UNS UserTypes Reference

Create reusable equipment templates (UserTypes).

> ⚠ **Two distinct "reference" concepts in FrameworX — do not confuse**
>
> UDTs on this page can declare members of type `Reference`, but the word "reference" covers two different mechanisms in FrameworX. Disambiguating up front:
>
> **(a) A `Reference`-type UDT member** — the case covered on this page. Declare a member with `Type=Reference`; on each instance, its value points to another tag's `/Attr` envelope (for example, `Mixer_M101.feedsInto = Reactor_R101/Attr`). This is the FX representation of an OWL `owl:ObjectProperty` — an ontology relation between named individuals — and round-trips through the RDF /OWL importer and exporter. See Industrial Ontology Integration How-to.
>
> **(b) The `Reference` tag type** — a separate mechanism, covered on the sibling UNS Tags Reference page. A runtime tag whose `.Link` property points to another tag of the Reference's target type, used for runtime pointer / alias behavior (one display driving many equipment instances). See the Tag Reference Open Display Example.

UNS UserTypes (Reference), also referred as UDTs (formerly DataTemplates), extend the platform's predefined data types by defining custom types with properties tailored to business needs, enabling modeling of solution-specific data structures like equipment status and asset attributes. A UserType provide:

- Custom data type definitions
- Reusable tag structures
- Hierarchical data modeling
- Consistent asset representation
- Property inheritance
- Template-based tag creation

UserTypes allow creating complex data structures that standard types cannot represent, such as pumps, motors, tanks, or production lines.

Ontology Metadata (10.1.5+)
AI / MCP page action (10.1.5+)
Creating UserTypes
UserType Properties
Using UserTypes
Nested Templates
Base Type Inheritance
Security Settings
Member Configuration
Relative Addressing
Important Notes
Common Use Cases
Best Practices Checklist
Troubleshooting

Ontology Metadata (10.1.5+)

Every UserType carries ontology metadata alongside the familiar configuration properties. Import populates these columns from RDF/OWL source files, export serializes them back to triples, and semantic search ranks matches across them. See Industrial Ontology Integration How-to for the full model.

Column	Where	Storage	UI	Round-trip
DisplayText	UserType row	Text, 255 chars	UserType properties grid	`rdfs:label` (primary) on the class IRI
Labels	UserType row	Text, unbounded, semicolon-delimited	UserType properties grid, label-chips	`skos:altLabel` per entry
SourceIri	UserType row	Text, unbounded	UserType properties grid	Subject IRI for class triples
BaseUserType	UserType row	Name reference to another UnsUserTypes row (resolved Name ID at write time). Enables inheritance. Inherited members are not shown in the Designer grid — they appear only in the Asset Tree and at runtime.	UserType properties grid (dropdown)	`rdfs:subClassOf <parent IRI>`
Attributes	UserType row	JSON column, unbounded. **Design-time only** at UserType level (class-level TBox annotations).	UserType properties grid, expandable JSON editor	Individual triples per `prefix:localName` (language-tagged for `@lang`-suffixed keys)

Member-def ontology columns (10.1.5+)

The UDT member grid gained two ontology columns in 10.1.5:

Column	Where	Storage	UI	Round-trip
SourceIri	Member-def row	Text, unbounded	UDT member grid column	Predicate IRI for the property declaration

Attributes	Member-def row	JSON column, unbounded. Design-time only.	UDT member grid (Attributes column)	Predicate-level annotations on the property declaration

Why no Labels column on member-def: property-level synonyms (`skos:altLabel` on data properties) are rare in real industrial ontologies (IOF, ISA-88, ISA-95) — altLabels cluster at the class level. `DisplayText` already covers the primary human-readable member name.

Mutability & retention

The ontology columns follow a **class vs. instance** split that matches W3C and industrial-ontology conventions: **IRIs are identity (immutable everywhere); Labels are `skos:altLabel` synonyms (edited on the class at design time, on tag instances at runtime); Attributes is an extensibility bag (design-time at class and member-def levels, runtime-mutable only at tag level)**.

