

# Knowledge Graph How-to

Configure your solution's data foundation to build a knowledge graph

[How-to Guides](#) [The Four Pillars How-to](#) [Data Foundation How-to](#) [Unified Namespace How-to](#) [Model Relations How-to](#)

Wire typed graph relations between Tags using Reference members and StartValue, optionally enriched with the /Attr dual-shape pattern. The result is a UNS that doubles as a queryable knowledge graph — visualizable, AI-groundable, and round-trippable with industrial ontologies.

Version 10.1.5+



## Scope of this page

This is a **step-by-step recipe**. It assumes you already understand UserTypes, Tags, and the Asset Tree from the [Unified Namespace How-to](#). For the concepts and column reference, see [UNS UserTypes Reference](#), [UNS Tags Reference](#), and [Industrial Ontology Integration How-to](#).

## Prerequisites

- [Step 1 — Create the UserTypes](#)
- [Step 2 — Wire the relation via StartValue](#)
- [Step 3 — \(Optional\) Add the /Attr metadata sibling](#)
- [Step 4 — Create the instance Tags](#)
- [Step 5 — Verify in the runtime](#)
- [Step 6 — Visualize with the Knowledge Graph control](#)
- [What this enables](#)
- [Troubleshooting](#)
- [See also](#)



## How the relation is made

In a knowledge graph, **nodes** are entities and **edges** are typed relations between them. In FrameworkX:

- Each **node** is a **Tag** — typically of a UserType.
- Each **edge** is a **UDT member declared with Type = Reference** on the source Tag's UserType. Two fields configure it:
  - Parameters** — the **target UserType** (which kind of Tag the edge may land on).
  - StartValue** — the **target Tag path**, written as <path>/Attr (which specific Tag the edge resolves to at startup). That's the whole mechanism. Steps 1 and 2 build the edge for the Pump Tank example. Step 3 adds the optional /Attr metadata sibling — a separate Tag carrying static design-sheet literals alongside the live equipment Tag.

## The example

A two-equipment water-treatment line:

```
Plant1/Pump_P1 --(FeedsInto)--> Plant1/Tank_T1
```

Pump\_P1 fills Tank\_T1. Each tag carries live process variables. You will add a typed graph relation between them and, optionally, a /Attr sibling carrying static metadata. End state: a runnable solution and a Knowledge Graph that renders the relation visually.

## Prerequisites

- A solution open in the Designer with the Unified Namespace module visible.
- Familiarity with creating UserTypes and Tags — see [Unified Namespace How-to](#).

## Step 1 — Create the UserTypes

Two UDTs. The interesting member is FeedsInto on **PumpType**: declared with **Type = Reference**, it becomes the graph edge.

### PumpType

Member	Type	Parameters	StartValue	Notes
Flow	Double			Live process value (m³/h)
Status	Text		stopped	Initial state at startup
FeedsInto	Reference	TankType	Plant1/Tank_T1/Attr	Typed pointer at the downstream Tank. See Step 2.

### TankType

Member	Type	StartValue	Notes
Level	Double		Live process value (%)
Status	Text	idle	Initial state at startup

Create both via **Unified Namespace UserTypes New**, then add the members in the grid.

For the canonical member-column definitions (every column, every constraint), see [UNS UserTypes Reference](#).

## Step 2 — Wire the relation via StartValue

A Reference UDT member needs two fields to declare a graph edge:

Field	Purpose	Example value
Parameters	The <b>target UserType</b> — what kind of Tag this Reference is allowed to point at.	TankType
StartValue	The <b>target Tag path</b> the Reference resolves to at runtime startup. Use the <code>/Attr</code> envelope of the target.	Plant1/Tank_T1/Attr

In JSON form (for `write_objects` or copy-paste into a row):

Error rendering macro 'code': Invalid value specified for parameter 'com.atlassian.confluence.ext.code.render.InvalidValueException'

```
{
  "Name": "FeedsInto",
  "Type": "Reference",
  "Parameters": "TankType",
  "StartValue": "Plant1/Tank_T1/Attr"
}
```

When the runtime starts, every `PumpType` instance gets its `FeedsInto.Link` resolved to `Plant1/Tank_T1/Attr` automatically. The `FrameworkX` object model now has a typed edge from `Pump_P1` to `Tank_T1`.



