AI Designer Connector

Enable AI-powered solution configuration through Model Context Protocol integration.

- Name: DesignerMCP, ConsoleMCP
- Version: 0.9.0.0
- Interface: TCP/IP
- Configuration: native

Generally Available in FrameworX 10.1.5

Shipped GA in FrameworX 10.1.5. Production-ready.

Set up the AI connection first

Before Claude (or another AI client) can use the tools on this page, the AI Designer needs a one-time setup: install the Claude Skill, install the FrameworX MCP bundle, and set the tool permissions.

Follow MCP and Claude Setup for the step-by-step guide. It covers the standard setup (Claude Desktop on the same machine as Designer), the setup for Claude on a different computer, and the GitHub Copilot and ChatGPT / Codex options.

Overview

The AI Designer connector enables AI models to interact with FrameworX Designer, providing intelligent assistance for solution configuration. Describe what you need in natural language, AI generates the configuration.

Two MCP servers are available:

- **DesignerMCP** — Live AI Designer. Claude connects to the running Designer and builds solutions with real-time visual feedback. Tools covering the full solution lifecycle.
- **ConsoleMCP** — File-based engineering via Claude Code. Generates JSON configuration files without a running Designer. Workspace-based equivalents of most tools, plus build. The engineer imports files into Designer for validation and deployment.

> *Note: This connector is for configuration-time operations (building solutions in Designer.exe). For querying live data from running solutions, see AI Runtime Connector*

Integration Architecture
Visual Indicator
Configuration
Available Tools
ConsoleMCP: Claude Code
Exchange Folder
Security
Table Types Quick Reference
Syntax Quick Reference
Common Workflows
Best Practices
Troubleshooting

Integration Architecture

AI Model		MCP Protocol		Solution Configuration
Claude Desktop, Claude.ai, Cursor, GitHub Copilot		Model Context Protocol		Designer.exe (live AI-driven workspace)
Claude Code (console tool)		Model Context Protocol		JSON files (ConsoleMCP)

Structured Methods

Manage Solutions	Read & Write Objects	Schema & Discovery	Designer Control	Documentation
Create, open, list solutions, get info & audit.	List, create, update, delete, rename objects.	Table schemas, display elements, protocol search.	Navigate, find, screenshot, runtime, security.	Search docs, fetch pages, inspect examples.

Overview

The AI Designer connector enables AI models to interact with FrameworX Designer, providing intelligent assistance for solution configuration. Describe what you need in natural language, AI generates the configuration.

Two MCP servers are available:

- **DesignerMCP** — Live AI Designer. Claude connects to the running Designer and builds solutions with real-time visual feedback. Tools covering the full solution lifecycle.
- **ConsoleMCP** — File-based engineering via Claude Code. Generates JSON configuration files without a running Designer. Workspace-based equivalents of most tools, plus build. The engineer imports files into Designer for validation and deployment.

Both share the same schema, documentation, and core object tools. ConsoleMCP simply operates on files instead of a live IDE, and excludes the Designer UI control tools.

> ***Note:*** *This connector is for configuration-time operations (building solutions in Designer.exe). For querying live data from running solutions, see AI Runtime Connector*

Visual Indicator

When AI is connected to Designer (DesignerMCP mode), you'll see:

- **"AI Designer" badge** — Orange label in the toolbar area
- **Orange border** — Glowing border around the main working area

This provides clear visual feedback that AI is actively controlling the Designer. ConsoleMCP has no visual indicator — it operates on files without a running Designer.

Configuration

For installation and connection setup (DesignerMCP, ConsoleMCP, HTTP/SSE mode, Claude Skill), see Claude Skill and MCP Setup.

Enable/Disable AI Designer

AI Designer is **Enabled** by default. To disable:

1. Open FrameworX Designer
2. Navigate to **Uns DataServers**
3. Uncheck **Allow MCP For Designer**

This setting is per-solution.

Available Tools

All tools are available in DesignerMCP. ConsoleMCP provides workspace-oriented equivalents — see ConsoleMCP: Claude Code below for what's different.

From FrameworX 10.1.6, this page lists the main tools without a tool count. The tool list in your AI client is the complete list for the installed version.

Tools marked ? are **DesignerMCP-only** — they require a running Designer IDE.

> **Most DesignerMCP tools take `solution_id`.** It routes the call to one of the solutions the Designer has open. Omitting it on a tool that requires it returns `MISSING_SOLUTION_ID`. Get the value from `list_solutions` or `open_solution`, then pass it on every subsequent call in the session. The session-entry tools (`list_solutions`, `open_solution`, `create_solution`) and the static schema and documentation tools (`get_table_schema`, `list_protocols`, `list_dynamics`, `list_elements`, `search_docs`, `get_solution_export_files`) do not require it. ConsoleMCP does not use `solution_id`; it addresses a workspace folder instead.
>
> From FrameworX 10.1.6, `get_build_info`, `check_product_update` and `delete_solution` do not require `solution_id` either.

