Skill MQTT TagProvider

Connect FrameworX to an MQTT broker via Discovery Services, explore its namespace with the built-in browser, link data into the AssetTree, and build a dynamic asset-navigation UI with DataGrid, TrendChart, and AlarmViewer — all driven by Asset() bindings.

Prerequisite skill: Read Skill Discovery Services first if you don't understand the difference between Device Module, DataLink, and Dynamic Tags.

When to Use This Skill

Use when:

- The user wants to connect to an **MQTT broker** (external or built-in) using Discovery Services
- The data structure is **dynamic or unknown upfront** — the broker defines what exists
- The user mentions **MQTT TagProvider**, **SparkplugB**, **IIoT**, **broker connection**, or **topic discovery**
- The user wants an **asset-navigation UI** where selecting a tree node changes displayed data
- The application is monitoring/visualization focused

Do NOT use when:

- The user wants **explicit point-by-point control** over specific MQTT topics with alarms on each — use Device Module with MQTT protocol (Channel Node Points)
- The data source is **not MQTT** (for OPC UA Discovery Services, similar patterns apply but connection details differ — adapt from this skill)
- The user just needs **simulated data** — use ValueSimulator with standard Tags

Prerequisites

- Solution must be open (`open_solution` or `create_solution` completed)
- An MQTT broker must be accessible (local or remote)
- For testing: FrameworX has a **built-in MQTT broker** and a **SparkplugB simulator**

Implementation Steps

Step 0: Prepare the MQTT Broker (Testing or Production)

Option A: Built-in broker for testing/demo

Navigate to Data Explorer MQTT Tools and start the built-in services:

```
designer_action('navigate', 'DataExplorer.MQTTTools')
```

From MQTT Tools, start the built-in broker and simulator using the tab actions:

```
designer_action('builtin_broker', 'start')
designer_action('simulator', 'start')
```

This creates a local MQTT broker on `localhost:1883` and a SparkplugB simulator publishing sample data (solar panels, cities, sensors).

Option B: External production broker

Ask the user for:

- Broker address and port
- Authentication (username/password if needed)
- TLS requirements
- Whether the broker uses standard MQTT topics or SparkplugB

Step 1: Create the Discovery Services Connection

First, get the protocol schema to understand the PrimaryStation format:

```
get_table_schema('UnsTagProviders')
list_protocols('mqtt')
```

PrimaryStation format for MQTT — 9 semicolon-separated fields (MQTTspB is 15; always confirm by calling `list_protocols`):

```
{BrokerURL};{Port};{ClientID};{UserName};{Password};{NetworkSecurity};{X509Certificate};{QoS};{KeepAlive}
```

Common configurations:

Scenario	PrimaryStation
Local built-in broker	`localhost;1883;FrameworX-001;;;None;;AtLeastOnce;10`
Remote broker	`mqtt.factory.local;1883;FrameworX-001;;;None;;AtLeastOnce;10`
With authentication	`broker.cloud.io;8883;MyClient;admin;password123;TLSv1.2;;AtLeastOnce;30`

Write the connection:

```
{
  "table_type": "UnsTagProviders",
  "data": [{
    "Name": "MQTT",
    "Protocol": "MQTT",
    "PrimaryStation": "localhost;1883;FrameworX-001;;;None;;AtLeastOnce;10",
    "Separators": "BranchSeparator=/;AttributeSeparator=/;",
    "Description": "Production MQTT Broker"
  }]
}
```

For SparkplugB brokers, use `MQTTspB` protocol instead:

```
{
  "Name": "SparkplugBroker",
  "Protocol": "MQTTspB",
  "PrimaryStation": "localhost;1883;FrameworX-SpB;;;;;None;false;;AtLeastOnce;10;false;false;true",
  "Separators": "BranchSeparator=/;AttributeSeparator=/;",
  "Description": "SparkplugB broker with richer metadata"
}
```

Naming decision: The `Name` identifies the connection. A linked folder of the Asset Tree points into it with `TagProviderLink` set to `Name.("path")` (Step 2), and a tag's `DataLink` uses the same form. Asset() paths run from the root of the Asset Tree through the linked folder (`Asset("/Production/MQTTData/...")`), never from this name. Choose a meaningful name: `ProductionMQTT`, `FactoryBroker`, `EdgeGateway`, etc.

