Skill Scripts Expressions

Server-side automation with Script Tasks and Expressions, reusable Script Classes, and client-side CodeBehind.

What This Skill Does

Implement automation logic in FrameworX: server-side Script Tasks for procedural logic, Expressions for lightweight tag-to-tag calculations, reusable Script Classes for shared methods, and client-side CodeBehind for display event handling.

When to Use This Skill

Use when:

- User needs server-side automation (calculations, sequencing, event handling)
- User needs one-liner formulas on many tags (Expressions)
- User wants reusable code libraries (Classes)
- User needs display-level event handling (CodeBehind)
- User asks "how do I run code" or "how do I compute values"

Do NOT use when:

- User only needs device connectivity (use Devices module)
- User only needs alarm thresholds (use Alarms module — no code needed)
- User needs trend/chart displays (use Display Construction skill)

Decision Tree — Which Script Type?

```
Is the code triggered by a display event (button click, display opening)?
  YES  CodeBehind (client-side, embedded in the display)
  NO

Is it a one-liner on many tags (scaling, unit conversion, status mapping)?
  YES  Expression (server-side, one row per tag)
  NO

Is it procedural logic (sequences, loops, conditionals, multi-step)?
  YES  Task (server-side, full method body)

Do you need shared utility methods callable from Tasks, Expressions, or CodeBehind?
   Class (server-side or client-side, plain .NET/Python methods)
```

Architecture Summary

Type	Runs On	Document Object?	Table
Task	Server	YES — read-modify-write	ScriptsTasks
Class	Server or Client	YES — read-modify-write	ScriptsClasses
Expression	Server	NO — simple merge	ScriptsExpressions
CodeBehind	Client	Embedded in DisplaysList	(part of display content)

Key distinction: Tasks and Classes are **document objects** — they contain code as structured content. You must read the existing object, modify it, and write the full object back. Sending partial content replaces the entire document.

Expressions are **simple objects** — send only the fields you want to set or change.

MCP Tools and Tables

Category	Items
Tools	`get_table_schema`, `write_objects`, `get_objects`
Tables	`ScriptsTasks`, `ScriptsClasses`, `ScriptsExpressions`, `ScriptsReferences`

Implementation Guide

Script Tasks

Step 1: Fetch Schema

```
get_table_schema('ScriptsTasks')
```

Step 2: Understand Task Structure

A Task is a method body that executes on triggers. Key properties:

- **ObjectName**: Task identifier (e.g., "CalculateEfficiency")
- **Language**: "CSharp", "VB", or "Python"
- **Domain**: "Server" (default) or "Client"
- **Execution**: Trigger mode — "Startup", "Continuous", "WhileTrue", "OnTrue", "OnChange", or "Periodic"
- **TriggerTag**: Tag path for OnChange/OnTrue/WhileTrue triggers
- **Period**: Interval in milliseconds for Periodic execution
- **Code**: The method body (NOT a full class — just the code inside the method)

Step 3: Write a Task

```
{
  "table_type": "ScriptsTasks",
  "data": [{
    "ObjectName": "CalculateEfficiency",
    "Language": "CSharp",
    "Execution": "Periodic",
    "Period": 5000,
    "Code": "double input = @Tag.Plant/Pump1/InputPower.Value;\ndouble output = @Tag.Plant/Pump1/OutputFlow.Value;
\nif (input > 0)\n    @Tag.Plant/Pump1/Efficiency.Value = (output / input) * 100;\nelse\n    @Tag.Plant/Pump1
/Efficiency.Value = 0;"
  }]
}
```

Predefined Tasks

- **ServerStartup** — runs once when runtime starts. Use for initialization.
- **ServerShutdown** — runs once when runtime stops. Use for cleanup.

Both are pre-created in new solutions and MCP-labeled. Modify with read-modify-write.

Task Code Patterns

Tag access in C#:

```
// Read
double level = @Tag.Plant/Tank1/Level.Value;
bool isRunning = @Tag.Plant/Pump1/Running.Value;

// Write
@Tag.Plant/Tank1/SetPoint.Value = 75.0;
@Tag.Plant/Pump1/Running.Value = true;

// Quality and Timestamp
int quality = @Tag.Plant/Tank1/Level.Quality;
DateTime ts = @Tag.Plant/Tank1/Level.Timestamp;
```

Tag access in Python:

```
level = Tag.Plant_Tank1_Level.Value     # Python uses _ instead of /
Tag.Plant_Tank1_SetPoint.Value = 75.0
```

Calling a Class method:

```
double result = @Script.Class.MathUtils.Clamp(value, 0, 100);
```

Script Classes

Step 1: Fetch Schema

```
get_table_schema('ScriptsClasses')
```

Step 2: Understand Class Structure

A Class is a plain .NET or Python class with static methods. Key properties:

- **ObjectName**: Class identifier (e.g., "MathUtils")
- **Language**: "CSharp", "VB", or "Python"

- **Domain**: "Server" or "Client"
- **Code**: Full class body (methods, fields, properties)