Solution Management (5 tools)

Tool	Purpose
list_solutions	List available solutions and templates on this machine (works without Designer running)

create_solution	Create a new solution, launch Designer, and return the MCP Context document with security info
open_solution	Open an existing solution, launch Designer, and return the MCP Context document with security info
get_solution_info?	Get solution statistics — object counts per module, recent changes, audit trail
delete_solution	From FrameworX 10.1.6. DesignerMCP only. Delete a disposable test solution and its side files in one step. The tool accepts only solution names starting with `ai-test-` or `QA-TEST-` and refuses any other name with `SANDBOX_REQUIRED`, before it looks for the file. It refuses with `SOLUTION_IN_USE` while a Designer has the solution open or its runtime is running, and with `PROTECTED_ARTIFACT` for a file in the installation, Demos, SystemSolutions or Templates folders. It removes the `.dbsln`, the runtime side files beside it and the solution's backup copies. It also removes the solution's Designer working folder and pre-upgrade copy, and lists what it did not remove in `notReached`. Deleting a solution already gone returns `deleted=false`, not an error. For workspace folders, ConsoleMCP provides `delete_workspace`.

The solutions visible to AI Designer are only the ones in folders defined as Allow Remote Access, by the Solution Center tool. By default the solutions in the sub-folder ..\Documents\FrameworXSolutions are visible.

See Solution Center — Server Information

MCP Authorization

The first `open_solution`, `create_solution`, or `inspect_external_solution` call in a conversation triggers an MCP authorization prompt. Approve it once — subsequent calls need no further authorization.

Context Package on Repeat Calls

From FrameworX 10.1.6, the bootstrap tools accept a `context` parameter: `list_solutions`, `open_solution`, `create_solution` and `get_solution_export_files` in DesignerMCP, and `list_workspaces`, `open_workspace`, `create_workspace` and `get_solution_export_files` in ConsoleMCP.

Value	Returns
`full` (default)	The whole Context Package, as in 10.1.5.
`summary`	The connection contract only: `_mcpServer`, `version`, `environmentContext`, `build_state` when present and the learner-profile summary, plus `mode` and `note`.
`none`	A small `{mode, note}` marker.

Pass `summary` or `none` only on a repeat call, when the conversation already holds the full package. The first bootstrap call of an MCP server process always returns the full package, whatever the value. Unknown values fall back to `full`.

Object Operations (5 tools)

Tool	Purpose
get_objects	Read objects from config tables. `detail='summary'` for listing, `detail='full'` for complete JSON. Singletons always return full config.
browse_object_model	Browse runtime namespace paths (Server, Client, Tag, Alarm, Device, etc.)
write_objects	Create or update objects from JSON. Modes: `upsert` (default), `create`. Supports `dry_run` for validation. Multi-table writes handle dependency order automatically. The `data` parameter accepts three shapes: 1. JSON string literal (backwards compatible): `data='[{"Name":"Foo"}]'` 2. Parsed JSON array (single-table, no escaping): `data=[{"Name":"Foo"}]` 3. Parsed JSON object (multi-table): `data={"tables":[{"table_type":"UnsTags","data":[...]}]}` Shapes 2 and 3 are recommended for plan files (no JSON-inside-JSON escaping).
delete_objects	Delete objects by name. Backend prevents deletion of referenced objects — use `find_object` to inspect usage first.

rename_objects	Rename an object with safe refactoring — all cross-references update automatically (linked by ID, not name).

Singleton Tables: SolutionSettings, AlarmsGlobalSettings, RuntimeStartup, and RuntimeExecutionProfiles have a single configuration row — no Name column needed. Use `get_objects` to read, `write_objects(solution_id=<id>, mode='upsert')` to modify.

Schema & Discovery (4 tools)

Tool	Purpose
get_table_schema	Get field definitions for any table type. No parameter lists all table types. For the four protocol tables (UnsTagProviders, DevicesChannels, DevicesNodes, DevicesPoints), `expand_protocols=true` inlines the protocol schemas under `protocolUnions`, one entry per installed driver. On a full install the response is large. From FrameworX 10.1.6, add `protocols='Modbus,OPCUA'` to limit the expansion to the named drivers. Names without a matching installed driver come back in `requestedNotFound`. A `protocolsNote` explains when the parameters had no effect, for example on another table or in a batch call.
list_protocols	Search and discover communication protocols. Fuzzy matching by vendor name. Auto-includes protocol schema when search returns exactly one match.
list_elements	Get display element properties. No parameter lists all element types by category. Query by type: `list_elements('Canvas')`, `list_elements('Dashboard')`, `list_elements('WizardSymbol')`.
list_dynamics	Browse dynamic behaviors (FillColor, Visibility, Action, Rotate, etc.) by category or search. Returns property schemas and ready-to-use JSON patterns.

