Skill Open Existing Solution

Open an existing FrameworX solution and choose the right workflow: continue building, maintain, audit, or learn.

Purpose

This skill guides AI sessions that open an existing FrameworX solution. Unlike new solution builds (which follow a fixed pillar sequence), existing solutions require understanding what's already there before taking action. The workflow depends on **why** the user is opening the solution.

Step 1: Open and Assess

`list_solutions  user picks  open_solution(name)`

Read the SolutionContext carefully. Then immediately:

`get_solution_info`

This returns module-level summary: which tables have objects, how many, last modified dates, and recent changes. This is your map of the solution.

> ***ConsoleMCP note:*** *In the ConsoleMCP (file-based) workflow, the equivalent commands use* ***workspace*** *vocabulary — `list_workspaces`, `open_workspace`, `get_workspace_info` — and `get_solution_info` is not available. Use `get_objects` on key tables to survey what exists, starting with `get_objects('UnsTags,AlarmsItems,DevicesChannels,DisplaysList')` for a quick overview.*

> ***DesignerMCP tip — `from_workspace`:*** *Both `open_solution` and `create_solution` 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 — useful when continuing development from a ConsoleMCP-authored workspace.*

Step 2: Identify the Workflow

There are four typical reasons a user opens an existing solution. If the user's intent is not clear from their request, ask:

> *"I can see this solution has [summary from get_solution_info]. Are you looking to: (A) continue building — add new modules or features, (B) maintain — fix or modify something specific, (C) audit — review and document what's here, or (D) learn — study how this solution works?"*

The workflow determines everything that follows.

Workflow A: Continue Development

The user has a partially built solution and wants to add new functionality.

Goal: Add new modules or features while respecting existing patterns.

Steps:

1. **Learn existing patterns first.** Before writing anything, run `get_objects` on the modules you'll be extending. Study:
   - Tag naming convention (ISA-95 paths? flat? folder structure?)
   - UserType patterns (are they using UDTs or simple variables?)
   - Existing alarm groups and naming
   - Display naming and PanelType choices
2. **Match what's there.** If existing tags use `Plant/Area1/Line1/Motor1`, your new tags must follow the same structure. If they use UDTs for equipment, create the same pattern for new equipment. Consistency matters more than your preference.
3. **Check MCP category.** New objects you create will be auto-flagged MCP. Existing objects may not be — you can read them but not modify without the user applying MCP category in Designer.
4. **Load skills for new modules.** If adding displays to a solution that had none: `search_docs(query='Skill Display Construction', labels='skill')`. If adding devices: `get_table_schema` + `list_protocols` as usual.
5. **Build normally — respect pillar boundaries.** Follow pillar order for new modules, but skip pillars that are already complete. If tags exist and you're adding alarms, start at Pillar 2. Even when extending an existing solution, complete the current pillar before fetching schemas for the next. Do NOT pre-fetch Display schemas while still writing Alarm objects.

6. **Batch size for large additions.** 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.

Common scenario: "Add a dashboard for the existing pump stations" — get_objects on UnsTags to find pump tag paths, then build displays that bind to those exact paths.

Workflow B: Maintenance

The user has a working solution and needs to fix, update, or modify something specific.

Goal: Surgical changes. Fix what's asked. Don't reorganize.

Steps:

1. **Understand the scope.** Ask what needs to change if not clear. Maintenance is targeted — "fix the alarm threshold on Tank3" not "redesign the alarm system."
2. **Find the objects.** Use `get_objects` with the target table type to locate specific objects. If the user refers to something by name, find it first:

   ```
   get_objects('AlarmsItems')   find Tank3 alarm
   get_objects('DisplaysList')  find the display to modify
   ```

3. **Read before write — always.** For document objects (Displays, Symbols, Scripts, Queries, Reports), read the full current content with `get_objects`, modify only what's needed, write back the complete object. This is critical — document objects use full replacement, not merge.
4. **MCP category matters here.** If the object the user wants to modify was manually created, it won't have MCP category. Your write will silently skip it ("Skipping existing" in response). Tell the user: "This object needs MCP category applied in Designer before I can modify it."
5. **Verify scope.** If a change might cascade (renaming a tag that alarms and displays reference), use `designer_action('find_object', name)` to check cross-references before modifying. Warn the user about impact.
6. **Don't over-deliver.** If the user asks to change one alarm threshold, change one alarm threshold. Don't offer to "also reorganize the alarm groups while we're here."

Common scenario: "The Modbus address for Pump2 pressure is wrong" — get_objects('DevicesPoints'), find the point, fix the Address field, write back.

Workflow C: Audit

The user wants to review, inspect, or document what's in the solution.

Goal: Report findings. Do not modify anything.

