `butt` everywhere else.

---

5. Animation primitives

Every animation lives in a `<style>` block and is gated by the root class `.<prefix>-running` / `.<prefix>-stopped`. The CSS template:

```
@keyframes <prefix>-flow-dn   { to { stroke-dashoffset: -16; } }
@keyframes <prefix>-flow-up   { to { stroke-dashoffset:  16; } }
@keyframes <prefix>-rotate    { to { transform: rotate(360deg); } }
@keyframes <prefix>-blink     { 0%,100%{opacity:1} 50%{opacity:.35} }
@keyframes <prefix>-wave      { /* path morph or opacity pulse */ }
@keyframes <prefix>-particle  { /* translateY+opacity */ }
@keyframes <prefix>-spray     { 0%,100%{opacity:.15} 50%{opacity:.5} }

.<prefix>-running .<prefix>-flow-line { animation: <prefix>-flow-dn .7s linea
r infinite; }
.<prefix>-stopped .<prefix>-flow-line { animation: none; }
.<prefix>-fault   #<prefix>-status-dot { animation: <prefix>-blink .5s step-e
nd infinite; }
```

5.1 Standard durations

Animation	Duration	Easing
Slow rotation (drum, turntable)	**6s – 7s**	`linear`
Fast rotation (impeller, motor shaft)	**0.5s – 1s**	`linear`
Pipe flow (dasharray shift)	**0.6s – 0.8s**	`linear`
Liquid wave / heat wave	**2s – 3s**	`ease-in-out`
Bubble travel	**2s – 3s**	n/a (`animateMotion`)
Particle / powder / spray	**1.1s – 1.8s**	`ease-in-out`
Status pulse (running)	**2s**	n/a (smooth opacity)
Status blink (fault)	**0.5s**	`step-end`

Stagger: when there are multiple instances (3 bubbles, 6 particles, 2 spray lines), use `style="animation-delay: .Ns"` increments of `0.1s - 0.5s` so they don't pulse in sync.

5.2 Stop behavior

`.<prefix>-stopped` rules MUST set `animation: none` on every animated class. For elements where freezing mid-cycle looks weird, set a static fallback opacity (e.g. `.rc101-stopped .rc101-spray { opacity: .08 }`).

---

6. Common visual elements

These show up across multiple widgets — reuse the exact geometry.

6.1 Inlet/outlet pipe

```
<rect x=".." y=".." width="20" height="30" rx="3"
      fill="#C0CCDF" stroke="#98A8C0" stroke-width="1"/>
<line x1=".." x2=".." y1=".." y2=".."
      stroke="#3D52D5" stroke-width="1.5"
      stroke-dasharray="5,3" opacity=".4"
      class="<prefix>-flow-line"/>
```

6.2 Valve diamond (action symbol)

```
<g transform="translate(cx,cy)">
  <polygon points="0,-9 9,0 0,9 -9,0"
           fill="#FFFFFF" stroke="#3D52D5" stroke-width="1.8"/>
  <polygon points="0,-4 4,0 0,4 -4,0"
           fill="#3D52D5" opacity=".25"/>
</g>
```

6.3 Motor block (always 46×34)

```
<rect x=".." y=".." width="46" height="34" rx="6"
      fill="url(#<prefix>-motor)" stroke="#166030" stroke-width="1.5"
      filter="url(#<prefix>-gs)"/>
<text x="cx" y="cy-6" font-family="monospace" font-size="13"
      fill="white" text-anchor="middle" font-weight="700">M</text>
<text x="cx" y="cy+6" font-family="monospace" font-size="7"
      fill="#80E8A8" text-anchor="middle">45 RPM</text>
<rect x=".." y=".." width="6" height="12" fill="#1A7840"/>
```

6.4 Steam / hot jacket strip

```
<rect x=".." y=".." width="13" height="200" rx="3"
      fill="url(#<prefix>-jk)" stroke="#C89828" stroke-width="1"/>
<!-- ribs every 18 viewBox units -->
<line x1=".." x2=".." y1="N" y2="N"
      stroke="#B07018" stroke-width=".8" opacity=".6"/>
<!-- bolts at top and bottom -->
<circle cx=".." cy=".." r="4"
        fill="#D4A030" stroke="#A07820" stroke-width="1"/>
```

6.5 Status dot (mandatory)

```
<circle id="<prefix>-status-dot" cx=".." cy=".." r="7"
        fill="#1A7A40" stroke="#FFFFFF" stroke-width="1.5">
  <animate attributeName="opacity"
           values="1;.6;1" dur="2s" repeatCount="indefinite"/>
</circle>
```

6.6 Drop-shadow filter — **DEPRECATED** (2026-05-06)

Do not add `<filter>` / `feDropShadow` to equipment widget SVGs or mockups. Filters render inconsistently across Chrome file://, Inkscape, and Electron — silent rendering bugs (elements disappearing) waste hours of debugging. The flat colour + stroke system already gives enough depth.

User-configurable shadows on shape widgets (rect-shape, ellipse-shape, etc.) via the `enableShadow` prop are unaffected — that's a runtime feature, not a static decoration.

---

7. Typography

All text inside widgets uses **`font-family="monospace"`** to read like an instrument panel, not chrome UI text.

Use	Size	Weight	Color
Equipment letter ("M", "P", "C")	13	700	#FFFFFF (on motor) / #4A6090 (on light bg)
Equipment readout (RPM, %)	7	400	#80E8A8 (motor) / #3D52D5 (digital)
Tag label (when present)	8 – 9	500 – 600	#4A6090

Editor / sidebar labels are NOT inside the widget — they come from the surrounding layout. Don't render the equipment NAME inside the SVG; that's what the screen-level label widget is for.

---

8. Layering order (back to front)

1. Heating / hot-zone backgrounds (subtle, behind everything)
2. Vessel body + main shell
3. Process media (liquid, bed, powder)
4. Heat waves, bubbles, particles (animated overlays inside the vessel)
5. Internal mechanism (impeller, drum, turntable)
6. Foreground vessel ellipse / lid (covers the top of the mechanism)
7. External fittings (jackets, ducts, supports, manholes)

8. Pipes / inlets / outlets / valves
9. Motor block (always on top of vessel)
10. Status dot (always topmost — never obscured by anything)

---

9. State semantics

The widget's `class` on the root `<g>` switches via the inline script:

```
function _setState(state) {
  g.className.baseVal = '<prefix>-' + state;
  if      (state === 'running') dot.setAttribute('fill', '#1A7A40');
  else if (state === 'fault')   dot.setAttribute('fill', '#BE2E18');
  else                          dot.setAttribute('fill', '#8898B8');
}
```

Tag bindings: - `data-tag-run` (boolean) — true → `running`, false → `stopped` - `data-tag-fault` (boolean) — true → `fault` (overrides run) - `data-state` (string) — author override / preview, observed via `MutationObserver`

`fault` always wins over `run` regardless of order; `stopped` is the default for `run = false` AND `fault = false`.

---

10. Do / Don't

☑ Do	✗ Don't
Use `#3D52D5` for action / flow / interactive accents	Use it as static decoration
Use the green motor (`#28A055 → #1A7840`) for every drive	Invent a new motor color per machine
Halt all animations in `.<prefix>-stopped`	Leave a flow line shifting after stop
Prefix every `id` and `class` with the equipment tag	Use generic IDs like `#shaft`, `#motor`
Layer the motor + status dot last (foreground)	Let the impeller paint over the motor
Use monospace for in-widget numerics	Use Inter/Manrope inside a widget
Match the viewBox aspect to the equipment silhouette	Pad with empty space "for safety"
Bolt count = realistic for the size (not 30 of them)	Render every screw — visual noise

---

11. Checklist for a new widget

Before commit, verify:

- ☐ viewBox aspect matches equipment silhouette
- ☐ Root `<g>` has `class="<prefix>-running"`
- ☐ `data-tag-run`, `data-tag-fault`, `data-state` attrs present on `<svg>`
- ☐ Status dot present, r=7, white border
- ☐ Inline `<script>` exposes `init / putValue / update`
- ☐ `MutationObserver` watches `data-state`
- ☐ `<defs>` has NO `<filter>` / `feDropShadow` (deprecated, see §6.6)
- ☐ Motor (if any) uses `#28A055 → #1A7840` gradient + RPM text
- ☐ All animations gated by `.<prefix>-running` and frozen by `.<prefix>-stopped`
- ☐ Fault state blinks the status dot
- ☐ Every `id` and gradient name is prefixed
- ☐ No `#3D52D5` used as decoration
- ☐ Vessel main outline is `#98A8C0` at 1.5px
- ☐ Vessel body uses the 4-stop blue-grey gradient
- ☐ Hot zones use the yellow-amber family, never blue
- ☐ All text uses `monospace`
- ☐ Layering order respected (status dot topmost)

---

12. Equipment rotation pattern

Equipment widgets must support 0° / 90° / 180° / -90° rotation via a `rotation` prop. The rotation contract:

1. **Author the silhouette around a single rotation centre.** Pick a visual "anchor" point that makes sense for the equipment (volute centre for a pump, coupling axis for a motor, drum centre for a coater). Every rotation pivots around that point.

2. **State indicator and glyphs sit at the rotation centre.** This is non-obvious but powerful — a circle (or any shape) at the rotation centre is invariant under rotation. Padlock / "COMM LOSS" text / any glyph placed at the centre stays readable horizontally without any counter-rotation, because rotating around its own centre by any multiple of 90° leaves it visually unchanged (for shapes with 90° symmetry — a circle qualifies).

3. **Rotate only `.<prefix>-body`, not the indicator group.** This keeps the indicator's position stable and avoids counter-rotation gymnastics.

4. **Footer (label, speed readout, anything operator reads as text) lives in HTML outside the SVG.** It never rotates. Always reads horizontally regardless of equipment orientation.

CSS template (replace `<prefix>` with the widget's prefix, e.g. ep):

```
.<prefix>-body {
  transition: transform .25s ease;
  transform-origin: <cx>px <cy>px;
}
.<prefix>-root[data-rotation="90"]  .<prefix>-body { transform: rotate(90deg);
 }
.<prefix>-root[data-rotation="180"] .<prefix>-body { transform: rotate(180de
g); }
.<prefix>-root[data-rotation="-90"] .<prefix>-body { transform: rotate(-90de
g); }
```

JS contract: - The widget's `init()` calls `_applyRotation()` which reads `root.dataset.rotation` (the engine prop, propagated from `widget.props.rotation`) and writes a normalised value (`''`, `'90'`, `'180'`, `'-90'`) onto `<prefix>-root` so the CSS matches. - The MutationObserver re-runs `_applyRotation()` when `data-rotation` changes (live edit from the panel).

Schema entry:

```
rotation: { section: 'appearance', label: 'Rotation', type: 'select',
            options: ['0','90','180','-90'], default: '0', suffix: '°' }
```

13. Reference files

- `r101-mezclador.svg` — vertical reactor with jacket, impeller, wave, bubbles
- `gr101-granulador.svg` — fluid-bed granulator with air jets, distribution plate, particles
- `rc101-recubridor.svg` — drum coater with rotation, spray arm, hot-air duct
- `sec101-secador.svg` — horizontal fluid-bed dryer with heater, particle bed, exhaust filter
- `cp101-compresor.svg` — tablet press with rotating turntable, hopper, tablet animation

When unsure, **read the reference SVGs first**. The values above are extracted from them; matching the reference is the best way to keep new widgets consistent.

Data Links — binding a nivel de celda

Estado: DRAFT · pendiente de revisión **Fecha:** 2026-05-17 **Referencia visual:** `~/Downloads/INDUXA-data-links.html` (mockup del usuario)

---

1 · Concepto

Data Link = drill-down navigation con contexto automático.

Click (o hover→menu→click) sobre un elemento de un widget → navega a otra pantalla / popup, **inyectando automáticamente** el contexto del elemento clickeado (nombre, valor actual, tag fuente, timestamp).

Ejemplo de UX: - Bar Chart de “Producción por línea”: click en la barra de Línea 3 → abre “Dashboard Línea 3” con `{ sourceName: 'Línea 3', sourceValue: 1480, sourceTag: 'L3_PROD', sourceTimestamp: <now> }` disponible vía `Induxa.dl.context()`. - Status Table fila Reactor R-01 → “Detalle de equipo” con el reactor pre-cargado. - Geo Map marker Luque → “Dashboard planta” filtrado por Luque.

---

2 · NO es

- ❌ Bindings (tag→property reactivo)
- ❌ Tag aliases (tag→tag mirror)
- ❌ Connection sources (Modbus / MQTT / OPC)

Es **navegación**, no flujo de datos.

---

3 · Schema en `.syk`

3.1 · Por widget

```
{
  "widgets": [{
    "id": "bar1",
    "type": "bar-chart",
    "dataLinks": [
      {
        "id": "dl_1718...",
        "label": "Dashboard Línea",
        "icon": "arrow-right",
        "target": "LINEA_DETAIL",
        "targetType": "screen",
        "openMode": "same",
        "context": {
          "name":      true,
          "value":     true,
          "tag":       false,
          "timestamp": false,
          "widgetId":  false
        }
      },
      {
        "id": "dl_1718...",
        "label": "Trend histórico",
        "icon": "trending-up",
        "target": "TREND_VIEW_POPUP",
        "targetType": "popup",
        "openMode": "popup",
        "context": { "tag": true, "timestamp": true }
      },
```

```
      {
        "id": "dl_1718...",
        "label": "Alarmas activas",
        "icon": "alert-triangle",
        "target": "alarms",
        "targetType": "builtin",
        "openMode": "same",
        "context": { "name": true },
        "extraParams": { "filterField": "line", "filterValue": "{name}" }
      }
    ]
  }]
}
```

3.2 · Campos

Campo	Tipo	Default	Notas
`id`	string	auto-gen `dl_<ms>_<rand>`	Único por widget
`label`	string	required	Texto en el menú (≤ 40 chars)
`icon`	string|null	null	Lucide icon name
`target`	string	required	Screen name / Popup ID / built-in view key
`targetType`	`screen` | `popup` | `builtin`	`screen`	`builtin` = alarms / history / reports / trends
`openMode`	`same` | `popup` | `new-window`	`same`	new-window solo soportado en Electron / desktop
`context`	`{name,value,tag,timestamp,widgetId: bool}`	`{name:true,value:true}`	Standard vars pre-pobladas por engine
`extraParams`	object	`{}`	Params custom — admite `{name}` `{value}` `{tag}` `{timestamp}` como placeholders

---

4 · Contract de eventos

Cada widget Analytics dispara un evento custom cuando el usuario clickea (o hace hover en su elemento) un “elemento” interno.