Protocol search examples:

- `list_protocols(search='siemens')` S7, S7Plus
- `list_protocols(search='allen')` EtherNet/IP, DF1, ControlLogix
- `list_protocols(search='modbus')` Modbus TCP, Modbus RTU

Display element queries:

- `list_elements()` — List all element types by category
- `list_elements('Canvas')` — Canvas display structure
- `list_elements('Dashboard')` — Dashboard grid and cell structure
- `list_elements('WizardSymbol')` — Industrial symbols: BLOWER, MOTOR, PUMP, TANK, VALVE

Dynamic behavior queries:

- `list_dynamics()` — Overview of all dynamics grouped by category
- `list_dynamics('color')` — FillColor, LineColor, TextColor dynamics
- `list_dynamics('action')` — ActionDynamic for click handlers
- `list_dynamics('animation')` — Rotate, Move, Scale, Skew dynamics
- `list_dynamics('FillColorDynamic')` — Full schema with property definitions

Designer UI & Control (3 tools)

Tool	Purpose
get_state	Get current Designer or Runtime state as lightweight text data. Active page, selected object, compilation errors, runtime health. From FrameworX 10.1.6, `get_state(target='designer')` also returns: • `documentState`: unsaved-change flags of the last display, script and report form loaded in the Designer editors. `display` carries `openName`, `flagChanged`, `flagDrawChanged`, `flagCodeChanged`, `disableFlagDrawChange` and `isReadOnly`. `script` and `reportForm` carry `openName` and `flagChanged`. Each entry stays after you navigate away, so compare `openName` before you read the flags. An entry is absent until the first document of its kind loads. • `gridSelection`: the selected rows of the active configuration grid, as `count` and `rowKeys`. Absent when the active page has no grid. • `inlineEditor`: present only while an inline cell editor waits for input, with its column and row. Close the editor with `designer_action('commit_edit')` or `designer_action('cancel_edit')`.
get_screenshot	Capture a screenshot of the Designer UI, Runtime view, or display/symbol previews. Saves image to the Exchange/Screenshots/ folder with a descriptive filename. AI reads the image via the filesystem MCP server.
designer_action	Perform Designer commands: navigation, find, runtime control and security. From FrameworX 10.1.6, also configuration-grid clipboard actions. The table below lists the main actions. An unrecognized action returns the list of valid actions.

designer_action available actions:

Action	Purpose	Options
`navigate`	Go to any module, table, or object	Target path: `'DataExplorer.MQTTTools'`, `'Tag.Folder1/Level'`, `'Display.MainPage'`
`find_object`	Find where an object is used — opens Find Results panel	Object name: `'Tag.Folder1/Level'`
`find_next`	Step through Find Results — Designer navigates to each location	(none)
`start_runtime`	Start solution runtime	(none)
`stop_runtime`	Stop runtime	(none)
`hot_reload`	Push design changes to running runtime without restart	(none)
`logon`	Authenticate a user for secured solutions	`'username:password'`
`logoff`	Return to Guest (anonymous) user	(none)
`select_row`	Select a specific row in a config grid by name and scroll it into view. Visual only, no mutation.	`'TableType.ObjectName'` (e.g., `'UnsTags.Plant1/Temperature'`)
`scroll`	Alias of `select_row`. Same behavior.	`'TableType.ObjectName'`
`expand`	Expand a tree node in the AssetTree, select it, and scroll into view. V1 supports the AssetTree only.	`'TreeName.path/segments'` (e.g., `'AssetTree.Plant1/Motor1'`)
`select_all_rows`	From FrameworX 10.1.6. Select every row of the active configuration grid. Returns the selected count.	(none)
`copy_rows`	From FrameworX 10.1.6. Copy the selected rows and return them in the response, in the grid's own copy format. The tool does not use the Windows clipboard. It empties password and secret columns and names them in `credentialColumnsRedacted`.	(none)
`paste_rows`	From FrameworX 10.1.6. Paste a payload through the grid's own paste. By default it overwrites the selected rows in order and adds extra lines as new rows. With `insert=true`, every line becomes a new row. It never writes password or secret columns, and `credentialColumnsIgnored` names the skipped columns. The response lists `overwrittenRows` and `createdRowKeys`. For a payload from `copy_rows`, pass `escaped` equal to its `escapedFormat` value. On a page read-only for the signed-in user, it returns `PASTE_NOT_ALLOWED`.	Payload text, or `{"payload":"...","insert":bool,"escaped":bool}`
`commit_edit` / `cancel_edit`	From FrameworX 10.1.6. Commit or discard the open inline cell editor. Returns `NO_INLINE_EDITOR` when no editor is open.	(none)

Note: `get_objects` and `write_objects` auto-navigate the Designer UI to the relevant context. Use `navigate` only for pages without configuration objects (DataExplorer tools, AlarmsMonitor, etc.).

Page-specific actions: Some Designer pages expose additional context-specific actions (e.g., expand_all, collapse_all on table views). These appear in `get_state` responses as `tabActions` and can be passed directly to `designer_action`.

Display Clipboard (2 tools)

Available in FrameworX 10.1.5 and later.

