Skill Display Construction - Basics

Purpose

This skill is the common ground for every FrameworX display build, regardless of paradigm. Load it whenever you do display work. Pair with **Skill Display Construction - Canvas** for pixel-positioned HMI/P&ID displays or with **Skill Display Construction - Dashboard** for grid-based data displays.

What this skill covers:

- The theme-first mental model (no hex unless it carries process meaning)
- `write_objects` mechanics on `DisplaysList` (document objects, read-before-write)
- The build loop (`write_objects` `get_state` check `errorList` visual checkpoint fix respond)
- Binding syntax that hits every display
- Spacing, typography, and minimum sizing tokens shared between Canvas and Dashboard
- Symbol-writing rules (Wizard, Library, Solution prefixes)

What this skill does NOT cover: the equipment cookbook, controls reference, or grid-cell layout — those live in Canvas and Dashboard.

Prerequisites

- Solution open with at least a minimal UNS (tags created). Displays reference tags; without tags there is nothing meaningful to bind.
- You know which theme family you are targeting. If you don't, pick `Steel` (light office) or `Onyx` (dark control room).

Tools you use over and over

Tool	When
`get_table_schema('DisplaysList')`	Once at start of session to confirm field names
`list_elements('ThemeColors')`	Once to get the brush catalogue
`list_elements('ThemeNames')`	Once to get the theme-pair catalogue (Light/Dark, Steel/Graphite, etc.)
`list_elements('<ElementName>')`	Before using an element type you have not used this session
`list_dynamics()` / `list_dynamics('<Type>')`	Before attaching a dynamic you have not used this session
`get_objects('DisplaysList', names=['X'], detail='full')`	Before every modification to an existing display (document-object rule)
`write_objects(table_type='DisplaysList', data=[...])`	After your plan is complete
`get_state(target='designer')`	After every write, to check `errorList` and `readOnly`
`get_screenshot(target='display', element='X')`	At visual checkpoints during a Canvas build. See §3 for the cap.

Section 1 — Theme-first mental model

The one-sentence rule

Do not hardcode hex colors unless the hex carries process meaning. Use `FillTheme`, `StrokeTheme`, `ForegroundTheme`, `BackgroundTheme`, `BorderBrushTheme` properties with semantic brush names from `list_elements('ThemeColors')`.

Hex is reserved for things with domain meaning that shouldn't change across themes — heat = red, water = blue, alarm = red, batch progress = amber. Everything else should adapt to whichever of the 13 theme pairs the operator is running.

The 13 theme pairs

Call `list_elements('ThemeNames')` for the authoritative list at runtime. The typical usage map:

Light / Dark pair	Use for
`Light` / `Dark`	Default office / default control room
`Platinum` / `Onyx`	Refined corporate office / premium control room
`Steel` / `Graphite`	Industrial office / industrial control room
`Pearl` / `Indigo`	Soft UI emphasis, OEM branding

`Sky` / `Navy`	Refreshing / deep contrast
`Gold` / `Coffee`	Warm accents for specialty applications
`ContrastLight` / `ContrastDark`	Accessibility, outdoor tablets, low-vision

Theme is switched at runtime via `@Client.Theme = "Dark"`. Every properly-themed display reflows automatically.

Pick a theme at the start of the build

Before placing any element, decide which theme family this solution is for:

- **Control room (dark, dim)** Onyx or Graphite
- **Office dashboard (bright, air-conditioned)** Steel or Platinum
- **Outdoor tablet (sunlight)** ContrastLight
- **OEM branded kiosk** Pearl/Indigo or Gold/Coffee
- **Unknown / default** Steel (light default) or Onyx (dark default)

Then use `*Theme` properties throughout — the displays render correctly under any theme, but look best under the one you designed for.

The brush catalogue

Call `list_elements('ThemeColors')` for the authoritative list. The 12 brushes you'll use 90% of the time:

Brush	Meaning	Typical use
`PanelBackground`	Card / section background	Background `Rectangle` inside a zone
`PageBackground`	Full-page background	The display-root Background override
`ControlBackground`	Control body	Gauge / chart / data-grid background
`TextForeground`	Primary text	Headings, values, labels
`TextSubtleForeground`	Meta text	Units, captions, "UPDATED AT" stamps
`TextAccentForeground`	Highlighted text	Section titles, accent links
`AccentBrush`	The solution's accent color	Active-state markers, selected-row borders, links
`StateGreen`	Running / OK state	Indicator fills, live-data values
`StateRed`	Stopped / fault state	Indicator fills, alarm text
`StateAlarm`	Active alarm (yellow)	Banded gauge danger zones, pulsing elements
`OnFill` / `OffFill`	HPG two-state fills	Status-indicator shape fills tied to a boolean
`Water`	Process water	Pipe runs carrying water/aqueous streams

Other brushes exist and are valid — `ElementBlue`, `ElementGreen`, `AlarmHighPriority`, `ColorCyan`, `ColorSlate`, `ColorAmber`, `ColorTeal`, `ColorCoral`, `ColorPurple`, `ColorDimmed`, `PopupBackground`, `DefaultFill`, `DefaultStroke`, `DefaultBorder`, and more. Check `list_elements('ThemeColors')`.