4.1 · Evento `INDUXA:element-click`

```
widget.dispatchEvent(new CustomEvent('INDUXA:element-click', {
  bubbles: true,
  detail: {
    widgetId:    'bar1',
    elementId:   'series-0-bar-2',   // único dentro del widget
    name:        'Línea 3',          // nombre humano del elemento
    value:       1480,               // valor numérico actual
    tag:         'L3_PROD',          // tag fuente (si aplica)
    timestamp:   1747... ,           // ms · timestamp del valor
    raw:         { ... }             // payload completo del widget (uso inte
rno opcional)
  }
}));
```

4.2 · Evento `INDUXA:element-hover` (opcional · para el menu tooltip)

Mismo `detail`. Disparado cuando el cursor entra en un elemento (con debounce 100 ms). Lo usa el engine para mostrar el menú flotante si el widget tiene ≥ 2 data links.

4.3 · Quién dispara qué

Widget	Elemento	Cuándo
`bar-chart`	cada barra	click + hover
`status-table` (nuevo)	cada row	click + hover
`geo-map`	cada marker	click + hover
`trend-chart`	trace al click derecho (no left, choca con zoom)	rightclick
`pie-chart` (si existe)	cada slice	click + hover
`numeric-display`, `gauge`, `kpi-tile`, `label-widget`, `text-display`	el widget entero	click
Buttons / toggles / sliders / inputs	NO disparan element-click (tienen su propio handler)	—

---

5 · Runtime engine — `app/js/data-link-engine.js`

5.1 · API pública (en `window.InduxaDataLinks`)

```
window.InduxaDataLinks = {
  /** Init at viewer load. Subscribes to events globally. */
  init(): void,

  /** Programmatic trigger — useful from scripts. */
  follow(widgetId: string, dataLinkId: string, context?: object): void,

  /** Inspect what was passed to the current screen. Returns the
```

```
      flattened context object from the last navigation. Null on root. */
  context(): { name?, value?, tag?, timestamp?, widgetId?, ... } | null,
};
```

También adjuntar a `Induxa.dl.*` para uso desde scripts:

```
Induxa.dl = {
  context: () => InduxaDataLinks.context(),
  follow:  (widgetId, dlId, ctx) => InduxaDataLinks.follow(widgetId, dlId, ct
x),
};
```

5.2 · Flujo

```
USER click on bar
   ↓
bar-chart dispatches  INDUXA:element-click  { name:'L3', value:1480, tag:'L3_
PROD', … }
   ↓
data-link-engine catches it  →  reads widget.dataLinks
   ↓
   case dataLinks.length === 0  →  no-op (or hover hint "no data links")
   case dataLinks.length === 1  →  follow it directly (no menu)
   case dataLinks.length  >= 2  →  show floating menu at cursor pos
                                     user picks one  →  follow it
   ↓
buildContext(link.context, evt.detail)  →  { name, value, tag, timestamp, … }
   ↓
templatize link.extraParams replacing {name}/{value}/etc.
   ↓
SWITCH on link.openMode:
  'same'        →  Induxa.navigate(target, params)
  'popup'       →  Induxa.openPopup(target, params)
  'new-window'  →  window.open(buildUrl(target, params))   // Electron only
```

5.3 · Floating menu

DOM injected once on `init()`:

```
<div class="idl-menu" id="idlMenu" hidden>
  <div class="idl-menu-title">DATA LINKS</div>
  <button class="idl-menu-item">…</button>
  …
</div>
```

Posicionado en `getBoundingClientRect()` del elemento clickeado. Cierra con: click outside / Esc / click en un item.

Tema **light hardcoded** (consistencia con resto del trabajo).

---

6 · Standard context variables

Pre-pobladas por el engine antes de navegar. La pantalla destino las lee con `Induxa.dl.context()`.

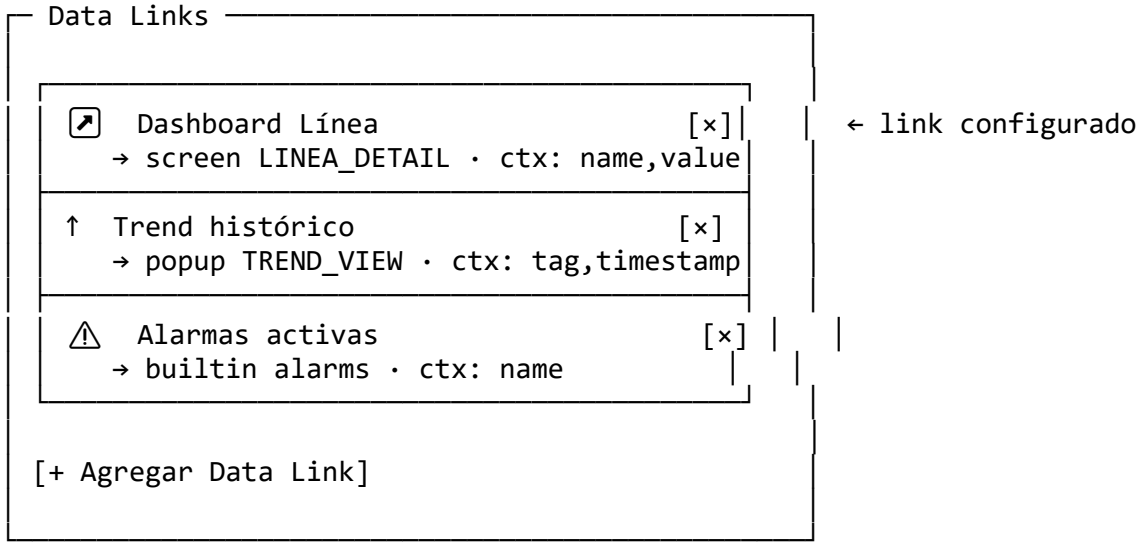
Variable	Tipo	Origen
`name`	string	`evt.detail.name`
`value`	any	`evt.detail.value`
`tag`	string|null	`evt.detail.tag`
`timestamp`	number|null	`evt.detail.timestamp`
`widgetId`	string	`evt.detail.widgetId`
`sourceScreen`	string	nombre de la pantalla origen
`sourceWidgetType`	string	`widget.type`
`clickedAt`	number	`Date.now()` al momento del click

Cualquier campo extra en `evt.detail` también se propaga (passthrough).

---

7 · Properties Panel — sección "Data Links"

Visible para CUALQUIER widget bindable. Vive como tab/sección en el widget-config-modal (junto a Bindings, Events, etc.).



Click en una row → expand inline edit. Campos:

- **Etiqueta** (`label`) · text input
- **Pantalla destino** (`target` + `targetType`) · select con grupos: Screens / Popups / Built-in views (alarms, history, reports, trends, events)

- **Pasar contexto** (`context.*`) · checkboxes para name/value/tag/timestamp/widgetId
- **Parámetros extra** (`extraParams`) · key-value table (opcional, plegable)
- **Abrir en** (`openMode`) · radio buttons Same / Popup / New window
- **Icono** (`icon`) · icon picker reutilizando el del Editor

Drag-handle al lado de cada row para reordenar.

---

8 · Integración con widgets existentes

8.1 · Widgets multi-element

Widget	Cambios necesarios
`bar-chart`	Cada bar `<rect>` agrega `data-element-id` y `data-element-name`. IIFE adentro hace `bar.addEventListener('click', …)` que dispatcha `INDUXA:element-click`. ~30 líneas.
`status-table` (NUEVO)	Widget tabla nuevo (no existe hoy según inventario). Cada `<tr>` dispatch. Si no hay tiempo, fallback: en `numeric-display` con `mode: 'table'`.
`geo-map` (si existe)	Verificar existencia. Si no, fuera de scope MVP.
`trend-chart`	Right-click sobre trace point → dispatch. Skip si complejo.

8.2 · Widgets single-element

Widgets que el usuario suele clickear como bloque entero (no tienen sub-elements):

- `numeric-display`, `label-widget`, `text-display`, `gauge`, `bar-progress`, `led`

Para estos, el "elemento" es el widget entero. El engine envuelve cualquier widget con `dataLinks[].length > 0` con un click handler que dispatcha `INDUXA:element-click` usando los props del propio widget (`name = widget.label`, `value = widget.value` from tag binding, `tag = widget.props.tagRead`).

8.3 · Widgets que NO deben tener Data Links

- Buttons / toggles / sliders / inputs / numeric-entry / dropdown / radio-group / checkbox / segmented / datetime-picker / spinner / text-input / button-multistate / button-momentary / button-latching

Estos tienen su propio click handler (escribir tag). Si el ingeniero quiere drill-down desde un button, ya tiene `clickAction: open_view` documentado.

---

9 · Phasing

Fase	Scope	Tiempo
F1	Schema + persistencia · auto-migrate `.syk` (idempotente, vacío) · Properties Panel section (CRUD UI)	1 día
F2	`data-link-engine.js` runtime: subscribe events, menu UI, navigate, context build	1 día
F3	`bar-chart` dispatcha `INDUXA:element-click` por barra. Demo end-to-end con 1 widget multi-element.	0.5 día
F4	Single-element widgets (numeric-display, gauge, kpi-tile, label, text-display) wrapping automático	0.5 día
F5	`geo-map` (si existe) + `status-table` nuevo widget (si entra en scope)	1 día
F6	Pantalla destino — recibe contexto. `Induxa.dl.context()` API. Demo: screen de "Detalle equipo" que lee el contexto	0.5 día
F7	Demo `.syk` + docs `docs/DATA-LINKS.md` (user-facing, no spec)	0.5 día
Total		**5 días**

---

10 · Open questions

1. **¿Built-in views como targets?** El mockup muestra "Alarmas activas" como link. Conviene mapear `targetType:'builtin'` a las views internas (`alarms`, `history`, `trends`, `reports`, `events`)? ¿O solo screens del proyecto?
2. **¿Filtro automático en built-in views?** Si Data Link va a `alarms` con `context.name='Línea 1'` → la vista de alarmas debe filtrar por línea. ¿Cómo se interpreta? ¿Parámetro `?filter=line:Línea+1` en la URL? Necesita patrón estándar.

3. **¿`status-table` como widget nuevo?** El mockup lo muestra pero no existe en el inventario (`/MEMORY` lista 21 widgets standard, no incluye status-table). ¿Lo creamos como parte de Data Links, o lo dejamos para una iteración separada?
4. **¿`geo-map` existe?** No aparece en el inventario de MEMORY. ¿Es widget existente o también nuevo?
5. **¿`new-window` mode?** Solo aplica en Electron (gateway con `window.open` que abre child window). En web standalone caería a `same`. ¿Lo soportamos o lo escondemos en UI?
6. **¿Permite multi-link en single-element widgets?** Un numeric-display con 3 data links — ¿menú flotante? ¿O solo permitir 1 link en single-elements y forzar menú a multi-element?
7. **¿Templating en extraParams?** `extraParams.filterValue = '{name}'` — ¿soportamos cualquier var del context como placeholder? ¿Solo las standard?

---

11 · Próximos pasos

1. Usuario revisa este spec y responde las open questions
2. Decisión sobre alcance MVP vs. completo
3. Arranco F1 con luz verde

Trigger phrases para retomar: - *"arrancar data links"* / *"fase 1 data links"* → ejecuta F1 - *"continuar data links"* → primera fase pendiente - *"status data links"* → reporta fases done/pending

Catálogo extendido y notas de implementación

> How widgets are loaded, manifested and bound. Where to put a custom one. Source: `app/js/ui-widget-engine.js`, `app/assets/ui-widgets/`.

Catalogue

The bundled widgets live under `app/assets/ui-widgets/`. Each widget is either a single HTML file or an HTML + manifest pair.

Category	Widget	File
UI Input	button-momentary	`button-momentary.html`
UI Input	button-latching	`button-latching.html`
UI Input	button-multistate	`button-multistate.html` (+ manifest)
UI Input	toggle-switch	`toggle-switch.html`
UI Input	checkbox	`checkbox.html`
UI Input	radio-group	`radio-group.html` (+ manifest)
UI Input	segmented	`segmented.html` (+ manifest)
UI Input	slider	`slider.html`
UI Input	spinner	`spinner.html`

Category	Widget	File
UI Input	numeric-entry	`numeric-entry.html`
UI Input	text-input	`text-input.html`
UI Input	dropdown	`dropdown.html` (+ manifest)
UI Input	datetime-picker	`datetime-picker.html`
Display	numeric-display	`numeric-display.html`
Display	text-display	`text-display.html`
Display	label-widget	`label-widget.html`
Display	gauge	`gauge.html`
Display	bar-progress	`bar-progress.html`
Shapes	rect-shape	`rect-shape.html`
Shapes	ellipse-shape	`ellipse-shape.html`
Shapes	line-shape	`line-shape.html`

Plus four canvas-rendered widgets that live inside the project itself (not as HTML files): **trend**, **led**, **bar**, **rect**.

Widget loader

`UI_WIDGET_CACHE` in `ui-widget-engine.js` lazy-loads each `<type>.html` the first time it is dropped on the canvas. The HTML body is grafted into a wrapper `<div>` plus a per-instance script context.

If a `<type>-manifest.json` exists, it declares:

- The exposed properties (label, type, default value, group).
- Which properties accept SEL expressions vs static values.
- Which properties are tag bindings.

The Properties Panel in the Editor reads the manifest to render the right UI control per property.

Anatomy of a widget HTML

```
<!-- button-momentary.html (excerpt) -->
<style scoped>
  .btn-mom { /* ... */ }
</style>

<div class="btn-mom"
     data-bind-bgcolor="{props.color}"
     data-bind-label="{props.label}"
     data-on-mousedown="onPress"
     data-on-mouseup="onRelease">
  <span data-bind-text="{props.label}"></span>
</div>
```

```
<script>
  function onPress(ctx) {
    if (ctx.bindings.tagWrite) Induxa.tag.write(ctx.bindings.tagWrite, true);
  }
  function onRelease(ctx) {
    if (ctx.bindings.tagWrite) Induxa.tag.write(ctx.bindings.tagWrite, false);
  }
</script>
```

Notes:

- **`data-bind-*`** attributes are reactive: they are rebound by `BindingManager` on every tag update.
- **`data-on-*`** wires DOM events to functions defined in the widget `<script>` block (or in user scripts loaded by `hmi-script-engine.js`).
- The `ctx` argument exposes `{ props, bindings, tags, screen, popup }` — see `app/js/hmi-script-engine.js` for the full surface.

Manifest — exposed properties

```
// button-multistate-manifest.json
{
  "displayName": "Multi-state Button",
  "category":    "input",
  "icon":        "toggle-left",
  "defaultSize": { "width": 120, "height": 40 },
  "properties": [
    { "name": "label",       "type": "string", "default": "Press" },
    { "name": "color",       "type": "color",  "default": "#0969DA" },
    { "name": "states",      "type": "states-table", "min": 2, "max": 8 }
  ],
  "bindings": [
    { "name": "tagRead",  "label": "Read tag",  "required": true },
    { "name": "tagWrite", "label": "Write tag", "required": false }
  ]
}
```

The Editor Properties Panel reads this and renders:

- A text input for `label`.
- A SynkroColorPicker for `color`.
- A custom multi-state grid for `states`.
- Two TomSelect tag pickers for `tagRead` and `tagWrite`.