Tool	Purpose

display_clipboard_export	Copy the active Display's elements through the Designer's real copy pipeline and return the clipboard XAML payload to the AI — the Windows clipboard is not used. Image content and symbol definitions ride inside the payload. `scope='all'` (default) copies every element; `scope='selection'` copies the current canvas selection.
display_clipboard_import	Paste a payload produced by `display_clipboard_export` into the active Display through the Designer's real paste pipeline — images are created, reused, or auto-renamed in the target solution, and symbols are re-materialized. `on_symbol_conflict` chooses `auto_rename` (default) or `replace` when the target already has a same-named symbol with different content. Optional `at='x,y'` sets the paste position; when omitted, the content keeps its original position.

Cross-solution copy/paste in one session: open the source solution navigate to the source Display `display_clipboard_export` open the target solution navigate to a target Display `display_clipboard_import`. No second Designer instance and no Windows clipboard involved.

Product and Build Information (2 tools)

Tool	Purpose
get_build_info	Report the installed product build: `productVersion` and `buildNumber`. No open solution needed. From FrameworX 10.1.6, `detail='identity'` returns only the product identity. Available in DesignerMCP and ConsoleMCP. From FrameworX 10.1.6, the response also carries: • `resolvedPaths`: the folders this MCP server reads from, as `binaries`, `protocols` and `root`, plus `dotnetRuntime`, the .NET runtime of the server. Use `binaries` and `protocols` to check whether a driver or provider sits where the product looks for it. Two MCP servers on one install report different paths when they load from different folders. • `hostDesigners`: the Designer instances this MCP server started, each with `solutionName`, `solutionDir`, `pid`, `startTimeUtc` and `ownedByThisHost`. A row with `ownedByThisHost=false` is a stale record of a process no longer running. Each row also carries `alive`, and the list carries `runningCount`, `exitedCount` and `unknownCount`. ConsoleMCP never starts a Designer, so its list is always empty. • `components`: build identity per product component (`library`, `kernel`, `toolkit`, `modules`, `designer`, `runtime`, `schemas`, `mcpHost`, `html5`, `protocols`). Each file row lists `found`, `fileVersion`, `assemblyVersion`, `lastWriteUtc` and `sizeBytes`. The `html5` and `protocols` items also report `fileCount` and `newestWriteUtc` for their folder.
check_product_update	From FrameworX 10.1.6. Run the same check as **Help > Check for Latest Update** in the Designer and return the result. `status` is `Latest`, `NewerAvailable` (with `latestPublished` and `downloadUrl`) or `Unknown` when the Release Notes page is unreachable. A pre-release build reports `Latest` with `installedIsPreRelease=true`. Read-only: no download and no install. Available in DesignerMCP and ConsoleMCP.

Busy replies. From FrameworX 10.1.6, a `DESIGNER_BUSY` reply names the internal request the Designer is running and how long it has run, so you tell a long action from a stuck one. The request name is an internal name, not an MCP tool name.

Tag Importers (2 tools)

From FrameworX 10.1.6. DesignerMCP only.

Tool	Purpose
list_importers	List the importer plug-ins registered in this Designer installation, in the same order as the Designer's Import Tags page. Each entry carries `title`, `description`, `dll`, `kind` and `headless`. `headless=true` marks the importers `run_importer` runs. Read-only.
run_importer	Run one importer against a file, with no file picker and no wizard. Pass `importer` (the title from `list_importers`) and `file_path` (an absolute path on the Designer machine). Only importers marked `headless=true` run this way; the others return `IMPORTER_NOT_HEADLESS`. The tool runs only on solutions whose name starts with `ai-test-` or `QA-TEST-` and returns `SANDBOX_REQUIRED` on any other solution. The response carries `rowsAdded`, the number of rows the import added per table, measured by the Designer.

Script Debugger (1 tool)

From FrameworX 10.1.6. DesignerMCP only. **script_debug** drives the Designer's script debugger. Select the operation with `action`:

Action	Purpose
`status`	Read the debugger session state without changing it.
`attach` / `detach`	Attach to or detach from the debug target given by `domain`: `server` (default, script tasks and classes) or `client` (Display code-behind, needs a running client).
`set_breakpoint`	Set a breakpoint at `line` of `source` (`Task.<name>`, `Class.<name>` or `Display.<name>`). Returns `breakpointId` and `resolvedLine`.

`clear_breakpoint`	Remove a breakpoint of `source` by `breakpoint_id` or by `line`.
`clear_all_breakpoints`	Remove all breakpoints of `source`, or every breakpoint when `source` is omitted.
`list_breakpoints`	List the breakpoints of `source`, with `bound`, `pendingAttach` and the failure code of a breakpoint which failed to bind.
`pause`	Break into the running target.
`continue`	Resume, then wait up to `wait_ms` (default 5000, maximum 15000) for the next stop.
`step_over` / `step_into` / `step_out`	Step, then wait for the new stop.
`read_variable`	Read the value and type of `variable` at the current stop. Accepts a local name or member access such as `row.Status`, not arithmetic expressions.
`read_watch`	Read the entries of the Designer watch list, as evaluated at the last stop.
`set_stop_on_exception`	Turn stop-on-exception on or off with `stop_on_exception`.

