Event Engine™

User's Guide

Version 4.0.95

2026-10-10

Factory Data Systems

Event Engine™ User's Guide

Welcome to the Event Engine™ User's Guide. Event Engine is Factory Data Systems' high-performance middleware for executing events and managing data between your plant-floor control systems and your business systems.

> **Brand-new install?** Begin with **Installation & First Launch** to get the service running and reach the web UI. Already installed and signed in? Start with **Chapter 0 — Getting Started** — the step-by-step map from a blank installation to your first working event: what to gather, what order to build in and *why*, and a worked example you can follow end-to-end.

> **Upgrading an existing installation?** Read the **Release Notes** first — they list anything a release needs you to do by hand — then follow I.6 — Upgrading an Existing Installation.

Table of Contents

Chapter	Title
—	Installation & First Launch
0	Getting Started: From a Blank Slate to Working Events
1	System Overview
2	Understanding Event Data Flow
3	The System Menu — Status, Logs, Settings, Backup & Restore
4	Product Licensing
5	Configuration: Database Connections
6	Configuration: Processors
7	Configuration: Triggers
8	UDTs & Collections
9	Tags
10	Events: Stored Procedures
11	Events: Database Table Access
12	Events: ZPL Label Printing
13	Events: Message Notifications
14	Events: The Event List & Telemetry

Chapter	Title
15	Monitoring Event Health
16	Support Options
A	Appendix A — Glossary
B	Appendix B — Release Notes
—	End User License Agreement (PDF)

How to Read This Guide

Setting up a new system? Read these two short chapters first, then follow the build order:

- Chapter 0 — Getting Started is the roadmap: what to gather up front, the build order with the *reason* behind each step, and a complete worked example with verification checkpoints.
- Chapter 2 — Understanding Event Data Flow explains who does what (PLC, Event Engine, database, printer) at every step of an event — ten minutes that make every later choice easier.

The remaining chapters are ordered the way a new configuration is normally built. Each step exists to fill in something the next step needs, so building top-to-bottom means every screen is ready when you reach it:

#	Build this	Why now
1	License the product (Ch. 4)	Scanning is blocked until the product is licensed.
2	**Database Connections** (Ch. 5)	Events read their procedure/table lists from a connection.
3	**Processors** — the PLCs (Ch. 6)	Every tag must live in a processor.
4	**Trigger Templates** — *when* things happen (Ch. 7)	Tags and events pick triggers from these.
5	**UDTs / Collections** — structured data (Ch. 8)	A structure tag is an instance of one of these.
6	**Tags** — real data in the controllers (Ch. 9)	Events bind to tags for triggers, payload, and results.
7	**Events** — the work (Ch. 10–13)	Tie a connection, tags, and triggers together.
8	**Start & monitor** (Ch. 3, Ch. 14)	Start scanning, then prove each event end-to-end.
9	**Watch for drift** (Ch. 15)	Once events are running, this is what tells you one is degrading before it stops the line.

Each chapter explains not just *what* to click, but *what effect* every option has on the running system, so you can configure with confidence.

Installation & First Launch

> **Read this before Chapter 0 — Getting Started.** Chapter 0 begins at the web UI with a running service. This chapter gets you *to* that point: it installs Event Engine, starts its Windows service, and signs you in to the web interface for the first time. Once the home page loads, continue to Chapter 0 for the build order.

Event Engine runs as a **Windows service** that hosts its own web user interface. There is no separate web server (IIS) to configure — the service listens directly on the port you choose and every screen in this guide is reached from a browser. Installation therefore has three parts: meet the prerequisites, run the installer, and reach the web UI.

I.1 Prerequisites — What the Host Needs

Install Event Engine on a server (or always-on PC) that can reach your PLCs and your databases on the plant network.

Requirement	Detail
Operating system	64-bit Windows — Windows Server 2019/2022 (recommended for production) or Windows 10/11.
.NET runtime	**Two** x64 packages from Microsoft: the **.NET Runtime 10.0** *and* the **ASP.NET Core Runtime 10.0**. The ASP.NET Core installer does not include the .NET Runtime, so both are needed. Install them before running the installer — setup checks for both and names whichever is missing (see I.2).
Administrative rights	Local administrator on the host — the installer registers a Windows service and reserves the listening URL, both of which require elevation.
A listening port	The TCP port the web UI will use (default **80**). Pick a free port; if another web server already uses 80, choose something like 8080.
Service account *(optional but recommended)*	A Windows account for the service to run as. The default is `LocalSystem`. Use a dedicated **domain service account** if you will connect to databases with *integrated security*, reach network printers, or browse Logix controllers across domains.
An administrators group	A Windows or Active Directory group whose members may change configuration — by default the host's built-in `Administrators` group (see I.4), which already exists on every machine. Nothing needs to be created; map the role to a dedicated group later if you want to separate Event Engine administration from Windows administration.
Network reachability	The host must be able to reach each PLC, each database, your SMTP relay (for messaging), and — for online licensing — `secure.softwarekey.com`. See I.5.

> **Where Event Engine installs:** program files go to `C:\Program Files\Factory Data Systems, LLC\Event Engine`, and all *data* (configuration, logs, licenses, compiled UDTs, OPC UA certificates) goes under `C:\ProgramData\Factory Data Systems, LLC\Event Engine`. The data folder is what you back up (Chapter 3.5); the licenses are bound to the machine (Chapter 4).

I.2 Installing the Service

Event Engine ships as a **single signed installer** — `EventEngineSetup.exe`. It carries the application, every protocol plugin, and the system configuration it performs on the host itself. There is nothing to unzip and no script to run afterwards.

Copy `EventEngineSetup.exe` to the host and **run it**. A plain double-click is enough — the installer requests elevation itself; accept the UAC prompt.

If Windows warns about the installer

Windows may stop to warn you before the installer runs — once in the browser while the file downloads, and again when you open it. **For a new release this is expected and does not mean the file is unsafe.** `EventEngineSetup.exe` is digitally signed by **Factory Data Systems, LLC**, but Microsoft Defender SmartScreen also considers how often it has seen that exact file downloaded, and every new release starts out unfamiliar to it. So check the **publisher**, not whether a warning appeared. The wording varies by browser and Windows version; the check is the same everywhere:

- **In the browser, while downloading** — Microsoft Edge shows *Make sure you trust EventEngineSetup.exe before you open it* because the file *isn't commonly downloaded*. Confirm the **Publisher** line names **Factory Data Systems, LLC** — Edge puts the company's location in front of it: `US, New York, Hilton, "Factory Data Systems, LLC"`. Then open the arrow beside **Delete**, choose **Keep**, and if Edge asks again choose **Show more** → **Keep anyway**.
- **In Windows, when you open it** — *Windows protected your PC / Microsoft Defender SmartScreen prevented an unrecognized app from starting*. Click **More info**, confirm **Publisher: Factory Data Systems, LLC**, then click **Run anyway**.
- **At the UAC prompt** — confirm **Verified publisher: Factory Data Systems, LLC**, then click **Yes**.

> ⚠️ **Stop if the publisher is wrong.** If any of these shows **Unknown publisher**, or any name other than *Factory Data Systems, LLC*, do **not** run the file — it is not a genuine Event Engine installer, or it was changed after it was signed. Delete it, download a fresh copy, and contact support if it happens again. You can check any copy before running it: right-click it → **Properties** → **Digital Signatures**. The signer must be *Factory Data Systems, LLC*, and **Details** must report *This digital signature is OK*.

Two checks before the first wizard page

Setup verifies both of these before it shows anything, and stops without changing the machine if either fails:

- **.NET prerequisites.** Event Engine needs **two** x64 shared runtimes — the **.NET Runtime 10** *and* the **ASP.NET Core Runtime 10**. Both are checked, because Microsoft's ASP.NET Core Runtime installer does **not** include the .NET Runtime, and a host with only one of them would install cleanly and then fail to start. If either is missing, setup names it, offers to open Microsoft's download page, and exits — install the **Windows x64 Installer** package for each, then run setup again.
- **License entitlement (upgrades only).** On a machine that already holds Event Engine licenses, setup compares this build's release date against your maintenance coverage *before touching a single file*. If the build was released after your coverage ended it refuses, and your **existing installation is left running, unchanged** — the application itself declines to start on a build its license does not cover, so the installer stops early rather than stranding you. Renew maintenance first (Chapter 4). A machine with no licenses yet — a fresh install, or the 30-day trial — is never blocked.

The wizard

Page	What it asks
Welcome	Confirms the version you are about to install.
License Agreement	The Event Engine end-user license agreement. You must accept it to continue.
Destination Location	Where the program files go. Accept the default `C:\Program Files\Factory Data Systems, LLC\Event Engine` unless you have a reason not to.
Listening Port	The TCP port the web UI listens on. **80** is the standard browser port and is right for most installs — just click **Next**. Choose another (e.g. 8080) only if something on this computer already uses port 80. Any value from 1 to 65535 is accepted. On an **upgrade** the box is pre-filled with the port already configured, so a customized port survives the reinstall.
Ready to Install	A summary. **Install** starts the work.

What the installer does

When you click **Install**, setup:

1. **Stops the `Event Engine` service** if one is already registered — before any file is copied, so an upgrade never tries to overwrite files a running service holds open.
2. **Copies the program files.** On an upgrade your existing `appsettings.json` is **kept as-is** — it holds settings the product itself writes (role groups, session timeout, SMTP relay, log locations) —

and the release's own defaults are laid down beside it as `appsettings.json.default` for reference.

3. **Configures the system** — a hidden, elevated step (`setup\configure.ps1` under the program folder) that does all of the following:
    - **URL reservations.** Three **name-based** reservations, so the service may listen without running as an administrator: `localhost`, the machine name, and the fully-qualified domain name (when DNS resolves one distinct from the machine name) — e.g. `http://localhost:<port>/EventEngine/`, `http://<server-name>:<port>/EventEngine/`. **Note there is no reservation for the server's IP address** (see I.3).
    - **Firewall rule.** An inbound *allow* rule named `Event Engine` for TCP on the port you chose. Reinstalling with a different port replaces the rule rather than leaving the old port open.
    - **HTTPS.** Generates a certificate for this server — covering `localhost`, the machine name and the FQDN — trusts it on this machine, binds it to port **443** (or **8443** when 443 is already in use; `/HttpsPort=<n>` on a silent install forces the port), reserves the matching `https://` URLs, and adds a second inbound rule, `Event Engine HTTPS`. On an upgrade the existing certificate and port are reused, and an existing `"EnableHttps": false` in `appsettings.json` is respected — an opted-out installation stays HTTP-only. What this means for browsers on other computers, and how to substitute a company certificate, is covered in the Release Notes and Chapter 3.6.
    - **Listening port.** Writes your chosen port into the service's `appsettings.json`.
    - **Data directories.** Creates the compiled-UDT folders under `C:\ProgramData\Factory Data Systems, LLC\Event Engine` (`UDTs` and `UDTs\Code`) and **restricts write access to SYSTEM and Administrators**, leaving ordinary users read-only. The service loads every DLL in those folders with its own privileges, so a folder any user could write to would be a way to run code as the service.
    - **OPC UA certificate store.** Creates `OpcPki` and its four sub-stores — `own`, `trusted`, `issuer`, `rejected` — under the same data folder, locked down the same way. It lives outside the program folder so it survives upgrades, and it is deliberately **not** shipped inside the installer: this client's own certificate is generated on this machine on first connect, and what sits in `trusted` decides which OPC servers the client will talk to. See Chapter 6 — OPC UA Certificates and Trust.
    - **The Windows service.** Registers `Event Engine` (display name *Event Engine Service*), set to start automatically. It is created, not started — the finish page starts it.

The finish page

The last page lists **every address the site now answers on** — the `localhost`, machine-name and fully-qualified-domain-name URLs, over both `http://` and `https://`, each with the scope it covers. **Write one of them down now**; it is how you and everyone else will reach Event Engine, and this page is the only place the installer shows them. When HTTPS addresses are listed, the page also notes that the generated certificate is trusted on this computer only — browsers elsewhere will warn until it is distributed or a company certificate is imported (Chapter 3.6). Two checkboxes are offered:

- **Start the Event Engine service now** — leave it ticked. This is what first starts the service.
- **Open Event Engine in your browser** — opens the `localhost` address.

With the first box ticked, closing the wizard leaves the service installed and running.

> **Verify the service:** open **Services** (`services.msc`) and confirm **Event Engine Service** shows *Running* with startup type *Automatic*. To run it under a domain service account, stop it, set the account on the service's **Log On** tab, and start it again — but first grant that account write access to the ProgramData data folder above. The `UDTs` , `UDTs\Code` and `OpcPki` folders need **Modify** granted explicitly: inheritance is switched off on them by the lockdown described above, so an account added higher up does not pick up access there.

> **Uninstalling reverses all of it.** The uninstaller stops and deletes the service and removes the URL reservations, both firewall rules and the HTTPS binding it created — along with the generated HTTPS certificate, but **never** one you imported. Your data folder under `C:\ProgramData\Factory Data Systems, LLC\Event Engine` — configuration, licenses, logs, compiled UDTs and OPC UA certificates — is **left in place**.

I.3 First Launch — Reaching the Web UI

Event Engine's interface lives at the path `/EventEngine` on the port you chose, and must be reached **by host name** — `localhost` , the server's machine name, or its fully-qualified domain name:

```
http://<host>:<port>/EventEngine
```

- On the server itself, with the default port 80: **`http://localhost/EventEngine`**
- From another PC on the network: **`http://<server-name>/EventEngine`** (or the server's fully-qualified domain name). Use `http://<server-name>:<port>/EventEngine` if you chose a non-default port.

The site also answers on **HTTPS** — `https://<host>/EventEngine` (add `:8443` when the finish page showed that port) — and HTTP requests are redirected there while the certificate is healthy. On the server itself this just works; **from another PC the browser will show a certificate warning** until the generated certificate is trusted there or a company certificate is imported — that is expected, not a fault. See Chapter 3.6.

> ⚠️ **Use the server name — not its IP address.** Event Engine is hosted on the Windows **HTTP.sys** stack, which answers only the **named** URL reservations the installer created (`localhost` , the machine name, and the FQDN). Browsing to the server's **IP address** (e.g. `http://192.168.1.50/EventEngine`) will **fail to connect even though the service is running and the port is open** — this is contrary to how most web servers behave, so it surprises people. Always use the name. If a client PC can't resolve the server name, fix name resolution with a DNS entry (or a line in the client's `hosts` file) — do **not** fall back to the IP address.

EVENT ENGINE™

Event Execution & Management for Manufacturing Systems

Eventing with the FDS *Event Engine*

Real-Time eventing from all of your manufacturing processes.

Enterprise wide visibility to manage your *entire* factory.

Web enabled with mobility in mind makes your data always available.

Quick Reference Links

Dashboard
Get a quick look at the overall health and performance of the system.

Errors
Review the most recent errors caught by the system so you can rectify them.

Copyright © 2026 Factory Data Systems, LLC

The web UI uses **Windows Authentication** — your browser signs you in automatically with your Windows account; there is no separate Event Engine username or password. If the browser prompts for credentials, enter your Windows domain credentials.

> **If the page doesn't load:** confirm you used the **server name, not an IP address** (the most common cause — see the warning above); confirm the **Event Engine Service** is *Running*; confirm you used the `/EventEngine` path (the root `/` is not served); and confirm a firewall on the host allows inbound traffic on your chosen port. If you reach the site but are refused access, your account is not yet in one of the role groups below — or, if your account belongs to a **different domain than the server**, the mapped group may need to be written in its `DOMAIN\Group` form (see I.4).

I.4 Roles and Windows Groups

Event Engine has two roles, and **membership is determined by Windows / Active Directory group membership** — there are no application-managed user accounts.

Role	Default group	Can do
General	`Authenticated Users`	View and operate: see the dashboard, logs, configuration, and event telemetry; start/stop scanning; run manual triggers.
Admins	`Administrators`	Everything General can do **plus** change configuration — create/edit/delete processors, tags, triggers, connections, events, and system settings; restore backups; import UDTs.

Both defaults are standard Windows groups that exist on every machine, so a fresh install works the same whether the host is **domain-joined or in a workgroup** — no groups need to be created first. Any signed-in Windows account can view and operate, and anyone who is an **administrator of the host machine** (a member of its local *Administrators* group, which includes Domain Admins on a domain-joined server) can change configuration. To grant someone configuration rights, add their account to the host's **`Administrators`** group — or, to avoid giving them Windows admin rights, create a dedicated group and map the Admins role to it in the UI. Each role can map to **more than one** group, and membership in *any* mapped group grants the role.

> **Changing the group names — use the UI.** Once you can sign in, the role-to-group mapping is managed on **System** → **Settings**, in the **Security & Session** card, where each role is a searchable multi-select of the groups available on your domain or machine. See Chapter 3.3 — Security & Session for full details, including the session timeout and the guard that stops you removing your own administrative access. Group changes take effect immediately; session timeout changes take effect after the next service restart.
>
> The same mapping also lives in `appsettings.json` (under the program folder) in the `Security:Roles` section, which is the fallback for the very first sign-in before the UI is reachable:
>
> ```
> "Security": {
>   "Roles": { "Admins": "Administrators", "General": "Authenticated Users" }
> }
> ```
>
> On a standalone (non-domain) machine, use local groups and local accounts.

> **A forest with more than one domain.** If your users' accounts live in a different domain from the server — common on a large corporate network — map the roles to **domain-qualified** group names, written `DOMAIN\Group` (for example `CORP\Domain Users`). An unqualified name is searched only in the server's own domain, and every domain in a forest has its own group called *Domain Users*, so the qualifier is what pins down which one you mean. The picker on the settings page will list another domain's groups as soon as you type its name and a backslash. See Chapter 3.3.

> **You won't get permanently locked out.** If the configured `Admins` group does not exist at all (never created, or mistyped), Event Engine lets a member of the server's **local Windows *Administrators* group** in as a one-time recovery path, so they can open the Security & Session card and set a valid group. This applies only when the configured group is entirely unresolvable, and every such access is logged.

I.5 Network and Ports

Event Engine reaches out to your other systems; make sure the host's firewall and your network allow the traffic each feature needs. The values below are the **protocol defaults** — adjust to match how your devices and databases are actually configured.

Direction	For	Typical port(s)
Inbound	Browsers reaching the web UI (HTTP)	the port you chose at install (default **80**)

Direction	For	Typical port(s)
Inbound	Browsers reaching the web UI (HTTPS)	**443**, or **8443** when 443 was taken at install (Chapter 3.6)
Outbound	Rockwell Logix / Legacy (EtherNet/IP)	TCP **44818** (and UDP 2222)
Outbound	Siemens S7 (ISO-on-TCP)	TCP **102**
Outbound	Modbus/TCP	TCP **502**
Outbound	GE SRTP	TCP **18245**
Outbound	OPC	per your OPC server's configuration
Outbound	Databases	per connector — SQL Server **1433**, PostgreSQL **5432**, MySQL **3306**, Oracle **1521** (Chapter 5)
Outbound	SMTP relay (messaging / notifications)	your relay's port — commonly **587** (Chapter 3.3)
Outbound	Online licensing	HTTPS **443** to `secure.softwarekey.com` (Chapter 4)

I.6 Upgrading an Existing Installation

An upgrade installs over the existing installation. Your configuration, licenses, logs, compiled UDTs and OPC UA certificates live under `C:\ProgramData\Factory Data Systems, LLC\Event Engine` and are kept — the installer stops the service, replaces the program files, and starts it again.

1. **Read the Release Notes for the version you are moving to**, and work through its *Upgrade checklist*. Some releases need one-time manual work that no installer can do for you.
2. **Take a configuration backup** from **System** → **Backup/Restore Configuration** (Chapter 3.5). This is the supported way back if the upgrade has to be reversed.
3. **Run the installer** on the host, exactly as for a new installation (I.2) — it stops the service before replacing any file, keeps your `appsettings.json`, and pre-fills the **Listening Port** page with the port you are already using.
4. **Sign in and confirm the system starts scanning**, then check the Error List on **System** → **Status** for anything the upgrade left needing attention.

> **Upgrading adds HTTPS.** From this release the site answers on HTTPS and redirects HTTP to it — operators will notice the address change, and browsers on other computers warn until the certificate question is settled. The upgrade needs no manual work, but read *The web interface now serves HTTPS* in the Release Notes before user questions arrive. Opting out (`"EnableHttps": false`) is respected across upgrades.

> ⚠️ **Upgrading to 4.0.0019 or later with OPC processors on secured endpoints.** Event Engine's OPC UA client certificate changed in 4.0.0019, so **each OPC server must be told to trust the new certificate — once, per server** — before those processors will connect again. Processors on unsecured endpoints with the default security options are unaffected, as are all non-OPC protocols. The full procedure is in the Release Notes; the background, the certificate store and its four sub-stores are covered in Chapter 6 — OPC UA Certificates and Trust.

> **Licenses are checked against the build's date.** A license entitles you to builds released within its entitlement window, so a build released after your maintenance coverage ended is **refused by the installer before any file is touched** rather than installed and then failing to start — your current installation keeps running. See I.2 and Chapter 4.

I.7 What's Next

The service is installed and you can sign in. From here:

1. **License the product** (or start the 30-day trial) — Chapter 4.
2. **Follow the build order** to your first working event — Chapter 0 — Getting Started.

A fresh installation automatically begins a **30-day trial**, so you can build and prove your first event before a license is in place — but license early, so you learn how many events your license covers before you build past it.

Next: Chapter 0 — Getting Started: From a Blank Slate to Working Events.

Chapter 0 — Getting Started: From a Blank Slate to Working Events

This chapter is the **map**. The rest of the guide explains each part of Event Engine in detail; this chapter tells you what order to build things in, *why* that order matters, and what to verify before moving on — so a brand-new installation reaches its **first working event** with no backtracking.

If you read nothing else first, read this chapter and Chapter 2 — Understanding Event Data Flow. Together they take about fifteen minutes and make every later choice easier to reason about.

> **Haven't installed Event Engine yet?** Start with Installation & First Launch to get the Windows service running and reach the web UI — this chapter assumes you can already open the interface in a browser.

> **New to the concepts?** Skim Chapter 1 — System Overview for the vocabulary (processor, tag, trigger, master tag, event) used throughout this chapter.

0.1 Before You Begin — What to Gather

Most setup delays come from missing information, not from the software. Collect the items below *before* you sit down at the UI. Each row notes what the item is for and the chapter that uses it.

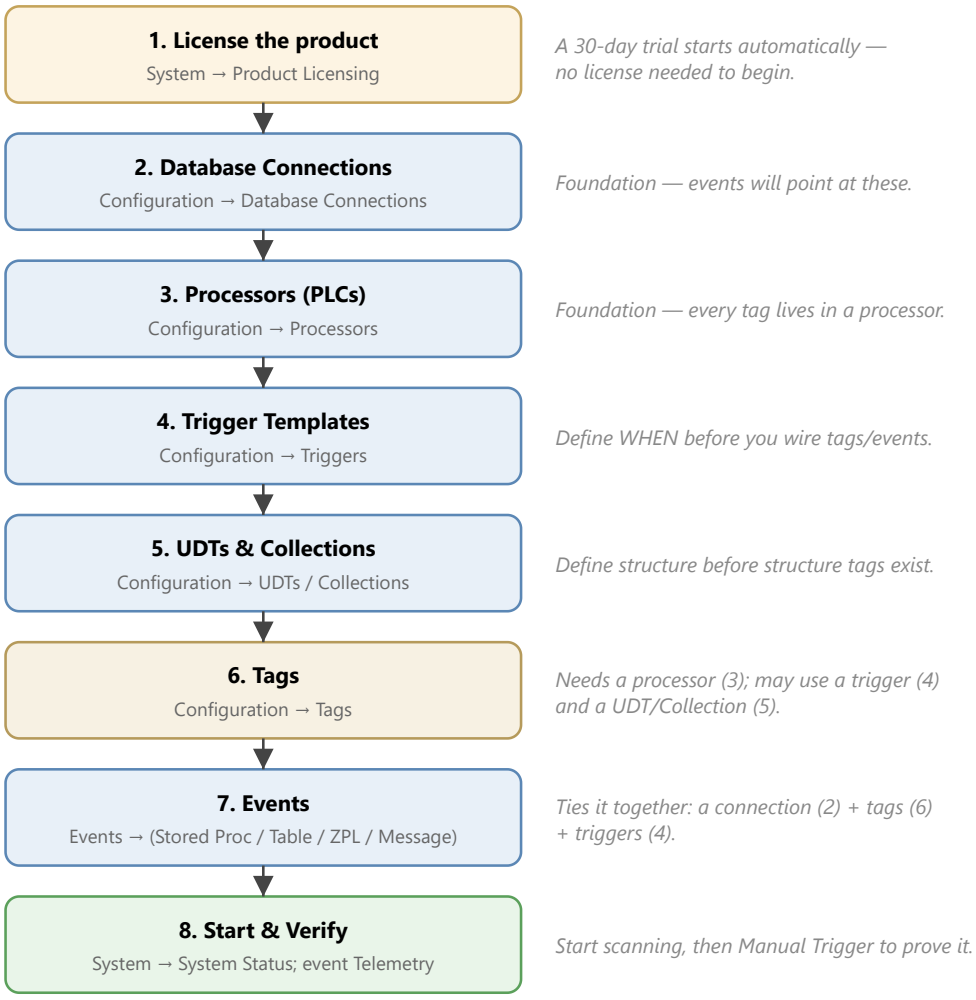
What to gather	Why you need it	Used in
License Id and License Password (from your purchase confirmation), plus the number of events you purchased	*Not needed to begin* — a fresh install runs on an automatic, fully-functional **30-day trial**. Gather these to run past the trial. Licensing meters **enabled events** only; every PLC protocol is available to every licensed installation	Ch. 4
Internet access status of the server (can it reach `secure.softwarekey.com`?)	Decides whether you license online or follow the offline file-exchange workflow	Ch. 4
Database host, database name, and a login (user/password, or confirmation that integrated security is set up) for each business database	Stored-procedure and table events connect through these; the login needs rights to the procedures/tables **and** to read their definitions	Ch. 5
PLC IP address(es), model, and rack/slot or routing path for each controller	Event Engine cannot read or write tags without reaching the controller	Ch. 6

What to gather	Why you need it	Used in
PLC structure exports — `.L5X` (Rockwell) or `.scl` / `.stl` / `.db` / `.udt` (Siemens) — for any UDTs you want to mirror	Lets Event Engine import the exact structures from your PLC program instead of re-typing them	Ch. 8
The tag names / addresses that act as triggers, payload, completion codes, and status messages	These are what your events read and write; Logix users can browse them live instead	Ch. 9
The exact Windows printer name(s) installed on the Event Engine host (ZPL events only)	A label event delivers to a printer by its exact installed name	Ch. 12
SMTP relay host, port, From address, and credentials (email/messaging only)	Required before any message event, failure notification, or system email can send	Ch. 3.3, Ch. 13

> **Tip:** you do not need *all* of these for every system — and you do not need a license at all to start, since a fresh install runs on an automatic 30-day trial. A pure data-logging system needs only a database connection, a processor, tags, and events (add a license before the trial ends). Gather the printer and SMTP rows only if you will print labels or send messages.

0.2 The Build Order (and Why It Matters)

Event Engine objects **depend on one another**, and the dependencies dictate the order. You build the foundations first (the things other objects point *at*), then the data, then the events that tie them together. Building out of order means coming back later to fill in a dropdown that was empty.



Read the order this way — each step exists to fill a dropdown the next step needs:

1. **License the product — or just use the automatic trial** (Chapter 4). You do **not** need to obtain or enter a license to get started: a fresh installation **automatically creates a fully-functional 30-day trial license** on first run, with the event-count limit bypassed. Every feature works out of the box — there is nothing to activate — so you can build and prove your first events on the trial alone. The Start button is disabled when there is *neither* a valid license *nor* an active trial, or when more events are enabled than the license allows. Activate a license before the 30 days run out — and do it early, so you learn how many events your license covers before you build past it.
2. **Create Database Connections** (Chapter 5). Do this before events, because choosing a connection is what lets an event list the database's stored procedures, tables, and columns. **Test Connection** before saving. *(Skip if you will only print labels or send messages.)*
3. **Create Processors** (Chapter 6) — one per PLC. Every tag belongs to a processor, so this must precede tags. Any protocol may be used: processor types are not separately licensed.
4. **Create Trigger Templates** (Chapter 7) — the reusable *when* rules. Build these before tags and events so the trigger dropdowns are populated when you wire them. You can pre-assign a tag

trigger on a tag (step 6) or even bake it into a Collection member (step 5).

5. **Import UDTs / build Collections** (Chapter 8) — only if your tags are *structures*. A structure tag is an instance of a UDT or Collection, so the type must exist first. Importing UDTs from your PLC export is the fastest, most faithful path. *(Skip if you only use atomic tags.)*
6. **Create Tags** (Chapter 9). A tag needs its processor (step 3); a structure tag needs its UDT/Collection (step 5); a trigger tag needs its template (step 4). Logix users should **Browse** the live controller so names and types match exactly.
7. **Create Events** (Chapters 10–13). This is where it all comes together: an event references a database connection (step 2), reads/writes tags (step 6), and fires on triggers (step 4). Using a **Master Tag** here auto-matches a structure's members to parameters/columns and filters the trigger/completion/message dropdowns for you.
8. **Start scanning and verify** (Chapter 3.1 and Chapter 14). Start from **System → System Status**, then use **Manual Trigger** on the event's Telemetry page to prove the full path end-to-end before you rely on the PLC handshake.

> **Why not just build events first?** Every event editor is a set of dropdowns that read from the objects above it — no connection means no procedure list; no tags means nothing to bind; no trigger means nothing to fire it. Building bottom-up means each screen is ready when you reach it.

0.3 Worked Example — "Log a Completed Part to the Database"

This example threads one realistic scenario through the whole build order. The goal: when the PLC finishes a part, Event Engine reads the part data, calls a stored procedure that logs it, and writes a completion code back so the PLC knows it succeeded. Adapt the names to your own system.

> Each step ends with a ✓ **Verify** checkpoint. Don't move on until it passes — catching a problem at its own step is far faster than debugging it from a failed event later.

Step 1 — License (or confirm the trial)

Open **System → Product Licensing** and confirm either an active license or a trial banner ("*A trial license is currently in effect until …*"). If you have a license, activate it now (Chapter 4.4 online, 4.5 offline).

✓ **Verify:** the licensing page shows a Valid license (or a trial with days remaining).

Step 2 — Database Connection

Configuration → Database Connections → Create → Microsoft SQL Server. Name it something you'll recognize in dropdowns, e.g. `MES-Production`. Enter the host, database, and login, then press **Test Connection**.

✓ **Verify:** you see *Connection Successful* before you save. (See Chapter 5.2.)

Step 3 — Processor

Configuration → Processors → Create → Rockwell Logix Controller. Name it `Line1CLX`, choose the model, and enter the IP address (and the slot or remote path as needed).

✓ **Verify:** the processor saves and appears in the index. (Full reachability is confirmed at start-up in step 8.)

Step 4 — Trigger Template

Configuration → Triggers → Create → Tag (Data Block) Based Trigger. Name it `RisingEdge-100ms`, set **Scan Interval** to 100 ms, and check **Rising Edge Only** (the classic handshake pattern: fire once when the PLC sets the bit, not again when it clears). Leave the change thresholds and quiet time at their defaults. (See Chapter 7.3.)

✓ **Verify:** the template appears in the Triggers index.

Step 5 — UDT (structure definition)

In Studio 5000, export the part-complete UDT to a `.L5X` file. Then in Event Engine: **Configuration → UDTs → Import**, choose **Logix (.l5x)**, select the file, and give it a **Namespace** like `Line1`. (See Chapter 8.1.)

✓ **Verify:** the UDT appears in **Configuration → UDTs** under your namespace, with the expected members. *(First-time imports load immediately; a re-import of an existing namespace requires a system reboot.)*

Step 6 — Tags

Configuration → Tags → Create. Select the `Line1CLX` processor first, then **Browse** the live controller and pick the part-complete structure tag — its data type fills in as your imported UDT. Saving a structure tag builds the whole member tree. On the structure's trigger member (e.g. `PartComplete.Trigger`), open it and assign the `RisingEdge-100ms` **Trigger Template** so it becomes a *trigger tag*. (See Chapter 9.2–9.3.)

✓ **Verify:** drilling into the structure tag shows its members (serial, result, completion code, status message), and the trigger member shows the trigger template assigned.

Step 7 — The Event

Events → Stored Procedure Events → Create.

1. **Details:** name it `LogPartComplete`, pick the `MES-Production` connection, choose the stored procedure, and select the part-complete structure tag as the **Master Tag**.
2. **Parameters:** press **Refresh** so Event Engine reads the procedure's parameters and best-matches them to the master tag's members by name. Fix any unmatched rows (tag, constant, function, or `-- NULL --`).

3. **Event Control:** the **Trigger Tag(s)**, **Completion Tag(s)**, and **Status Message Tag(s)** dropdowns are already filtered to the master tag's members — select the trigger, completion, and message members.
4. **Event Options:** leave **Timeout** above your worst-case procedure time, keep **Enabled** on, and **Save**. (See Chapter 10.2.)

✓ **Verify:** the event appears in **Events** → **All Events** with no validation errors.

Step 8 — Start and Prove It

System → **System Status** → **Start.** Wait for the light to turn 🟢 green (it passes through amber while it connects to processors and warms up tag groups).

Then open the event's **Telemetry** page (**Events** → **All Events** → **Telemetry**) and press **Manual Trigger**. This runs the *real* event — the procedure executes and the completion code is written to the PLC — so confirm the dialog only when you intend that. Watch the three-phase timing (*Tag Collection* → *Work* → *Write Back*) and the success counter increment. (See Chapter 14.2.)

Event Telemetry

OP200 - Add Product

Nominal



Event Telemetry

Scanning

Total Event Calls: 2	Failed Event Calls: 0
Successful Calls: 2	Success Rate (% Passing Events): 100
Minimum Duration (msec.): 25.8	Maximum Duration (msec.): 109.94
Last Call Duration (msec.): 25.8	Average Duration (msec.): 67.87
Last Tag Collection Time (msec.): 14.92	Average Tag Collection Time (msec.): 51.47
Last Work Time (msec.): 5.48	Average Work Time (msec.): 7.93
Last Write Back Time (msec.): 5.11	Average Write Back Time (msec.): 8.18
Time of Last Call: 09/02/2026 04:00:29	Last Count Reset:

Error List

No Current Errors

Back to Index | Details | Edit | Delete | Reset Counts | Manual Trigger | Event History | Health

Copyright © 2026 Factory Data Systems, LLC

✓ **Verify:** the event records a successful call, the row appears in your database, and the PLC's completion tag reads **1**. Your first event works end-to-end. 🎉

0.4 After Your First Event — Scaling Out

Once one event works, the rest is repetition and breadth:

- **Roll it out across identical stations.** Use **Duplicate** on the event (Chapter 10.3): point the copy at the next station's master-tag structure and Event Engine re-wires the bindings automatically. Pre-assigning member **Functions** in a Collection (Chapter 8.2) makes every new station's tag come out event-ready.
- **Add the other event types** as needed: print a label after the log completes (Chapter 12), read a recipe row back to the PLC (Chapter 11), or email a fault summary (Chapter 13 — configure the SMTP relay in Chapter 3.3 first).
- **Chain events** with **Cascaded Events** — e.g., fire a label print only after the database log succeeds.
- **Back up the configuration** (Chapter 3.5) now that you have a known-good baseline, and again before any large change.

0.5 Common First-Time Pitfalls

Symptom	Likely cause	Where to look
Start button is disabled	No valid license and no active trial, or the enabled-event count exceeds the licensed quota	Hover the button for the reason; Chapter 4
A procedure / table / column dropdown is empty	No database connection chosen yet, or the connection can't be reached	Re-select the connection; re-test it in Chapter 5
No tags to choose in the event editor	Tags weren't created, or none have the needed *Function* (trigger/completion/message)	Create/assign tags in Chapter 9; pre-assign functions via Chapter 8
The trigger dropdown is empty	No trigger template assigned to a tag (tag triggers) or to the event (time triggers)	Assign the template in Chapter 7.5
Event saved but never fires	System isn't scanning (light not green), event is Disabled, or the trigger condition is never met	Chapter 3.1; check the trigger rules in Chapter 7
Imported UDT changes don't appear	A re-import into an existing namespace needs a system reboot to take effect	Chapter 8.1
Email/notification options are greyed out	The SMTP relay isn't configured	Chapter 3.3

Symptom	Likely cause	Where to look
Label won't print	Printer name doesn't match an installed Windows printer exactly	Use the **Copy** button; Chapter 12

When an event *does* fire but fails, its **Telemetry** page error list almost always names the exact problem — and the data-flow troubleshooting map in Chapter 2.6 tells you which phase (PLC read, work, or write-back) to investigate.

Next: continue to Chapter 1 — System Overview for the concepts behind each step, or jump straight to the chapter for the step you're on using the table of contents in the guide home.

Chapter 1 — System Overview

What is Event Engine?

Event Engine™ is industrial middleware from Factory Data Systems, LLC. It connects your plant-floor control systems (PLCs) to your business systems — databases, label printers, and email — and executes **events** between them in real time.

An *event* is a unit of work that Event Engine performs when something happens — for example:

- A PLC sets a "part complete" bit → Event Engine reads the part's data structure from the controller, calls a stored procedure in your database, and writes a completion code back to the PLC.
- A pallet arrives at a label station → Event Engine reads the pallet data and prints a ZPL label on the correct Zebra printer.
- A machine faults → Event Engine emails the maintenance team with live process values embedded in the message.
- A scheduled time arrives → Event Engine runs a database job every shift, day, or hour.

Everything is configured from a built-in web user interface — no programming is required to build, monitor, and maintain events.

EVENT ENGINE™

Event Execution & Management for Manufacturing Systems

Eventing with the FDS *Event Engine*

Real-Time eventing from all of your manufacturing processes.

Enterprise wide visibility to manage your *entire* factory.

Web enabled with mobility in mind makes your data always available.

Quick Reference Links

Dashboard
Get a quick look at the overall health and performance of the system.

Errors
Review the most recent errors caught by the system so you can rectify them.

Copyright © 2026 Factory Data Systems, LLC

Why Event Engine is Different from Traditional Middleware

1. Significantly Faster

Event Engine was built because time is money on a machine cycle. It reaches the controller through a **native driver** and the database through a native client — no OPC server and no ODBC layer in the path — and each event carries a whole structure in one exchange instead of a tag at a time. On a recipe event of roughly 850 tags of mixed data types against a Rockwell Logix controller, that runs **40× faster than the same event over OPC on the same hardware**, and the more a single event carries, the wider the gap. OPC UA is still there when a controller or gateway needs it (Chapter 6) — the margin above is what a native driver buys you where one exists. Several design decisions contribute to this:

- **Whole-structure reads and writes.** Instead of reading dozens of individual tags one at a time, Event Engine reads an entire data structure from the controller in as few operations as the protocol allows. The telemetry pages break this out for you (*Tag Collection Time*, *Work Time*, and *Write Back Time*) so you can see exactly where each millisecond goes.
- **Single Write optimization.** Each event can consolidate its payload writes with its completion-code write into a single write operation ("Use Single Write" in the event options), removing an entire round trip to the PLC.
- **Database connection pooling.** Each database connection keeps a warm pool of its own — by default 10 connections held open and at most 20 concurrent — so an executing event borrows a ready connection instead of paying for a physical login (Chapter 5).
- **Asynchronous event execution.** Events are dispatched on an internal event bus and execute concurrently, so one slow database call does not hold up the rest of the system.

2. Transfer of Entire Data Structures

Traditional middleware forces you to map one tag to one database column, one at a time. Event Engine works at the *structure* level:

- A single **Master Tag** — a structure tag in the controller — can serve as the foundation for an entire event. Its members are automatically matched to stored procedure parameters or table columns by name.
- Result sets returned from the database can be written back into entire tag structures in the PLC, not just scalar values.

This means the same UDT you engineered in the PLC program becomes the unit of data exchange with your database, with no flattening or re-mapping effort.

3. Breaking the Tag Boundary — UDTs Imported from the PLC

Event Engine does not limit you to atomic tags:

- **UDTs (User Defined Types)** can be imported directly from controller export files — `.L5X` files for Rockwell Logix controllers, and `.scl`, `.stl`, `.db`, or `.udt` sources for Siemens S7 controllers. The imported definitions are compiled and organized into *namespaces* so types from different programs never collide.
- **Collections** let you define your own reusable structures right in the web UI when an import isn't appropriate.
- Tags can then be created as instances of these structures — including **arrays of structures** — and a single structure tag carries trigger members, payload members, completion-code members, and status-message members all at once.

See Chapter 8 — UDTs & Collections for details.

4. Multiple Event Types

Beyond classic stored-procedure execution, Event Engine supports:

Event Type	What it does	Chapter
Stored Procedure	Executes a database stored procedure, passing tag values in and writing outputs/result sets back to the PLC	10
Database Table	Reads from or inserts into database tables and views directly — no stored procedure required (single-row read, multi-row/result-set read, and insert)	11
ZPL Label	Sends a parameterized ZPL command string to any installed or network printer — printer selection can even come from a tag value	12
Message Notification	Sends a parameterized SMTP email message to the recipients of your choice	13

Database connectivity itself is plugin-based, with connectors for **Microsoft SQL Server, PostgreSQL, MySQL, and Oracle**.

5. Advanced Telemetry

Event Engine continuously measures itself and shows you the results:

- A **system dashboard** with run state, total event calls, events and errors in the past hour, and a rolling error list.
- A **per-event telemetry page** with success rate, minimum / maximum / last / average durations, and a unique three-phase breakdown of each event: *tag collection time* (reading the PLC), *work time* (the database call, print job, or message), and *write-back time* (returning results and completion codes to the PLC).
- **Reset counters** and **manual trigger** functions for commissioning and troubleshooting.

See Chapter 14.

6. User-Friendly Web UI

The entire product is configured and monitored through a browser:

- Web-enabled with mobility in mind — your data is always available, plant-wide.
- Role-based security based on Windows / Active Directory groups: a **General** role for viewing and operating, and an **Admins** role for configuration changes.
- Fully translated into **Spanish (es-MX)** and **German (de-DE)** as well as English, chosen from the flag icons in the navigation bar and remembered per Windows user (Chapter 3.4).
- Built-in searching, sorting, and paging on every list page — with **live updates**: statistics, status, and errors stream to the open page as they happen, with no page refresh required.
- Import/export of configuration items as JSON files and full configuration backup/restore as a ZIP archive, so configurations can be moved between systems.

System Architecture at a Glance

> 📊 For a diagrammed walk-through of how data moves between the PLC, Event Engine, and your business systems during an event, see Chapter 2 — Understanding Event Data Flow.

- **Device plugins** provide the PLC protocols: Rockwell Logix, Allen-Bradley Legacy, Modbus/TCP, GE SRTP, Siemens S7, and OPC UA. (Chapter 6)
- **Subscriber plugins** provide the event types: Stored Procedure, Database Table, ZPL Print, and Message. (Chapters 10–13)
- **Database connector plugins** provide client database access: SQL Server, PostgreSQL, MySQL, and Oracle. (Chapter 5)
- **Configuration** is stored as JSON files on disk (by default under `C:\ProgramData\Factory Data Systems, LLC\Event Engine\Configuration`), which makes backup, restore, and migration straightforward. (Chapter 3)

Key Terms Used Throughout This Guide

Term	Meaning
Processor	A configured connection to a PLC or controller (name, protocol, IP address, etc.).
Tag	A named piece of data in a processor — atomic value, array, or structure.
Trigger Template	A reusable definition of *when* something should happen (tag-change based or time based).
Trigger Tag	A tag assigned a trigger template; when its condition is met, events fire.
Master Tag	A structure tag chosen as the foundation of an event; its members are matched to event inputs/outputs.

Term	Meaning
Completion Tag	An integer tag the event writes a completion code into (0 = in-process, 1 = success, codes less than 0 = error code).
Status Message Tag	A string tag the event writes status/error text into for the controller's benefit.
Payload Tag	A tag used purely to carry data in an event, with no control responsibilities.
Event	A configured action (stored procedure, table access, label print, or message) executed when triggered.
Cascaded Event	An event fired automatically upon the *successful* completion of another event.

Next: Chapter 2 — Understanding Event Data Flow · Guide home

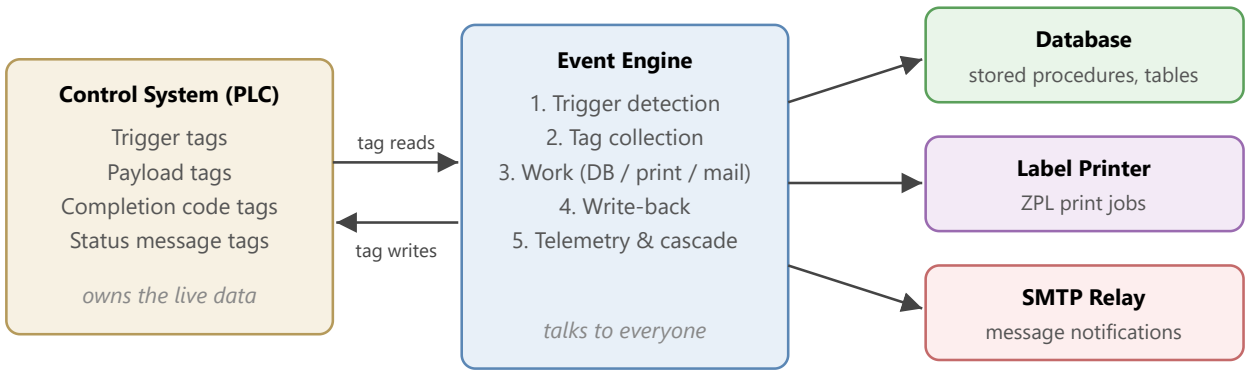
Chapter 2 — Understanding Event Data Flow

This chapter walks through *how data actually moves* when an event runs — which part of the system (PLC, Event Engine, database, printer, mail relay) does what, and in what order. Understanding the flow makes every configuration choice in Chapters 7–14 easier to reason about: you'll know exactly which phase a setting affects and where to look when something is slow or failing.

2.1 The Big Picture

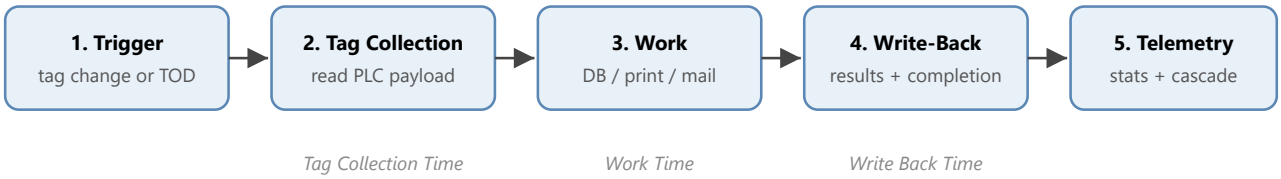
Three parties participate in every event:

Party	Role in an event
The control system (PLC)	Owns the live data. It *initiates* tag-triggered events (by changing a trigger tag), *supplies* the payload values, and *receives* the results: output data, a completion code, and a status message. The PLC never talks to the database, printer, or mail relay — Event Engine does that on its behalf.
Event Engine	The middleman and the only party that talks to everyone. It watches trigger tags on a scan interval, collects payload data from the PLC, performs the event's work against the business system, writes results back to the PLC, and records telemetry for every run.
The business system	The database (stored procedure / table events), the label printer (ZPL events), or the SMTP relay and recipients (message events). It performs the actual business operation and, for database events, may return data.



The Five Phases of Every Event

Regardless of type, every event execution moves through the same five phases. Event Engine times the middle three individually — they are exactly the *Tag Collection Time*, *Work Time*, and *Write Back Time* metrics on the telemetry pages (Chapter 14).



Phase 1 — Trigger. For *tag triggers*, Event Engine reads the trigger tag from the PLC at the template's scan interval and evaluates its rules (rising edge, change thresholds, quiet time — see Chapter 7). For *time-based triggers*, Event Engine's own scheduler fires at the configured time of day / repeat interval — no PLC involvement at all. When a trigger fires, the event's completion tag is set to **0 (In-Process)** so the PLC knows work has begun.

Phase 2 — Tag Collection. Event Engine reads every PLC tag the event needs — stored procedure parameter tags, table column tags, label field tags, message field tags — consolidating them into as few protocol reads as possible. A Master Tag structure pays off here: one structure read can collect the entire payload.

Phase 3 — Work. The event-type-specific operation runs against the business system. This is the only phase that differs between event types — see 2.2 through 2.5.

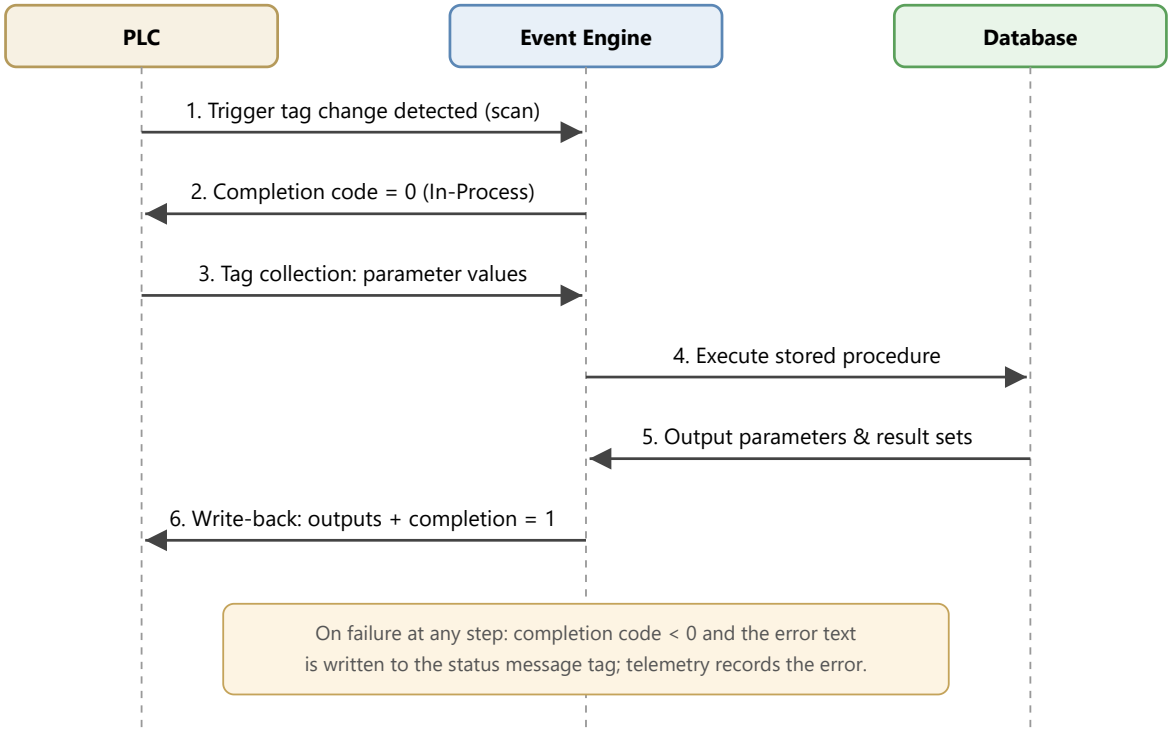
Phase 4 — Write-Back. Results return to the PLC: output parameters and result sets (database events), then the **completion code** — **1** for success, an error code (**< 0**) for failure — and any **status message** text. With *Use Single Write* enabled, the payload write-back and the completion write are combined into one operation.

Phase 5 — Telemetry & Cascade. The run's timings and outcome are recorded (visible immediately on the dashboard and the event's Telemetry page). On failure, a notification email is sent if configured. On success, any **Cascaded Events** are fired next, each starting its own five-phase cycle.

> **The PLC-side handshake pattern:** the controller program sets its trigger tag, then watches the completion tag: `0` = Event Engine is working, `1` = done (results are valid), `< 0` = failed (read the status message tag for the reason). The PLC should not re-trigger until it sees a final code — unless the event allows overlapping executions.

2.2 Stored Procedure Event Flow

The PLC supplies parameter values; the database executes the procedure; outputs flow back to the PLC. (Configuration: Chapter 10.)



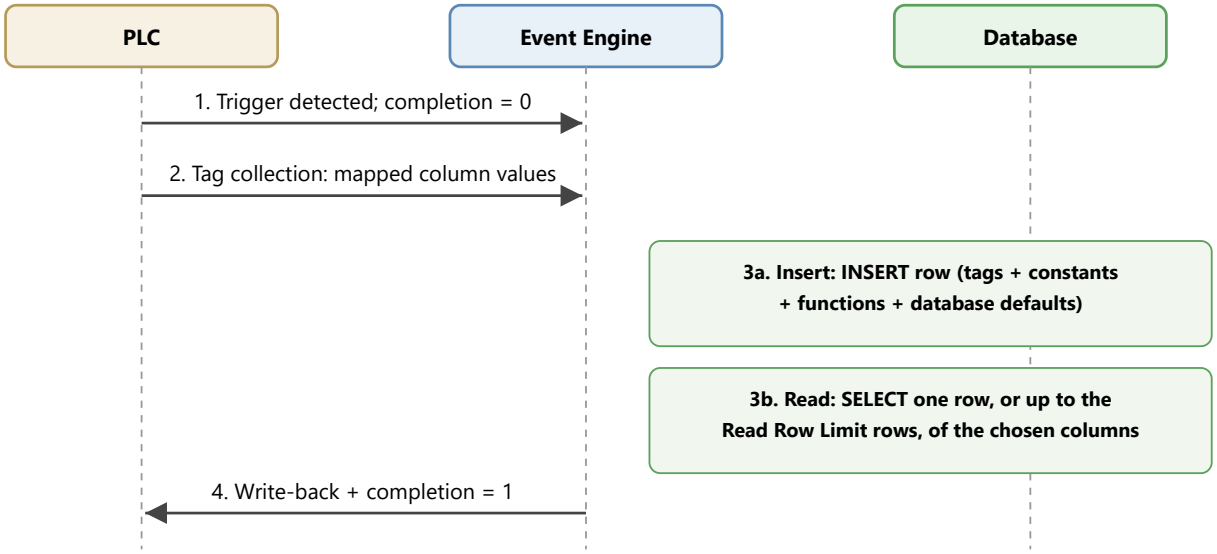
Step-by-step, with the responsible party:

1. **PLC** sets the trigger tag; **Event Engine** detects it on the next scan of the trigger template's interval. (Or Event Engine's scheduler fires a time-based trigger.)
2. **Event Engine** → **PLC:** completion tag is written to 0 (In-Process).
3. **Event Engine** ← **PLC:** all parameter tags are read (tag collection). `-- NULL --` parameters are skipped, constants and functions are filled in by Event Engine itself.
4. **Event Engine** → **Database:** the stored procedure executes on the configured connection, with the per-event Timeout bounding the call.
5. **Database** → **Event Engine:** output parameter values and any result sets come back.
6. **Event Engine** → **PLC:** output values are written to their matched tags, result sets to their target structures, then completion = 1 (combined into one write when *Use Single Write* is on).

> **Important nuance:** the database call (step 4) may *commit* even if a later write-back step fails — the PLC sees an error code, but the row may exist in the database. Design procedures to be idempotent (safe to re-run) where the PLC may retry.

2.3 Database Table Event Flow

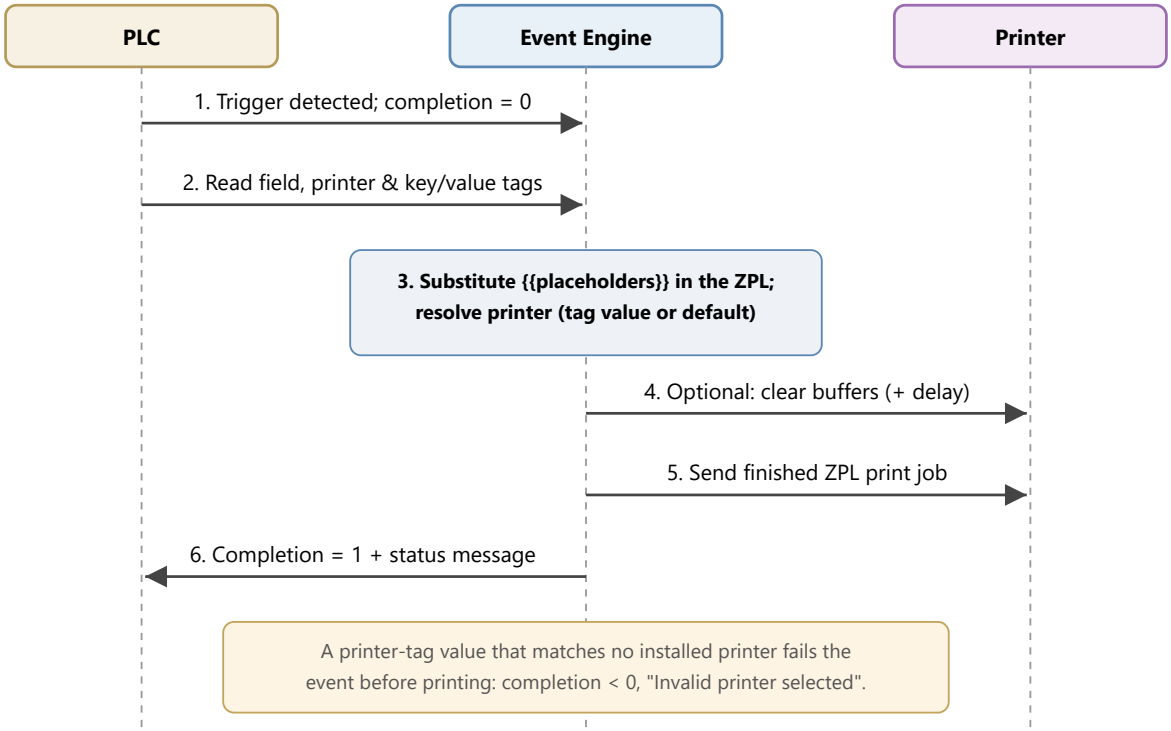
Table events skip the stored procedure: Event Engine itself composes the SQL for the chosen table or view. (Configuration: Chapter 11.) The flow matches 2.2 with a different work phase, and the direction of data depends on the operation:



- **Insert** (PLC → Database): the collected tag values — plus any constants, Date/Time function values, and database defaults — become one new row. Nothing data-wise returns to the PLC; the write-back phase carries only the completion code and status message.
- **Single Row Read** (Database → PLC): one row is selected and each chosen column's value is written to its individual target tag during write-back.
- **Multi-Row Read / Result Set** (Database → PLC): up to *Read Row Limit* rows are selected and written into the target tag during write-back. The target can be any tag whose shape holds the rows — a structure or array of structures, a tag collection, an array, or a single atomic tag. **The whole target is written every time:** elements the read did not fill are set to their default values rather than left holding the previous read's rows. The bigger the row limit and target, the longer the write-back phase — watch *Write Back Time* on the telemetry page when sizing these.

2.4 ZPL Label Print Event Flow

The PLC supplies field values (and optionally the printer name); Event Engine builds the final ZPL string and delivers it to the Windows print path. The printer is the business system — no database is involved. (Configuration: Chapter 12.)

