Skill New Solution

Create a complete starter solution from scratch: plan UNS structure, create tags, connect a Value Simulator, configure historian and alarms, build a dashboard, and start runtime.

What This Skill Does

Build a complete starter solution from scratch: plan the UNS tag structure, create tags organized by asset, connect a Value Simulator for testing, configure historian logging and basic alarms, create a dashboard display, and start runtime so the user sees live data.

This skill follows the **Four Pillars** build order:

1. UNS (Tags) 2. Industrial (Devices, Alarms, Historian) 3. Business (Scripts) 4. Client (Displays)

When to Use This Skill

Use when:

- User says "create a new solution" or "get me started"
- User wants a demo or proof-of-concept
- User is new to FrameworX
- Building a solution with simulated data for testing

Do NOT use when:

- User has a specific protocol in mind (Modbus, OPC UA, etc.) — adapt the Devices steps accordingly, but the overall flow remains the same
- User only needs to modify an existing solution
- User wants a specific advanced feature (use the relevant module skill)

Prerequisites

- FrameworX Designer installed and running
- MCP tools connected

MCP Tools and Tables

Category	Items
Tools	`create_solution`, `get_table_schema`, `write_objects`, `get_objects`, `list_protocols`, `list_elements`, `list_dynamics`, `designer_action`, `get_solution_info`
Tables	`UnsEnumerations`, `UnsUserTypes`, `UnsTags`, `DevicesChannels`, `DevicesNodes`, `DevicesPoints`, `HistorianHistorianTables`, `HistorianHistorianTags`, `AlarmsItems`, `DisplaysList`

Implementation Steps

Step 1: Create the Solution

```
create_solution(name='MyFirstSolution', template='Blank')
```

Templates available:

- **Blank** — empty solution, you build everything
- **HeaderLayout** — pre-configured layout with header region (saves layout work later)

`create_solution` creates a new solution and opens it. Use `open_solution(name)` only for a solution that already exists — it has no `template` parameter.

After `create_solution` completes, the full engineering context is delivered. Proceed to the next step — do not pre-fetch schemas.

> ***Keep the `solution_id`.*** *The `create_solution` (or `open_solution`) response returns a `solution_id`. It is the required routing key on `get_objects`, `write_objects`, `designer_action`, `get_solution_info` and the other session-bound tools — omitting it fails the call with `MISSING_SOLUTION_ID`. Pass it on every call below. Solution-independent tools (`get_table_schema`, `list_protocols`, `list_dynamics`, `search_docs`) do not take it.*

Step 2: Plan the UNS (Tag Structure Design)

Before creating any tags, analyze the user's requirements and decide on the tag structure.

Key decision: Simple variables vs UserTypes (UDTs).

- **Simple variables** — use when equipment is unique or has few tags. Each tag is an independent path: `Plant/Tank1/Level`, `Plant/Tank1/Temperature`.
- **UserTypes** — use when multiple instances share the same member structure. Define a `Motor` type once with members `Speed`, `Current`, `Running`, `Fault` — then instantiate as `Plant/Line1/Motor1`, `Plant/Line1/Motor2`. All instances auto-get the same members.

Decision rule: If the user describes 2+ pieces of identical equipment (e.g., "3 pumps", "Tank1 and Tank2 with the same readings"), use a UserType. If equipment is one-off or members differ, use simple variables.

Naming convention: Use consistent asset paths with `/` folders so the same display template can be reused across instances (e.g., `Plant/Tank1/...`, `Plant/Tank2/...`). This enables asset-driven navigation later.

For this starter skill, we use simple variables. For UserType-based solutions, fetch `get_table_schema('UnsUserTypes')` and define types before creating tags.

Step 3: Create Tags (Pillar 1 — UNS)

> ***PILLAR BOUNDARY — Pillar 1: UNS*** *Complete all tag design and creation before moving to Pillar 2. Do NOT fetch Device, Alarm, Historian, or Display schemas during this step.*