Binding modes (per property, per binding)

Mode	Stored as	Use when
`static`	Literal value	Static label, fixed colour
`expr`	SEL expression	“Red when temp>50, otherwise green”

Mode	Stored as	Use when
`state`	Tag → result map (state machine)	"Pump 0 → STOPPED, 1 → RUN, 2 → FAULT"

The Properties Panel shows a small icon button next to each property that cycles through the available modes.

Adding a new widget

1. Create `app/assets/ui-widgets/<type>.html`.
2. (Optional) `app/assets/ui-widgets/<type>-manifest.json`.
3. Reference any new property in your HTML via `data-bind-*` or `props.*`.
4. Add the widget to the palette: edit `widgetPaletteToggle` config in `app/views/editor.html` to slot it under the right category.
5. Reload the editor (Ctrl+Shift+R).

A widget without a manifest still works — it just falls back to a "raw config" JSON editor in the Properties Panel.

Lifecycle hooks

Defined in the widget's `<script>` block, called by `hmi-script-engine.js`:

Hook	Called when
`onMount(ctx)`	Widget instance inserted into the DOM
`onUnmount(ctx)`	Removed (screen change, delete)
`onUpdate(ctx, changedKeys)`	Any binding produced a new value
`onPress / onRelease / onClick / onChange`	DOM events bound by `data-on-*`

`ctx` is stable for the lifetime of the widget instance — store per-instance state on it.

Symbols / templates

`app/js/symbol-engine.js` lets the engineer wrap a group of widgets + relative bindings into a reusable **symbol** (template). Symbols can be nested. Their tag references are rebased when an instance is dropped, so a "Pump symbol" with `{tagRunning}` → `{V101_RUNNING}` for one instance and `{V102_RUNNING}` for another.

This is the closest INDUXA currently has to UDT instances on the screen side. (UDTs themselves are tag-side templating — see the UDT engine.)

Performance

- `BindingManager` keeps an O(1) reverse index `tag → widgets`. A tag update only re-renders the widgets that depend on it.

- Each `data-bind-*` is one SEL compile + one cached function.
- Widgets that need every tick (clocks, timers) should subscribe to a single `Induxa.tick` instead of polling.
- Canvas-rendered widgets (trend especially) own their own redraw scheduler — never call into them from a per-tag-update path.

Pantallas y ciclo de vida

Una **pantalla** (*screen*) es un lienzo del Viewer que agrupa widgets bajo un nombre, un tipo y un conjunto de scripts de ciclo de vida. El proyecto define todas sus pantallas; el Viewer carga la pantalla activa y mantiene una caché de la anterior para transiciones rápidas.

Tipos de pantalla

Tipo	Aparece en pestañas	Caso típico
`main`	sí	vistas de planta, dashboards
`popup`	no — se abre desde un botón o un script	edición de set-points, ayudas
`docked`	no — se ancla a un borde del layout	banner de alarmas, navegación

Pantallas marcadas como `disabled` quedan ocultas para el operador incluso si su tipo es `main`. Es la forma idiomática de tener trabajo en curso en el proyecto sin exponerlo en producción.

Hooks de ciclo de vida

Cada pantalla puede declarar hasta cuatro scripts:

Hook	Cuándo se ejecuta	Esperado bloquear navegación
`onOpen`	al cargar la pantalla, antes de mostrarla al operador	sí (se hace `await`)
`onClose`	al salir hacia otra pantalla, antes de descargarla	sí (se hace `await`)
`onRefresh`	cada `refresh_interval` ms mientras la pantalla esté visible	no
`onIdle`	una vez tras `idle_timeout` ms sin entrada de usuario	no

`refresh_interval` y `idle_timeout` se configuran por pantalla en milisegundos (mínimo 1000 ms). Si están a 0, los timers correspondientes no se arman aunque el script esté presente.

> onRefresh y onIdle se **pausan automáticamente** cuando la pestaña del navegador no está visible (document.hidden). Esto evita ráfagas de actividad cuando el operador minimiza la ventana y la pantalla recupera el ritmo al volver al foco.

Datos por pantalla

Dentro de los scripts de pantalla, Induxa.screen.{setData, getData} ofrecen un almacén Map por instancia de pantalla:

```
// onOpen
Induxa.screen.setData('startTs', Date.now());

// onIdle / onClose
const dur = Date.now() - Induxa.screen.getData('startTs');
Induxa.log(`Pantalla ${Induxa.screen.name} estuvo abierta ${dur} ms`);
```

El almacén se **borra automáticamente** al ejecutar onClose. No es persistente entre sesiones; para eso usa tags internal.

Navegación

```
Induxa.navigate('Reactor');                      // pantalla principal
Induxa.openPopup('SetPointEdit', { tag: 'TEMP_SP' });   // popup parametrizad
o
```

Cuando un popup se abre con parámetros, el script de la pantalla popup los recibe vía Induxa.popup.params():

```
// onOpen del popup SetPointEdit
const tag = Induxa.popup.params().tag;
const cur = Induxa.tag.read(tag)?.value ?? 0;
document.getElementById('sp-input').value = cur;
```

El popup se cierra a sí mismo con Induxa.popup.close().

Casos de uso

UC-18 — Dashboard que se refresca cada 5 segundos

```
// onRefresh, refresh_interval = 5000
const oee_a = Induxa.tag.read('LINE_A_OEE')?.value || 0;
const oee_b = Induxa.tag.read('LINE_B_OEE')?.value || 0;
Induxa.tag.write('TOTAL_OEE', (oee_a + oee_b) / 2);
```

UC-19 — Pantalla con auto-cierre

```
// onIdle, idle_timeout = 300000  (5 min)
Induxa.notify('Cerrando por inactividad', 'warn');
Induxa.navigate('Main');
```

UC-20 — Popup parametrizado para editar set-points

Botón en pantalla principal:

```
// onClick
Induxa.openPopup('SetPointEdit', { tag: 'TEMP_SP', min: 0, max: 200 });
```

Pantalla popup `SetPointEdit`, script onOpen:

```
const p = Induxa.popup.params();
Induxa.screen.setData('tag', p.tag);
Induxa.screen.setData('min', p.min);
Induxa.screen.setData('max', p.max);
const cur = Induxa.tag.read(p.tag)?.value ?? 0;
// ... rellenar widgets del popup con cur, p.min, p.max
```

Botón *Save* del popup:

```
const v   = parseFloat(document.getElementById('sp-input').value);
const tag = Induxa.screen.getData('tag');
const min = Induxa.screen.getData('min');
const max = Induxa.screen.getData('max');

if (!Number.isFinite(v) || v < min || v > max) {
  Induxa.notify(`Valor fuera de rango [${min}, ${max}]`, 'error');
  return;
}
const r = await Induxa.tag.writeWithContext(tag, v, {
  widgetType: 'numeric-entry', confirmed: true,
});
if (r.success) {
  Induxa.notify('Set-point actualizado', 'success');
  Induxa.popup.close();
} else {
  Induxa.notify(r.error, 'error');
}
```

UC-21 — Pre-cargar datos al entrar

```
// onOpen — cargar histórico de receta y dejarlo accesible a la pantalla
const last = Induxa.tag.read('LAST_RECIPE_NAME')?.value;
Induxa.screen.setData('lastRecipe', last);
if (last) Induxa.notify('Última receta: ' + last, 'info');
```

Material técnico de referencia

Popups parametrizables

How operator-facing popups receive parameters from the calling widget, and how `{popup.x}` substitution inside template fields is resolved. Source: `app/js/popup-engine.js`.

Why parameterised popups

Without parameters, a designer who wants ten reactor detail popups has to draw ten almost-identical screens. With one parameterised template plus ten button bindings, they draw the popup once and the runtime substitutes per-button.

Typical flow:

```
[Reactor V101 button] → opens popup "Reactor Detail" with { id: "V101" }
[Reactor V102 button] → opens popup "Reactor Detail" with { id: "V102" }
                            │
                            ▼
   Inside the popup template:
       {popup.id}         → 'V101' / 'V102' (string substitution)
       {SM_REACTOR_{popup.id}_STATE}  → tag reference resolved at open time
```

Parameter substitution types

The popup engine recognises **three** substitution types — each behaves differently. This is the part that bites engineers most often.

1. `identifier` (default)

```
target field:    SM_REACTOR_{popup.id}_STATE
runtime value:   SM_REACTOR_V101_STATE
```

The token is replaced **before** the field is parsed. The result becomes a normal tag name. Use this when you want `popup.id` to participate in **building a tag name**.

Common example: `tagRead` binding on a label inside the popup.

2. `string`

```
target field:    Reactor {popup.id} Detail
runtime value:   Reactor V101 Detail
```

Plain text replacement, used in caption / title / message fields. The string keeps quotes if any.

3. `tagref`

```
target field:    {popup.tagRead}                    (mode: tagref)
runtime value:   the actual tag name string (not its value)
```

Use this when the **caller passes a tag name** that the popup should read, and the popup widget expects a tag reference. The substitution preserves the binding semantics — the tag is still resolved live by the binding engine.

Authoring a popup template

1. Create a screen marked `is_popup: true` (Screen properties → Popup).
2. Declare the parameters it expects (Screen → Popup parameters):

```
[
  { "name": "id",       "type": "identifier", "required": true },
  { "name": "title",    "type": "string",     "default": "Reactor" },
  { "name": "tagSp",    "type": "tagref",     "required": false }
]
```

3. Use {popup.<name>} anywhere a string field accepts it (titles, labels, tag bindings, formulas).

Calling a popup

From a widget script:

```
Induxa.openPopup('reactor-detail', {
  id:    'V101',
  title: 'Reactor V101',
  tagSp: 'V101_SETPOINT'
});
```

Or via a button binding (no script needed):

```
{
  "type": "button",
  "bindings": {
    "onClick": {
      "mode": "popup",
      "popup_id": "reactor-detail",
      "params": { "id": "V101" }
    }
  }
}
```

Substitution rules — the edge cases

- **Missing required param:** the popup refuses to open and a toast flags the caller widget.
- **Unknown param:** logged (warning), substituted as empty string.
- **Recursive substitution:** values containing {popup.x} are NOT re-expanded. One pass only.
- **Identifier with spaces or punctuation:** the engine slugifies it for use inside a tag name (same logic as state-machine slugification). This means {popup.id} of "Pump #2" becomes PUMP_2 in SM_PUMP_2_STATE.
- **Tagref pointing nowhere:** the popup opens, but the affected widget shows quality=BAD (the binding engine handles missing tags gracefully).

Implementation notes

- `app/js/popup-engine.js` is loaded by both the Editor (for design-time preview) and the Viewer (for runtime open).
- It hooks into `BindingManager.beforeRender` so substitution happens before SEL compilation — meaning compiled expressions are cached *post-substitution*, one per popup-instance.
- When a popup is closed and reopened with new params, the binding cache is invalidated for that popup-instance.

Quick sanity test

Open the popup with a known param, then in DevTools:

```
// returns the popup-instance's resolved param map
Induxa.popup.getParams('reactor-detail-instance-1');
```

UDTs — Tipos definidos por el usuario

Un **UDT** (User-Defined Type) es una plantilla reutilizable que agrupa varias propiedades relacionadas bajo un mismo concepto: una bomba, una válvula, un lazo PID, un motor, una zona de horno. A partir de la plantilla, instancias el UDT tantas veces como equipos físicos tengas, y INDUXA SCADA genera los tags correspondientes con nomenclatura consistente.

Por qué usar UDTs

Sin UDTs:

```
PUMP_01_RUN, PUMP_01_FAULT, PUMP_01_FLOW, PUMP_01_HOURS
PUMP_02_RUN, PUMP_02_FAULT, PUMP_02_FLOW, PUMP_02_HOURS
…
PUMP_12_RUN, PUMP_12_FAULT, PUMP_12_FLOW, PUMP_12_HOURS
```

48 tags creados a mano, 48 oportunidades de error, y cuando hay que añadir `MAINT_DUE` toca crear 12 más. Si el script `onValueChange` de `_FAULT` cambia, hay que repetirlo 12 veces.

Con UDT:

1. Defines `UDT_Pump` con las propiedades `Run`, `Fault`, `Flow`, `Hours`, sus tipos, scan rates, scripts y alarmas.
2. Creas 12 instancias `Pump_01`, `Pump_02`, …
3. INDUXA SCADA genera 48 tags `Pump_01/Run`, `Pump_01/Fault`, …, propagando alarmas y scripts.
4. Si añades `MaintDue` al UDT y regeneras, los 12 instancias reciben el nuevo tag.

Estructura

Una **definición** vive en `project.udt_definitions[]`:

```
{
  "id":   "udt_pump",
  "name": "UDT_Pump",
  "description": "Bomba centrífuga estándar",
  "properties": [
    { "name": "Run",   "data_type": "bool",    "access": "rw" },
    { "name": "Fault", "data_type": "bool",    "access": "ro",
      "alarm": { "priority": "High", "message": "{instance} fault" } },
    { "name": "Flow",  "data_type": "float32", "access": "ro", "unit": "m3/h",
      "scan_rate_ms": 1000 },
    { "name": "Hours", "data_type": "float32", "access": "ro", "unit": "h" }
  ],
  "protocol_template": {
    "protocol":      "modbus",
    "connection_id": "PLC_1",
    "base_address":  "40100"
  }
}
```

Una **instancia** vive en `project.udt_instances[]`:

```
{
  "id":            "udti_pump_01",
  "udt_id":        "udt_pump",
  "instance_name": "Pump_01",
  "group":         "Bombas",
  "io_mapping": [
    { "property": "Run",   "address": "00033" },
    { "property": "Fault", "address": "00034" },
    { "property": "Flow",  "address": "40110" },
    { "property": "Hours", "address": "40114" }
  ]
}
```

Los tags generados se llaman `Pump_01/Run`, `Pump_01/Fault`, … y llevan metadatos `_udt_instance`, `_udt_type`, `_udt_property`. La regeneración respeta los `id` existentes para preservar referencias históricas.

Scripts heredados

Si una propiedad declara `scripts: { onValueChange, onQualityChange }`, los tags generados de **todas las instancias** los heredan. Esto permite definir una vez la lógica común a todas las bombas:

```
{
  "name": "Fault",
  "data_type": "bool",
  "access": "ro",
  "scripts": {
    "onValueChange": "if (current.value === true && prev.value === false) { I
```

```
nduxa.tag.write(tagPath.replace('/Fault','/Run'), false); }"
  }
}
```

> **Override por instancia.** Si modificas el script de un tag generado vía la API REST, INDUXA SCADA marca esa modificación con `_udt_scripts_override = true` y la **respeta** en futuras regeneraciones. Para volver al comportamiento estándar del UDT, elimina el override y regenera.

Direcciones por instancia

Cada propiedad puede declarar un `address_offset` relativo a la `base_address` de la instancia. Sintaxis dependiente del protocolo:

Protocolo	Sintaxis	Ejemplo
Modbus	`+N` o `+N.bit`	`+6`, `+6.8`
OPC-UA	sufijo de `node_id`	`/Speed`, `.Acked`
MQTT	sufijo de tópico	`/speed`, `speed`

Así una instancia `Pump_03` con `base_address = 40130` y propiedad `Flow` con `address_offset = +10` resuelve a `40140` automáticamente. Si una instancia necesita una dirección distinta a la calculada, se sobrescribe en `io_mapping[].address`.