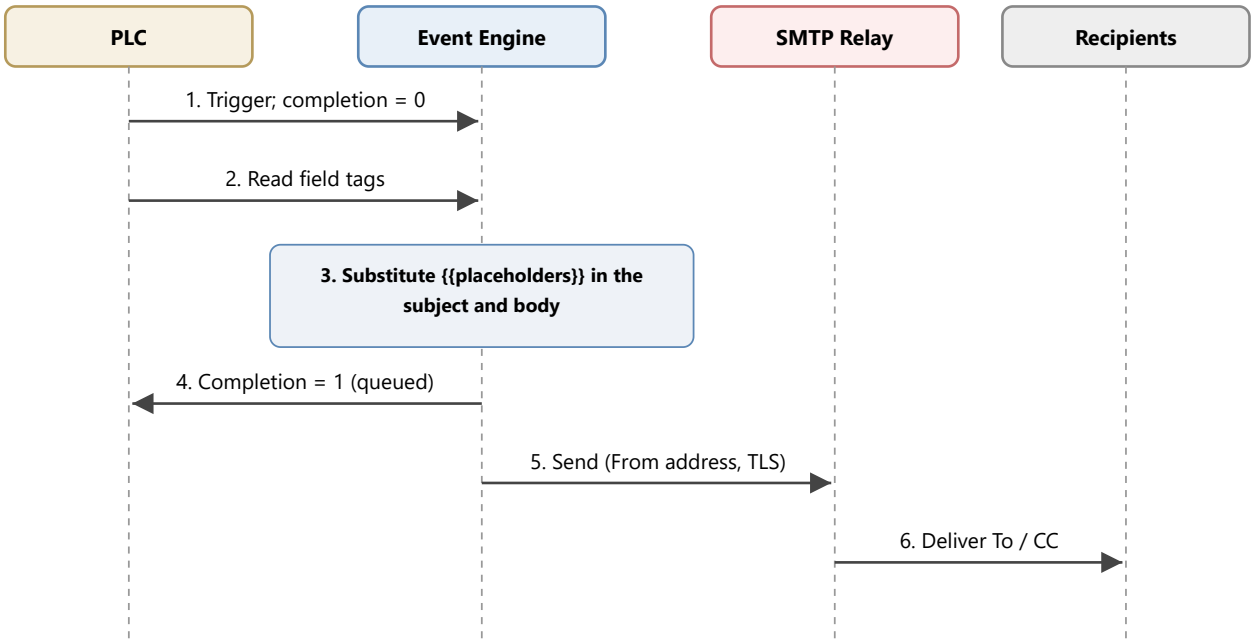


Notes on the work phase:

- Field values come from the sources configured per placeholder: a tag, the key/value tag lookup, a Date/Time function, or a constant (Chapter 12.2).
- "Success" means the job was handed to the resolved printer — Event Engine cannot see media-out or ribbon errors inside the printer, so pair critical print stations with a physical applied-label check where needed.

2.5 Message Notification Event Flow

The PLC supplies the live values; Event Engine builds the email and hands it to your SMTP relay, which delivers it to the recipients. (Configuration: Chapter 13; the relay itself is configured once in **System → Settings**, Chapter 3.3.)



Notes on the work phase:

- Event Engine's "success" (step 4) is reported **at queueing**: completion = 1 is written once the message is built and accepted into the send queue — before the relay is contacted. The hand-off to the relay (step 5) and delivery (step 6) happen asynchronously; a failure there — including no relay being configured at all — marks the run **failed** in the event's history and telemetry but does not rewrite the completion code. A passing completion code with no email in the inbox therefore points at the send path or the mail system, and the failed run naming the reason is in Event History (Chapter 13.5).
- Time-triggered messages (e.g., a shift summary) follow the same flow; only step 1 differs — the trigger comes from Event Engine's scheduler instead of a PLC tag.

2.6 Watching the Flow in Practice

The telemetry system measures the exact phases described above, so the flow model doubles as a troubleshooting map:

Symptom on the Telemetry page	Phase	Where to look
High **Tag Collection Time**	Phase 2	PLC comms: scan loads, network, oversized payload structures, or other events queuing behind this one on the same processor (Ch. 6.3)
High **Work Time**	Phase 3	The business system: slow procedure/query, printer spooler, SMTP relay

Symptom on the Telemetry page	Phase	Where to look
High **Write Back Time**	Phase 4	Large result sets/structures returning to the PLC; consider *Use Single Write*
Failures with completion < 0	Any	The event's error list names the failing step; the status message tag carries the same text to the PLC

For commissioning, the **Manual Trigger** function (Chapter 14.2) runs the full five-phase cycle on demand so you can watch each timing move as you tune.

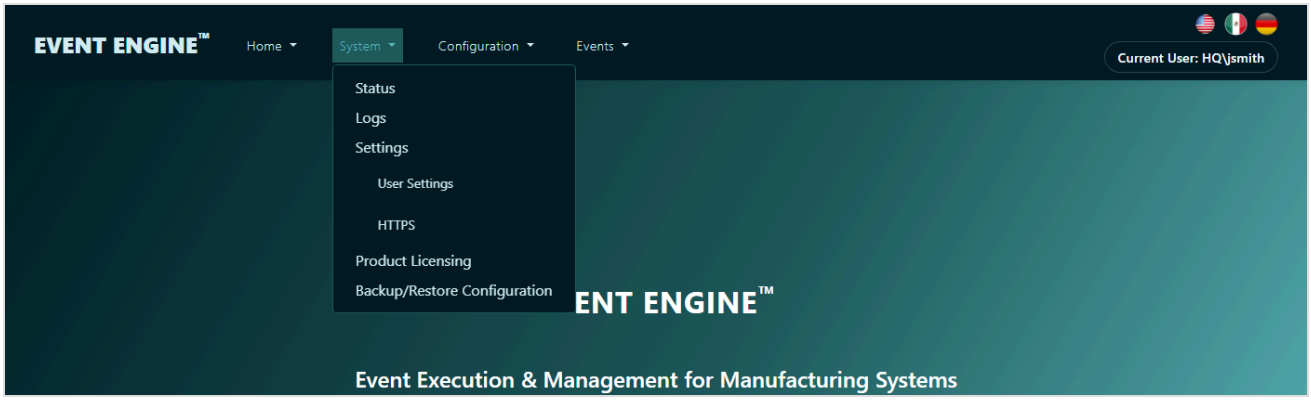
Next: Chapter 3 — The System Menu · Guide home

Chapter 3 — The System Menu

The **System** menu is the operational heart of Event Engine. From here you start and stop the system, review logs, change system-wide settings, set your personal preferences, manage licensing (covered in Chapter 4), and back up or restore the configuration.

The System menu contains:

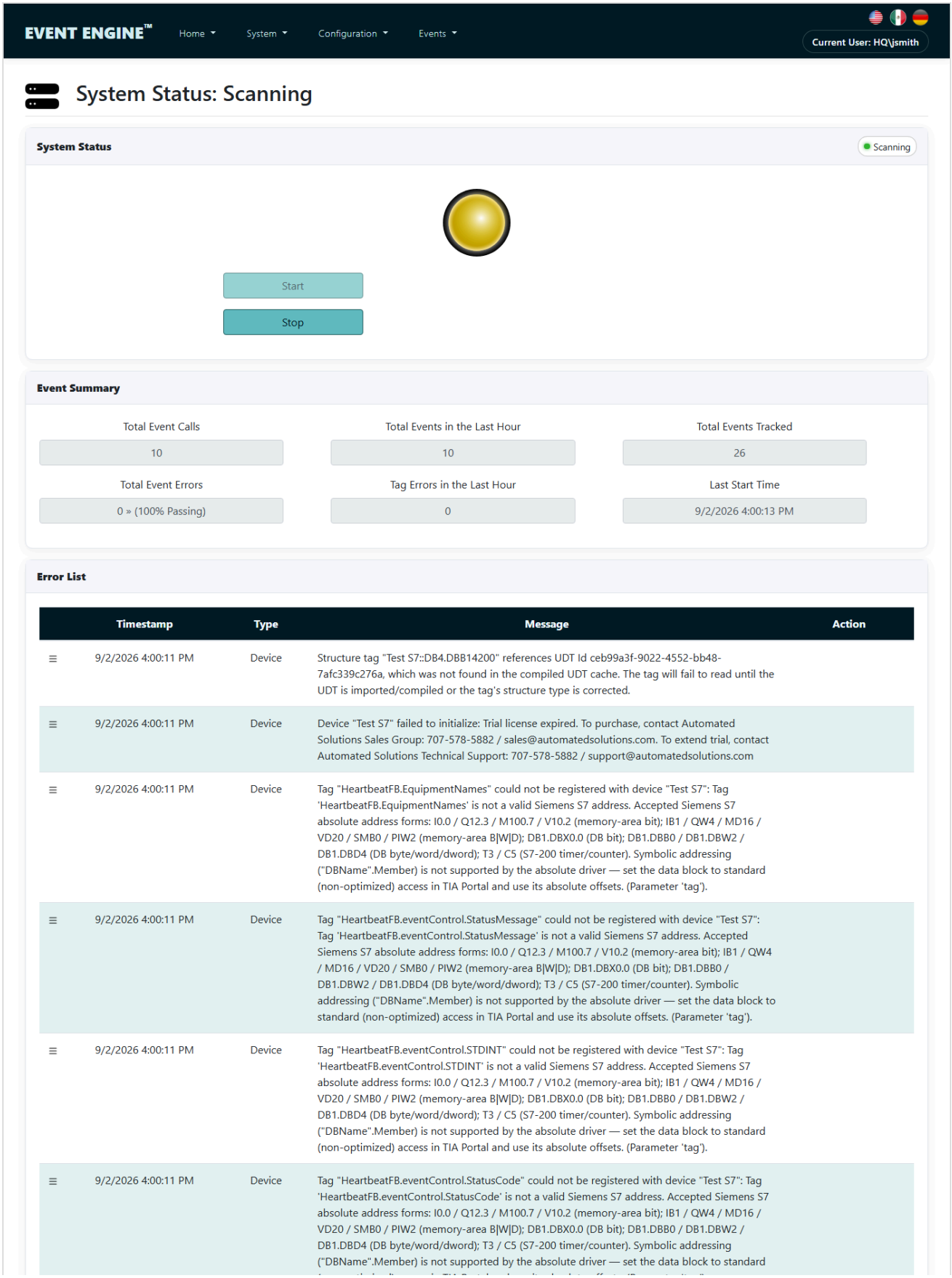
- **System Status** — the dashboard and the Start/Stop control
- **Logs** — log file review and the recent error list
- **Settings** — system-wide configuration (logging, telemetry queue, SMTP, event health, security & session)
- **HTTPS** — the site's certificate and the HTTP → HTTPS redirect (see 3.6)
- **User Settings** — your personal preferences (page size)
- **Product Licensing** — see Chapter 4
- **Backup/Restore Configuration**

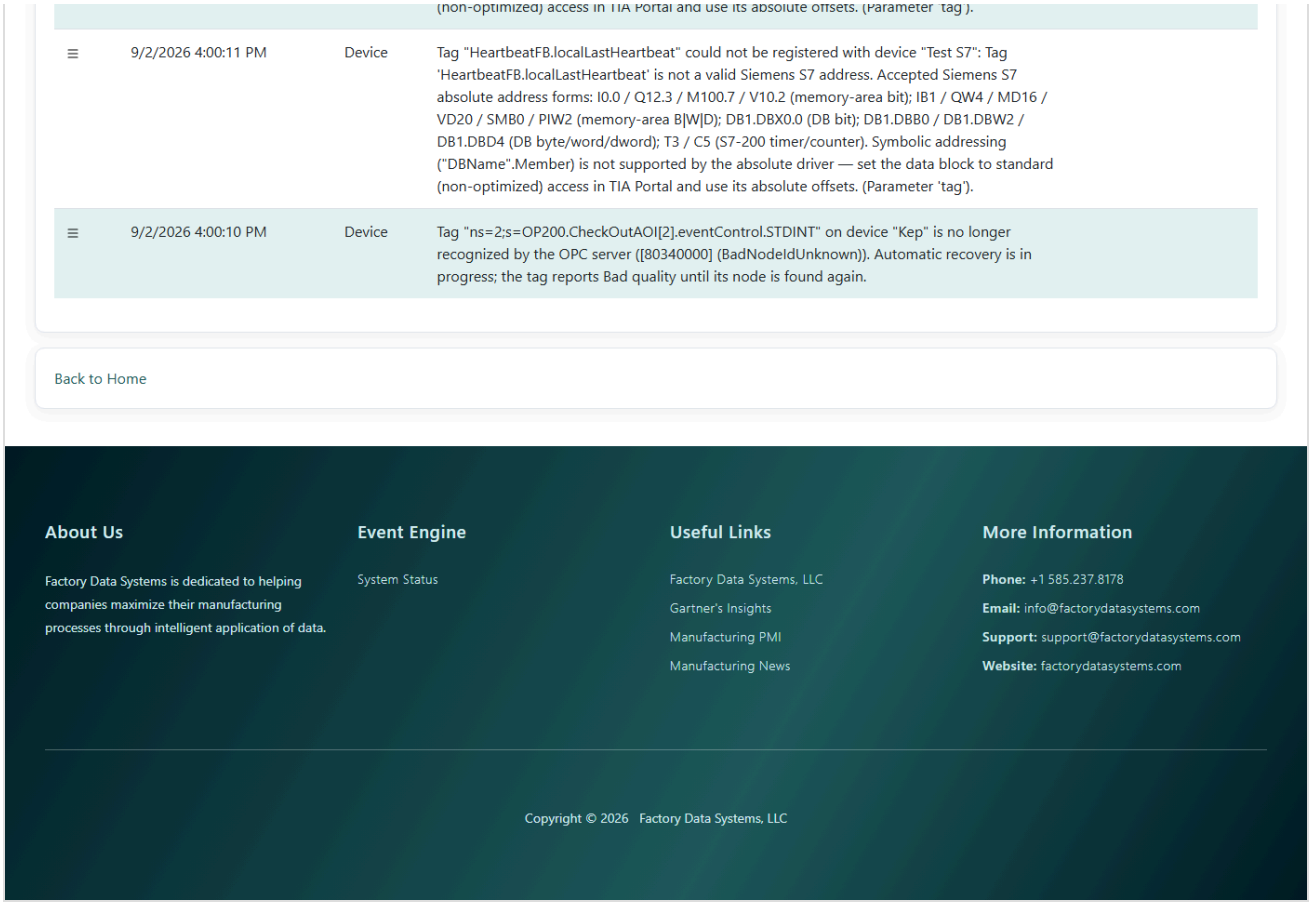


3.1 The System Status Page (Dashboard)

System → System Status

This page answers the most important question — *is the system running?* — and gives a quick overview of health and performance.


EVENT ENGINE™
Home
System
Configuration
Events
Current User: HQ\jsmith
System Status: Scanning
System Status
Scanning
Start
Stop
Event Summary
Total Event Calls
10
Total Events in the Last Hour
10
Total Events Tracked
26
Total Event Errors
0 » (100% Passing)
Tag Errors in the Last Hour
0
Last Start Time
9/2/2026 4:00:13 PM
Error List
Timestamp
Type
Message
Action
9/2/2026 4:00:11 PM
Device
Structure tag "Test S7::DB4.DBB14200" references UDT Id ceb99a3f-9022-4552-bb48-7afc339c276a, which was not found in the compiled UDT cache. The tag will fail to read until the UDT is imported/compiled or the tag's structure type is corrected.
9/2/2026 4:00:11 PM
Device
Device "Test S7" failed to initialize: Trial license expired. To purchase, contact Automated Solutions Sales Group: 707-578-5882 / sales@automatedsolutions.com. To extend trial, contact Automated Solutions Technical Support: 707-578-5882 / support@automatedsolutions.com
9/2/2026 4:00:11 PM
Device
Tag "HeartbeatFB.EquipmentNames" could not be registered with device "Test S7": Tag 'HeartbeatFB.EquipmentNames' is not a valid Siemens S7 address. Accepted Siemens S7 absolute address forms: I0.0 / Q12.3 / M100.7 / V10.2 (memory-area bit); IB1 / QW4 / MD16 / VD20 / SMB0 / PIW2 (memory-area B|W|D); DB1.DBX0.0 (DB bit); DB1.DBB0 / DB1.DBW2 / DB1.DBD4 (DB byte/word/dword); T3 / C5 (S7-200 timer/counter). Symbolic addressing ("DBName".Member) is not supported by the absolute driver — set the data block to standard (non-optimized) access in TIA Portal and use its absolute offsets. (Parameter 'tag').
9/2/2026 4:00:11 PM
Device
Tag "HeartbeatFB.eventControl.StatusMessage" could not be registered with device "Test S7": Tag 'HeartbeatFB.eventControl.StatusMessage' is not a valid Siemens S7 address. Accepted Siemens S7 absolute address forms: I0.0 / Q12.3 / M100.7 / V10.2 (memory-area bit); IB1 / QW4 / MD16 / VD20 / SMB0 / PIW2 (memory-area B|W|D); DB1.DBX0.0 (DB bit); DB1.DBB0 / DB1.DBW2 / DB1.DBD4 (DB byte/word/dword); T3 / C5 (S7-200 timer/counter). Symbolic addressing ("DBName".Member) is not supported by the absolute driver — set the data block to standard (non-optimized) access in TIA Portal and use its absolute offsets. (Parameter 'tag').
9/2/2026 4:00:11 PM
Device
Tag "HeartbeatFB.eventControl.STDINT" could not be registered with device "Test S7": Tag 'HeartbeatFB.eventControl.STDINT' is not a valid Siemens S7 address. Accepted Siemens S7 absolute address forms: I0.0 / Q12.3 / M100.7 / V10.2 (memory-area bit); IB1 / QW4 / MD16 / VD20 / SMB0 / PIW2 (memory-area B|W|D); DB1.DBX0.0 (DB bit); DB1.DBB0 / DB1.DBW2 / DB1.DBD4 (DB byte/word/dword); T3 / C5 (S7-200 timer/counter). Symbolic addressing ("DBName".Member) is not supported by the absolute driver — set the data block to standard (non-optimized) access in TIA Portal and use its absolute offsets. (Parameter 'tag').
9/2/2026 4:00:11 PM
Device
Tag "HeartbeatFB.eventControl.StatusCode" could not be registered with device "Test S7": Tag 'HeartbeatFB.eventControl.StatusCode' is not a valid Siemens S7 address. Accepted Siemens S7 absolute address forms: I0.0 / Q12.3 / M100.7 / V10.2 (memory-area bit); IB1 / QW4 / MD16 / VD20 / SMB0 / PIW2 (memory-area B|W|D); DB1.DBX0.0 (DB bit); DB1.DBB0 / DB1.DBW2 / DB1.DBD4 (DB byte/word/dword); T3 / C5 (S7-200 timer/counter). Symbolic addressing ("DBName".Member) is not supported by the absolute driver — set the data block to standard



The Status Light and Start/Stop Buttons

The colored indicator shows the current scan state:

The heading beside the light always names the exact state — *Scanning*, *Starting*, *Stopping*, *Idle*, *Off-line* or *Error*. The light itself is a **health** indicator and deliberately maps several states onto three colors:

Light	When	Meaning
🟢 Green	Scanning, no recent error	The system is running: tags are being scanned and events will fire.
🟡 Amber	Starting or Stopping — **or** scanning with an error in the last 2 minutes	A transition is in progress (starting connects to every processor and warms up tag groups, which can take a while on large systems; the heading shows "Please wait…"), *or* the system is running normally but something failed very recently. Check the Error List to tell the two apart.
🔴 Red	Idle, Off-line or **Error**	The system is stopped. No tags are scanned and **no events will fire**.

> **The `Error` state.** A start that fails leaves the system in *Error* rather than silently back at Idle, so a failed start is visible rather than looking like one that never happened. It is a **stopped** state: nothing is scanning. From here **Stop** acts as a reset — it clears the fault back to Idle — and **Start** is available to try again. The Error List names what failed.

The buttons follow the state, not the light: **Start** is available whenever the system is stopped (Idle, Off-line or Error), and **Stop** whenever it is Starting, Scanning, or in Error. Press either and confirm the dialog to change states. The request returns immediately and the (potentially long-running) startup or shutdown work completes in the background — the status light, heading, and buttons update live on the page as it proceeds.

> **Why might Start be disabled?** If the product has neither a valid license nor an active trial, or the number of enabled events exceeds the licensed quota, the Start button is disabled and a tooltip explains the licensing problem. Resolve the license (Chapter 4) or disable excess events, then return here. Processor protocols are not separately licensed and never block Start.

The Event Summary Card

Six live fields summarize event activity since the last counter reset:

Field	Meaning
Total Event Calls	Total number of event executions tracked by telemetry.
Total Events in the Last Hour	Event executions in the past 60 minutes.
Total Events Tracked	The number of *configured* events in the system.
Total Event Errors	Total failed event calls (with failure rate).
Tag Errors in the Last Hour	Failed event calls in the past 60 minutes. *(Despite the "Tag Errors" wording on screen, this counter reflects failed event calls — not tag-level read/write errors.)*
Last Start Time	When the system last started scanning — whether from automatic startup at application load or a user pressing Start. Blank if the system has not started since the application loaded.

The Error List

Below the summary is a paged list of the most recent errors captured by the telemetry service, newest first. For an event (action) failure, **Details** opens that event's Event History (Chapter 14.3) on the run that failed, with its row already expanded — the trigger that fired it, the phase timings, and the full failure

detail, all in place. Errors that are not event runs — device, system, and licensing — have no execution behind them, so the message in this list is the whole record and no link is offered.

While you are on the first page, new errors appear at the top of the list as they happen. On a deeper page, a badge reports how many new errors have arrived so the rows never shift underneath you — return to the first page to see them.

Live Updates

The dashboard updates itself in real time — the status light, the summary counters, and the error list all change on screen as events execute, without reloading the page. A small **Live** indicator (where applicable, in the corner of the status card) shows the connection state:

Indicator	Meaning
🟢 **Live**	Connected — values on the page update automatically as they change.
🟡 **Connecting…**	The live connection is being (re-)established. The page still works; values catch up automatically once connected.
🔴 **Off-line**	No live connection (for example, the service is restarting). The page keeps retrying in the background; a manual refresh always shows current values.

If the connection drops and reconnects, the page automatically resynchronizes to the current values — nothing that happened while disconnected is lost, because the display always reflects the server's current state. Hidden browser tabs release their connection after a minute and reconnect (and resynchronize) when you return to them.

3.2 Logs

System → Logs

Event Engine writes rolling log files using structured logging. The Logs page shows:

1. **The recent error list** — the same telemetry errors as the dashboard, for convenience (and like the dashboard's list, it updates live as errors occur).
2. **A log file selector** — all `*.log` files in the configured log folder, newest first. File names include the date (`*-YYYYMMDD.log`).

EVENT ENGINE™ Home System Configuration Events

Current User: HQ\jsmith

Logging Index

Select the Log File to Review (*-YYYYMMDD.log)

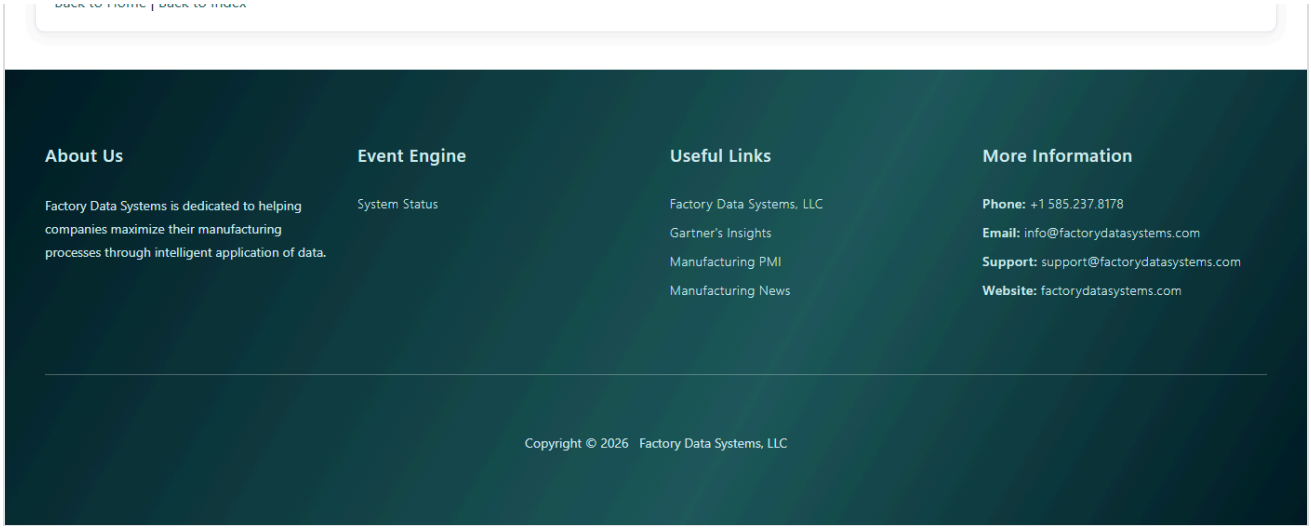
Select the Log File to Review (*-YYYYMMDD.log)

Event Engine-20260902.log

Select

Error List

	Timestamp	Type	Message	Action
≡	9/2/2026 4:00:11 PM	Device	Structure tag "Test S7::DB4.DBB14200" references UDT Id ceb99a3f-9022-4552-bb48-7afc339c276a, which was not found in the compiled UDT cache. The tag will fail to read until the UDT is imported/compiled or the tag's structure type is corrected.	
≡	9/2/2026 4:00:11 PM	Device	Device "Test S7" failed to initialize: Trial license expired. To purchase, contact Automated Solutions Sales Group: 707-578-5882 / sales@automatedsolutions.com. To extend trial, contact Automated Solutions Technical Support: 707-578-5882 / support@automatedsolutions.com	
≡	9/2/2026 4:00:11 PM	Device	Tag "HeartbeatFB.EquipmentNames" could not be registered with device "Test S7": Tag 'HeartbeatFB.EquipmentNames' is not a valid Siemens S7 address. Accepted Siemens S7 absolute address forms: I0.0 / Q12.3 / M100.7 / V10.2 (memory-area bit); IB1 / QW4 / MD16 / VD20 / SMB0 / PIW2 (memory-area B|W|D); DB1.DBX0.0 (DB bit); DB1.DBB0 / DB1.DBW2 / DB1.DBD4 (DB byte/word/dword); T3 / C5 (S7-200 timer/counter). Symbolic addressing ("DBName".Member) is not supported by the absolute driver — set the data block to standard (non-optimized) access in TIA Portal and use its absolute offsets. (Parameter 'tag').	
≡	9/2/2026 4:00:11 PM	Device	Tag "HeartbeatFB.eventControl.StatusMessage" could not be registered with device "Test S7": Tag 'HeartbeatFB.eventControl.StatusMessage' is not a valid Siemens S7 address. Accepted Siemens S7 absolute address forms: I0.0 / Q12.3 / M100.7 / V10.2 (memory-area bit); IB1 / QW4 / MD16 / VD20 / SMB0 / PIW2 (memory-area B|W|D); DB1.DBX0.0 (DB bit); DB1.DBB0 / DB1.DBW2 / DB1.DBD4 (DB byte/word/dword); T3 / C5 (S7-200 timer/counter). Symbolic addressing ("DBName".Member) is not supported by the absolute driver — set the data block to standard (non-optimized) access in TIA Portal and use its absolute offsets. (Parameter 'tag').	
≡	9/2/2026 4:00:11 PM	Device	Tag "HeartbeatFB.eventControl.STDINT" could not be registered with device "Test S7": Tag 'HeartbeatFB.eventControl.STDINT' is not a valid Siemens S7 address. Accepted Siemens S7 absolute address forms: I0.0 / Q12.3 / M100.7 / V10.2 (memory-area bit); IB1 / QW4 / MD16 / VD20 / SMB0 / PIW2 (memory-area B|W|D); DB1.DBX0.0 (DB bit); DB1.DBB0 / DB1.DBW2 / DB1.DBD4 (DB byte/word/dword); T3 / C5 (S7-200 timer/counter). Symbolic addressing ("DBName".Member) is not supported by the absolute driver — set the data block to standard (non-optimized) access in TIA Portal and use its absolute offsets. (Parameter 'tag').	
≡	9/2/2026 4:00:11 PM	Device	Tag "HeartbeatFB.eventControl.StatusCode" could not be registered with device "Test S7": Tag 'HeartbeatFB.eventControl.StatusCode' is not a valid Siemens S7 address. Accepted Siemens S7 absolute address forms: I0.0 / Q12.3 / M100.7 / V10.2 (memory-area bit); IB1 / QW4 / MD16 / VD20 / SMB0 / PIW2 (memory-area B|W|D); DB1.DBX0.0 (DB bit); DB1.DBB0 / DB1.DBW2 / DB1.DBD4 (DB byte/word/dword); T3 / C5 (S7-200 timer/counter). Symbolic addressing ("DBName".Member) is not supported by the absolute driver — set the data block to standard (non-optimized) access in TIA Portal and use its absolute offsets. (Parameter 'tag').	
≡	9/2/2026 4:00:11 PM	Device	Tag "HeartbeatFB.localLastHeartbeat" could not be registered with device "Test S7": Tag 'HeartbeatFB.localLastHeartbeat' is not a valid Siemens S7 address. Accepted Siemens S7 absolute address forms: I0.0 / Q12.3 / M100.7 / V10.2 (memory-area bit); IB1 / QW4 / MD16 / VD20 / SMB0 / PIW2 (memory-area B|W|D); DB1.DBX0.0 (DB bit); DB1.DBB0 / DB1.DBW2 / DB1.DBD4 (DB byte/word/dword); T3 / C5 (S7-200 timer/counter). Symbolic addressing ("DBName".Member) is not supported by the absolute driver — set the data block to standard (non-optimized) access in TIA Portal and use its absolute offsets. (Parameter 'tag').	
≡	9/2/2026 4:00:10 PM	Device	Tag "ns=2;s=OP200.CheckOutAOI[2].eventControl.STDINT" on device "Kep" is no longer recognized by the OPC server ([80340000] (BadNodeIdUnknown)). Automatic recovery is in progress; the tag reports Bad quality until its node is found again.	



Select a file and open it to view the **Log File Display**. The viewer loads the file from the *bottom up* — the newest entries appear first — and fetches older lines in chunks as you scroll, so even very large files open quickly. The **Export File** link downloads the raw log file so you can attach it to a support request.

3.2.1 Logging Levels

The minimum logging level is set in **System** → **Settings** (see 3.3). Levels, from most to least verbose:

Level	Use
Verbose	Everything, including per-validation and per-read detail. Extremely chatty — for short, focused troubleshooting sessions only. A Verbose selection is *not* persisted to the configuration file, so it automatically reverts to the saved level when the service restarts.
Debug	Detailed diagnostic flow (license checks, tag-load detail, etc.).
Information	Normal operational milestones — startup, shutdown, licenses loaded, configuration changes. This is the recommended day-to-day setting.
Warning	Unexpected but recoverable conditions (e.g., licensing server unreachable).
Error	Failures — failed events, failed license validation, exceptions.
Fatal	Catastrophic failures only.

> **Effect on the system:** the lower (more verbose) the level, the more messages are written and the more disk and CPU resources logging consumes. On a busy system, leaving Verbose or Debug enabled long-term can measurably affect performance and fill the log directory.

3.3 Settings (System Configuration)

System → **Settings** *(changing settings requires the Admins role)*

This page holds the system-wide settings, in four cards — **System Settings** and **Security & Session** on the left, **SMTP Relay Settings** and **Event Health** on the right, and one **Save** button for all four. Hovering over any field marked ⓘ in the UI shows the same explanation given below. Saving the page broadcasts a configuration-updated notification to the running system.

EVENT ENGINE™ Home System Configuration Events Current User: HQ\jsmith

Configuration Settings

System Settings

Minimum Logging Level
Verbose

Auto Start
True

Telemetry Queue Size
100

Log File Location
C:\ProgramData\Factory Data Systems, LLC\Event Engine\Log Browse...

Maximum Log File Size (MB)
25

Telemetry History Location
C:\ProgramData\Factory Data Systems, LLC\Event Engine\Log Browse...

Telemetry History Retention (days)
30

Telemetry History Buffer (records)
10000

Security & Session

Group changes take effect immediately. Session timeout changes take effect after the next service restart.

Administrators Group (Windows/AD)
× Administrators

General Users Group (Windows/AD)
× Authenticated Users

To use a group from another domain, type DOMAIN\Group — for example CORP\Domain Users. Type DOMAIN\ on its own to list that domain's groups.

Session Idle Timeout (minutes)
20

SMTP Relay Settings

SMTP Relay Address
smtp.example.com

SMTP Relay Port
587

SMTP User Name (if used)
eventengine@example.com

Password

From Address
no-reply@example.com

System Message Deliver Address
plant-alerts@example.com

☑ Use Encryption

Maximum Concurrent Sends
4

SMTP Operation Timeout (in seconds)
10

☐ Startup Message

☐ Shutdown Message

☐ Module Fault Message

Test Connection Send Test Message

Event Health

These settings are read when the service starts. Changes take effect after the next service restart.

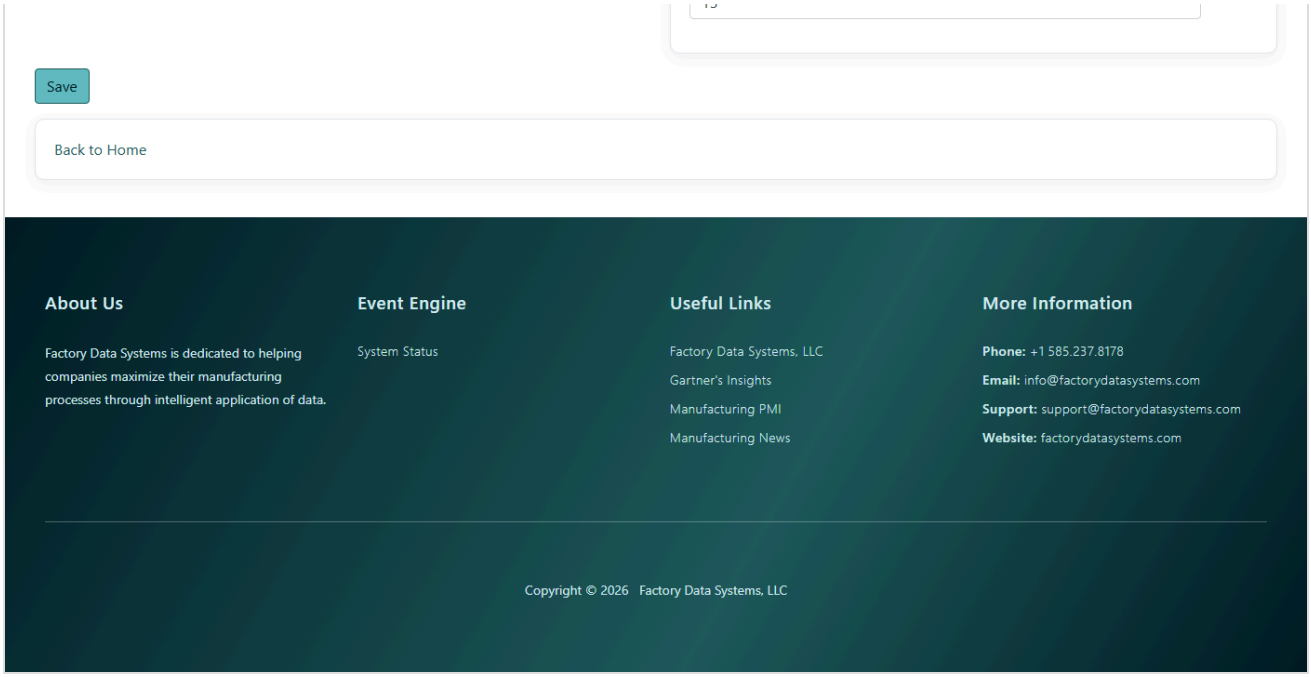
Monitor Event Health
True

Health Notifications
Daily digest

Digest Interval (hours)
24

Observer Poll Interval (seconds)
60

Observer State Save Interval (minutes)
15



System Settings

Setting	Default	What it does and how it affects operation
Minimum Logging Level	Information	The minimum severity written to the log file (see 3.2.1). Lower level = more messages = more resources consumed. Takes effect immediately. A *Verbose* selection is temporary and reverts at the next service restart.
Auto Start	true	If true, tag scanning starts automatically when the Event Engine service boots. Set to false on development systems where you want the engine idle until manually started.
Telemetry Queue Size	200	How many recent executions are kept **in memory** for each event — this feeds the live statistics and recent-runs lists on the Dashboard and the per-event Telemetry pages. This is separate from the on-disk telemetry history (the *Telemetry History* settings below); it does not change the CSV files or how many records survive there. Larger values show more recent history in the UI but use more memory. **A change does not take effect until the system is restarted** — the UI will flag that a restart is required.
Log File Location	`C:\ProgramData\Factory Data Systems, LLC\Event Engine\Logs`	The folder where log files are written. Make sure the service account can write here.

Setting	Default	What it does and how it affects operation
Maximum Log File Size (MB)	25	When the current log file reaches this size a new file is started (10 – 50). Larger files mean fewer files but slower viewing/downloading.
Telemetry File Location	*(blank)*	Where the daily telemetry history (CSV) files are written (Chapter 14.4). **Leave it blank to inherit** — the files then go to a `Telemetry` folder beside the log folder above, which is what most installations want. Set it only to put telemetry on different storage from the logs (for example, logs on the system disk and telemetry on a data volume). The service account must be able to write wherever it points. New files go to a changed location straight away, but Event Health keeps reading the previous location until the system is restarted — the UI will flag that a restart is required.
Telemetry History Retention (days)	30	How many days of daily telemetry history (CSV) files to keep — see Chapter 14.4. Files older than this are deleted automatically once per day. Takes effect without a restart.
Telemetry History Buffer (records)	10000	How many telemetry history records may wait to be written to disk when storage is briefly slow, before the oldest are dropped so event processing is never held up. Raise it if you use slow or network storage and want to tolerate longer hiccups; it uses a little more memory. **A change does not take effect until the system is restarted** — the UI will flag that a restart is required.

SMTP Relay Settings

These settings are **required if you use any messaging capability** — Message Notification events (Chapter 13), per-event *Notify on Failure* addresses, and the system start/stop/fault notifications below all send through this relay.

Setting	Default	What it does
SMTP Relay Address	*(empty)*	Host name or IP of your SMTP relay/smart host. **Required for every email feature** — Message events, per-event *Notify on Failure*, and the system notices below. While it is empty, the *Notify on Failure* controls on every event page are disabled, and a Message event behaves as in the warning below.
SMTP Relay Port	587	TCP port of the relay.

Setting	Default	What it does
SMTP User Name (if used)	*(empty)*	Not all SMTP relays require user name/password authentication — some use other forms of validation, so this may be blank.
Password	*(empty)*	The relay password. It is stored RSA-encrypted in the configuration; the field always displays blank, and leaving it blank on save keeps the existing password.
From Address	*(empty)*	The address notification messages are marked as being sent from. **Required for the system notices below** (see the warning after this table).
System Message Delivery Address	*(empty)*	Where *system-level* messages are sent — the startup, shutdown and module-fault notices below, **and license-expiry warnings**. One address, or several separated by commas or semicolons (`controls@example.com; maintenance@example.com`); every address listed receives each notice. Event-level message addresses are configured on each event, so this address matters only for notices the engine raises itself. **Required for those notices** (see the warning after this table).
Use Encryption	true	Uses an encrypted (TLS) connection to the relay.
Maximum Concurrent Sends	4	How many messages may be in flight to the relay at once. Each concurrent send opens its own connection, and **most hosted relays cap how many connections one account may hold open** — exceed that cap and the relay starts refusing sends. Raise it only if your relay documents a higher limit and a burst of events is queueing faster than it drains; lower it to 1 for a relay that permits a single connection.
SMTP Operation Timeout (seconds)	10	How long the system waits for the relay before treating the send as failed.
Startup Message	enabled	When checked, an email is sent to the system message address each time the system begins scanning.
Shutdown Message	enabled	When checked, an email is sent each time the system stops scanning and goes idle.
Module Fault Message	enabled	When checked, an email is sent when a system module faults.

> ⚠️ **System notices need three fields, not one — and say nothing when they are missing.** A *system* notice (startup, shutdown, module fault, and the license-expiry warning) is raised by the engine with no event behind it, so it needs all three of **SMTP Relay Address**, **From Address** and **System Message Delivery Address**. With any one of them blank, the three checkboxes above have no effect: the notice is skipped **silently**. There is no run to fail, no Error List entry, and no warning on this page.
>
> This is the easy configuration to get half-right. Setting only the relay is enough for Message events and *Notify on Failure* — which carry their own recipients — so messaging visibly works while every system notice is quietly discarded. **The one that costs you is the license-expiry warning:** it rides on the same three fields, and it is precisely the mail an unattended system needs, since without it the first sign of a lapsed license is scanning stopping (Chapter 4).
>
> If you want system notices, fill in all three and confirm with **Send Test Message** below — it sends from **From Address** to **System Message Delivery Address**, which is exactly the pair a system notice uses.

Testing the relay

Two buttons at the bottom of the card check the settings **as they appear on screen**. Neither requires you to save first, and neither writes anything — so you can try a relay, a port or a password and find out whether it works before committing to it. Both are available to administrators only.

Button	What it does
Test Connection	Connects to the relay, negotiates encryption if **Use Encryption** is checked, signs in if a user name is set, and hangs up. **No mail is sent.**
Send Test Message	Sends one real message, from **From Address** to **System Message Delivery Address**, through the same mail path every event uses.

Test Connection reports each of the three things it proved, because "it connected" on its own does not tell you whether the encryption setting or the credentials were part of it:

- *Connection successful (nnnms).* — the relay answered on that address and port.
- *The session was encrypted with TLS / was not encrypted, as configured* — follows the **Use Encryption** checkbox. A relay that offers no encryption on the port you gave it **fails** the test rather than quietly connecting without it, because every message would then fail the same way.
- *The relay accepted the user name and password / no user name is configured, so the connection was made anonymously / the relay offers no sign-in method this test can use*. The last one is not a failure: the connection is proven, but the relay authenticates by a method the test cannot speak (Windows integrated authentication, for example), so the credentials were left unchecked.

When either button fails, the relay's own words are shown underneath — "*535 Authentication credentials invalid*" is the difference between a password to fix and a network to investigate.

> **Note: Send Test Message** reports that the relay *accepted* the message. That is the last step the Event Engine can observe — the same limit a Message event reports under (Chapter 13.2). Check the destination mailbox to confirm the message was actually delivered.
>
> Port 465 (implicit TLS, encrypted from the first byte) is **not supported**; use the relay's STARTTLS port, which is 587 on most relays.

> ⚠️ **With no relay address, a message event's run is recorded as failed.** The event still builds its email and still writes **completion 1** to the controller, because success is reported at *queueing* (Chapter 13.2) — but the send then finds no relay, and that run is marked **Failed** in the event's telemetry and Event History, with the reason "*The message could not be sent because no SMTP relay address is configured.*" on the Error List.
>
> The completion code the controller already saw is **not** rewritten: it says the message was queued, which it was. So a passing handshake with no email in the inbox is exactly the case Event History exists to answer — the failed run is there with the reason attached. Configure the relay before enabling any messaging.

Event Health Settings

The **Event Health** card sits directly under the mail relay, because the one setting an operator usually comes here to change — whether findings are emailed — is useless without a relay to send them through. These settings govern Event Health; the defaults suit almost every installation, and none of them affect event execution.

> **These take effect as soon as you save — no restart.** The health observer picks up a change within seconds: switching it off stops it, switching it back on resumes where it left off, and a new poll interval, save interval, notification setting or digest interval applies from its next pass. The same is true of a change made by editing `appsettings.json` directly.
>
> Shortening the **Digest Interval** brings the next digest forward — it is sent no later than one new interval from the save. Lengthening it lets the digest already waiting go out when it was due; the ones after it use the new interval.

Setting	Default	What it does
Monitor Event Health	True	Whether the health observer runs at all. It reads the telemetry history files and adds nothing to event processing. Turning it off empties the Event Health page and the health badges until it is turned back on; nothing else changes, and the history it has gathered is kept.

Setting	Default	What it does
Health Notifications	Daily digest	How findings reach a mailbox: **Daily digest** (the day's changes in one message, but anything Critical straight away), **Immediate** (a message per change), or **Off** (pages only). Findings always appear on the Event Health page regardless. Uses the SMTP relay settings above — with no relay configured, nothing is sent.
Digest Interval (hours)	24	How long a digest message covers. Ignored unless *Health Notifications* is **Daily digest**.
Observer Poll Interval (seconds)	60	How long the observer waits between passes over the telemetry history files once it has caught up. A shorter interval notices a new execution sooner; it does not change how much is kept.
Observer State Save Interval (minutes)	15	How often the observer writes its resume file. A crash between saves costs only a re-read of the history since the last one, so a longer interval is cheap.

One health setting is deliberately not on this page. `RingCapacity` (default 2500) — how many recent executions are held per event for the health comparisons — lives only in `appsettings.json` under `ApplicationSettings:EventHealth:RingCapacity`. Raising it widens the comparison windows and uses more memory, but **changing it discards the saved health history and rebuilds it from the telemetry files**, which is why it is kept off a page an operator edits casually. Change it by editing that file and restarting the service.

> **Retention sets the ceiling.** On a fresh installation the health history can only be built from the telemetry files that still exist, so **Telemetry History Retention** above limits how much baseline is available at first. After that the reader keeps its own history, which outlives the files.

Security & Session

These settings control **who may use Event Engine** and **how long a sign-in stays valid**. Event Engine uses Windows Integrated Authentication — you are signed in as your Windows account — and authorizes that account by its **Windows or Active Directory group membership**. Before this card existed these values could only be set by hand-editing the configuration file; they are now managed here.

> **When changes take effect.** Changes to the **group mappings apply immediately** — the next request is evaluated against the new groups; no restart is needed. The **session idle timeout** is wired into the web pipeline when the service starts and cannot be hot-swapped, so saving a change to it flags that a restart is required.
>
> Changing a **user's membership in Active Directory** is a separate matter, and the two directions behave differently. *Adding* someone to a mapped group is normally picked up within a few minutes without them signing out. *Removing* someone is not: their Windows sign-in token still lists the group until they **sign out and back on**, so they keep the access until then. This is ordinary Windows behavior — the same reason a file share does not notice a group change immediately — but when you are revoking access deliberately, force a sign-out rather than assuming it is instant.

The two roles

Access is governed by two roles, each mapped to one or more groups:

Role	What members can do
Administrators	Full configuration rights — change these settings, create and edit processors, tags, collections, triggers and events, restore a configuration backup, and manage licensing.
General Users	Sign in and *operate and observe* the system — view the dashboard, logs and configuration, start and stop scanning — but not change the configuration.

Administrators are not automatically General Users; in practice you list your administrator group in **both** fields (or make admins members of the general group) so they can reach the view pages too.

The settings

Setting	Default	What it does and how it affects operation
Administrators Group (Windows/AD)	`Administrators`	The group(s) whose members have administrative rights. This is a **multi-select** searched against the domain (or the local machine on a workgroup server). Select one or more; a user who belongs to **any** of the listed groups is an administrator. At least one group is required.
General Users Group (Windows/AD)	`Authenticated Users`	The group(s) whose members may view and operate the system. Same multi-select behavior — membership in any listed group grants the General role. At least one group is required.
Session Idle Timeout (minutes)	20	How long a signed-in session may sit idle before it expires and the browser must re-authenticate. Range 1–1440 minutes (1 minute to 24 hours). Shorter is more secure; longer is more convenient.

> **Type to search.** The pickers do not pre-load every group — a full listing takes minutes on a large corporate directory. Click the box and start typing, and matching groups are fetched as you go. The search returns up to 50 matches at a time, so narrow the term if what you want is not shown.

Groups in another domain

By default the search looks in the domain this server belongs to. In a forest with more than one domain that will not show you everything: a group belonging to a *different* domain — including the domain many of your users' accounts actually live in — is not in that list.

To use one, **prefix the group with its domain**:

What you type	What happens
`CORP\`	Lists the groups in the `CORP` domain, so you can browse and pick one
`CORP\dom`	Lists `CORP` groups matching "dom" — for example `CORP\Domain Users`
`CORP\Domain Users`	Offered as **Use "CORP\Domain Users"** — click it to accept the name exactly as typed

Anything you type is accepted, not only what the search offers, so a group in a domain the search cannot reach can still be entered by name. Qualified entries are stored the way you entered them (`CORP\Domain Users`), and that is deliberate: **every** domain in a forest has its own group called *Domain Users*, so an unqualified name is ambiguous. Qualifying it pins down exactly which one you mean.

> **The name is checked before it is saved.** Because you can type freely, a typo in a field that controls access to the whole application is a real risk — an unrecognized group would simply grant nobody. Any group you **add** is resolved when you press Save, and if it cannot be found the page comes back with an error naming it and **nothing is saved**. Groups already saved are not re-checked, so a domain controller outage cannot block you from saving an unrelated change on this page.

You cannot lock yourself out

When you change the **Administrators Group**, Event Engine first checks that **your own account is a member of at least one of the groups you selected**. If it is not, the change is rejected with an explanation and nothing is saved — this prevents the mistake of handing administration to a group you do not belong to and losing access to the very page that controls it.

This check reads your Windows sign-in token, the same way the access check itself does, so it works even when your own account lives in a different domain from the server.

How the groups are set up at installation

Out of the box the Administrators role is `Administrators` and the General role is `Authenticated Users` . Both are standard Windows groups that exist on every machine, so the defaults work the same on a **domain-joined or workgroup** host with no setup: any signed-in account can view and operate, and any

administrator of the host machine can configure — the built-in `Administrators` group covers the *server's* local administrators, including Domain Admins on a domain-joined server.

There is no hidden "local administrators always win" rule once you map a role to a different group: a member of the host's Administrators group has no configuration rights unless a group they actually belong to is mapped. The single exception is the break-glass path below, which exists only so a mistyped group cannot lock the system permanently.

How membership is checked

Understanding this helps when someone is unexpectedly refused. Event Engine checks two things, in order:

1. **The user's Windows sign-in token.** Windows works out the full list of groups a user belongs to when they sign in, and that list already spans every domain in the forest. Reading it needs no call to a domain controller, and it identifies each group by its unique Windows security identifier — which is what makes `CORP\Domain Users` and `HQ\Domain Users` genuinely different entries rather than the same name twice.
2. **A directory lookup**, if the token did not grant access. This catches the case where a user was added to a group *after* they signed in, so a new member does not have to sign out and back on before they can get in.

A user is admitted if **either** check grants the role, so the second is a safety net, never a second chance to refuse.

> **Users whose accounts live in another domain.** This is the case the token check exists for. A directory lookup can only search one domain at a time, so on a server in one domain it cannot reliably resolve an account belonging to another — the user is refused everywhere, on every page, even though they are plainly a member of the group you mapped. If you see that, map the role to the **domain-qualified** group name (`CORP\Domain Users`) as described above. Setting the role to a broad built-in group such as *Everyone* will also appear to fix it — but that is not a fix, it disables the check entirely and grants the role to every signed-in user.

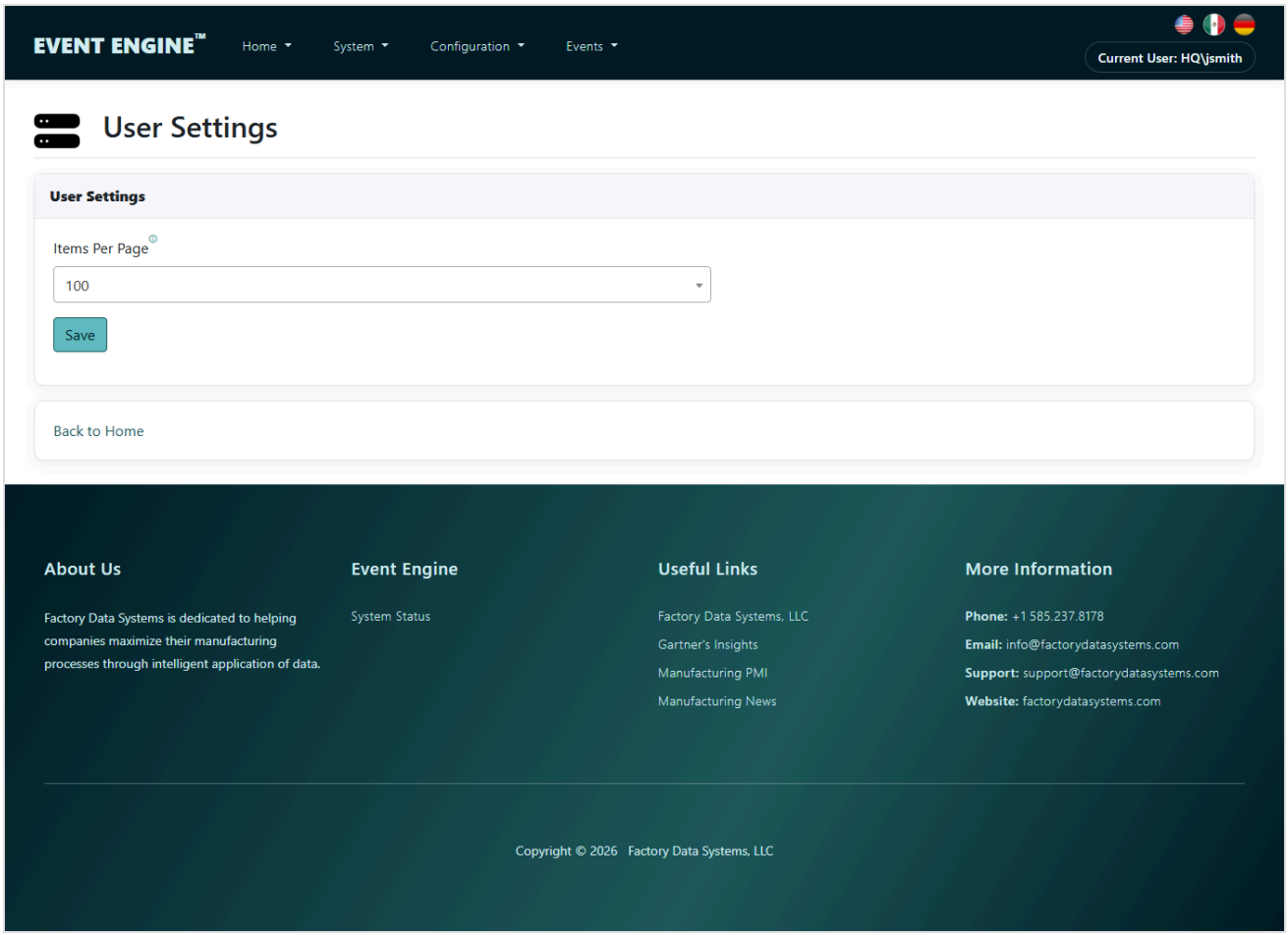
> **Break-glass recovery.** To make sure a missing group can never permanently lock the configuration, Event Engine includes one safety net: if **none** of the configured Administrators groups exist at all, a member of the server's **local Windows *Administrators* group** is allowed in so they can come to this page and set a valid group. This applies *only* when the configured group is entirely unresolvable — a normal "you're simply not a member" denial is never bypassed — and every such access is recorded in the log as a warning. Once a valid administrators group is in place, the safety net no longer applies.

> **Built-in groups.** The pickers always include the broad built-in groups — *Everyone*, *Authenticated Users*, *Users*, and *Administrators* — even though they are not enumerable from the directory.
>
> Be aware what the first three actually mean here: they are treated as "grant the role to anyone who is signed in", and no membership is tested at all. Mapping the **Administrators** role to *Everyone* therefore makes **every user who can reach the site an Event Engine administrator**, including on this settings page. They are useful for a lab or a first launch, but in production point each role at a real, narrowly-scoped group.

3.4 User Settings

System → User Settings

These preferences are stored **per Windows user**, so each person who uses the web UI gets their own values.



Setting	Default	Effect
Items Per Page	25	The number of rows shown at one time on every list page (events, tags, processors, errors, …). Choose 10, 25, 50, 75 or 100. Larger values mean fewer pages but slower page loads on big systems.

Choosing your language

Event Engine ships fully translated into **Spanish (es-MX)** and **German (de-DE)** alongside English. The choice is made from the **flag icons in the navigation bar**, not from this page: click a flag and the interface reloads in that language.

Like Items Per Page, the choice is stored **per Windows user** and applied to every page you open afterwards, so each person using the same server sees their own language. It covers the whole interface — menus, labels, field tooltips, validation messages, and the values that stream into live-updating pages.

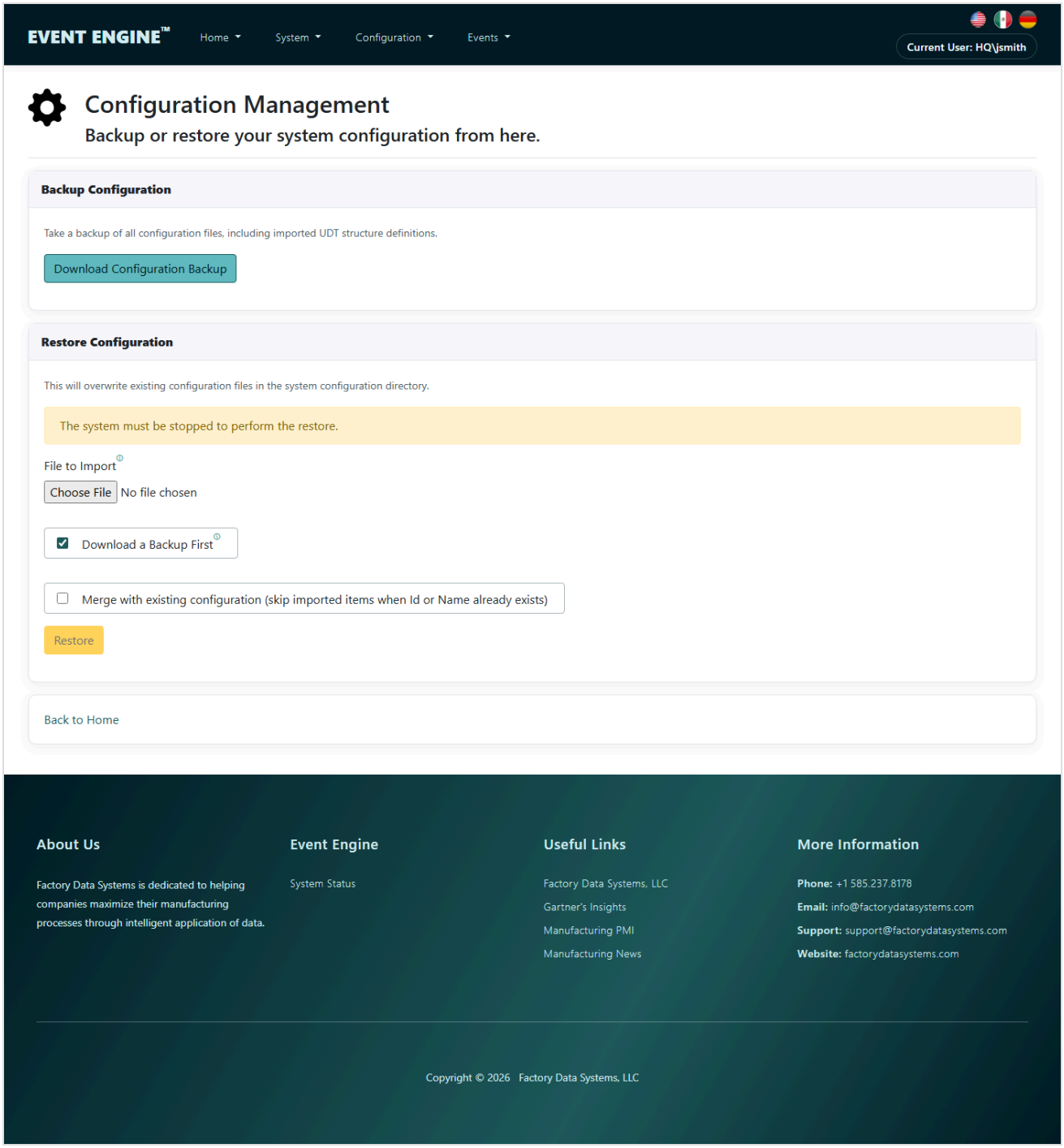
> **What stays in English.** Text that originates outside the interface is shown as it was produced: database and driver error text, log file contents, and names you typed yourself (processors, tags, events). This help is translated a page at a time: a page that has not been translated into your language yet is shown in English, with a note at the top saying so.