Fetch the tag schema first:

```
get_table_schema('UnsTags')
```

Create tags organized by asset path using `/` folders:

```
{
  "table_type": "UnsTags",
  "data": [
    { "Name": "Plant/Tank1/Level",       "Type": "Double", "Description": "Tank 1 level %" },
    { "Name": "Plant/Tank1/Temperature", "Type": "Double", "Description": "Tank 1 temp °C" },
    { "Name": "Plant/Tank1/Pressure",    "Type": "Double", "Description": "Tank 1 pressure bar" },
    { "Name": "Plant/Tank1/ValveOpen",   "Type": "Digital", "Description": "Inlet valve state" },
    { "Name": "Plant/Tank2/Level",       "Type": "Double", "Description": "Tank 2 level %" },
    { "Name": "Plant/Tank2/Temperature", "Type": "Double", "Description": "Tank 2 temp °C" },
    { "Name": "Plant/Tank2/Pressure",    "Type": "Double", "Description": "Tank 2 pressure bar" },
    { "Name": "Plant/Tank2/ValveOpen",   "Type": "Digital", "Description": "Inlet valve state" }
  ]
}
```

IMPORTANT — Digital tags. The `Value` of a Digital tag is the number 0 or 1, so it can be used in writes to PLCs and in math expressions. When an expression needs a logical value (true/false), use the `State` property instead:

```
if ( @Plant/Tank2/ValveOpen.State ) { ... }
```

The `State` property evaluates to `true` when the value is 1, and `false` when the value is 0.

Step 4: Connect Value Simulator (Pillar 2a — Devices)

> ***PILLAR BOUNDARY — Pillar 2: Industrial*** *Pillar 1 (UNS) must be written and confirmed before starting here. Now fetch Device, Historian, and Alarm schemas as needed — one module at a time.*

The Value Simulator generates changing data for testing — no real device needed. It's a built-in protocol.

4a: Get Protocol Details

```
list_protocols('ValueSimulator')
```

This returns the protocol-specific field formats for Channel, Node, and Point configuration. **Never guess these formats** — always fetch first.

4b: Fetch Device Schemas

```
get_table_schema('DevicesChannels,DevicesNodes,DevicesPoints')
```

4c: Create the 3-Table Pipeline

Devices follow a strict pipeline: **Channel → Node → Point**. All three are required, and dependencies must exist before referencing objects.

Use multi-table `write_objects` to create all three in one call (dependency order is handled automatically):

IMPORTANT: 'Name' and 'Protocol' are mandatory when creating Device Channels. The Protocol value, should be exactly as it shows in list_protocols

```
[
  {
    "table_type": "DevicesChannels",
    "data": [{ "Name": "SimChannel", "Protocol": "ValueSimulator" }]
  },
  {
    "table_type": "DevicesNodes",
    "data": [{ "Name": "SimNode", "Channel": "SimChannel" }]
  },
  {
    "table_type": "DevicesPoints",
    "data": [
      { "TagName": "Plant/Tank1/Level",       "Node": "SimNode", "Address": "<Dependent from the Protocol
schema>" },
      { "TagName": "Plant/Tank1/Temperature", "Node": "SimNode", "Address": "<Dependent from the Protocol
schema>" },
      { "TagName": "Plant/Tank1/Pressure",    "Node": "SimNode", "Address": "<Dependent from the Protocol
schema>" },
      { "TagName": "Plant/Tank1/ValveOpen",   "Node": "SimNode", "Address": "<Dependent from the Protocol
schema>" },
      { "TagName": "Plant/Tank2/Level",       "Node": "SimNode", "Address": "<Dependent from the Protocol
schema>" },
      { "TagName": "Plant/Tank2/Temperature", "Node": "SimNode", "Address": "<Dependent from the Protocol
schema>" },
      { "TagName": "Plant/Tank2/Pressure",    "Node": "SimNode", "Address": "<Dependent from the Protocol
schema>" },
      { "TagName": "Plant/Tank2/ValveOpen",   "Node": "SimNode", "Address": "<Dependent from the Protocol
schema>" }
    ]
  }
]
```

Important: TagName in DevicesPoints must use the **full tag path** including asset folders: `Plant/Tank1/Level` — not bare `Level`.

Adapt the protocol-specific fields (Address, ProtocolOptions, etc.) based on what `list_protocols('ValueSimulator')` returned.