Step 3: Write a Class

```
{
  "table_type": "ScriptsClasses",
  "data": [{
    "ObjectName": "MathUtils",
    "Language": "CSharp",
    "Domain": "Server",
    "Code": "public static double Clamp(double value, double min, double max)\n{\n    if (value < min) return min;
\n    if (value > max) return max;\n    return value;\n}\n\npublic static double Scale(double raw, double rawMin,
double rawMax, double engMin, double engMax)\n{\n    return engMin + (raw - rawMin) * (engMax - engMin) / (rawMax
- rawMin);\n}"
  }]
}
```

Runtime Access

- Server-domain: `@Script.Class.MathUtils.Clamp(value, 0, 100)`
- Client-domain: accessed from CodeBehind and client-side scripts
- Cross-language: C# classes can call Python classes and vice versa

Script Expressions

Step 1: Fetch Schema

```
get_table_schema('ScriptsExpressions')
```

Step 2: Understand Expression Structure

An Expression is a one-line server-side calculation bound to a tag. Key properties:

- **ObjectName**: Target tag with namespace prefix — `Tag.Path/Name` (REQUIRED format)
- **Expression**: The formula or method call
- **Execution**: "OnChange" or "Periodic"
- **Period**: Interval in milliseconds (for Periodic)
- **TriggerTag**: Tag to watch (for OnChange — defaults to the ObjectName tag)

Step 3: Write Expressions

Use Expressions when many tags each need a simple computation — avoids creating a full Task document per tag.

```
{
  "table_type": "ScriptsExpressions",
  "data": [
    {
      "ObjectName": "Tag.Plant/Tank1/LevelPercent",
      "Expression": "@Tag.Plant/Tank1/LevelRaw * 100 / 4095",
      "Execution": "OnChange"
    },
    {
      "ObjectName": "Tag.Plant/Tank1/TempFahrenheit",
      "Expression": "@Tag.Plant/Tank1/TempCelsius * 9 / 5 + 32",
      "Execution": "OnChange"
    },
    {
      "ObjectName": "Tag.Plant/Tank1/Status",
      "Expression": "@Script.Class.StatusLogic.EvaluateStatus(@Tag.Plant/Tank1/Level, @Tag.Plant/Tank1/Pressure)",
      "Execution": "Periodic",
      "Period": 1000
    }
  ]
}
```

Critical: ObjectName must include the `Tag.` namespace prefix. The tag itself (in UnsTags) does NOT have the prefix — but the Expression binding DOES.

CodeBehind (Display-Embedded)

CodeBehind is NOT a separate table — it's embedded in the display content (DisplaysList). It runs client-side in the display's local runtime.

Lifecycle Events

- `DisplayOpening()` — runs once when the display loads
- `DisplayIsOpen()` — runs cyclically while the display is visible
- `DisplayClosing()` — runs once when the display closes

When to Use

- Button click handlers and operator UI events
- Local display state (variables that don't need server persistence)
- Client-side calculations specific to one display
- Animation triggers based on local interaction

How to Add CodeBehind

CodeBehind is part of the display's document content. When writing a display via `write_objects('DisplaysList', ...)`, include the CodeBehind in the content structure. Fetch the display first (document object — read-modify-write), then add or modify the CodeBehind section.

See the **Display Construction** skill for the full display writing workflow.

Script References

If your code needs external .NET assemblies:

```
get_table_schema('ScriptsReferences')
```

Add assembly references so Tasks and Classes can use additional libraries.

Common Pitfalls

Mistake	Why It Happens	How to Avoid
Sending partial Task code	Forget it's a document object	Always read-modify-write for Tasks and Classes
Expression ObjectName without `Tag.` prefix	Confusing tag path vs expression binding	Always `Tag.Path/Name` for Expression ObjectName
Using `@Label.` in server Tasks	Confusing with display symbol syntax	`@Label.` is only for symbol definitions. Use `@Tag.` in scripts
Python tag paths with `/`	Python doesn't support `/` in identifiers	Use `_` instead of `/` in Python tag access
CodeBehind accessing server objects	Assumes it runs on server	CodeBehind runs client-side. Use `@Tag.` for server tag access, but local variables are client-only
Creating a Task for each tag's scaling	Over-engineering	Use Expressions for bulk one-liner calculations
Forgetting hot_reload after Task changes	Code changes don't apply automatically	Call `designer_action('hot_reload')` or restart runtime

Patterns and Recipes

Pattern: Startup Initialization

Modify the predefined ServerStartup task:

```
get_objects('ScriptsTasks', names=['ServerStartup'])
```

Add initialization code, write back the full document.

Pattern: Bulk Scaling with Expressions

When 50 analog tags each need raw-to-engineering conversion:

1. Create all tags in UnsTags (raw + scaled pairs)
2. Write one Expression row per scaled tag
3. Each Expression is a simple formula — no Task overhead

Pattern: Class + Expression Combo

For complex logic applied to many tags:

1. Write the logic once in a Script Class method
2. Each Expression calls the Class method with its tag as argument
3. One code change in the Class updates all tag calculations

See also

- Skill C# Foundations. FX-specific C# patterns: @ namespaces, compile-subset rules, TK toolkit index, ListObj iteration. Load before writing Task or CodeBehind bodies.
- Skill Display Construction. Display authoring router — covers the full Display CodeBehind workflow.