Every response carries the session state: `attached`, `stopped`, `location` and `callStack`. A continue or step reply says whether the debugger took the command (`commandConsumed`) and where the target stopped, or returns `timedOut=true` when no stop came within `wait_ms`. While the tool waits, other calls to the same Designer return `DESIGNER_BUSY`, so keep `wait_ms` short. While a step runs, the tool refuses a continue or another step with `DEBUGGER_COMMAND_PENDING`.

Order of operations. Breakpoints need debug information, and the build produces it only while a debugger is attached. Use this sequence: `script_debug('attach')`, build the solution, restart the runtime with `designer_action('stop_runtime')` and `designer_action('start_runtime')`, run `script_debug('attach')` again, then set breakpoints.

When `set_breakpoint` fails, it returns `BREAKPOINT_FAILED` with `breakpointFailureCode`: `-1` name not found, `-2` no debug information (never built), `-3` source newer than the build, `-4` the object has build errors.

A breakpoint set through the tool stays armed after a hit. When the script is open in an editor tab, the breakpoint appears in the margin at the editor's next breakpoint refresh. The tool reads values and does not change them. Windows (.NET 4.8) solutions are supported. Multiplatform (.NET 10) solutions are supported on Windows x64 only. On a Multiplatform (.NET 10) solution, server debugging returns `DEBUGGER_PLATFORM_UNSUPPORTED` when this Designer cannot debug the .NET 10 runtime, and the message gives the reason.

Documentation & Knowledge (2 tools)

Tool	Purpose
search_docs	Search FrameworX documentation with label/section filtering — or fetch full page content by URL. Supports skills, code snippets, examples, and tutorials.
get_solution_export_files	Open an external reference solution (local or from web) to inspect its full JSON configuration. Use to study and replicate complete implementation patterns. Creates a subfolder at Exchange named after the application containing solution json files. AI reads files via the filesystem MCP server. For a documentation example, the response also returns a local copy of the example `.dbsln` to open or run. *Available in FrameworX 10.1.6 and later.*

search_docs modes:

- **Search mode:** `search_docs('alarm configuration')` — returns titles and snippets
- **Fetch mode:** `search_docs(fetch_url='<url from results>')` — returns full page content with code examples

Documentation labels: concept, tutorial, how-to, example, reference, connector, code, control, use-case, skill

AI Skills: Use `search_docs('', labels='skill')` to discover available step-by-step implementation guides. Skills are multi-module playbooks that prevent common mistakes in complex configurations.

inspect_external_solution workflow:

1. `search_docs(query, labels='example')` — find documented solution examples
2. `get_solution_export_files(solution_path)` — list available JSON files
3. `get_solution_export_files(solution_path, file_name='Tags.json')` — read specific configuration

Local copy of a documentation example:

Available in FrameworX 10.1.6 and later.

When you pass the `dbslnUrl` of an example from `search_docs(labels='example')`, the file-list response (the call without `file_name`) also returns these fields:

Field	Meaning
`localDbslnPath`	Path of a local copy of the example `.dbsln`, saved in the Exchange folder (default `C:\Users\Public\Documents\FrameworX\Exchange`) under the example's readable name, for example `Alarm Areas Example.dbsln`.
`localDbslnCreatedThisCall`	`true` when this call created the copy. `false` when the copy was already in the Exchange folder. The tool reports an existing copy as is, including any changes made to it since it was created.
`localDbslnNote`	Appears instead of the two fields above when the tool fails to make the copy, for example after a network error. The response still returns the JSON files.

To use the copy, AI Designer opens it with `open_solution(<localDbslnPath>)`. AI Console runs it with `run_workspace (name=<localDbslnPath>)`.

- The tool never overwrites or refreshes the copy. To get a newer published version, delete the copy, any file whose name starts with the example's name, and the example's `exportFolder`, then call the tool again.
- The copy is the file as published. Designer upgrades it when it opens it.
- The path refers to the computer where the MCP server runs.

inspect_application workflow:

1. `get_solution_export_files()` — exports current solution to Exchange/<AppName>/ folder
2. AI uses the filesystem MCP server to list and read exported JSON files
3. Useful for solution analysis, documentation, comparison, and migration

---

ConsoleMCP: Claude Code

ConsoleMCP is the file-based companion for Claude Code. It provides workspace-oriented equivalents of most DesignerMCP tools, plus create_solution_file. They share the same schemas, documentation, and object operations, and write to JSON files instead of a live Designer.

For installation and configuration, see Claude Skill and MCP Setup — Option C.