Caso de uso

UC-22 — 8 bombas con un único cambio

1. Defines `UDT_Pump` con 5 propiedades.
2. Creas 8 instancias `Pump_01` … `Pump_08`.
3. Negocio pide añadir `LastMaint` (`datetime`).
4. En el Editor, añades la propiedad al UDT y pulsas *Sync All*.
5. Las 8 instancias reciben `Pump_xx/LastMaint` automáticamente, con tag id estable y sin tocar pantallas (los widgets que bindean por nombre simplemente lo ven aparecer).

Material técnico de referencia

Especificación técnica de UDT

Status: Spec frozen — ready for implementation **Owner:** Editor / Tag domain **Date:** 2026-04-27
Scope: UDT data model only. Faceplates explicitly out of scope (Phase 3).

---

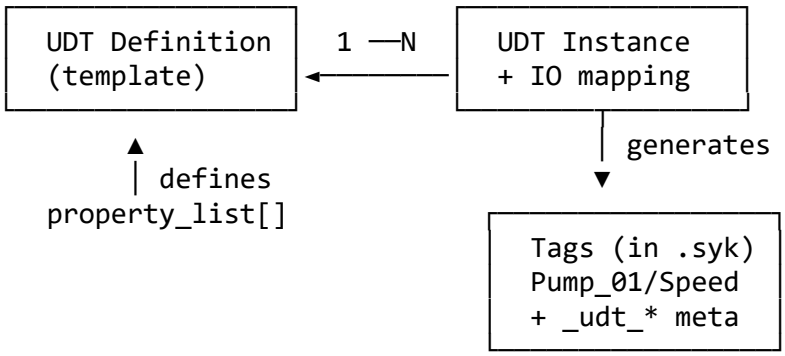
1. Goals

Bring object-oriented tag modeling to INDUXA: define a `Pump` once with its 6 properties, then instantiate it 50 times mapping each instance's IO to different Modbus/MQTT/OPC-UA endpoints. Generated tags behave like any normal tag (bindings, alarms, history, screens).

Non-goals for MVP: - Faceplates (visual templates) — separate spec, Phase 3 - UDT inheritance (parent type) — Phase 2 - Nested UDTs (UDT inside UDT) — Phase 2 - Per-instance property override beyond IO mapping — Phase 2 - Cross-project UDT library (`.sdt` files) — Phase 2

---

2. Architecture summary



- Both **definitions** and **instances** live inside the project `.syk` file. Portable, no global state.
- Generated tags carry `_udt_instance`, `_udt_type`, `_udt_property` metadata so the system knows where they came from.
- Editing a definition **auto-propagates** to all its instances (regenerate tags, additive + destructive).

---

3. Data model

3.1 `project.udt_definitions[]` (new array in `.syk`)

```
{
  "id":           "udt_pump_centrifugal",
  "name":         "Pump Centrifugal",
  "description":  "Standard centrifugal pump with VFD",
  "category":     "Rotating Equipment",
  "icon":         "fan",
  "version":      "1.0.0",
  "parent_id":    null,
  "properties": [
    {
      "name":         "RunFeedback",
      "description":  "Pump running confirmation",
      "data_type":    "bool",
      "access":       "read",
```

```
      "unit":          "",
      "default":       false,
      "scan_rate_ms":  1000,
      "deadband":      0,
      "logging":       true,
      "alarm":         null
    },
    {
      "name":          "Speed",
      "description":   "Rotational speed",
      "data_type":     "float32",
      "access":        "read",
      "unit":          "rpm",
      "default":       0,
      "scan_rate_ms":  500,
      "deadband":      0.5,
      "logging":       true,
      "alarm": {
        "priority": "High",
        "message":  "{instance} overspeed",
        "HH": 1800,
        "H":  1700,
        "L":  null,
        "LL": null
      }
    }
  ],
  "_created_at":  "2026-04-27T20:00:00Z",
  "_created_by":  "dherrera",
  "_updated_at":  "2026-04-27T20:00:00Z"
}
```

Field rules:

Field	Type	Required	Validation
`id`	string	yes	`udt_<snake_case>`, generated from name
`name`	string	yes	1-60 chars
`category`	string	no	free text, used for filter
`icon`	string	no	Lucide icon name, default box
`version`	semver	no	default `1.0.0`
`parent_id`	string|null	no	reserved Phase 2, must be `null` in MVP
`properties[]`	array	no	can be empty; min 1

Field	Type	Required	Validation
			to be useful
`properties[].name`	string	yes	matches `/^[A-Z][a-zA-Z0-9_]*$/` (CamelCase, no spaces)
`properties[].data_type`	enum	yes	`bool` | `int16` | `int32` | `uint16` | `uint32` | `float32` | `float64` | `string`
`properties[].access`	enum	yes	`read` | `write` | `readwrite`
`properties[].scan_rate_ms`	int	no	default 1000, min 100
`properties[].deadband`	number	no	default 0
`properties[].alarm`	object|null	no	only valid for numeric types if `HH`/`H`/`L`/`LL` present

3.2 `project.udt_instances[]` (new array in `.syk`)

```
{
  "id":            "udti_a8f4b2c1",
  "udt_id":        "udt_pump_centrifugal",
  "instance_name": "Pump_01",
  "description":   "Main feed pump skid A",
  "group":         "Plant1/SkidA",
  "io_mapping": [
    {
      "property":      "RunFeedback",
      "protocol":      "modbus",
      "connection_id": "conn_plc_main",
      "address":       "40001"
    },
    {
      "property":      "Speed",
      "protocol":      "opcua",
      "connection_id": "conn_kepserver",
      "address":       "ns=2;s=Pump1.Speed"
    }
  ],
  "_created_at":  "2026-04-27T20:05:00Z",
  "_created_by":  "dherrera",
  "_updated_at":  "2026-04-27T20:05:00Z"
}
```

Field rules:

Field	Type	Required	Validation
`id`	string	yes	auto `udti_<8-hex>`
`udt_id`	string	yes	must reference existing definition
`instance_name`	string	yes	matches `/^[A-Za-z][A-Za-z0-9_-]*$/`, unique within project
`group`	string	no	free text, used for tag tree organization
`io_mapping[]`	array	no	one entry per definition property
`io_mapping[].protocol`	enum	yes if mapped	`modbus` | `mqtt` | `opcua` | `internal`. Reserved for Phase 1.5: `gateway-sql`, `gateway-api`
`io_mapping[].connection_id`	string	yes if mapped	references `project.connections[].id` for that protocol
`io_mapping[].address`	string	yes if mapped	format depends on protocol

3.3 Generated tags (in `project.tags[]`)

Each instance × property = one tag. Existing tag schema, plus three metadata fields:

```
{
  "id":             "tag_xxxx",
  "name":           "Pump_01/RunFeedback",
  "description":    "Main feed pump skid A — Pump running confirmation",
  "data_type":      "bool",
  "protocol":       "modbus",
  "connection_id":  "conn_plc_main",
  "address":        "40001",
  "scan_rate_ms":   1000,
  "deadband":       0,
  "access":         "read",
  "group":          "Plant1/SkidA",
  "logging":        true,
  "_udt_instance":  "Pump_01",
  "_udt_type":      "udt_pump_centrifugal",
  "_udt_property":  "RunFeedback"
}
```

Tag naming: `<instance_name>/<property_name>` — fixed in MVP, not configurable. Slash gives free tree-view in Tag Manager.

Read-only flag in UI: any tag with `_udt_instance` set is shown as **read-only in Tag Manager edit forms** — only the parent UDT instance can modify it. Direct deletion blocked; user must delete the instance.

3.4 Schema migration

On project load, if either array is missing, initialize as empty:

```
project.udt_definitions = project.udt_definitions || [];
project.udt_instances   = project.udt_instances   || [];
```

No version bump needed — both arrays are additive.

---

4. Engine — `server/engines/udt-engine.js`

Singleton module, initialized in `server.js`:

```
udtEngine.init({
  getProject:  () => activeProject,
  saveProject: () => saveActiveProject(),
  tagEngine:   tagEngine,
  alarmEngine: alarmEngine,
  audit:       (user, msg) => logAudit(user, msg)
});
```

4.1 Public methods

Method	Description
`listDefinitions()`	returns `project.udt_definitions` with `instance_count` per def
`getDefinition(id)`	full definition object or `null`
`createDefinition(data, user)`	inserts, returns saved object
`updateDefinition(id, data, user)`	mutates, **triggers `propagateDefinitionChange()`**
`deleteDefinition(id, user)`	rejects if `instance_count > 0`
`listInstances()`	returns instances with `mapping_status` and `tag_count`
`getInstance(id)`	full instance with its tags
`createInstance(data, user)`	inserts, initializes empty IO mapping
`updateInstance(id, data, user)`	mutates, **regenerates tags if io_mapping changed**
`deleteInstance(id, user)`	removes instance + cascade-deletes its tags

Method	Description
`generateTags(instanceId, user)`	idempotent; returns `{ created: N, updated: M, removed: K }`
`propagateDefinitionChange(udtId)`	called after `updateDefinition`; runs `generateTags` for every instance referencing it
`validateIoMapping(instanceId)`	`{ total, mapped, unmapped, complete, pct }`

4.2 generateTags() algorithm

```
For each property in definition:
  desiredTagName = `${instance.instance_name}/${property.name}`
  mapping = instance.io_mapping.find(m => m.property === property.name)

  Build tag object using:
    - definition property defaults (data_type, scan_rate, etc.)
    - mapping override (protocol, connection_id, address)
    - instance.group as tag.group
    - alarm metadata if property.alarm

  If tag with that name exists in project.tags AND _udt_instance matches:
    → update in place (preserve id, value, quality, history)
  Else:
    → create new tag

For each existing tag with _udt_instance === instance.instance_name:
  If its _udt_property is NOT in current definition.properties:
    → delete (definition removed that property)

If property has alarm config:
  alarmEngine.upsertUdtAlarm(tag, property.alarm, instance.instance_name)
```

Idempotency: running twice with same input yields same result.

4.3 Auto-propagation flow (decision #3)

```
PUT /api/v1/udt/definitions/:id
  ↓
udtEngine.updateDefinition(id, newData)
  ↓ (after save)
udtEngine.propagateDefinitionChange(id)
  ↓
For each instance where instance.udt_id === id:
  result = generateTags(instance.id)
  Aggregate: definitionsAffected += 1, tagsCreated += result.created, ...
  ↓
Return to client:
  {
    ok: true,
```

```
    definition: {...},
    propagation: {
      instances_affected: 5,
      tags_created: 2,
      tags_updated: 28,
      tags_removed: 1
    }
  }
```

UI shows toast: "Updated. 5 instances synced (2 new tags, 1 removed)".

4.4 Cascade delete

```
deleteInstance(id):
  1. Find all tags where _udt_instance === instance.instance_name
  2. Remove them from project.tags
  3. Tell alarmEngine to cleanup any alarms tied to those tags
  4. Remove instance from project.udt_instances
  5. Save project
  6. Audit log

deleteDefinition(id):
  1. Count instances where udt_id === id
  2. If count > 0: reject with 409 "Cannot delete: N instances exist. Delete
instances first."
  3. Else: remove from project.udt_definitions, save, audit
```

---

*5. REST API — mounted at /api/v1/udt/**

All endpoints require role `admin` or `engineer`. All return `application/json`.

5.1 Definitions

`GET /api/v1/udt/definitions`

List all UDT definitions in active project.

Response 200:


```
{
  "ok": true,
  "data": [
    {
      "id": "udt_pump_centrifugal",
      "name": "Pump Centrifugal",
      "category": "Rotating Equipment",
      "icon": "fan",
      "version": "1.0.0",
      "property_count": 6,
      "instance_count": 5,
      "_created_at": "...",
```

```
      "_updated_at": "..."
    }
  ]
}
```

GET /api/v1/udt/definitions/:id

Full definition with properties.

Response 200: the full definition object (section 3.1). **Response 404:** `{ "ok": false, "error": "Definition not found" }`

POST /api/v1/udt/definitions

Request body: `{ name, description?, category?, icon?, version?, properties? }`.

Response 200: saved definition object. **Response 400:** validation error.

PUT /api/v1/udt/definitions/:id

Request body: any subset of mutable fields. `id` is immutable.

Response 200:

```
{
  "ok": true,
  "definition": {...},
  "propagation": {
    "instances_affected": 5,
    "tags_created": 2,
    "tags_updated": 28,
    "tags_removed": 1
  }
}
```

DELETE /api/v1/udt/definitions/:id

Response 200: `{ "ok": true }` **Response 409:** `{ "ok": false, "error": "Cannot delete: 5 instances exist." }`

5.2 Instances

GET /api/v1/udt/instances

Response 200:

```
{
  "ok": true,
  "data": [
    {
      "id": "udti_a8f4b2c1",
      "udt_id": "udt_pump_centrifugal",
      "udt_name": "Pump Centrifugal",
      "instance_name": "Pump_01",
```

```
      "group": "Plant1/SkidA",
      "mapping_status": { "total": 6, "mapped": 4, "unmapped": 2, "complete":
 false, "pct": 67 },
      "tag_count": 6,
      "_updated_at": "..."
    }
  ]
}
```

GET /api/v1/udt/instances/:id

Full instance with `io_mapping` and `tags` array.

POST /api/v1/udt/instances

Request body: `{ udt_id, instance_name, description?, group?, io_mapping? }`.

Validation: - `udt_id` must exist - `instance_name` must be unique in project - `instance_name` must match `/^[A-Za-z][A-Za-z0-9_-]*$/`

Response 200: saved instance + initial empty io_mapping (one entry per property).

PUT /api/v1/udt/instances/:id

Request body: any subset of mutable fields. If `io_mapping` is in body, tags are regenerated automatically.

Response 200:

```
{
  "ok": true,
  "instance": {...},
  "regenerated": { "created": 0, "updated": 6, "removed": 0 }
}
```

DELETE /api/v1/udt/instances/:id

Response 200:

```
{
  "ok": true,
  "tags_removed": 6
}
```

POST /api/v1/udt/instances/:id/generate-tags

Force regeneration without changing IO mapping. Useful as “Sync” action.

Response 200: same as PUT response above.

5.3 Diagnostics

GET /api/v1/udt/status

```
{
  "ok": true,
  "definitions_count": 4,
  "instances_count": 17,
  "tags_generated":   102,
  "instances_incomplete": 3
}
```

---

6. UI — three windows

6.1 Entry point

In `app/views/editor.html` left sidebar, the existing **"UDT"** placeholder item is wired to open `/udt-instances` as a floating window via the SYN-060 pattern (`setWindowOpenHandler`).