Step 5: Configure Historian (Pillar 2b)

5a: Check if Default Table Exists

Most solutions include a default StorageLocation and Table1:

```
get_objects('HistorianHistorianTables', solution_id=<id>)
```

If Table1 exists, skip to 5b. If not, create the pipeline:

```
get_table_schema('HistorianStorageLocations,HistorianHistorianTables')
```

5b: Log Tags to Historian

```
get_table_schema('HistorianHistorianTags')
```

```
{
  "table_type": "HistorianHistorianTags",
  "data": [
    { "TagName": "Plant/Tank1/Level",       "TableName": "Table1" },
    { "TagName": "Plant/Tank1/Temperature", "TableName": "Table1" },
    { "TagName": "Plant/Tank1/Pressure",    "TableName": "Table1" },
    { "TagName": "Plant/Tank2/Level",       "TableName": "Table1" },
    { "TagName": "Plant/Tank2/Temperature", "TableName": "Table1" },
    { "TagName": "Plant/Tank2/Pressure",    "TableName": "Table1" }
  ]
}
```

Tuning tips (optional for a starter solution):

- `DeadBand`: minimum value change before logging (0 = log every change)
- `Deviation`: overrides interval-based logging when value changes significantly
- Boolean tags (ValveOpen) typically don't need historian logging in a starter

Step 6: Add Basic Alarms (Pillar 2c)

Predefined groups exist: Critical, Warning, AuditTrail. Verify with `get_objects('AlarmsGroups', solution_id=<id>)` before creating new groups — don't waste a call creating groups that already exist.

```
get_table_schema('AlarmsItems')
```

Important: The field names shown below are representative examples. Always verify exact property names against the `get_table_schema ('AlarmsItems')` response before writing — the schema is the source of truth.

```
{
  "table_type": "AlarmsItems",
  "data": [
    { "TagName": "Plant/Tank1/Level",       "Group": "Critical", "Condition": "HiHi", "Limit": 95, "Message":
"Tank 1 level critical high" },
    { "TagName": "Plant/Tank1/Level",       "Group": "Warning",  "Condition": "Hi",   "Limit": 85, "Message":
"Tank 1 level high warning" },
    { "TagName": "Plant/Tank1/Level",       "Group": "Warning",  "Condition": "Lo",   "Limit": 15, "Message":
"Tank 1 level low warning" },
    { "TagName": "Plant/Tank1/Level",       "Group": "Critical", "Condition": "LoLo", "Limit": 5,  "Message":
"Tank 1 level critical low" },
    { "TagName": "Plant/Tank1/Temperature", "Group": "Critical", "Condition": "HiHi", "Limit": 90, "Message":
"Tank 1 temp critical high" },
    { "TagName": "Plant/Tank1/Temperature", "Group": "Warning",  "Condition": "Hi",   "Limit": 80, "Message":
"Tank 1 temp high warning" }
  ]
}
```

TagName must use the full path — `Plant/Tank1/Level`, not bare `Level`.

Repeat a similar pattern for Tank2 tags as appropriate.

Step 7: Create Dashboard (Pillar 4 — Displays)

> ***PILLAR BOUNDARY — Pillar 4: Interaction*** *Pillar 2 (Industrial) must be complete before building displays. Fetch Display and element schemas only now.*

Now build the operator interface. For a starter solution, write directly into the predefined **MainPage** display.

Why MainPage — it is both the first screen and the preview. The **Startup** layout (the application frame) has five fixed regions: Header, Footer, Menu, Submenu, and **Content**. The page assigned to the **Content** region is two things at once — the screen the application opens to at runtime, *and* the thumbnail Solution Center renders as this solution's preview. Templates wire Content to `Display.MainPage` (the "Project home page"), so writing your home screen into `MainPage` lands it in both places automatically.

If you build the home screen as a different page instead of MainPage, point the Content region at it: read the Startup layout in full (`get_objects ('DisplaysLayouts', names=['Startup'], detail='full', solution_id=<id>)`), set the `Content` member's `Page` to `Display.<YourPage>` in its `Members` array, and write the whole layout back (`DisplaysLayouts` is a document object — full replacement).

Make it presentable. A project-name cover with at-a-glance status is a good default home, because this page is the operator's first impression *and* the preview customers see in Solution Center.

> ***Hard rule — never finish a solution with the Content page blank.*** *If the Startup layout's Content region points at an empty or placeholder page, the application opens to a blank screen* and *the Solution Center preview is blank. The preview is a cached screenshot, refreshed whenever that page is rendered and saved — saving the page in the Designer editor, or building/publishing the solution, regenerates it. Starting runtime (Step 8) publishes the solution, so a normal new-solution build refreshes the preview automatically.*

This step covers a **Dashboard** (grid-based responsive layout). For **Canvas displays** (pixel-precise positioning, process diagrams, P&ID-style screens), or for advanced display features like visual dynamics, symbol wiring, and CodeBehind — **load the Display Construction skill**: `search_docs ('displays', labels='skill')`.