> **Where did Auto Refresh go?** Earlier versions offered an *Auto Refresh Period* that reloaded monitoring pages on a timer. Those pages (the dashboard, the event lists, the per-event telemetry page, and the error lists) now update **live** as values change, so the setting — and the page reloads — are gone. See *Live Updates* in section 3.1.

3.5 Configuration Backup & Restore

System → **Backup/Restore Configuration** *(restore requires the Admins role)*

Event Engine stores its entire configuration — processors, tags, collections, triggers, events, database connections, user settings — as JSON files in the configuration directory (by default `C:\ProgramData\Factory Data Systems, LLC\Event Engine\Configuration`). This page packages and restores that directory.



Taking a Backup

Click **Download Configuration Backup**. The system builds a `ConfigBackup_<date>_<time>.zip` and downloads it. The system can keep running during a backup.

The archive contains:

In the zip	What it is
Every configuration file in the configuration directory	Processors, tags, collections, triggers, events, database connections, user settings
`UDTs/Code/…`	The **generated source** of your imported UDTs
`BackupInfo.txt`	A stamp identifying the installation the backup came from

> **The UDT sources ride along on purpose.** Tags reference their structure type by identity, so a configuration restored onto a machine that never ran your UDT imports would land full of tags pointing at types that do not exist (Chapter 9.8). Carrying the source lets the restore rebuild those types. Only the **source** is included — the compiled DLLs are build output and are recompiled on restore or at startup.

> **What is *not* in it:** licenses (bound to the machine — Chapter 4), log and telemetry files, and the OPC UA certificate store (Chapter 6). Include `C:\ProgramData\Factory Data Systems, LLC\Event Engine` in a file-level backup if you want those too.

> **Recovery copies stay behind.** Alongside each configuration file the system keeps local recovery state — a `.bak` holding what the file contained before its last write, occasionally a `.tmp` left by a process that stopped mid-write, and a `.corrupt-<timestamp>` copy quarantined from a file that could not be read. These are **not** included in the archive, and a restore ignores any that an older backup contains. They describe one machine's history, not your configuration: restoring them onto a system would only accumulate copies of copies, and a stale one taken into a *merge* would be read as real configuration. If you want them, take them from the configuration directory with a file-level backup.

> **Tip:** take a backup before any large configuration change, before upgrading, and as part of your normal disaster-recovery routine. Because it carries the UDT sources, it is also the supported way to migrate a whole configuration to another Event Engine installation.

Restoring a Backup

1. **Stop the system first.** The Restore button stays disabled while the system is scanning, and the page warns: "*The system must be stopped to perform the restore.*"
2. Choose the backup `.zip` file (only ZIP files created by the backup tool are accepted).
3. Optional — **Download a Backup First**: automatically takes a backup of the *current* configuration before restoring, so you can roll back.
4. Optional — **Merge with existing configuration**: instead of overwriting, imported items are *added* to the current configuration, and any imported item whose Id or Name already exists is skipped (a

"collision"). Use this to combine configurations from two systems. Leave it unchecked for a true restore, which overwrites the existing configuration files.

5. Click **Restore**.

When the restore finishes, Event Engine reloads its configuration repositories, clears its caches, and refreshes the tag list automatically — a service restart is not normally required. The result banner reports what happened, for example: *"Merge restore complete. Files processed: 12. Items imported: 87. Collisions skipped: 3. Runtime refresh complete. Repositories reloaded: 12."*

> **Safety notes:** the upload is validated before extraction, and a corrupt or non-backup ZIP is rejected. If the runtime refresh reports failed repositories, restart the Event Engine service to fully load the restored configuration.
>
> A restore writes the configuration files one at a time. If one cannot be written — another program has it open, for example — the restore puts back every file it had already written, and the banner says the configuration is unchanged. If a file cannot be put back either, the banner names it and says the configuration is now a mix of the backup and what it replaced: run the restore again before relying on it.

3.6 HTTPS

System → HTTPS *(requires the Admins role)*

The web interface serves **HTTPS alongside HTTP**. The installer sets this up without any manual work — it generates a certificate for this server, binds it to port 443 (or 8443 when 443 is taken), and turns on a redirect from HTTP — so this page exists for the things an administrator *does* need: seeing which certificate is bound, replacing it with one your whole network trusts, and controlling the redirect. What a site experiences at upgrade is summarized in the Release Notes.

The status card

The card at the top shows one of three states, and the certificate currently bound:

State	Meaning
HTTPS enabled	A certificate is bound to the HTTPS port and the site answers on `https://`.
Enabled, no binding	Configuration says HTTPS is on, but no certificate is bound to the port — the `https://` address will not answer. The Error List names the cause; re-running the installer repairs the standard setup.
HTTPS disabled	The installation has opted out (see below). The page offers to enable it.

The certificate table shows the **subject**, the **names it covers** (`localhost`, the machine name and the fully-qualified domain name for the generated certificate), the **expiry date** — with a red warning badge once it is 30 days out — the **thumbprint** (useful for confirming which certificate a client is actually seeing), and its **origin**: generated at install, or imported by you.

The HTTP → HTTPS redirect

The **Redirect HTTP requests to HTTPS** switch controls whether a browser that opens the plain HTTP address is sent to the HTTPS one. Two properties are deliberate:

- **It only redirects while the binding is healthy** — certificate bound, loadable and unexpired. The moment that stops being true, the site falls back to answering on plain HTTP rather than redirecting browsers to an address that would fail. A lapsed certificate therefore degrades gracefully instead of locking operators out of the very pages needed to fix it, and the Error List reports what is wrong.
- **The switch applies immediately** — no service restart.

Importing a company certificate

The generated certificate is trusted **only on the server itself**; browsers on other computers show a certificate warning until either that certificate is distributed to them (for example by Group Policy) or — the cleaner fix — you import one issued by your company's certificate authority here.

The import is a **two-step, validate-then-import** flow, so a wrong file or password is caught before anything changes:

1. Choose the **`.pfx`** (or `.p12`) file and enter its password, then click **Validate**. The page shows the certificate's subject, expiry and the names it covers, along with any warnings — for example a certificate that does not cover the server's name, or one that is close to expiry. Nothing has been changed yet.
2. If the result is what you expect, click **Import**. The certificate is installed to the machine's store, the private-key permissions the service needs are granted automatically, and the HTTPS binding is switched to it **immediately — no service restart**. Any change to the chosen file or password re-disables Import until you validate again.

The file must contain the **private key** (which is what a `.pfx` is) and be valid for **server authentication**; your certificate authority's standard web-server certificate is exactly that. Request it for the DNS names operators actually browse to — the server's name and fully-qualified domain name.

> **Why the page may say imports are only possible from the server itself.** The certificate password travels with the upload, so the page accepts an import only from a browser **on the server itself** or over an **already-working HTTPS connection** — never over plain HTTP from another machine, where the password would cross the network unencrypted. If the import card is disabled with that notice, sign in from the server, or browse to the site's `https://` address first.

Expiry monitoring

Event Engine checks the certificate on a schedule and raises Error List warnings for the things that would otherwise be discovered by an operator's browser: **expiry within 30 days** (and expiry itself), a **missing binding**, and a certificate left covering the **wrong names** after the machine is renamed. Import a replacement certificate before expiry and the binding follows it with no restart; if the certificate does lapse, the site keeps answering on HTTP as described above.

Enabling HTTPS when it is disabled

If the installation opted out, the page shows an **Enable HTTPS** card instead of the import card: choose the port (443 unless something else owns it) and click Enable. A certificate is generated for this server and the configuration updated; **this path requires a service restart** to take effect. The same loopback-or-HTTPS rule as importing applies.

To opt out, set `"EnableHttps": false` in the service's `appsettings.json` and restart the service. The opt-out is respected by every future upgrade.

> **Uninstalling** removes the HTTPS bindings and only the certificate the installer generated — a certificate you imported stays in the machine's store.

Next: Chapter 4 — Product Licensing · Guide home

Chapter 4 — Product Licensing

System → Product Licensing

Event Engine is licensed software. Licensing is managed entirely from the web UI and is backed by the SoftwareKey licensing platform. This chapter explains the trial period, the license types, and every activation, renewal, and deactivation workflow — both online and offline.

> **Where licenses live:** activated license files are stored on the server at `C:\ProgramData\Factory Data Systems, LLC\Event Engine\Licenses`. Each license is bound to the machine it was activated on — primarily the machine's BIOS/system UUID (on a virtual machine, the VM's BIOS GUID), plus the operating-system volume serial number and the computer name — so license files cannot simply be copied between machines. Use deactivation + activation for a planned move; for restores from backup and unplanned machine changes, see 4.9.

4.1 The Trial License

The first time Event Engine runs on a machine **without a valid license**, it automatically starts a **30-day trial**:

- The trial start date is anchored on first run and a `trial.xml` license file is written to the licenses folder.
- During the trial the product is **fully functional** — event-count limits are bypassed.
- A banner reports "*A trial license is currently in effect until <date>.*"
- When the trial expires, the system can no longer start scanning, and the UI reports: "*The trial license for this installation has expired. Please contact Factory Data Systems support for assistance.*"

The trial cannot be reset by deleting files — the start date is independently recorded on the machine. To continue after the trial, purchase and activate a license.

4.2 License Types

Event Engine is licensed by event capacity:

Type	What it licenses	How it is enforced
Event license (event counter)	The number of **enabled** events the system may run. Multiple event licenses stack — the total allowance is the sum of (counter × quantity ordered) across all valid event licenses.	If the number of enabled events exceeds the licensed total, the system will not start, and you cannot enable additional events. The UI warns when the count *meets* the quota and blocks when it *exceeds* it.

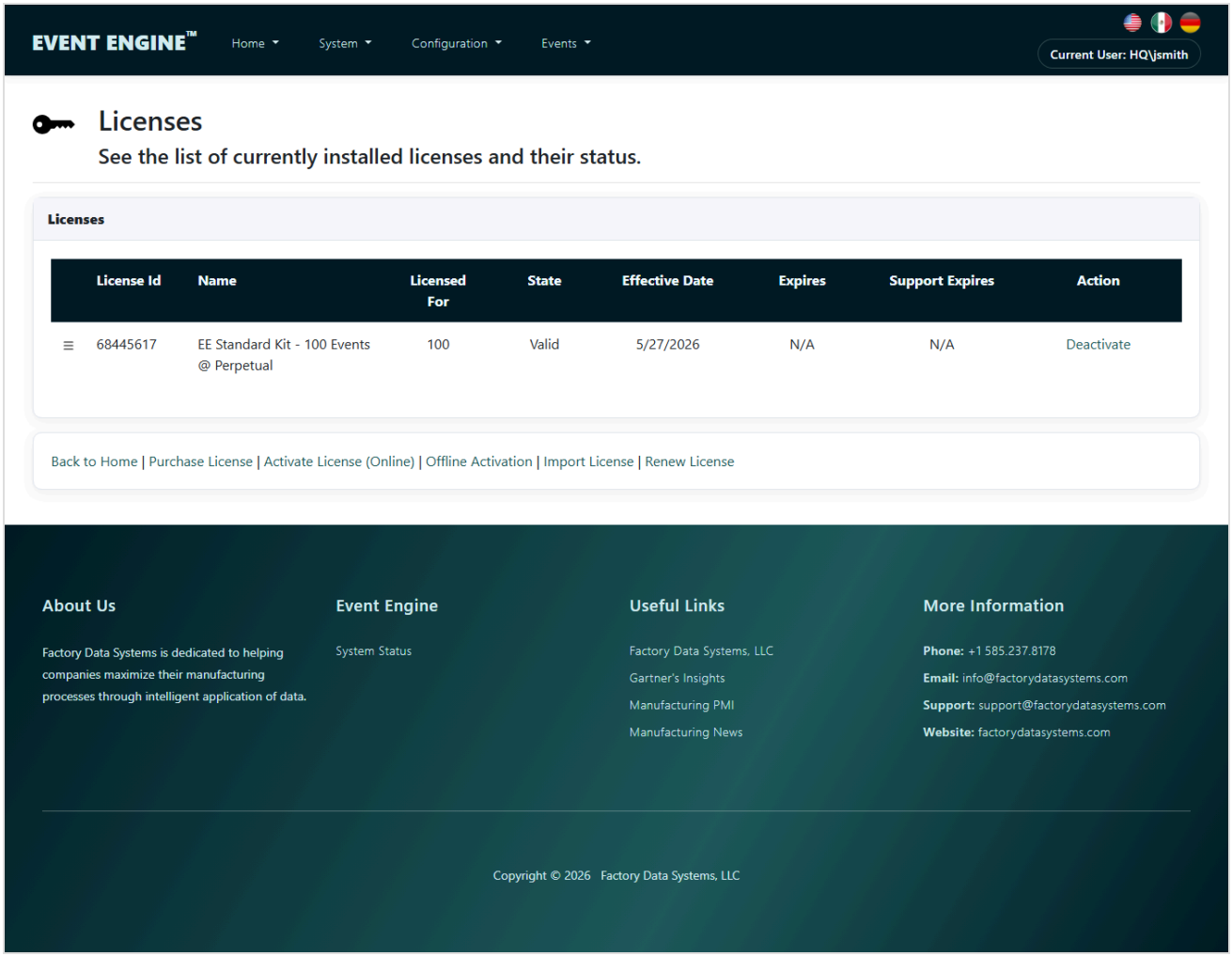
> **Processor licenses are no longer required.** All PLC protocol families (Logix, SiemensS7, Modbus, SRTP, Legacy, Opc) are available to every licensed installation. Processor licenses purchased previously may still appear in the license list; they no longer affect anything and do not need to be renewed.

Perpetual vs. Subscription Licenses

- A **perpetual** license **never expires** — it validates indefinitely on the activated machine. The date it carries is its maintenance and support agreement's paid-through date, and the license list shows that same date in both **Expires** and **Support Expires**. When it passes, the license keeps validating and the installed version keeps running, but versions of Event Engine built after that date will not run until maintenance is renewed (see 4.11). A perpetual license issued without a date shows "N/A" in both columns.
- A **subscription** license has an expiration date and must be renewed. Maintenance and support run with the subscription term, so **Expires** and **Support Expires** show the same date:
    - Beginning **60 days** before expiration the license list shows a **Renew** button and the system issues warnings such as "*License 'X' expires in N day(s). Please renew soon.*" An expiration warning e-mail (at most one a day) is also sent — but **only if all three of SMTP Relay Address, From Address and System Message Delivery Address are configured** (Chapter 3.3). With any of them blank the warning is skipped silently, which on an unattended system means the first sign of a lapsed license is scanning stopping. Worth confirming before you rely on it.
    - After the expiration date there is a **14-day grace period** during which the license still validates, with the warning "*License 'X' expired N day(s) ago and is within its grace period. Please renew immediately.*" During the grace period, an expired license also attempts an automatic online refresh, so a renewed subscription usually "heals" itself if the server has internet access.
    - After the grace period the license is invalid: scanning stops / cannot start, and the event reports "*Event Engine stopped scanning … because the trial period has ended or a required license is missing or expired.*"

4.3 The License List Page

System → Product Licensing



The table shows every installed license:

Column	Meaning
License Id	The license number — you'll need it for activation and support.
Name	The product option name (what was purchased).
Licensed For	The licensed quantity (e.g., number of events). Shows 1 for non-counted licenses.
State	**Valid**, **Expired**, **Version not covered** (this build postdates the support date — see 4.11), or **Invalid** (fails validation for another reason — e.g., moved hardware).
Effective Date	When the license became effective.
Expires	The end date the license carries, or **N/A** if it carries none. On a subscription this is a real expiration; on a perpetual license it is the maintenance date and the license does **not** stop working when it passes (see 4.2).

Column	Meaning
Support Expires	The date the maintenance and support agreement is paid through — the same date as Expires, named for what it actually governs. Shown in red once it has passed. A lapsed support date does **not** stop the installed version from running; it only means newer versions of Event Engine are not covered until maintenance is renewed.
Action	**Renew** (shown within 60 days of the Support Expires date), **Refresh License** (re-download an expired license after renewal), **Deactivate** (release this machine's activation).

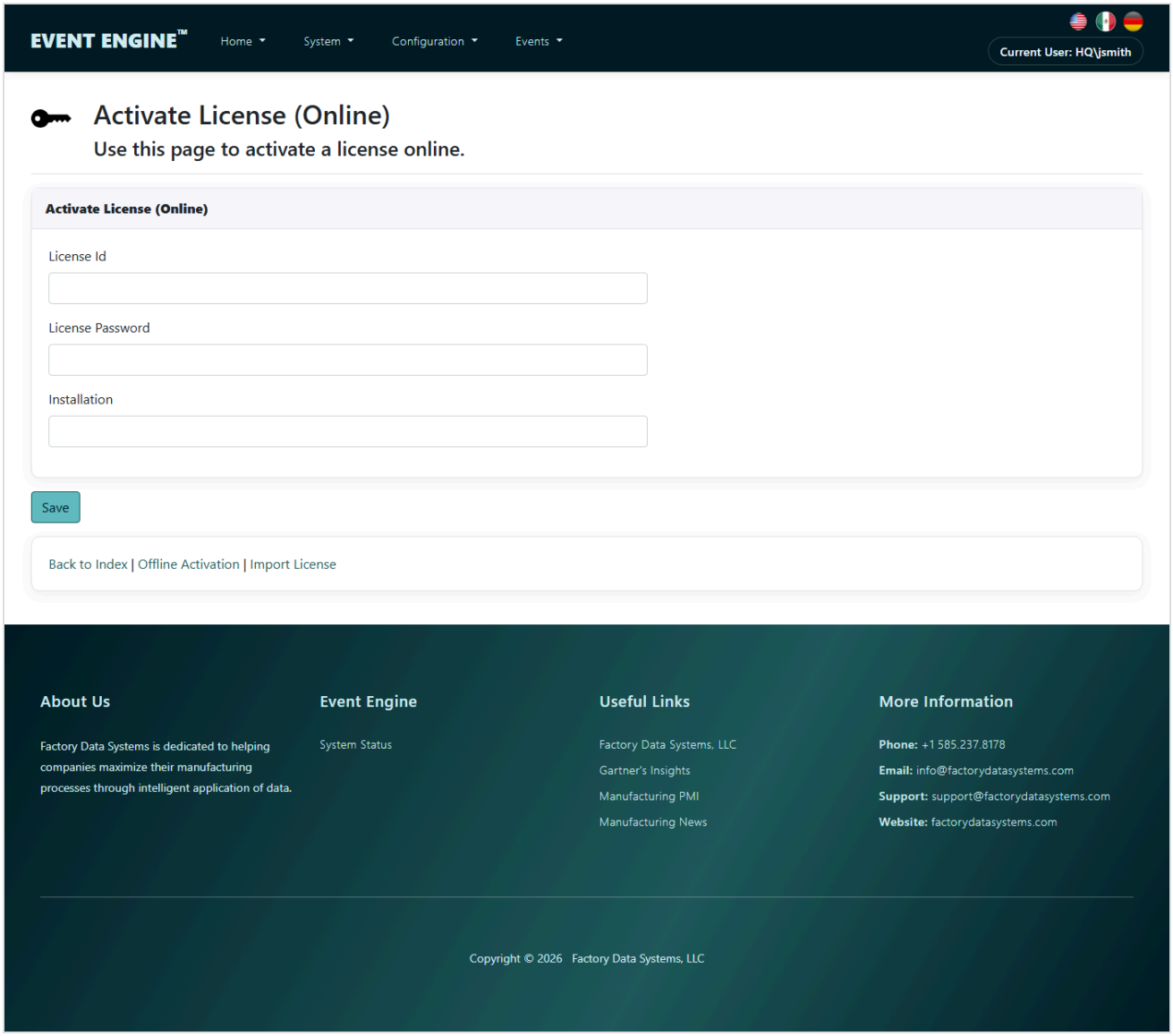
The links at the bottom of the page lead to every licensing workflow:

- **Purchase License** — opens the Factory Data Systems storefront (see 4.10).
- **Activate License (Online)** — see 4.4.
- **Offline Activation** — see 4.5.
- **Import License** — used in the offline workflows.
- **Renew License** — see 4.6 / 4.7.

4.4 Activating Online

Use this when the Event Engine server has internet access. (If the licensing server cannot be reached, this page automatically redirects you to Offline Activation.)

1. Go to **System** → **Product Licensing** → **Activate License (Online)**.
2. Enter the **License Id** and **License Password** from your purchase confirmation.
3. Optionally enter an **Installation** name (helps you recognize this machine in the customer portal).
4. Click **Save**.



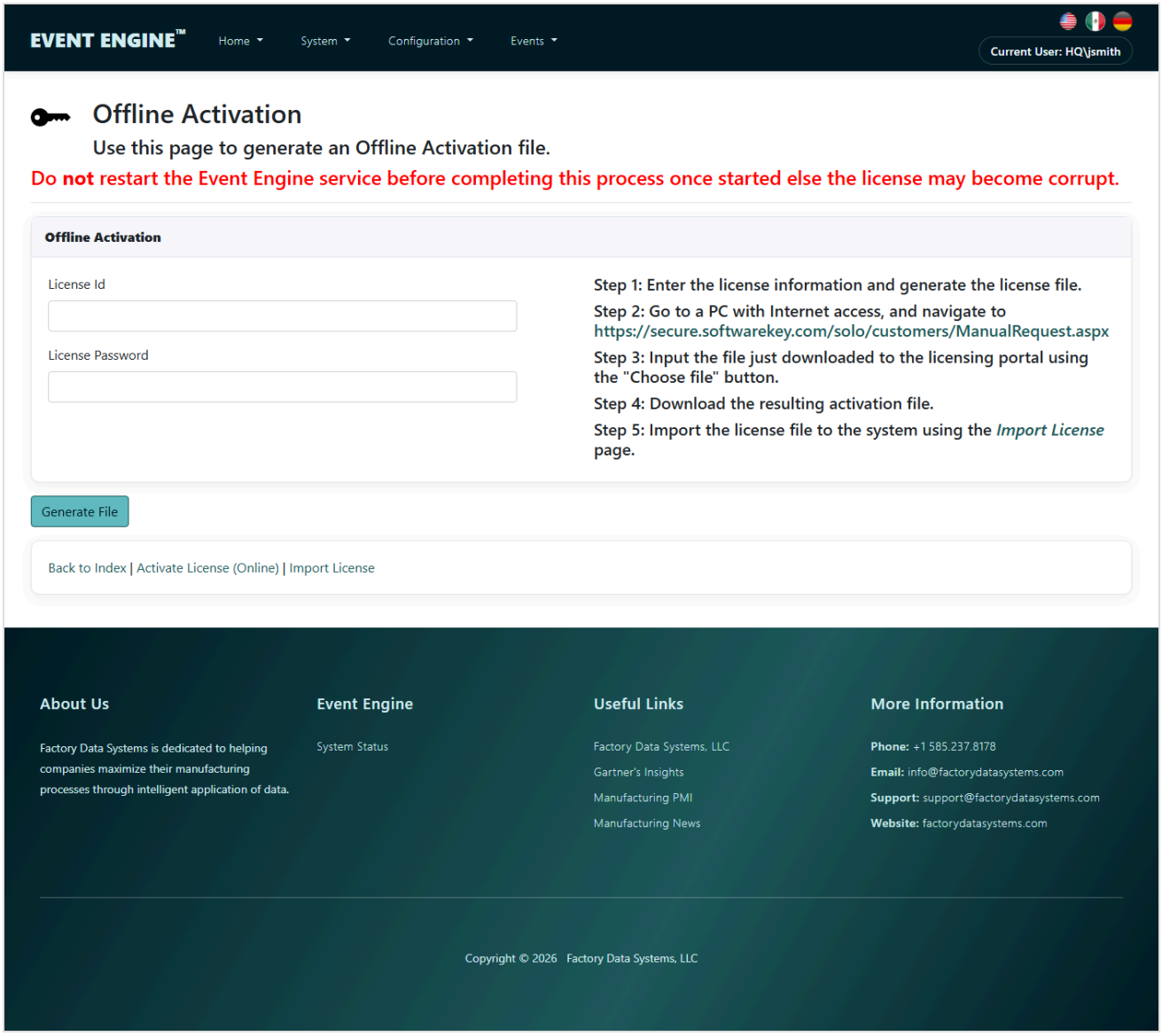
Event Engine contacts the SoftwareKey activation service, downloads the license file, validates it, and adds it to the license list. A license that is already active on this machine is rejected with *"This license has already been activated!"*

4.5 Activating Offline

Use this when the Event Engine server has **no internet access**. You will carry a request file to any internet-connected PC and bring a response file back.

> ⚠️ **Important:** Do **not** restart the Event Engine service between generating the request file and importing the response file — the request and response are cryptographically tied to the running session, and restarting in between can corrupt the activation. If that happens, simply generate a new request and repeat.

1. Go to **System** → **Product Licensing** → **Offline Activation**.
2. **Step 1:** Enter the **License Id** and **License Password**, then click **Generate File**. The browser downloads `Event Engine License File.txt` — the activation request.
3. **Step 2:** On a PC with internet access, go to the license activation portal: https://secure.softwarekey.com/solo/customers/ManualRequest.aspx
4. **Step 3:** Upload the request file using the portal's **Choose file** button and submit.
5. **Step 4:** Download the resulting activation response file from the portal.
6. **Step 5:** Back on the Event Engine server, go to **System** → **Product Licensing** → **Import License**, select the response file (`.xml`), and import it.

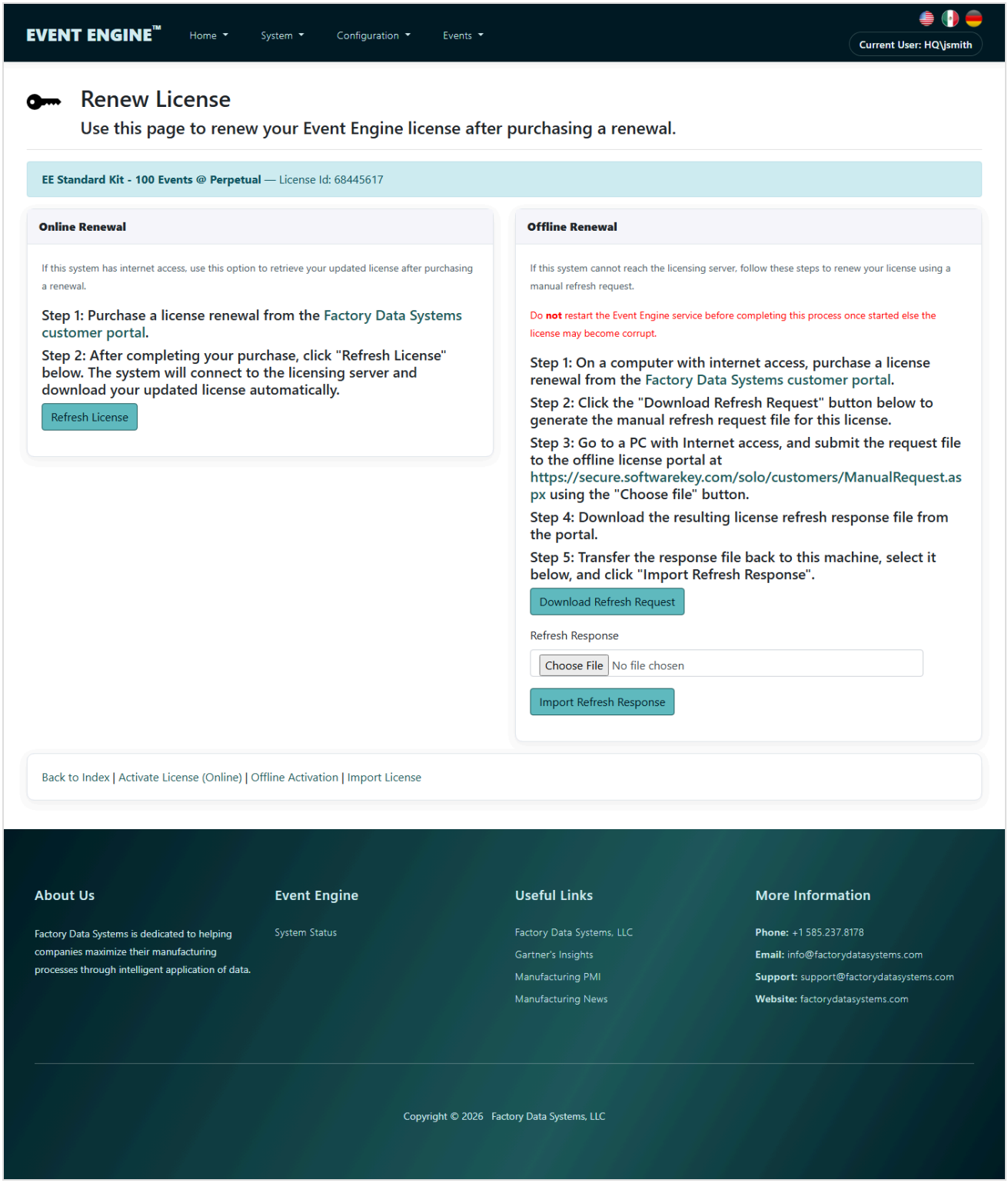


The Import License page also accepts license files exported directly from the Factory Data Systems customer portal — you may import one or more `.xml` license files at a time.

4.6 Renewing Online

Use this after purchasing a subscription renewal when the server has internet access.

1. Purchase a license renewal from the Factory Data Systems customer portal (see 4.10).
2. Go to **System** → **Product Licensing**, click **Renew** next to the license (or open **Renew License** and select the license from the list).
3. In the **Online Renewal** card, click **Refresh License**. The system connects to the licensing server and downloads your updated license automatically. A success banner confirms: "*The license was refreshed successfully.*"



> The **Refresh License** button also appears directly in the license list for licenses that have expired — use it any time the licensing server holds a newer license than the local file.

4.7 Renewing Offline

Use this after purchasing a renewal when the server has **no internet access**.

> ⚠️ The same warning applies as offline activation: do **not** restart the Event Engine service between downloading the refresh request and importing the refresh response. If the import fails, generate a *new* request, submit it to the portal, and import the *new* response.

1. **Step 1:** On a computer with internet access, purchase the renewal from the Factory Data Systems customer portal.
2. **Step 2:** On the Event Engine server, open **Renew License** for the license and click **Download Refresh Request**. The browser downloads `Event Engine License <id> Refresh Request.txt`.
3. **Step 3:** On the internet PC, submit the request file to the offline license portal at https://secure.softwarekey.com/solo/customers/ManualRequest.aspx using **Choose file**.
4. **Step 4:** Download the refresh response file the portal produces.
5. **Step 5:** Back on the server, in the Offline Renewal card, select the response file under **Refresh Response** and click **Import Refresh Response**.

4.8 Deactivating a License (Moving to Another Machine)

To move a license to different hardware you must first release it from the current machine:

- **Online:** in the license list click **Deactivate** and confirm. The activation is returned to your license account and the local license file is removed.
- **Offline:** if the licensing server is unreachable you are taken to the **Offline Deactivation** page: download the deactivation request file, then submit it to https://secure.softwarekey.com/solo/customers/ManualRequest.aspx from an internet PC so the activation is credited back to your account. The local license is removed when the request file is generated.

4.9 Backups, Restores, and Machine Changes

Event Engine is designed so that **restoring a backup never locks you out**.

Restoring to the same machine — nothing to do

If you restore a backup, snapshot, or checkpoint **onto the same machine or VM it was taken from** (for example, rolling back a failed Windows update with a Veeam restore or a Hyper-V checkpoint), your licenses keep working immediately. There is nothing to reactivate and no warning appears.

Renaming the computer or changing its disks also does **not** disturb licensing, as long as it is still the same machine (licenses activated with an older version of Event Engine may still warn after such changes — reactivating once on the current version resolves this permanently).

Restoring or moving to a different machine — 30 days to reactivate

If Event Engine detects that a license is running on a **different machine** than the one it was activated on — typically after restoring a backup onto a newly created VM, or migrating to replacement hardware — it does **not** stop working. Instead:

- The product remains **fully functional for 30 days** with your normal licensed capacities.
- Every page shows the warning: *"This system appears to have been restored from a backup or moved to a different machine. License "X" remains fully functional for N more day(s); please reactivate it to keep this system licensed."*
- **To clear the warning, reactivate each affected license on this machine** using the normal activation workflow — online (4.4) or offline (4.5). Your License Id and password are the same ones from your original purchase confirmation.

If the 30 days elapse without reactivation, the licenses stop validating and scanning stops until you reactivate. The countdown is per machine and cannot be restarted by rebooting, re-importing the license file, or refreshing the license.

> **This is not the trial.** The reactivation window is a property of your *purchased licenses* and is independent of the 30-day trial (4.1): your licensed event counts stay exactly as purchased (not the trial's unlimited bypass), and it works even though the trial on the restored machine was used up long ago. Likewise, when the window lapses, the trial does **not** come back — the only way forward is reactivating the license.

> **If activation reports no activations remaining:** the license is still recorded as active on the old machine. Deactivate it there if the machine still exists (see 4.8), or contact Factory Data Systems support to release the old activation.

> **Planning to decommission a machine?** Prefer the clean path: deactivate on the old machine first (4.8), then activate on the new one. The 30-day window is the safety net for *unplanned* moves — disasters, failed hardware, and emergency restores.

4.10 Licensing Links (Quick Reference)

Purpose	Link
Customer portal / storefront (purchase, renewals, view your licenses)	https://secure.softwarekey.com/solo/customers/Default.aspx?AuthorID=5547024

Purpose	Link
Manual request portal (offline activation / renewal / deactivation files)	https://secure.softwarekey.com/solo/customers/ManualRequest.aspx

4.11 Licensing Troubleshooting

Symptom	Cause / Resolution
Start button disabled with *"An event usage plan is not currently licensed..."*	No valid event-counter license. Activate one, or check the State column for Expired/Invalid.
"The number of events configured has exceeded the quota allowed by the license."	Disable events until you are within your licensed count, or purchase additional event capacity.
"The license could not be refreshed."	The server cannot reach the licensing service — verify internet/proxy/firewall access to `secure.softwarekey.com`, or use the offline renewal workflow.
"This license has already been activated!"	The license is already in the list — no action needed.
Banner: *"This system appears to have been restored from a backup or moved to a different machine..."*	Normal after restoring onto a new VM or replacement hardware. Fully functional for the days shown — reactivate each named license on this machine (4.4 or 4.5) to clear it. See 4.9.
State shows **Version not covered**, or the banner *"This version of Event Engine (built ...) is newer than your maintenance coverage (through ...)"*	This build was released after the **Support Expires** date on the license. Renew maintenance and use **Refresh License** to pull the extended date down, or reinstall the version you are entitled to. The version you were running before the upgrade is unaffected.
License shows **Invalid** but not Expired	The machine no longer matches the license at all (e.g., a license file copied to an unrelated machine). Activate the license on this machine normally; contact support if activations are exhausted.
Renamed the computer / changed disks and licensing complains	Should not happen for licenses activated on the current version — the license recognizes the same machine through such changes. For licenses activated on older versions, reactivate once on this machine to rebind.
Restored a license file from a backup and it is ignored	Deactivated licenses are remembered by the machine; a restored copy of a deactivated license file will not load. Activate normally instead.

Next: Chapter 5 — Database Connections · Guide home

Chapter 5 — Configuration: Database Connections

Configuration → Database Connections

> **Where this fits: Step 2** of the setup roadmap. Do this any time after licensing, but **before** the events that use it (Ch. 10, Ch. 11) — choosing a connection is what lets an event list the database's procedures, tables, and columns. *(Skip this chapter if you will only print labels or send messages.)*

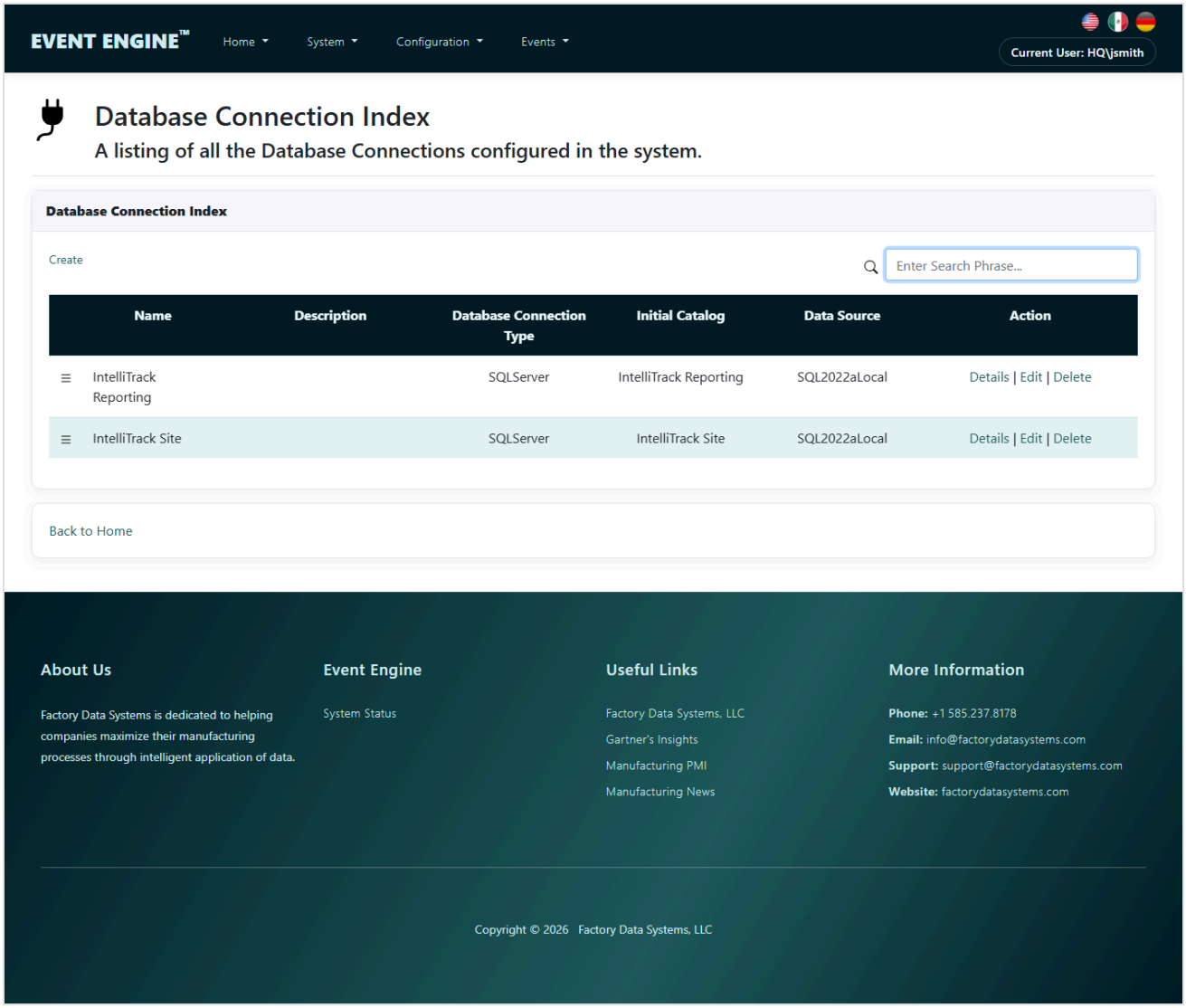
A **Database Connection** tells Event Engine how to reach one of *your* databases — the databases your events will call stored procedures in, insert rows into, or read rows from. Every Stored Procedure event (Chapter 10) and Database Table event (Chapter 11) references exactly one database connection.

Database connectivity is plugin-based. The shipped connectors are:

Connector	Typical use
Microsoft SQL Server	SQL Server and Azure SQL databases
PostgreSQL	PostgreSQL databases
MySQL	MySQL / MariaDB databases
Oracle	Oracle databases

5.1 The Database Connection Index

The index lists every configured connection with searching (by name, description, server, and database) and paging. From here you can create, view, edit, or delete connections.



> **Effect of deleting or editing a connection:** every event that references the connection is affected immediately. Deleting a connection that events depend on will break those events, so the system warns you when a connection is in use.

5.2 Creating a Connection

1. Open **Configuration** → **Database Connections** and choose **Create**.
2. Pick the database type (SQL Server, PostgreSQL, MySQL, or Oracle). Each type opens its own create page provided by that connector plugin.
3. Fill in the fields (below) and use **Test Connection** before saving.

Create New SQL Server Connection

Create a new connection for use with a SQL Server database

Details

Name

Description

Host

Database

Minimum Pool Size
10

Maximum Pool Size
20

Connect Timeout (seconds)
15

Integrated Security

Encrypt Connection

Trust Server Certificate

User Name

Password

Repeat Password

Test Connection

Create

Back to Index

Copyright © 2026 Factory Data Systems, LLC

Connection Fields

Field	What it does / how to choose a value
Name	A unique, friendly name. This is what you'll pick from dropdowns when building events, so make it descriptive (e.g., `MES-Production`).
Description	Free text shown in lists.
Host	The name or IP address of the server hosting the database.
Database	The name of the database (initial catalog) on that server.
Minimum Pool Size	The number of connections opened up front and kept open to this database (1 – Maximum Pool Size, default **10**). Events served by one of these warm connections never wait on a new physical login, so on servers where opening a connection is slow, raising the minimum keeps event latency flat.
Maximum Pool Size	The most *concurrent* connections Event Engine may open to this database (1 – 500, default **20**). Each executing event borrows a pooled connection for the duration of its database work; once the maximum is reached, additional events wait for a free connection. Raise this only if you have very high event concurrency *and* your DBA confirms the database can handle it.
Integrated Security	(SQL Server) Uses the Event Engine service's *Windows identity* to authenticate instead of a username/password. This requires special configuration of the service account and database, so in most cases it is **not** used — prefer a dedicated SQL login unless your administrator says otherwise. When checked, the username/password fields are disabled.
User Name	The database login used to access the database (when not using Integrated Security).
Password / Repeat Password	The login's password, entered twice to guard against typos. The two entries must match.
Encrypt Connection	(SQL Server) Encrypts all traffic between Event Engine and the database, including the login itself. **Required** by Azure SQL and by any server configured with *Force Encryption* — those servers refuse an unencrypted connection outright. Default **off**, so existing connections are unaffected by an upgrade; turn it on for any new connection whose server supports it.
Trust Server Certificate	(SQL Server) Skips validation of the server's certificate. Only applies when **Encrypt Connection** is checked, and is disabled otherwise. Use it only for a server presenting a self-signed or internally issued certificate this machine does not trust. The traffic stays encrypted, but the server's identity is no longer verified — see the note below.

Choosing encryption settings

Work through it in this order:

1. **Try Encrypt Connection on its own first.** If the server has a certificate your Windows machines already trust — anything issued by your internal CA, or Azure SQL — this is all you need, and it is the correct end state.

2. **If that fails with a certificate or trust error**, the host does not trust the server's certificate. The *best* fix is to install the issuing authority into the Event Engine host's Trusted Root store, which leaves validation intact for every connection.
3. **Only if that is not possible**, check **Trust Server Certificate**. This is common for an on-premises SQL Server using the self-signed certificate it generated at install time.

> **What Trust Server Certificate gives up.** Encryption without validation still stops someone *reading* the traffic, but it no longer proves the server on the other end is the one you meant to reach. Anyone able to redirect the connection could present their own certificate and see everything, including the login. That is why it is a fallback and not a default.

Leave both unchecked only for a server that does not support encryption at all.

Test Connection

The **Test Connection** button attempts a live connection with the values entered on the form and reports:

- ✅ *Connection Successful*
- ❌ *Connection Failed. Please check your credentials and try again.*

Always test before saving — a bad connection won't show up again until an event tries to use it.

5.3 How Events Use Connections

- When you build a **Stored Procedure** event, choosing a connection lets Event Engine query the database's available stored procedures and read each procedure's parameter list automatically.
- When you build a **Database Table** event, choosing a connection lets Event Engine list the tables and views and read their column definitions.
- At run time, the event executes against this connection using the configured pool; the per-event **Timeout** (in the event's options) bounds how long an individual call may take.

> **Permissions tip:** grant the configured login only what events need — typically `EXECUTE` on the relevant procedures and `SELECT` / `INSERT` on the relevant tables. Event Engine discovers procedures, parameters, and columns through the connection, so the login also needs permission to read that metadata.

Next: Chapter 6 — Processors · Guide home

Chapter 6 — Configuration: Processors

Configuration → Processors

> **Where this fits: Step 3** of the setup roadmap. Create your processors **before** tags (Ch. 9) — every tag lives in a processor.

A **Processor** is a configured connection to a PLC or controller. Every tag in Event Engine belongs to exactly one processor, so creating your processors is the first step toward scanning real data.

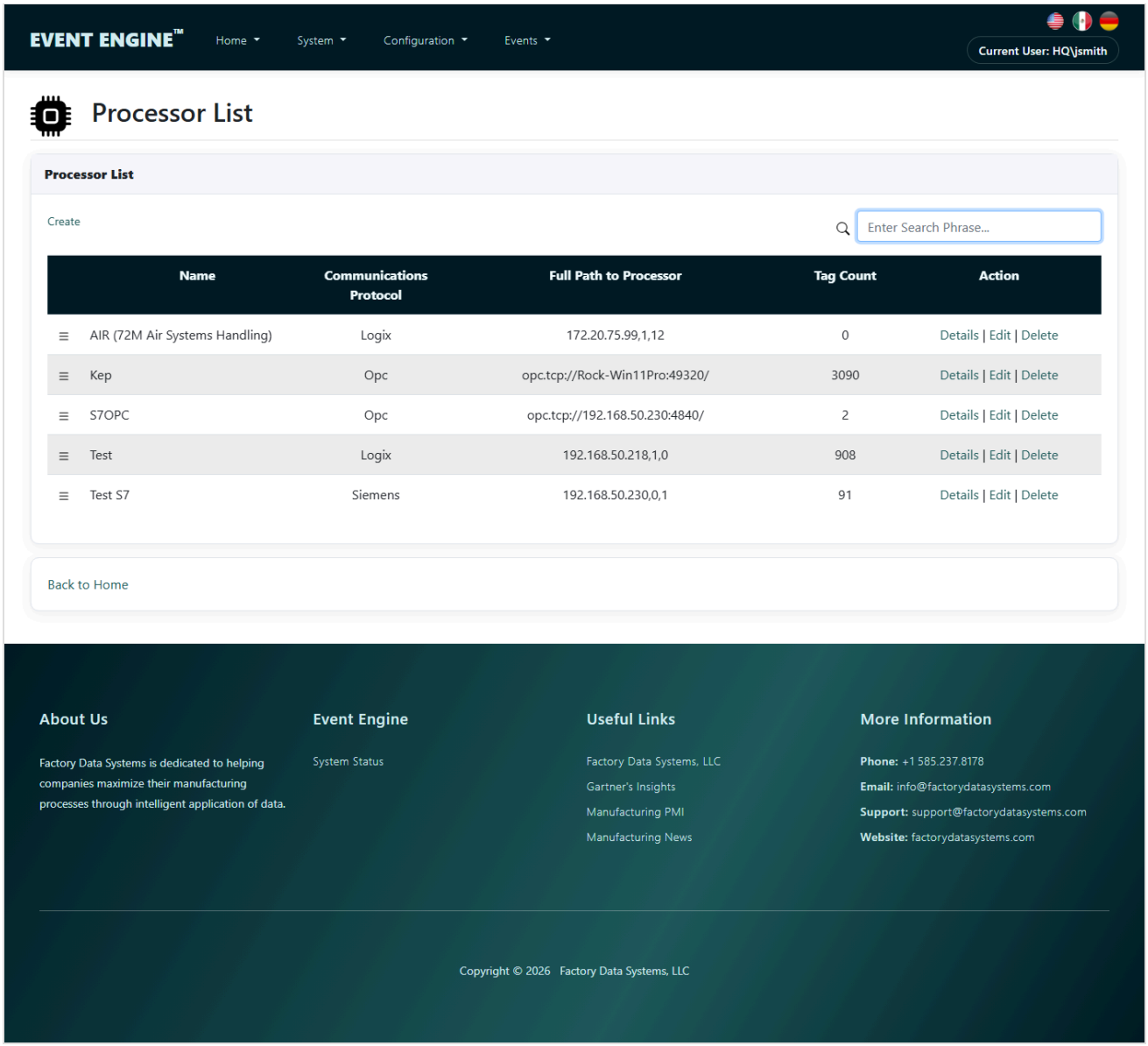
Device communication is plugin-based. The shipped protocols are:

Protocol	Description	Notes
Rockwell Logix Controller (`Control Logix PLC`)	Rockwell Ethernet driver for the Logix family (ControlLogix, CompactLogix, GuardLogix, etc.) and Micro800	Supports online **tag browsing** and **UDT import** from `.L5X` files
Allen-Bradley Legacy Controller	AB Ethernet driver for legacy controllers (PLC-5, SLC 500, MicroLogix)	Tags are entered using native file addressing
Siemens S7	Siemens S7 Ethernet driver for S7-200, S7-300, S7-400, S7-1200, S7-1500	Supports **UDT import** from `.scl` / `.stl` / `.db` / `.udt` sources; each tag uses either absolute addressing (DB/M/I/Q) or, on S7-1200/1500, symbolic addressing
Modbus/TCP	Modbus/TCP client (master) driver	Tags address coils, discrete inputs, and registers
GE SRTP	GE SRTP Ethernet driver for PACSystems, Series-90, and VersaMax	Tags use %R/%I/%Q style references
OPC UA	OPC UA client driver	Connects through an OPC UA server rather than directly to the PLC. **OPC UA only** — an OPC Classic (OPC **DA**) server has to be fronted by a DA-to-UA gateway

> **Licensing:** all protocol families are available to every licensed installation — no per-protocol license is required (see Chapter 4).

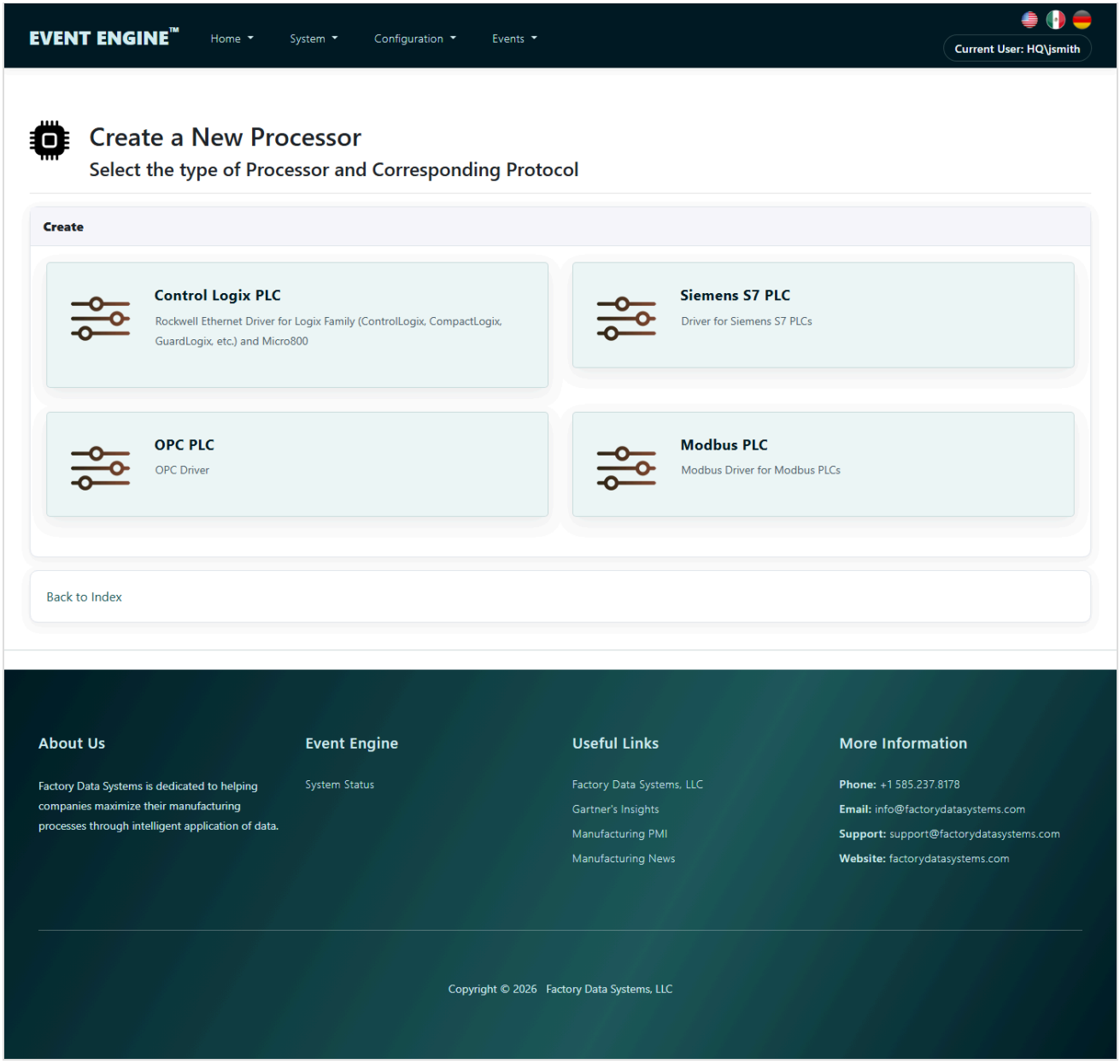
6.1 The Processor Index

The index lists every processor with its protocol, tag count, and actions. Search and paging work the same as on the other index pages.



6.2 Creating a Processor

1. Open **Configuration** → **Processors** → **Create**.
2. Choose the protocol. Each protocol opens its own create page with the fields that protocol needs.

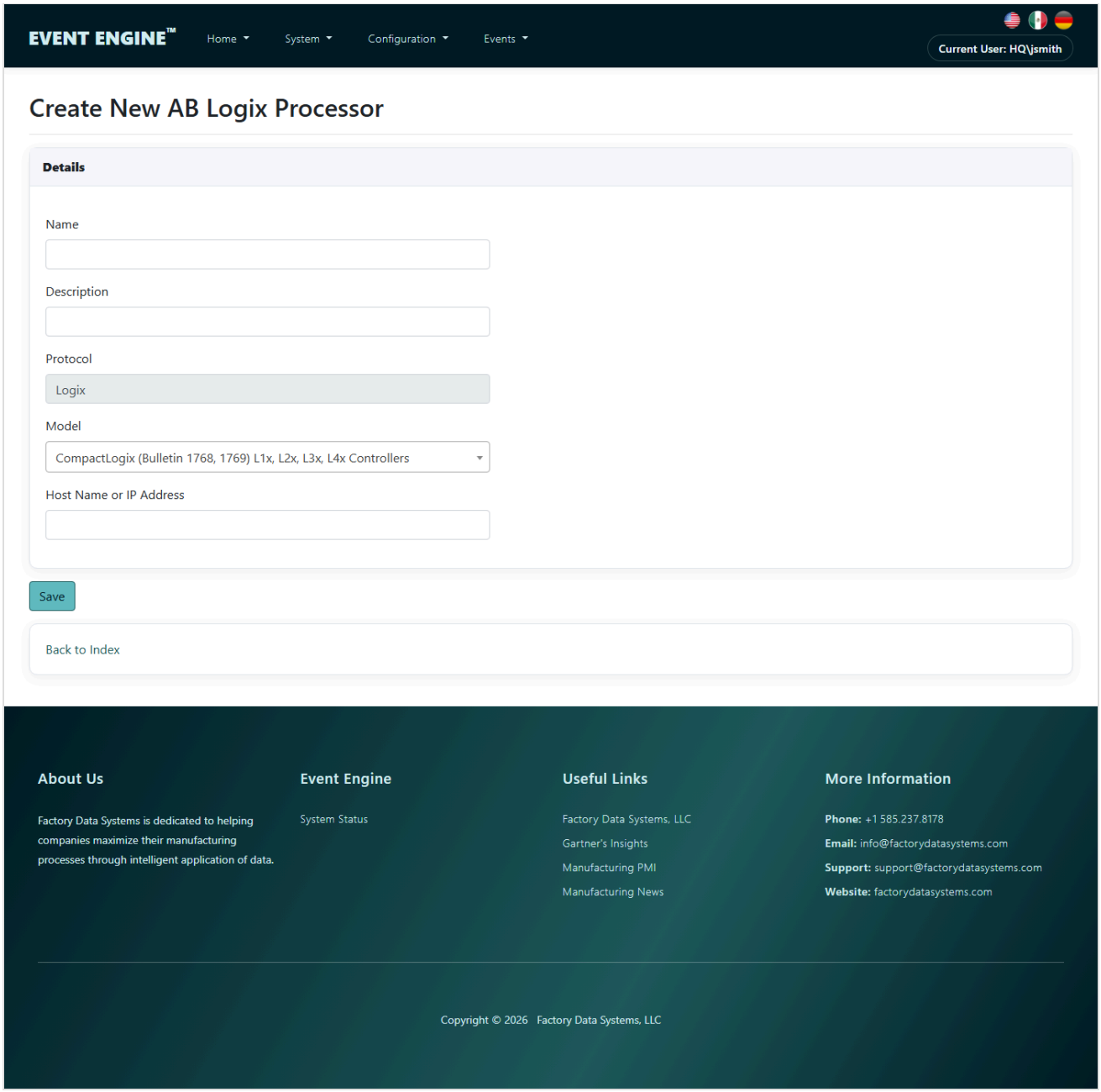


Fields Common to All Protocols

Field	What it does
Name	A unique, friendly name (often called the "shortcut"). Tags display as `ProcessorName::TagName`, so keep it short and meaningful (e.g., `Line1CLX`).
Description	Free text shown in lists.
Communications Protocol	Fixed by the create page you chose.

Field	What it does
Processor Model	The specific controller model. Choosing a model adjusts which connection fields apply (for example, a rack-mounted ControlLogix shows backplane/slot fields; a Micro800 does not).
IP Address	The controller's (or gateway's / OPC server's) address.