In `packages/editor/electron/main.js` `EDITOR_WINDOW_CONFIG`, add three entries:

```
'/udt-instances': { width: 1100, height: 700, title: 'UDT Instances — INDUXA SCADA', minWidth: 900, minHeight: 550 },
'/udt-library':   { width: 1000, height: 680, title: 'UDT Library — INDUXA SCADA',   minWidth: 800, minHeight: 550 },
'/udt-editor':    { width: 1100, height: 720, title: 'UDT Editor — INDUXA SCADA',    minWidth: 900, minHeight: 600 },
```

Server routes (in `server/routes.js` or equivalent):

```
GET /udt-instances → app/views/udt-instances.html
GET /udt-library   → app/views/udt-library.html
GET /udt-editor    → app/views/udt-editor.html
```

All three behind `requireAuth(['admin', 'engineer'])`.

Also add in the Editor menubar: `Tools → User Defined Types... → opens /udt-instances`.

6.2 /udt-instances (PRIMARY — most-used)

Layout: header + toolbar + table.

Header: - INDUXA logo + page title "UDT Instances" - Buttons: `[+ New Instance]` `[⧉ Library]`

Toolbar: - Search box (filters by instance_name, udt_name, group) - Filter pills: All / ✓ Complete / ⚠ Incomplete mapping - Counter "N instances"

Table columns: | Instance Name | UDT Type | Group | IO Mapping | Tags | Actions |

- **IO Mapping** column shows progress bar (4/6 ⚠ amber, 6/6 ✓ green)

- **Actions:** [🔌 IO Map] [⚡ Generate] [🗑 Delete]

Modals: - **New Instance:** select UDT type → name → group → description → save → opens IO modal automatically - **IO Mapping:** table `property × (protocol, connection, address)`, with progress bar live-updating, footer buttons `[Cancel]` `[Save Only]` `[Save & Generate Tags]` - **Delete confirm:** "Delete Pump_01? 6 tags will be removed."

6.3 `/udt-library`

Layout: header + toolbar + card grid.

Header: logo + "UDT Library" + `[+ New UDT]`.

Toolbar: search + category pills (auto-built from existing definitions) + counter.

Card per definition: - Icon (Lucide) + name + category + version badge - Description (if any) - Stats: "6 properties · 5 instances" - Buttons: `[Edit]` (opens `/udt-editor?id=xxx`) `[Use →]` (opens `/udt-instances` and pre-selects this UDT in New Instance modal) [🗑]

Modals: - **New UDT:** name (required), category, icon, version, description → on save opens `/udt-editor?id=xxx` for property definition - **Delete:** blocked if instance_count > 0, shows count and asks user to delete instances first

6.4 `/udt-editor?id=xxx`

Layout: two columns.

Left (300px): Template Info section — name, category, icon, version, description fields.

Right (rest): Properties section. - Header: "Properties (N)" + `[+ Add Property]` - Table: `Name | Type | Access | Unit | Scan Rate | Deadband | Logging | Alarm | Actions` - Row actions: edit / move up / move down / delete

Top bar: UDT name display + [💾 Save] + `[Exit]`.

Add/Edit Property modal: - Name (CamelCase validation) - Data Type (enum dropdown) - Access (read / readwrite / write) - Unit (text) - Default value - Scan rate ms - Deadband - Logging checkbox - "Configure alarm" toggle → reveals: priority, message, LL/L/H/HH (LL/L/H/HH only shown for numeric types)

Save behavior: - PUT to `/api/v1/udt/definitions/:id` - Backend auto-propagates - Toast: `"Saved. 5 instances synced (created 2 tags, removed 1)"` if propagation occurred - If no instances: simple `"Saved"`

6.5 Tag Manager touchpoints

Minimal: - Tags with `_udt_instance` show a small **UDT badge** (purple chip) next to the name - Hover badge: tooltip `"Belongs to Pump_01 (Pump Centrifugal) · click to open instance"` → click opens `/udt-instances?focus=<instance_id>` - Edit form for UDT-generated tags: most fields read-only with helper text `"Edit through UDT Definition`

(`Pump Centrifugal)`". Only `description` and `unit` remain editable as overrides (consistent with the existing comm-tag pattern). - Delete is blocked: "`Cannot delete UDT-generated tag. Delete the instance Pump_01 instead.`"

No “Create from UDT” button in Tag Manager toolbar — entry stays via Editor sidebar to keep responsibilities clean.

---

7. Permissions

Action	admin	engineer	operator	viewer
Read definitions/instances	✓	✓	—	—
Create/edit/delete definition	✓	✓	—	—
Create/edit/delete instance	✓	✓	—	—
Generate tags	✓	✓	—	—
View UDT-generated tags in Tag Manager (read-only fields)	✓	✓	✓	✓

---

8. Audit log entries

Event	Format
Definition created	`UDT definition created: pump_centrifugal (6 properties)`
Definition updated	`UDT definition updated: pump_centrifugal — propagated to 5 instances`
Definition deleted	`UDT definition deleted: pump_centrifugal`
Instance created	`UDT instance created: Pump_01 (Pump Centrifugal)`
Instance updated	`UDT instance io_mapping changed: Pump_01 — 6 tags regenerated`
Instance deleted	`UDT instance deleted: Pump_01 — 6 tags removed`
Tags generated	`UDT tags generated: Pump_01 — created 6, updated 0, removed 0`

---

9. Files to create / modify

Create

File	Purpose
`server/engines/udt-engine.js`	core engine
`server/routes/udt-routes.js`	REST endpoints (mounted by api.js)
`app/views/udt-instances.html`	primary instances window
`app/views/udt-library.html`	library window
`app/views/udt-editor.html`	template editor window
`app/js/udt-instances.js`	instances UI logic
`app/js/udt-library.js`	library UI logic
`app/js/udt-editor.js`	template editor UI logic
`app/css/udt.css`	shared styles for the 3 windows
`docs/SYN-UDT-001.md`	this document

Modify

File	Change
`server/api.js`	mount `/api/v1/udt/*` routes
`server/server.js`	init udtEngine after tagEngine + alarmEngine
`app/views/editor.html`	wire sidebar "UDT" item to open `/udt-instances`; add `Tools → User Defined Types...` menu entry
`app/js/editor.js`	sidebar click handler for UDT
`app/views/tags.html`	add UDT badge rendering + read-only behavior for `_udt_instance` tags + blocked delete
`app/js/tag-manager.js`	UDT badge logic + helper tooltip
`packages/editor/electron/main.js`	add 3 entries to `EDITOR_WINDOW_CONFIG`
`server/routes.js` (or routing module)	add 3 GETs returning the new HTML files
Project schema migration in project loader	initialize empty `udt_definitions[]` and `udt_instances[]`
`server/alarm-engine.js`	add `upsertUdtAlarm()` helper to manage alarms tied to UDT-generated tags

---

10. Implementation plan (sprints)

Sprint	Deliverable	Acceptance
MVP-1	Schema migration + udt-engine.js + REST endpoints +	curl-driven end-to-end works: create def → create instance

Sprint	Deliverable	Acceptance
	auto-propagation logic	→ set IO → tags appear in `project.tags`
MVP-2	`/udt-library` window: list/create/delete definitions	engineer can create a Pump Centrifugal template via UI
MVP-3	`/udt-editor` window: properties table + add/edit/reorder/delete + alarm config	engineer can add 6 properties with alarms
MVP-4	`/udt-instances` window: list + new instance modal + IO mapping modal + generate tags	full flow from sidebar to materialized tags via UI
MVP-5	Editor sidebar wiring + Tools menu + Tag Manager UDT badges + read-only fields	UDT discoverable from Editor; generated tags clearly distinguished
MVP-6	Cascade delete + audit log + edge cases (rename, dup names, validation messages) + QA	spec all green
Phase 1.5	IO mapping accepts `gateway-sql` and `gateway-api` protocols	UDT instance can read from a SQL query via Gateway tag-bridge
Phase 2	Inheritance (`parent_id`), nested UDTs, per-instance overrides, “Export to Library” (`.sdt` files)	spec to be written
Phase 3	Faceplates (visual template + creator window + runtime overlay)	separate spec SYN-UDT-007

---

11. Edge cases & decisions

Case	Behavior
Engineer renames a property in definition	Treated as delete-old + create-new. Existing values lost. UI warns: `"Rename detected: 'OldName' will be removed and 'NewName' created blank. Continue?"`
Engineer changes property data_type	Tag is updated; current value reset to default. Toast warning.
Engineer removes a property from definition	Cascade: each instance’s tag for that property

Case	Behavior
	is deleted. Auto-propagation report shows `tags_removed`.
Engineer renames an instance	All tags renamed `OldName/X → NewName/X`. History references stay valid (tags identified by id, not name).
Two instances with same `instance_name`	Rejected at creation. POST returns 400.
Definition referenced by 5 instances; user tries to delete	Reject with 409, force user to delete instances first.
Project has `udt_definitions` array but loaded into older INDUXA version	Older version ignores unknown arrays. Re-saving in older version preserves them (JSON.parse → JSON.stringify roundtrip).
IO mapping incomplete (some properties unmapped)	Tag is still created with `protocol: 'internal'` as fallback, value defaults to property default. Mapping bar shows ⚠.
Connection referenced by `connection_id` is deleted	Tags become orphaned. Tag Manager shows quality bad. UDT instance status shows ⚠. User must remap.

---

12. Out of scope explicitly

- Faceplates (visual templates that open as modal overlays). Tracked separately.
- UDT inheritance (`parent_id` is reserved in schema, ignored in MVP).
- Nested UDTs (a UDT containing instances of other UDTs).
- Per-instance property override of values beyond IO mapping.
- Cross-project UDT library (`.sdt` files in `data/udts/`).
- Bulk UDT operations ("regenerate all", "remap all instances of type X").
- UDT versioning with migrations (changing version number is currently cosmetic).

These are deferred to Phase 2 / Phase 3 specs to keep MVP shippable.

---

13. Verification checklist

- ☐ Create a new project, no `udt_*` arrays yet → arrays auto-init on save
- ☐ Create UDT "Pump Centrifugal" with 6 properties via `/udt-library` + `/udt-editor`
- ☐ Create instance "Pump_01" → IO mapping modal opens automatically
- ☐ Map all 6 properties to a Modbus connection → "Save & Generate Tags" → 6 tags appear in Tag Manager
- ☐ Tags have UDT badge in Tag Manager

- ☐ Edit Pump_01's IO (move Speed to OPC-UA) → tag updated, value preserved if same data_type
- ☐ Open `/udt-editor` for Pump Centrifugal, add a 7th property → 1 new tag appears in each instance, toast "5 instances synced"
- ☐ Try to delete the definition → blocked with 409 message
- ☐ Delete Pump_01 instance → 7 tags removed
- ☐ Now delete the definition → succeeds
- ☐ Project save/load roundtrip preserves `udt_definitions` and `udt_instances`
- ☐ operator role cannot reach any `/udt-*` URL (returns 403 / redirect)

End of spec. Implementation starts at MVP-1.

Alarmas

INDUXA SCADA implementa un motor de alarmas alineado con **ISA-18.2**: cada alarma tiene un ciclo de vida claro (Normal → Unack → Acked → Unack/RTN → Normal), prioridad, persistencia, y auditoría completa de eventos.

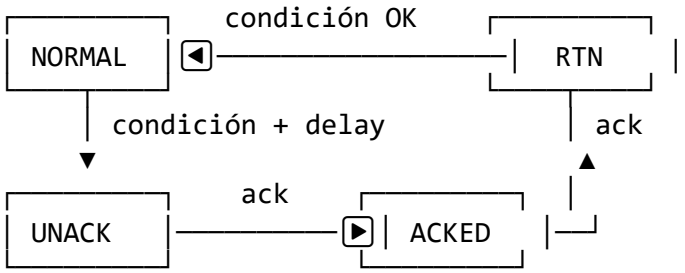
Anatomía de una definición de alarma

```
{
  "id":           "al_overtemp",
  "name":         "Reactor over-temperature",
  "tag_id":       "tag_reactor_temp",
  "priority":     "High",
  "condition":    "{REACTOR_TEMP} > 80",
  "message":      "Reactor over-temperature: {REACTOR_TEMP}°C",
  "deadband":     1.5,
  "delay_ms":     2000,
  "ack_required": true,
  "auto_clear":   true,
  "group":        "Reactor"
}
```

Campo	Significado
`condition`	expresión IEL (Capítulo 4)
`priority`	`Low` / `Medium` / `High` / `Critical`
`deadband`	banda muerta de retorno a normal
`delay_ms`	tiempo que la condición debe sostenerse antes de disparar
`ack_required`	si `false`, la alarma se autoreconoce
`auto_clear`	si `true`, vuelve a Normal cuando la condición

Campo	Significado
	ya no se cumple

Estados ISA-18.2



- **NORMAL**: condición cumplida en el sentido “no alarma”.
- **UNACK**: condición se ha disparado, falta acuse del operador.
- **ACKED**: operador ha confirmado la alarma; sigue presente.
- **RTN** (Return to Normal): condición ya no se cumple, falta acuse.

Acuse de alarmas

Desde un widget *button* o un script:

```
await fetch('/api/v1/alarms/' + alarmId + '/ack', {
  method: 'POST',
  credentials: 'same-origin',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ note: 'Revisado por turno A' }),
});
```

El acuse exige permiso `ackAlarms` (admin, engineer u operator).

Histórico

Cada transición de estado se persiste en SQLite. Consulta:

```
GET /api/v1/alarms/history?from=…&to=…&priority=High&limit=200
```

Caso de uso

UC-23 — Alarma de sobre-temperatura con histéresis

- `condition`: `{REACTOR_TEMP} > 80`
- deadband: `1.5` → la alarma vuelve a NORMAL solo cuando `REACTOR_TEMP <= 78.5`.
- `delay_ms`: `2000` → la condición debe sostenerse 2 s para evitar spurious triggers.
- `priority`: `High` → bandera roja en el alarm-banner.

UC-24 — Alarma generada desde UDT

Las propiedades de un UDT pueden declarar bloques `alarm`:

```
{
  "name":  "Fault",
  "alarm": { "priority": "High", "message": "{instance} fault" }
}
```

Al instanciar el UDT, cada `Pump_xx/Fault` genera automáticamente la alarma `Pump_xx fault`. Al borrar la instancia, la alarma se borra.

Máquinas de estado

Las **máquinas de estado finito** (FSM) modelan procesos discretos con número acotado de modos: arranque por etapas, secuencia batch/CIP, secuencias de seguridad, modos de operación de un equipo. INDUXA SCADA ejecuta FSMs declarativas en el server, sin necesidad de código.