Layer	SourceIri	Labels	Attributes
UserType (this page)	Design-time · immutable identity	Design-time	Design-time
Member-def (this page)	Design-time	Not modeled	Design-time
Tag instance (see UNS Tags Reference)	Design-time · immutable	Runtime-editable · retentive	Runtime-editable · retentive

Bindings can *read* all three columns on every layer; *writes* go through script (`@Tag.X.SetAttribute(...)`) or the Designer editors — never through bindings directly. For RDF/OWL export, per-tag runtime mutations on `Labels` / `Attributes` must be snapshotted back to the cold-start data with `promote_retentive_attributes` before running the exporter.

AI / MCP page action (10.1.5+)

When Designer is navigated to the UserTypes page, the following action is exposed via the `designer_action` MCP tool:

- `designer_action("relationship_graph")` — mirrors the **Knowledge Graph** toolbar button. Generates an interactive HTML view (pan, zoom, filter via cytoscape.js) and opens it in the default browser. A Mermaid Markdown side-car lands next to the HTML in the Exchange folder under `Visualizations/` for diff and source-control use.

The action is listed in the `tabActions` array returned by `get_state()` only when the UserTypes tab is active. See Generate a visual report of your UNS for the broader visual-report reference.

Creating UserTypes

1. Navigate to **Unified Namespace UserTypes**
2. Click **New** button
3. In dialog:
    - **Name**: Template identifier
    - **Description**: Documentation
4. Click **OK**
5. Add members in grid:
    - Type in first row
    - Press Enter to add
    - Configure properties per member

UserType Properties

Property	Description	Required
Name	Member name within template	Yes
Type	Data type (built-in or nested template)	Yes
Array	Array size for member	No
Parameters	Type-specific settings	No
Min/Max	Value limits	No
ScaleMin /Max	Engineering scale	No
Units	Engineering units	No
Format	Display formatting	No

Enumeration	Value mappings	No
StartValue	Initial value	No
Retentive	Value persistence	No
Domain	Server/Client scope	No
Visibility	External access	No
DisplayText	UI display text. Imports from `skos:prefLabel` or primary `rdfs:label` (10.1.5+).	No
Labels	Alternative names, semicolon-delimited. UserType class level only (not on member-def). Imports from `skos:altLabel`. Searchable by `search_uns`. Design-time only on the UserType row — see "Mutability & retention" above (10.1.5+).	No
SourceIri	Full source IRI for round-trip with external ontologies. Populated by the ontology importer, user-editable at design time. Immutable at runtime (identity — see "Mutability & retention" above) (10.1.5+).	No
BaseUserType	Parent UserType for single-parent inheritance. The child automatically receives all parent members, which are not shown in the Designer grid but appear in the Asset Tree and at runtime. See "Base Type Inheritance" below. Maps to `rdfs:subClassOf` on export (10.1.5+).	No
Attributes	JSON column for ontology annotations not covered by other columns (`dcterms:*`, alt-language labels, custom predicates). **Design-time only** at UserType and member-def level — edit in Designer and re-publish. For runtime-mutable per-tag annotations see the UNS Tags Reference (10.1.5+).	No
RelativeAddress	Device addressing	No
Description	Member documentation. Defined once on the UserType and shared by every tag instance of the member. Tag Properties shows it read-only on instances. To annotate one instance, set the Description on the instance tag itself. From FrameworX 10.1.6, the engineering API follows the same rule: Description is not part of the per-instance override set. See EngWrapper API Reference.	No

---

Using UserTypes

Creating Tags from Templates

1. Go to **Unified Namespace Tags**
2. Click **New Item**
3. In **Type** dropdown, select template
4. Tag inherits all template members