What's Different

Tools that require a running Designer IDE and are not available in ConsoleMCP:

Excluded Tool	Why
get_solution_info	Queries live solution state (object counts, audit trail). ConsoleMCP provides **get_workspace_info** for the file-based equivalent.
rename_objects	Safe refactoring needs the live solution's cross-reference index. In ConsoleMCP, rename an object by editing its `Name` field with `write_objects`; cross-references resolve when the workspace is imported into Designer.
designer_action (includes get_state, get_screenshot, find_object, hot_reload, navigate, start/stop runtime, logon/logoff)	Controls the Designer UI
display_clipboard_export / display_clipboard_import	Run the live Display editor's copy/paste pipelines — no file-based equivalent.
delete_solution, **list_importers**, **run_importer**, **script_debug**	From FrameworX 10.1.6. These tools act on a live Designer or on a `.dbsln` file. For workspace folders, ConsoleMCP provides `delete_workspace`.

Behavioral differences in shared tools:

Tool	DesignerMCP	ConsoleMCP
create_solution / open_solution (DesignerMCP) **create_workspace / open_workspace** (ConsoleMCP)	Launches Designer.exe, opens solution in IDE. Both DesignerMCP tools accept an optional `from_workspace=<path>` parameter — when supplied, all `*.json` files in the workspace folder are imported into the solution after it is opened or created. The target `.dbsln` is auto-backed up first (3-file retention). Always-replace upsert semantics. The response includes a `workspaceImport` section with per-file results.	Creates or opens a workspace folder in `Documents/FrameworX/Exchange/`

list_solutions (DesignerMCP) **list_workspaces** (ConsoleMCP)	Scans Solution Center folders for .fxproj files	Scans Exchange folder for workspace folders
write_objects	Writes to live solution via named pipe, backend validates in real time	Writes JSON files following ExportFormat 1.2, validates against local schema
get_objects	Reads from live solution	Reads from JSON files in the solution folder
delete_objects	Deletes in live solution, backend prevents deletion of referenced objects	Removes from JSON files, cleans up empty files
browse_object_model	Browses the object model of the live solution open in Designer	Browses the same namespace roots from the static namespace cache plus the workspace JSON files. TagProvider paths return the provider's static member schema only, never live values — this host is file-based and has no runtime.

ConsoleMCP Tool Naming and Build

ConsoleMCP uses **workspace** vocabulary for its solution-management commands: `list_workspaces`, `open_workspace`, `create_workspace`, `get_workspace_info`. DesignerMCP keeps its `*_solution` names unchanged. ConsoleMCP also ships a dedicated `create_solution_file` tool:

Tool	Purpose
create_solution_file	Invoke `SolutionCreator.exe` headlessly to compile a workspace into a deployable `.dbsln`. Signature: `create_solution_file(solution_name, workspace=None, template=None, platform=None, output_folder=None, description=None)`. Defaults: `workspace` falls back to the active workspace folder; `template` auto-picks `HeaderLayout` when available, otherwise `blank`; `platform` auto-picks `Multiplatform` (.NET 10) when the .NET 10 runtime is deployed alongside the install, otherwise `Windows` (.NET Framework 4.8); `output_folder` is the standard Solutions folder. Returns `{success, solutionPath, exitCode, log, ...}`. Long builds (Display- or Script-heavy workspaces) return `{status:"building", buildPid, solutionPath, startedAt}` instead, meaning the `.dbsln` is not yet complete. Poll `verify_solution_file(solution_name, building_pid=<buildPid>)` until the status is no longer `building`.
delete_workspace	From FrameworX 10.1.6. Delete a disposable test workspace folder and everything inside it in one step. Signature: `delete_workspace(name)`, with a workspace name in the Exchange folder or an absolute folder path. The tool accepts only names starting with `ai-test-` or `QA-TEST-` and refuses any other name with `SANDBOX_REQUIRED`. It refuses with `WORKSPACE_IN_USE` while a runtime runs under the workspace name: call `stop_workspace` first. A `.dbsln` built from the workspace stays in place. Deleting a workspace already gone returns `deleted=false`, not an error.

Typical ConsoleMCP workflow: `open_workspace` (or `create_workspace`) `write_objects` to generate JSON files `create_solution_file(solution_name, workspace=...)` to produce the deployable `.dbsln`.

ExportFormat 1.2 File Structure

Each JSON file generated by ConsoleMCP follows this envelope:

```
{
  "ObjectIdentification": {
    "TableType": "UnsTags",
    "SolutionName": "MySolution"
  },
  "ExportMetadata": {
    "ExportDate": "2026-02-21T14:30:00Z",
    "ExportUser": "ConsoleMCP",
    "ExportProduct": "FrameworX",
    "ExportFormat": "1.2"
  },
  "Data": [
    { "Name": "Tank1/Level", "Type": "Analog", "EngUnits": "%" },
    { "Name": "Tank1/Pump", "Type": "Digital" }
  ]
}
```

When to Use ConsoleMCP vs DesignerMCP

Scenario	Use
Claude Code on same machine as Designer — want live building in Designer	**DesignerMCP**
Claude Desktop or Claude.ai — want live building in Designer over the network	**DesignerMCPHttp**
Claude Code — generate configs without a running Designer	**ConsoleMCP**
Analyzing existing projects, auditing configurations	**ConsoleMCP**
Batch-building or generating solutions from specifications	**ConsoleMCP**

Both servers can be installed side by side. They register as separate MCP servers and do not conflict.