7a: Fetch Element Schemas

```
list_elements('Dashboard,TrendChart,RadialGauge,TextBlock')
get_table_schema('DisplaysList')
```

7b: Write the Dashboard

CRITICAL: Verify the exact Dashboard JSON structure against `list_elements('Dashboard')` output. The schema returned by that call is the source of truth for property names (DashboardDisplay, Cells, Columns, Rows, etc.).

```
{
  "table_type": "DisplaysList",
  "data": [{
    "ObjectName": "MainPage",
    "PanelType": "Dashboard",
    "DashboardDisplay": {
      "Columns": ["*", "*", "*", "*"],
      "Rows": ["Auto", "*", "*"]
    },
    "Cells": [
      {
        "Row": 0, "Col": 0, "ColSpan": 4,
        "Content": {
          "Type": "TextBlock",
          "Text": "Plant Overview",
          "FontSize": 24, "HorizontalAlignment": "Center"
        }
      },
      {
        "Row": 1, "Col": 0,
        "Cell": { "HeaderLink": "Tank 1 Level" },
        "Content": {
          "Type": "RadialGauge",
          "LinkedValue": "@Tag.Plant/Tank1/Level.Value",
          "Minimum": 0, "Maximum": 100
        }
      },
      {
        "Row": 1, "Col": 1,
        "Cell": { "HeaderLink": "Tank 1 Temp" },
        "Content": {
          "Type": "RadialGauge",
          "LinkedValue": "@Tag.Plant/Tank1/Temperature.Value",
          "Minimum": 0, "Maximum": 100
        }
      },
      {
        "Row": 1, "Col": 2,
        "Cell": { "HeaderLink": "Tank 2 Level" },
        "Content": {
          "Type": "RadialGauge",
          "LinkedValue": "@Tag.Plant/Tank2/Level.Value",
          "Minimum": 0, "Maximum": 100
        }
      },
      {
        "Row": 1, "Col": 3,
        "Cell": { "HeaderLink": "Tank 2 Temp" },
        "Content": {
          "Type": "RadialGauge",
          "LinkedValue": "@Tag.Plant/Tank2/Temperature.Value",
          "Minimum": 0, "Maximum": 100
        }
      },
      {
        "Row": 2, "Col": 0, "ColSpan": 4,
        "Cell": { "HeaderLink": "Level Trends" },
        "Content": {
          "Type": "TrendChart",
          "Duration": "1m",
          "Pens": {
            "Type": "TrendPenList",
            "Children": [
              { "Type": "TrendPen", "LinkedValue": "@Tag.Plant/Tank1/Level", "PenLabel": "T1 Level", "Stroke":
"#FF2196F3", "Auto": true },
              { "Type": "TrendPen", "LinkedValue": "@Tag.Plant/Tank2/Level", "PenLabel": "T2 Level", "Stroke":
"#FFFF9800", "Auto": true }
            ]
          }
        }
      }
    ]
  }]
}
```

CRITICAL: PanelType is REQUIRED. Omitting it silently defaults to Canvas, which breaks Dashboard grid positioning.

Note: Verify TrendChart Pens schema and Dashboard Cell structure against `list_elements('TrendChart')` and `list_elements('Dashboard')` output — the exact property names are defined there.

Step 8: Start Runtime

`designer_action('start_runtime', solution_id=<id>)`

The Value Simulator starts feeding changing data. Historian begins logging. Alarms evaluate against the setpoints. The dashboard shows live values.

This step is essential — always start runtime at the end of a new solution build so the user immediately sees live data.

Verification

After starting runtime:

1. `get_solution_info(solution_id=<id>)` — confirm object counts match expectations
2. Navigate to AlarmsMonitor: `designer_action('navigate', 'AlarmsMonitor', solution_id=<id>)` — alarms should evaluate against limits when simulator values cross setpoints
3. Navigate to HistorianMonitor: `designer_action('navigate', 'HistorianMonitor', solution_id=<id>)` — confirm tag values are being logged

Scaling Up

The starter examples above are small (8 tags, 8 points, 6 historian tags, 6 alarms). When building larger solutions:

Batch size: Keep each `write_objects` call to a reasonable size — typically under 75 objects. For very large sets (200+ alarms, 300+ device points), split into logical groups by process area or equipment type. This keeps errors isolated and gives the user visible progress between batches. Avoid exceeding 150 objects in a single call.