Anatomía

```
{
  "id":           "fsm_pump_seq",
  "name":         "Pump start sequence",
  "initial":      "Idle",
  "states": [
    { "name": "Idle",     "entry": [ { "tag": "PUMP_RUN", "value": false } ]
 },
    { "name": "Priming",  "entry": [ { "tag": "PUMP_PRIME", "value": true } ]
 },
    { "name": "Running",  "entry": [ { "tag": "PUMP_RUN",  "value": true  },
                                     { "tag": "PUMP_PRIME","value": false } ]
 },
    { "name": "Stopping", "entry": [ { "tag": "PUMP_RUN", "value": false } ] }
  ],
  "transitions": [
    { "from": "Idle",     "to": "Priming",  "on": "tag_rising",   "tag": "STA
RT_REQ" },
    { "from": "Priming",  "to": "Running",  "on": "tag_duration", "tag": "PRI
ME_OK", "value": true, "duration_ms": 5000 },
    { "from": "Running",  "to": "Stopping", "on": "tag_compare",  "expr": "{S
TOP_REQ} || {FAULT}" },
    { "from": "Stopping", "to": "Idle",     "on": "elapsed",      "duration_m
s": 3000 },
    { "from": "any",      "to": "Idle",     "on": "tag_rising",   "tag": "EME
RGENCY_STOP" }
  ],
  "output_tag": "PUMP_FSM_STATE"
}
```

El estado actual se expone como string en `output_tag`. Una transición desde `from: "any"` cubre todos los estados (típicamente para emergencias).

Tipos de transición

on	Disparo
`tag_rising`	flanco de subida del tag (false → true)
`tag_falling`	flanco de bajada (true → false)
`tag_compare`	expresión IEL evalúa a verdadero
`tag_duration`	el tag mantiene `value` durante `duration_ms`
`elapsed`	han pasado `duration_ms` desde la entrada al estado
`always`	inmediato (útil para encadenar)

Acciones de entrada

`entry` es un array de escrituras a tags al entrar a un estado. Las escrituras se hacen secuencialmente. Si fallan, la transición queda registrada pero el estado se considera entrado de todos modos (los efectos colaterales son responsabilidad del diseñador).

Caso de uso

UC-25 — Secuencia batch con cancelación

Estados: `Idle → Filling → Mixing → Heating → Discharging → Idle`. Se permite cancelar volviendo a `Idle` en cualquier momento si se levanta `CANCEL_REQ`.

```
"transitions": [
  { "from": "Idle",       "to": "Filling",    "on": "tag_rising",   "tag": "S
TART_BATCH" },
  { "from": "Filling",    "to": "Mixing",     "on": "tag_compare",  "expr": "
{LEVEL} >= {TARGET_LEVEL}" },
  { "from": "Mixing",     "to": "Heating",    "on": "elapsed",      "duration
_ms": 60000 },
  { "from": "Heating",    "to": "Discharging","on": "tag_compare",  "expr": "
{TEMP} >= {TARGET_TEMP}" },
  { "from": "Discharging","to": "Idle",       "on": "tag_compare",  "expr": "
{LEVEL} <= 5" },
  { "from": "any",        "to": "Idle",       "on": "tag_rising",   "tag": "C
ANCEL_REQ" }
]
```

Un widget *state-timeline* enlazado a `BATCH_FSM_STATE` muestra al operador la secuencia y el tiempo en cada paso.

Material técnico de referencia

Especificación del motor de state machines

> Source: `server/state-machine-engine.js`. Tests: `test/state-machine-engine.test.js`. Editor UI: the **States** tab in the editor sidebar.

What it is

A finite-state-machine (FSM) evaluator that runs every 200 ms (configurable) and:

1. Evaluates each enabled state machine's transitions against tag values.
2. Updates auto-generated `SM_<SLUG>_*` tags so the rest of the system (alarms, scripts, widgets) can react with normal binding logic.
3. Persists per-FSM history of transitions to `data/sm-history/<smId>.json`.
4. Surfaces engine-level errors as alarms via the alarm engine.

Data model

A state machine in `.syk`:

```
{
  "id":           "sm_blower_ctrl",
  "name":         "Blower 1 Ctrl",
  "enabled":      true,
  "initialState": "idle",
  "states": [
    { "id": "idle",     "label": "Idle",     "onEntry": [...], "onExit": [...]
 },
    { "id": "starting", "label": "Starting", "tag_duration": 5 },
    { "id": "running",  "label": "Running"   },
    { "id": "fault",    "label": "Fault"     }
  ],
  "transitions": [
    { "from": "idle",     "to": "starting", "condition": "{ENABLE} && !{FAULT}
", "actions": [...] },
    { "from": "starting", "to": "running",  "condition": "{TIMER_DONE}" },
    { "from": "running",  "to": "fault",    "condition": "{FAULT}" }
  ]
}
```

Transitions are evaluated in array order; the first true `condition` wins.

Auto-generated tags

For every FSM the engine creates internal tags (idempotent on reload):

Suffix	Type	Meaning
`_STATE`	string	Current state id (`'idle'`)
`_STATE_LABEL`	string	Human label (`'Idle'`)

Suffix	Type	Meaning
_RUNNING	bool	True while the FSM is enabled and ticking
_ENABLE	bool	Operator-writable on/off switch
_ELAPSED	int	Seconds in current state
_UPTIME_S	int	Seconds since FSM started
_TRANSITIONS	int	Cumulative transition count
_MESSAGE	string	Last message emitted by tag_pulse / actions
_QUALITY	int	Aggregate health (0=bad, 1=uncertain, 2=good)
_STUCK	bool	True if state lasts longer than its threshold
_ERROR	bool	Engine raised an error during eval
_LAST_ERROR	string	Last error message

The slug is `_slugify(name || id)`. Examples:

Display name	Slug	Sample tag
Reactor V101 CIP	REACTOR_V101_CIP	SM_REACTOR_V101_CIP_STATE
Blower 1 Ctrl	BLOWER_1_CTRL	SM_BLOWER_1_CTRL_STATE_LABEL
Pump #2 (Aux)	PUMP_2_AUX	SM_PUMP_2_AUX_RUNNING

Cascade rename

When the engineer renames an FSM, every existing SM_<OLD_SLUG>_* tag is renamed to SM_<NEW_SLUG>_*. This used to be done with a regex that greedily ate the slug — it caused _CTRL_CTRL_CTRL growth on every reload for FSMs whose slug ended in an uppercase token.

The current implementation uses a **known-suffix table** sorted longest first (so _STATE_LABEL matches before _STATE):

```
const SM_SUFFIXES = [
  '_STATE_LABEL', '_LAST_ERROR', '_TRANSITIONS', '_UPTIME_S',
  '_RUNNING', '_MESSAGE', '_ELAPSED', '_ENABLE',
  '_QUALITY', '_ERROR', '_STUCK', '_STATE',
];
```

Tested in `test/state-machine-engine.test.js` — three back-to-back reload cycles must leave the names unchanged.

Action types

Every state can have onEntry / onExit arrays. Transitions can carry `actions`. Each action has a `type`:

Type	Effect
tag_write	Set a tag to a literal value or SEL expression

Type	Effect
`tag_pulse`	Set, then revert to a default value after Nms
`tag_increment`	Atomic +1 / -N (added recently)
`log`	Append a line to `_MESSAGE` and the audit log

Conditions and `tag_write` values are SEL expressions (see `SEL-EXPRESSION.md`).

Lifecycle

Hook	Called from
`start()`	`server.js` after engines are wired
`stop()`	`process.on('SIGINT/...')` — flushes history sync
`reload()`	After `switchActiveProject()` — keeps history per id
`tickAll()`	Internal 200 ms timer

A clean shutdown calls `flushAllHistory()` so the last few transitions are not lost.

History

Per-FSM ring buffer (default `HISTORY_MAX = 500`) persisted to `data/sm-history/<smId>.json` with debounced writes (`HISTORY_DEBOUNCE_MS = 1500`). The Editor's States tab and the Viewer's History overlay both consume this.

Common gotchas

- An FSM whose **initialState** does not match any state id is logged as an error and the FSM stays disabled.
- `tag_duration` is **wall-clock** seconds, not eval-tick count — it uses the state-entry timestamp.
- If two transitions from the same state have overlapping conditions, the **first one in array order** wins. There is no priority field; reorder them.
- The cascade-rename only touches tags marked `auto_generated:true, generated_by:'state-machine'`. Manual tags whose name happens to start with `SM_` are never renamed.

Scheduler

El **Scheduler** ejecuta jobs JavaScript periódicos o disparados por evento. Es la solución idiomática para lógica que requiere temporizadores, ráfagas controladas o llamadas a servicios externos — casos donde un script de tag no encaja porque corre síncrono y acotado a 500 ms.

Tipos de job

`type`	Disparo	Configuración relevante
`clock`	cada N ms	`interval_ms` (mínimo 250 ms)

type	Disparo	Configuración relevante
`calendar`	expresión cron	`cron` (5 campos estándar)
`change`	cuando un tag cambia	`tag`, opcional `value`
`startup`	una vez, 2 s tras arrancar el server	—

Anatomía

```
{
  "id":          "job_pump_runtime",
  "name":        "pump-runtime",
  "type":        "clock",
  "interval_ms": 60000,
  "enabled":     true,
  "timeout":     5000,
  "script":      "const r = Induxa.tag.read('PUMP_RUN'); if (r && r.value) { c
onst m = Induxa.tag.read('PUMP_RUN_MIN'); Induxa.tag.write('PUMP_RUN_MIN',
(m?.value||0) + 1); }"
}
```

`timeout` por job: por defecto 5 s, mínimo 1 s, máximo 30 s. Si se excede, el job se aborta y se registra `_lastResult: 'timeout'`.

Estado en runtime

El scheduler mantiene metadatos por job:

Campo	Significado
`_lastRun`	timestamp ISO de la última ejecución
`_lastResult`	`success` / `error` / `timeout`
`_lastError`	mensaje del último error
`_lastMs`	duración de la última ejecución

Estos campos **no** se persisten en el `.syk`: se reinician en cada arranque. Consulta:

`GET /api/v1/scheduler/jobs/:id/status`

Casos de uso

UC-26 — Acumular minutos

```
// type=clock, interval_ms=60000
const r = Induxa.tag.read('PUMP_RUN');
if (r && r.value) {
  const m = Induxa.tag.read('PUMP_RUN_MIN');
  Induxa.tag.write('PUMP_RUN_MIN', (m?.value || 0) + 1);
}
```

UC-27 — Backup nocturno

```
// type=calendar, cron='0 2 * * *'
const ts = new Date().toISOString().slice(0, 10);
Induxa.tag.write('LAST_BACKUP_TS', ts);
Induxa.log('Daily backup stamped: ' + ts);
```

UC-28 — Resetear contadores al arranque

```
// type=startup
Induxa.tag.writeMultiple({
  ERRORS_TODAY: 0,
  WARNINGS_TODAY: 0,
});
Induxa.log('Daily counters reset on boot');
```

UC-29 — Reaccionar a un cambio sin script de tag

Útil si quieres **descoplar** la reacción de la definición del tag (p. ej. el tag está en un UDT compartido y no quieres editarlo).

```
// type=change, tag='ALARM_ACTIVE'
const v = Induxa.tag.read('ALARM_ACTIVE');
if (v && v.value) {
  Induxa.tag.write('NOTIFICATION_BADGE', (Induxa.tag.read('NOTIFICATION_BADGE
')?.value||0) + 1);
}
```

Recetas

Una **receta** es un conjunto nombrado de set-points y parámetros que define un modo de operación: producir el producto A vs el producto B, ejecutar un ciclo CIP corto vs uno completo, ajustar el lazo PID para una temporada distinta.

INDUXA SCADA mantiene un store de recetas por proyecto, persistente y versionable. Las recetas se aplican vía la API REST o se invocan desde scripts y widgets.

Estructura

```
{
  "id":          "rec_prod_a",
  "name":        "Producto A — Setup verano",
  "description": "Set-points de verano para línea A",
  "category":    "Producción",
  "enabled":     true,
  "params": {
    "TEMP_SP":     65.0,
    "FLOW_SP":     12.5,
    "TIMER_BATCH": 600,
    "RECIPE_NAME": "Producto A v3"
```

```
  },
  "_createdAt":  "2026-01-15T10:00:00Z",
  "_updatedAt":  "2026-04-22T14:30:00Z"
}
```

API

Endpoint	Método	Permiso	Acción
`/recipes`	GET	`viewSynoptic`	Listar recetas
`/recipes`	POST	`manageTags`	Crear receta
`/recipes/:id`	PUT	`manageTags`	Editar receta
`/recipes/:id`	DELETE	`manageTags`	Borrar receta
`/recipes/:id/apply`	POST	`operate`	Aplicar receta (escribe los tags listados)

`apply` ejecuta cada par `tag → valor` como un `writeWithContext` auditable. Si alguna escritura falla, se reporta en el resultado pero las anteriores no se revierten — el orden es el del array `params`.

Caso de uso

UC-30 — Cambiar receta desde la pantalla

Widget *dropdown* enlazado a un tag `RECIPE_SELECTED`. Botón *Aplicar*:

```
// onClick
const id = Induxa.tag.read('RECIPE_SELECTED')?.value;
if (!id) { Induxa.notify('Selecciona una receta', 'warn'); return; }

const ok = await Induxa.confirm('Aplicar receta ahora?');
if (!ok) return;

const r = await fetch('/api/v1/recipes/' + id + '/apply', {
  method: 'POST',
  credentials: 'same-origin',
});
const data = await r.json();
if (data.success) Induxa.notify('Receta aplicada', 'success');
else              Induxa.notify('Fallo: ' + data.error, 'error');
```

Drivers de campo

INDUXA SCADA conecta con dispositivos vía el **Gateway**, un proceso separado que aloja un driver por protocolo soportado. El Gateway publica los valores leídos al Server vía REST/WebSocket; los comandos del operador viajan en sentido contrario.

Modbus TCP / RTU

Soporta funciones estándar 0x01–0x06, 0x0F–0x10. Mapping de direcciones con prefijo de área (0xxxx coils, 1xxxx discretas, 3xxxx input regs, 4xxxx holding regs).

```yaml
type:           modbus
name:           PLC_1
host:           192.168.1.10
port:           502
unit_id:        1
poll_interval:  500     # ms
read_timeout:   2000
endianness:     'big'   # word order para float32 / int32
swap_words:     false
```

Tag típico	Modbus
`bool`, coil	`00033`
`bool`, discreta	`10001`
`int16`, holding	`40001`
`float32`, holding (par 40001+40002)	`40001`, `data_type: float32`

Calidad y manejo de errores

Si un poll falla (timeout, exception code), el driver:

1. Marca esos tags como `quality: BAD`.
2. Incrementa `COM_<connection>_ERRORS`.
3. Tras `max_consecutive_errors`, marca la conexión como `QUARANTINED` y reduce frecuencia de reintento.

MQTT

Soporta MQTT 3.1.1 y 5.0, TLS, autenticación user/pass o cert, last-will y persistencia. Un tag puede:

- Suscribirse a un tópico (`mqtt_topic_sub`) con wildcards.
- Publicar a un tópico (`mqtt_topic_pub`) en cada escritura.
- Filtrar mensajes por campo JSON (`mqtt_filter_field` / `mqtt_filter_value`).
- Extraer un valor de un payload JSON (`mqtt_json_path`).

Sparkplug B

Si la conexión está marcada como `sparkplug: true`, el Gateway decodifica automáticamente NBIRTH/NDEATH/NDATA y publica los metrics como tags. Soporta historian rebirth y REBIRTH command.