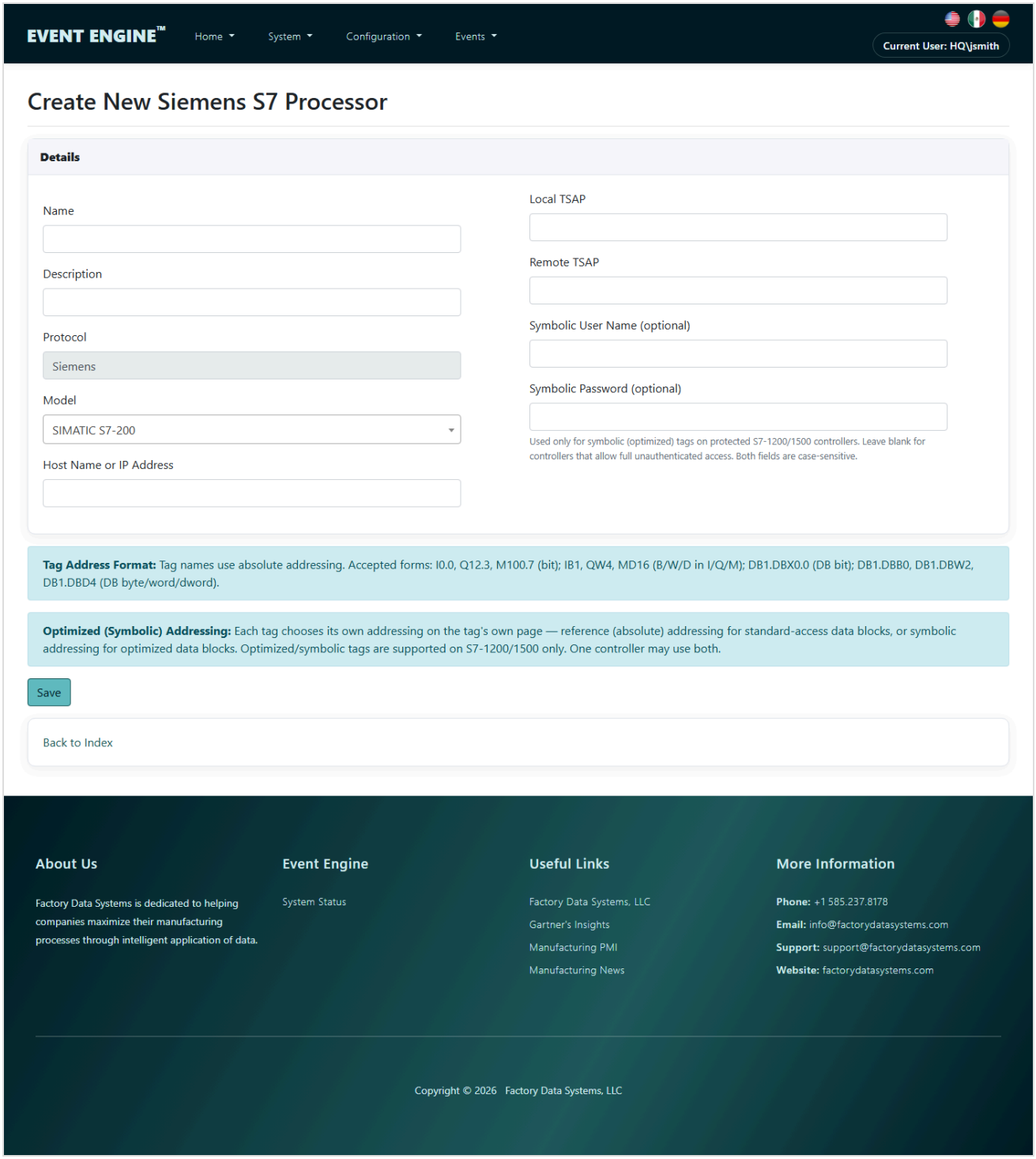
Rockwell Logix Fields



Field (as labelled on the page)	What it does
Backplane	The backplane number for rack-based controllers. Fixed at **1** and shown read-only — there is nothing to choose.
Slot	The slot the controller CPU occupies (0 – 16).
Remote Path	For processors reached by *tunneling* through an outward-facing rack (e.g., a "CLX Gateway"): provide only the routing path **after** the initial outward-facing rack. Leave blank for directly connected processors.

> **There is no per-processor connection count to tune.** Event Engine performs one read or write batch against a processor at a time and batches each event's tags into as few protocol operations as the driver allows; parallelism comes from events on *different* processors running at once. See 6.3.

Siemens S7 Fields



Field	What it does
Port	TCP port (S7 communication is normally 102).

Field	What it does
Rack / Slot	CPU rack and slot — model dependent (e.g., S7-300 commonly rack 0 slot 2; S7-1200/1500 rack 0 slot 1).
Local TSAP / Remote TSAP	TSAP values for models that connect via TSAP addressing (e.g., S7-200 family). Only the fields relevant to the selected model are shown.
Symbolic User Name / Symbolic Password	Optional credentials, used **only** by tags that use symbolic addressing (below) against a protected S7-1200/1500 controller. Both are case-sensitive. Leave them blank for a controller that allows full unauthenticated access; absolute addressing ignores them.

Absolute and symbolic addressing

> **Symbolic (optimized) addressing is not available in this release.** The **Use symbolic (optimized) addressing** checkbox on the tag's page is shown unchecked and disabled, and every Siemens tag uses reference (absolute) addressing. Symbolic addressing is served by a separate Siemens driver that carries its own runtime license, which this release does not include. The rest of this section describes both modes, because the choice returns when that driver ships.

The addressing mode is chosen per tag, not per processor. Each Siemens tag carries its own **Use symbolic (optimized) addressing** checkbox on the tag's page (Chapter 9), and Event Engine routes that tag to the driver its setting selects. One processor can hold tags of both kinds at once — which is what the controller looks like anyway, since `Optimized block access` is a property of each individual data block in TIA Portal, not of the CPU.

	Reference (absolute) — box unchecked	**Symbolic (optimized) — box checked**
The tag's **Name** is	an absolute S7 address (`DB1.DBW2` , `M100.7`)	the symbolic name from your TIA program (`"HB_DB".eventControl.Speed`)
Controllers	S7-200 / 300 / 400 / 1200 / 1500	**S7-1200 / 1500 only**
Required in TIA Portal	`Optimized block access` **unchecked** on the data block (block properties), and **PUT/GET communication permitted** on S7-1200/1500 CPUs	nothing — the data block may keep its default `Optimized block access` setting; supply the symbolic credentials above if the controller is protected
Data types	all of them, including UDT / Structure	all of them, including UDT / Structure
Available in this release	yes	**no** — see the note above

Each mode is held to its own rules when a tag is saved. An absolute address must match the syntax described below, and a symbolic form (`"DBName".Member`) entered on a tag that is *not* using symbolic

addressing is rejected at that point. A symbolic name is sent to the controller exactly as you type it — Event Engine only trims stray whitespace and drops quotation marks the controller does not need — so keep the quoting your TIA program uses. A symbolic name the controller cannot resolve is reported against that tag when scanning starts, rather than being left to return bad data.

An existing configuration that already contains symbolic tags still imports and loads unchanged — the restriction is on choosing the mode, not on holding it.

The processor page shows the accepted **absolute** address formats in a help box, together with a reminder that the mode is picked per tag — useful because Siemens tags are created by address or symbol rather than by browsing. Accepted absolute forms include memory-area bits (`I0.0` , `Q12.3` , `M100.7` , `V10.2`), memory-area byte/word/dword registers (`IB1` , `QW4` , `MD16` , `VD20` , `SMB0` , `PIW2`), data-block addresses (`DB1.DBX0.0` for bits, `DB1.DBB0` / `DB1.DBW2` / `DB1.DBD4` for byte/word/dword), and S7-200 timer/counter current values (`T3` , `C5`). **BOOL tags must use a bit address** (`M100.7` , `DB1.DBX0.0`).

Other Protocols

- **Modbus/TCP** — IP address and port; tags then specify the register/coil type and address (Chapter 9).
- **GE SRTP** — IP address plus model selection.
- **AB Legacy** — IP address plus model selection; tags use native addresses (e.g., `N7:0` , `F8:3`).
- **OPC UA** — host name, port, and path for the OPC UA server, which together form the endpoint `opc.tcp://host:port/path` , plus the user identity (anonymous, or user name and password) and the security options described below. Tags are then browsed/addressed through the server's namespace. Event Engine is an OPC UA **client** only: it does not act as an OPC server, and it does not speak OPC Classic (OPC **DA**). A DA server has to be fronted by a DA-to-UA gateway, which Event Engine then connects to as an ordinary UA server.

OPC UA Certificates and Trust

An OPC UA client is not merely a socket: it is an **application** with a certificate of its own. When Event Engine and an OPC server open a *secured* endpoint (`Sign` or `SignAndEncrypt`) they exchange certificates, and there are **two independent trust decisions** — one on each side:

Direction	Who decides	Where it is configured
The server trusts **Event Engine's** client certificate	The OPC server's administrator	On the OPC server, in its own certificate store. Event Engine cannot grant this to itself.
Event Engine trusts **the server's** certificate	You	In Event Engine's certificate store, or by the **Auto Accept Untrusted Certificates** option on the processor.

Getting one and not the other is the usual cause of a secured OPC processor that will not connect.

> **Unsecured connections exchange no certificates at all.** A processor whose server offers an unsecured endpoint, and which has both **Use Only Secure Endpoints** and **Use High Level Endpoint** unchecked (the defaults), negotiates an unsecured channel — none of this section applies to it.

Event Engine's own certificate

Event Engine creates its client certificate automatically the first time an OPC processor tries to connect (or the tag browser is first opened). Nothing has to be generated or installed by hand. On an OPC server it identifies itself as:

Subject / issued to	`CN=Event Engine OPC UA Client`
Application URI	`urn:<hostname>:FactoryDataSystems:EventEngine`, where `<hostname>` is the Windows host Event Engine runs on
Application name	`Event Engine OPC UA Client`

The application URI carries the host name on purpose: it identifies **one installation**. Several Event Engine servers talking to the same OPC server are told apart by it, and each is trusted separately. It also means that **renaming the Event Engine host changes the identity** — the existing certificate no longer matches the installation, a replacement is created on the next start, and every OPC server has to be told to trust the replacement.

To have a server trust Event Engine: start Event Engine and let one connection attempt fail. The server puts the refused certificate in its own *rejected* (or *quarantined*) store; its administrator moves it into the server's *trusted* store; then reconnect. This is a one-time step per server, and it is required after any upgrade that replaces the client certificate — see the Release Notes for the full procedure.

Where the certificates are kept

Event Engine keeps its own OPC UA certificate store — a **PKI store** — outside the program folder so that an upgrade cannot replace it:

```
C:\ProgramData\Factory Data Systems, LLC\Event Engine\OpcPki
```

It contains four sub-folders, which are the four stores OPC UA defines:

Sub-store	What is in it
`own`	Event Engine's own client certificate **and its private key**. Created on the first connection attempt and reused afterwards. This is the certificate your OPC servers are asked to trust.
`trusted`	The server certificates Event Engine accepts. Putting a server's certificate here is what permanently trusts that server.

Sub-store	What is in it
`issuer`	Certificate-authority certificates needed to complete a chain, for servers whose certificate is issued by a CA rather than self-signed. Not needed for self-signed server certificates.
`rejected`	Server certificates Event Engine refused, copied here so you can inspect one and then decide to trust it. Nothing in this folder is trusted.

> **Write access to this folder is restricted to SYSTEM and Administrators** by the installer, and should stay that way: whatever is in `trusted` decides which OPC servers Event Engine will talk to, so anyone who can write there can redirect a processor to a server of their choosing.

Two practical consequences:

- **Include the folder in your file-level backup** of `C:\ProgramData\Factory Data Systems, LLC\Event Engine`. The in-product configuration backup (Chapter 3.5) covers configuration only. Losing the store is not fatal — a new certificate is created — but every OPC server then has to trust the new one.
- **The location can be moved** with the `ApplicationSettings:Opc:PkiPath` setting in `appsettings.json` if your site keeps certificates elsewhere. The service account must be able to write to wherever it points.

Trusting a server's certificate

When Event Engine refuses a server's certificate it copies that certificate into the `rejected` folder and records the subject it refused in the log (Chapter 3.2); the processor's failure to connect appears on the Error List. To trust that server permanently, **move the certificate file from `rejected` into `trusted`** and let the processor reconnect. If it does not pick the change up immediately, stop and start scanning.

That is the recommended approach, and it is a deliberate decision about one named server.

When not to use "Auto Accept Untrusted Certificates"

The **Auto Accept Untrusted Certificates** option on the processor's *Advanced Options* is the shortcut: with it enabled, Event Engine accepts a server certificate that failed validation instead of refusing it. It is **checked by default**, which keeps a first connection working on a plant network where certificates are not managed.

It is a real reduction in security, so understand what it does and does not do:

- It only decides whether **Event Engine accepts the server**. It has **no** influence on whether the server accepts Event Engine, so it can never substitute for the trust step described above.
- Accepting a certificate this way is **per connection**. Event Engine never writes an auto-accepted certificate into `trusted`, so accepting one connection does not silently trust that server forever — and does not hide the fact that the certificate is untrusted.

- Every acceptance is logged as a warning naming the certificate's subject, so the log shows what has been accepted.

Turn it off — and populate `trusted` explicitly instead — when:

- the connection carries recipes, production commands, or anything else whose alteration matters;
- a user name and password are configured on the processor, since those credentials are sent over the channel to whichever server answered;
- the OPC server is reached across a network segment you do not control, or through routing or address translation that could put something else at that address;
- your site's security policy requires that a peer be identified before it is trusted — accepting an unvalidated certificate means the connection is encrypted but the other end is unverified.

With the option off, a server whose certificate is not in `trusted` is refused, the processor's failure to connect appears on the Error List, and the log names both the certificate that was refused and the store to put it in.

The other security options

These sit alongside **Auto Accept Untrusted Certificates** under *Advanced Options*, and together they decide which endpoint a processor negotiates:

Option	Default	Effect
Use Only Secure Endpoints	unchecked	Refuses unsecured endpoints and connects on the **strongest** secured endpoint the server offers. If the server offers no secured endpoint, the connection fails with a message saying so rather than quietly falling back.
Use High Level Endpoint	unchecked	Prefers the strongest endpoint the server offers, without excluding unsecured ones.
Auto Accept Untrusted Certificates	**checked**	Accepts a server certificate that failed validation, per connection. See above.
Verify Server Certificate	unchecked	Also requires the server's certificate to match the host name the processor connects to. Enable it for managed certificates; leave it off where a server's certificate names something other than the address you use, which is common on plant networks.
Auto Upgrade Endpoint Policy	unchecked	Retained for compatibility with existing configurations. It has **no effect** on endpoint selection.

With both selection options unchecked, a processor connects on the **least** secured endpoint the server offers — an unsecured one where that is available, which is why a default configuration exchanges no certificates.

If a secured processor will not connect

What you see	What it means
A connect failure on the Error List naming `BadCertificateUntrusted`, `BadSecurityChecksFailed` or `BadCertificateChainIncomplete`	Almost always **the server refusing Event Engine's certificate**. Check the server's rejected store and its own event log. This does not clear on retry.
Rejected OPC UA server certificate … in the log, naming a subject and the `trusted` folder	Event Engine refusing the server. Move the certificate from `rejected` to `trusted`, or reconsider **Auto Accept Untrusted Certificates**.
the server offers no endpoint that matches this processor's security settings	**Use Only Secure Endpoints** is checked but the server publishes only unsecured endpoints. Configure a secured endpoint on the server, or clear the option.
The OPC UA client certificate could not be created or loaded from …	The service account cannot write to the PKI folder, or the host was renamed and the stored certificate no longer matches this installation. The message names the folder.
Accepting untrusted OPC UA server certificate … as a warning in the log	The connection succeeded only because **Auto Accept Untrusted Certificates** is enabled. The channel is encrypted but the server was not verified — see above for whether that is acceptable here.

OPC Server Gateways (e.g., KEPServerEX in front of a PLC)

When the OPC server is a gateway to a real controller (for example, a KEPServerEX channel using the *Allen-Bradley ControlLogix Ethernet* driver), every Event Engine read and write becomes a controller transaction performed by the server on Event Engine's behalf.

> **The performance of an OPC processor is set by the OPC server's configuration, not by Event Engine.** If OPC events are slow, the tuning that fixes them is almost always in the OPC server, not in Event Engine. This section explains why, and how to confirm it before spending time in the wrong place.

Why Event Engine cannot tune this for you

OPC UA is a protocol between a *client* and a *server*. It defines how a client asks for values and how the server answers. It defines **nothing** about how the server then talks to the controller behind it — there is no OPC UA service meaning "batch your device requests," "use a larger CIP connection," or "write that structure in one transaction."

That downstream behavior is entirely the OPC server vendor's own configuration, reachable only through the vendor's tooling. Two OPC servers in front of the same PLC, given the identical request from Event Engine, can differ by more than an order of magnitude purely because of how each is configured.

What Event Engine already does

Event Engine sends the most efficient request the protocol allows:

- **Writes are fully batched.** An entire event write-back — result-set values, output parameters, completion codes and status messages together — is accumulated and sent as **one** OPC UA `WriteNodes` service call, no matter how many tags it covers. There is no per-tag call.
- **Reads are batched** the same way, one `ReadNodes` call per device for all tags an event needs.
- **Connection handling is failure-only.** A healthy read or write performs no connection probe; the session is checked and re-established only after an operation actually fails, and the batch is retried once.
- **Every call is bounded**, so a stalled server cannot hang the engine indefinitely.

Once that single request reaches the server, what the server does with it is out of Event Engine's hands. A server that turns one 850-value write into 850 individual controller transactions will be slow, and no client-side change can prevent it.

Why an OPC processor has far more tags than a native driver

A native driver (Allen-Bradley Logix, Siemens S7) can carry a **compiled UDT**: a whole structure array is a *single* tag that the driver writes to the controller in one native operation.

OPC processors cannot use compiled UDTs. The same structure is modelled as nested Collections, which produce **one tag per leaf member**. The same physical data therefore looks very different to the two processors:

Same 50-element recipe array, 17 members per element	Native Logix processor	OPC processor
Tags configured	**1** (compiled UDT, 50 elements)	**850** leaf tags (+ 51 container tags)
Values sent per write-back	1 array write	850 values in one `WriteNodes` call
Typical measured duration	~100–180 ms	depends entirely on server tuning (see below)

This is expected and is not a fault. It does mean an OPC write-back asks the server to move far more individually-addressed values, which is exactly the workload that server-side batching settings exist to optimize.

Confirming it is the server

Use the *Duration* and *Write Back Time* columns on Event History. When an OPC event is slow, `Write Back Time` is usually nearly the whole `Duration`, while `Tag Collection Time` and `Work Time` stay small.

Divide `Write Back Time` by the number of values written to get a **per-value cost**:

Per-value cost	Interpretation
Under ~0.5 ms	The server is batching to the controller properly.
A few ms	The server is performing roughly one controller transaction per value. Tune the server.
Tens of ms	Per-value transactions against a cold or busy server. Tune the server; check its own load.

To confirm the ceiling is the server's and not Event Engine's, write the same values with the OPC server vendor's own client tool. If the vendor's tool is no faster, Event Engine is already sending the optimal request and the remaining cost is server-side.

KEPServerEX with a ControlLogix device

The defaults are usually too conservative for event-sized write batches. These settings have proven necessary:

Setting	Where	Default	Recommended	Why
Protocol Mode	Device → *Logix Options*	Logical	**Symbolic**	Logical addressing fails on AOI-member tags (CIP privilege/type errors); symbolic behaves like RSLinx.
CIP Connection Size	Device → *Logix Communication Parameters*	500 bytes	**4000 bytes**	Fewer packets per read/write batch (firmware 20+).
Request Timeout	Device → *Timing*	1000 ms	**2000 ms+**	Large write batches need more than a single fast round trip.
Inter-Request Delay	Device → *Timing*	0 ms	**0 ms**	Any delay is multiplied by the number of transactions.
Write Optimizations → Duty Cycle	**Channel**	10	**10** (the maximum)	Caps how fast a queue of pending writes drains. Verify rather than change.

> **Do not go looking for *Allow Function File Block Writes* or *Request Size*.** Both live under the device's *ENI DF1/DH+/CN Gtwy Communication Parameters* group, which applies only to ENI / DF1 / DH+ / ControlNet Gateway models — for a **ControlLogix 5500** device they are greyed out and inapplicable (function files are a PLC-5/SLC concept). Enabling them does nothing for Logix UDT or array writes, and the Configuration API will silently refuse them. The name is misleading: "block writes" here is not batching.

The properties above are the whole of the ControlLogix driver's write tuning. Once Protocol Mode is Symbolic (required for AOI-member tags), Connection Size is at 4000, Inter-Request Delay is 0 and Duty Cycle is at its maximum, **there is nothing further to tune server-side** — the remaining cost is one CIP transaction per tag written, which is inherent to symbolic addressing of individual structure members. See *Confirming it is the server* above for how to measure the per-value cost, and expect a few milliseconds per value at that point.

> **A device property cannot be changed while a client is using it.** KEPServerEX refuses property changes on a device an OPC client currently references, and the Configuration API reports this as `not_applied` with "*active client reference or property is disallowed/disabled/read-only*" — the request still returns HTTP 200, so the setting silently stays as it was. **Stop the Event Engine scan first**, apply the change, then start the scan again, and re-read the property to confirm it took. The same applies when changing the setting in the KEPServerEX user interface.
>
> Event Engine is rarely the only client. KEPServerEX's own **OPC UA Gateway** service connects to the local server as a client (`Created session with downstream server` in the event log) and re-attaches within seconds of every runtime restart, so restarting Kepware never clears it — stop that service too if nothing consumes its endpoint. The server's *Event Log* names the blocking device: "*One or more changes were not applied to '<device>' since it is being referenced by a client.*"

String tags also need gateway-specific addressing: Kepware cannot serve a Logix `STRING` structure directly — address the character member with an explicit length instead, e.g. `MyTag.localRecipeName.DATA/82` (use the actual string type's length: `/20` for `STRING_020`, `/100` for `STRING_100`, and so on).

Other OPC servers

The setting names differ, but look for the same four things in any OPC server that fronts a controller:

1. **Block / bulk transfer of structures and arrays** — the single highest-impact setting for event write-backs.
2. **Transaction or packet size** — how many values fit in one request to the controller.
3. **Any per-request delay or duty cycle** — a small delay multiplied by hundreds of values dominates everything else.

4. **Server-side scan or cache mode for the tags Event Engine uses** — see *Value Freshness on OPC Reads* below.

> **Address-space changes:** if channels, devices, or tags are rebuilt on the OPC server while Event Engine is connected, the server refuses to modify objects a client is using. Event Engine detects nodes that have gone missing and re-resolves them automatically, but the cleanest sequence is still to stop the Event Engine scan before restructuring the OPC server's namespace, then start it again.
>
> Re-resolution retries quickly at first and then backs off to once every five minutes, and it never stops: a tag whose node is still missing after about four minutes is reported on the Error List, and goes on being retried at that rate for as long as it stays missing. **No restart is needed** — however long the server takes to be put right, each affected tag recovers on its own within five minutes of its node being served again.
>
> That Error List entry names one of two causes — the server no longer exposes the node, or the node is there and will not serve a value — and they send you to different places. Because the retries continue, a tag can move from one to the other while it waits; the entry is rewritten when that happens, so what it names is what the most recent attempt found.

Value Freshness on OPC Reads

An OPC server keeps its own cache of tag values, refreshed by its scan of the device. When an event runs, Event Engine reads every tag that event consumes, and every one of those reads carries the OPC UA `maxAge = 0` — *read the device, do not answer me from your cache*. **Asking is not the same as being obeyed:** a server is free to answer from its cache regardless, and **whether it does is decided by the server's own configuration**.

A cached answer gives no sign of itself in the value. The timestamps do: it carries the time of the scan that produced it, not the time of the read.

If events act on values that lag the machine, the fix is in the OPC server rather than in Event Engine: for KEPServerEX, the client interface's scan mode governs this (*Do Not Scan, Demand Poll Only* polls the device only when a client reads it).

A first read after a connection is established is legitimately older than a steady-state one, since setting up the OPC session itself takes time.

> **This applies to reads, not to trigger scanning.** A trigger fires from a change the OPC server pushes to Event Engine, which reports when the value last *changed* — a tag that legitimately holds the same value for an hour is not stale. Note that the freshness of a trigger is still bounded by the OPC server's own scan rate: Event Engine cannot be notified of a change sooner than the server notices it.

6.3 How Processors Affect the Running System

- When the system starts, Event Engine connects to every processor that has scannable tags and builds optimized tag groups. A processor that cannot be reached logs errors and its events will fail until communications are restored.
- **A processor serves one read or write batch at a time.** Each processor has an access gate that an event's tag collection or write-back holds for the duration of that batch, so events sharing a processor take turns on it while events on *different* processors run in parallel. Nothing about this is configurable. When the queuing matters, the run's Event History detail reports it against the slowest device read or write as "… *waiting for the device*" (Chapter 14.3) — that, not a connection setting, is where you look when tag collection is slow on a busy processor.
- Deleting a processor removes its tags from scanning; the UI prevents deleting a processor whose tags are used by events until those references are removed.

> **Editing while running:** configuration changes take full effect when the system is restarted (stopped and started from the System Status page). The header shows a warning icon when a change requires a refresh or restart.

Next: Chapter 7 — Triggers · Guide home

Chapter 7 — Configuration: Triggers

Configuration → Triggers

> **Where this fits: Step 4** of the setup roadmap. Build your trigger templates **before** the tags (Ch. 9) and events (Ch. 10–13) that use them, so the trigger dropdowns are already populated when you wire them up.

7.1 What is a Trigger?

A **trigger** defines *when* events execute. In Event Engine, triggers are built as reusable **Trigger Templates**: you define the timing/condition rules once, then apply the template to as many tags (or events) as you like. This keeps large systems consistent — change the template, and every trigger that uses it follows.

There are two trigger types you can create:

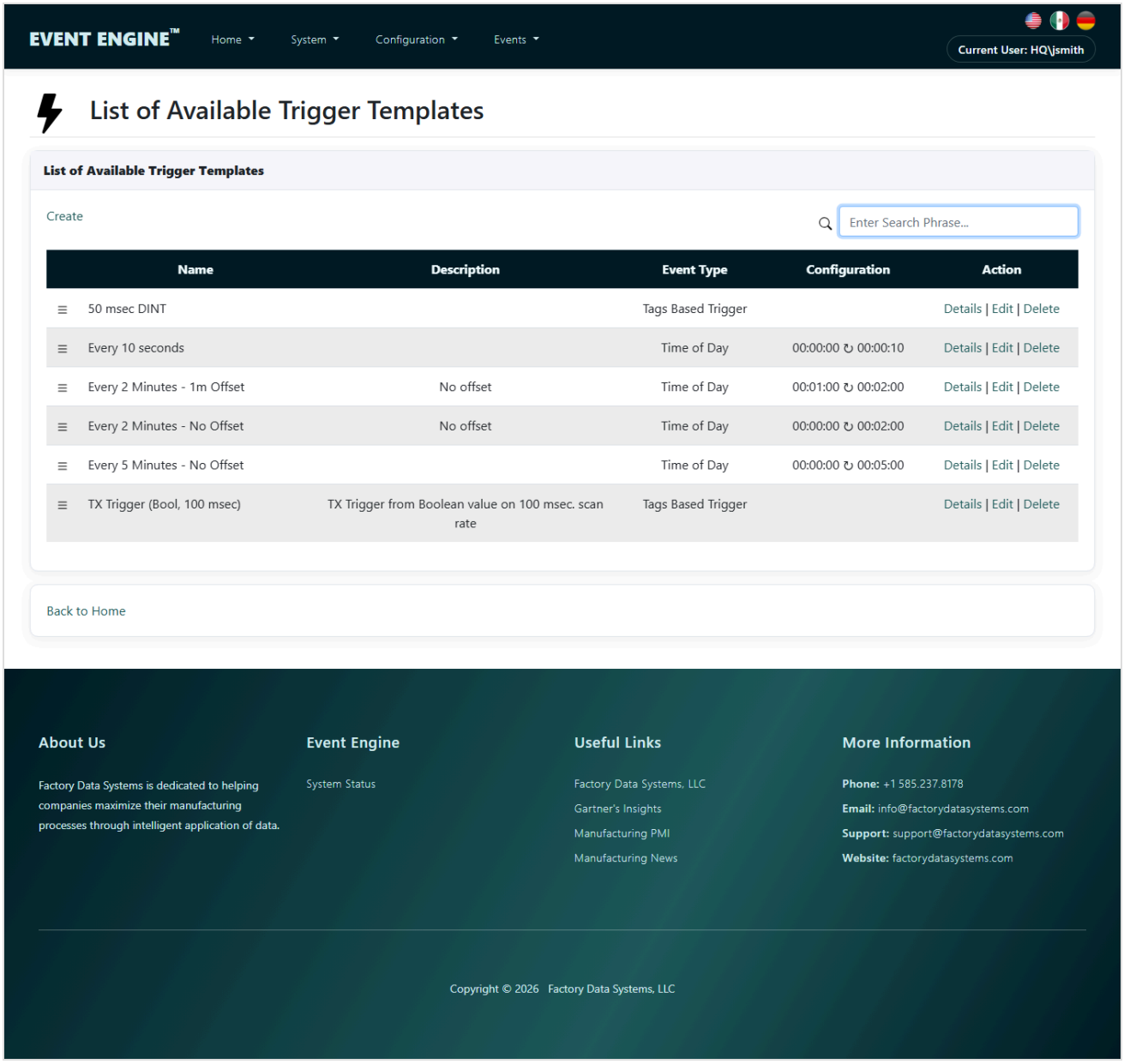
Type	Initiates events based on…	Typical uses
Tag (Data Block) Based Trigger	Condition monitoring of a tag's value — change detection, thresholds, rising edges	"Part complete" handshake bits, counters, process-value logging
Time Based Trigger	A prescribed *time of day* (and/or system startup) with an optional repeat interval	Shift-end reports, hourly summaries, scheduled database jobs

How a trigger reaches an event differs by type:

- A **tag trigger template** is assigned to a *tag* (making it a **trigger tag**). Events then list that tag in their **Trigger Tag(s)** selection. When the tag's trigger condition is met, every event subscribed to it fires.
- A **time-based (TOD) template** is attached *directly to events* via the **Time Based Triggers** selection on each event's Event Control card.

7.2 The Trigger Template Index

The index lists every template in the system with its type and where it is used. Editing a template shows the warning: "*Changes here globally affect all triggers using this template*" — and changes take effect globally the next time the system is loaded. (Individual tags can be updated immediately by re-saving the affected tag.)



Click **Create** to choose the trigger type:

Create a New Event Trigger

Select the type of trigger you would like to create from the different trigger types.

Create

Tag (Data Block) Based Trigger

Initiate events based on tag condition monitoring and detection.

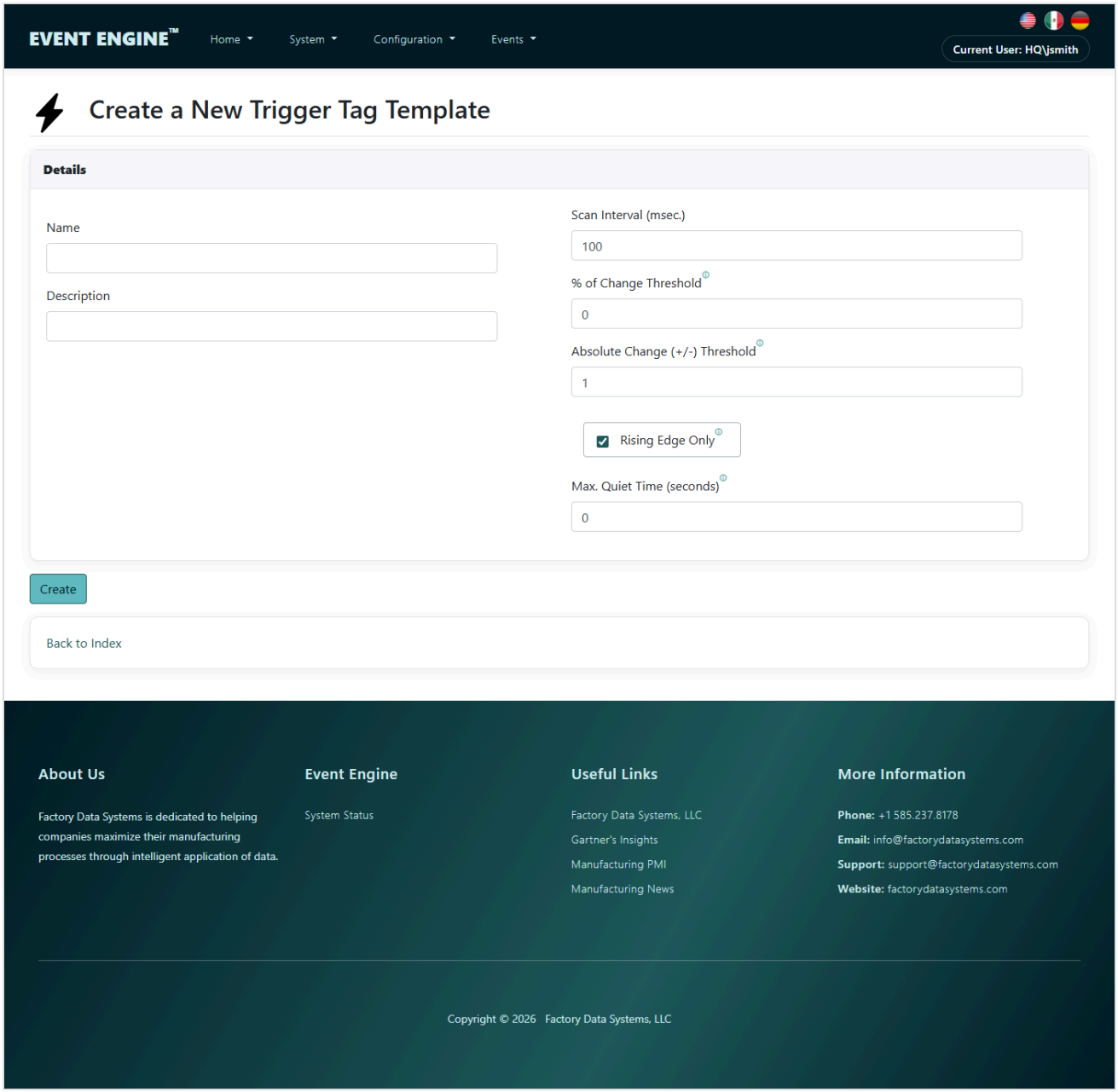
Time Based Triggers

Time based trigger events fire at a prescribed *Time of Day* on the selected days of the week, and can then repeat at a specified interval until an end time.

Back to Index

Copyright © 2026 Factory Data Systems, LLC

7.3 Tag (Data Block) Based Trigger Templates



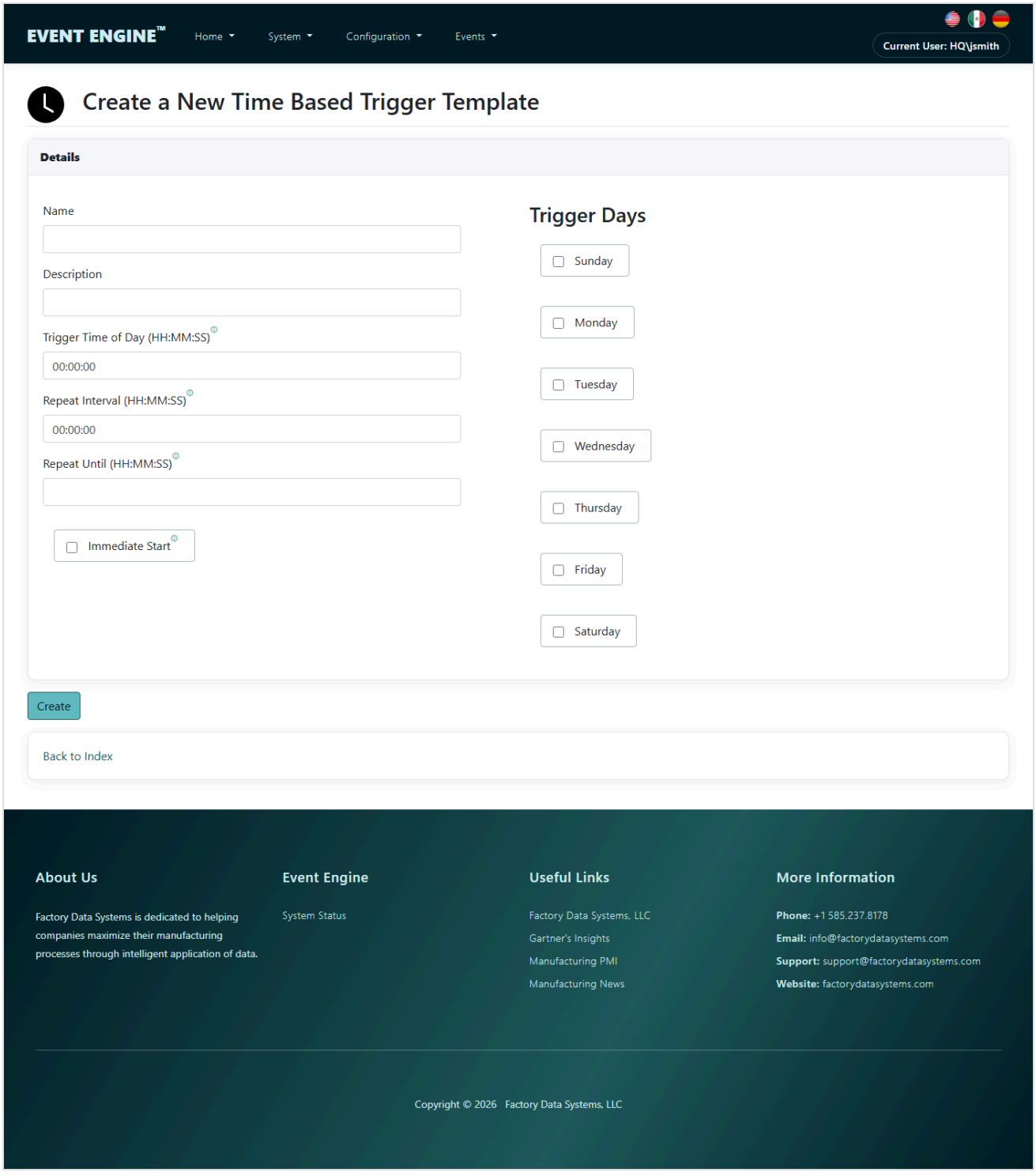
Field	What it does and how it affects operation
Name / Description	Identify the template. The display name (`Name (Description)`) is what appears in the tag editor's Trigger Template dropdown.
Scan Interval (msec.)	How often Event Engine reads the trigger tag from the controller (**10 – 10000 ms**). This is the heartbeat of the trigger: a 100 ms scan detects changes within ~100 ms but creates 10 reads/second of comms load per tag group. Choose the slowest interval that meets your latency requirement.

Field	What it does and how it affects operation
% of Change Threshold	The *percent* change since the last logged value that must be seen before triggering again. Operates independently of the other mechanisms. Set to **0 to disable** percent-change triggering.
Absolute Change (+/-) Threshold	The *absolute* change in value since the last logged value required to trigger. Also independent of the other mechanisms. Set to **0 to disable** value-change triggering.
Rising Edge Only	When set, only a *rising* edge initiates the action: a Boolean going 0→1, or a numeric value increasing. Falling edges / decreasing values are ignored. Use this for classic handshake bits so the event fires once when the PLC sets the bit, and not again when it clears.
Max. Quiet Time (seconds)	The maximum time the trigger will go without firing when no change threshold is crossed — a "heartbeat" trigger. Set to **0 to disable** quiet-time triggering.

> **How the mechanisms combine:** each enabled mechanism (percent change, absolute change, quiet time) operates independently — whichever fires first causes the trigger. For pure event handshakes the common pattern is *Rising Edge Only* with both change thresholds at their defaults.

7.4 Time Based Trigger Templates

Time-based triggers start at a prescribed *time of day* (or at system startup) and can then repeat at a specified interval.



Field	What it does and how it affects operation
Name / Description	Identify the template.

Field	What it does and how it affects operation
Trigger Time of Day (HH:MM:SS)	The time of day the trigger fires on each enabled day. **Required** — the template will not save without a valid time. This is also the *anchor* the repeat interval counts from (below).
Repeat Interval (HH:MM:SS)	After the trigger time, the trigger fires again at each interval until **Repeat Until** (or the end of the day). Repeats are **aligned to the trigger time**, so a system started mid-day joins the existing cadence at the next aligned moment rather than starting a fresh series from the moment it booted. Leave at **00:00:00 to disable** repeating — the trigger then fires once at the trigger time on each enabled day.
Repeat Until (HH:MM:SS)	The last time of day at which the repeat interval may fire. Leave **blank to repeat until the end of the day**. Requires a repeat interval, and may not be earlier than the trigger time; either mistake is rejected on save.
Trigger Days (Sunday – Saturday)	The days of the week the trigger may occur. **At least one day is required.**
Immediate Start	Fires the trigger **once when the system starts**, in addition to the normal schedule. This is a real event execution, not just a timer start — use it for "every N minutes" jobs that should produce a result as soon as the system is running rather than waiting for the next aligned repeat.

> **The schedule is anchored, not relative.** A template with trigger time 08:00:00 and a one-hour interval fires at 08:00, 09:00, 10:00 … regardless of when the system started. Starting the service at 09:20 joins that series at 10:00 — it does not begin a new hourly series at 10:20. **Immediate Start** is what you add when you also want a run at 09:20.

Daylight saving time

Trigger times are **local wall-clock times**, so on the two days a year the clock changes there are times that either never happen or happen twice. The engine resolves both so that a trigger fires **exactly once on each enabled day — never zero times, never twice.**

Situation	What the trigger does
Clocks go forward and the trigger time falls in the skipped hour (for example 02:30 where 02:00 jumps to 03:00)	Fires an hour later on the clock — at 03:30. The whole skipped hour moves across intact, so a 02:00 collect and the 02:30 report that depends on it stay 30 minutes apart instead of both firing the moment the clock jumps.
Clocks go back and the trigger time falls in the repeated hour (for example 01:30 where 02:00 returns to 01:00)	Fires the **first** time the clock reads that time, and does not fire again when the hour repeats. A daily trigger therefore stays 24 hours from its previous run, not 25 — which matters for any event whose query covers "since the last run".

A **repeat interval** is a duration, not a time of day, so it is unaffected: an hourly repeat is an hour of real time either way. On the day the clocks go back that means a repeating trigger fires twice at the same wall-clock reading, because that hour genuinely happens twice; on the day they go forward it produces no firing at all inside the skipped hour, because that hour does not exist. A **Repeat Until** time inside the repeated hour ends the series at its *first* occurrence, so the window keeps its normal length rather than gaining an hour.

If a schedule must be immune to all of this — for a site coordinating with a plant in another country, say — set the server's time zone to one without daylight saving.

Examples

Goal	Configuration
End-of-shift report at 7:00 AM and 7:00 PM weekdays	Two templates (07:00:00 and 19:00:00), Monday–Friday checked, interval 00:00:00
Database sync every 5 minutes, around the clock	Time 00:00:00, all days checked, interval 00:05:00, Repeat Until blank, **Immediate Start** checked
Hourly summary during the day shift only	Time 06:00:00, interval 01:00:00, **Repeat Until 14:00:00**, days as needed

7.5 Applying Triggers

- **Tag triggers:** open the tag (Chapter 9), pick the template under **Trigger Template**, save. The tag becomes a *trigger tag*, and it will appear in the **Trigger Tag(s)** dropdown of every event editor. Trigger members can also be designated inside Collections (Chapter 8) so that every instance of a structure has its trigger member pre-wired.
- **Time-based triggers:** open the event and select the template under **Time Based Triggers** on the Event Control card.

> **Deleting a template** that tags or events still use is blocked until those references are removed — the details page shows you exactly which objects use the template.

Next: Chapter 8 — UDTs & Collections · Guide home

Chapter 8 — UDTs & Collections

Configuration → UDTs and **Configuration → Collections**

> **Where this fits: Step 5** of the setup roadmap, and needed only when your tags are *structures*. Define the structure here **before** creating the structure tags (Ch. 9) that instantiate it. *(Skip this chapter if you use only atomic tags.)*

Event Engine's biggest departure from traditional middleware is that it works with **structures**, not just individual tags. Two mechanisms define structures:

	UDTs	**Collections**
What they are	Pre-compiled structure definitions, programmed in C# and loaded as plugins — most commonly **imported directly from PLC export files**	Structure *templates* you build by hand in the web UI
Where they come from	Imported from Rockwell `.L5X` exports or Siemens `.scl` / `.stl` / `.db` / `.udt` sources (or shipped with the product)	Created interactively on the Collections pages
Organized by	**Namespaces** ("folder names" that keep same-named UDTs from different programs separate)	Communications protocol
Editable in the UI?	No — re-import to change (definitions mirror the PLC program)	Yes — full create/edit/delete of data members
Best for	Mirroring the exact UDTs engineered in your PLC program	Custom groupings, protocols without UDT import, quick prototypes

Both serve the same purpose downstream: when you create a **tag** whose data type is a structure, you pick a UDT or a Collection as its type, and Event Engine creates the full member tree for you.

8.1 UDTs

The UDT Index

Configuration → UDTs lists every UDT known to the system — both natively programmed UDTs and ones you've imported — with name, namespace, and description. Search covers all three. The details page shows the UDT's **members** (name and data type) and its properties, including its total size in bytes.

EVENT ENGINE™ Home System Configuration Events

Current User: HQ\jsmith

UDT List

UDT List

Import UDT(s)

Enter Search Phrase...

Namespace	Name	Description	Action
IntelliTrack	STRING_035		Details
IntelliTrack	STRING_100		Details
IntelliTrack	udt_Genealogy_v1	Used for Building Genealogines	Details
IntelliTrack	udt_KeyValuePair	Key Value Pair Associated Data	Details
IntelliTrack	udt_LogValue_v1	UDT for Logging Values Back to IntelliTrack at Check-In	Details
IntelliTrack	udt_ValueType_v1	Holds Attribute Value for Recipes and Data Logging	Details
IntelliTrack	Value_Type	Imported Siemens type Value_Type	Details
IntelliTrack___Siemens	udt_LogValue_v1	Imported Siemens type udt_LogValue_v1	Details
IntelliTrack___Siemens	udt_ValueType_v1	Imported Siemens type udt_ValueType_v1	Details
IntelliTrack___TEST	Carrier_StackData	WC5 Carrier Stack Data Table Read/Write	Details
IntelliTrack___TEST	p_cursor_type	For Oracle Testing	Details
IntelliTrack___TEST	p_detail_type		Details
IntelliTrack___TEST	p_items_type		Details
IntelliTrack___TEST	p_recent_type	UDT Structure for Oracle Testing	Details
IntelliTrack___TEST	p_summary_type		Details
IntelliTrack___TEST	STRING_002		Details
IntelliTrack___TEST	STRING_003		Details
IntelliTrack___TEST	STRING_010		Details
IntelliTrack___TEST	STRING_020		Details
IntelliTrack___TEST	STRING_035		Details
IntelliTrack___TEST	STRING_100		Details
IntelliTrack___TEST	STRING_200		Details
IntelliTrack___TEST	udt_Alarm	Information for a Single Alarm Event	Details
IntelliTrack___TEST	udt_Genealogy_v1	Used for Building Genealogines	Details
IntelliTrack___TEST	udt_KeyValuePair	Key Value Pair Associated Data	Details
IntelliTrack___TEST	udt_LogValue_v1	UDT for Logging Values Back to IntelliTrack at Check-In	Details
IntelliTrack___TEST	udt_QueueItem_v1	Describes Item in an Equipment or Process Queue	Details
IntelliTrack___TEST	udt_Recipe_100	Complete recipe related UDT holding all recipe limits, values, and set-points. Suffix == Attribute Qty.	Details
IntelliTrack___TEST	udt_Recipe_25	Complete recipe related UDT holding all recipe limits, values, and set-points. Suffix == Attribute Qty.	Details
IntelliTrack___TEST	udt_Recipe_50	Complete recipe related UDT holding all recipe limits, values, and set-points. Suffix == Attribute Qty.	Details

≡	IntelliTrack___TEST	udt_ValueType_v1	Holds Attribute Value for Recipes and Data Logging	Details
≡	IntelliTrack___TEST	udt_WorkOrder_v1	The Work Order Information	Details
≡	oracle	p_cursor_type	For Oracle Testing	Details
≡	oracle	p_detail_type		Details
≡	oracle	p_items_type		Details
≡	oracle	p_recent_type	UDT Structure for Oracle Testing	Details
≡	oracle	STRING_035		Details
≡	oracle	STRING_100		Details
≡	Siemens	Value_Type	Imported Siemens type Value_Type	Details
≡	SmartTrack	udt_BinaryAttribute	Attribute Holding Container for Binary Data	Details
≡	SmartTrack	udt_BinaryAttribute_v1	Attribute Holding Container for Binary Data	Details
≡	SmartTrack	udt_CurveAttribute	Attribute Holding Container for Curve Definition	Details
≡	SmartTrack	udt_CurveAttribute_v1	Attribute Holding Container for Curve Definition	Details
≡	SmartTrack	udt_ETRRequest_v1	Used for Requesting ETR Values from the Db	Details
≡	SmartTrack	udt_ETRRequest_v2	Used for Requesting ETR Values from the Db	Details
≡	SmartTrack	udt_Genealogy	Used for Building Genealogines	Details
≡	SmartTrack	udt_Genealogy_v1	Used for Building Genealogines	Details
≡	SmartTrack	udt_NumericAttribute	Attribute Holding Container for Numeric Data	Details
≡	SmartTrack	udt_NumericAttribute_v1	Attribute Holding Container for Numeric Data	Details
≡	SmartTrack	udt_Recipe_Large_v1	Complete recipe related UDT holding all recipe limits, values, and set-points.	Details
≡	SmartTrack	udt_Recipe_Medium_v1	Complete recipe related UDT holding all recipe limits, values, and set-points.	Details
≡	SmartTrack	udt_Recipe_Small_v1	Complete recipe related UDT holding all recipe limits, values, and set-points.	Details
≡	SmartTrack	udt_StringAttribute	Attribute Holding Container for String Data	Details
≡	SmartTrack	udt_StringAttribute_v1	Stores the data associated with a single SmartTrack string attribute or ETR value.	Details
≡	SmartTrack	udt_WorkOrder	The Work Order Information	Details
≡	SmartTrack	udt_WorkOrder_v1	The Work Order Information	Details

Back to Home

Copyright © 2026 Factory Data Systems, LLC

UDT Details

IntelliTrack.STRING_035

Details

Namespace
IntelliTrack

Name
STRING_035

Description

Communications Protocol

Size (bytes)
80

UDT Members

Name	Data Type
LEN	Int32
DATA	Byte[]

Back to Index

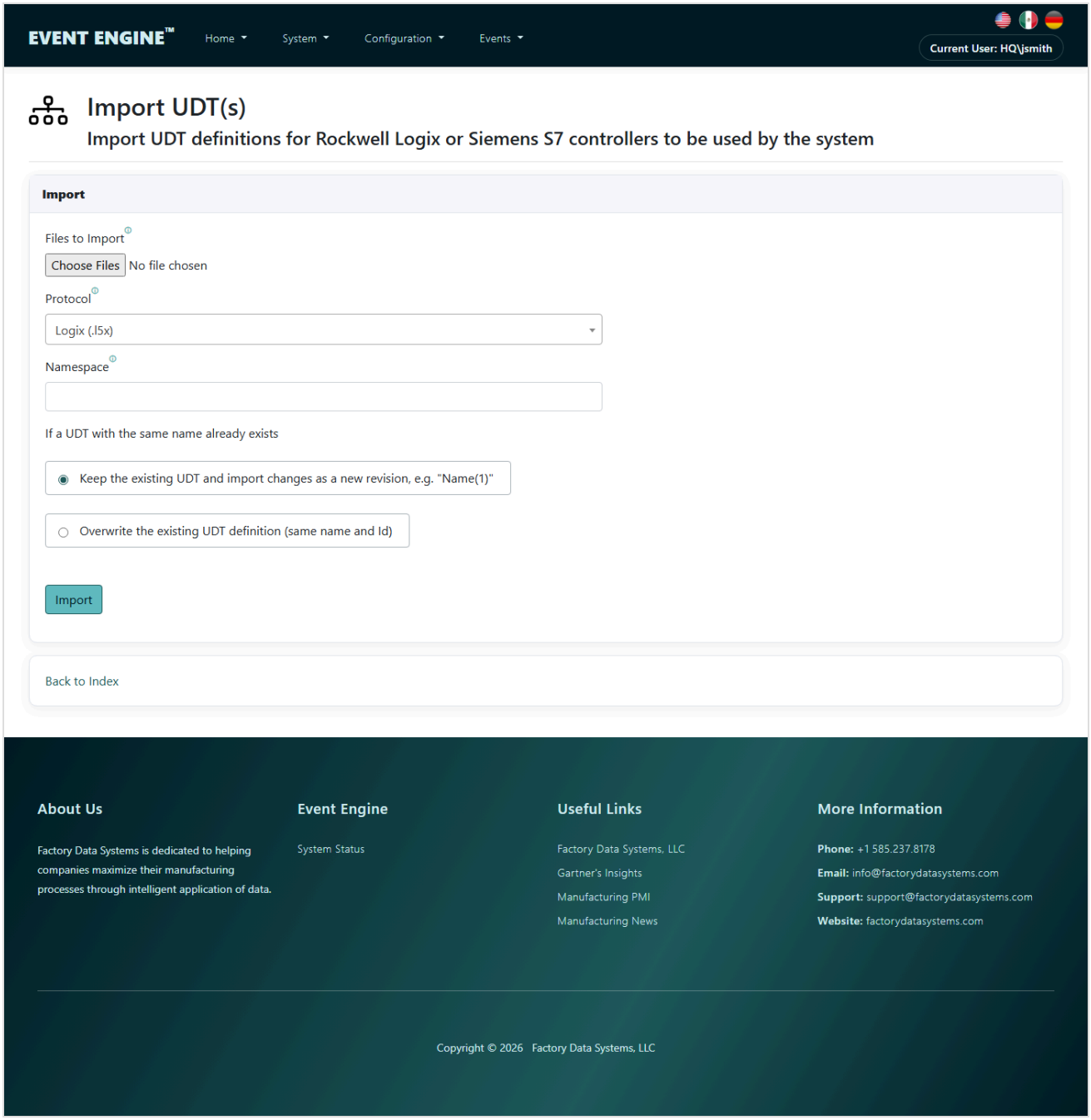
Copyright © 2026 Factory Data Systems, LLC

> **PLC type names vs. .NET type names.** The tag editor lists data types using PLC-style names (DINT, REAL, …), but a UDT's member pages show the mapped **.NET** type each one corresponds to — e.g., a Logix `DINT` member appears as `Int32` and a `REAL` as `Float32`. Only the label differs; the structure is identical, so a freshly imported UDT whose members read `Int32` / `Float32` is correct, not mis-imported. See Chapter 9.4 for the full per-protocol type list.

Importing UDTs from the PLC

Configuration → UDTs → Import *(Admins role)*

This is where Event Engine "breaks the tag boundary": you export structure definitions from your PLC programming software and import them directly.



Field	Notes
Files to Import	Select one or more files to import into the same namespace. **Logix:** `.L5X` export files (export the UDTs — or the whole program — from Studio 5000). **Siemens:** `.scl`, `.stl`, `.db`, or `.udt` source files (export from TIA Portal). Files not matching the selected protocol are ignored.
Protocol	`Logix (.l5x)` or `Siemens (.scl, .stl, .db, .udt)`.

Field	Notes
Namespace	A namespace or "folder name" that keeps these UDTs separate from other UDTs which may be different but share the same name. Use one namespace per controller program or per area, e.g., `Line1`, `PackagingV2`.
If a UDT with the same name already exists	How a *changed* definition that collides with an existing UDT of the same name (in the same namespace) is handled. **Keep the existing UDT and import changes as a new revision** (the default): the existing UDT — and every tag using it — is left untouched, and the change comes in as a new type named `Name(1)` (then `Name(2)`, …). **Overwrite the existing UDT definition**: the definition is replaced in place under the same name and identity, and **every existing tag of that type binds to the new definition**. See "Re-importing and revisions" below.

What happens on import:

1. The files are parsed and each UDT definition is converted to code and **compiled into a DLL** stored under the UDT plugin folder (`C:\ProgramData\Factory Data Systems, LLC\Event Engine\UDTs`).
2. On a first-time import, the new types are loaded immediately and appear in the UDT index and in the tag editor's data type list (as `Custom`/structure types).
3. **Re-importing into an existing namespace merges with it.** Types already imported into the namespace stay registered even when the new files don't mention them, and each type in the new files is compared against the existing definition of the same name (see "Re-importing and revisions" below). This is the supported way to push a PLC program change into Event Engine — re-export from the PLC software and re-import with the same namespace.

Re-importing and revisions

On a re-import, each UDT is matched by **name within its namespace** and its **member layout** is compared with the definition already in the system:

- **Unchanged layout** — the type silently keeps its name and identity; nothing to do. Tags, collections, and events that reference it are unaffected.
- **Changed layout** — the collision option you selected on the Import page decides:
  - **New revision (default):** the existing UDT is left exactly as it is — same name, same identity, every existing tag still bound to it — and the changed definition is imported as a new type named `Name(1)` (then `Name(2)`, and so on). You migrate tags to the revision when the PLC actually receives the new block, and retire the old type at your own pace. This is the zero-surprise option: nothing that currently works changes behind your back.
  - **Overwrite:** the definition is replaced under the existing name and identity, and **every tag of that type follows the new layout immediately**. Use this only when all PLCs using the type have already been updated to the new block layout.

> **Nested structures revise together.** If UDT `Parent` contains a member of UDT `Sub` and `Sub`'s layout changed, then `Parent`'s physical layout changed too — even though its own definition text did not. With the revision option, the system handles this automatically: `Sub` becomes `Sub(1)` **and** `Parent` becomes `Parent(1)` (whose member is a `Sub(1)`). Old and new layouts are never mixed inside one structure, and the original `Parent` / `Sub` pair remains intact for tags still pointing at PLCs with the old block. With the overwrite option both definitions update in place and no renaming occurs.

> **Updating UDTs safely:** after overwriting a changed UDT (and rebooting if prompted), review tags created from the old definition. Tags instantiate the structure as it existed when the tag was created, so re-save (or recreate) structure tags whose layout changed, and re-open dependent events so the parameter matching reflects the new member list. Revisions avoid this: existing tags stay on the untouched original type.

> **When a restart is needed — and the trade-off.** A restart is required when a definition that is *already compiled and loaded into the running process* has to be replaced — typically an Overwrite, or a same-layout re-import into a namespace that is in use. The .NET runtime cannot always unload an assembly whose types are still referenced, so the previously loaded UDT stays live until the process ends; the header shows a warning icon until the reboot is performed: "*Your UDT has been imported, but a system restart (reboot) is required for the changes to take effect.*" The trade-off is real: a restart **stops scanning and halts every event on the machine** while the service comes back up — not just the events that use the UDT you changed. Plan it for a maintenance window, and remember that triggers fired while the service is down are not captured. Note that a revision compiles into the same namespace DLL, so the restart prompt can still appear when that namespace is in use — but because a revision never *changes* a definition tags depend on, the restart is only about making the new type available, never about existing tags silently changing meaning. That safety is why the revision option is the default and the recommended way to roll out structure changes on a system that must keep running.

When part of an import doesn't compile

An upload usually holds many structures, and one bad definition no longer costs you the rest.

- **A defective structure is rejected on its own.** Every other structure in the upload is accepted and put in place, and the page names the ones that failed: "*These structure(s) … failed to compile and were NOT imported: … Every other structure in the import was accepted and is in place.*" Correct those definitions in the source and import them again.
- **Unless rejecting it would orphan something.** If a structure that *did* compile depends on one that didn't, importing the good half would leave a type referencing something absent — so the whole

import is discarded instead, and **nothing on the server is changed**: "*...the structures already in this namespace are unaffected.*"

Either way the outcome is stated at import time rather than discovered later, and a rejected import never leaves a namespace half-built.

> **Check for orphans after deleting or re-importing a UDT.** A structure *tag* whose type is removed keeps pointing at a type that no longer exists. The **Tags With Missing Structure Type** report (Chapter 9.8) lists exactly those, with the events that still reference each one.