Separators: `BranchSeparator=/;AttributeSeparator=/;` maps MQTT topic hierarchy to UNS path hierarchy. This is almost always correct for MQTT.

Step 2: Link to AssetTree

Create an AssetTree folder that links to the Discovery Services connection. This makes the data navigable in the AssetsTree UI control.

```
get_table_schema('UnsAssetTree')
```

```
{
  "table_type": "UnsAssetTree",
  "data": [
    { "Name": "Production" },
    {
      "Name": "MQTTData",
      "Parent": "Production",
      "TagProviderLink": "MQTT.(\"spBv1.0/GroupID\")",
      "Description": "MQTT production data"
    }
  ]
}
```

Key fields:

- **Name**: the folder name. Users see it in the navigation tree. The folder's path in the Asset Tree is `Production/MQTTData`.
- **Parent**: the folder that holds the linked folder. Required: a link cannot sit at the root of the Asset Tree, and the import refuses a link row with no Parent. Write the parent folder row first, as above.
- **TagProviderLink**: `ProviderName.("path")`, the connection Name from Step 1 and the topic branch the folder starts at. Everything below that branch appears under the folder. `MQTT.("")` links the entire broker namespace. There is no InitialBranch column.

From this folder down, the tree is dynamic — no further configuration needed.

In displays and expressions, read a value under the folder with `Asset("/Production/MQTTData/<path below spBv1.0/GroupID>")`: the path runs from the root of the Asset Tree through the linked folder, never from the connection Name. In C# or VB.NET code, read the same path with `TK.Data.AssetValue`.

Step 3: Build the Layout

The standard asset-navigation pattern uses a Layout with three regions: Header (top), Menu (left side with AssetTree), and Content (main area).

```
get_table_schema('DisplaysLayouts')
list_elements('Layout')
```

Read the existing startup layout first to understand current structure:

```
get_objects('DisplaysLayouts', detail='full', names=['Startup'])
```

Write or modify the layout:

```
{
  "table_type": "DisplaysLayouts",
  "data": [{
    "Name": "Startup",
    "Description": "Asset navigation — tree on left, dynamic content on right",
    "Members": [
      {"Region": "Header", "Page": "Display.Header"},
      {"Region": "Menu",   "Page": "Display.AssetNav"},
      {"Region": "Content", "Page": "Display.MainPage"}
    ]
  }]
}
```

Note: `Startup` (layout ID 0) is the predefined startup layout — it loads on application launch. Read it first before modifying to preserve any existing configuration.

Step 4: Create the Header Display

A simple Canvas display with the application title and optional navigation buttons:

```
get_table_schema('DisplaysList')
list_elements('TextBlock,IndustrialIcon')
```

```
{
  "table_type": "DisplaysList",
  "data": [{
    "Type": "Display",
    "Name": "Header",
    "PanelType": "Canvas",
    "DisplayMode": "Page",
    "Width": 1200,
    "Height": 60,
    "Description": "Application header bar",
    "Elements": [
      {
        "Type": "TextBlock",
        "Name": "Title",
        "Left": 20,
        "Top": 15,
        "Text": "MQTT Production Monitor",
        "FontSize": 18,
        "FontWeight": "Bold"
      }
    ]
  }]
}
```

Use themed colors from `list_elements('ThemeColors')` instead of hardcoded hex values.

Step 5: Create the AssetTree Navigation Display

A Canvas display containing the AssetsTree control. When users click nodes, the platform updates `Client.Context` properties automatically.

```
list_elements('AssetsTree')
```

```
{
  "table_type": "DisplaysList",
  "data": [{
    "Type": "Display",
    "Name": "AssetNav",
    "PanelType": "Canvas",
    "DisplayMode": "Page",
    "Width": 300,
    "Height": 700,
    "Description": "Asset tree navigation panel",
    "Elements": [
      {
        "Type": "AssetsTree",
        "Name": "NavTree",
        "Left": 0,
        "Top": 0,
        "Width": 300,
        "Height": 700
      }
    ]
  }]
}
```

What happens when the user clicks a node:

Property	Description	Example value
`Client.Context.AssetPath`	Path of the selected node, ending with the branch separator	`Production/MQTTData/Barcelona/Panel1/`
`Client.Context.AssetName`	Full name of the selected node	`Production/MQTTData/Barcelona/Panel1`
`Client.Context.AssetNodeName`	Leaf node name	`Panel1`

These properties drive the content display reactively — all elements bound to them update automatically.