Steps:

1. **Start with the overview.** `get_solution_info` gives module-level summary with object counts and recent changes. Present this as the starting point.
2. **Dive into modules the user cares about.** Use `get_objects` on specific tables to list what exists:

   ```
   get_objects('UnsTags')          all tags and their structure
   get_objects('AlarmsItems')      all alarm configurations
   get_objects('DevicesChannels')  all device connections
   get_objects('DisplaysList')     all display pages
   ```

3. **Use audit tools.** Navigate to TrackChanges for detailed history:

   ```
   designer_action('navigate', 'RecentChanges')
   designer_action('navigate', 'CrossReference')
   designer_action('navigate', 'UnusedObjects')
   ```

4. **Check cross-references.** `designer_action('find_object', name)` shows where any object is used. Useful for understanding dependencies and finding orphaned objects.
5. **Screenshots for documentation.** This is a valid use of `get_screenshot` — capture displays, runtime views, or Designer pages for solution documentation.
6. **Summarize findings.** Organize by module. Report: what exists, what seems incomplete, what has no cross-references (potentially unused), and any patterns you notice (good or concerning).

Common scenario: "What's in this solution? Give me a summary" — full module-by-module audit with counts, patterns, and structure assessment.

Workflow D: Learn

The user wants to study a solution to understand FrameworX patterns — either their own older work, a colleague's solution, or a reference example.

Goal: Explain how and why things are built the way they are.

Steps:

1. **Start with the architecture.** `get_solution_info` then explain the high-level structure: which pillars are used, how many tags, what protocols, what types of displays.
2. **Trace the data flow.** This is the most valuable thing you can do for a learner. Pick a tag and trace it end-to-end:

   ```
   Tag: Plant/Line1/Motor1.Speed
     fed by: DevicesPoints (Modbus address 40001)
     watched by: AlarmsItems (HighLimit at 1800 RPM)
     logged by: HistorianTags (10-second interval)
     displayed on: MainDashboard (CircularGauge binding)
   ```

   Use `get_objects` on each module and `designer_action('find_object')` to build this picture.
3. **Explain design decisions.** When you see patterns, explain them:

   - "They used UserTypes for Motors — so all 12 motors share the same member structure. Change the UserType once, all instances update."
   - "The displays use Dashboard for monitoring and Canvas for the process diagram — that's the standard two-paradigm approach."
   - "Alarm groups are organized by severity (Critical, Warning) not by area — that's a common choice for smaller plants."
4. **Point to documentation.** When explaining a concept, link to the relevant docs: `search_docs(query='topic')` to find the right page.
5. **Use reference solutions for comparison.** If the user wants to see how something *should* be done:

   ```
   search_docs(query='example', labels='example')   find a documented example and its dbslnUrl
   get_solution_export_files(dbslnUrl)              study the example's JSON configuration
   open_solution(localDbslnPath)                    open the example in Designer (10.1.6 and later)
   ```

   Starting in FrameworX 10.1.6, the `get_solution_export_files` response for a `dbslnUrl` includes `localDbslnPath`, a local copy of the example `.dbsln` in the Exchange folder. Open the copy to explore the example in Designer. In AI Console, run it with `run_workspace (name=<localDbslnPath>)`. The tool never 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.
6. **Invite questions.** After the overview, ask what they want to understand deeper. Learning is conversational, not a monologue.

Common scenario: "I inherited this solution from a colleague, help me understand it" — full architecture walkthrough with data flow tracing and design decision explanations.

Quick Reference

Workflow	First Moves	Key Rule
Continue Dev	get_objects on related modules match patterns	Consistency with existing > your preference
Maintenance	get_objects on target read before write	Surgical. Don't over-deliver.
Audit	get_solution_info get_objects per module	Report. Don't modify.
Learn	get_solution_info trace data flows	Explain the why, not just the what.

Applies to All Workflows

- Always run `get_solution_info` first — it's your map. (DesignerMCP only; in ConsoleMCP, use `get_workspace_info` or survey key tables with `get_objects`.)
- Always `get_objects` before writing to any module — know what's there.
- MCP category controls what you can modify — if a write silently skips, that's why.
- Document objects (Displays, Symbols, Scripts, Queries, Reports) require full-replacement writes — read first, modify, write back.
- `designer_action('find_object')` is your best friend for understanding object relationships.
- Keep `write_objects` batches under 75 objects. Split large additions by process area or equipment type.

See also

- Water Treatment Plant AI Designer Example. Worked reference solution to study (Workflow D — Learn): a complete Highland Municipal Water Treatment Plant built by FrameworX AI Designer, with a downloadable solution and the build prompt that produced it.
- Skill New Solution. Create a complete starter solution from scratch (tags, simulator, historian, alarms, dashboard).
- Skill ML.NET. Add machine-learning models to an existing solution as Script Classes.
- Skill MQTT Integration. Add MQTT connectivity (TagProvider or Device Module) to an existing solution.