Siemens source-file requirements

Siemens imports parse TIA Portal source text ("Generate source from blocks" — `.scl`, `.stl`, `.db`, `.udt`). The importer checks the source up front and **stops with a message naming the offending member** rather than importing something that would misbehave later:

- **Member names must be usable as identifiers** — letters, digits, and underscores, not starting with a digit. TIA allows quoted names with spaces or punctuation (`"Motor Speed"`), but Event Engine matches database result-set columns to structure members **by member name**; renaming the member behind the scenes would silently break that column matching. If the import reports a member name it cannot use, rename the member in TIA Portal and re-export. (Names that happen to collide with programming keywords — `long`, `event`, etc. — are fine and handled automatically.)
- **Every referenced structure must be in the same import.** If a DB or UDT has a member of another UDT type, include that type's definition in the uploaded files (in TIA, generate source with dependent blocks included). The one exception is the system date-time structure **DTL**, which Event Engine provides automatically. Instance-only system types (e.g., IEC timers/counters inside optimized function blocks) cannot be read via absolute addressing and are rejected.
- **Data blocks must use standard (non-optimized) access.** A Siemens Structure/UDT tag is addressed absolutely, so set `S7_Optimized_Access` off for the block in TIA Portal. (Symbolic addressing is available for atomic Siemens tags but not for structures — see Chapter 6.)
- **Type names** may contain dots or spaces (`"Motor.Data"`); the UDT keeps that display name in the index and pickers. Two type names that differ only in punctuation (e.g., `"A B"` and `"A_B"`) cannot be imported together — the import will say so; rename one in TIA Portal.

Special note — Siemens strings

For Siemens S7, string members are represented with the controller's string types (`S7 STRING` / `S7 WSTRING`), and the **Max. String Length** on a member sizes the underlying controller string — including arrays of strings, where every element is sized to the declared `STRING[n]` length. Keep these lengths matched to the PLC declaration or string reads/writes will fail. A value longer than the declared length is

hard-clipped to capacity on write (the overflow is dropped and a warning is logged — no `"..."` marker is inserted into PLC data).

8.2 Collections

The Collection Index

Configuration → **Collections** lists the collection templates in the system. Search and paging work as usual. The details page shows where a collection is used; a collection that is **in use cannot be deleted** until it has been removed from the listed objects, and the edit page warns which objects a change may affect.

EVENT ENGINE™ Home System Configuration Events

Current User: HQ\jsmith

Collection Templates Index

A listing of all the tag collection templates in the system.

Collection Templates Index

Create | Import

Enter Search Phrase...

Export Selected

Namespace	Name	Description	Action
FDS	EventControl	Event Control & Monitoring AOI - v1	Details | Edit | Delete
IntelliTrack AOI	AddProduct	IntelliTrack AOI - v1	Details | Edit | Delete
IntelliTrack AOI	CheckIn	IntelliTrack AOI - v1	Details | Edit | Delete
IntelliTrack AOI	CheckOut	IntelliTrack AOI - v1	Details | Edit | Delete
IntelliTrack AOI	Genealogy	IntelliTrack Genealogy AOI for Adoption/Consume/Split/Merge - v1	Details | Edit | Delete
IntelliTrack AOI	GetAlarmList	IntelliTrack AOI - v1	Details | Edit | Delete
IntelliTrack AOI	Heartbeat	Rockwell Logix IntelliTrack AOI - v1	Details | Edit | Delete
IntelliTrack AOI	LogOEE	IntelliTrack AOI - v1	Details | Edit | Delete
IntelliTrack AOI	Print Parameters	Print Parameters Structure	Details | Edit | Delete
IntelliTrack AOI	PrintOnDemand	IntelliTrack AOI - v1	Details | Edit | Delete
IntelliTrack AOI	Recipe_100	IntelliTrack AOI - v1, Capacity: 100 Attributes	Details | Edit | Delete
IntelliTrack AOI	Recipe_25	IntelliTrack AOI - v1, Capacity: 25 Attributes	Details | Edit | Delete
IntelliTrack AOI	Recipe_50	IntelliTrack AOI - v1, Capacity: 50 Attributes	Details | Edit | Delete
IntelliTrack AOI	RecipeMonitor_100	IntelliTrack AOI - v1, Capacity: 100 Attributes	Details | Edit | Delete
IntelliTrack AOI	RecipeMonitor_25	IntelliTrack AOI - v1, Capacity: 25 Attributes	Details | Edit | Delete
IntelliTrack AOI	RecipeMonitor_50	IntelliTrack AOI - v1, Capacity: 50 Attributes	Details | Edit | Delete
IntelliTrack AOI	UserDefinedProcedure	IntelliTrack AOI - v1, Generic Implementation - SQL Modifications not Required	Details | Edit | Delete
IntelliTrack FB	Event Control	Event Control & Monitoring FB - v1	Details | Edit | Delete
IntelliTrack FB	Heartbeat	Siemens IntelliTrack Symbolic Addressing FB - v1	Details | Edit | Delete
IntelliTrack FB	Recipe_50	Siemens IntelliTrack Symbolic Addressing FB - v1, Capacity: 50 Attributes	Details | Edit | Delete
IntelliTrack FB DA	HeartbeatFBDA	Siemens IntelliTrack Direct Addressing FB - v1	Details | Edit | Delete
IntelliTrack OPC	AddProduct	IntelliTrack AOI - v1.2 (OPC)	Details | Edit | Delete
IntelliTrack OPC	CheckIn	IntelliTrack AOI - v1 (OPC)	Details | Edit | Delete
IntelliTrack OPC	CheckOut	IntelliTrack AOI - v1 (OPC)	Details | Edit | Delete
IntelliTrack OPC	EventControl	Event control & monitoring (OPC)	Details | Edit | Delete
IntelliTrack OPC	ExpiryDate	IntelliTrack AOI - v1.0 (OPC)	Details | Edit | Delete
IntelliTrack OPC	Genealogy	IntelliTrack Genealogy AOI for Adoption/Consume/Split/Merge - v2.0 (OPC)	Details | Edit | Delete

≡	IntelliTrack OPC	GetAlarmList	IntelliTrack AOI - v2.0 (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	GetQueue	IntelliTrack AOI - v1.0 (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	GetWorkOrder	IntelliTrack AOI - v1.0 (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	Heartbeat	Rockwell Logix IntelliTrack AOI - v2.1 (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	PrintOnDemand	IntelliTrack AOI - v1.0 (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	Recipe_100	IntelliTrack AOI - v4.0, Capacity: 100 Attributes (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	Recipe_25	IntelliTrack AOI - v4.0, Capacity: 25 Attributes (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	Recipe_50	IntelliTrack AOI - v1, Capacity: 50 Attributes (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	RecipeMonitor_100	IntelliTrack AOI - v1.0, Capacity: 100 Attributes (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	RecipeMonitor_25	IntelliTrack AOI - v1.0, Capacity: 25 Attributes (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	RecipeMonitor_50	IntelliTrack AOI - v1.0, Capacity: 50 Attributes (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	udt_Alarm	Alarm, warning and information list entry (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	udt_Genealogy_v1	Consumed-item configuration for Genealogy transactions (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	udt_KeyValuePair	Key/Value parameter pair (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	udt_LogValue_v1	UDT for Logging Values Back to IntelliTrack at Check-In (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	udt_QueueItem_v1	Staged product queue entry (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	udt_ValueType_v1	Holds Attribute Value for Recipes and Data Logging (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	udt_WorkOrder_v1	Work Order configuration detail (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	UserDefinedProcedure	IntelliTrack AOI - v3.1, Generic Implementation - SQL Modifications not Required (OPC)	Details | Edit | Delete
≡	IntelliTrack OPC	wc5_CarrierDb	Carrier stack database read/write AOI - v1.0 (OPC)	Details | Edit | Delete

Back to Home

Copyright © 2026 Factory Data Systems, LLC

Creating a Collection

Configuration → Collections → Create

Edit Collection

EventControl

This object is in use and changing it may affect the following objects:

(Tag) Test::Program:MainProgram.AddProductAOI[0].eventControl
(Tag) Test::Program:MainProgram.AddProductAOI[1].eventControl
(Tag) Test::Program:MainProgram.AddProductAOI[2].eventControl
(Tag) Test::Program:MainProgram.AddProductAOI[3].eventControl
(Tag) Test::Program:MainProgram.AddProductAOI[4].eventControl
(Tag) Test::Program:MainProgram.AddProductAOI[5].eventControl
(Tag) Test::Program:MainProgram.AddProductAOI[6].eventControl
(Tag) Test::Program:MainProgram.AddProductAOI[7].eventControl
(Tag) Test::Program:MainProgram.AddProductAOI[8].eventControl
(Tag) Test::Program:MainProgram.AddProductAOI[9].eventControl
(Tag) Test::Program:MainProgram.CheckInAOI[0].eventControl
(Tag) Test::Program:MainProgram.CheckInAOI[1].eventControl
(Tag) Test::Program:MainProgram.CheckInAOI[2].eventControl
(Tag) Test::Program:MainProgram.CheckInAOI[3].eventControl
(Tag) Test::Program:MainProgram.CheckInAOI[4].eventControl
(Tag) Test::Program:MainProgram.CheckOutAOI[0].eventControl
(Tag) Test::Program:MainProgram.CheckOutAOI[1].eventControl
(Tag) Test::Program:MainProgram.CheckOutAOI[2].eventControl
(Tag) Test::Program:MainProgram.CheckOutAOI[3].eventControl
(Tag) Test::Program:MainProgram.CheckOutAOI[4].eventControl
(Tag) Test::Program:MainProgram.CheckOutAOI[5].eventControl
(Tag) Test::Program:MainProgram.CheckOutAOI[6].eventControl
(Tag) Test::Program:MainProgram.CheckOutAOI[7].eventControl
(Tag) Test::Program:MainProgram.CheckOutAOI[8].eventControl
(Tag) Test::Program:MainProgram.CheckOutAOI[9].eventControl
(Tag) Test::Program:MainProgram.GenealogyAOI[0].eventControl
(Tag) Test::Program:MainProgram.GenealogyAOI[1].eventControl
(Tag) Test::Program:MainProgram.GenealogyAOI[2].eventControl
(Tag) Test::Program:MainProgram.GenealogyAOI[3].eventControl
(Tag) Test::Program:MainProgram.GenealogyAOI[4].eventControl
(Tag) Test::Program:MainProgram.HeartbeatAOI.eventControl
(Tag) Test::RecipeAOI[0].eventControl
(Tag) Test::RecipeAOI[1].eventControl
(Tag) Test::RecipeAOI[2].eventControl
(Tag) Test::RecipeAOI[3].eventControl
(Tag) Test::RecipeAOI[4].eventControl
(Tag) Test::RecipeAOI[5].eventControl
(Tag) Test::RecipeAOI[6].eventControl
(Tag) Test::RecipeAOI[7].eventControl
(Tag) Test::RecipeAOI[8].eventControl
(Tag) Test::RecipeAOI[9].eventControl
(Tag) Test::RecipeMonitorAOI[0].eventControl
(Tag) Test::RecipeMonitorAOI[1].eventControl
(Tag) Test::RecipeMonitorAOI[2].eventControl
(Tag) Test::RecipeMonitorAOI[3].eventControl
(Tag) Test::RecipeMonitorAOI[4].eventControl
(Tag) Test::RecipeMonitorAOI[5].eventControl
(Tag) Test::RecipeMonitorAOI[6].eventControl
(Tag) Test::RecipeMonitorAOI[7].eventControl
(Tag) Test::RecipeMonitorAOI[8].eventControl
(Tag) Test::RecipeMonitorAOI[9].eventControl

Details

Namespace
FDS

Name
EventControl

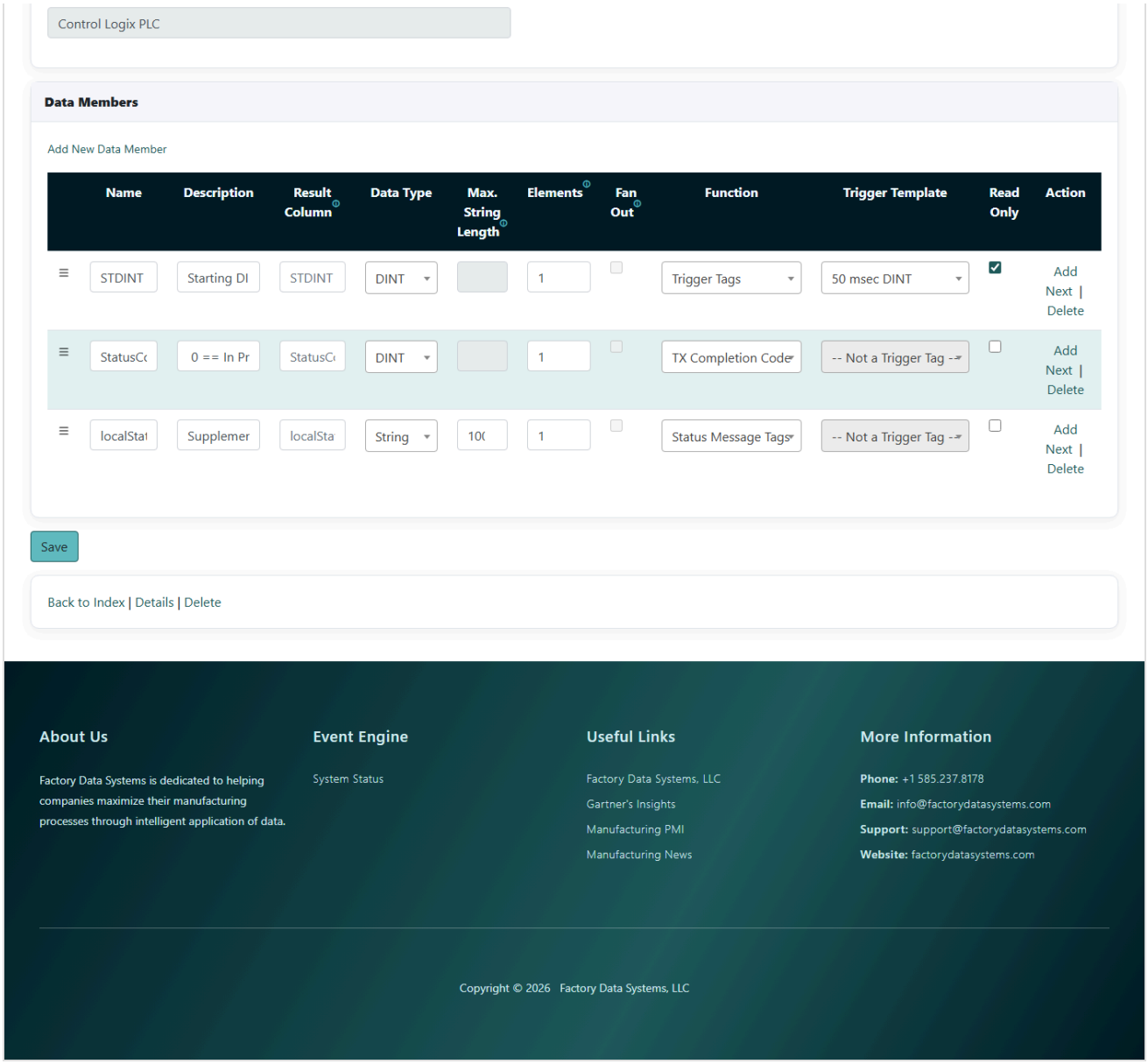
Description
Event Control & Monitoring AOI - v1

Communications Protocol

Date Created
6/8/2026 5:34:32 PM

Last Edited
8/21/2026 3:14:53 PM

Last Edited By
HQ\jsmith



> The figure shows an **existing** collection opened for editing, because a blank Create page has no data members to point at. The two pages carry the same cards in the same order; editing one that is already in use adds the amber banner listing what a change would affect.

1. Enter a **Name** and **Description**.
2. Choose the **Communications Protocol**. A collection is protocol-specific because each protocol supports a different set of data types; once the protocol is chosen, the **Add New Data Member** link appears.
3. Add **Data Members** — one row per member of the structure:

The columns, in the order the grid shows them:

Member column	What it does
Name	The member name. When a tag is created from this collection, member tags are created beneath it using these names.
Description	Free text.
Data Type	One of the protocol's supported data types — or another structure, allowing **nested collections**.
Max. String Length	For string members: the maximum string length allowed by the controller for this member. Used to size the underlying controller string when the collection is instantiated.
Elements	Number of elements if the member is an array (1 for a scalar).
Fan Out	For array members: create one child tag per element (`Member[index]`) whenever a tag is built from this collection, so events can bind an individual element rather than the whole array. Leave it off when the array is only ever read or written as a unit — the child tags are configuration you would otherwise have to maintain. Subject to the same protocol limits as *Create Children on Save* on a tag (Chapter 9.2).
Function	The member's role in events: **Payload (default)** — carries data only; **Trigger** — this member initiates events (pair it with a trigger template); **Completion Code** — receives the event completion code; **Message Tag** — receives status/error text. Pre-assigning functions here means every tag created from the collection comes out event-ready.
Trigger Template	For Trigger members, which trigger template (Chapter 7) governs them.
Read Only	Marks the member as not writable by Event Engine.

4. Drag rows to reorder, then **Save**.

Why pre-assign member functions?

A common pattern is one collection per transaction type, e.g.:

```
PartComplete (Collection, Logix)
├─ Trigger        BOOL     Function: Trigger     (template: RisingEdge-100ms)
├─ SerialNumber   STRING   Function: Payload
├─ Result         DINT[10] Function: Payload
├─ CompletionCode DINT     Function: Completion Code
└─ StatusMessage  STRING   Function: Message Tag
```

Create one tag of type `PartComplete` per station, select it as the event's **Master Tag**, and the event editor automatically finds the trigger, completion, and message members — no per-tag wiring.

8.3 Choosing Between a UDT and a Collection

- **The structure already exists in the PLC program?** Import it as a **UDT** — it stays faithful to the controller definition and updates by re-import.
- **You need a structure the PLC doesn't define** (e.g., a logical grouping spanning addresses on a Modbus device, or a database-shaped record)? Build a **Collection**.
- **You need member functions (trigger/completion/message) baked in?** Collections let you assign these per member at the template level.
- Both can be used as the data type of new tags, and both can be nested inside other collections (structures within structures), to a maximum nesting depth of 10 levels.

Next: Chapter 9 — Tags · Guide home

Chapter 9 — Tags

Configuration → Tags

> **Where this fits: Step 6** of the setup roadmap. A tag needs its processor (Ch. 6); a *structure* tag needs its UDT/Collection (Ch. 8); a *trigger* tag needs its template (Ch. 7). Tags are then bound by the events you build next (Ch. 10–13).

A **tag** is a named piece of data in a processor — an atomic value, an array, or an entire structure. Tags are what events read from and write to, what triggers watch, and what database parameters and label fields bind to.

9.1 The Tag Index

The index lists all tags with searching and paging. Structure tags can be **drilled down** — click a structure to see its child-tag detail list (*Tag Structure Detail List — All Child Tags of X*), and keep drilling through nested levels. Use **Up One Level** / **Back to Parent** to navigate back.

EVENT ENGINE™ Home System Configuration Events Current User: HQ\jsmith

Tag Index

A listing of all the tags in the system.

Tag Index

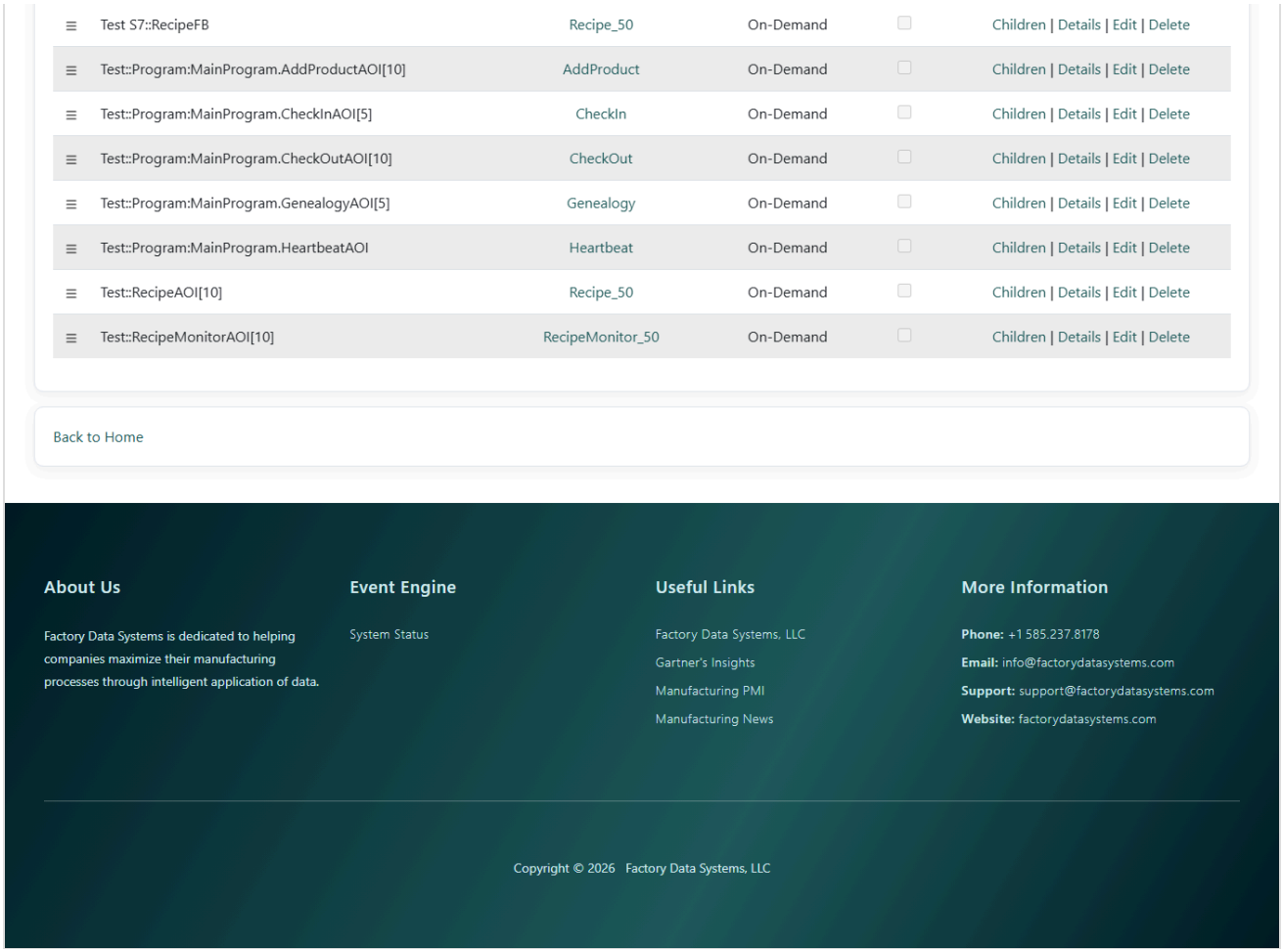
Create | Delete Unused Tags | Tags With Missing Structure Type

Enter Search Phrase...

Tag Name	Data Type	Scan Interval (msec.)	Read Only	Action
Kep::ns=2;s=OP200.CheckInAOI[2] OP400 Check-In (IntelliTrack OPC / CheckIn)	CheckIn	On-Demand	☐	Children | Details | Edit | Delete
Kep::ns=2;s=OP200.CheckOutAOI[2] OP400 CheckOut (IntelliTrack OPC / CheckOut)	CheckOut	On-Demand	☐	Children | Details | Edit | Delete
Kep::ns=2;s=OP200.RecipeAOI[0]	Recipe_50	On-Demand	☐	Children | Details | Edit | Delete
Kep::ns=2;s=OP200.RecipeAOI[2] OP400 Recipe (IntelliTrack OPC / Recipe_50)	Recipe_50	On-Demand	☐	Children | Details | Edit | Delete
Kep::ns=2;s=OP200.RecipeMonitorAOI[2] OP400 Recipe Monitor (IntelliTrack OPC / RecipeMonitor_50)	RecipeMonitor_50	On-Demand	☐	Children | Details | Edit | Delete
S7OPC::ns=3;s=DeviceManual	String(82)	On-Demand	☐	Details | Edit | Delete
S7OPC::ns=3;s=OperatingMode	Int32	On-Demand	☐	Details | Edit | Delete
Test S7::DB300.DBB110 (localEquipmentName) CheckIn OP300 - localEquipmentName [@equipmentName]	String(82)	On-Demand	☐	Details | Edit | Delete
Test S7::DB300.DBB194 (localSerialNumber) CheckIn OP300 - localSerialNumber [@serialNo]	String(82)	On-Demand	☐	Details | Edit | Delete
Test S7::DB300.DBB278 (localNewSerialNumber) CheckIn OP300 - localNewSerialNumber [@newSerialNo]	String(82)	On-Demand	☐	Details | Edit | Delete
Test S7::DB300.DBB362 (localAlias) CheckIn OP300 - localAlias [@alias]	String(82)	On-Demand	☐	Details | Edit | Delete
Test S7::DB300.DBB446 (localNewMaterial) CheckIn OP300 - localNewMaterial [@newMaterial]	String(82)	On-Demand	☐	Details | Edit | Delete
Test S7::DB300.DBB530 (localNewMaterialRevision) CheckIn OP300 - localNewMaterialRevision [@newMaterialRev]	String(20)	On-Demand	☐	Details | Edit | Delete
Test S7::DB300.DBB552 (localNewContainer) CheckIn OP300 - localNewContainer [@newContainer]	String(82)	On-Demand	☐	Details | Edit | Delete
Test S7::DB300.DBB636 (localNewLocation) CheckIn OP300 - localNewLocation [@newLocation]	String(82)	On-Demand	☐	Details | Edit | Delete
Test S7::DB300.DBB720 (localOperator) CheckIn OP300 - localOperator [@operator]	String(82)	On-Demand	☐	Details | Edit | Delete
Test S7::DB300.DBB8 (eventControl.localStatusMessage) CheckIn OP300 - eventControl.localStatusMessage [completion message written back by EE]	String(100)	On-Demand	☐	Details | Edit | Delete
Test S7::DB300.DBB804 (localWorkOrder) CheckIn OP300 - localWorkOrder [@workOrder]	String(82)	On-Demand	☐	Details | Edit | Delete
Test S7::DB300.DBB888 (localShippingCode) CheckIn OP300 - localShippingCode [@shippingCode]	String(82)	On-Demand	☐	Details | Edit | Delete
Test S7::DB300.DBB978[50] (localPPDItems) CheckIn OP300 - localPPDItems [@ppdItems]	IntelliTrack___Siemens.udt_LogValue_v1 (Imported Siemens type udt_LogValue_v1)	On-Demand	☐	Details | Edit | Delete
Test S7::DB300.DBD0 (eventControl.STDINT) CheckIn OP300 - eventControl.STDINT [trigger (50 msec DINT template)]	DINT	On-Demand	☐	Details | Edit | Delete
Test S7::DB300.DBD4 (eventControl.StatusCode)	DINT	On-Demand	☐	Details | Edit | Delete

	CheckIn OP300 - eventControl.StatusCode [completion code written back by EE]				
≡	Test S7::DB300.DBD972 (NewPosition) CheckIn OP300 - NewPosition [@newPosition]	DINT	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB300.DBX976.0 (UseNewSN) CheckIn OP300 - UseNewSN [@useNewSN]	Bool	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB300.DBX976.1 (IsAudit) CheckIn OP300 - IsAudit [@isAudit]	Bool	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB300.DBX976.2 (Passed) CheckIn OP300 - Passed [@pfResult]	Bool	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB300.DBX976.3 (OKToShip) CheckIn OP300 - OKToShip [@okToShip]	Bool	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB300.DBX976.4 (InvalidAttribute) CheckIn OP300 - InvalidAttribute [AOI output - not bound by EE]	Bool	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBB110 (localEquipmentName) CheckOut OP300 - localEquipmentName [@equipmentName]	String(82)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBB194 (localSerialNumber) CheckOut OP300 - localSerialNumber [@serialNumber]	String(82)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBB278 (localContainer) CheckOut OP300 - localContainer [@container]	String(82)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBB362 (localAlias) CheckOut OP300 - localAlias [@alias]	String(82)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBB446 (localContextSN) CheckOut OP300 - localContextSN [@contextSN]	String(82)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBB530 (localWorkOrder) CheckOut OP300 - localWorkOrder [@workOrder]	String(82)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBB614 (localMaterialName) CheckOut OP300 - localMaterialName [@materialName]	String(82)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBB698 (localMaterialRevision) CheckOut OP300 - localMaterialRevision [@materialRev]	String(20)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBB720 (localOperator) CheckOut OP300 - localOperator [@operator]	String(82)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBB8 (eventControl.localStatusMessage) CheckOut OP300 - eventControl.localStatusMessage [completion message written back by EE]	String(100)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBB804 (localLastProcess) CheckOut OP300 - localLastProcess [@lastProcess]	String(82)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBB888 (localResultMessage) CheckOut OP300 - localResultMessage [@resultMessage]	String(100)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBD0 (eventControl.STDINT) CheckOut OP300 - eventControl.STDINT [trigger (50 msec DINT template)]	DINT	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBD1002 (Result) CheckOut OP300 - Result [@result]	DINT	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBD1006 (SkipToken) CheckOut OP300 - SkipToken [@skipToken]	DINT	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBD4 (eventControl.StatusCode) CheckOut OP300 - eventControl.StatusCode [completion code written back by EE]	DINT	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBD990 (Position) CheckOut OP300 - Position [@position]	DINT	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBD994 (InventoryStatus) CheckOut OP300 - InventoryStatus [@inventoryStatus]	DINT	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBD998 (Attempts) CheckOut OP300 - Attempts [@attempts]	DINT	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB301.DBX1010.0 (isAudit) CheckOut OP300 - isAudit [@isAudit]	Bool	On-Demand	☐	Details | Edit | Delete

≡	Test S7::DB301.DBX1010.1 (OKToWork) CheckOut OP300 - OKToWork [@okToWork]	Bool	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB302.DBB110 (localEquipmentName) Recipe_50 OP300 - localEquipmentName [@equipmentName]	String(82)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB302.DBB194 (localRecipeName) Recipe_50 OP300 - localRecipeName [@masterRecipeName]	String(82)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB302.DBB278 (localRevision) Recipe_50 OP300 - localRevision [@masterRecipeRevision]	String(20)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB302.DBB300 (localContextSN) Recipe_50 OP300 - localContextSN [@contextSN]	String(82)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB302.DBB384 (localOperator) Recipe_50 OP300 - localOperator [@operator]	String(82)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB302.DBB478 (localRecipe.material) Recipe_50 OP300 - localRecipe.material [@material]	String(82)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB302.DBB562 (localRecipe.materialRevision) Recipe_50 OP300 - localRecipe.materialRevision [@materialRev]	String(20)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB302.DBB584 (localRecipe.description) Recipe_50 OP300 - localRecipe.description [@description]	String(82)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB302.DBB668 (localRecipe.lastUpdated) Recipe_50 OP300 - localRecipe.lastUpdated [@lastUpdated]	String(35)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB302.DBB706[50] (localRecipe.values) Recipe_50 OP300 - localRecipe.values [result-set write-back target]	IntelliTrack___Siemens.udt_ValueType_v1 (Imported Siemens type udt_ValueType_v1)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB302.DBB8 (eventControl.localStatusMessage) Recipe_50 OP300 - eventControl.localStatusMessage [completion message written back by EE]	String(100)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB302.DBD0 (eventControl.STDINT) Recipe_50 OP300 - eventControl.STDINT [trigger (50 msec DINT template)]	DINT	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB302.DBD4 (eventControl.StatusCode) Recipe_50 OP300 - eventControl.StatusCode [completion code written back by EE]	DINT	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB302.DBD468 (qtyAttributes) Recipe_50 OP300 - qtyAttributes [@qtyAttributes]	DINT	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB302.DBD472 (localControlInstanceId) Recipe_50 OP300 - localControlInstanceId [@controlInstanceNumber]	DINT	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB302.DBX476.0 (OVR) Recipe_50 OP300 - OVR [AOI output - not bound by EE]	Bool	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB303.DBB110 (localEquipmentName) RecipeMonitor_50 OP300 - localEquipmentName [@equipmentName]	String(82)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB303.DBB196[50] (localValues) RecipeMonitor_50 OP300 - localValues [@values]	IntelliTrack___Siemens.udt_ValueType_v1 (Imported Siemens type udt_ValueType_v1)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB303.DBB8 (eventControl.localStatusMessage) RecipeMonitor_50 OP300 - eventControl.localStatusMessage [completion message written back by EE]	String(100)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB303.DBD0 (eventControl.STDINT) RecipeMonitor_50 OP300 - eventControl.STDINT [trigger (50 msec DINT template)]	DINT	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB303.DBD4 (eventControl.StatusCode) RecipeMonitor_50 OP300 - eventControl.StatusCode [completion code written back by EE]	DINT	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB303.DBX194.0 (IsRunning) RecipeMonitor_50 OP300 - IsRunning [@isRunning]	Bool	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB4.DBB0[50] recipeValues	IntelliTrack.Value_Type (Imported Siemens type Value_Type)	On-Demand	☐	Details | Edit | Delete
≡	Test S7::DB4.DBB14200	Error: Data Type Not Found	On-Demand	☐	Details | Edit | Delete
≡	Test S7::HeartbeatFB	Heartbeat	On-Demand	☐	Children | Details | Edit | Delete

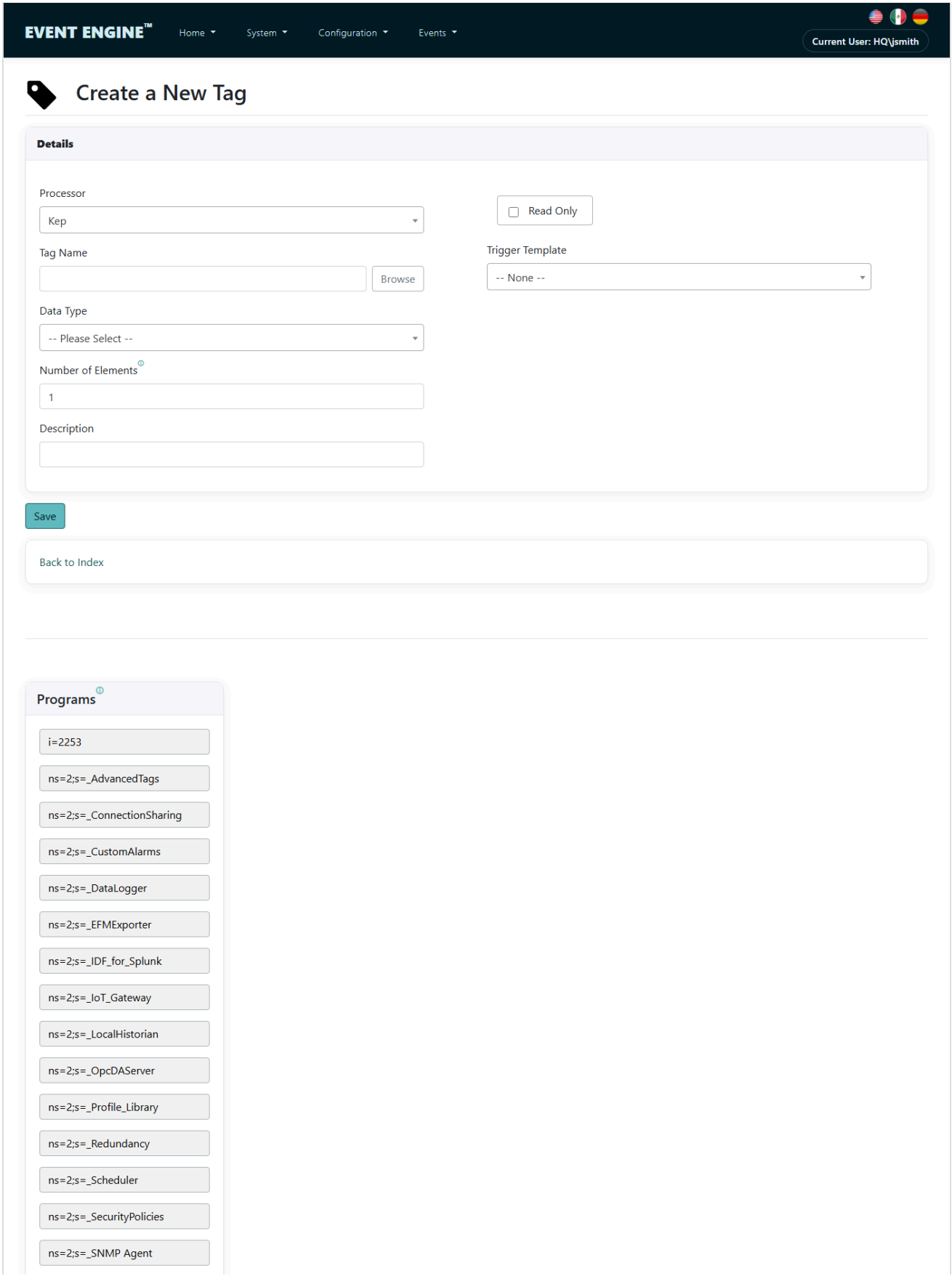


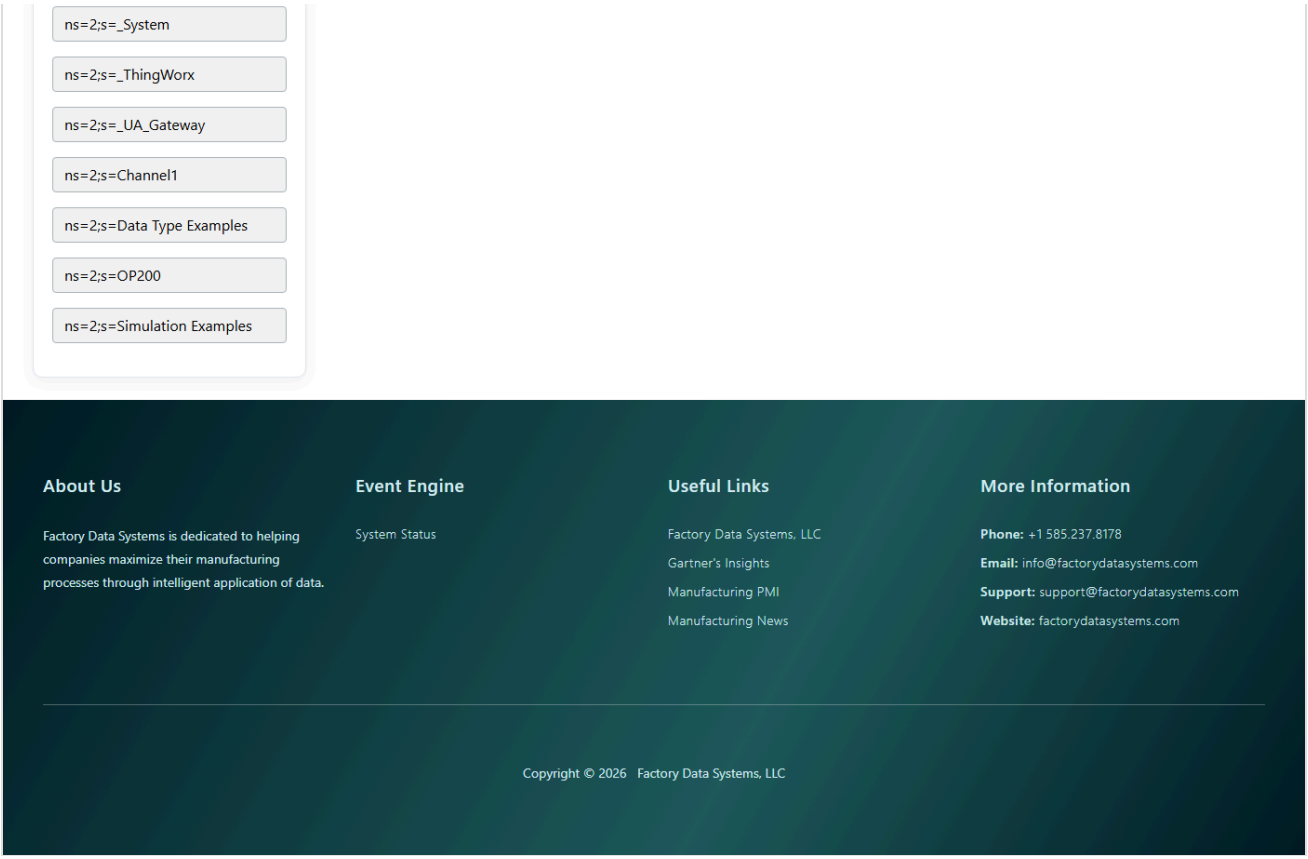
Tags display using the convention `Processor::TagName`. The processor prefix is for display only — the underlying tag name is what's addressed in the controller. Three things are appended when they apply, in this order:

Shown	When	Example
`[n]`	The tag is an array of *n* elements	`Line1CLX::Results[10]`
`(symbol)`	A **Symbol** is recorded on the tag (9.2)	`S7Cell::DB4.DBB0 (Recipe.Header)`
`(tag type)`	The processor is **Modbus** — the register class	`Pump::40001 (Holding (Output) Register (from 40001))`

9.2 Creating a Tag

Configuration → Tags → Create


EVENT ENGINE™
Home
System
Configuration
Events
Current User: HQ\jsmith
Create a New Tag
Details
Processor
Kep
Read Only
Tag Name
Browse
Trigger Template
-- None --
Data Type
-- Please Select --
Number of Elements
1
Description
Save
Back to Index
Programs
i=2253
ns=2;s=_AdvancedTags
ns=2;s=_ConnectionSharing
ns=2;s=_CustomAlarms
ns=2;s=_DataLogger
ns=2;s=_EFMExporter
ns=2;s=_IDF_for_Splunk
ns=2;s=_IoT_Gateway
ns=2;s=_LocalHistorian
ns=2;s=_OpcDAServer
ns=2;s=_Profile_Library
ns=2;s=_Redundancy
ns=2;s=_Scheduler
ns=2;s=_SecurityPolicies
ns=2;s=_SNMP Agent



Fields

Field	What it does and how it affects operation
Processor	Which controller the tag lives in. **Select this first** — it determines the available data types and whether browsing is offered. A processor is required.
Tag Name	The address/name of the tag in the controller. For Rockwell *program-scoped* tags use the form `Program:ProgramName.TagName`. For protocols without browsing, enter the native address (see the processor page for accepted formats — e.g., `N7:0` for AB Legacy, `DB10.DBD4` style addresses for Siemens, register references for Modbus, `%R100` style for GE).
Browse button	For PLC families that support it — **Rockwell Logix** and **OPC** — opens the live **Tag Browser** (9.3). For other families the hint reads: "*Tag browse is not available for this PLC family — enter the address manually.*"
Data Type	The tag's data type, drawn from the selected processor's protocol (9.4) — including imported UDTs and Collections, which appear as structure types. Choose the type that matches the PLC declaration exactly; mismatches cause read/write errors or corrupted values.
Max. String Length	Shown for string tags: the maximum string length allowed by the controller for the given tag (1 – 32767). Must match the controller's declared string size — Event Engine treats it as the true capacity and cuts longer values to fit it (9.6).

Field	What it does and how it affects operation
Number of Elements	The number of elements (size) of an array — use 1 for a scalar. Maximum **1000**. Arrays display as `Name[n]` on list pages.
Description	Free text.
Read Only	The tag's value can be read but never written by Event Engine. Use for inputs and any data the PLC owns.
Create Children on Save?	For array tags (Elements > 1): creates one child tag per element (`Name[index]`) so individual elements can be referenced by events. Offered on both Create and Edit — on Edit the box reflects whether children currently exist, so **unchecking it and saving removes them**. Not available for Collection types, for Modbus processors, or for Siemens tags using reference (absolute) addressing: none of those can express `Name[index]` as a device address.
Trigger Template	Assigning a template (Chapter 7) makes this a **trigger tag** — it will be scanned at the template's interval and can initiate events. Tags without a template are *on-demand* (payload) tags, read only when an event needs them. (Assigned on the Edit page after creation, or pre-assigned via a Collection member.)
Tag Type	Shown for Modbus processors: the register class the address refers to — Discrete Output/Coil (from 00001), Discrete Input (from 10001), Holding (Output) Register (from 40001), Input Register (read-only, from 30001), and other Modbus classes.
Use symbolic (optimized) addressing	Shown for Siemens S7 processors. **Unavailable in this release:** the box is shown unchecked and disabled, so the Tag Name is always an absolute address (`DB1.DBW2`). Symbolic addressing needs a separate Siemens driver license this release does not include — see Chapter 6.
Array Lower Bound	Shown for a Siemens symbolic array tag (Elements > 1): the array's first index as declared in the controller (e.g. `1` for `Array[1..10]` ; most arrays start at 0). Generated element child tags are named from this index.
Symbol	*(Optional)* The controller symbol that goes with an address — for example the TIA name of the data-block member at `DB4.DBB0` , or what Modbus register `40001` actually holds. It **supplements** the address and never replaces it: nothing is addressed by it. Its value is that an absolute address names a location rather than a thing, so `DB4.DBB0` alone tells an operator picking a tag nothing; with a symbol recorded, every picker, breadcrumb and Error List entry shows `DB4.DBB0 (Recipe.Header)` . Offered on the protocols whose tag names *are* addresses — Siemens S7 in reference (absolute) mode, Modbus, GE SRTP and AB Legacy. Symbolic (optimized) Siemens tags don't need it, because there the Tag Name already *is* the controller symbol. Leave blank for none.

Saving a **structure tag** creates the entire member tree from the UDT/Collection definition — for large structures the page shows "*Creating tag structure. Please wait…*". Structures may nest up to **10 levels** deep; a deeper (or circular) structure is rejected with a recursion error.

> **Performance note — trigger tags vs. payload tags:** every trigger tag adds continuous scan load at its template's scan interval. Payload (on-demand) tags cost nothing until an event runs, when they are read as part of the event's *tag collection* phase. Configure only true triggers as trigger tags and leave bulk data as payload for the fastest system.

9.3 Browsing Tags (Rockwell Logix and OPC)

For Logix and OPC processors the **Browse** button opens a live browser against the running controller (or, for OPC, against the server's address space):

1. **Programs** — choose *Controller Tags* (controller scope) or any program to see the tags within that scope. Click a program name to see the tags within it.
2. **Tags** — click a tag to select it; structure tags can be **drilled into** (click the structure name) to select an individual member, including array elements such as `Stations[0].Status`.
3. Selecting a tag fills the tag name and data type on the create form automatically, including the correct structure type when the browsed tag is a UDT instance.

Browsing reads the controller's own tag database, so names, types, and array sizes are guaranteed to match the PLC program — use it whenever it's available.

> **OPC browsing** walks the server's address space instead of a controller tag database, and the steps above map onto it directly: folders take the place of programs, and a node is selected the same way. The browser fetches a node's **complete** child list even when the server returns it in several responses, so a large folder is never silently truncated.

The remaining protocols — Siemens S7, Modbus/TCP, GE SRTP and AB Legacy — have no browse capability; their tags are created by typing the address (see the processor page for the accepted formats, and consider recording a **Symbol** so the address is recognizable later).

9.4 Data Types by Protocol

The data type list offered for a tag comes from the processor's protocol. Native types per protocol:

Protocol	Supported data types
Rockwell Logix	BOOL, SINT/Int8, INT/Int16, DINT/Int32, LINT/Int64, REAL/Float32, Float64, String, Structure
AB Legacy	BOOL, SINT/Int8, INT/Int16, DINT/Int32, LINT/Int64, REAL/Float32, Float64, String, Structure

Protocol	Supported data types
Siemens S7	BOOL (bit-address tags, e.g. `M100.7` / `DB1.DBX0.0`), Int8/UInt8, Int16/UInt16, DINT/Int32/UInt32, SINT, REAL/Float32, BCD8, BCD16, BCD32, String, Structure. 64-bit types (LINT, LREAL, …) are not addressable as standalone tags — carry them inside a UDT/Structure tag.
Modbus/TCP	Int16/UInt16, Int32/UInt32, REAL, BCD16, BCD32, String
GE SRTP	BOOL, SINT/Int8, INT/Int16, DINT/Int32, LINT/Int64, REAL/Float32, Float64, String, Structure
OPC	BOOL, SINT/Int8, INT/Int16, UInt16, DINT/Int32, UInt32, LINT/Int64, REAL/Float32, Float64, String, Structure

In addition, every protocol that supports *Structure* also offers your imported **UDTs** and **Collections** as selectable types (shown as custom/structure types in the list).

> The list shows PLC-style names (DINT, REAL, …). Internally each maps to the corresponding .NET type; the *Data Type* shown on UDT member pages is the .NET type, not the PLC name.

9.5 Arrays

- Set **Number of Elements** to the array length (maximum 1000). List pages display arrays as `Name[n]`.
- **Create Children on Save** generates a child tag per element (`Name[index]`) so events can bind individual elements. Not offered for Collection types, for Modbus processors, or for Siemens reference (absolute) tags. On a Siemens *symbolic* array, the child names start from **Array Lower Bound** rather than 0.
- On an **OPC** processor, each child is read, written and monitored as that element of the array tag's node. Some servers, KEPServerEX among them, refuse a write to a single array element; to write elements there, add the server's own per-element tags as tags of their own instead of creating children.
- Arrays *of structures* are supported via the structure type's element count — each element becomes a full structure instance in the tag tree.

9.6 String Tags: Capacity and Truncation

A controller's string tag holds a fixed number of characters — 82 for a standard Rockwell `STRING`, whatever the declaration says for a Siemens `STRING[n]` or a user-defined string type. **Max. String Length** is where you record that number, and Event Engine treats it as the truth: it will not write a value longer than the capacity the tag declares.

What happens to a value that does not fit

The value is **cut to exactly the capacity** and written. It is never rejected, and the event does not fail — a status message a few characters over its tag is normal traffic, not a fault.

On every protocol except Siemens, the end of the value is replaced with a **truncation marker** so that anyone reading the value back can see that it was cut:

Tag capacity	Value written	Marker
5 or more	`Line 3 stalled at st...`	`...` — the last three characters
4 or fewer	`Lin~`	`~` — a single character, because `...` would leave nothing of the value

The marker is part of the value, not extra: a 20-character tag receives exactly 20 characters. This matters because the alternative — writing a value that *looks* complete — is how a truncated status message gets read as the whole story.

> **Siemens is deliberately different.** Siemens string values are cut to capacity with **no marker**. They land in PLC memory that HMIs and ladder comparisons read as data, where three dots would be corruption rather than a signal. If you compare a Siemens string in logic, compare against a length the tag can actually hold.

Where truncation shows up

Anything Event Engine writes into a string tag is subject to it — a result-set column mapped to a string tag, a substituted value, a UDT string member. The one you will meet most often is the **event status message** written to an event's *Message Tag*: it carries the event id, the event name, a timestamp and a duration (or the error text when the event failed), which routinely runs past 82 characters. A long event name is the usual reason an operator sees the tail of a status message replaced by `...`.

Reading it in the log

Every truncation on a device write is recorded as a **warning** naming the tag, the loss, and the exact text that reached the controller:

```
String truncated to fit tag "Program:Main.StatusMsg": 118 characters exceeded the tag's maximum
string length of 82, so 36 were dropped and "70: OP400 - Recipe: Successful at 2026-08-25 14:02:11
```

```
in 412ms." was written (the trailing marker shows the value was cut).
Source: event "OP400 - Recipe" (70).
```

Source names the event behind the write, so a truncation can be traced back to the event that produced it rather than only to the tag that received it.

The status-message truncation above is the exception: because it is expected on every completed event, it is recorded at **Debug** rather than Warning — raise the log level (Chapter 3) if you need to see those. Without that exception a busy site produces six-figure warning counts a day, which buries the truncations that do deserve attention.

What to do about it

- **If the value should fit:** raise **Max. String Length** to the capacity the controller actually declares. A tag left at the 82-character default while the controller holds a longer string is the most common cause of unexpected truncation.
- **If the tag really is that small:** shorten what is written to it. For status messages, a shorter **event name** buys the most room, since it is repeated in the message.
- **A tag whose Max. String Length is 0** cannot be written at all: Event Engine refuses the write and fails the event rather than guessing a capacity, and the Error List names the tag. Set the field to the controller's declared size.

> Browsing does **not** discover string capacity — neither Rockwell Logix browse nor OPC reports it, and OPC UA has no per-node string length at all. **Max. String Length** is always yours to set.

9.7 Editing and Deleting Tags

- **Edit** lets you change the data type, elements, description, read-only flag, trigger template, symbol, and tag type. The processor cannot be changed after creation — recreate the tag instead.
- Tags that are **in use cannot be deleted**: the delete page lists every event, parameter, or structure that references the tag so you can detach it first.
- The details page records **Date Created**, **Last Edited**, and **Last Edited By** for change tracking.

> **After editing tags used by running events:** restart scanning (Stop/Start on the System Status page) so tag groups are rebuilt with the new definitions; the header warning icon indicates when a refresh or restart is pending.

9.8 Two Maintenance Reports

Large configurations accumulate tags that nothing uses, and — after a UDT is deleted or re-imported under a new identity — tags whose structure type no longer exists. Two links at the bottom of the **Tags** index find each case. Both require the Admins role.

Delete Unused Tags

Configuration → Tags → Delete Unused Tags

Finds tag structures that **no event references**, and offers to delete them in one sweep. A structure is judged as a whole: if *any* member of it is used by any event, the entire structure is kept. The page is a preview first — it lists every root tag that would go, with its processor and data type, and deletes nothing until you confirm.

> ⚠️ **This is permanent.** The confirmation warns "*N unused tag structure(s) will be permanently deleted, including all of their child tags. This action cannot be undone.*" Take a configuration backup (Chapter 3.5) first.

The deletable set is recomputed on the server when you confirm, not taken from the page you were looking at. If an event started using one of the candidates in between, nothing is removed and the index reports it — so the sweep cannot delete a tag that became live while you were reading.

Tags With Missing Structure Type

Configuration → Tags → Tags With Missing Structure Type

Lists every structure tag whose type no longer matches any compiled UDT — the usual cause is a UDT deleted, or re-imported in a way that gave it a new identity. Each row shows the tag, the missing type, and **which events reference it**.

This report is **read-only by design, and deliberately not part of the sweep above**. A tag can be both actively used by events *and* pointing at a type that no longer exists — that combination is precisely how the problem goes unnoticed — so it never appears in the unused-tag sweep, and bulk-deleting it would destroy working configuration. Fix each one individually: either re-import the missing UDT (Chapter 8.1), or edit the tag to point at a valid type.

> **Why it matters:** events that read such a tag fail at tag collection, and the tag's structure can no longer be rebuilt. Opening one in the tag editor is safe — the Data Type list shows a disabled **Missing UDT** entry carrying the orphaned reference, so the broken value stays visible and the browser cannot quietly select a different type in its place — but the tag stays broken until you re-point it or restore the type. Check this report after any UDT deletion or re-import, and after restoring a configuration backup onto a machine whose UDTs were built separately.

Next: Chapter 10 — Events: Stored Procedures · Guide home

Chapter 10 — Events: Stored Procedures

Events → Stored Procedure Events

> **Where this fits: Step 7** of the setup roadmap — where everything comes together. Before building this event, make sure its database connection (Ch. 5), its tags (Ch. 9), and its triggers (Ch. 7) already exist. When it's saved, start and verify it (Ch. 14.2).

10.1 What a Stored Procedure Event Does

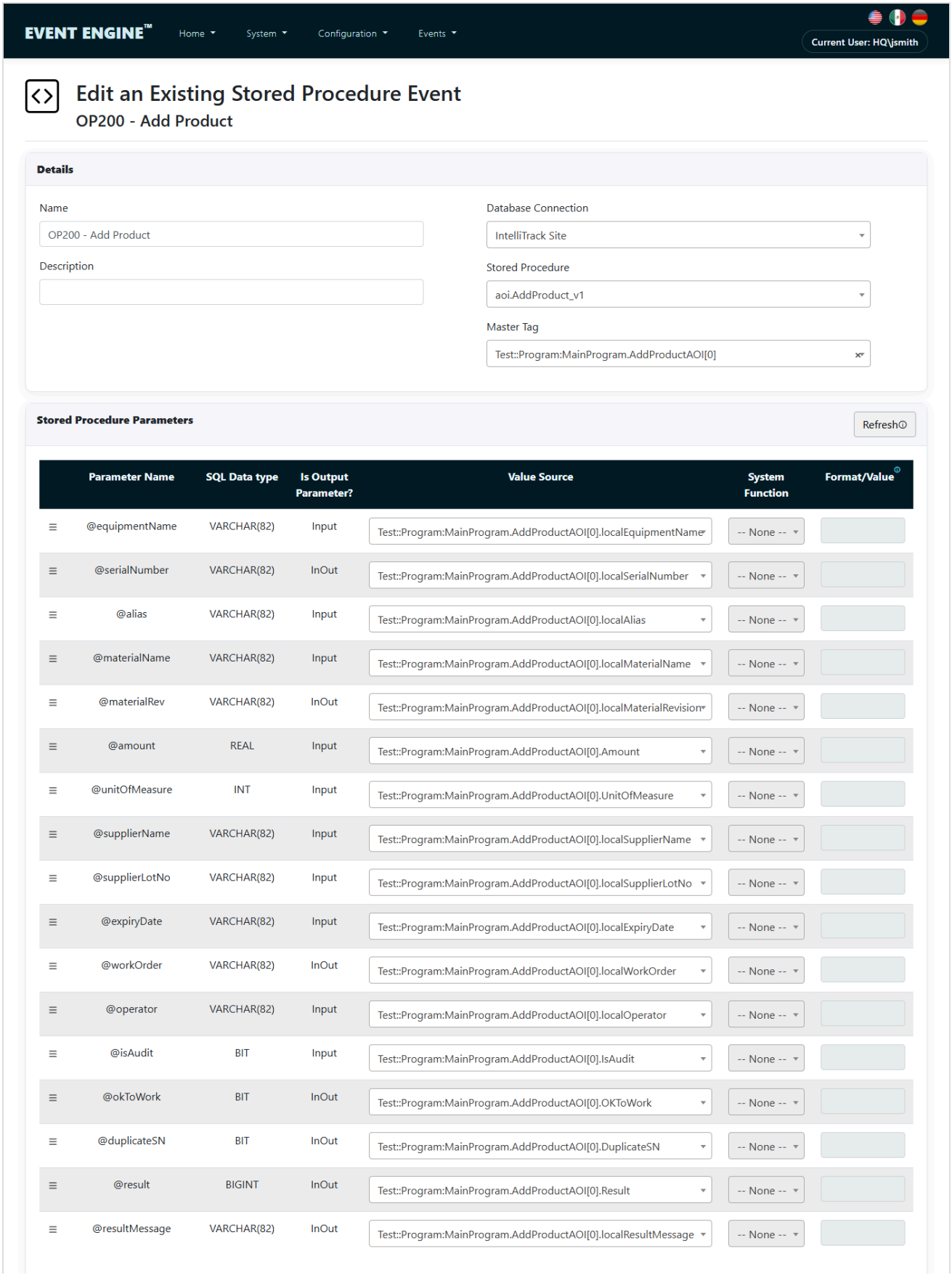
A Stored Procedure event executes a stored procedure in one of your databases when triggered:

1. **Tag collection** — Event Engine reads every PLC tag bound to the procedure's parameters (optimized into as few reads as possible — this is the *Tag Collection Time* in telemetry).
2. **Work** — it calls the stored procedure on the selected database connection, passing the tag values as parameters (*Work Time*).
3. **Write back** — output parameters and result sets are written back to the controller, followed by the completion code and any status message (*Write Back Time*).