Step 6: Create the MainPage (Dynamic Content)

The MainPage binds to Client.Context values to react to the user's tree selection. One display serves the entire asset hierarchy.

```
list_elements('DataGrid,TrendChart,TextBlock,AlarmViewer')
```

Dashboard layout approach:

```
{
  "table_type": "DisplaysList",
  "data": [{
    "Type": "Display",
    "Name": "MainPage",
    "PanelType": "Dashboard",
    "Columns": 2,
    "Rows": 3,
    "Description": "Dynamic content — reacts to AssetTree selection",
    "Elements": [
      {
        "Type": "TextBlock",
        "Name": "AssetTitle",
        "Row": 0,
        "Column": 0,
        "ColumnSpan": 2,
        "Text": "",
        "LinkedValue": "@Client.Context.AssetName",
        "FontSize": 20,
        "FontWeight": "Bold"
      },
      {
        "Type": "DataGrid",
        "Name": "AssetData",
        "Row": 1,
        "Column": 0,
        "ColumnSpan": 2,
        "Height": 300
      },
      {
        "Type": "TrendChart",
        "Name": "AssetTrend",
        "Row": 2,
        "Column": 0,
        "Height": 250,
        "Duration": "00:05:00"
      },
      {
        "Type": "AlarmViewer",
        "Name": "AssetAlarms",
        "Row": 2,
        "Column": 1,
        "Height": 250
      }
    ]
  }]
}
```

The key pattern — DataGrid bound to context:

The DataGrid displays all children (sub-tags and attributes) of whatever the user selected in the AssetTree. Check `list_elements('DataGrid')` for the exact property names to bind the DataGrid to `Client.Context.AssetPath`. The DataGrid automatically shows all tags below the selected path.

CodeBehind (display events):

```
Public Sub DisplayOpening()
    ' Optionally start the tree at the folder that holds the linked folder (an Asset Tree folder path)
    @Client.Context.TreeInitialBranch = "Production"
End Sub
```

Step 7: Start Runtime and Verify

```
designer_action('start_runtime')
```

After starting:

1. The Discovery Services connection contacts the MQTT broker and discovers topics
2. The AssetTree populates with the linked folder structure
3. Clicking any node updates `Client.Context` properties
4. DataGrid, TrendChart, and other elements react to the selection

Take a screenshot to verify:

```
get_screenshot(target='runtime')
```

MQTT vs SparkplugB — Which Protocol?

Aspect	MQTT (standard)	MQTTspB (SparkplugB)
Protocol name in FrameworX	`MQTT`	`MQTTspB`
Topic structure	Flat or custom hierarchy	`spBv1.0/GroupID/NBIRTH` (fixed SparkplugB convention)
Metadata	Topic names only	Rich: data types, birth/death certificates, metrics
Discovery quality	Basic (topic tree)	Full (typed variables with descriptions)
When to use	Generic MQTT brokers, custom topic schemes	Industrial IoT with SparkplugB-compliant edge nodes

If the broker publishes SparkplugB messages, always prefer `MQTTspB` — you get typed variables and richer metadata.

Variations

Variation A: Built-in Broker as Data Hub

Use FrameworX's built-in MQTT broker as a central data hub. External devices publish to it, and the Discovery Services connection reads from it — all on `localhost`.

```
{
  "Name": "LocalHub",
  "Protocol": "MQTT",
  "PrimaryStation": "localhost;1883;FrameworX-Hub;;;None;;AtLeastOnce;10",
  "Separators": "BranchSeparator=/;AttributeSeparator=/;",
  "Description": "Built-in broker as central data hub"
}
```

Start the built-in broker from Runtime Startup or via `designer_action('builtin_broker', 'start')`.