### Target the /Attr envelope, not the naked tag

The canonical Reference target is the **target Tag's /Attr envelope** — for example `Reactor_R101/Attr`, not `Reactor_R101`. The `/Attr` suffix is stripped on RDF export so the OWL entity IRI reflects identity, not storage layout (see [UNS Asset Tree Reference](#) for the export contract). Pointing at the naked tag may still produce a working runtime edge, but the OWL round-trip will not match the documented convention.



### StartValue scope in 10.1.5

**StartValue declared on the UDT member applies to every instance.** If two `PumpType` tags must feed different Tanks, you currently need per-instance overrides — and per-instance member `StartValue` is **not yet writable from the Tags grid in 10.1.5** (the UI write surface is in-flight; see [UNS Asset Tree Reference](#) troubleshooting). For single-instance examples and tutorial scenarios, the UDT-level `StartValue` used above is sufficient and is the simplest path. For multi-instance solutions today, set the per-instance `StartValue` via `write_objects` or `T.Eng.Project` until the UI catches up.

## Step 3 — (Optional) Add the /Attr metadata sibling

The `/Attr` dual-shape pattern stores static, design-sheet-style metadata on a sibling tag — separate from the live equipment tag. The naked tag carries dynamic process variables; the `/Attr` sibling carries literals like `Manufacturer`, `ModelNumber`, `InstallationDate`.

When to use which shape:

Use case	Pattern
Plain process equipment, no descriptive metadata	Naked tag only
Conceptual containers (Enterprise, Site, Area) — folders, not equipment	<code>/Attr</code> only
Equipment + design-sheet metadata + ontology round-trip	Both (naked + <code>/Attr</code> siblings)

For `Pump_P1` to carry both live process data AND nameplate metadata, declare a second UDT:

**PumpType\_Attr**

Member	Type	StartValue	Notes
Manufacturer	Text	Grundfos	Static literal, never updated at runtime
ModelNumber	Text	CR-32-4	Static literal
MaxFlowRate	Double	50	Design rating (m³/h)

Then create a sibling Tag at `Plant1/Pump_P1/Attr` of type **PumpType\_Attr** (see Step 4).

For the full dual-shape rationale and the OWL/RDF round-trip semantics, see [Industrial Ontology Integration How-to](#) — sections "**Two paradigms**" and "**FrameworkX architecture and ontology alignment**".

## Step 4 — Create the instance Tags

In **Unified Namespace Asset Tree**, create the folder `Plant1` and add three tags inside it:

Tag path	Type	Notes
<code>Plant1/Pump_P1</code>	<b>PumpType</b>	Live pump
<code>Plant1/Pump_P1/Attr</code>	<b>PumpType_Attr</b>	Optional — static nameplate metadata (Step 3)
<code>Plant1/Tank_T1</code>	<b>TankType</b>	Live tank

Either:

- right-click the `Plant1` folder **New Tag**, set the **Type** to the UDT, OR
- paste a row into the **Tags** grid with the **Name** and **Type** columns set.

The Asset Tree auto-creates the folder hierarchy from the slashes in the tag path.

## Step 5 — Verify in the runtime

Start the runtime (F5). With the configuration above, the FrameworkX object model now exposes:

```
@Tag.Plant1/Pump_P1          <- PumpType instance (live)
@Tag.Plant1/Pump_P1.Flow     <- Double, live
@Tag.Plant1/Pump_P1.Status   <- Text, live
@Tag.Plant1/Pump_P1.FeedInto <- Reference -> Tank_T1
@Tag.Plant1/Pump_P1.FeedInto.Link <- "Plant1/Tank_T1/Attr"

@Tag.Plant1/Pump_P1/Attr     <- PumpType_Attr instance (static, optional)
@Tag.Plant1/Pump_P1/Attr.Manufacturer <- "Grundfos"

@Tag.Plant1/Tank_T1         <- TankType instance (live)
@Tag.Plant1/Tank_T1.Level   <- Double, live
@Tag.Plant1/Tank_T1.Status  <- Text, live
```

Quick sanity check from a Script or the runtime monitor:

### Confirm the Reference resolved at startup

```
// Read the resolved target through the Reference
string target = @Tag.Plant1/Pump_P1.FeedInto.Link;
// Expected: "Plant1/Tank_T1/Attr"

// Read a value of the target tag through the Reference
double level = @Tag.Plant1/Pump_P1.FeedInto.Value; // routed to Plant1/Tank_T1
```

For the full Reference-tag runtime semantics, see [UNS Tags Reference Reference Type](#).