This is the workhorse event type for MES transactions: part tracking, genealogy, recipe download, production counts, and so on.

10.2 Creating a Stored Procedure Event

Events → Stored Procedure Events → Create (or **Events → All Events → Create → Stored Procedure**)


EVENT ENGINE™
Home
System
Configuration
Events
Current User: HQ\jsmith
Edit an Existing Stored Procedure Event
OP200 - Add Product
Details
Name
OP200 - Add Product
Description
Database Connection
IntelliTrack Site
Stored Procedure
aoi.AddProduct_v1
Master Tag
Test::Program:MainProgram.AddProductAOI[0]
Stored Procedure Parameters
Refresh
Parameter Name
SQL Data type
Is Output Parameter?
Value Source
System Function
Format/Value
@equipmentName VARCHAR(82) Input Test::Program:MainProgram.AddProductAOI[0].localEquipmentName -- None --
@serialNumber VARCHAR(82) InOut Test::Program:MainProgram.AddProductAOI[0].localSerialNumber -- None --
@alias VARCHAR(82) Input Test::Program:MainProgram.AddProductAOI[0].localAlias -- None --
@materialName VARCHAR(82) Input Test::Program:MainProgram.AddProductAOI[0].localMaterialName -- None --
@materialRev VARCHAR(82) InOut Test::Program:MainProgram.AddProductAOI[0].localMaterialRevision -- None --
@amount REAL Input Test::Program:MainProgram.AddProductAOI[0].Amount -- None --
@unitOfMeasure INT Input Test::Program:MainProgram.AddProductAOI[0].UnitOfMeasure -- None --
@supplierName VARCHAR(82) Input Test::Program:MainProgram.AddProductAOI[0].localSupplierName -- None --
@supplierLotNo VARCHAR(82) Input Test::Program:MainProgram.AddProductAOI[0].localSupplierLotNo -- None --
@expiryDate VARCHAR(82) Input Test::Program:MainProgram.AddProductAOI[0].localExpiryDate -- None --
@workOrder VARCHAR(82) InOut Test::Program:MainProgram.AddProductAOI[0].localWorkOrder -- None --
@operator VARCHAR(82) Input Test::Program:MainProgram.AddProductAOI[0].localOperator -- None --
@isAudit BIT Input Test::Program:MainProgram.AddProductAOI[0].IsAudit -- None --
@okToWork BIT InOut Test::Program:MainProgram.AddProductAOI[0].OKToWork -- None --
@duplicateSN BIT InOut Test::Program:MainProgram.AddProductAOI[0].DuplicateSN -- None --
@result BIGINT InOut Test::Program:MainProgram.AddProductAOI[0].Result -- None --
@resultMessage VARCHAR(82) InOut Test::Program:MainProgram.AddProductAOI[0].localResultMessage -- None --

Result Sets & Cursors

Add New Result Set

Name	Description	Result Set Target Tag	Action

Event Control

Trigger Configuration

Trigger Tag(s)

× Test::Program:MainProgram.AddProductAOI[0].eventControl.STDINT ×

Time Based Triggers

× -- None --

Completion Signal Configuration

Completion Tag(s)

× Test::Program:MainProgram.AddProductAOI[0].eventControl.StatusCode ×

Status Message Tag(s)

Search for a tag...

Event Options

Timeout (seconds)

2

Cascaded Events

× -- None --

- [x] Enabled
- [ ] Allow Overlapping Events
- [x] Use Single Write

Notification Configuration

- [ ] Notify on Failure

Notification Addresses

Tracking Information

Date Created

6/25/2026 9:09:38 AM

Last Edited

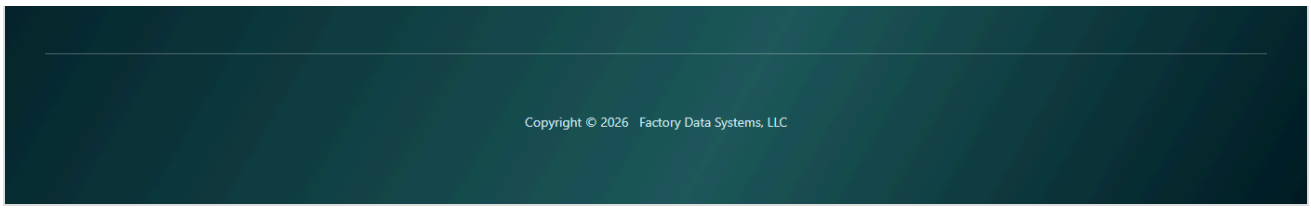
9/2/2026 9:34:05 AM

Last Edited By

HQ\jsmith

Save

Back to Index | Telemetry | Details | Delete



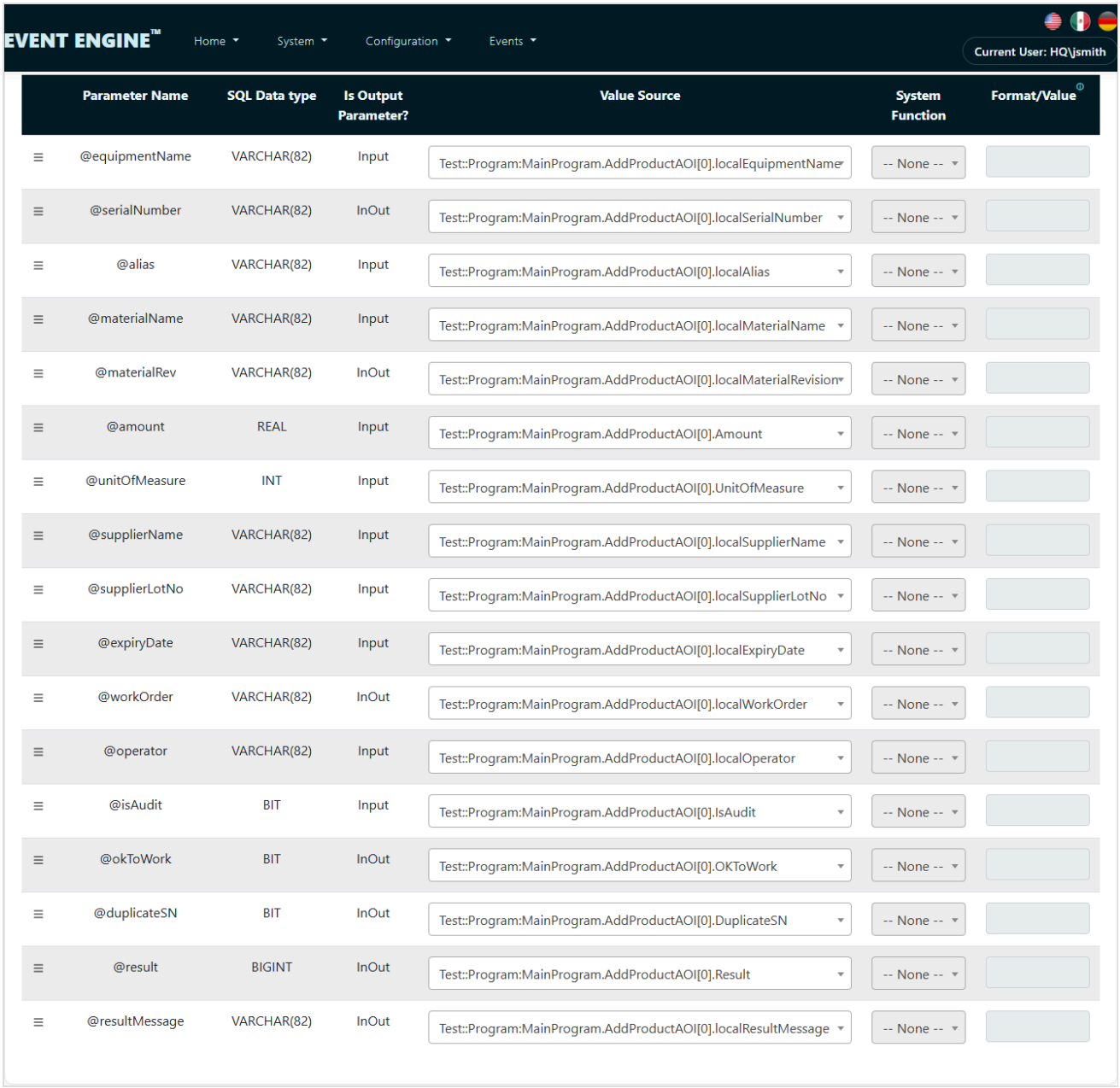
> The figure shows an **existing** event opened for editing. The create and edit pages are the same five cards in the same order, but the parameter table below only fills in once a procedure has been chosen, so a configured event is what the steps that follow can actually be read against.

Step 1 — Details

Field	What it does
Name	Unique event name (shown on the event list and telemetry).
Description	Free text.
Database Connection	Which database to call (Chapter 5). Selecting a connection loads the list of available stored procedures from the database. A connection-status banner shows whether the database is reachable; if it shows "*Database Connection is Unavailable*", fix connectivity before continuing.
Stored Procedure	The procedure to execute. The dropdown shows the currently-saved procedure; press **Refresh** (or change the **Db Connection**) to read the live list from the database and pick a different one.
Master Tag	*(Optional but powerful)* A tag **structure** used as the foundation for the event's tags. When selected, the structure's members are *married up* to the procedure's parameters by name, and the Trigger / Completion / Message dropdowns are filtered to the structure's members with those functions.

Step 2 — Stored Procedure Parameters

When a procedure is selected, Event Engine reads its **parameter definitions from the database** and builds the parameter table: parameter name, SQL data type, direction, and a **Value Source** selection for each.



For each parameter you choose the value source:

- **A tag** — the tag's value is passed in. If the parameter is an **output** parameter, the database's returned value is written back to that tag during write-back.
- `-- NULL --` — passes a NULL value into the database and *inhibits the return write* (if applicable).
- **Constant** — a fixed value you type.
- **Function** — a system-provided value (date/time with a format you specify).

> **Ref-cursor (cursor) parameters don't appear here.** If a procedure returns a cursor through an output parameter — Oracle `SYS_REFCURSOR`, PostgreSQL `refcursor` — its rows are a *result set*, not a scalar value, so Event Engine hides it from this grid and handles it in the **Result Sets & Cursors** section below. The parameter is still bound to the procedure (so the database returns the cursor); it simply isn't tag-mapped here.

The **Refresh** button re-reads the available procedures and the parameter definitions from the database: the **first press refreshes the procedure and parameter lists while keeping your existing tag selections; pressing it again also runs the tag-matching algorithm** (best-match by name against the master tag's members — note this overwrites prior selections).

> **Database contact is deferred.** Opening the **Details**, **Edit**, or **Delete** view never contacts the database — the procedure and parameters are shown from saved configuration, so these views still work when the target database is offline. On the **Edit** view, the live procedure list is loaded only when you press **Refresh** (or change the **Db Connection**).

> **Type matching matters:** Event Engine sends the tag value using the SQL type the database declares. If a tag's PLC type can't represent the SQL type (or vice versa), the event fails with a parameter error — keep PLC structure members and procedure parameters type-aligned. For exactly how nulls, empty strings, and each PLC type are converted and bound, see **§10.5 Parameter Value Binding & Type Conversion** at the end of this chapter.

Step 3 — Result Sets & Cursors (optional)

A **result set** is tabular output a procedure returns *independent of its scalar parameters* — rows written into a target tag rather than into individual output parameters (e.g., a `SELECT` at the end of the procedure returning a recipe table). Click **Add New Result Set** for each set the procedure returns, **in the order the procedure returns them**, and choose the **Result Set Target Tag** where the rows are written.

Three rules govern this section, each with its own heading below. The procedure must return the **same number of result sets on every execution**; the target tag may be **any tag whose shape can hold the set**; and whatever the set does not fill is **overwritten with default values**, not left as the last execution wrote it.

How tabular results reach Event Engine depends on the database:

Database	How tabular results come back
SQL Server / MySQL	Directly, from `SELECT` statements in the procedure body.
Oracle	Through an `OUT SYS_REFCURSOR` parameter the procedure `OPEN`s.
PostgreSQL	Through a `refcursor` parameter, or a `SETOF` / `TABLE`-returning routine.

Cursor parameters are mapped for you. When a procedure has a ref-cursor output (Oracle `SYS_REFCURSOR`, PostgreSQL `refcursor`), Event Engine automatically adds a result set for it, pre-named after the parameter — you only need to choose the **Result Set Target Tag**. The cursor parameter is hidden from the parameter grid (Step 2) and is *not* tag-mapped, because its payload is the result set, not a scalar value.

Every execution must return the same number of result sets

Conditional result sets are not supported. The number of result sets you add here must equal the number the procedure returns, **on every single execution** — not usually, not on the happy path. Sets are paired with what comes back *by position*, so if the counts disagree there is no way to know which returned set was meant for which tag, and no way to tell whether the event did what it was configured to do.

A count mismatch therefore **fails the event**. It is not a warning and the event does not partially complete: the completion tag receives an error code and the status message tag receives the failure text, exactly as for any other event failure. The message names the procedure, both counts, and the input values that were sent, so a conditional `OPEN` or `SELECT` can be traced back to the code path that produced it without a SQL trace:

> Stored procedure "usp_GetRecipe" returned 1 result set(s) but 2 are configured. The counts must match exactly. Inputs sent: [@LineId=3, @PartNumber=A-1044].

Event Engine counts what the database **actually sends**, not what the procedure intended to send. The procedure patterns that break this rule, and what to write instead:

Pattern that breaks it	What to do instead
`IF … SELECT … ELSE SELECT … SELECT …` — branches returning a *different number* of sets	Make every branch return the same number. A branch with nothing to report returns an **empty** set (`SELECT <columns> WHERE 1 = 0`), not no set.
A cursor `OPEN`ed only when a row was found	`OPEN` it unconditionally over a query that is allowed to return zero rows. An empty set is perfectly valid.
A diagnostic or debugging `SELECT` left in the procedure	Remove it, or assign it into a variable so it is not sent — otherwise add a result set for it.
An error handler that returns an extra set on failure	Report errors through an output parameter and the completion / status message tags instead.

An empty result set costs nothing and keeps the count constant. It is the shape to reach for whenever a code path has no rows to return — and because an empty set default-fills its target (see below), the controller sees a cleared destination rather than stale data.

Choosing the Result Set Target Tag — any tag whose shape fits

The **Result Set Target Tag** picker offers **every tag in the system**, not just structures. There is only one requirement: **the tag's shape must be able to hold the shape of the result set.** A structure, a tag collection, an array of atomic values and a single atomic tag are all valid targets:

Target tag	What it holds	Reach for it when the set returns
Structure (UDT) array — `Elements` > 1	One row per element, up to `Elements`. Each row's columns fill that element's members by name.	Many rows of several columns — a recipe table, a list of parts.
Single structure (UDT) — `Elements` = 1	Exactly one row; its columns fill the members by name.	One row of several columns.
Tag collection (Custom) container — `Elements` > 1	One row per collection element, up to `Elements`.	The same as a UDT array, where the destination is a tag collection rather than an imported UDT (Chapter 8).
Atomic array — `Elements` > 1	Every value the set produces, laid into the array in order — row by row, and the columns within each row.	A single column over many rows — serial numbers into a `STRING[50]`.
Single atomic tag — `Elements` = 1	Exactly one value: one row, one column.	A single value — a count, a next serial number, a status code.

Map the container, not one of its elements. Choosing `Recipe[0]` makes the target a *one-row* destination, so a second row has nowhere to land. Pick the container (`Recipe`, shown in the picker with its element count as `Recipe[10]`) whenever the set can return more than one row.

A set too large for its target fails the event — data is never silently dropped. More rows than the target holds, or more than one value for a single-value tag, stops the write-back with a message naming the tag, its capacity, and what the set actually returned:

> Result set larger than destination: tag "Recipe" holds 10 collection element(s) but the result set returned 25 rows. Increase the tag's element count, or map the result set to the collection container tag rather than a single element.

Raise the tag's element count, narrow what the procedure returns, or map the container instead of a single element.

> **Members are matched to columns by name** (case-insensitive), and the match is asymmetric. The target structure may declare **more members than the result set returns** — any member with no matching column is left at its **default value** (`0` , empty string, …) and the rest of the row is still written, so the structure can be a superset of the result set. The reverse is not allowed: a result **column with no member to land in fails the write-back** with an "*Unmatched columns*" error rather than silently dropping data. So a member stuck at its default is usually a **name mismatch** (defaulted silently), whereas an extra column is reported as an error.

> ⚠️ **Use caution and ensure proper data-type mapping** between the result columns and the target structure members.

Rows the result set does not supply are written as default values

Event Engine fills the whole target, every time. Where the result set runs out, the balance of the target is **overwritten with default values** — `0` for numerics, empty for strings — not left holding whatever was there before:

- A 10-element target that receives **4 rows** has those 4 elements written from the rows, and the **remaining 6 written as empty**.
- A result set that returns **no rows at all clears the entire target**.

This is deliberate, and it is the safe behavior. Target tags are long-lived: leaving the unfilled balance untouched would mix *this* execution's rows with the *previous* execution's inside one tag, and the controller has no way to tell which rows belong to which run. Clearing the balance means the target always holds exactly what this execution returned, and nothing else.

> **What this means for your PLC logic.** Never treat "the element still has a value" as evidence that a row was returned for it — after a short result set, it will not have one. If the controller needs to know how many rows are real, return the row count from the procedure as an output parameter and read that, or check a per-row key member for its default value.

> **A returned row holding a database NULL is a different case.** No row at all means the query had nothing to say about a destination it is nonetheless mapped to, so that destination is defaulted. A row that *is* returned but whose column is NULL is the query saying *leave this alone*: for a single-value tag, nothing is written and the tag keeps its current value.

The discovery panel — what the procedure actually returns

Above the result-set rows, Event Engine reports what it could work out about the selected procedure's result sets **without executing it**, so you can configure them without reading the procedure's SQL. It appears as soon as a procedure is selected ("*Analyzing result sets…*" while it works) and lists, for each set it

found, the columns and their declared types — `SerialNumber (nvarchar(50))`, `Weight (decimal(10,3))`, and so on.

A badge reports **how trustworthy the answer is**, because a procedure whose result sets depend on its code path cannot always be pinned down:

Badge	Meaning
Determined without executing	The full set was resolved. The count and every column shape are reliable.
From the database catalog	Resolved from the database's own recorded metadata. Also reliable.
Partially determined — more may exist	Some sets resolved; others could not. Treat the list as a floor, not a total.
First result set only — more may exist	Only the first could be resolved.
Not available	Nothing could be determined: "*The result sets could not be determined automatically. Configure them manually below.*"

Anything short of an exact count is shown in amber rather than green. Where a procedure returns different sets on different code paths, the panel covers **every** path — an `IF` / `ELSE` chain contributes both arms — so the shape list can legitimately be longer than the number of sets any single run produces. That is a reporting artifact of the panel, not permission for the counts to vary: if those arms return *different numbers* of sets, the procedure has to be changed before the event will run reliably (see *Every execution must return the same number of result sets* above).

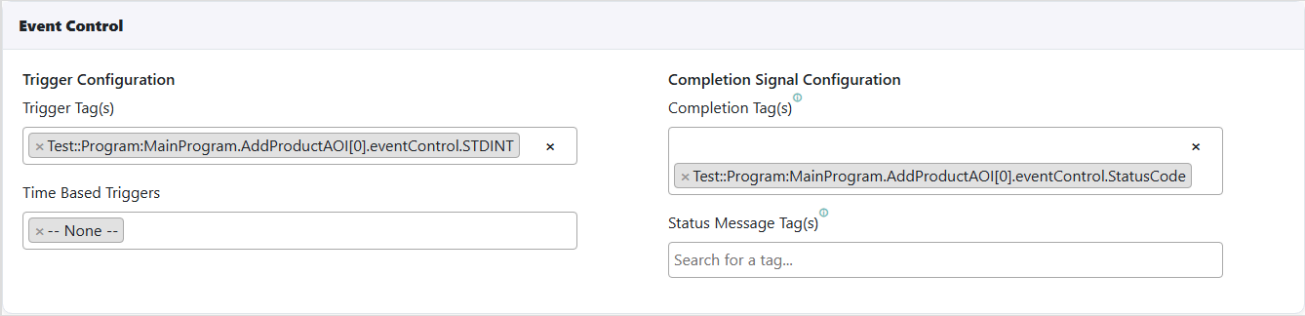
> **This is advisory, and it does not configure anything for you.** The panel tells you what to expect; you still add each result set and choose its target tag, in the order the procedure returns them. If discovery is unavailable or the panel fails, the editor is unaffected — nothing is blocked and no result set is added or removed on your behalf.

> **Availability:** result-set discovery is implemented for **Microsoft SQL Server** connections. On PostgreSQL, MySQL and Oracle connections the panel reports *Not available*; configure the result sets from the procedure's definition as described above. For Oracle and PostgreSQL, the ref-cursor mapping described above already does most of that work.

> **Order is what matters.** Result sets are matched to what the procedure returns *by position*, so keep the rows here in the same order the procedure produces its cursors / `SELECT`s. Position is also why the counts must agree exactly on every execution — see *Every execution must return the same number of result sets* above.

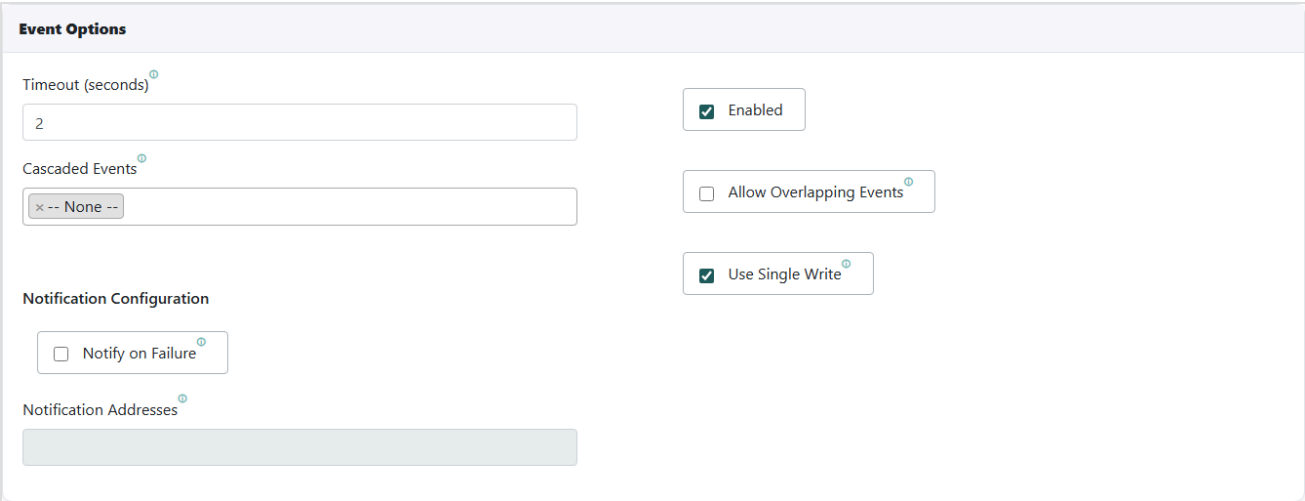
> **Oracle note:** a PL/SQL `BOOLEAN` output parameter cannot be returned across most database drivers. If an event fails to bind a boolean, change the parameter to `NUMBER(1)` (0/1) in the procedure.

Step 4 — Event Control



Field	What it does
Trigger Tag(s)	One or more trigger tags (Chapter 7) that fire this event. With a Master Tag selected, the list is filtered to the structure's Trigger members.
Time Based Triggers	One or more TOD trigger templates that fire this event on schedule. An event may have tag triggers, time triggers, or both.
Completion Tag(s)	Integer tag(s) that receive the completion code. **Codes: 0 = In-Process, 1 = Success, codes < 0 = Error Code.** The PLC should set its trigger and then watch the completion tag.
Status Message Tag(s)	String tag(s) that receive a status message — e.g., an error message — back in the controller with supplemental detail about the event's outcome.

Step 5 — Event Options



Option	Default	What it does and why it matters
Timeout (seconds)	2	How long the system waits for the event to complete before aborting (1 second or more; there is no practical upper limit, so a long-running batch procedure can be given the time it actually needs). **The default is deliberately short and is often too short for a real procedure** — size it above your worst-case procedure duration before going live. **Caution:** the results of an aborted event may be uncertain (the database call may still have committed).
Cascaded Events	none	Events fired automatically upon the **successful** completion of this event. Multiple selections allowed. Use to chain a label print after a successful DB transaction, etc.
Notify on Failure	off	When checked (requires SMTP configured in System Settings), a failure email is sent to the **Notification Addresses** (semicolon-separated) each time this event fails.
Enabled	on	Disabled events never fire and don't count against scanning, but remain configured. Note: enabled-event count is what your event license meters.
Allow Overlapping Events	—	If selected, overlapping triggers yield multiple concurrent executions of this event. If deselected, subsequent triggers are **ignored** until the current execution completes. Leave off for handshake-driven transactions; consider on for queue-style tables keyed by data.
Use Single Write	—	Consolidates payload write-backs with the completion-code write into one write operation, improving event speed. Deselect to perform a separate write verification instead.

Click **Save**. The event appears in the event list, and (while the system is scanning) is live immediately.

10.3 Maintaining Stored Procedure Events

- **Index** (**Events → Stored Procedure Events**): the per-type list with the same live statistics as the master event list (Chapter 14).
- **Details / Edit / Delete** — standard lifecycle. Deleting prompts for confirmation; an event's references (tags, connection) remain untouched.
- **Duplicate** — clones an event under a **new name, description, and a new Master Tag**; the tag bindings are re-pointed at the corresponding members of the new master structure. This is the fast way to roll the same transaction out across many identical stations.
- **Export / Import** — events can be exported to JSON and imported on another system (Events → All Events → Import).

10.4 Runtime Behavior Summary

1. Trigger fires (tag condition or TOD).
2. Completion tag is set to **0** (In-Process).
3. Parameter tags are collected from the PLC.

4. The procedure executes on the configured connection (bounded by Timeout).
5. Output parameters / result sets are written back; with *Single Write*, payload and completion writes are combined.
6. Completion tag is set to **1** on success, or an error code (< 0); the status message tag gets the error text. Telemetry records the run and, on failure, the error appears on the dashboard, in the event's telemetry, and (optionally) in a notification email.
7. On success, any **Cascaded Events** fire next.

For a diagrammed, step-by-step view of this flow — including which party (PLC, Event Engine, database) performs each step — see Chapter 2.2.

10.5 Parameter Value Binding & Type Conversion

When the procedure runs, Event Engine converts each tag value to the parameter's declared SQL type before binding it. Most conversions are direct, but **null values, empty strings, and PLC-specific types** are handled deliberately so that ordinary PLC states — an uninitialized date string, an empty buffer, a boolean — don't fail the call. The behavior below is for the **Oracle** connector; other databases bind the value using the declared SQL type with their provider's native null handling.

> **Why this matters for PLCs:** controllers have no native date/time type, so date tags are *strings* that are **empty on power-up or reset** — that is the correct, expected state, not bad data. Oracle's client driver rejects an empty string (and a NULL) for its date/time types, so Event Engine substitutes a safe default instead of failing the event.

Null and empty tag values

Tag value ↓ / Parameter type →	**Date / Timestamp**	**Number (`NUMBER`, `BINARY_FLOAT/DOUBLE`)**	**Character (`VARCHAR2`, `NVARCHAR2`, `CHAR`, `CLOB`)**	**Binary (`RAW`, `LONG RAW`, `BLOB`)**
Populated	Parsed to a date/time (string must be a recognizable date, else *that one parameter* errors)	Bound as the number	Bound as text (> 4000 chars auto-promoted to `CLOB`)	A byte array → one `byte[]`; a hex string → decoded to bytes
Empty / whitespace string	**Minimum** (`0001-01-01`)	`NULL`	Empty string (Oracle stores `''` as `NULL`)	`NULL`
Null / `--NULL--`	**Minimum** (`0001-01-01`)	`NULL`	`NULL`	`NULL`

> The temporal **minimum** is `0001-01-01` (and `0001-01-01 +00:00` for *TIMESTAMP WITH TIME ZONE*). It is a sentinel meaning "no value was supplied" — downstream logic, queries, and reports should treat that timestamp as *unset*, not as a real date.

PLC type → Oracle binding

PLC value (CLR type)	Bound as	Notes
`BOOL`	`0` / `1` into a `NUMBER`, or native `BOOLEAN`	A PL/SQL `BOOLEAN` parameter requires **Oracle 23ai or newer**. **For portability across Oracle versions, declare boolean parameters as `NUMBER(1)`** — Event Engine converts the `BOOL` to `0` / `1` automatically, so the same event runs on any Oracle version.
`SINT` / `USINT`	Widened to an integer	These small CLR integer types can't be bound directly by the Oracle driver, so they are widened.
`INT` / `DINT` / `LINT`	`NUMBER`	
`REAL` / `LREAL`	`BINARY_FLOAT` / `BINARY_DOUBLE` / `NUMBER`	
`STRING`	`VARCHAR2` / `NVARCHAR2` / `CHAR` / `CLOB`	Strings longer than 4000 characters are promoted to `CLOB` automatically.
`STRING` holding a date	`DATE` / `TIMESTAMP`	Parsed with an invariant (culture-independent) format; empty → minimum (above).
`STRING` holding **hex**	`RAW` / `LONG RAW` / `BLOB`	Decoded to bytes — the way a RAW/GUID value round-trips through a string tag (see note below).
`SINT[]` (byte array)	`RAW` / `LONG RAW` / `BLOB`	Sent as a single `byte[]`; each element is reduced to its low byte (e.g. `SINT -1` → `0xFF`).
Other scalar **array** tags	PL/SQL **associative array**	For an index-by / `TABLE OF` parameter. Binary targets are the exception noted above — they take one `byte[]`, not an array.

> **Binary values and string tags (RAW / BLOB / GUID).** A binary value written back into a **string tag** is stored as a **hex string** (e.g. a 16-byte GUID → 32 hex characters), and a hex string fed *into* a RAW/BLOB parameter is decoded back to bytes — so the value round-trips cleanly. If a binary parameter receives text that is *not* hex (for example a stringified byte array left by an older write-back), it binds as `NULL`, so a stale value never blocks the event.

> **All parameters are validated together.** If more than one parameter can't be bound, the error message lists *every* offending parameter — its name and value — in a single pass, so you can correct them all at once instead of discovering them one event run at a time.

UDT object-collection parameters (Oracle)

A procedure can accept an **Oracle object collection** — a `TABLE OF` / `VARRAY` of an object type (for example `ee_item_tab` as `TABLE OF ee_item_obj`) — to insert or process **many rows in one call**. When you map a **Structure** or **Collection** tag to such a parameter, Event Engine binds the rows **natively** to the collection; the procedure receives a real Oracle collection, not a string.

This is fully **automatic** — there is nothing to configure and no code to write:

- When the procedure is first used, Event Engine reads the collection's definition (owner, type name, and attributes) from the Oracle data dictionary and builds the binding for it in the background.
- The object's **attributes are matched to the structure's members by name** (case-insensitive).
- The binding is **cached for the life of the running service**, so subsequent events reuse it with no extra work.

Rebuild & restart behavior. Editing the event reuses the cached binding. The binding is (re)built only when needed:

You change…	Effect
Create a **new** procedure event using a collection type not yet used since startup	Built automatically on the event's first run — no restart
Edit the event without changing the Oracle **type's attributes** (e.g. remap a tag, change a result set)	Cached binding is still correct — no rebuild needed
Change the Oracle **type's attributes** (add/remove/retype a column) for a type already used since startup	**Requires restarting the Event Engine service** to take effect

> **Why a restart is required for type changes.** Once a collection type has been bound in a running service, its mapping cannot be swapped out in place — the .NET runtime cannot unload it and Oracle's client keeps a single mapping per type for the life of the process. Restarting the service rebuilds the mapping from the type's current definition. Brand-new types (not yet used since the last restart) do **not** need a restart — they are built on first use. Weigh the trade-off before you `ALTER` a type that is in use: the restart **stops scanning and halts every event on the machine** until the service is back, not only the events that bind this type, and any triggers that fire while it is down are missed. On a running line, schedule it for a maintenance window.

> **Avoiding the restart — version the Oracle type instead of altering it.** The same principle that applies to PLC UDTs (Ch. 8.1) applies here: a restart is only needed when you change a type whose mapping is *already bound in the running process*. You can avoid it by introducing the change as a **new** Oracle type rather than `ALTER`-ing the existing one. Create the revised object and collection types under new names — for example append a revision suffix (`ee_item_obj` → `ee_item_obj_v2`, with `ee_item_tab_v2 AS TABLE OF ee_item_obj_v2`) — and point the procedure (or a new overload) at the `_v2` types. Because the service has not bound those names since startup, the binding is built **on first use** with no reboot. Migrate events onto the new types at your own pace and drop the old types once nothing references them. The cost is the two type versions coexisting for a while; the gain is a zero-downtime change and a clean rollback if the new revision is wrong. This is the recommended way to evolve object-collection types on a system that must stay online.

> **Boolean attributes — use `NUMBER(1)`.** The current Oracle .NET driver cannot bind a native 23ai `BOOLEAN` **attribute inside an object type**. Declare boolean members of a UDT as `NUMBER(1)` (`0` / `1`); Event Engine maps `BOOL` tag values automatically. If a UDT attribute is left as `BOOLEAN`, the event fails with a clear message telling you to change it to `NUMBER(1)`. (A `BOOLEAN` *scalar* parameter or table column is unaffected — only object **attributes** have this limitation.) When a future driver adds support, native `BOOLEAN` attributes work with no change to Event Engine.

Next: Chapter 11 — Events: Database Table Access · Guide home

Chapter 11 — Events: Database Table Access

Events → Table Events

> **Where this fits: Step 7** of the setup roadmap, alongside the other event types. Before building this event, make sure its database connection (Ch. 5), its tags (Ch. 9), and its triggers (Ch. 7) already exist. When it's saved, start and verify it (Ch. 14.2).

11.1 What a Database Table Event Does

A Database Table event reads from or writes to a database **table or view directly** — no stored procedure required. This is Event Engine's *expanded database table support*: where traditional middleware typically requires a stored procedure (and a DBA) for every interaction, a table event is configured entirely in the web UI by mapping columns to tags.

Three operations are supported:

Operation	Direction	What it does
Insert	PLC → DB	Inserts one row, with each included column's value drawn from a tag, constant, function, or the column's database default.
Single Row Read	DB → PLC	Reads one row and writes each selected column's value to an individual target tag.
Multi-Row Read / Result Set	DB → PLC	Reads up to *Read Row Limit* rows and writes them into a target tag **structure/array**.

> Other operations (Update, Upsert, Delete) are defined in the data model but are not currently available — the UI enforces: "*Only Read and Insert operations are supported.*"

11.2 Creating a Table Event

Events → Table Events → Create (or **Events → All Events → Create → Database Table**)

Step 1 — Details

Field	What it does
Name / Description	Identify the event. Event names must be unique across all event types.
Master Tag	*(Optional)* A tag structure used as the foundation for column/tag matching, exactly as in stored procedure events.

Field	What it does
Database Connection	Which database to use. Selecting it loads the list of tables and views.
Table / View	The database object to operate on, listed as `schema.name`.
Operation	Insert, Single Row Read, or Multi-Row Read / Result Set. The mapping grid below changes with the operation.
Read Row Limit	The maximum number of rows the query may return. **Required for both read operations** and must be 1–1000; the event fails validation without it. For a Multi-Row Read, size it to the target array length in the PLC. For a Single Row Read it is the safety bound on the query — set it to 1 so a filter that unexpectedly matches many rows cannot pull back more than the one row the event writes out.

The workflow is progressive: choose the connection first, then the object, then the operation — each selection enables the next ("*Select connection, object, and operation to configure mappings.*").

Step 2 — Column Mapping / Column Selection

Once connection + object + operation are chosen, Event Engine reads the object's **column definitions from the database** (name, data type, nullability, defaults) and presents the mapping grid. The **Refresh** button re-reads the columns; the first press keeps your selections, a second press also runs the best-match tag matching.

Insert mapping

For each column you decide whether to **Include** it, and if included, its **Value Source**:

- **Tag** — value comes from the selected tag at run time.
- **Constant** — a fixed value (blank means NULL).
- **Function** — a system value: Date, Time, or Date & Time, with a .NET format string you specify.
- **Use Default** — let the database apply the column's default value.

The grid flags **Required** and **Nullable** columns and shows each column's database default. The page validates before saving — e.g., "*Column 'X' requires a non-null value.*" — so you can't save an insert that the database would reject for missing values. Database-generated columns (identity, computed) are handled by the database and are not mapped.

Single Row Read mapping

For each column you want returned, choose the **Target Tag** that receives the value. Columns you leave unmapped are simply not selected.

Multi-Row Read / Result Set mapping

Choose the columns to include and the target tag that receives the rows. Make sure the target's member types and element count line up with the selected columns and the Read Row Limit.

The target does not have to be a structure. As with a stored-procedure result set, the picker offers every tag, and the only requirement is that **the tag's shape can hold the shape of the result set** — a UDT structure or array of structures, a tag collection container, an array of atomic tags, or a single atomic tag for a single value. The full table of shapes is in Chapter 10.2 — *Choosing the Result Set Target Tag*. Map the **container**, not one of its elements: a single element holds exactly one row. A result set larger than its target fails the event with a message naming the tag and its capacity, rather than dropping the surplus.

Members are matched to columns **by name** (case-insensitive). The rule is asymmetric, by design:

- **The structure may have more members than the result set has columns.** Any member with **no matching column is left at its default value** (`0` for numerics, empty for strings, etc.) and the rest of the row is still written. So the structure can be a *superset* of the result set — a Logix UDT member that exists only in the PLC, or a column you chose not to include, is simply defaulted.
- **The result set may *not* have more columns than the structure has members.** If a selected column has **no member to land in**, the read **fails** with an "*Unmatched columns*" error rather than silently dropping the value. Either add the matching member to the structure or deselect that column.

> ⚠️ As with stored-procedure result sets: **ensure proper data-type mapping** between columns and target members. A member that should receive data but stays at its default is usually a **name mismatch** — and note the two failure modes differ: a member with no column is defaulted silently, while a column with no member is reported as an error.

Rows the read does not return are written as default values. The rule above covers *columns*; the same principle applies to *rows*, and it is the one most likely to surprise. Event Engine fills the whole target on every execution: where the rows run out, the balance is **overwritten with defaults** (`0` for numerics, empty for strings) rather than left holding the previous read's data. A 25-element target that matches 4 rows keeps those 4 and **clears the other 21**; a read that matches **no rows clears the entire target**. This is deliberate — these tags are long-lived, so leaving the balance alone would mix this read's rows with the last one's in a single tag with nothing to tell them apart. Do not use "a value is still present" as evidence in PLC logic that a row was read; if the controller needs the count, check a key member of each element against its default. This matters more here than for a stored procedure, because a filtered read legitimately matching fewer rows than last time is the *normal* case, not an error.

Step 3 — Filters (read operations)

Both read operations — Single Row Read and Multi-Row Read — offer a **Filters** table beneath the column mapping. This is how you narrow *which* rows the event reads; with no filters at all, the query simply

returns the first *Read Row Limit* rows of the table or view.

Press **Add Filter** for each condition. Every filter is one row:

Column	What it does
Column	Which column to test. The list offers only the columns you selected for this event — a filter cannot reference a column the event isn't reading.
Operator	The comparison (below).
Value	The value to compare against. Left blank for the two null operators.

Filters are combined with **AND** — every condition must hold for a row to be returned.

The operators

Operator	Meaning	Value box
`=` `<>` `>` `>=` `<` `<=`	Ordinary comparisons	One value
Contains / **Starts With** / **Ends With**	Text matching	One value
In	Matches any of several values	**Comma-separated** — `A,B,C`. At least one value is required.
Between	Inclusive range	**Two comma-separated values** — `10,20`. Both are required.
Is Null / **Is Not Null**	Tests for a null column	*(no value — the box is disabled)*

Values are always sent to the database as **bound parameters**, never pasted into the SQL text, so a value containing a quote or a semicolon is data and cannot alter the statement.

> ⚠️ **Filter values are constants, not tags.** A filter compares against a fixed value you type when configuring the event; there is no option to take the comparison value from a PLC tag. That makes filters right for a *fixed* selection — `Status = 'ACTIVE'`, `Line = 3`, `ScrapCode Is Not Null` — and it means **a table event cannot look a row up by a key the controller supplies at run time**. When the PLC has to choose the row (a recipe keyed by the part number in a tag, say), use a **Stored Procedure event** (Chapter 10) and pass the key as a parameter; that is the supported path for a controller-driven lookup.

An invalid filter fails the event rather than returning wrong rows: a column that isn't among the event's selected columns, an **In** with no values, or a **Between** with fewer than two are each reported against the offending column.

Step 4 — Event Control & Event Options

These cards are identical to stored procedure events — trigger tags / time-based triggers, completion tags (0 = in-process, 1 = success, < 0 = error), status message tags, timeout, cascaded events, failure notification, enabled, allow overlapping, and single write. See Chapter 10, sections 10.2 steps 4–5 for full descriptions.

For a diagrammed view of how data moves between the PLC, Event Engine, and the database for each operation type, see Chapter 2.3.

11.3 Typical Uses

- **Insert**: lightweight data logging — cycle data, alarms, audit rows — straight into a table, with `Date & Time` function columns for timestamps.
- **Single Row Read**: download a *fixed* set of setpoints — one row selected by constant filters (`Line = 3 AND Active = 1`), written to setpoint tags, handshake completed. For a row the **controller** picks at run time, use a stored procedure event instead (see the filter note in 11.2 step 3).
- **Multi-Row Read**: pull a work-order queue or a lookup table into a PLC array in one event, narrowed by filters and bounded by *Read Row Limit*.

11.4 Maintaining Table Events

Index, Details, Edit, Delete, and Duplicate work the same as for stored procedure events (Chapter 10.3). The duplicate function requires a new name and a new master tag so the copy can be re-pointed at another station's structure.

Next: Chapter 12 — Events: ZPL Label Printing · Guide home

Chapter 12 — Events: ZPL Label Printing

Events → ZPL Print Events

> **Where this fits: Step 7** of the setup roadmap, alongside the other event types. A label event needs its tags (Ch. 9), its triggers (Ch. 7), and the **exact Windows printer name(s)** installed on the Event Engine host. No database connection is required. When it's saved, start and verify it (Ch. 14.2).

12.1 What a ZPL Print Event Does

A ZPL Print event sends a **ZPL command string** (Zebra Programming Language) to the printer of your choice when triggered, with live data substituted into the label at print time. Use it to print part labels, pallet tags, license plates, and similar barcode labels directly from a PLC handshake — no print server application required.

The label fields are parameterized with `{{Placeholder}}` markers inside the ZPL text. At print time each placeholder is replaced by its configured value source (tag value, key/value lookup, system function, or constant).

12.2 Creating a ZPL Print Event

Events → ZPL Print Events → Create (or **Events → All Events → Create → ZPL Label**)

EVENT ENGINE™ Home System Configuration Events

Current User: HQ\jsmith

Create New ZPL Label Print Event

Details

Name: Carton Label

Description:

Master Tag: Search for a tag...

Default Printer: -- None -- Copy Refresh

Printer Tag: -- Use Default Printer --

Fall Back to Default Printer

Key/Value Tag: Search for a tag...

ZPL Command Code

```
^XA
^FO50,50^A0N,40,40^FDOrder: {{OrderNumber}}^FS
```

Analyze

Label Fields

	Field Name	Field Alias	Value Source	System Function	Format/Value
≡	OrderNumber	OrderNumber	-- Please Select --	-- None --	
≡	PartNumber	PartNumber	-- Please Select --	-- None --	
≡	Quantity	Quantity	-- Please Select --	-- None --	
≡	SerialNumber	SerialNumber	-- Please Select --	-- None --	

Event Control

Trigger Configuration

Trigger Tag(s): Search for a tag...

Time Based Triggers: × -- None --

Completion Signal Configuration

Completion Tag(s): Search for a tag...

Status Message Tag(s): Search for a tag...

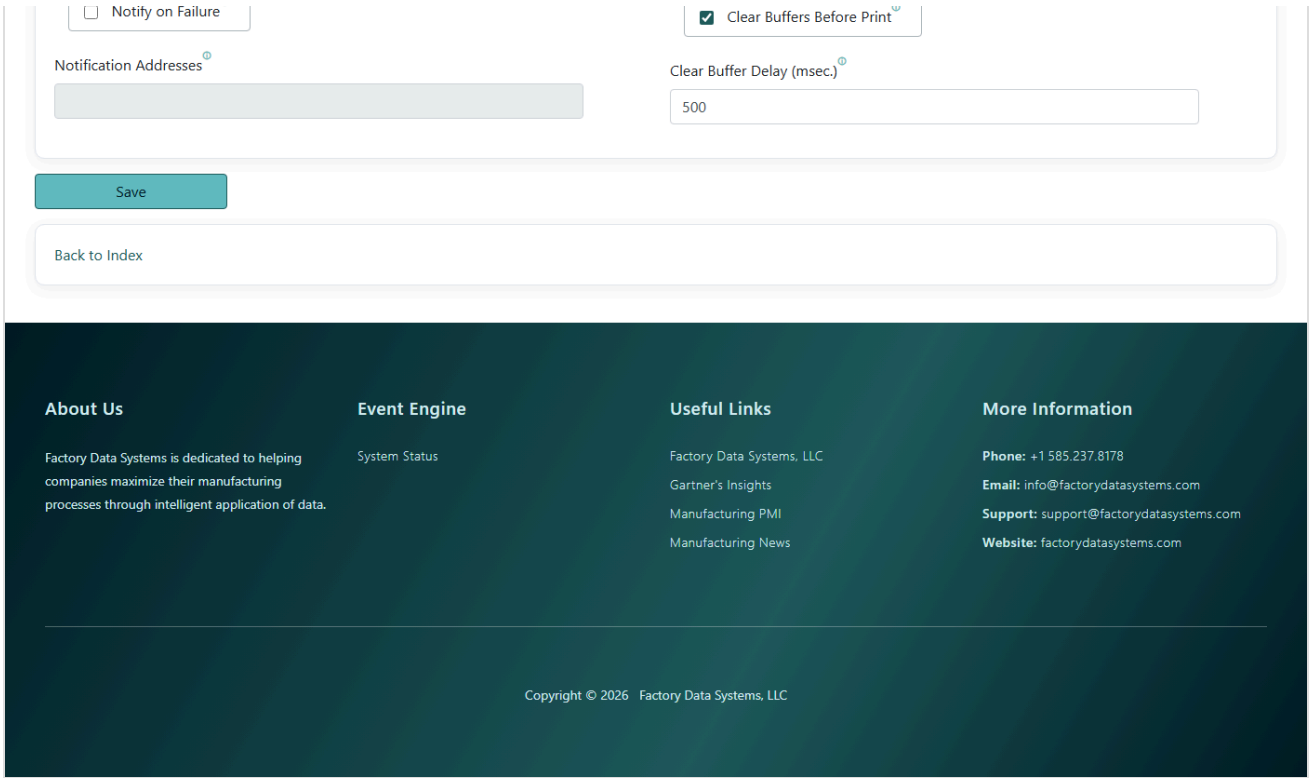
Event Options

Cascaded Events: × -- None --

Enabled

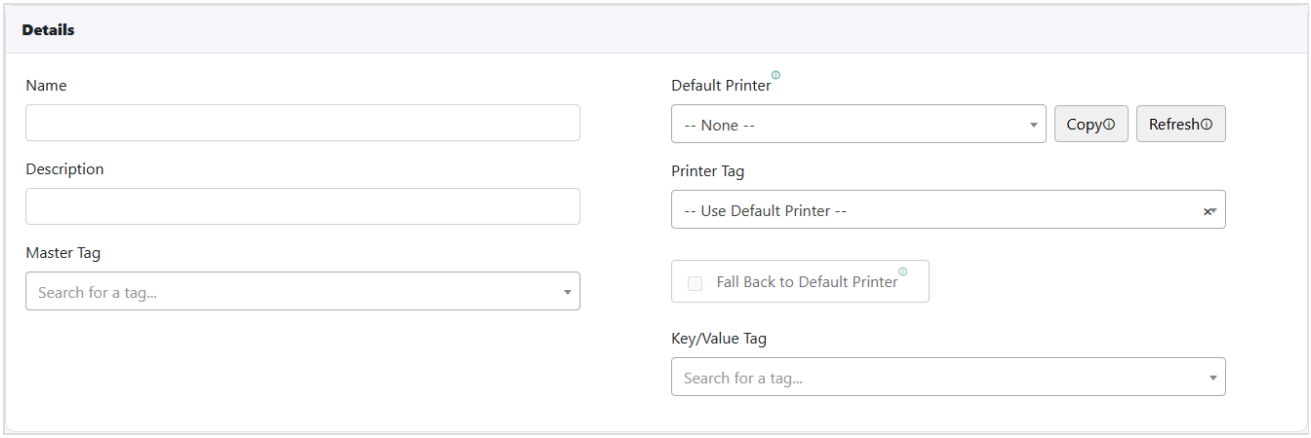
Allow Overlapping Events

Notification Configuration



Step 1 — Details

Field	What it does
Name / Description	Identify the event.
Master Tag	*(Optional)* A tag structure used as the foundation; filters the trigger/completion/message/printer/key-value tag selections to the structure's members.
Default Printer	Any installed or network-shared printer available to the Event Engine host. The **Copy** button copies the selected printer's name (handy for populating a PLC string); **Refresh** re-queries the host for installed printers (the list also refreshes automatically every ten minutes).
Printer Tag	*(Optional)* A string tag whose value should match an installed printer name. When set, the **PLC chooses the printer at run time** — e.g., each station writes its own printer name. Select `-- Use Default Printer --` to always use the default printer. Either a default printer or a printer tag is required.
Fall Back to Default Printer	*(Off by default)* What happens when the printer chosen for a label can't take the job — see When the Chosen Printer Is Unavailable under 12.3.
Key/Value Tag	*(Optional)* A tag containing key/value pairs that can be matched to field names (see *Key/Value Tag* under step 3).

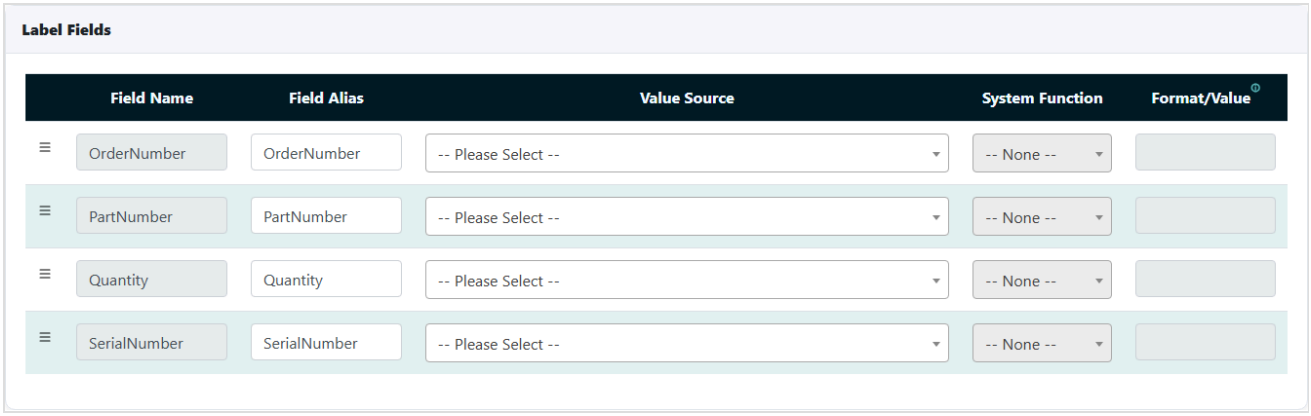


Step 2 — ZPL Command Code and Analyze

Paste your ZPL into the **ZPL Command Code** box, marking every variable field with double-brace placeholders:

```
^XA
^FO50,50^A0N,40,40^FDPart: {{PartNumber}}^FS
^FO50,110^A0N,40,40^FDSerial: {{SerialNumber}}^FS
^FO50,170^A0N,40,40^FDDate: {{PrintDate}}^FS
^FO50,230^BY3^BCN,100,Y,N,N^FD{{SerialNumber}}^FS
^XZ
```

Click **Analyze**. Event Engine scans the ZPL text, extracts the distinct placeholder names, and builds the **Label Fields** table — one row per placeholder. Re-running Analyze after editing the ZPL reconciles the table: existing field configurations are kept for placeholders that still exist, and new placeholders are added.



Step 3 — Label Fields

For each field:

Column	What it does
Field Name	The placeholder name found in the ZPL (read-only). This is the name that is matched — what gets replaced on the label is `{{Field Name}}`.
Field Alias	A friendly display name. It labels the row **on this page only**: it is not what the placeholder matches, and changing it does not change what is printed.
Value Source	Where the printed value comes from: a **tag**, **Key/Value Tag** (see below), **Function**, or **Constant** — see the note under the table.
System Function	When the source is Function: **Date**, **Time**, or **Date & Time** — each with the format you specify.
Format/Value	For functions, the .NET date/time format string (e.g., `yyyy-MM-dd HH:mm:ss` — the column header links to Microsoft's format-string reference). For constants, the constant value itself.

Choosing a Value Source. It is a single searchable dropdown offering the tags and the other kinds of source together. Open it and a **Value Types** group is listed first — **Constant**, **Function** and **Key/Value Tag** — with the tag list beneath it; type to search the tags by name. When a Master Tag is set, its members are grouped separately from the rest. A field nobody has configured yet reads `-- Please Select --`.

Key/Value Tag: the system searches the configured Key/Value Tag for an element whose *key* matches the **Field Name**; if found, the corresponding *value* is printed, and if not found an empty string is substituted. This lets one generic label event print arbitrary fields driven entirely by PLC data.

Step 4 — Event Control & Event Options

Event Control (trigger tags, time-based triggers, completion tags, status message tags) works exactly as in Chapter 10.

Event Options is a *printer-specific* card, not the standard one. It offers:

Option	Default	What it does
Cascaded Events	none	Events fired on this event's successful completion.
Notify on Failure / **Notification Addresses**	off	Failure email, as in Chapter 10.
Enabled	on	Disabled events never fire but stay configured.
Allow Overlapping Events	off	Whether a new trigger may start while a print is still running.

Option	Default	What it does
Clear Buffers Before Print	**on**	Sends a ZPL "clear buffers" command to the printer before the label. Guards against stale jobs in the printer's buffer reprinting old data after a fault.
Clear Buffer Delay (msec.)	500	The delay after the clear-buffer command before the print command is sent, giving the printer time to flush. Only applies while Clear Buffers is on.

There is deliberately no **Timeout** on this card. The Timeout offered on the stored-procedure and table-access cards is a *database command* timeout — it bounds the call those events make to the database. A label print makes no such call: it is handed to the Windows print spooler, and a print that cannot reach its printer fails on that hand-off rather than on a clock.

> **Clear Buffers is on by default, and it costs you the delay on every print.** With both defaults, each label waits 500 ms before its print command is sent. That is the safe setting and the right one for most stations; if a fast station needs the half-second back and its printer has no history of stale-buffer reprints, turn Clear Buffers off (which disables the delay box) rather than shortening the delay.

> **There is no *Use Single Write* option here.** That option consolidates a *payload* write-back with the completion write, and a label event writes no payload to the controller — only the completion code and status message. The same is true of Message events (Chapter 13).

12.3 Runtime Behavior

1. Trigger fires; completion tag → 0 (In-Process).
2. Field tags (and the printer/key-value tags) are read from the PLC.
3. The printer is chosen: the Printer Tag's value if one is configured and holds a name, otherwise the **Default Printer**. If that printer can't take the job, see the next section.
4. Placeholders are substituted into the ZPL string.
5. If configured, the clear-buffers command is sent, followed by the delay.
6. The finished ZPL is sent to the chosen printer.
7. Completion code/status message are written back; telemetry records the run — including the **Printer** the label went to.

For a diagrammed view of this flow, see Chapter 2.4.

> **A Printer Tag that reads back empty is not a failure.** It means the PLC didn't override the choice, so the **Default Printer** is used. (A tag that can't be *read* — because its device is unreachable — is a different thing and aborts the event, so a dead PLC never looks like a blank printer name.)

When the Chosen Printer Is Unavailable

A printer is **unavailable** when either of these is true at print time:

- **It isn't a configured printer** — no Windows print queue by that name exists on the Event Engine host. This is the usual case: a Printer Tag holding a typo, a stale name, or a printer that has since been removed.
- **Its queue is offline** — the queue exists but the spooler reports it offline, paused, in error, or being deleted. *(Only checked when **Fall Back to Default Printer** is on — see the third note below.)*

Transient conditions are deliberately **not** treated as unavailable: a printer that is busy, printing, warming up, low on ribbon, or out of media is one an operator clears in a minute, and the label prints when they do.

Fall Back to Default Printer	**Then**	**Result**
Off *(default)*	—	**The event fails.** The Error List and the event's failure detail name the printer, the reason, and the label.
On	Default Printer is available	The label prints on the **Default Printer** and the event **succeeds**. The run records a **Printer Fallback** detail naming the printer that was wanted, why it couldn't be used, and the printer used instead.
On	Default Printer is also unavailable	**The event fails**, naming both printers and what was wrong with each.

> **Off is the right default, and turning it on is a real decision.** A label that comes out of a printer nobody is standing at is a *successful* event with a physical consequence no one sees — a carton ships with the wrong label, or a label sits in an output bin in another building. Failing the event instead puts the problem on the Error List and into the completion status written back to the controller, where somebody acts on it. Turn the fallback on where the Default Printer is genuinely an acceptable destination — a shared label printer beside the line, not one across the plant.

> **The option only does something when a Printer Tag chooses the printer.** An event that names its printer directly has nowhere to fall back to — the Default Printer *is* the printer that failed — so it fails either way. Leaving the box ticked on such an event is harmless, not a fallback.

> **Turning the option on adds an offline check to every print.** Event Engine asks the spooler about the queue before sending the label, within a **150 ms** budget. For a locally installed queue — including a Zebra on a TCP/IP or LAN port — the answer comes back in about a millisecond, because it is read from the local spooler's own records. If the query doesn't answer in time (in practice, a queue *connected to a remote print server* that isn't responding), Event Engine keeps the printer it was going to use rather than rerouting on a guess; the print then fails visibly if the printer really is dead.

12.4 Maintaining ZPL Events

Index, Details, Edit, Delete and **Duplicate** work as for stored procedure events (Chapter 10.3). Duplicate asks for a new name and a new master tag and re-points the field, printer, key/value, trigger, completion and message bindings at the corresponding members of the new structure — the fast way to roll one label out across identical print stations. Events can also be exported to and imported from JSON via **Events** → **All Events**.

12.5 Tips

- Test labels by leaving the printer powered with no media error — the event only confirms delivery to the print path, not physical print quality.
- Keep `{{` and `}}` exactly double — single braces are ordinary ZPL text and are not parsed.
- Printer names must match Windows printer names exactly when driven from a Printer Tag — use the **Copy** button next to the printer list to capture the exact string for your PLC program.
- **A label event that suddenly fails with a printer message is usually a name, not a printer.** Check the Printer Tag's value against the **Default Printer** list first (**Refresh** it — the cached list is up to ten minutes old), then whether the queue is offline in Windows.
- **Check event history for a *Printer Fallback* detail** on events with the fallback turned on. A run of them means the primary printer is not coming back on its own — the labels are printing, so nothing else will tell you.