Variation B: Multiple Brokers

Create multiple Discovery Services connections, each with a different Name:

```
{
  "table_type": "UnsTagProviders",
  "data": [
    {"Name": "FactoryA", "Protocol": "MQTT", "PrimaryStation": "mqtt-a.factory.com;1883;..."},
    {"Name": "FactoryB", "Protocol": "MQTT", "PrimaryStation": "mqtt-b.factory.com;1883;..."}
  ]
}
```

Create separate AssetTree folders for each. The same Layout and MainPage pattern works — `Client.Context` handles any source.

Variation C: Hybrid — Dynamic Browsing + DataLink for Critical Points

Use Discovery Services for browsing the full MQTT namespace, but create local tags with DataLink for the few points that need alarms and historian:

```
{
  "table_type": "UnsTags",
  "data": [
    {
      "Name": "Critical/ReactorTemp",
      "Type": "Double",
      "DataLink": "MQTT.(\"plant/reactor/temperature\")",
      "Min": 0,
      "Max": 500,
      "Description": "Reactor temperature — alarmed and historized"
    }
  ]
}
```

Then configure alarms and historian on `Critical/ReactorTemp` as a normal tag. The DataLink binds it to the MQTT topic automatically. DataLink is written `ProviderName.("path")`; `/Provider/path` and `Provider/path` do not link.

Variation D: Add Historian Queries to TrendCharts

If connecting to a historian-capable Discovery Services connection (Canary, InfluxDB, etc.), TrendCharts using Asset() syntax query historical data for the selected time range. The connection row has no IsHistorian column; the protocol driver declares the capability.

Common Pitfalls

Mistake	Why it happens	How to avoid
Creating local UnsTags for MQTT data	Habit from Device Module workflow	Dynamic tags don't need local tags. Only use DataLink for points needing alarms/historian.
Wrong PrimaryStation format	Forgetting semicolons or field order	Always call `list_protocols('mqtt')` first — the format is 9 semicolon-separated fields (15 for MQTTspB).
Topics with special chars not accessible	MQTT topics like `spBv1.0/GroupID` contain dots	Keep the topic inside a quoted string: `MQTT.("spBv1.0/GroupID")` in TagProviderLink, and `Asset("/Production/MQTTData/Barcelona/Panel1")` in displays and expressions. The path runs from the root of the Asset Tree through the linked folder, not from the connection Name.
AssetTree empty after runtime start	Connection not established or linked folder misconfigured	Check Services Monitor for connection status. Verify that TagProviderLink is `<connection Name>.("path")` and that the linked folder has a Parent folder.
Using MQTT protocol for SparkplugB broker	SparkplugB has specific message format	Use `MQTTspB` protocol for SparkplugB brokers — you get typed variables and metadata.
DataGrid shows nothing	Not bound to Client.Context properly	Check `list_elements('DataGrid')` for the correct binding properties.
Trying to set alarms on dynamic tags	No local tag exists for dynamic data	Create a local tag with DataLink for that specific path, then configure alarms on it.

Quick Reference — MCP Tool Calls

Task	Tool call
Get MQTT protocol format	`list_protocols('mqtt')`
Get SparkplugB format	`list_protocols('mqtt sparkplug')`
Create Discovery Services connection	`write_objects('UnsTagProviders', data=[...])`
Create AssetTree linked folder	`write_objects('UnsAssetTree', data=[...])`
Browse AssetsTree element schema	`list_elements('AssetsTree')`
Browse DataGrid schema	`list_elements('DataGrid')`
Navigate to MQTT Tools	`designer_action('navigate', 'DataExplorer.MQTTTools')`
Start built-in broker	`designer_action('builtin_broker', 'start')`
Start built-in simulator	`designer_action('simulator', 'start')`
Check runtime state	`get_state(target='runtime')`
Navigate to Services Monitor	`designer_action('navigate', 'ServicesMonitor')`

See also

- Skill MQTT Integration. Router skill: choose TagProvider vs Device Module before loading detailed steps.
- Skill MQTT Device Module. Static-tag path for alarms, historian, and scripting on MQTT data.
- Skill Discovery Services. Conceptual foundation: three connection patterns, DataLink, Asset() syntax.
- Skill Display Construction. General display building patterns (Canvas/Dashboard, themed colors, element placement).
- TagProvider Services. Concept overview.
- UNS TagProvider Services Reference. All properties.
- MQTT Integration Reference. Device Module vs Discovery Services comparison.
- Sorba AI Analytics Connector. Module Extension example.