Pillar boundaries still apply at scale. Even with 500 tags and 200 alarms, the sequence is the same: write all tags first, confirm success, then move to Devices, then Historian, then Alarms, then Displays. The temptation to "plan everything and write it all at once" grows with scale — resist it. Module-by-module builds catch cross-reference errors early and give the user visible progress.

Adapting for Real Protocols

To adapt this starter for a real device instead of Value Simulator:

1. In Step 4, replace `list_protocols('ValueSimulator')` with the target protocol (e.g., `list_protocols('ModbusTCP')`, `list_protocols('OPCUA')`)
2. Configure Channel/Node with real connection parameters (IP, port, etc.)
3. Map Points to real device addresses
4. Everything else (Tags, Historian, Alarms, Dashboard) remains the same

Common Pitfalls

Mistake	Why It Happens	How to Avoid
Guessing protocol field formats	Skipping `list_protocols`	Always call `list_protocols` before writing device tables
Bare tag names in DevicesPoints	Forgetting asset path	Always use full path: `Plant/Tank1/Level`
Creating unnecessary HistorianTables	Not checking defaults	Check `get_objects('HistorianHistorianTables', solution_id=<id>)` first
Creating unnecessary AlarmsGroups	Not checking defaults	Predefined groups exist (Critical, Warning, AuditTrail) — check `get_objects('AlarmsGroups', solution_id=<id>)` first
Omitting PanelType on dashboard	Seems optional	Always set PanelType explicitly
Blank first screen and blank Solution Center preview	Home content written, but the Startup layout's Content page left empty — or a new home page built and never wired into Content	Write the home screen into `MainPage`, or point the Startup `Content` region at your page — never leave Content empty or on a placeholder
Pre-fetching all schemas at once	Over-eager optimization	Fetch each module's schema only when ready to write it
Screenshotting to verify writes	Unnecessary self-validation	Trust `write_objects` success. User sees live updates
Forgetting to start runtime	Skill ends at config	Always end with `designer_action('start_runtime', solution_id=<id>)`

Call rejected with `MISSING_SOLUTION_ID`	Omitting the routing key returned by `create_solution` / `open_solution`	Pass `solution_id` on every session-bound call (`get_objects`, `write_objects`, `designer_action`, `get_solution_info`, ...)
Passing `template` to `open_solution`	Confusing opening with creating	`open_solution` has no `template` parameter — use `create_solution(name=..., template=...)` for a new solution
Using alarm field names from examples without checking schema	Examples may not match current schema	Always verify field names against `get_table_schema('AlarmsItems')`
Using simple variables for repeated equipment	Didn't plan UNS structure	Step 2: if 2+ identical equipment instances exist, use UserTypes
Cramming 200+ objects in one write_objects call	No batch size awareness	Keep under 75 objects per call; split large sets by process area. Avoid exceeding 150 in a single call
Fetching Pillar 2+ schemas during Pillar 1	Trying to plan ahead	Each pillar is a phase gate — complete current pillar before fetching schemas for the next

What's Next

- **Canvas displays, process diagrams, visual dynamics:** Load the **Display Construction** skill — `search_docs('displays', labels='skill')`
- **Scripts, expressions, automation:** Load the **Scripts and Expressions** skill — `search_docs('scripts', labels='skill')`
- **Real device connectivity:** Use `list_protocols()` to browse available protocols, then adapt Step 4
- **Asset navigation pattern:** The Display Construction skill covers how to build asset-driven dynamic displays

See also

- Water Treatment Plant AI Designer Example. Worked end-to-end example — a complete Highland Municipal Water Treatment Plant solution generated by FrameworX AI Designer, with a downloadable solution and the build prompt to reproduce it.
- Skill Open Existing Solution. Open and assess an existing solution; pick the right workflow (build / maintain / audit / learn).
- Skill Display Construction. Router for Display authoring (Canvas, Dashboard, HMI symbols).
- Skill ML.NET. Add machine-learning models to a solution as Script Classes.
- Skill MQTT Integration. Connect to MQTT brokers (TagProvider or Device Module).