Next: Chapter 13 — Events: Message Notifications · Guide home

Chapter 13 — Events: Message Notifications

Events → Message Notifications

> **Where this fits: Step 7** of the setup roadmap, alongside the other event types. A message event needs its tags (Ch. 9), its triggers (Ch. 7), and — before anything can send — the **SMTP relay** configured once in **System → Settings** (Ch. 3.3). No database connection is required. When it's saved, start and verify it (Ch. 14.2).

13.1 What a Message Notification Event Does

A Message Notification event sends a **parameterized SMTP email** to the recipients of your choice when triggered — "*Notify others with messaging when a significant event occurs.*" Like ZPL labels, the subject and body support `{{Placeholder}}` substitution, so the message carries live values from the control system: alarm details, batch numbers, quality results, etc.

> **Prerequisite:** the SMTP Relay settings in **System → Settings** must be configured first (Chapter 3.3). All message events send through that relay, from the configured **From Address**.

Message events are distinct from the built-in notifications (startup/shutdown/fault messages and per-event *Notify on Failure*): a Message event is a first-class event with its own triggers, completion handshake, and telemetry.

13.2 Creating a Message Event

Events → Message Notifications → Create (or **Events → All Events → Create → Message**)

The page is five cards, filled in from the top — **Details**, **Message Body**, **Message Fields**, **Event Control** and **Event Options** — with a single **Save** button beneath them. Nothing is written until you press Save.

EVENT ENGINE™ Home System Configuration Events

Current User: HQ\jsmith

Create New Message Notification Event

Details

Name

Description

Master Tag

Search for a tag...

Key/Value Tag

Search for a tag...

Subject

To Address

CC Address

Message Body

Line {{LineNumber}} completed order {{OrderNumber}}.

Analyze

Message Fields

Field Name	Field Alias	Value Source	System Function	Format/Value
CompletedAt	CompletedAt	-- Please Select --	-- None --	
LineNumber	LineNumber	-- Please Select --	-- None --	
OrderNumber	OrderNumber	-- Please Select --	-- None --	
PartNumber	PartNumber	-- Please Select --	-- None --	
Quantity	Quantity	-- Please Select --	-- None --	

Event Control

Trigger Configuration

Trigger Tag(s)

Search for a tag...

Time Based Triggers

× -- None --

Completion Signal Configuration

Completion Tag(s)

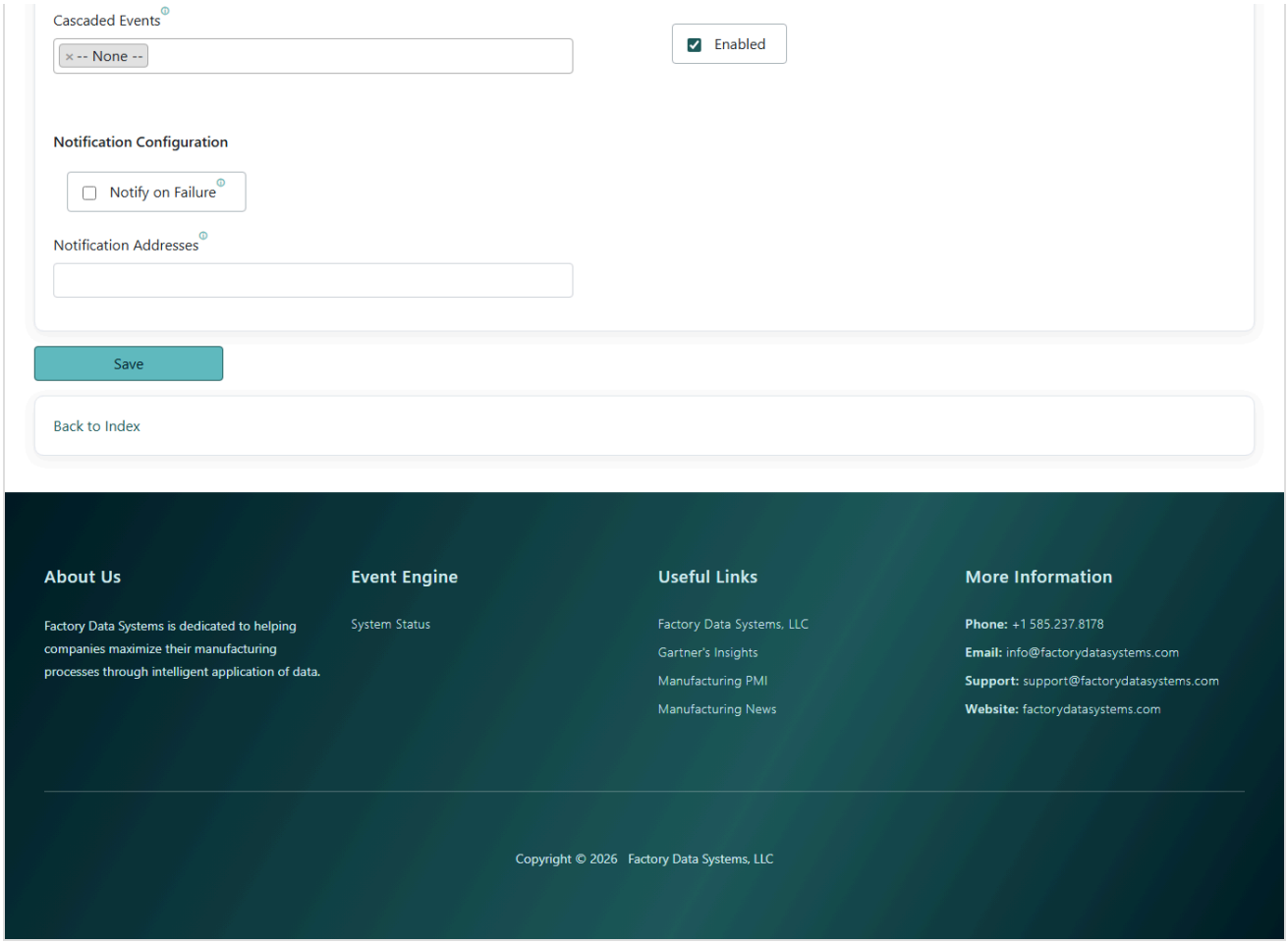
Search for a tag...

Status Message Tag(s)

Search for a tag...

Success is reported when the message is queued: completion code 1 tells the controller the email was built and accepted into the send queue — not that it was delivered. A delivery failure at the SMTP relay afterwards is recorded in the event's history, but the completion code is not changed.

Event Options



Step 1 — Details

The first card identifies the event and addresses the mail. Name, Description, Master Tag and Key/Value Tag are in the left column; Subject, To Address and CC Address in the right.

Field	What it does
Name / Description	Identify the event.
Master Tag	*(Optional)* A tag structure used as the foundation; filters trigger/completion/message/key-value selections to the structure's members.
Key/Value Tag	*(Optional)* A tag containing key/value pairs matched to field names at send time (same mechanism as ZPL events) — see *Value Source* in step 3.
Subject	The email subject. May contain `{{Placeholders}}`; Analyze reads it along with the body.
To Address	One or more valid email addresses — separate multiple entries with a semicolon (`;`).
CC Address	Optional CC list, also semicolon-separated.

Step 2 — Message Body and Analyze

The **Message Body** card holds the body of the email and the **Analyze** button that reads it. Mark every variable part with double-brace placeholders:

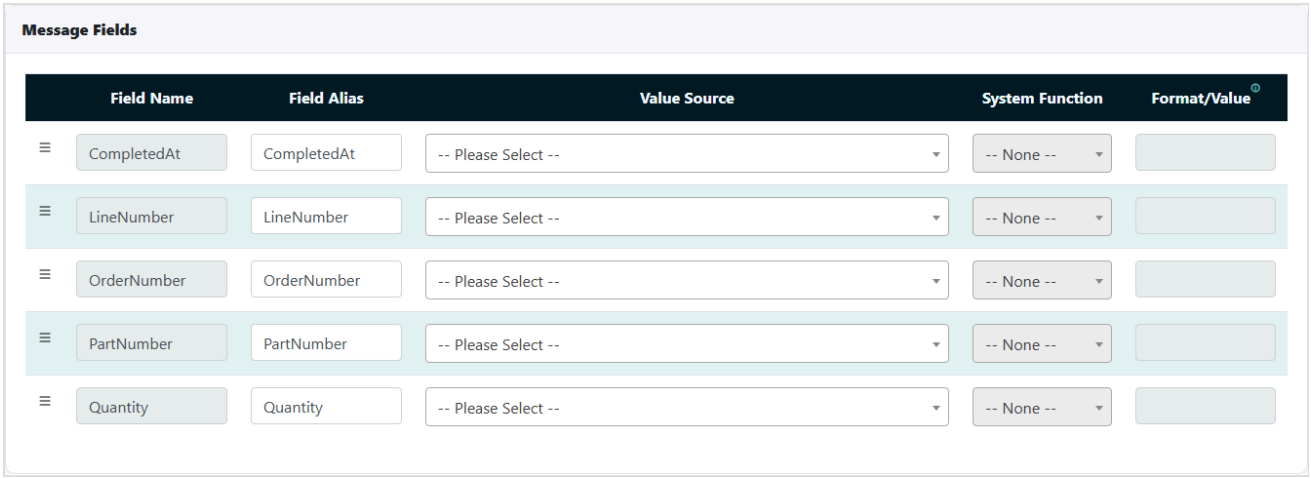
```
Machine {{MachineName}} reported fault {{FaultCode}} at {{FaultTime}}.

Batch: {{BatchNumber}}
Operator: {{OperatorId}}
```

Click **Analyze**. Event Engine scans **both the Subject and the Body** for `{{Placeholder}}` markers and builds the **Message Fields** table below — one row per distinct placeholder. Re-analyzing after edits keeps existing field configurations for placeholders that still exist and adds new ones.

> The Message Body card also appears on the **Details**, **Delete** and **Duplicate** pages, in the same place and read-only, so you can see what an event actually sends without opening it for editing.

Step 3 — Message Fields



One row per placeholder, in five columns:

Column	What it does
Field Name	The placeholder name found in the Subject or Body (read-only). This is the name that is matched — what gets replaced in the message is `{{Field Name}}`.

Column	What it does
Field Alias	A friendly display name. It labels the row **on this page only**: it is not what the placeholder matches, and changing it does not change a single character of the message that is sent.
Value Source	Where the value comes from — see below.
System Function	When the source is *Function*: **Date**, **Time**, or **Date & Time**.
Format/Value	For functions, the .NET date/time format string (e.g. `yyyy-MM-dd HH:mm:ss` — the column header links to Microsoft's format-string reference). For constants, the constant value itself.

Value Source is a single searchable dropdown that offers the tags and the other kinds of source together. Open it and a **Value Types** group is listed first — **Constant**, **Function** and **Key/Value Tag** — with the tag list beneath it; type to search the tags by name. When a Master Tag is set, its members are grouped separately from the rest. A row nobody has configured yet reads `-- Please Select --`.

Value Source	Where the value comes from
A **tag**	That tag's value, read at send time.
Key/Value Tag	The Key/Value Tag chosen in step 1 is searched for an element whose *key* matches the **Field Name**; the matching *value* is substituted, and an empty string if no key matches. This lets one generic message event carry arbitrary fields driven entirely by PLC data.
Function	Date, Time or Date & Time, rendered with the format string in *Format/Value*.
Constant	The fixed text typed into *Format/Value*.

The same picker, with the same **Value Types** group, configures ZPL label fields (Chapter 12.2 step 3).

Step 4 — Event Control & Event Options

Event Control works as in Chapter 10: trigger tags and/or time-based triggers initiate the message, and completion tags (0 / 1 / error code) and status message tags close the loop with the controller. Beside the completion fields the card carries a standing note that completion code 1 means *queued, not delivered* — the same point made in full below, kept next to the setting it constrains so nobody wires a completion handshake without it.

Event Options is the smallest card of the four event types. Cascaded Events and the **Enabled** checkbox sit at the top, with *Notify on Failure* and its addresses grouped underneath a **Notification Configuration** heading. It offers only:

Option	Default	What it does
Cascaded Events	none	Events fired on this event's successful completion — that is, on successful *queueing* (see below).

Option	Default	What it does
Notify on Failure / **Notification Addresses**	off	Failure email, as in Chapter 10. Both controls are **greyed out until an SMTP relay address is configured** (Ch. 3.3) — there would be nothing to send them through.
Enabled	on	Disabled events never fire but stay configured.

> **Timeout, Allow Overlapping Events and Use Single Write are not configurable for message events**, and are not shown. Message events always allow overlapping executions — a second alarm is never dropped because the first is still being built — and always use a fixed **100-second** timeout, which is generous because the work being timed is only building and queueing the message, not delivering it. *Use Single Write* is absent for the same reason as on ZPL events (Chapter 12.2): there is no payload write-back to consolidate, only the completion code and status message.

> **When is a Message event "successful"?** Success is reported **when the message is queued**, not when it is delivered. Completion code **1** tells the controller the email was built (all substitutions resolved) and accepted into Event Engine's send queue; the hand-off to the SMTP relay and delivery to the recipients happen asynchronously **after** the completion write. Cascaded events likewise fire at successful queueing. If the send then fails — the relay rejects the message, times out, is unreachable, **or no relay address is configured at all** — that run is recorded as **failed** in the event's history and telemetry (Chapter 14) with the reason attached, but the completion code already written to the controller is not rewritten. Design PLC logic accordingly: completion = 1 means "*notification queued*" — use Event History to audit delivery.

13.3 Typical Uses

- **Fault escalation** — trigger on an alarm bit; the body carries fault code, station, and time.
- **Shift/production summaries** — time-based trigger plus tags holding shift counters.
- **Quality alerts** — trigger when a measurement tag exceeds limits (use a tag trigger template with an absolute-change or percent threshold), with the readings embedded in the message.
- **Cascade notifications** — set a Message event as the *Cascaded Event* of a critical database transaction so a confirmation email follows every successful run.

For a diagrammed view of the message flow — PLC to Event Engine to SMTP relay to recipients — see Chapter 2.5.

13.4 Maintaining Message Events

Index, Details, Edit, Delete and **Duplicate** work as for stored procedure events (Chapter 10.3). Duplicate asks for a new name and a new master tag and re-points the field, key/value, trigger, completion and

message bindings at the corresponding members of the new structure. Events can also be exported to and imported from JSON via **Events** → **All Events**.

13.5 Troubleshooting Message Delivery

Symptom	Check
Event succeeds but no email arrives	Completion code 1 means *queued*, not *delivered* (13.2) — **check Event History for that run**, which is where a send failure is recorded. A run reading "*no SMTP relay address is configured*" means the relay was never set (Ch. 3.3); other send failures name what the relay said. If the run really is clean, the message left Event Engine: check spam filters and whether the From Address is acceptable to your relay.
Event fails with SMTP timeout	The relay is unreachable from the Event Engine host, or the **SMTP Operation Timeout** is too short.
Placeholders print literally (`{{X}}`)	The field wasn't configured after Analyze, or braces aren't exactly doubled. Re-run Analyze and assign a value source.
Empty values in the message	Key/value lookup found no matching key (substitutes an empty string), or the source tag read failed — check the event's telemetry errors.

Next: Chapter 14 — The Event List & Telemetry · Guide home

Chapter 14 — Events: The Event List & Telemetry

> **Where this fits: Step 8** of the setup roadmap — the last one. Once the system is scanning (Ch. 3.1), this is where you prove each event works end-to-end and where you return whenever one is slow or failing.

14.1 The Event List (Index) Page

Events → All Events

This is the master list of every configured event of every type, with live statistics. (Each event type also has its own filtered index under the Events menu.)

EVENT ENGINE™ Home System Configuration Events

Current User: HQ\jsmith

Event List

Event List

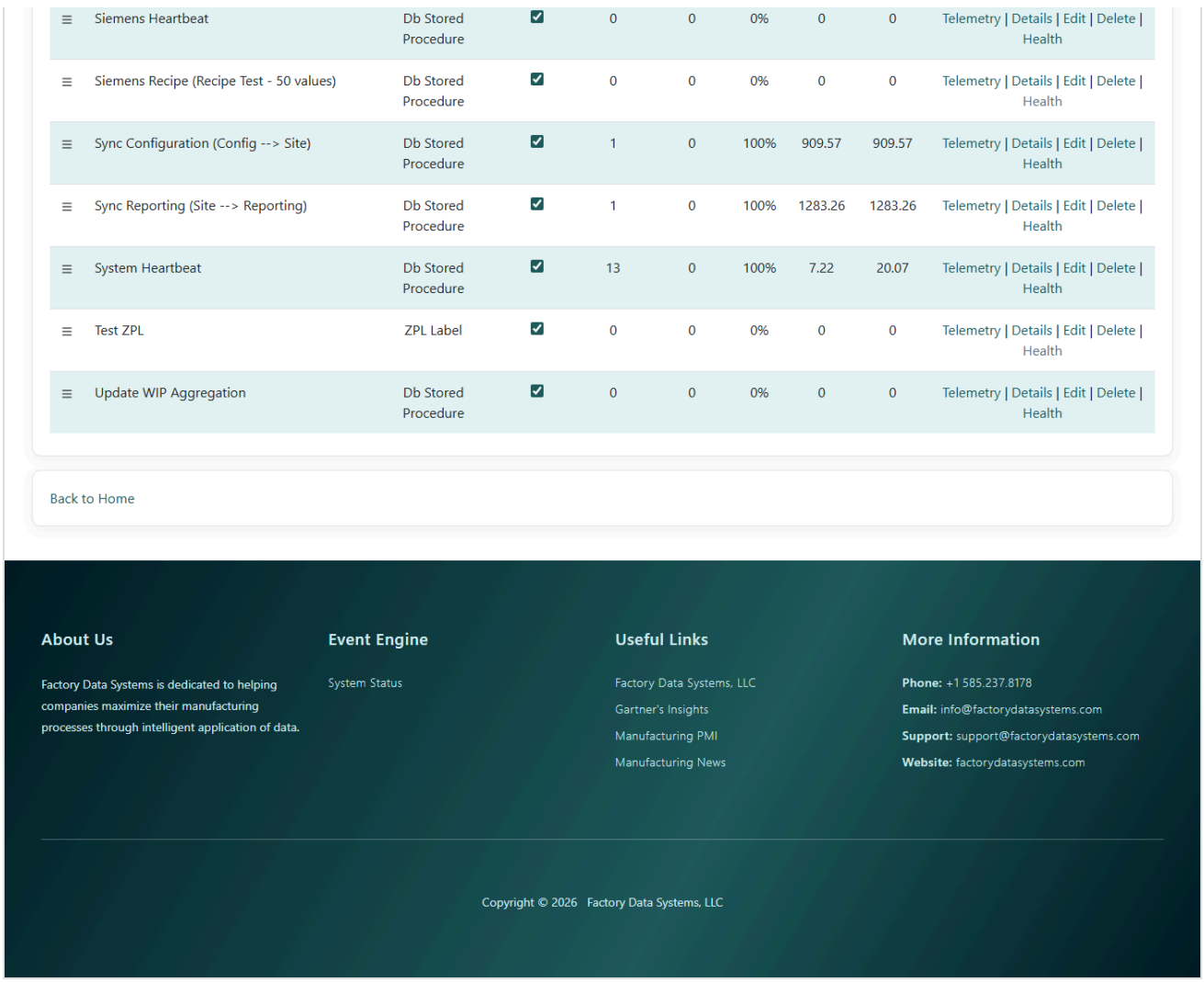
Scanning

Create | Import

Enter Search Phrase...

Enable Selected | Disable Selected | Export Selected

Name	Event Type	Enabled	Total Calls	Failed Calls	Pass Rate (%)	Last (msec.)	Average (msec.)	Action
OP200 - Add Product	Db Stored Procedure	☑	16	0	100%	15.87	27.87	Telemetry | Details | Edit | Delete | Health
OP200 - Check-In	Db Stored Procedure	☑	15	0	100%	36.38	49.42	Telemetry | Details | Edit | Delete | Health
OP200 - Recipe	Db Stored Procedure	☑	2	0	100%	79.71	191.48	Telemetry | Details | Edit | Delete | Health
OP200 - Recipe Monitor	Db Stored Procedure	☑	1	0	100%	194.59	194.59	Telemetry | Details | Edit | Delete | Health
OP200 Recipe_0 (OPC)	Db Stored Procedure	☐	0	0	0%	0	0	Telemetry | Details | Edit | Delete | Health
OP300 - Check-In	Db Stored Procedure	☑	0	0	0%	0	0	Telemetry | Details | Edit | Delete | Health
OP300 - CheckOut	Db Stored Procedure	☑	0	0	0%	0	0	Telemetry | Details | Edit | Delete | Health
OP300 - Recipe	Db Stored Procedure	☑	0	0	0%	0	0	Telemetry | Details | Edit | Delete | Health
OP300 - Recipe Monitor	Db Stored Procedure	☑	0	0	0%	0	0	Telemetry | Details | Edit | Delete | Health
OP400 - Check-In	Db Stored Procedure	☑	0	0	0%	0	0	Telemetry | Details | Edit | Delete | Health
OP400 - CheckOut	Db Stored Procedure	☑	0	0	0%	0	0	Telemetry | Details | Edit | Delete | Health
OP400 - Recipe	Db Stored Procedure	☑	2	0	100%	7317.22	8387.55	Telemetry | Details | Edit | Delete | Health
OP400 - Recipe Monitor	Db Stored Procedure	☑	0	0	0%	0	0	Telemetry | Details | Edit | Delete | Health
OP500 - Add Product	Db Stored Procedure	☑	0	0	0%	0	0	Telemetry | Details | Edit | Delete | Health
OP500 - Check-In	Db Stored Procedure	☑	0	0	0%	0	0	Telemetry | Details | Edit | Delete | Health
OP500 - Genealogy (Cartridges to Cartons)	Db Stored Procedure	☑	0	0	0%	0	0	Telemetry | Details | Edit | Delete | Health
OP500 - Recipe	Db Stored Procedure	☑	2	0	100%	83.45	180.64	Telemetry | Details | Edit | Delete | Health
OP500 - Recipe Monitor	Db Stored Procedure	☑	0	0	0%	0	0	Telemetry | Details | Edit | Delete | Health
Rebuild WIP Updates	Db Stored Procedure	☑	0	0	0%	0	0	Telemetry | Details | Edit | Delete | Health



Columns

Column	Meaning
Name	The event's display name.
Event Type	Db Stored Procedure, Database Table, ZPL Label, or Message Notification.
Enabled	Whether the event will fire. Disabled events are skipped (and don't count against the event license).
Total Calls	Executions since the last counter reset.
Failed Calls	Failed executions since the last reset.
Pass Rate (%)	Percentage of passing executions.
Last (msec.)	Duration of the most recent execution.
Average (msec.)	Average execution duration.

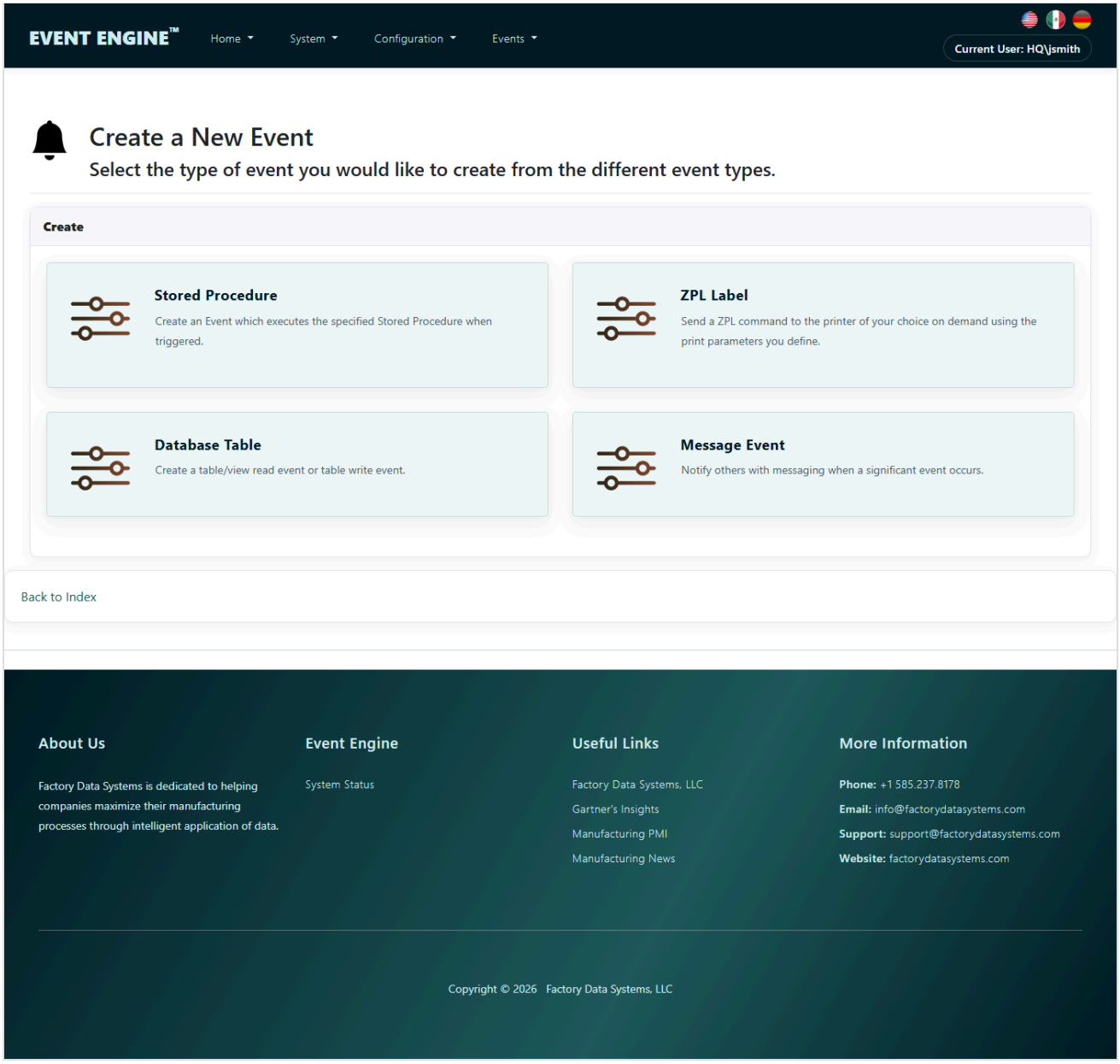
Column	Meaning
Action	**Telemetry** (the event telemetry page, 14.2), **Details**, **Edit**, **Delete**.

Every column is **sortable** (click the sort icon in the header), and the search box filters the list. The statistics columns (Total Calls, Failed Calls, Pass Rate, Last, Average) update **live in place** as events execute — no page refresh — so the list can serve as a live operations display. A **Live** indicator above the table shows the connection state (see Chapter 3, *Live Updates*).

> Live updates change cell *values* only; they never re-sort or re-page the table underneath you. A row whose position is stale under the active sort simply updates in place — re-apply the sort (or reload) to reorder.

Toolbar Functions

- **Create** — opens the event-type chooser (Stored Procedure, Message, ZPL Label, Database Table).
- **Import** — imports events from a JSON file previously exported from this or another system.
- **Selection buttons** — select rows, then: **Enable Selected** / **Disable Selected** (bulk enable/disable; enabling is license-checked), **Export Selected** (JSON download), and **Reset All Counters**.



14.2 The Event Telemetry Page

Events → **All Events** → **Telemetry** (per event)

Each event has its own telemetry page — this is the *advanced telemetry* that sets Event Engine apart, breaking each execution into its PLC-read, work, and PLC-write phases. The whole page is **live**: the status light, every metric, and the error list update on screen as the event executes.

Event Telemetry

OP200 - Add Product

Nominal



Event Telemetry

Scanning

Total Event Calls: 16	Failed Event Calls: 0
Successful Calls: 16	Success Rate (% Passing Events): 100
Minimum Duration (msec.): 15.87	Maximum Duration (msec.): 109.94
Last Call Duration (msec.): 15.87	Average Duration (msec.): 27.87
Last Tag Collection Time (msec.): 8.31	Average Tag Collection Time (msec.): 17.44
Last Work Time (msec.): 3.53	Average Work Time (msec.): 4.93
Last Write Back Time (msec.): 3.76	Average Write Back Time (msec.): 5.14
Time of Last Call: 09/02/2026 04:02:16	Last Count Reset:

Error List

No Current Errors

Back to Index | Details | Edit | Delete | Reset Counts | Manual Trigger | Event History | Health

Copyright © 2026 Factory Data Systems, LLC

Status Light

Light	Meaning
🟢 Green	The system is scanning, the event is enabled, and its recent window is clean.
🟡 Amber	The system is scanning and the event is enabled, but its **recent window** contains a failure.
🔴 Red	The event is disabled — **or the system is not scanning**. A red light here does not by itself mean this event has a problem; check the System Status page first.

The *recent window* is the **last 10 runs, plus everything in the last 10 minutes** — the two combined, not whichever is smaller. So a failure stays amber until ten later runs have gone by *and* ten minutes have passed: on a busy event the run count clears it, and on an event that fires twice an hour the clock does.

Metrics

Metric	Meaning
Total Event Calls / Failed Event Calls / Successful Calls / Success Rate (%)	Counts since the last reset.
Minimum / Maximum / Last / Average Duration (msec.)	End-to-end execution time.
Last / Average Tag Collection Time (msec.)	Time to read all tags associated with the event from the PLC — everything the event needs before it can do its work. High values point at PLC comms: scan loads, connection counts, network.
Last / Average Work Time (msec.)	Time for the event's actual work apart from PLC interaction — executing the stored procedure, printing the label, sending the message. High values point at the database/printer/relay.
Last / Average Write Back Time (msec.)	Time to write results back to the PLC — operation outputs (e.g., database return parameters) and completion codes. The *Use Single Write* option directly reduces this phase.
Time of Last Call / Last Count Reset	Timestamps for context.

Using the three-phase breakdown, you can tell at a glance whether a slow event is a *controller* problem (tag collection), a *business-system* problem (work), or a *write-back* problem — which is usually the first question in any performance investigation.

Error List

Below the metrics is the event's own error history — timestamp, type, and full message for each failure — paged with your items-per-page setting.

Each error is prefixed with the **phase** it occurred in — `[TagCollection]`, `[Work]`, `[Writeback]`, `[Cascade]`, or `[Finalize]` — so a failure identifies *where* in the pipeline it happened, not just that it happened.

Each row's **Details** link opens this event's Event History (14.3) on the run that produced the error, with its row already expanded — the same destination the system Error List (Chapter 3) offers.

> ℹ️ **Overlapping triggers.** When a trigger fires while a previous run of the same event is still in progress and *Allow Overlapping Events* is off, the new trigger is rejected. The rejection appears here (and on the system Error List) as an error naming the run that was still in progress — but it does **not** create an execution record, so it never disturbs the metrics or history of the run you actually need to investigate.

Action Links

- **Event History** — this event's run-by-run history table (14.3), filtered to just this event.
- **Details / Edit / Delete** — jump straight to the event's configuration pages.
- **Reset Counts** *(only while the system is scanning)* — clears this event's counters; sub-metrics start fresh.
- **Manual Trigger** *(only while the system is scanning)* — fires the event immediately, exactly as if its trigger had fired. Invaluable during commissioning: trigger, watch the metrics and error list, adjust, repeat.

> ⚠️ **Manual Trigger executes the real event** — the stored procedure runs, the row inserts, the label prints, the email sends, and completion codes are written to the PLC. Confirm the dialog only when that's what you intend.

14.3 The Event History Page

Events → **Event History** (all events), or **Event History** from any event's Telemetry page (that event only).

Where the Telemetry page (14.2) shows *aggregates*, Event History shows the **individual runs** — one row per execution, newest first, paged with your items-per-page setting. A search box filters the list, and it is reachable either for all events (**Events** → **Event History**) or filtered to one event from that event's Telemetry page.

The table columns are:

Column	Meaning
Log Id	The run's logging id (`EV-...`) — search the system log file for this id to see every log line the run produced.
Event	The event name; links to its Telemetry page. *(Omitted when the page is already filtered to a single event.)*
Start Time	Execution start, to the millisecond.
Dispatch (msec.)	Time between trigger detection and execution start. Delay here is queueing *inside Event Engine*, before any event work began.
Duration / Tag Collection / Work / Writeback (msec.)	The end-to-end time and the three-phase breakdown, per run.
Status	Success, Failed, or In Progress. A run that logged a recoverable error (e.g. a printer fallback) but completed its work still shows Success.

Every row **expands** (the toggle in the first column) to reveal the rest of what telemetry recorded for that run:

In the expanded row	Meaning
Trigger	*Why* the event ran: the trigger tag and the value that tripped it, the TOD trigger name, `Manual trigger by <user>`, or the cascade source. A cascaded run also shows its **parent event id** for correlation.
End Time	Execution end, to the millisecond. Blank while a run is still in progress.
Details	Type-specific facts recorded during the run (below), one line per cascaded child with its outcome and duration, and every error prefixed with its phase — `[TagCollection]`, `[Work]`, `[Writeback]`, `[Cascade]`, `[Finalize]`.

ℹ️ **Arriving from an Error List.** A **Details** link on the system Error List (Chapter 3) or on an event's Telemetry page (14.2) opens this page directly on the run that failed: the page turns to whichever page holds that run, and its row arrives highlighted and already expanded. Paging or searching from there returns to the ordinary view. Telemetry is held in memory in a fixed-size queue and a **Reset Counts** clears it, so a run can age out from under an older link — the page then says the record is no longer held and shows the rest of the history as usual.

The completion code is not on this page. It is captured for every run and written to the **telemetry CSV** (14.4), along with a flag for whether the write-back itself succeeded — see the column list there. On screen, use **Status** for the run's outcome and the expanded **Details** for the reason it failed.

What appears under Details

- **Database events** — the connection and procedure/table used, the command timeout, result-set and row counts (or rows affected), and the split of database time into **Db Connect (msec.)** (waiting for a connection: pool, network, login — an Event Engine/environment delay) versus **Db Execute (msec.)** (the database actually running the command). A slow event tells you at a glance which side owns the delay. **Db Connections Opened** counts how many connections the run acquired, which is what **Db Connect (msec.)** is the total for — so the two read together: a large connect time spread over one acquisition points at a slow login or an exhausted pool (Chapter 5), whereas many acquisitions in one run means the event is making several database round trips. Table events also record the **Operation** (Insert / Single Row Read / Multi-Row Read). On failure, stored-procedure runs snapshot the parameter values that were bound.
- **Tag collection** — how many tags were read across how many devices, the **slowest device read** (so a slow collection identifies the slow PLC), and any significant wait on a device's access gate (contention with scanning).
- **Message events** — recipients, the post-substitution subject, and the body size.
- **ZPL events** — the printer that actually printed, an explicit note when a **fallback** to the system default printer occurred, and the label size.
- **Cascaded events** — one line per child run with its outcome and duration, so a chain's time is attributable to the specific child.

14.4 Telemetry History Files

The in-memory history above is bounded by the **Telemetry Queue Size** setting and clears when the configuration restarts. For history that survives restarts, every completed run is also appended to a **daily CSV file**:

- **Location:** the folder named by **Telemetry File Location** in (Chapter 3.3). When that setting is left blank the files go to a `Telemetry` folder inside the configured log folder (e.g. `C:\ProgramData\Factory Data Systems, LLC\Event Engine\Logs\Telemetry`). One file per day, named `telemetry-YYYYMMDD.csv`.
- **Format:** plain CSV with a header row, one line per run. The files open directly in **Excel** for filtering, pivoting, and charting.
- **Retention:** files older than **Telemetry History Retention (days)** (Chapter 3.3, default 30) are deleted automatically once per day.

The CSV carries **everything the Event History page shows and two things it does not** — the completion code and whether the write-back succeeded. Per run: start and end time, event name and id, run id, telemetry type, status, trigger source, parent event id, dispatch latency, duration, tag-collection / work / write-back times, **completion code**, **write-back succeeded**, the details, cascaded child outcomes, and the errors.

> **This is where you audit completion codes.** Because the completion code is the value the *controller* saw, the CSV is the record to reach for when a PLC-side handshake and Event Engine's own view of a run appear to disagree — and it is the only place the "write-back itself failed" flag is exposed.

The **Telemetry History Files** card at the bottom of the Event History page lists the files newest-first. **Select** opens the file in the same viewer used for system log files — newest records first, loading more as you scroll — and **Download (CSV)** saves the raw file for Excel.

14.5 Telemetry Retention Summary

Store	Scope	Bound	Survives restart?
In-memory queues (Telemetry + Event History pages)	Last *N* runs per event	**Telemetry Queue Size** (Chapter 3.3)	No
Counters (totals, rates)	Since last reset	Reset via **Reset Counts** / **Reset All Counters**	No
Daily CSV files	Every completed run	**Telemetry History Retention (days)**	Yes

---

14.6 Where to Go Next

The pages in this chapter tell you **what happened** — this run, this failure, this hour. They will not tell you that an event has been getting steadily slower for three weeks, because every individual run still looks fine.

That question has its own chapter: **Chapter 15 — Monitoring Event Health**, which reads the same history files described above and reports what has *changed*.

Next: Chapter 15 — Monitoring Event Health · Guide home

Chapter 15 — Monitoring Event Health

The Telemetry and History pages answer *what happened*. This chapter covers a different question: **has anything changed about the way your events are running?**

An event whose stored procedure has slowed from 40 ms to 900 ms over three weeks is still succeeding, so nothing on any other page says a word about it — right up to the day it crosses its command timeout and stops the line. Event Health watches for that kind of drift and tells you while there is still time to act.

Section	
15.1	What Event Health answers
15.2	How the observer works
15.3	**How the numbers are worked out**
15.4	The five states
15.5	The Event Health list
15.6	The Event Health detail page
15.7	What the findings mean
15.8	Why findings are slow to appear and slow to clear
15.9	Health notifications
15.10	Settings

15.1 What Event Health Answers

Events → Event Health

Three questions the other pages cannot answer:

- **Is anything getting worse?** Not "did this run fail", but "is this event slower, or failing more often, than it used to be".
- **Is anything already broken and always has been?** An event that has failed every run for a month never *got* worse, so no comparison will ever mention it.
- **Is the monitoring itself still working?** A monitor that quietly stopped reading shows no problems — and so does a plant where nothing is wrong.

EVENT ENGINE™ Home System Configuration Events

Current User: HQ\jsmith

Event Health

Event Health

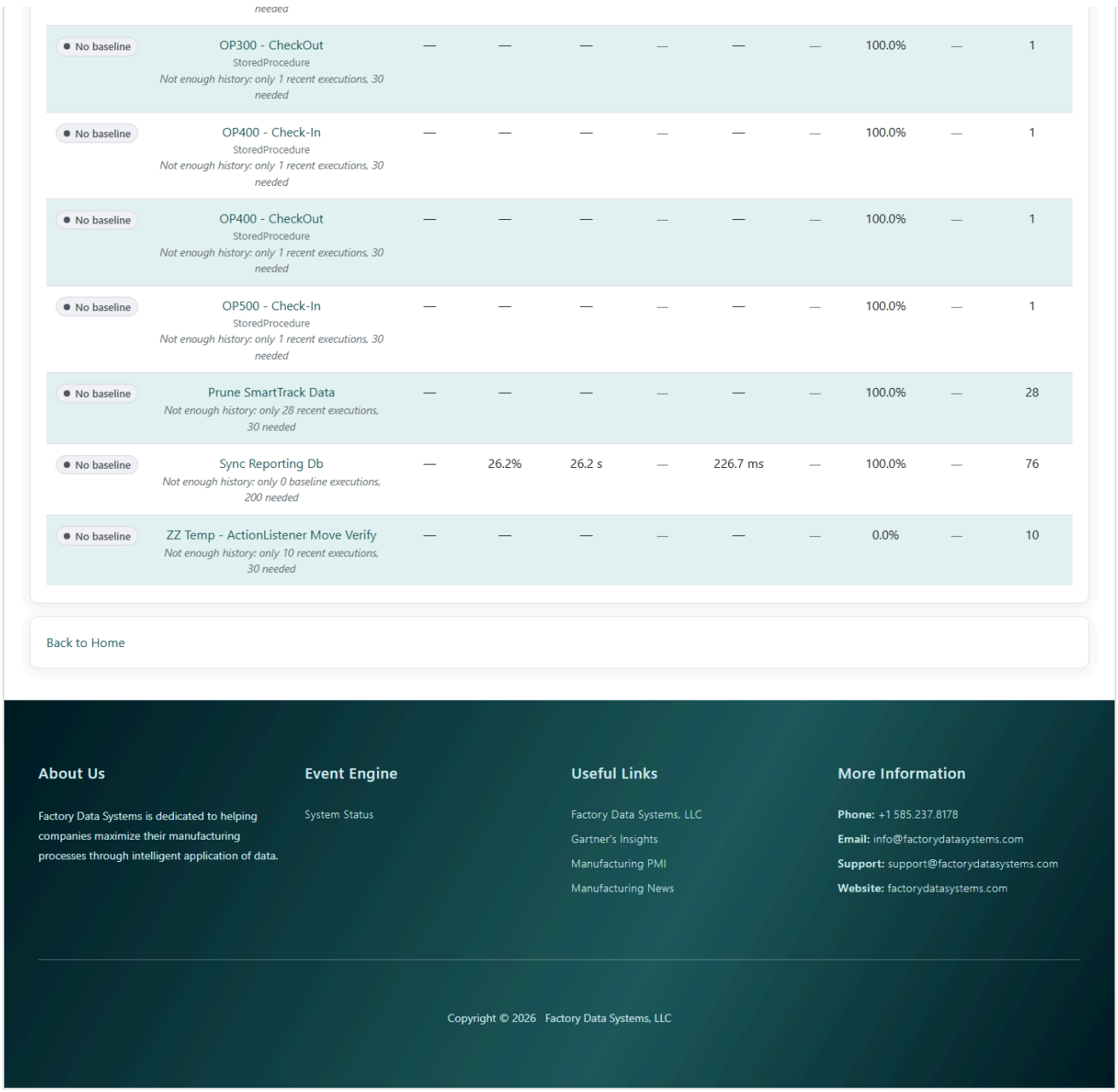
Scanning

All 26 | Needs attention 0 | No baseline 9

Enter Search Phrase...

Observer | Last read 9/2/2026 4:02 PM | Last evaluated 9/2/2026 4:00 PM | Reading history (2 KB remaining) | 26 events, 466 executions read | Resumed from saved state

State	Event	Findings	Timeout used	Duration		Database work		Failure rate		Samples
				p95	vs. baseline	p95	vs. baseline	Recent	vs. baseline	
Nominal	OP200 - Add Product StoredProcedure	—	0.5%	38.2 ms	-36%	10.5 ms	+9%	0.0%	-100%	2,500
Nominal	OP200 - Check-In StoredProcedure	—	1.1%	121.2 ms	+59%	21.8 ms	+9%	0.0%	-100%	2,500
Nominal	OP200 - Recipe StoredProcedure	—	1.3%	275.5 ms	-74%	25.3 ms	-92%	0.0%	-100%	1,593
Nominal	OP200 - Recipe Monitor StoredProcedure	—	5.0%	340.0 ms	-37%	248.1 ms	-48%	0.0%	-100%	2,467
Nominal	OP200 Recipe_0 StoredProcedure	—	0.7%	12.5 s	+68%	13.8 ms	-96%	100.0%	+86%	2,500
Nominal	OP300 - Recipe StoredProcedure	—	0.0%	377.9 ms	-6%	0.0 ms	-100%	96.0%	+5%	2,500
Nominal	OP300 - Recipe Monitor StoredProcedure	—	9.7%	636.9 ms	-41%	484.1 ms	-9%	0.5%	-94%	732
Nominal	OP400 - Recipe StoredProcedure	—	1.2%	8,995.6 ms	-19%	23.2 ms	-93%	47.5%	-40%	2,500
Nominal	OP400 - Recipe Monitor StoredProcedure	—	4.7%	4,979.0 ms	+165%	236.0 ms	-47%	84.0%	+203%	1,158
Nominal	OP500 - Recipe StoredProcedure	—	1.3%	282.1 ms	-48%	25.7 ms	-92%	0.0%	-100%	2,500
Nominal	OP500 - Recipe Monitor StoredProcedure	—	6.4%	419.2 ms	-49%	319.1 ms	-47%	0.0%	-100%	997
Nominal	Rebuild WIP Updates StoredProcedure	—	0.5%	2,765.1 ms	+173%	2,732.3 ms	+651%	0.0%	-100%	988
Nominal	Siemens Heartbeat StoredProcedure	—	8.1%	343.7 ms	+12%	161.6 ms	+15%	0.0%	—	2,500
Nominal	Sync Configuration StoredProcedure	—	1.2%	2,166.1 ms	+33%	2,165.7 ms	+38%	0.0%	-100%	2,487
Nominal	Sync Reporting StoredProcedure	—	0.1%	2,291.2 ms	+16%	2,290.6 ms	+17%	0.0%	-100%	2,477
Nominal	System Heartbeat StoredProcedure	—	0.2%	23.2 ms	-1%	3.2 ms	+40%	0.0%	—	2,500
Nominal	Update WIP Aggregation StoredProcedure	—	1.3%	26.7 ms	-91%	26.0 ms	-92%	0.0%	-100%	1,958
No baseline	Load Machine Data *Not enough history: only 0 baseline executions, 200 needed*	—	0.3%	266.0 ms	—	127.2 ms	—	100.0%	—	139
No baseline	OP300 - Check-In StoredProcedure *Not enough history: only 1 recent executions, 30*	—	—	—	—	—	—	100.0%	—	1



	needed									
No baseline	OP300 - CheckOut StoredProcedure *Not enough history: only 1 recent executions, 30 needed*	—	—	—	—	—	—	100.0%	—	1
No baseline	OP400 - Check-In StoredProcedure *Not enough history: only 1 recent executions, 30 needed*	—	—	—	—	—	—	100.0%	—	1
No baseline	OP400 - CheckOut StoredProcedure *Not enough history: only 1 recent executions, 30 needed*	—	—	—	—	—	—	100.0%	—	1
No baseline	OP500 - Check-In StoredProcedure *Not enough history: only 1 recent executions, 30 needed*	—	—	—	—	—	—	100.0%	—	1
No baseline	Prune SmartTrack Data *Not enough history: only 28 recent executions, 30 needed*	—	—	—	—	—	—	100.0%	—	28
No baseline	Sync Reporting Db *Not enough history: only 0 baseline executions, 200 needed*	—	26.2%	26.2 s	—	226.7 ms	—	100.0%	—	76
No baseline	ZZ Temp - ActionListener Move Verify *Not enough history: only 10 recent executions, 30 needed*	—	—	—	—	—	—	0.0%	—	10

Back to Home

About Us

Factory Data Systems is dedicated to helping companies maximize their manufacturing processes through intelligent application of data.

Event Engine

System Status

Useful Links

Factory Data Systems, LLC
Gartner's Insights
Manufacturing PMI
Manufacturing News

More Information

Phone: +1 585.237.8178
Email: info@factorydatasystems.com
Support: support@factorydatasystems.com
Website: factorydatasystems.com

Copyright © 2026 Factory Data Systems, LLC

15.2 How the Observer Works

A background reader watches the telemetry history files the system already writes and builds a picture of how each event normally behaves.

It adds nothing to event execution. It holds no reference to the running engine and sits on no part of the event path — it reads files after the fact, so it cannot slow an event down no matter what it finds or how much history it is working through.

Because it reads files rather than memory, it also survives restarts, and it keeps far more history than the pages that read memory do.

It remembers its findings across a restart too — which are open, and which you have already been told about — so restarting the service does not announce the same findings again, and anything waiting for the daily summary is still sent.

Reads new telemetry every	60 seconds
Re-runs the rules every	5 minutes
Keeps a resolved finding visible for	7 days

> **On first run it reads back through the history you already have.** A new installation with no telemetry files behind it will show every event as *No baseline* until enough executions accumulate. An existing installation usually has a usable picture within a few minutes of starting.

Reaching one event's health page

Every page that shows a single event — the event list, the event's detail page, and its Telemetry and History pages — carries a **Health** link straight to that event's own health page.

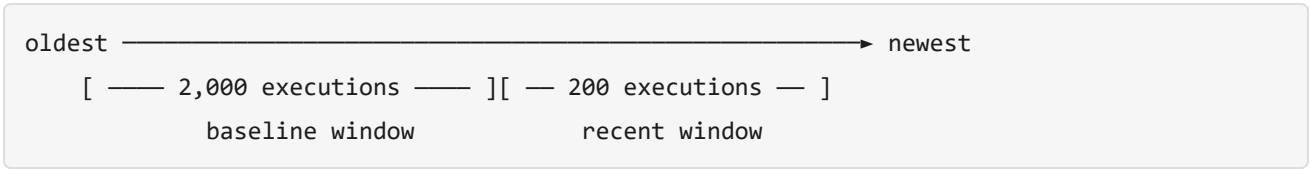
On an event the observer has no history for, **Health** appears greyed rather than as a link, with a tooltip explaining why. That is almost always an event that has never run: the observer builds its picture from completed executions, so an event that has produced none has nothing to show yet. Run it once and the link appears at the observer's next pass. (While Event Health is switched off, the word is not shown at all.)

15.3 How the Numbers Are Worked Out

Every figure on these pages comes from the same small set of ideas. They are worth ten minutes, because they are what makes the difference between a number that means something and a number that merely looks alarming.

The two windows

Every comparison on these pages is **recent behaviour against earlier behaviour**, using two windows of executions:



Window	Size	What it is
Recent	last **200** executions	what is being judged
Baseline	the **2,000** executions immediately before those	what it is judged against

Two things about this are deliberate and matter:

The windows are counted in executions, never in hours or days. Production schedules change, shifts get added and dropped, and product mix varies. A window measured in time would report a quiet shift as an improvement and a busy one as a regression. Counted in executions, a schedule change alters how *long* a window takes to fill, not what it *means*.

The baseline stops where the recent window starts. It does not span all the history — it explicitly excludes the runs being judged. If a slow drift were allowed to contribute to its own reference, it would drag the reference along behind it and the comparison would never fire. That is exactly the failure mode this feature exists to catch, so the separation is built into the one place the windows are produced.

Why the median, not the average

Every central figure on these pages is a **median** — the middle value, the one half the runs come in under. Not the mean.

Event durations have long tails. One run that waited 26 seconds on a lock moves an average substantially and a median not at all. An average would have you chasing a single bad night; the median tells you where the event actually lives.

What "baseline" means on screen

Wherever the word **Baseline** appears next to a number, it means **that statistic, measured over the baseline window**:

- **Baseline median** — the median of the 2,000 earlier executions.
- **Baseline p95** — the p95 of those same 2,000 executions.

The **Change** figure is always the same statistic on both sides, expressed as a proportion:

```
Change  =  (recent − baseline) ÷ baseline        both from the same statistic
```

So a p95 is compared against a baseline p95, and a median against a baseline median — never one against the other. Comparing a recent p95 against a baseline median produces a number that is a genuine shift and the event's own tail length added together, and nothing on screen would let you tell how much of it was which.

What p95 tells you — and what it doesn't

p95 is the value 95% of runs come in under. It describes the slow end of normal: not the typical run, not the worst run ever, but roughly "the worst it gets on an ordinary day".

It is useful for capacity questions — a p95 approaching the command timeout is a real risk even when the median is comfortable. But read it carefully:

> **A high p95 is not by itself a problem.** Some events are naturally spiky and always have been. What matters is whether the p95 has *moved*, which is why the detail page shows the baseline p95 beside it. A p95 eight times the median on an event whose Change is near zero is telling you the event has a long tail, not that anything has gone wrong.

p95 is a display figure, not a trigger. No rule fires on it. The rules work on the median and on the measure of spread described next.

How the thresholds scale

The obvious way to decide "has this got slower" is a fixed ratio — say, 50% above baseline. It does not work here. Across a real production installation, the ratio between an event's p95 and its median ranged from about 1.1× to 20×. One event's ordinary variation is another event's emergency.

So instead of a fixed ratio, the rules scale to **each event's own noisiness**, measured as **MAD** — the median absolute deviation, which is the median of how far each run sits from the median.

MAD is used rather than a standard deviation for the same reason the median is used rather than an average: outliers do not inflate it. A handful of terrible runs makes a standard deviation large enough to hide a real regression behind it; MAD stays describing the ordinary runs.

In practice, on a healthy production event MAD sits around a tenth of the median. A rule set at four MAD is therefore roughly "40% slower" on that event — while a naturally noisier event, where MAD is a fifth of the median, gets proportionally more room before anyone is told. One fixed ratio cannot do both, which is the whole reason for the arrangement.

> MAD is not shown anywhere on the pages. It is the ruler, not a reading.

Why long, slow drift needs a different measurement

The 2,500 executions the two windows are drawn from is a lot of runs but not a lot of *time* — on an event firing 8,000 times a day it is about seven hours. A regression that adds a millisecond a day is completely invisible in seven hours.

So the drift rule reads a second, much longer record: one summary point per **500 executions**, up to **720 points** — around 360,000 executions of reach. It trades resolution for span, which is the right trade for detecting a slope and the wrong one for a percentile.

This is why an event can show a perfectly steady p95 against its baseline and still open a **Latency drift** finding. The two are looking at different lengths of history, on purpose.

15.4 The Five States

State	Meaning
Nominal	Evaluated against a usable baseline, and no rule fired.
Watch	Something is close to a threshold, or has just started. Shown on screen; no notification.
Degrading	A confirmed finding is open. Notifies.
Critical	A confirmed finding at critical severity — close to causing a stoppage. Notifies immediately.
No baseline	**Not a pass.** Too few executions, too little history, the event was recently changed, or it has stopped running. The system is telling you it cannot judge this event right now.

Nominal is the absence of findings, not a measurement. It means every rule ran against enough history to be fair, and none of them fired. There is no number behind it.

An event is judged to have a usable baseline only when both of these hold:

Baseline window holds at least	**200** executions
Recent window holds at least	**30** executions

> **"No baseline" is deliberately not green.** An event the system knows nothing about must never look like one it has checked and found healthy. If an event sits here for a long time, it either runs very rarely or it has stopped running — both worth knowing.

Events that have stopped running

An event with **no executions for 7 days** is not judged at all — by any rule, including the ones that need no baseline. Every rule works from an event's executions, so an event that is not running has nothing new

to report, and judging it anyway could only repeat old news on a timer.

- **Its findings stay exactly as they were last seen.** They are still listed on its health page with their *Last seen* time, and the event keeps the state they gave it — but none is mentioned again, and none is reported as resolved. An event that stopped running has not recovered.
- **With nothing open, it shows as No baseline**, with the date it last ran.
- **It is judged again as soon as it runs.** A finding that is still true carries on where it left off; anything new opens in the usual way.

Seven days clears a weekend and most planned shutdowns, so an idle plant's health page does not turn grey over a long weekend.

15.5 The Event Health List

Every column sorts, and the numeric columns carry a **change** figure beside the value. That change is usually the more useful of the two: the slowest event on your plant is often simply the one that does the most work, while the event whose time has *doubled* is the one worth looking at.

Column	Meaning
State	The event's overall state, taken from its worst open finding.
Event	Name, type, and — where relevant — why there is no baseline yet.
Findings	How many separate things are currently open for this event.
Headroom	How much of its command timeout the event is using. See the warning below.
p95 duration	The total duration 95% of recent runs came in under, and how that p95 has changed against the baseline p95.
p95 DB work	The same, for time spent executing in the database.
Fail rate	Proportion of recent runs that failed, and how that has changed.
Samples	How many executions the figures are based on.

The chips above the table filter to **everything**, **anything needing attention**, or **events with no baseline**.

> **Headroom is necessary but not sufficient.** It measures time spent in the database call that the command timeout governs. An event can show plenty of headroom and still be failing every run, because its time is going somewhere else entirely. Never judge an event on headroom alone — that is what the State column is for.

The observer strip

Along the top of the page is a line describing the reader itself: when it last read, whether it is still catching up, how many events and executions it has seen, and whether any lines were unreadable.

This is there for one reason. A monitor that quietly stopped reading shows no findings — and so does a plant where nothing is wrong. The strip is how you tell those two apart, which is why it is shown even when everything is fine.

15.6 The Event Health Detail Page

Clicking an event's name opens everything known about it.

Health: OP200 - Add Product

OP200 - Add Product

Nominal

200 recent of 2,500 retained, baseline 2,000
8/31/2026 10:54 AM to 9/2/2026 3:53 PM

Failure rate: **0.0%** (Baseline 0.1%)
Type: StoredProcedure

Timeout used

0.5%

Metric trends

	Median	Baseline median	vs. baseline	p95	Baseline p95	Max
Total duration	23.1 ms	23.7 ms	-3%	38.2 ms	59.7 ms	295.2 ms
Tag collection	12.6 ms	12.8 ms	-2%	19.9 ms	20.2 ms	207.2 ms
Work	4.0 ms	4.0 ms	-1%	10.5 ms	9.7 ms	60.3 ms
Database connect	0.0 ms	0.0 ms	+0%	0.1 ms	0.1 ms	0.9 ms
Database execute	4.0 ms	4.0 ms	-1%	10.5 ms	9.6 ms	60.1 ms
Write-back	5.2 ms	6.4 ms	-19%	9.0 ms	17.0 ms	107.6 ms
Dispatch latency	0.1 ms	0.2 ms	-18%	0.3 ms	0.4 ms	42.2 ms

Open findings

Nothing open.

Errors

Device communication	1	100.0%

Evidence

View this event's execution history | Telemetry

Back to event health | Back to Home

Copyright © 2026 Factory Data Systems, LLC

- **Header** — the state, and how much history it is based on. Every number below is only as good as those sample counts, so they are stated first.
- **Timeout headroom** — a gauge with the Watch, Degrading and Critical marks drawn on it.
- **Metric trends** — one row per measurement, described below.
- **Open findings** — what is currently wrong, when it was first seen, and how many executions support it.
- **Errors** — a breakdown of failures by kind, so you can see at a glance whether an event's failures are database timeouts, connection problems, or the application refusing the work by design.
- **Evidence** — links through to that event's History and Telemetry pages. The health page explains; those pages prove.

Metric trends

One row for each measurement the system records — total duration, tag collection, work, database connect, database execute, write-back, and dispatch — so you can see *which part* of an event moved, not just that it moved.

For a stored procedure or database table event, the work *is* the database call, so it is shown as its two parts — **Database connect** and **Database execute** — and there is no separate **Work** row. Message and label events keep their **Work** row. The latency rules (15.7) judge the same rows, so a slow database call is reported once, under the part that slowed, rather than again as work.

Column	What it is
Median	Median of the recent window.
Baseline median	Median of the baseline window.
Change	How far the median has moved, as a proportion of the baseline median.
p95	p95 of the recent window.
Baseline p95	p95 of the baseline window.
Max	The single slowest run in the recent window.

Each recent statistic sits next to its own baseline, so the two comparisons stay separate:

- **Median vs. baseline median** — has the typical run changed?
- **p95 vs. baseline p95** — has the *slow end* changed?

Those can move independently, and the difference tells you something. A median that has moved with a flat p95 usually means everything got a little slower. A flat median with a p95 that has doubled means the typical run is fine but the bad ones got much worse — which points at contention, a cache that stopped helping, or an occasional lock, rather than at the query itself.

The **Max** column has no baseline counterpart on purpose: a single worst run is dominated by whatever one-off happened that day, so a "baseline max" would be noise presented as a measurement.

The small chart at the end of each row is drawn from the long history described in 15.3, so it spans weeks rather than the few hours the two windows cover.

