

MCP SDK Reference

Cross-tool conventions for the FrameworX MCP SDK — build-visibility propagation from the compile engine into every write-capable MCP tool response.

[AI Integration](#) MCP SDK Reference

FrameworX has always compiled user-authored .NET (Script Tasks, Script Classes, Script Expressions, and Displays code-behind) incrementally as objects are saved, and RuntimeBuildAndPublish in Designer has always produced a per-object error table. What is new in 10.1.5 is that this compile output is now propagated into MCP tool responses, so AI agents see the same build feedback an engineer sees in the Designer Output pane.

This page defines the shared build block once. Sibling pages ([designer_action Reference](#) and [ConsoleMCP Reference](#)) reference this shape instead of redefining it.

[Principle: MCP Surfaces, It Does Not Implement](#)
[The build Block](#)
[Affected Tables](#)
[Where The Block Appears](#)
[Source Of The Data](#)
[Agent Usage Pattern](#)

Principle: MCP Surfaces, It Does Not Implement

The FrameworX compile engine is unchanged in 10.1.5. No new compiler, no new validation passes, no new error categories. The MCP SDK is a thin adapter over product capability — when the product gains a signal worth exposing, the adapter routes it; when the product lacks a signal, the adapter does not fabricate one. Build visibility follows this principle: the compile output already existed inside Designer and inside the produced .dbs1n; the 10.1.5 work was to surface it to MCP callers.

Practically, this means the build block described below is drawn from the same internal sources an engineer already consults — the incremental compile result held by the Designer save pipeline, the error columns on the four *Contents tables in the solution file, and the per-object diagnostics produced by RuntimeBuildAndPublish.

The build Block

All MCP tools that report compile output return the same JSON shape. The shape is stable across DesignerMCP, ConsoleMCP, and the new verify_solution_file tool.

```
{
  "objects": [
    {
      "type": "Script|Display",
      "name": "<object name>",
      "status": "ok|error",
      "diagnostics": [
        { "line": <int>, "msg": "<string>" }
      ],
      "elapsedMs": <int>
    }
  ],
  "summary": {
    "built": <int>,
    "failed": <int>,
    "skipped": <int>,
    "timestamp": "<ISO8601>"
  }
}
```

Fields

Field	Meaning
objects[].type	Either Script (covers the three Scripts tables) or Display.
objects[].name	Name of the compiled object, as it appears in the owning table.
objects[].status	ok if the incremental compile produced no diagnostics, error otherwise.

<code>objects[].diagnostics[]</code>	Zero or more entries, each with a source line number and a compiler message. Empty array when status is ok.
<code>objects[].elapsedMs</code>	Compile time for that object in milliseconds.
<code>summary.built</code>	Count of objects whose status is ok.
<code>summary.failed</code>	Count of objects whose status is error.
<code>summary.skipped</code>	Count of objects not compiled in this run (for example, unchanged objects when <code>rebuild_all=false</code>).
<code>summary.timestamp</code>	ISO-8601 timestamp of the build run.

Affected Tables

Four tables carry user-authored .NET and participate in the build pipeline:

Table	Carries
<code>ScriptsTasks</code>	Background script tasks.
<code>ScriptsClasses</code>	Named script classes reusable across the solution.
<code>ScriptsExpressions</code>	Reusable compiled expressions.
<code>DisplaysList</code>	Display code-behind.

A write that touches a row in any other table does not produce a build block (there is nothing to compile). A write that touches a row in one of these four tables carries per-row compile feedback in its response.

Where The Block Appears

Surface	Mechanism	Reference
DesignerMCP <code>designer_action('build', ...)</code>	Returns a full build block from <code>RuntimeBuildAndPublish</code> .	designer_action Reference
DesignerMCP <code>write_objects</code>	Each affected row in the response carries a <code>compile</code> field populated from the save-time incremental compile.	designer_action Reference
DesignerMCP <code>get_objects (detail='full')</code>	Each row of the four affected tables carries a <code>lastCompile</code> field from the most recent compile.	designer_action Reference
DesignerMCP <code>SolutionContext / RefreshContext</code>	A compact <code>build_state</code> section reports last-build pass/fail counts and timestamp.	designer_action Reference
ConsoleMCP <code>create_solution_file / update_solution_file</code>	Response includes the build block read from the produced <code>.dbsln</code> .	ConsoleMCP Reference
ConsoleMCP <code>verify_solution_file</code>	New tool. Opens a current-version <code>.dbsln</code> and returns Name inventory plus the build block.	ConsoleMCP Reference

Source Of The Data

DesignerMCP draws the build block from the live Designer compile pipeline — the same path that populates the Output pane when an engineer saves a script or runs `RuntimeBuildAndPublish`.

ConsoleMCP has no live Designer. It reads the same information directly from the produced solution file. A `.dbsln` is a SQLite database; each of the four affected tables has a companion `*Contents` table (`ScriptsTasksContents`, `ScriptsClassesContents`, `ScriptsExpressionsContents`, `DisplaysListContents`) whose error columns hold the compile result written at build time. ConsoleMCP queries those columns and assembles the build block. This is not stdout parsing of `SolutionCreator` logs — it is a direct read of the compiled result persisted inside the `.dbsln`.

Agent Usage Pattern

A well-behaved agent writes, reads the `compile` field on the response, and stops on the first `error` status rather than piling up additional writes on a broken tree. Typical loop:

```
1. write_objects([...])                # Designer or Console
2. inspect response[i].compile.status  # for each affected row
3. if any status == "error":
    fix the offending diagnostics and retry
    else:
        proceed to the next batch
4. at commit boundary (Designer): designer_action('build', rebuild_all=true)
   at commit boundary (Console): create_solution_file / update_solution_file
5. inspect summary.failed; stop if > 0
```

The commit-boundary step is what catches cross-object breakage (an edit to a Script Class that breaks a Script Task referencing it): the row-level compile on a single write only reports what the incremental pipeline saw for that object; the commit-boundary build reports the whole tree.

In this section...