OPC-UA

Servidor cliente con soporte de:

- Sesiones autenticadas (anónimo, user/pass, X.509).
- Tipos de dato OPC-UA estándar.
- Suscripciones nativas (clase de scan onchange).
- Browse on demand para descubrir tags.

Siemens S7

Cliente nativo S7-300/400/1200/1500 vía libnodave-equivalent. Direcciones nativas (`DB10.DBD20`, `M5.0`, `I0.3`).

Internal y Calculated

No requieren driver físico. Internal vive en el Server; calculated se evalúa por el motor de tags calculados.

Diagnóstico

Cada conexión expone tags de diagnóstico autogenerados:

Tag	Significado
`COM_<name>_STATUS`	bool: conectado
`COM_<name>_QUALITY`	0=bad, 1=uncertain, 2=good
`COM_<name>_ERRORS`	contador de errores consecutivos
`COM_<name>_LATENCY`	ms del último poll
`COM_<name>_QUARANTINED`	nº de tags en cuarentena

Útiles para:

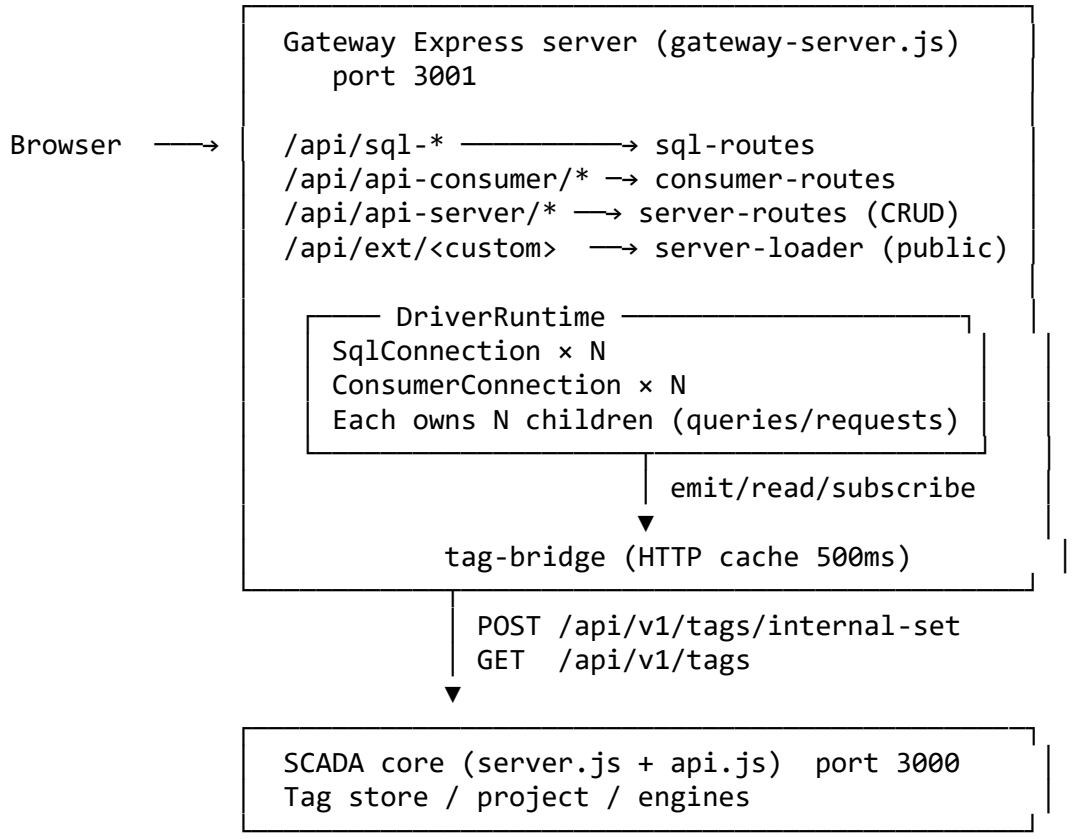
- Mostrar en una pantalla de “estado de comunicaciones”.
- Disparar alarmas sobre la planta de comunicaciones, no solo sobre el proceso.
- Loguear MTBF de comunicaciones.

Material técnico de referencia

SQL Query Tags y API Manager

Implementation reference for **SYN-SQLQT-001** and **SYN-APIMGR-001**. Status: **fully implemented** (backend + UI). Subject to follow-up adjustments.

High-level architecture



Storage

`data/db/gateway.db` — dedicated SQLite, WAL mode, FK enforced.

Table	Purpose
`_migrations`	applied migration log
`sql_connections`	SQL Query Tag connections
`sql_queries`	child queries per connection
`api_consumer_connections`	API Consumer connections
`api_consumer_requests`	child requests per connection
`api_server_keys`	issued API keys (hash only)
`api_server_endpoints`	exposed endpoints per key

Migrations are applied automatically on `getDb()`. Schema versioning is done by ID strings; new versions only need to add a new entry to the `MIGRATIONS` array in `gateway/db/gateway-config-db.js`.

Encryption (secret-vault)

`gateway/crypto/secret-vault.js` — AES-256-GCM symmetric encryption.

- Key derivation: PBKDF2-SHA256 over the gateway's `instance_id` (`data/instance.json`) with a per-vault salt (`data/.vault-salt`).
- Storage format: `{v:1, alg:'aes-256-gcm', iv, tag, ct}` JSON-serialised in TEXT columns.
- Pass-through for `$ENV:NAME` — value resolved at read time from `process.env`. Use this to keep credentials out of the DB entirely.
- API: `encrypt`, `decrypt`, `mask`, `serialize`, `deserialize`.
- Plaintext **never** returned by any HTTP API. UI shows `••••••••`. Updates that send the masked placeholder are silently ignored — to actually change a credential you must send the new plaintext.

Tag namespacing

Spec calls for paths like `/SQL/<conn>/<query>` and `/API/<conn>/<req>`. The SCADA tag engine uses **flat names**, so the driver emits flat:

- Connection status: `<DRIVER>_<conn>__status_connected` etc.
- Read result: `<DRIVER>_<conn>_<query>`
- Sidecar metadata: `<DRIVER>_<conn>_<query>__ok`, `__ts`, `__error`

The UI re-renders the flat name as a path (`/SQL/PG1/TEMP`) when shown in cards / config tables, but bindings, scripts and the Tag Manager still see the flat names.

Driver runtime (shared base)

`gateway/drivers/driver-runtime.js` exposes three classes used by both SQL and API Consumer:

- **`DriverRuntime`** — registry + lifecycle for one driver type. Holds Connection instances. `addConnection / removeConnection / startAll / stopAll`.
- **`DriverConnection`** — abstract `connect()` / `disconnect()` to be implemented by subclasses. Owns N children. Auto-reconnects on failure with exponential backoff (cap 60 s). Publishes per-conn status tags (`_status/connected`, `_status/last_connected`, `_status/error_msg`, `_status/latency_ms`).
- **`DriverChild`** — abstract `execute()` to be implemented. Owns the cyclic / on_tag_update / manual scheduling. Wraps each `execute()` in retry-with-backoff, applies error policy (`keep_last` / `null` / `default_value`), keeps an in-memory ring buffer of the last 100 executions for the `/log` endpoint.

SQL Query Tag

Engines supported

Engine	npm	Notes
`mysql`	`mysql2`	promise API, pool `connectionLimit`, `?` placeholders

Engine	npm	Notes
`mssql`	`mssql`	rewrites `?` → `@p0`, `@p1`, … internally
`postgres`	`pg`	rewrites `?` → `$1`, `$2`, … internally
`sqlite`	`better-sqlite3`	`cfg.database` is a file path; resolved relative to repo root

All adapters expose the same shape: `createPool(cfg)` returning `{ async query(sql, params), async end() }` plus a `testConnection(cfg)` that probes with `SELECT 1`.

Parameter substitution

In query SQL, `{{tag.<name>}}` placeholders are replaced before execution with positional `?` placeholders (the underlying engine binds the values type-safely). Example:

```
INSERT INTO events (ts, area, value)
VALUES (now(), {{tag.AREA_NAME}}, {{tag.TEMPERATURE}})
```

If a tag value is null at substitution time, `param_null_behavior` applies:

Value	Effect
`skip`	abort execution, log "Skipped — null parameter"
`use_null`	bind NULL
`use_default`	bind the configured `default_value`

Read mapping

For `type='read'`, the driver maps the **first row, first column** of the result to the tag value, then coerces to `data_type` (`string`/`float`/`integer`/`boolean`).

API Consumer

Auth strategies

`auth_type`	Behaviour
`none`	no headers added
`api_key`	`auth_value` injected as `auth_key_header` (default `X-API-Key`) or `auth_key_param` if set instead
`bearer`	`Authorization: Bearer <auth_value>`
`basic`	HTTP basic auth (`auth_username` / `auth_password`)
`oauth2_client`	`client_credentials` grant against `oauth2_token_url` with `client_id`/`client_secret`/`scope`. Token

auth_type	Behaviour
	cached in memory; refreshed automatically when now > expiry - 60 s.

Read mapping

Response is parsed as JSON; json_path (jsonpath-plus) extracts the single value (first match). Result coerced to data_type. Without a json_path the entire JSON is stored as a string.

Write requests

body_template is a JSON string with {{tag.x}} placeholders. Resolved against the tag bridge before send. param_null_behavior applies the same way as SQL.

API Server

Auth chain (every request to /api/ext/...)

1. Extract key from X-API-Key header or ?api_key= query
2. SHA-256 hash → look up api_server_keys.key_hash
3. Refuse if enabled=0 or expires_at < now
4. IP whitelist (CIDR allowed via ip-range-check)
5. Sliding 60-s window rate limit per key (rate_limit_rpm)
6. Permission: POST requires permissions='read_write'
7. Resolve resource via resource-handlers.execute()
8. Bump last_used_at, last_used_ip, requests_today

Errors: 401 invalid/missing/expired key, 403 ip/permission, 429 rate limit, 404 tag not found.

Resource types

resource_type	resource_target	Handler status
tag	tag name	**fully wired** — returns {tag, value, quality, timestamp}
tag_write	tag name	**fully wired** — body must include value
tag_history	tag name	**stubbed** — returns {_stub:true}, fillable from SCADA /api/v1/history
alarms_active	—	stubbed
alarms_history	—	stubbed
system_status	—	stubbed
connections_status	—	stubbed

The stubbed handlers return a documented placeholder so the contract is visible in development. Filling them in is a pure proxy job to existing SCADA REST endpoints with the service JWT.

Key generation flow

1. UI sends `POST /api/api-server/keys` with name + permissions + …
2. Server generates `sk_live_<32 base64url chars>`, stores SHA-256 hash
3. Response contains:
    - `data` — sanitized row (no hash, no plaintext)
    - `plain_key` — the only time the plaintext is ever sent
    - `_warning` — copy text for the UI
4. UI shows a one-time reveal modal with a copy button

To rotate a key: revoke (sets `enabled=0`) or delete and create new. Hash comparison uses SHA-256 (timing-safe via `crypto.subtle` would be a follow-up improvement; current `===` is acceptable since the plaintext is hashed before comparison and there's no length oracle).

Tag bridge

`gateway/drivers/tag-bridge.js` — HTTP plumbing between Gateway-side drivers and the SCADA tag store.

- `start(refreshMs=500)` — begins polling `GET /api/v1/tags` and populating an in-memory cache (name → {value, quality, ts}).
- `emitTag(name, value, opts)` — `POST /api/v1/tags/internal-set`. Auto- retries with `POST /api/v1/tags` if the tag doesn't exist (creates an internal+auto_generated tag, then re-POSTs).
- `readTag(name)` — synchronous cache lookup, used by `{{tag.x}}` substitution (no per-call HTTP).
- `subscribeToTag(name, cb)` — fires when the cached value changes between two refresh ticks. Used by `mode='on_tag_update'`.

The auth uses the gateway's service JWT (`{ username:'_gateway', role:'admin' }`), refreshed every 5 minutes by the existing gateway service-jwt machinery.

SCADA-side endpoint

```
POST /api/v1/tags/internal-set
{ name: "SQL_PG1_TEMP", value: 23.5, quality: "GOOD" }
```

Service-grade tag set used by Gateway-side drivers. Resolves by name, auto-creates the tag as internal + auto_generated when missing, skips operator write-audit, requires admin role (the service JWT qualifies).

UI

Both sections live under `packages/gateway/`:

- `js/gw-sql-querytag.js` — section module for SQL Query Tag
- `js/gw-api-manager.js` — section module for API Manager (Consumer + Server tabs)
- `views/gateway-v2.html` — adds two nav items + two `<script>` tags

Each module is self-contained: vanilla JS, no external deps, injects its own scoped CSS once. Polling refreshes the list every 4 s for live status colours.

Endpoints (full list)

SQL Query Tag

```
GET    /api/sql-connections
POST   /api/sql-connections
PUT    /api/sql-connections/:id
DELETE /api/sql-connections/:id
POST   /api/sql-connections/:id/test
POST   /api/sql-connections/test            (alias for new-conn modal)

GET    /api/sql-connections/:id/queries
POST   /api/sql-connections/:id/queries
PUT    /api/sql-queries/:id
DELETE /api/sql-queries/:id
POST   /api/sql-queries/:id/execute
GET    /api/sql-queries/:id/log
```

API Consumer

```
GET    /api/api-consumer/connections
POST   /api/api-consumer/connections
PUT    /api/api-consumer/connections/:id
DELETE /api/api-consumer/connections/:id
POST   /api/api-consumer/connections/:id/test
POST   /api/api-consumer/connections/test

GET    /api/api-consumer/connections/:id/requests
POST   /api/api-consumer/connections/:id/requests
PUT    /api/api-consumer/requests/:id
DELETE /api/api-consumer/requests/:id
POST   /api/api-consumer/requests/:id/execute
GET    /api/api-consumer/requests/:id/log
```

API Server (internal CRUD)

```
GET    /api/api-server/keys
POST   /api/api-server/keys                    → returns plain_key once
PUT    /api/api-server/keys/:id
DELETE /api/api-server/keys/:id
POST   /api/api-server/keys/:id/revoke

GET    /api/api-server/keys/:id/endpoints
POST   /api/api-server/keys/:id/endpoints
PUT    /api/api-server/endpoints/:id
```

```
DELETE /api/api-server/endpoints/:id
GET    /api/api-server/keys/:id/log
```

API Server (externally exposed, dynamic)

```
<METHOD> /api/ext/<path>      defined per (key, endpoint)
```

Routes are mounted on a dynamic Express router that gets wiped and rebuilt on every `reload()` (cheap for the expected scale: dozens of endpoints).