Exchange Folder

Several AI Designer tools use the **Exchange folder** to share files between the Designer and AI, rather than embedding large data in MCP responses. The filesystem MCP server (configured during setup) gives AI direct read access to these files.

Subfolder	Tool	Contents
`Exchange/<SolutionName>_json/`	`get_solution_export_files`	Solution JSON files from Export (Tags.json, Channels.json, Displays.json, etc.)
`Exchange/<ExampleName>.dbsln`	`get_solution_export_files`	Local copy of a documentation example, reported as `localDbslnPath` (10.1.6 and later)
`Exchange/Screenshots/`	`get_screenshot`	Captured images with descriptive filenames (e.g., Screenshot_MainPage_2026-02-15_14-30-00.jpg)
`Exchange/<SolutionName>-json/`	ConsoleMCP	JSON configuration files generated by Claude Code

This file-based approach keeps MCP responses lightweight and avoids large base64-encoded data in the conversation context.

Security

MCP Category and Update Protection

Objects created by AI receive `Category = "MCP"` (via SolutionCategories) to track AI-created vs manually-created.

Object Category	What AI Can Update
Contains "MCP"	All fields (full replacement)
No "MCP" (for example, created in Designer)	Description field only

How it works:

1. AI creates object Category set to "MCP"
2. AI updates Limited to Description field
3. To re-enable Manually add "MCP" to Category

> ***Note:*** *MCP Category protection applies to DesignerMCP only. ConsoleMCP writes plain JSON files — ownership is managed when the engineer imports into Designer.*

Solution Authentication

Solutions can have security enabled with user accounts and edit permissions.

How authentication works with AI (DesignerMCP):

1. AI opens solution response includes current username (`Guest` by default) and edit permissions
2. If permissions are not `Unrestricted`, AI informs the user that some operations may be restricted
3. User provides credentials AI calls `designer_action('logon', 'username:password')`
4. Successful login permissions expand to match the user's role
5. `designer_action('logoff')` returns to Guest

AI can browse and read the solution even as Guest. Permission restrictions only affect write operations.

Security: AI never echoes, logs, or repeats passwords in its responses.

> ***Note:*** *ConsoleMCP does not have authentication — it operates on JSON files in the Exchange folder. Security is enforced when the engineer imports files into a secured solution in Designer.*

Table Types Quick Reference

Module	TableType	Notes
UNS	UnsTags	Process tags organized in folder paths
	UnsUserTypes	UDT templates
	UnsTagProviders	External data sources
	UnsEnumerations	Enum mappings
Devices	DevicesChannels	Protocol configuration
	DevicesNodes	Device addresses
	DevicesPoints	Tag-to-address mapping
Alarms	AlarmsGroups	Behavior rules
	AlarmsItems	Tag-bound triggers
	AlarmsAreas	Optional hierarchy
	AlarmsGlobalSettings	Module configuration (singleton)
Historian	HistorianStorageLocations	Data repository
	HistorianHistorianTables	Storage coordination
	HistorianHistorianTags	Tags to log
Datasets	DatasetsDBs	Database connections
	DatasetsQueries	SQL queries
	DatasetsTables	Direct table access
	DatasetsFiles	File operations
Scripts	ScriptsTasks	Event-triggered code
	ScriptsClasses	Reusable libraries
	ScriptsExpressions	One-liner calculations
	ScriptsReferences	External DLL references
Reports	ReportsForms	Documents
	ReportsWebData	JSON/XML
Displays	DisplaysList	UI screens (Canvas or Dashboard)
	DisplaysSymbols	Reusable user-authored components
	DisplaysLayouts	Layout regions
	DisplaysImages	Image library
Security	SecurityUsers	Accounts
	SecurityPermissions	Permission groups
	SecurityPolicies	Session/password rules
	SecuritySecrets	Credentials (not accessible by AI)

Solution	SolutionCategories	Object labels (MCP flag)
	SolutionSettings	Global solution config (singleton)
Runtime	RuntimeStartup	Startup configuration (singleton)
	RuntimeExecutionProfiles	Dev/Prod replacement connections (singleton)

Syntax Quick Reference

Context	Syntax	Example
Scripts (C#, VB, Python)	@ prefix + .Value	`@Tag.Tank1/Level.Value`
Expressions	No prefix, no .Value	`Tag.Tank1/Level + 10`
Display bindings	@ prefix + .Value	`@Tag.Tank1/Level.Value`
String embedding	Curly braces with @ prefix inside	`"Level: {@Tag.Tank1/Level} %"`

Path Syntax

- Folder separator: `/` (slash) — only for Tags and Symbols: `Tag.Area1/Line1/Tank1`
- Namespace/member separator: `.` (dot) — for all other access: `Server.DateTimeInfo.Second`
- UDT members: `Tag.Area1/Line1/Loop1.Setpoint` (dot after tag path for member access)
- Built-in namespaces: dot only, never slash: `Alarm.Group.Critical.TotalCount`

Critical: Never use `.` for folders or `/` for members. `Tag.Area1.Line1.Tank1` is WRONG. `Tag.Area1/Line1/Loop1/Setpoint` is WRONG.