Template Instance Example

```
Template: Motor
Members:
  - Running (Digital)
  - Speed (Double)
  - Temperature (Double)
  - AlarmStatus (Integer)

Tag: Motor1 (Type: Motor)
Access:
  @Tag.Motor1.Running
  @Tag.Motor1.Speed
  @Tag.Motor1.Temperature
  @Tag.Motor1.AlarmStatus
```

When the UserType represents an ontology-imported equipment class (10.1.5+)

When the UserType comes from an OWL ontology import (e.g., `S88_Unit` auto-created from `s88:Unit` by `import_graph_model`), instantiate it TWICE per real-world entity to capture both halves of the dual-shape pattern:

- One naked UserType-instance Tag at `<path>/<entityName>` — live equipment members (Temperature, Speed, Status) fed at runtime by Devices, Scripts, or simulators.
- One `/Attr` Tag at `<path>/<entityName>/Attr` — static ontology metadata members (rdfs:label, dcterms:identifier, manufacturer, classification) with `StartValue` defined. The importer creates this one automatically.

Both tags live as siblings under the auto-created folder. Conceptual containers (Enterprise / Site / Area / Cell with only Name + Description literal members) typically use `/Attr`-only. See Industrial Ontology Integration How-to Two paradigms for the full pattern.

Nested Templates

Templates can contain other templates:

```
Template: PumpStation
Members:
  - Pump1 (Type: Motor)
  - Pump2 (Type: Motor)
  - FlowRate (Double)
  - Pressure (Double)

Access:
  @Tag.Station1.Pump1.Running
  @Tag.Station1.Pump2.Speed
```

Per-Instance Start Values on Nested Members

Available from FrameworX 10.1.6. A per-instance start value also reaches members of a nested UserType, not only direct members. Address the nested member with a dot-separated path, for example `Drive.MaxHz` or `Drive.Leaf.Speed`. The path is accepted when it ends on a basic value member of the type. Each instance keeps its own values: a start value set on one instance does not change other instances or the UserType definition.

A name not ending on a basic value member is skipped, and the skip message states the rule. This applies to a member typed as a UserType, a Reference member, a slash-separated path and a member missing from the type.

When you work through the AI Designer tools, set these values in the `UnsTagsProperties` table, with the dotted path as the member `Name`. In 10.1.6, reading back a tag's per-instance settings lists direct members only, so a nested start value is stored but not listed on read. In 10.1.5, per-instance start values apply to direct members only.

Base Type Inheritance

A UserType can declare another UserType as its **Base Type** (the `BaseUserType` property). This creates a parent-child relationship: the child automatically receives all of the parent's members and stays in sync when the parent changes.

How it works

- **Inherited members are not shown in the Designer.** The child's member grid in the Designer only shows the members defined directly on that type. Members from the base type are not listed there — to see or edit them, open the base type itself.
- **Inherited members appear in the Asset Tree and at runtime.** When a tag is created using the derived type, all members — both inherited and own — are fully available in the Asset Tree and accessible at runtime (e.g., `@Tag.Motor1.Speed`).
- **Inherited members stay in sync.** Any change to a member in the base type — name, range, units — is immediately reflected in every derived type without any action needed on the child.
- **The child can add its own members.** Additional rows defined on the derived type exist only on that type and are fully editable in the Designer.

Example

```
Base type: Motor
Members defined in Designer:
  - Running: Digital
  - Speed: Double (0-1800 RPM)
  - Temperature: Double (0-200 C)

Derived type: PremiumMotor (Base Type: Motor)
Members shown in Designer (own only):
  - PowerFactor: Double
  - EfficiencyClass: Text

At runtime / in Asset Tree, PremiumMotor tags expose all members:
  - Running, Speed, Temperature  (from Motor)
  - PowerFactor, EfficiencyClass (own)
```

Setting the Base Type

1. Navigate to **Unified Namespace UserTypes**.
2. Select the UserType row to configure.
3. In the properties panel, pick the parent UserType from the **BaseUserType** dropdown.
4. The child is now linked to the base. Its own member grid shows only the child's direct members; the inherited ones appear when the type is used in the Asset Tree or at runtime.