Known follow-ups

- Wire the four stubbed resource handlers (`tag_history`, `alarms_*`, `system_status`, `connections_status`) by proxying to existing SCADA REST endpoints.
- Use `crypto.subtle.timingSafeEqual` for API-key hash comparison.
- Editor-side bindings to the new tags work today (flat names go through the standard tag store) but the editor's tag picker doesn't surface the path representation. Cosmetic improvement only.
- Test connection in the Edit modal currently re-uses any password sent in body, otherwise the previously-stored one. Could be split into two endpoints if needed.
- File transfer: no quotas yet on the per-process driver pool — production deployments should set `pool_max` conservatively per database.

API REST y WebSocket

El Server expone una API REST versionada bajo `/api/v1` y un canal WebSocket en `/ws` para actualizaciones en tiempo real. El Editor, el Viewer y el Gateway son clientes de esta API; cualquier sistema externo puede integrarse usando los mismos endpoints, autenticando con JWT o (para el Gateway) con la clave de servicio.

Autenticación

```
POST /api/v1/auth/login
Body: { "username": "...", "password": "..." }
→ Set-Cookie: <session>; HttpOnly; SameSite=Strict
```

El token de sesión viaja en una cookie `httpOnly` de sesión firmada por el Server. Todas las llamadas posteriores deben incluir la cookie. Sesiones limitadas por rol (admin: ilimitadas; operator/ engineer: configurable; viewer: persistentes a través de reinicios).

Endpoints principales

Categoría	Endpoint	Notas
Tags	`GET /tags`, `POST /tags`, `PUT /tags/:id`, `DELETE /tags/:id`	`manageTags` para CUD
Escritura	`POST /tags/write`, `POST`	requiere `operate`

Categoría	Endpoint	Notas
	`/tags/write-batch`	
Force write	`POST /tags/:name/force-write`	`operate`, audit log obligatorio
Pantallas	`GET /screens`, `PUT /screens/:id`	`editSynoptic`
Alarmas	`GET /alarms`, `POST /alarms/:id/ack`	`ackAlarms`
Recetas	`GET /recipes`, `POST /recipes/:id/apply`	varios
Globales	`GET/POST/PUT/DELETE /global-scripts`	`manageTags`
Errores de scripts	`GET /scripts/errors`, `DELETE /scripts/errors`	`manageTags`
UDT	`GET /udt-defs`, `POST /udt-instances/:id/sync`	`manageTags`
Auditoría de escrituras	`GET /audit/writes`	`admin`

WebSocket

El cliente abre `/ws`, manda `{ type: 'auth', token: <JWT> }` y recibe:

Mensaje	Cuándo
`auth_ok` / `auth_error`	tras autenticación
`tag_update`	al cambiar el valor o calidad de un tag
`tags_changed`	al crear/editar/borrar tags (refresca el árbol)
`alarm_update`	nuevo estado de alarma
`project_saved`	el proyecto se ha guardado (Editor o Gateway)
`layout_updated`	la layout del Viewer ha cambiado
`global_scripts_changed`	un global cliente o gateway se editó (hot-reload)
`deploy_available`	hay un nuevo `.svk` listo para desplegar

> El Viewer suscribe `subscribe_alarms` tras `auth_ok` para recibir el flujo de alarmas. Otros canales (auditoría, sparkplug) tienen sus propias suscripciones.

Ejemplo de integración externa

Lectura sincronizada de un tag desde un servicio Python:

```python
import requests

s = requests.Session()
```

```
s.post('http://INDUXA-server/api/v1/auth/login',
       json={'username':'svc-bot','password':'…'})

r = s.get('http://INDUXA-server/api/v1/tags/REACTOR_TEMP')
print(r.json()['data']['value'])
```

Para integraciones de alta frecuencia, prefiere el WebSocket sobre polling.

Seguridad

Modelo

INDUXA SCADA implementa control de acceso basado en roles (RBAC), sesiones JWT firmadas, audit log de cada escritura, y separación de procesos Editor/Viewer/Gateway/Server. La seguridad se aplica **en el Server**: los clientes no son de confianza.

Políticas de contraseña

Configurables a nivel proyecto en `project.security.password_policy`:

Parámetro	Valores típicos
`min_length`	8–16
`require_uppercase`	true/false
`require_lowercase`	true/false
`require_digit`	true/false
`require_special`	true/false
`expiry_days`	0 (nunca) o 30/60/90
`history_size`	5 (no reusar las últimas N)
`max_failed_attempts`	5
`lockout_minutes`	15

Sesiones

Aspecto	Detalle
Token	JWT HS256, firmado con clave del proyecto
Transporte	cookie `httpOnly`, `SameSite=Strict`
Duración	configurable por rol (admin 8 h, viewer indefinido)
Revocación	server mantiene `_revokedJtis` en memoria + persistido para viewers
Restauración tras reinicio	viewers persistidos automáticamente
Sesiones simultáneas	limitadas por usuario (configurable)

Audit log

Cada escritura de tag (operador o servicio) genera una entrada en SQLite:

Campo	Significado
`timestamp`	epoch ms
`tag_name`, `tag_type`	tag escrito
`value_raw`, `value_prev`	valor escrito y valor anterior
`username`, `role`	derivados del JWT, **no** del request
`widget_id`, `widget_type`	origen UI
`confirmed`	si pasó por `confirm()`
`source`	`operator` / `gateway` / `script`
`session_id`	jti del JWT
`screen_name`	pantalla activa del operador
`result`	`success` / `error`; `error_msg` si falló

Consulta vía `GET /api/v1/audit/writes` (rol admin).

Trust del Gateway

El Gateway autentica al Server vía clave de servicio (`x-gateway-key`) o vía mTLS si está habilitado. El Server reconoce al Gateway como usuario `_gateway` con rol `admin` y registra sus escrituras con `source: 'gateway'` para distinguirlas de operadores humanos.

Comunicaciones cifradas

Canal	Cifrado
Editor ↔ Server	HTTPS recomendado en producción
Viewer ↔ Server	HTTPS recomendado en producción
Gateway ↔ Server	mTLS opcional, key-based fallback
MQTT externos	TLS 1.2+, autenticación user/pass o cert
OPC-UA	basic128Rsa15 / basic256Sha256
Modbus	sin cifrado (usar segmentación de red)

Recomendaciones de despliegue

1. **Sin Internet directo**: el Server debe estar en una red interna y exponerse vía VPN o reverse proxy con WAF.
2. **TLS obligatorio** en Editor/Viewer en producción.
3. **Rotar la clave del proyecto** al cambiar de personal con acceso admin.
4. **Backup encriptado** del `.syk` (contiene definición de usuarios, hashes, claves).
5. **Auditoría externa**: replicar los logs de escritura a un SIEM externo.

6. **Aislar el Gateway**: si se compromete el Server, el Gateway sigue de pie y mantiene la planta segura.

Buenas prácticas para scripts

- **Nunca** escribas credenciales o tokens en `Induxa.log` — acaban en stdout del server y de ahí en cualquier agregador.
- **Valida** entradas que vienen de la pantalla antes de escribir tags críticos (set-points, comandos de seguridad).
- **No confíes** en `Induxa.user.role` en server-side; el server ya autoriza, no necesitas re-validar (y si lo haces, asegúrate de que el campo viene del JWT, no de un parámetro de script).

Apéndice A — Referencia rápida de la API INDUXA

Server-side (tag y scheduler)

```
Induxa.tag.read(path)           → { value, quality, timestamp } | null
Induxa.tag.write(path, value)   → boolean
Induxa.tag.writeMultiple(map)   → boolean        (solo scheduler)
Induxa.log(message)             → void
Induxa.global.<fn>(...)         → user-defined gateway global
Induxa.db.query(...)            → reservado, lanza error
Induxa.http.post(...)           → reservado, lanza error
console.log                     → alias de Induxa.log
```

Sandbox variables (tag scripts):

```
tagPath, current, prev, isInitial, missed
```

Browser-side (pantallas y widgets)

```
Induxa.tag.read(name)                  → { value, quality, timestamp } | null
Induxa.tag.readMultiple(names)         → { name: { value, quality, timestamp }
 }
Induxa.tag.write(name, value)          → Promise<{ success, error? }>
Induxa.tag.writeWithContext(name, value, ctx) → Promise<{ success, writeId? }>
Induxa.tag.writeBatch(writes, ctx)     → Promise<{ success, results }>
Induxa.tag.writeMultiple(map)          → Promise<{ success, results? }>

Induxa.navigate(screenName)            → void
Induxa.openPopup(viewId, params)       → void
Induxa.notify(msg, type)               → void   ('info'|'success'|'warn'|'err
or')
Induxa.confirm(msg)                    → Promise<bool>

Induxa.user.name                       → string
Induxa.user.role                       → 'admin'|'engineer'|'operator'|'viewe
```

```
r'
Induxa.user.hasRole(min)              → bool
Induxa.user.getCurrentUser()          → string
Induxa.user.getCurrentRole()          → string

Induxa.screen.name                    → string  (solo en screen scripts)
Induxa.screen.previous                → string | null
Induxa.screen.next                    → string | null
Induxa.screen.setData(k, v)           → void
Induxa.screen.getData(k)              → any

Induxa.popup.params()                 → object  (solo en popup screen script
s)
Induxa.popup.id()                     → string
Induxa.popup.close()                  → void

Induxa.global.<fn>(...)               → user-defined client global
Induxa.db / Induxa.http               → no disponibles en navegador
```

Cuándo usar `writeWithContext`

Siempre que el origen de la escritura sea un widget o una acción de operador. La diferencia con `write` es que la entrada de auditoría incluye `widget_id`, `widget_type`, `confirmed` y `screen_name`, permitiendo trazar luego "qué botón fue pulsado y por quién".

Eventos de widget — `eventData` por evento

Evento	eventData
`onClick`, `onDoubleClick`, `onMouseEnter`, `onMouseLeave`	`{ x, y }`
`onRightClick`	`{ x, y }` (menú contextual nativo suprimido)
`onChange`	`{ value, checked }`

Códigos de retorno habituales

success	error (cuándo)
`false`	tag no existe → `Tag "X" not found`
`false`	tag readonly → `Tag "X" is read-only`
`false`	sin permiso → `Insufficient permissions`
`false`	timeout de driver → `Write timeout`
`false`	loop guard activo → escritura silenciosamente descartada (ver `/scripts/errors`)

Apéndice B — Glosario

Término	Definición
Binding	Conexión declarativa entre una propiedad de widget y un tag (modos `static`, `expr`, `state`).
Calidad (quality)	Atributo de cada tag: `GOOD` / `UNCERTAIN` / `BAD` / `STALE`. Indica la confianza en el valor.
Calculated tag	Tag cuyo valor se deriva de otros tags vía fórmula IEL o pipeline declarativa.
Driver	Componente del Gateway que habla con un protocolo de campo (Modbus, MQTT, OPC-UA, S7).
Edge case	Caso límite. En tags: tag con calidad `BAD` o sin valor histórico.
Editor	Aplicación de diseño donde se construye el proyecto.
FSM	Finite State Machine — máquina de estado finito declarativa, ejecutada en server.
Gateway	Proceso que aloja drivers de campo y publica al Server.
IEL	INDUXA Expression Language — lenguaje ligero de expresiones para bindings, fórmulas y condiciones.
Internal tag	Tag virtual residente en el Server, sin conexión a hardware.
JWT	JSON Web Token; formato de credencial usado para sesiones.
Lienzo (canvas)	Área de diseño de una pantalla, expresada en píxeles lógicos.
Loop guard	Mecanismo que detecta y pausa escrituras a un mismo tag con frecuencia anormal.
PLC	Programmable Logic Controller.
Project file	Archivo `.syk` que contiene la definición completa del proyecto.
Receta	Conjunto nombrado de set-points que se aplica como una transacción.
Scheduler	Motor de jobs JavaScript periódicos o disparados por evento.
Snapshot de despliegue	Archivo `.svk` derivado del `.syk`, optimizado

Término	Definición
	para el Viewer en producción.
Sparkplug B	Especificación de payload sobre MQTT para datos industriales (Eclipse Tahu).
Tag	Variable nombrada con `value`, `quality`, `timestamp`. Unidad atómica de información.
UDT	User-Defined Type — plantilla reutilizable para encapsular equipos.
Viewer	Aplicación cliente para operadores; renderiza pantallas y procesa interacciones.
Widget	Componente visual reutilizable del Editor (botón, gauge, gráfica, …).

Apéndice C — Preguntas frecuentes

¿Por qué mi script `await fetch(...)` no funciona en un script de tag?

Los scripts de tag corren en un sandbox **síncrono** del server, sin acceso a `fetch`, `setTimeout` ni `Promise` reales. Mueve la lógica a un **scheduler job** o, cuando esté disponible, usa `Induxa.http.post`.

El widget muestra `--` aunque el tag tiene valor

Tres causas comunes:

1. La calidad del tag es `BAD` o `STALE`. Mira el indicador del driver (`COM_<conn>_STATUS`).
2. El binding está en modo `expr` y la expresión IEL devuelve `null` por error de evaluación. Revisa `/api/v1/scripts/errors`.
3. La política de calidad del widget (`qualityPolicy`) está configurada como `placeholder`.

¿Mis expresiones de proyectos antiguos siguen funcionando?

Sí. La sintaxis de IEL es estable: cualquier expresión válida en versiones anteriores del producto sigue siendo válida hoy. No hay necesidad de actualizar `formula`, condiciones de alarma ni bindings de widget al actualizar la versión.

Cambié un global cliente y el viewer no lo aplica

Comprueba en DevTools → Network que el evento WebSocket `global_scripts_changed` ha llegado y que `GlobalScriptLibrary.reload()` se ha ejecutado. Si el navegador descartó el WebSocket (red inestable), una recarga manual basta.

¿Cómo limito que un viewer-rol pueda escribir tags?

No hace falta hacer nada — el Server bloquea cualquier escritura de un usuario sin permiso `operate`, devolviendo `403`. Si quieres que el botón **se vea** deshabilitado en el Viewer, configura `minRole` en el widget.

Mi script de tag se ejecuta una vez al arrancar y no quiero eso

Es la característica `isInitial`. Comprueba el flag al inicio:

```
if (isInitial) return;
// resto del script
```

¿Cuántos tags soporta el sistema?

Limitado por la licencia, no por el motor. Internamente, hasta ~50 000 tags con scan rates típicos en hardware moderno son manejables. Para cifras superiores, hablar con soporte.

¿Puedo versionar el `.syk` con git?

Sí. Es JSON con saltos de línea legibles. La recomendación es:

- Mantener un repositorio por proyecto en producción.
- Commits semánticos por cambio de ingeniería.
- Tags git para cada deploy a producción.
- No versionar `.svk` (es generado).

¿Cómo reseteo la sesión de un usuario sospechoso?

Endpoint `POST /api/v1/auth/sessions/:jti/revoke` (admin). El JWT queda invalidado al instante, incluso si no ha expirado.

El idle_timeout de la pantalla no se dispara

Asegúrate de:

1. Tener `idle_timeout >= 1000` (mínimo 1 segundo).
2. Tener un `onIdle` con código (no string vacío).
3. La pantalla está visible (`document.visibilityState === 'visible'`). Cuando la pestaña se oculta, el watcher se desactiva.