## Step 6 — Visualize with the Knowledge Graph control

1. Open **Unified Namespace Asset Tree**. Click the **Knowledge Graph** button at the top of the tree. This regenerates `SolutionSettings.KnowledgeGraphSource` from the current UDT + Tag + relation state.
2. Open or create a Display. From the **Components Panel Charts**, drop **Knowledge Graph** onto the canvas.
3. In the control's **Properties** panel:
  - Bind **Selected node path** to a Tag of type Text (for example `Tag.UI.SelectedNodePath`). The control writes the clicked node's full UNS path (dot notation, e.g. `Plant1.Tank_T1`) into that Tag.
  - Bind **Selected node type** to a Tag of type Text (for example `Tag.UI.SelectedNodeType`). The control writes the clicked node's UserType name (e.g. `TankType`) into that Tag.
4. Run the Display. Clicking a node updates the two bound Tags. Wire those Tags to a ChildDisplay source, a Trend Chart, or any other control to drive type-aware drill-down.

The expected render for this example:

```

+-----+   FeedsInto   +-----+
| Pump_P1 | -----> | Tank_T1 |
| PumpType |          | TankType |
+-----+             +-----+

```

For control configuration depth — render modes, source regenerators, HTML5 / OpenSilver parity, design-time preview semantics — see [KnowledgeGraph Control Reference](#).

## What this enables

Once the UNS carries Reference edges (and, optionally, `/Attr` metadata siblings), the same three building blocks — Reference members, StartValue, dual-shape — unlock four capabilities:

- **Visualize** the plant graph with the Knowledge Graph Display control (Step 6).
- **Ground AI queries** — the [Local AI](#) assistant walks Reference edges to answer questions like *"what feeds into this tank?"* or *"trace upstream of Pump\_P1"*.
- **Export to RDF / OWL / JSON-LD / Turtle / N-Triples** — see [Export your UNS to RDF/OWL/GraphDB](#). The `/Attr` suffix is stripped on export so OWL entity IRIs match the source ontology.
- **Re-import enriched ontologies** from external authoring tools via [Industrial Ontology Integration How-to](#), with the `SourceIri` column providing the join key for diff and overlay.

## Troubleshooting

**Reference member shows no value at runtime.** Confirm `Parameters` is set to the target UserType name (case-sensitive) and `StartValue` is the path of an existing Tag. The target must exist when the runtime starts; references to missing tags resolve to null.

**Knowledge Graph control shows no edges.** Click the **Knowledge Graph** button on the Asset Tree to regenerate `KnowledgeGraphSource`, or invoke `TK.GenerateUnsVisual()` from a Script. Edits to UDTs or Tags only refresh the source on the next regeneration; auto-refresh applies on subsequent renders.

**Multiple PumpType instances all feed the same Tank.** UDT-level `StartValue` applies to every instance. Per-instance override via the Tags grid is in-flight in 10.1.5 — set per-instance `StartValue` via `write_objects` (DesignerMCP) or `T.Eng.Project` (Engineering API) until the UI surface ships. See [UNS Asset Tree Reference](#) troubleshooting for the current status.

**OWL round-trip drops the relation.** The Reference target must point at the `/Attr` envelope (`Tank_T1/Attr`, not `Tank_T1`). The exporter strips the `/Attr` suffix on the way out so the OWL IRI is clean; an export that targets the naked tag will not match the round-trip contract documented in [Industrial Ontology Integration How-to](#).

## See also

- [Unified Namespace How-to](#) — basic UNS configuration (prerequisite).
- [UNS UserTypes Reference](#) — UDT member-column definitions.
- [UNS Tags Reference](#) — Reference Type runtime semantics; ontology columns (`Labels`, `SourceIri`, `Attributes`).
- [UNS Asset Tree Reference](#) — dual-shape folder layout; per-instance `StartValue` status.
- [Industrial Ontology Integration How-to](#) — standards coverage; the "Two paradigms" section.
- [KnowledgeGraph Control Reference](#) — control properties, source regenerators, render modes.
- [Local AI](#) — AI grounding on the UNS graph.
- [LocalAI KnowledgeGraph Demo](#) — reference solution exercising every column.
- [RDF Triples and the Industrial Ontology Foundry \(IOF\)](#) — the triples model behind the round-trip.

In this section...