> ⚠ Inheritance is single-parent: a UserType can have at most one Base Type. Circular references (A inheriting from B while B inherits from A) are not allowed and are prevented by the Designer.

Security Settings

Property	Controls	Applies To
EditSecurity	Modify permission	Design-time
ReadSecurity	Read access	Runtime
WriteSecurity	Write access	Runtime

Member Configuration

Supported Base Types

- Digital
- Integer
- Long
- Double
- Decimal
- Text
- DateTime
- Custom Templates

Array Members

```
Template: TankFarm
Members:
  - Tanks (Type: Tank, Array: 10)

Access:
  @Tag.TankFarm1.Tanks[0].Level
  @Tag.TankFarm1.Tanks[1].Temperature
```

Relative Addressing

For device communication:

```
Template: AnalogInput
Members:
  - Value (RelativeAddress: ".PV")
  - Status (RelativeAddress: ".ST")

Device mapping automatically appends to base address
```

Important Notes

When setting properties for array elements or template members (StartValue, Min, Max, Enumeration), these values will not display in the Designer DataGrid. Access them through the Tag Properties dialog or at runtime.

From FrameworX 10.1.6, the per-instance override set is Start Value, Min, Max, Disable, Display Text and Enumeration. The member Description belongs to the UserType and is the same on every instance.

From FrameworX 10.1.6, a Reference member of a UserType instance points at runtime to the target set for that instance, not to the UserType default. Set the target with **Set Reference Target** on the Asset Tree or the Tags page. A target that no longer resolves leaves the reference empty.

From FrameworX 10.1.6a, a Reference member with the target type **Dynamic** accepts a tag of any type, including a UserType tag. When the target is a UserType tag, the member itself has no value, the same as an empty reference: a display binding, `TK.GetObjectValue` and `TK.SetObjectValue` on the member read or write nothing. The same applies to each element of an array of Dynamic references. Read the member's `Link` property to find the linked tag. A Dynamic reference gives no access to the members of its target. To reach them, set the target type to the UserType of the target tag. See UNS Tags Reference Reference Type.

Property changes to one instance do not propagate to others. Each template instance operates independently — so one UserType can be reused across screens with a different **Enumeration** on each instance's member. See UNS Tags Reference Per-instance enumeration override.

Common Use Cases

Motor Template

```
Members:
  - Running: Digital
  - Speed: Double (0-1800 RPM)
  - Current: Double (0-100 Amps)
  - Temperature: Double (0-200 °C)
  - HoursRun: Long
  - LastMaintenance: DateTime
```

Tank Template

```
Members:
  - Level: Double (0-100 %)
  - Temperature: Double (-50-150 °C)
  - Pressure: Double (0-10 Bar)
  - HighAlarm: Digital
  - LowAlarm: Digital
  - Product: Text
```

Production Line

```
Members:
  - Station1: Motor
  - Station2: Motor
  - ConveyorSpeed: Double
  - ProductCount: Integer
  - BatchID: Text
  - QualityScore: Double
```

Best Practices Checklist

- **Plan template hierarchy** - Design before implementation
- **Use meaningful names** - Clear member identification
- **Document templates** - Describe purpose and usage
- **Standard units** - Consistent engineering units
- **Reuse templates** - Avoid duplication
- **Version control** - Track template changes
- **Test thoroughly** - Verify all members

Troubleshooting

Template not appearing:

- Save after creation
- Check for naming conflicts
- Verify no circular references
- Review template hierarchy

Members not accessible:

- Confirm tag uses template type
- Check member names
- Verify array indices
- Review security settings

Values not retained:

- Check retentive settings
- Verify database connection
- Review member configuration
- Test with simple template

In this section...