Common Workflows

- **Creating Tags** — "Create a Double tag called TankLevel in the Tanks folder with range 0-100"
- **Device Communication** — "Connect to a Siemens S7-1500 PLC at 192.168.1.10" AI will: Search protocols Present options Create Channel Create Node Help map Points
- **Creating Displays** — "Create a dashboard with 4 cells showing TankLevel, TankTemp, PumpStatus, and AlarmCount"
- **Configuring Alarms** — "Create high and low alarms for TankLevel: High at 90 (Critical), Low at 10 (Warning)"
- **Finding Object Usage** — "Where is tag Tank1Level used?" AI uses: `designer_action('find_object', 'Tag.Tank1Level', solution_id=<id>)` — opens Find Results panel in Designer showing all cross-references.
- **Deleting Objects Safely** — "Delete the old TestDisplay" AI uses: `designer_action('find_object', 'Display.TestDisplay', solution_id=<id>)` to check references first, then `delete_objects` if safe.
- **Renaming with Safe Refactoring** — "Rename Tag.OldName to Tag.NewName" AI uses: `rename_objects` — all cross-references update automatically (linked by ID, not name).
- **Applying Changes Without Restart** — "Apply my tag changes to the running system" AI uses: `designer_action('hot_reload', solution_id=<id>)`
- **Logging In to Secured Solutions** — "Log in as admin" AI prompts for password, then uses: `designer_action('logon', 'admin: password', solution_id=<id>)`
- **Inspecting Solution Configuration** — "Export and analyze the current solution" AI uses: `get_solution_export_files()` to export JSON files to the Exchange folder, then reads them via the filesystem server for analysis.

Best Practices

Be Specific

Instead of "Create some tags", say:

"Create these tags in the Production folder: MixerSpeed (Double, 0-1000 RPM), MixerRunning (Digital), BatchCount (Integer)"

Validate Before Bulk Operations

"Validate this configuration before creating 50 tags"

AI uses `write_objects` with `dry_run=true` to check for errors without committing.

Use Skills for Complex Tasks

"Search for a skill on Modbus TCP configuration"

AI searches for step-by-step guides that prevent common mistakes.

Review Changes

"Show me the recent changes" or "Navigate to the Tags tab"

Troubleshooting

AI Designer server not starting

- Verify .NET 8.0 runtime is installed
- Check that Designer is running
- Confirm MCP is enabled in Designer settings
- Check firewall settings

Claude doesn't see Designer tools

- Ensure claude_desktop_config.json path is correct (double backslashes)
- Restart Claude completely (close via Task Manager)
- Verify AI Designer shows "running" in Claude settings

Filesystem MCP server not connecting

- Verify Node.js is installed (`node --version` in command prompt)
- Check that the Exchange folder path in config matches your Windows username
- Verify "filesystem" shows "running" in Claude Desktop Settings Developer

"Update blocked" message

- Object doesn't have MCP in Category
- AI can only update Description field
- To enable: add "MCP" to Category in Designer

Changes not appearing in Designer

- Refresh Designer view (F5)
- For displays, close and reopen the display editor
- Verify operation completed in Claude's response

No visual indicator (orange border)

- Verify MCP connection is active in Claude Desktop
- Check Designer settings for MCP enabled
- Restart Designer if needed

Permission errors on write operations

- Solution may have security enabled
- Ask AI to check current permissions: look at the security field in the response
- Log in with appropriate credentials: tell AI "log in as [username]"
- After successful login, retry the operation

Connector process stopped or kept running unexpectedly

- From FrameworX 10.1.6, each AI Designer and AI Console connector process writes a small lifetime log, `McpLifecycle_<Connector>_<pid>.log`, in the product TraceLogs folder.
- The log records which launching process and grandparent the connector watches, a fallback to watching the launching process only, a watch it failed to start, and why it shut down after a watched process ended. A clean disconnect by the AI client records no exit reason.
- The connector writes each record at the moment of the decision, and the record survives a forced kill of the connector. Each file stops growing at 64 KB, and files older than 30 days are removed.
- The HTTP-hosted connectors do not watch a launching process.

ConsoleMCP: "No active workspace"

- Call `open_workspace` or `create_workspace` first

ConsoleMCP: "Control schemas not found"

- Ensure `ControlSchemas.json` is in the MCP folder alongside ConsoleMCP.exe

ConsoleMCP fails to launch

- Run `dotnet --list-runtimes` — install .NET 8.0 from dotnet.microsoft.com

Related Documentation

- AI Integration
- AI Designer In Action
- Claude Skill and MCP Setup — covers AI Designer, ConsoleMCP, and Claude Skill installation
- AI Runtime Connector
- AI ML Integration Connector

In this section...