That long history keeps total duration, work and the two database times only. The **Tag collection**, **Write-back** and **Dispatch latency** rows therefore show *No long-term history* where the chart would be, and the **Latency drift** rule does not watch them. Their other columns compare the recent window with the baseline exactly as every other row does.

15.7 What the Findings Mean

Finding	What it is telling you
Timeout headroom	The event is using an increasing share of its command timeout. At 100% the database call is cancelled and the event fails.
Latency step	A measurement jumped. Usually a deployment, a configuration change, or a query that lost an index.
Latency drift	A measurement has been climbing steadily over a long period. This is the one that catches slow growth no single day's comparison would show.
Failure rate	The event is failing more often than it used to.
Chronic failure	The event fails most of the time and always has. No "it got worse" rule can see this, which is why it has its own.
Write-back divergence	The work succeeded but the completion could not be written back to the controller — so the PLC is waiting for a code that never arrives.
Contention burst	The same kind of infrastructure failure — deadlock, timeout, connection loss — happened repeatedly in a short run of executions.

The numbers behind each finding

These are the shipped defaults, fitted by replaying real telemetry rather than chosen in the abstract.

Finding	Opens when
Timeout headroom	Database execute time reaches **50%** of the command timeout (Watch), **70%** (Degrading), **85%** (Critical).
Latency step	The recent median rises at least **100 ms** *and* that rise is at least **4 × the baseline MAD**.
Latency drift	A fitted slope over the long history explains at least **half** the variation ($R^2 \geq 0.5$) *and* the total rise is at least **50%** of where it started, over at least **30** history points.
Failure rate	Recent failure rate is at least **3 ×** the baseline rate, *and* at least **5%**, *and* at least **3** runs actually failed.

Finding	Opens when
Chronic failure	Failure rate reaches **10%** (Watch), **25%** (Degrading), **50%** (Critical), over at least **50** executions — regardless of trend.
Write-back divergence	More than **1%** of recent runs succeeded without their completion reaching the controller.
Contention burst	A single infrastructure error category recurs within the last **50** executions — 3 deadlocks, 2 command timeouts, 5 connection failures, and so on by category.

Two of these deserve a note:

The latency step rule needs both conditions. The absolute 100 ms floor exists because heartbeat-style events run at a median near 25 ms, where a rule scaled purely to dispersion would report a few milliseconds of ordinary variation as a regression.

Chronic failure is the only rule that ignores history. Every other rule asks "is this worse than before?", and all of them are blind to something that was broken from the start. A real production installation contained two events that had failed on more than 98% of runs for a month; no comparative rule would ever have mentioned either.

> **Contention burst thresholds are a judgement, not a measurement.** Neither corpus used for fitting contained enough lock contention to tune against — one produced no deadlocks at all across 3.35 million executions. A site that genuinely sees contention should expect to revisit these.

15.8 Why Findings Are Slow to Appear and Slow to Clear

A finding must hold for **two consecutive checks** before it opens, and clear for **two** before it resolves. With the rules running every five minutes, that is a ten-minute minimum on each edge.

That is deliberate. Anything sitting near a threshold would otherwise flip between "degrading" and "recovered" every few minutes, and a monitor that does that quickly gets ignored — which costs you the warning you actually needed later.

The same caution applies in the other direction: the observer will not resolve a finding for an event it did not manage to evaluate, or for one that has stopped running. Silence is never treated as recovery.

A restart does not reset any of this. Findings are saved with the observer's history, along with whether you have been told about them, so restarting the service neither announces an open finding again nor makes one earn its two checks a second time.

15.9 Health Notifications

Findings always appear on the pages. Whether they also reach a mailbox is controlled by `ApplicationSettings:EventHealth:Notify` (Chapter 3.3):

Mode	Behaviour
Digest *(default)*	One summary message per day of everything that opened or resolved — **except** anything Critical, which is sent straight away.
Immediate	A message on every change.
Off	No messages. The pages still show everything.

A message is sent when a finding **opens**, when it **gets worse**, and when it **resolves**. A finding that stays open is mentioned again at most once a day, and not at all while its event has stopped running. Recoveries are reported as well as failures — knowing something cleared is what makes the next warning worth reading.

Changes waiting for the daily summary are kept across a restart, and so is the time the next summary is due, so a service that restarts every day still sends one. A summary that falls due is sent at the next check, whether or not anything else has changed since.

Reading the daily summary

The summary is laid out to be read at a glance:

- **The counts come first** — how many findings are *Degrading* or at *Watch*, and how many are *Back to normal*. The preview line your inbox shows beside the subject sums them up.
- **Needs attention** lists each event with something open, **worst first**; among events that are equally serious, the one with the most findings comes first. Every finding carries the same coloured severity chip as the detail page, with its figures in bold. A slowdown also shows how many times its usual duration it now takes — ▲ **6.4×** — and a finding that has become more serious since you were last told about it is marked ▲ **Worse**.
- **Back to normal** lists what has recovered, in grey, with each finding as it last stood.
- **Each finding appears once**, as it stood at the last change. A finding that opened and cleared between two summaries is listed only under *Back to normal*, marked *Opened and cleared since the last summary*; it is never shown as a live problem. The subject's *new* and *resolved* counts follow the same rule.
- **Only changes are listed.** A finding that is still open and unchanged is not repeated; its own reminder mentions it again, at most once a day.
- The line above the title names the server that sent the message.

A notice sent on its own — a Critical finding, or any change when **Notify** is *Immediate* — uses the same chips, and says under its title when the finding was first seen and how many executions it rests on.

Such a notice's subject, repeated as its title, names the event and says what happened to the finding. A finding at **Critical** says it is critical instead — *Event health: Line 4 Check-In is critical* — whether it opened at Critical or got worse until it reached it, so the messages that cannot wait stand out in a full inbox. Every

Event Health subject, the daily summary's included, contains the same words (*Event health* in English), so a mail rule can match on those to catch them all.

Notices about a specific event go to that event's own **Notification Addresses** (Chapter 10) when it has any, so they reach whoever already receives that event's failure notices. The daily summary spans events, so it goes to the system message address instead.

All of this needs SMTP configured (Chapter 3.3). Without it, the pages still work and nothing is sent.

15.10 Settings

Event Health is configured under `ApplicationSettings:EventHealth`. The settings, their defaults and what each one does are listed in **Chapter 3.3 — Settings**, along with where the file lives and how to apply a change.

Two things worth knowing that the settings table does not say:

> **Turning the observer off does not turn off telemetry.** The history files are written by the engine regardless of this setting, so switching the observer back on later lets it read back through everything it missed. Nothing is lost by leaving it off for a while.

> **Raising `RingCapacity` widens the comparison windows.** The recent and baseline windows are drawn from the executions it retains, so a larger ring means comparisons reach further back — at the cost of memory, and of discarding the saved history once when the setting changes.

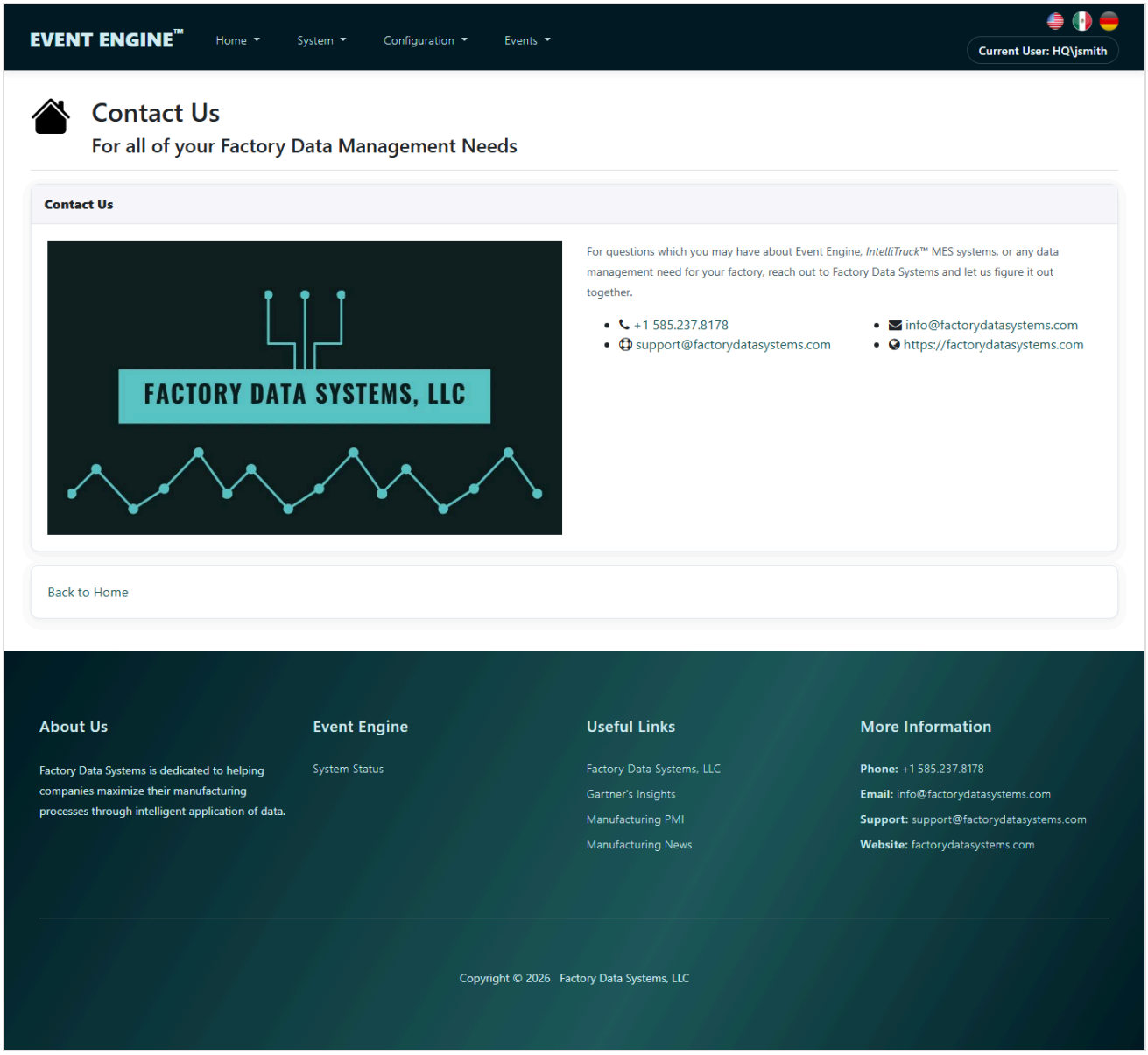
Next: Chapter 16 — Support Options · Guide home

Chapter 16 — Support Options

16.1 Contacting Factory Data Systems

For questions about Event Engine, *IntelliTrack* MES systems, or any data-management need for your factory, reach out to Factory Data Systems — the same information is on the **Home** → **Contact** page inside the product.

Channel	Details
Support email	support@factorydatasystems.com
General inquiries	info@factorydatasystems.com
Web	https://factorydatasystems.com



16.2 Licensing Support

- Purchases, renewals, and your license account: Factory Data Systems customer portal — https://secure.softwarekey.com/solo/customers/Default.aspx?AuthorID=5547024
- Offline activation / renewal / deactivation file processing — https://secure.softwarekey.com/solo/customers/ManualRequest.aspx
- See Chapter 4 for every licensing workflow and the licensing troubleshooting table.

16.3 Before You Call — Information to Gather

Collecting these up front dramatically shortens support cases:

1. **Application version** — shown on the **Home** → **About** page (*Application Version: vX.Y*).
2. **Log files** — download the relevant day's log from **System** → **Logs** → **Export File** (Chapter 3.2). If the issue is reproducible, consider temporarily raising the logging level to *Debug*, reproducing, and capturing that log.
3. **The error details** — from the dashboard error list or the failing event's telemetry page (timestamp, category, full message).
4. **A configuration backup** — **System** → **Backup/Restore Configuration** → **Download Configuration Backup** gives support an exact copy of your processors, tags, and events with no plant data included.
5. **License IDs** — from **System** → **Product Licensing**, if the issue is licensing related.

16.4 Self-Service Checklist

Many issues resolve quickly with these checks:

Symptom	First checks
System won't start	Licensing state (Chapter 4); the Start button tooltip explains the block.
Events not firing	Is the system scanning (green light)? Is the event Enabled? Is the trigger tag's template configured (rising edge, thresholds)?
Event failing	Open the event's **Telemetry** page — the error list usually names the exact database/printer/SMTP problem. Use **Manual Trigger** to reproduce on demand.
Slow events	Use the telemetry three-phase breakdown (tag collection vs. work vs. write-back) to identify the slow leg — see Chapter 14.
No emails	SMTP relay settings in **System** → **Settings** (Chapter 3.3); the messaging troubleshooting table in Chapter 13.
Labels not printing	Printer name/availability on the host; the ZPL tips in Chapter 12.

16.5 End User License Agreement

Use of Event Engine is governed by the Factory Data Systems End User License Agreement, which is presented during installation. A copy is available here at any time:

- Event Engine End User License Agreement (PDF)

16.6 About Factory Data Systems

With over 20 years of experience in manufacturing facilities — automotive, medical devices, food & beverage, and process industries — Factory Data Systems specializes in MES system integration, data

collection, reporting, and SCADA, including the *IntelliTrack*™ MES system and Event Engine™. Our vision is to improve the quality and efficiency of our customers' manufacturing processes in meaningful ways through the intelligent use of data collection and analysis.

Next: Appendix A — Glossary · Appendix B — Release Notes · Guide home

Appendix A — Glossary

A single lookup for the vocabulary used throughout this guide. Each term links to the chapter where it is explained or configured in full. Terms appear in alphabetical order.

Term	Meaning	Where it's configured
Cascaded Event	An event fired automatically upon the *successful* completion of another event — used to chain work, e.g. print a label only after a database log succeeds.	Ch. 10.2
Collection	A structure *template* you build by hand in the web UI (protocol-specific), as an alternative to importing a UDT. Member functions (trigger/completion/message) can be pre-assigned so tags come out event-ready.	Ch. 8.2
Completion Tag	An integer tag the event writes a completion code into: **0 = In-Process, 1 = Success, codes < 0 = Error Code**. The PLC watches this tag to learn the event's outcome. For Message events, success (1) is reported when the message is queued for delivery, not when it is delivered (Ch. 13.2).	Ch. 10.2
Database Connection	A configured, named connection to a business database (SQL Server, PostgreSQL, MySQL, or Oracle) that stored-procedure and table events read and write through.	Ch. 5
Event	A configured unit of work — stored procedure, database table access, label print, or message — executed when triggered.	Ch. 10–13
Event License (event counter)	The license that meters how many **enabled** events the system may run; multiple event licenses stack. PLC protocols are *not* separately licensed.	Ch. 4.2
Fan Out	A collection-member option that creates one child tag per array element (`Member[index]`) on every tag built from that collection, so events can bind an individual element. The tag-level equivalent is *Create Children on Save*.	Ch. 8.2
Filter (table event)	A condition narrowing which rows a Database Table **read** returns. Filters combine with AND, and their values are **constants typed into the event**, never tag values — a controller-driven lookup needs a stored procedure instead.	Ch. 11.2
Master Tag	A structure tag chosen as the foundation of an event; its members are automatically matched to the procedure's parameters or the table's columns by name, and the trigger/completion/message dropdowns are filtered to its members.	Ch. 10.2
Namespace	A "folder name" applied to imported UDTs that keeps same-named structures from different PLC programs from colliding (e.g., `Line1` , `PackagingV2`).	Ch. 8.1

Term	Meaning	Where it's configured
OPC UA	The OPC Unified Architecture protocol, spoken to an OPC UA **server** at an `opc.tcp://host:port/path` endpoint. Event Engine is a UA **client** only: it does not act as an OPC server, and it does not speak OPC Classic (OPC **DA**) — a DA server has to be fronted by a DA-to-UA gateway.	Ch. 6.2
Payload Tag	A tag used purely to carry data in an event, with no control responsibilities. Read on demand during the event's *tag collection* phase, so it adds no continuous scan load.	Ch. 9.2
Processor	A configured connection to a PLC or controller (name, protocol, IP address, etc.). Every tag lives in a processor.	Ch. 6
Ref Cursor (REFCURSOR)	A cursor a procedure returns to the caller as a *result set* rather than a scalar value — Oracle `SYS_REFCURSOR`, PostgreSQL `refcursor`. Event Engine hides it from the parameter grid and auto-creates a result set for it.	Ch. 10.2
Repeat Until	On a time-based trigger, the last time of day at which its repeat interval may fire; blank means "until the end of the day". Repeats are anchored to the trigger time, not to when the system started.	Ch. 7.4
Result Set	Tabular output a procedure or table operation returns independent of its scalar parameters (e.g., a recipe table), written back into any target tag whose shape can hold it — a structure, a tag collection, an array, or a single atomic tag. Rows the set does not supply are written as default values, not left as they were, and a procedure must return the **same number of sets on every execution**. Delivered directly from `SELECT`s (SQL Server, MySQL) or through a ref cursor (Oracle `SYS_REFCURSOR`, PostgreSQL `refcursor`).	Ch. 10.2
Status Message Tag	A string tag the event writes status or error text into for the controller's benefit.	Ch. 10.2
Symbol (tag)	An optional controller symbol recorded beside an address-style tag name, so `DB4.DBB0` or `40001` is recognizable in pickers and error messages. Display only — nothing is addressed by it.	Ch. 9.2
Tag	A named piece of data in a processor — an atomic value, an array, or an entire structure. Tags are what events read from and write to, what triggers watch, and what database parameters and label fields bind to.	Ch. 9
Telemetry	The continuous self-measurement Event Engine exposes — run state, call/error counts, and the three-phase per-event breakdown (*Tag Collection* → *Work* → *Write Back*).	Ch. 14
Trigger Tag	A tag assigned a trigger template; when its condition is met, its events fire. Trigger tags are scanned continuously at the template's interval.	Ch. 9.2

Term	Meaning	Where it's configured
Trigger Template	A reusable definition of *when* something should happen — tag-change based or time based. Tags and events select triggers from these.	Ch. 7
UDT (User Defined Type)	A pre-compiled structure definition, most commonly **imported directly from a PLC export file** (Rockwell `.L5X` or Siemens `.scl` / `.stl` / `.db` / `.udt`), mirroring the structure engineered in the PLC program.	Ch. 8.1
Use Single Write	An event option that consolidates payload write-backs with the completion-code write into one operation, removing a round trip to the PLC.	Ch. 10.2

Back to the guide home.

Release Notes

What changed in each release, and — more importantly — **anything you have to do yourself when you upgrade**. Releases are listed newest first.

> **Before any upgrade:** take a configuration backup from **System** → **Backup/Restore Configuration** (Chapter 3.5). It is the supported way back if an upgrade has to be reversed.

To confirm which version a host is running, open **Apps & features** in Windows and read the version beside *Event Engine*, or check the installer file's **Details** tab in its file properties.

Next release — not yet published

Documented here as it lands, so nothing is reconstructed from memory at release time. The version number and date are filled in when the build is cut.

If you grep the log for truncation warnings, your pattern has to change

This is one of three items in this release that can break something you have set up. The others are a mail rule that looks for *is degrading* in Event Health subjects (below) and, where messages are sent in German, one that looks for *Scannen* in a status subject (below). If nothing at your site watches the log file for a particular phrase, there is nothing to do.

Event Engine warns whenever a string value is too long for the tag it is being written to and has to be shortened. That warning is the **only** record that it happened — the write succeeds, the tag ends up holding a cut value, and nothing else reports it. Its wording has changed.

Before	`Maximum allowable string length is 82, actual length is 150. String will be truncated.`
Now	`String truncated to fit tag "Program:Main.Status": 150 characters exceeded the tag's maximum string length of 82, so 71 were dropped and "…" was written (the trailing marker shows the value was cut). Source: event "Recipe Load" (42).`

Anything matching on `Maximum allowable string length` or `String will be truncated` — a log alert, a SIEM rule, a scheduled search, a spreadsheet someone pastes the log into — **will stop matching.** Match on `String truncated to fit` or `String hard-clipped to fit` instead.

The old message told you that something had been lost. It never told you *what* was lost, from which tag, or on whose behalf, so answering "which value got cut, and what did the controller actually end up holding?" meant reproducing it. The new one carries the tag, the two lengths, how many characters were dropped, the exact text that reached the controller, and the event behind the write.

A value that fits is still not warned about, so the warning stays worth reading.

A cut value now always shows that it was cut

A string too long for its tag has always been shortened with a trailing `...` so anyone reading the value back can see it was cut. On tags with room for fewer than five characters that marker did not fit, and the value was quietly clipped instead — which meant the tags most likely to lose content were the only ones that gave no sign of it.

Those now end in a single `~`. A four-character tag sent `ABCDEFGHIJ` reads back `ABC~` rather than `ABCD`. If you have logic or an HMI comparing against a value from a very short string tag, this is the one place a stored value changes shape.

Siemens tags are unaffected: that protocol deliberately clips without a marker, because the value lands where logic compares it.

A failed read or write now says why, not just which tag

When a batch of tags fails, the Error List entry named the tags and stopped there — the controller's actual reason (a tag that does not exist, one whose External Access forbids writing, a connection that dropped) went to the log file on the server and no further. It now travels with the message, so the page an operator is already looking at carries the answer.

Where one cause hit many tags it is stated once rather than repeated per tag, and long lists are capped with a count of the remainder; the full list is still in the log.

Relatedly, a failed write to a **string** tag used to always add "check that the tag's maximum length matches the controller" whatever had really gone wrong, so a tag the controller did not have sent you looking at string capacities that were never the problem. That advice now appears only when the controller gave no reason of its own.

OPC array elements are always read and written through their array

On an OPC processor, a child tag made by **Create Children on Save** (`Name[index]`) is now always read, written and monitored as that element of the array tag's node. Event Engine no longer tries the child's own name on the server first and switches form when that fails, so a failure reaches you exactly as the server reported it rather than being retried under another address.

Two setups behave differently:

- **A KEPServerEX project that defines an array tag and also a tag per element.** KEPServerEX refuses a write to one element of an array tag, so writing an element child now fails with `BadWriteNotSupported`, where it used to go through the per-element tag. To write single elements, browse to the server's per-element tags and add them as tags of their own, without creating children on the array.

- **A tag named like an element that was not created as one** (typed by hand, or imported without its array) is used as the exact node name it carries. If the server has no node by that name, the tag reports the node as unknown instead of quietly reading the array's element.

An OPC tag whose node goes missing now always comes back on its own

Nothing to do — this removes a restart that used to be necessary.

When an OPC server stops exposing a tag's node, Event Engine retries with a widening delay, and after eight attempts it reports the tag on the Error List and marks it Bad. Reporting it used to mean giving up on it as well: the retries stopped. If the server's address space was then repaired while the connection stayed up — a KEPServerEX channel or device rebuilt, a tag renamed and put back, a server project reloaded, none of which drops the OPC UA session — the tag stayed Bad until someone restarted Event Engine or re-saved the tag.

Event Engine no longer stops retrying any tag, for any reason. The retry delay doubles from one second up to five minutes and then stays there, for as long as the failure lasts, so a tag recovers by itself within five minutes of its node being served again however long the repair took. Reporting it after eight attempts — about four minutes in — is now only that: a report. The Error List entry is still raised once, not once per retry.

This matters most for scanned tags. One that is read on demand already recovered on its next read, and any tag recovered after a reconnect; the case that had no way back was a scanned tag on a healthy connection, because a tag whose node is gone has no live subscription left to notice the node returning.

A node that is still missing when a retry runs does not restart its own retry count, so a server that has genuinely lost a channel is asked once every five minutes per tag and no more. And while the server itself is unreachable, the retries wait for the connection to come back rather than polling for it, and resume the moment it does.

The Error List entry is corrected if the cause changes. It names one of two failures — the server no longer exposes the node, or the node is there and will not serve a value — and those send you to opposite places. Since the retries continue, a tag can move from one to the other while it waits: a missing node recreated but still mid-rebuild, or a rebuilding node finally removed. The entry is rewritten when that happens, and only then; a retry that finds the same cause again adds nothing.

A whole channel going missing at once is still handled in bulk, and now stops being when it subsides. When enough of a processor's tags lose their nodes together, Event Engine probes them in one batch per retry instead of searching the address space for each, and raises a single Error List entry for the event rather than one per tag. That bulk handling now ends once the number of tags still waiting falls back to a handful, so a tag renamed after the bulk of the event has cleared is searched for and matched to its new node again. Previously it ended only when every last tag recovered — which, on a channel nobody repairs, never happened.

You can test the mail relay before you commit to it

System → **Settings** now has two buttons at the bottom of the SMTP Relay Settings card (Chapter 3.3). Both answer for the values **as they stand on screen** — neither needs you to save first, and neither writes anything — so a relay, a port or a password can be tried and the answer read before the configuration is committed.

Button	What it does
Test Connection	Connects, negotiates encryption if *Use Encryption* is checked, signs in if a user name is set, and hangs up. No mail is sent.
Send Test Message	Sends one real message, from *From Address* to *System Message Delivery Address*, through the same mail path every event uses.

Test Connection reports each of the three things it proved, because "it connected" on its own does not say whether the encryption setting or the credentials were part of it: the connection and how long it took, whether the session was encrypted *as configured*, and whether the relay accepted the sign-in. A relay offering no encryption on the port you gave it **fails** the test rather than quietly connecting without it, since every message would then fail the same way. When either button fails, the relay's own words are shown underneath — "*535 Authentication credentials invalid*" is the difference between a password to fix and a network to investigate.

Both follow the database **Test Connection** button rather than inventing a second shape, including its rule that a blank password means the saved one — otherwise the button would refuse every configuration an operator arrives wanting to test.

> **One behaviour change, and only in one case.** The mail settings had been read in five separate places, each free to spell a key or default it differently, and two already had: an installation whose upgrade dropped the `SMTPTimeout` key ran on a 30-second timeout while the Settings page showed 10. There is now one reader, and the default is **10** — what `appsettings.json` ships and what this guide documents. Nothing changes for an installation that has the key set, which is almost all of them.

Mail sent after a quiet spell is no longer lost

Event Engine keeps its connection to the mail relay open between messages, so that a burst of messages goes out over a single sign-in. A relay closes a connection that has been left unused — some in under five minutes — and the next message sent over it was refused and never sent again. The log recorded an error ending *4.4.2 Idle timeout, bye bye*. Where mail is occasional, that was a large share of all mail: event notifications, failure notices and Event Health messages alike, and nearly every Event Health daily summary, which goes out after hours with no other mail. A **Message** event still reported success each time, because it completes once its message is queued, so an event's history does not show which of its messages were lost.

- A connection left unused for more than 30 seconds is now closed, and the next message opens a new one.
- If a relay closes a connection sooner than that, the message is sent again, once, over a new connection, and delivered. The log records this as information — *The SMTP relay had closed an idle connection … sending the message again on a new one* — rather than as an error.

A message the relay refuses over a new connection is not sent again, and neither is one that timed out, since the relay may already have taken it. A relay that drops a connection without saying so, at the very moment it accepts a message, could see that message arrive twice; a lost notification is the worse of the two. No action is required.

A restore that stops partway no longer leaves a mixed configuration

A restore or merge writes the configuration files one at a time. If one of them could not be written — another program had it open, for example — the files written before it kept the restored contents and the rest kept the old ones, and the banner said only *Restore failed*. Event Engine went on using the configuration it had before the restore, and the next change saved to one of the restored files quietly wrote the old contents back over it. What was left was neither the backup nor the configuration it replaced, and an event could refer to a tag or a connection that existed in only one of them.

- A restore that stops partway now puts back every file it had already written, and the banner says so: *the configuration is unchanged*, followed by what stopped it.
- If a file cannot be put back, the banner names it and says the configuration is now a mix of the two. Run the restore again before relying on it.

No action is required.

Event Health no longer repeats its notices every time the service restarts

Restarting Event Engine made Event Health announce every current finding again about five minutes after it came back — the same messages, once per restart, for as long as each condition lasted. The observer carried its history across a restart but not what it had concluded from it, so it met every finding for the first time. The daily summary failed the other way: whatever was waiting for it was discarded on restart, so a service restarted more often than once a day never sent one.

Both are now saved with the observer's history (Chapter 15.8):

- A finding you have been told about is not announced again after a restart, and its once-a-day reminder carries on from when you were last told.
- Changes waiting for the daily summary, and the time it is due, survive a restart. The summary also goes out as soon as it is due, rather than waiting for some other finding to change.

Events that have stopped running are no longer judged. An event with no executions for 7 days keeps its findings on its health page, but they are neither repeated nor reported as resolved. Previously such an

event was reminded about daily for as long as it stayed stopped, and could be reported *back to normal* 45 days after it last ran, simply because the comparison could no longer be made. With nothing open it shows *No baseline* with the date it last ran (Chapter 15.4). The reason an event has no baseline now also appears in the page's own language rather than in English.

No action is required. The first start after upgrading announces current findings one last time — except for events that have stopped running — and restarts are quiet from then on.

Event Health messages can now be read at a glance

Event Health mail arrived as one unbroken paragraph: the daily summary ran every event and finding together, and a notice about a single finding ran its details into one line. Mail from Event Engine is always sent as HTML, but these messages were written as plain text, so every line break was lost.

Both kinds of message are now laid out properly (Chapter 15.9):

- The summary opens with the counts, then lists the events that need attention, worst first, with a coloured severity chip on every finding and its figures in bold. Recovered findings follow under *Back to normal*.
- **A finding that opened and cleared on the same day is shown once, as recovered.** It used to appear twice — once still failing, once resolved — so a summary could lead with a problem that had already gone away. The *new* and *resolved* counts in the subject follow the same rule.
- A slowdown shows how many times its usual duration it now takes, and a finding that has become more serious since it was reported is marked **Worse**.
- **A notice about a Critical finding now says it is critical** — *Event health: Line 4 Check-In is critical* — whether the finding opened at Critical or got worse until it reached it. It used to read *is degrading* like any other new finding, so the messages that cannot wait looked no different in an inbox from the rest.
- Each message names the server that sent it, fits a phone screen, and follows the mail client's dark mode where the client supports one.

No action is required, unless a mail rule looks for the words *is degrading* in a subject: a Critical finding's notice no longer contains them. Add *is critical* to the rule, or match on *Event health*, which every Event Health subject contains. If your notices are sent in Spanish or German, use the matching words from a notice you have received.

Event Health settings apply without a restart

The five settings on the **Event Health** card of the System Configuration page (Chapter 3.3) used to be read only when the service started, so every change asked for a restart. They now apply as soon as you save, and the card no longer flags a restart for them:

- **Monitor Event Health** starts or stops the observer on the spot. Switching it back on resumes where it left off rather than re-reading the history.

- **Health Notifications**, **Digest Interval**, **Observer Poll Interval** and **Observer State Save Interval** take effect from the observer's next pass. Shortening the digest interval brings the next digest forward, instead of waiting out the rest of the longer one.
- A change made by editing `appsettings.json` directly applies the same way.

Switching the observer off now also clears the Event Health page and the health badges until it is switched back on, rather than leaving the last readings on screen with nothing keeping them current.

Fixed along the way: restarting the service with the observer switched off wrote an empty health history over the saved one. Switching it back on then rebuilt the history only as far back as the telemetry files still reach, and announced current findings again. The saved history is now left alone while the observer is off.

`RingCapacity`, which is not on the page, still takes effect only after a restart. So does a new **Telemetry History Location** as far as Event Health is concerned: new history files go to the new folder straight away, but the observer keeps reading the previous one until the restart. Saving a new location now flags that a restart is required; before, nothing did, and the Event Health page went quietly out of date. No action is required.

Engine and license messages are laid out the same way

The other messages Event Engine sends on its own — scanning started or stopped, a start that failed, a device that would not come up, scanning stopped for licensing, and the license warning — now share the layout of the Event Health messages. Each names the server that sent it and when, and sets any failure apart in red. Along the way:

- **A failed start says it failed.** Its message was headed *Event Engine system is now scanning*; it now reads *Event Engine could not start scanning*, followed by the reason.
- **A device message names the device.** It carries the sentence the Error List shows — *Device "Line 4 PLC" failed to initialize: …* — where it used to give the fault alone.
- **Text inside a message can no longer break its layout.** An error, a device fault or a license name is shown as written, even when it contains characters such as `<` or `&`.
- **The license warning lists everything it was sent for.** It used to list only licenses that had expired or were about to, so a system that must be reactivated, or a build newer than its maintenance coverage, was sent a heading over an empty list. Each line is now marked **Expired**, **Reactivate**, **Version not covered** or **Expiring**, most urgent first; the first and third are the words the license list uses.

Subject lines are unchanged, except in German: the message that scanning has started now names the state as the System page does, *Event Engine-Status: Scan läuft*, where it said *Scannen*. No action is required, unless a mail rule looks for *Scannen* in a subject. Look for *Scan läuft* instead, or match on *Event Engine-Status*, which every status subject contains.

4.0.0038 — 23 August 2026

Everything below is new since 4.0.0025. Interim builds 4.0.0026 – 4.0.0037 are not documented separately — this entry covers the series. **Upgrading to this build needs no manual work.** The one section below headed *If a host on 4.0.0032 – 4.0.0037 will not start* applies only to a host already running one of those interim builds.

If a host on 4.0.0032 – 4.0.0037 will not start

Upgrading to 4.0.0038 is the fix, and a host coming from 4.0.0031 or earlier straight to this build is never affected.

An upgrade has always kept the operator's own `appsettings.json` — that is what preserves a site's port, paths and credentials — but nothing ever merged in a key that a newer version had begun to require. Build 4.0.0032 added one (`ApplicationSettings:DataPersistence.File:Files:Secrets`, part of moving mail relay credentials out of that file), so a host upgrading from 4.0.0031 or earlier to any build between 4.0.0032 and 4.0.0037 came up without it, and the service exited during startup.

It presents as a service that will not start:

Windows says	"A timeout was reached (30000 milliseconds) while waiting for the Event Engine Service service to connect" — but it says so after about **five seconds**, not thirty. A fast answer means the process exited rather than ran long.
The log says	`Value cannot be null. (Parameter 'path2')`, and nothing else.

From this build the installer merges the keys a new version introduces into the file it preserves. The merge is strictly additive — a key already present is never modified, reordered or removed, and comments and formatting are left as they were — and the previous file is kept beside it as `.bak`. A required key that is missing now names itself in the error instead of reporting a null path.

To repair a host without upgrading it, open `appsettings.json` in the installation folder and add `"Secrets": "secrets.txt"` to the `Files` block (the file contains exactly one). Save it as UTF-8 **without** a byte-order mark and do not reformat it — it carries `//` comments that a JSON round-trip would discard. `appsettings.json.default` sits beside it for comparison.

Event Health — a warning before an event stops the line

Events → Event Health

An event whose stored procedure has slowed from 40 ms to 900 ms over three weeks is still succeeding, so nothing on any other page says a word about it — right up to the day it crosses its command timeout and stops the line. A background reader now watches the telemetry history files the system already writes, learns how each event normally behaves, and reports the ones that have changed (Chapter 15).

- **A health list of every event** — its state, how many findings are open, how much of its command timeout it is using, its p95 total and database durations, and its failure rate. Each figure is shown beside how it has changed against that event's own earlier behaviour, because the event whose time has *doubled* is usually worth more attention than the one that is merely slowest.
- **A detail page for each event** carrying the trend behind every measurement and the reasoning behind every open finding, so a number can be judged rather than taken on trust.
- **Seven rules**: timeout headroom, latency step, latency drift, failure rate, chronic failure, write-back divergence and contention burst. **Chronic failure** is the one that finds an event that has been failing most of its runs since the day it was configured — it never *got* worse, so no comparison would ever have mentioned it.
- **Findings are deliberately slow to appear and slow to clear.** One must hold for two consecutive checks before it opens, and clear for two before it resolves, so an event sitting near a threshold cannot flip between states every few minutes.
- **Nothing is added to event execution.** The reader sits on no part of the event path and reads the telemetry files after the fact, so it cannot slow an event down whatever it finds. It survives restarts, and on first run it reads back through the history you already have.
- **Findings can also reach a mailbox** — one summary a day with anything Critical sent straight away, a message per change, or nothing at all. The settings live in `appsettings.json` under `ApplicationSettings:EventHealth` (Chapter 3.3); the defaults suit almost every installation, and none of them affect event execution.

Event Health needs no setup. It is on after the upgrade, needs nothing configured, and reads telemetry the system is already writing. A new installation with no telemetry behind it shows every event as *No baseline* until enough executions accumulate.

Communications

- **GE SRTP now routes to the controller you configured.** Port, slot and rack never reached the driver, so every processor silently used its defaults — port 18245, slot 1, rack 0 — and a controller anywhere else was unreachable with nothing to say why. A PACSystems processor also ran as a Series 90 one whatever was selected. **Existing GE sites should check Slot after upgrading**, because the field had no effect before and now has one (Chapter 6).
- **OPC nodes with more than one dimension are declined rather than flattened.** A 2×3 node used to read as six values with the row boundaries gone — indistinguishable from a genuine six-element array — and an element tag selected a whole row rather than a cell. Such a tag now reports Bad quality with one Error List incident naming it, and the browse list says why the node will be declined (Chapter 9).
- **A replaced OPC session no longer leaves duplicate subscriptions behind**, so a tag stops being notified twice after a server restart.
- **OPC node recovery is bounded.** A node that returns unusable is now distinguished from one that is simply gone, retries are budgeted, and a tag that exhausts its budget is parked with the reason

stated instead of retrying indefinitely.
- **A device that fails to initialize says what actually failed** — the underlying error rather than the wrapper — once per distinct fault however many tags or retries follow, and reports again if it fails after having recovered.
- **Siemens symbolic addressing no longer restricts UDTs.** The whole mode is instead gated on the driver licence it needs, and the *Use symbolic (optimized) addressing* checkbox is visibly disabled with the reason given. Existing tags keep the addressing they were configured with; new tags default to reference addressing (Chapter 6).

Events and triggers

- **A value is frozen at the moment it changes.** Drivers reuse one tag object and overwrite its value in place, so a handler could read whatever had arrived by the time it ran rather than the value it was notified about — a false → true → false pulse was evaluated as three falses, neither edge fired, and in the field that is indistinguishable from the controller never having triggered.
- **A trigger time is now defined on the two days a year it is not a time** (Chapter 7). Where the clock goes back and a time happens twice, the first occurrence wins — a daily trigger used to fire an hour late and sit 25 hours from its previous run, quietly widening any query that covers "since the last run". Where the clock goes forward and the time does not exist, the work slides forward intact, so 02:30 fires at 03:30 rather than not running that day. A repeat interval still steps in absolute time.
- **Configuration changes are announced in order and awaited**, so a save cannot report success before the running engine has taken it up — and **a change that saved but was not adopted now says so in a banner**, stating that the change was stored, so it is not saved a second time.
- **A message event with no relay configured is recorded as failed**, with the reason, instead of being dropped with a log warning behind a passing handshake (Chapter 13). *Notify on Failure* also read the relay setting under the wrong name on three event screens, which left it always enabled on two of them and always disabled on the third.
- **Memory no longer grows with every tag the process has ever seen.** The value dispatcher and the event bus each kept a slot for every tag id since startup, so a site that edits tags grew until the service was restarted.

Database connections

- **Test Connection now tests the connection the engine would actually open** (Chapter 5). Each screen built its own connection string, so PostgreSQL and Oracle ignored the connect timeout you set, and SQL Server tested without encryption whatever the connection required — the button could pass a configuration that fails at runtime, or the reverse. The connector now builds the string for the probe exactly as it does for the engine, and the driver's failure detail comes back with the result instead of being swallowed.
- **The open duration is reported for every dialect**, not only SQL Server.

- **Test Connection uses the saved password when the box is left blank.** Save already did, so the two buttons on one form disagreed about the same connection.
- **Connect timeout is a property of each connection** rather than one process-wide setting — 15 seconds by default, clamped to 1–300. A connection saved before the setting existed does not become one that never times out.
- **Multiple Active Result Sets is restored for SQL Server**, and a connection string is assembled once per set of credentials rather than on every open; the password decrypt behind that assembly was costing more than the connection it was for.

Security and configuration

- **Mail relay credentials live in the configuration store**, protected the way database passwords already were, and are carried by configuration backup and restore. They used to sit in `appsettings.json`, which a build or a reinstall could overwrite; an existing setting is migrated at first start, so mail keeps sending across the upgrade (Chapter 3.3).
- **Three defects in the new HTTPS surface are closed** — among them a certificate-validation step that would accept a PFX password over plain HTTP, where only the page's disabled fieldset had been stopping it (Chapter 3.6).
- **A plugin that is not the one the installer placed is reported.** Every packaged assembly is recorded with a SHA-256 and checked as it loads; a mismatch — an interrupted upgrade, a file an anti-virus quarantined and restored, a hand edit nobody remembers making — reaches the Error List. It reports rather than refuses, because refusing a protocol plugin costs a plant its controller connection; `ApplicationSettings:Plugins:EnforceIntegrity` turns refusal on for a site that wants it.
- **A running service can be asked whether it is healthy.** `/health/live` and `/health/ready` answer requests made from the machine itself, for a service-manager restart policy or a monitoring agent. Readiness covers configuration load, licence state and the plugin catalog: a configuration file that failed to load, or a catalog with no protocol plugins at all, reports unhealthy — states that used to be visible only as a log line at the moment they happened.
- **One operator's language no longer changes another's page.** Culture was applied process-wide, so the last request to run decided the language and number formatting for every request already in flight, and for triggers, subscribers and the telemetry writer with them.
- **A user's settings are one row per identity.** The lookup is indexed instead of scanned — it runs on every page render and grew with the number of operators who had ever signed in — a Windows identity is matched case-insensitively so `CORP\jim` and `CORP\Jim` are one person, and a page view no longer writes settings.

Installer and packaging

- **New settings keys are merged into the `appsettings.json` an upgrade preserves**, with the previous file kept as `.bak` (see the section at the top of this entry).

- **Every packaged assembly's version and build date is verified before signing**, so a stale build output cannot ship an assembly stamped with an older version than the release containing it.
- **The messaging subscriber no longer ships the whole application.** Its plugin folder went from 84 files to 54 — it had been carrying `EventEngineWeb.exe`, the PLC driver, the Logix plugin, the UDT compiler and the SQL Server connector.
- **The end-user licence agreement text is updated.**

Interface and system

- **A mixed-protocol UDT namespace can be updated from the UI again.** Importing a Logix `.L5X` into a namespace that also held a Siemens-generated type compiled that namespace without the Siemens references; the compile failed, the rollback discarded the whole upload, and there was no way through it (Chapter 8).
- **The UDT import page no longer calls itself Logix-only** — it has accepted Siemens S7 sources since they were added, while its heading and file tooltip still said otherwise.
- **Editing a collection rebuilds that collection's instances where they stand**, instead of climbing to the outermost structure and regenerating unrelated ones — which surfaced any pre-existing drift as collateral deletions and could refuse the edit while naming tags the operator had never touched.
- **The messages the web framework writes on your behalf are translated**, so a mistyped number or a missing required field reads in the operator's language.
- **The log and the Error List no longer share one sentence.** The log stays English for whoever is diagnosing an installation; what an operator reads on screen is translated.
- **A failed start says how far it got.** The startup breadcrumbs were written at a level a default installation never records, so a service that died during startup left nothing behind to say where.
- **Merge adoption reports what it announced**, rather than the totals it set out to announce.
- **Two rendering fixes:** the HTTPS page draws its redirect checkbox beside its label instead of on top of it, and a collection's properties put Namespace above Name.

Known limitations

- **A new plugin entry is still not merged into an existing `appsettings.json`.** The upgrade merge treats a list as a single value, because merging elements positionally into a list an operator may have edited is guesswork — so a release that adds a plugin still needs that entry added by hand.
- **Siemens symbolic (optimized) addressing remains unavailable**, now pending the driver licence it requires rather than the UDT restriction that used to describe it. Reference addressing is unaffected, and existing configurations import and load unchanged.

Upgrade checklist

- ☐ Configuration backup taken (**System** → **Backup/Restore Configuration**)

- ☐ Upgrade installed, service running, web UI reachable
- ☐ GE SRTP sites only: **Slot** — and **Rack** on PACSystems — checked on each processor
- ☐ Error List reviewed: plugin integrity, device initialization and declined OPC tags all report there
- ☐ **Events** → **Event Health** opened once to confirm the reader is running — the strip along the top of the page says when it last read

---

4.0.0025 — 17 August 2026

Everything below is new since the 4.0 build dated 22 July 2026. Interim builds 4.0.0013 – 4.0.0024 are not documented separately — this entry covers the series, and the **Required action** below applies to anyone upgrading from a build older than 4.0.0019.

Required action — trust the new OPC UA client certificate

If any of your OPC processors connect over a secured endpoint, each OPC server must be told to trust Event Engine's new client certificate — once, per server — before those processors will connect again. Nothing else in this release requires manual work.

Why this is necessary

This release replaces the OPC UA client inside Event Engine. An OPC UA client is not just a socket: it is an **application** with a certificate of its own, and it presents that certificate whenever it opens a secured channel. The new client has a **new certificate**, so a server that was told to trust the old one has never seen this one and will refuse the connection.

This is how OPC UA is designed to behave, and no change on the Event Engine side can avoid it. It is the same step an administrator performs when any new OPC client is introduced to a server.

Who is affected

Affected	Any **OPC** processor that negotiates a **secured** endpoint (`Sign` or `SignAndEncrypt`). That is the case when the server offers only secured endpoints — the usual configuration for a hardened gateway — **or** when the processor has **Use Only Secure Endpoints** or **Use High Level Endpoint** checked under *Advanced Options*.
Not affected	An OPC processor whose server offers an **unsecured** endpoint *and* which has both **Use Only Secure Endpoints** and **Use High Level Endpoint** unchecked — the defaults. Such a processor negotiates an unsecured channel, no certificates are exchanged, and there is nothing to trust.
Not affected	Every non-OPC processor. Rockwell Logix, AB Legacy, Siemens S7, Modbus/TCP and GE SRTP do not use certificates at all.

To check a processor, open **Configuration** → **Processors**, open the OPC processor, and expand **Advanced Options**. If you are unsure whether a server offers an unsecured endpoint, perform the trust step anyway — trusting a client certificate that is never presented does no harm.

> **"Auto Accept Untrusted Certificates" does not cover this.** That setting decides whether *Event Engine* accepts the *server's* certificate. It has no influence on whether the server accepts Event Engine's. The two directions are trusted independently, and only the server's administrator can grant the one described here. See Chapter 6 — OPC UA Certificates and Trust.

The procedure

1. **Upgrade and start Event Engine.** On the first OPC connection attempt the new client certificate is created automatically in Event Engine's certificate store (`C:\ProgramData\Factory Data Systems, LLC\Event Engine\OpcPki\own`). Nothing needs to be generated by hand.
2. **Let one connection attempt fail.** Start scanning from **System** → **Status** so each OPC processor tries to connect. The secured attempts are refused, and the server places Event Engine's certificate in its own **rejected** (sometimes *quarantined* or *untrusted*) certificate store.
3. **On the OPC server, move that certificate to the trusted store.** Every vendor has its own screen for this — look for *trusted clients*, *client certificates*, or *rejected certificates* in the server's administration tool (in KEPServerEX this is the *OPC UA Configuration Manager*). Some servers require their runtime or service to be re-initialised before the change takes effect.
4. **Reconnect.** Stop and start scanning, or wait for the next connection attempt. The processor's tags begin updating and no further certificate errors appear on the Error List.
5. **Repeat for each OPC server.** Trust is granted per server, not per processor: two processors pointing at the same server need the step done once.

How to identify the certificate on the server

Servers list rejected certificates by subject and application URI. Event Engine's is:

Subject / issued to	`CN=Event Engine OPC UA Client`
Application URI	`urn:<hostname>:FactoryDataSystems:EventEngine` , where `<hostname>` is the name of the Windows host Event Engine runs on
Application name	`Event Engine OPC UA Client`

The application URI deliberately carries the host name, so several Event Engine installations talking to the same server are told apart by it — and each one is trusted separately.

What you will see if the step is skipped

The processor never connects. **System** → **Status** reports connection failures for it on the Error List, naming a certificate or security status — `BadCertificateUntrusted` , `BadSecurityChecksFailed` , or `BadCertificateChainIncomplete` — and every event that uses its tags fails. This does not clear on retry; the connection cannot succeed until the certificate is trusted.

When the step has to be repeated

- **The Event Engine host is renamed**, which changes the application URI and therefore the certificate identity.
- **Event Engine is moved to a different server**, which is a new installation with its own certificate.
- **The certificate is deleted or expires** and Event Engine creates a replacement.

In each case the replacement is a certificate the servers have not seen, so the same one-time step applies again. This is also why the certificate store belongs in your file-level backup of the `C:\ProgramData\Factory Data Systems, LLC\Event Engine` folder — the in-product configuration backup covers configuration only, not the certificate store.

The web interface now serves HTTPS

The upgrade itself needs no manual work — HTTPS is configured automatically — but **what your site sees changes**, so read this before operators do.

What changes on upgrade:

- **The site answers on HTTPS as well as HTTP.** The installer binds port **443**, or **8443** when 443 is already in use by something else; the finish page lists the exact addresses. On silent installs, `/HttpsPort=<n>` on the setup command line forces the port.
- **A certificate is generated for this server** — covering `localhost`, the machine name and the fully-qualified domain name — and trusted on the host itself. It is created once and reused across upgrades; a replacement is generated only when it expires or the machine is renamed.
- **A second inbound firewall rule**, `Event Engine HTTPS`, is added for the HTTPS port.
- **HTTP requests are redirected to HTTPS** while a valid certificate is bound. The redirect is deliberately conditional: if the certificate expires or its binding disappears, the site falls back to answering on plain HTTP — so a lapsed certificate can never lock operators out of the very pages needed to fix it — and the Error List says why. The redirect can also be switched off on the settings page below.

> **Browsers on other computers will warn until you act.** The generated certificate is trusted only on the server itself; every other computer shows a certificate warning for the `https://` address (the site is still reachable, and plain HTTP addresses keep working). Two remedies — either is fine:
>
> - **Import a certificate issued by your company's certificate authority** on **System → HTTPS** — a guided validate-then-import of a `.pfx` file, applied immediately with no restart (Chapter 3.6).
> - **Distribute the generated certificate** to your client computers' Trusted Root store (for example by Group Policy), if your site has no certificate authority.

Managing and opting out:

- The new **System** → **HTTPS** page shows the bound certificate — subject, the names it covers, expiry, thumbprint, and whether it is the generated one or an import — hosts the import flow and the redirect switch, and warns **30 days ahead of expiry**. Expiry, a missing binding, and a certificate left stale by a machine rename are all reported on the Error List (Chapter 3.6).
- **Opting out:** set `"EnableHttps": false` in the service's `appsettings.json` and restart the service. The setting is respected by every future upgrade — HTTPS is never forced back on.
- **Uninstalling** removes the HTTPS bindings and only the certificate the installer generated; a certificate you imported is left in the machine's store.

OPC UA communications

The new client stack brings the following with it. Existing processor configurations are read unchanged and no edit is required.

- **Endpoint selection is unchanged.** Which endpoint a processor negotiates was reproduced from the behaviour of the previous client, verified against a server, so a processor connects exactly as it did before. The one addition: servers that offer only newer security policies (the Aes and ECC families), which the previous client could not represent and therefore could not connect to at all, now work.
- **Tags recover automatically when the server's address space changes.** Rebuilding a channel, device or tag on the OPC server used to leave the affected Event Engine tags dark until a restart. Missing nodes are now re-resolved on their own.
- **Subscription health is monitored.** A subscription the server has quietly stopped publishing to is detected and rebuilt, so triggers cannot stop firing while ordinary reads still appear healthy.
- **Write-backs are faster and no longer cut short.** An event's entire write-back is sent as one `WriteNodes` call; large batches are partitioned and may run concurrently, tunable per connection. Request timeouts now scale with the size of the batch, which removes a fault where a large recipe write-back against a gateway failed at a flat five-second bound and only succeeded on the retry — visible as multi-second events and triggers that appeared to be missed.
- **The tag browser no longer truncates.** A node with more children than the server returns in a single response used to produce a short list, silently omitting the rest. The full list is now fetched.
- **Large reads are chunked.** A read batch bigger than the server's per-call limit used to fail outright; it is now split automatically.

Licensing

- **Perpetual and subscription licenses are handled distinctly**, and expiration is enforced according to the license type rather than uniformly.
- **Upgrade entitlement is checked by build date.** A build released within your license's entitlement window installs and runs; a later one is refused with an explanation, rather than installing and then failing.

- **The version and build date are signed.** Event Engine refuses to start if that signature has been tampered with.
- **Per-protocol licensing is no longer enforced.** Every protocol family is available to every licensed installation (Chapter 4).
- **Restored or cloned virtual machines are detected**, and a reactivation grace window is granted instead of the license simply failing (Chapter 4).

Installer

Setup is now a **wizard**, not a console script. The `setup.bat` that asked for the port at a command prompt is retired; the product ships as one signed `EventEngineSetup.exe` that does the whole job — license agreement, port, service registration, firewall and folder permissions (Installation & First Launch).

- A single **signed setup program**, code-signed and timestamped — and so is the uninstaller it leaves behind.
- **Prerequisites are checked before anything is installed.** Both the **.NET Runtime** and the **ASP.NET Core Runtime** are required — Microsoft's ASP.NET Core installer does not include the .NET Runtime, and a host with only one of them installed cleanly and then failed to start. Setup now names whichever is missing and exits without changing the machine.
- **The entitlement check above runs before any file is copied**, so when a build is refused your existing installation is left running, untouched.
- **An upgrade no longer resets your settings.** The installed `appsettings.json` — role groups, session timeout, SMTP relay and its stored password, log locations — is kept as it is, and the release's own defaults are written beside it as `appsettings.json.default` for comparison.
- **The service is stopped before any file is replaced**, so an upgrade cannot fail part-way on files a running service holds open. On the port page, the port already in use is pre-filled, so a customized port survives the reinstall.
- **The machine-name and domain-name URL reservations are now actually created.** A faulty check let setup report success while adding none of them, which left the site reachable only where a reservation happened to exist already. All three — `localhost`, the machine name and the fully-qualified domain name — are created, and the finish page lists the resulting addresses so you can write one down.
- **Opens the chosen web port** in Windows Firewall during setup, and re-points the rule if you reinstall on a different port.
- **Locks down the data folders** — compiled UDTs and the OPC UA certificate store are writable only by SYSTEM and Administrators, because whatever is in the certificate store decides which OPC servers this client will talk to.
- **Uninstalling reverses the setup work**: the service is stopped and removed along with the URL reservations and the firewall rule. Your data folder under `C:\ProgramData\Factory Data Systems, LLC\Event Engine` is left in place.

Interface and system

- **Pages update live.** Status, event and telemetry screens are patched in place over a live connection instead of reloading on a timer, so a page you are reading no longer jumps.
- **Start and stop reflect the real engine state** — starting, scanning and stopping are distinct, and the buttons follow them.
- **Event History gained a search filter**, links through to telemetry, details, edit and delete, and makes a cascaded run's originating event searchable (Chapter 14).
- **Spanish (es-MX) and German (de-DE)** are fully translated and selectable from the flag picker in the navigation bar; live updates stay in the language you chose.
- **Security groups are searched on demand**, and a group in another domain of a forest can be named in its `DOMAIN\Group` form (Chapter 3.3).
- **UDT handling is more robust** — namespace grouping, and the configuration backup now carries the UDT **sources** so a restore can rebuild the structure types its tags reference (Chapter 3.5).
- **A bad structure no longer sinks a whole UDT import.** A definition that fails to compile is rejected on its own and named at import time; the rest of the upload is accepted. The import is still discarded whole when rejecting one structure would orphan another that depends on it (Chapter 8.1).
- **A report of tags whose structure type no longer resolves**, with the events referencing each — read-only, because such a tag is often still in use (Chapter 9.8). The tag editor now shows an orphaned type as a disabled *Missing UDT* entry instead of letting the browser select a different type in its place.
- **An unused-tag sweep** finds tag structures no event references and deletes them in one confirmed pass (Chapter 9.8).
- **Stored procedure result sets can be discovered** from the procedure itself when configuring an event, with a badge reporting how complete the answer is. Microsoft SQL Server connections (Chapter 10.2).
- **Database table reads take filters** — a WHERE builder over the columns the event reads, with comparison, text, `In`, `Between` and null operators, all bound as parameters (Chapter 11.2).
- **Database connection pools are configurable** with minimum and maximum sizes and pool telemetry (Chapter 5).
- **Tags can record a controller symbol** beside an address, so `DB4.DBB0` is recognizable in pickers and error messages (Chapter 9.2).
- **Time-based triggers gained a bounded repeat.** A repeat interval now runs until a **Repeat Until** time (or the end of the day) and is anchored to the trigger time, so a system started mid-day joins the existing cadence instead of starting a fresh series (Chapter 7.4).
- **Tag browsing is advertised only by drivers that implement it** — Rockwell Logix and OPC. Other families show a hint to enter the address manually instead of offering a button that could not work (Chapter 9.3).

- **Subscriber failure messages are localized**, so a database or table event's failure text reaches the Error List and the controller's status message tag in the operator's language.
- **Help pages print.** Each page has a *Print / Save as PDF* button.

Known limitations

- **The generated HTTPS certificate is trusted only on the server itself.** Until a company certificate is imported (**System** → **HTTPS**) or the generated one is distributed to clients, browsers on other computers show a certificate warning for the `https://` address — see *The web interface now serves HTTPS* above. This is inherent to a self-signed certificate, not a fault.
- **OPC processors cannot use compiled UDTs.** A structure is modelled as nested Collections, which produces one tag per leaf member — the same physical data needs far more tags on an OPC processor than on a native driver. This is expected; see Chapter 6 for what it means for write-back sizing.
- **The OPC certificate trust step above is manual and per server.** There is no way for a client to grant itself trust, so it cannot be automated from this side.
- **Siemens symbolic (optimized) addressing is unavailable in this release.** The **Use symbolic (optimized) addressing** checkbox on a Siemens tag is shown unchecked and disabled, so every Siemens tag uses reference (absolute) addressing (Chapter 9, Chapter 6). Symbolic addressing is served by a separate Siemens communication driver that carries its own per-machine runtime license, which this release does not include. The support itself is complete for every data type — atomic, array, UDT/Structure and structure array — and the choice returns when that driver ships. An existing configuration containing symbolic tags still imports and loads unchanged.
- **Siemens symbolic whole-array reads use a temporary internal workaround.** The current Siemens communication driver only reads a whole array when the item is addressed from its starting element, so Event Engine transparently addresses a whole-array symbolic tag as its first element (`MyArr` is read as `MyArr[0]`). Nothing changes in your configuration and no action is needed; the workaround is removed in a future release once the corrected driver ships. It has no effect while symbolic addressing is unavailable.

Upgrade checklist

- [ ] Configuration backup taken (**System** → **Backup/Restore Configuration**)
- [ ] OPC processors reviewed for **Use Only Secure Endpoints** / **Use High Level Endpoint**
- [ ] Administrator access available on each OPC server that needs the trust step
- [ ] Upgrade installed, service running, web UI reachable
- [ ] Scanning started and each OPC server's rejected store checked for Event Engine's certificate
- [ ] Certificate moved to trusted on each server, and each processor confirmed connected
- [ ] Error List clear of certificate and security failures
- [ ] HTTPS posture decided — company certificate imported on **System** → **HTTPS**, the generated certificate distributed to client computers, or `"EnableHttps": false` set — and operators told

what the `https://` address and any browser warning mean

See also: Installation & First Launch · Chapter 6 — Processors · Support Options

---

Next: Installation & First Launch · Chapter 6 — Processors · Guide home