Note on Element*-family brushes: `ElementRed`, `ElementOrange`, `ElementYellow` may not apply consistently on all shape types. If you don't see your intended color, prefer `StateRed` / `ColorAmber` / `StateAlarm`.

How theme properties work on the wire

When you write `{"FillTheme": "ElementBlue"}` the writer stores it as `{"Fill": "theme:ElementBlue"}`. Both input forms are equivalent:

- `{"FillTheme": "StateGreen"}` stored as `{"Fill": "theme:StateGreen"}`
- `{"Fill": "theme:StateGreen"}` stored as `{"Fill": "theme:StateGreen"}`
- `{"Fill": "#FFEF4444"}` stored as `{"Fill": "#FFEF4444"}` (hex)
- `{"Fill": "#FF000000", "FillTheme": "ElementBlue"}` stored as `{"Fill": "#FF000000"}` (hex wins)

Use `FillTheme` / `StrokeTheme` / `ForegroundTheme` / `BackgroundTheme` / `BorderBrushTheme` for readability — it documents your intent. The writer collapses to the prefixed form. If you mean a hardcoded color, use `Fill`/`Stroke`/`Foreground` etc. directly.

The display-root Background footgun

Every newly created display gets `"Background": "#FFFAFAFA"` (light gray) baked in. If you're building for a dark theme, this shows through at every gap or transparent edge. **Always set the root Background** explicitly:

```
{
  "Name": "MyDisplay",
  "PanelType": "Canvas",
  "Size": "1600 x 900",
  "Width": 1600, "Height": 900,
  "Background": "theme:PageBackground",
  "Elements": [ /* ... */ ]
}
```

Or, for a full-bleed dark background on a dark-themed display, place a full-size Rectangle with `FillTheme: "PageBackground"` as the first element.

Section 1.5 — The visual quality bar

The first 5 seconds an operator looks at the display decide whether the build is good. Compile cleanliness is necessary but not sufficient — `errorList` is blind to whether anyone would actually want to use the screen. Before responding as finished, three things must be true within five seconds of imagining the operator opening the display:

1. **The page is not empty.** Every display, especially a landing display like `MainPage`, has at least one piece of content visible above the fold. An empty `MainPage` is a UX failure even when `errorList` is clean.
2. **The largest text is readable from operator working distance.** Hero numbers 26–32, body text never below 14, meta text never below 10 (see §5 typography ramp). A page where every label is 9 pt is a fail regardless of layout.
3. **The layout has obvious top-to-bottom or left-to-right rhythm.** Zones, cells, or sections — never a free-floating cluster of small elements with no spatial story. The viewer's eye should know where to start.

If any of the three fails, the build is not finished. Pair this with the `errorList`-blind list in §3 below: `errorList` handles compile correctness, the visual quality bar handles operator-perceived correctness. The two together — not `errorList` alone — gate "done."

Closing self-check before respond. Ask: "Would I be embarrassed to show this to an operator?" If the answer is "maybe" or "a little", run the visual checkpoint in §3 again before responding.

Section 2 — Write mechanics

The canonical display envelope

```
{
  "Name": "MyDisplay",
  "PanelType": "Canvas",
  "DisplayMode": "Page",
  "Navigate": "true",
  "Size": "1600 x 900",
  "OnResize": "StretchFill",
  "Width": 1600,
  "Height": 900,
  "Background": "theme:PageBackground",
  "Elements": [ /* ... */ ]
}
```

Non-negotiable rules

- **The identifier field is `Name`.** Never `ObjectName`.
- **`PanelType` is required.** `"Canvas"` or `"Dashboard"`. Omitting it silently defaults to Canvas.
- **No `JsonFormat` wrapper.** All properties at the top level.
- **`DisplayMode`** controls window style: `"Page"` (normal), `"Dialog"` (modal popup), `"Popup"` (floating non-modal), `"PopupWindow"` (separate window — not available on Web, avoid unless explicitly requested).
- **`OnResize`**: `"StretchFill"` is the sensible default. Use `"NoAction"` for fixed-pixel displays.
- **`DisplaysDraw` is NOT a writable table.** It is the Designer visual editor UI. Use `DisplaysList` for all display create/edit operations.

Displays are document objects — read before write

Displays, Symbols, Scripts, Queries, and Reports are **document objects**: `write_objects` replaces the entire document on save. Omitted content is deleted.

When modifying an existing display:

1. Read the full document: `get_objects('DisplaysList', names=['MyDisplay'], detail='full')`
2. Modify your working copy (add/change/remove elements)
3. Write the **complete merged document** back

Never send a partial update — you'll silently delete everything you didn't include.

Keep the element list in working memory. During a session that adds, modifies, or removes elements across multiple turns, track what the display holds — same discipline a programmer uses to keep the state of the code they're editing in their head. When you write, every element you intend to keep must be in the payload, and elements you removed must be absent. `errorList` will not tell you about an element you forgot to drop; only the user will notice it's still on screen.

`MainPage` is predefined in new solutions. Write main content directly into it — no need to create a new display for the landing screen.

Category and MCP-label guard

AI-created objects are automatically tagged with the `MCP` category, which makes them editable / overwritable by subsequent AI writes. Objects without this label are "owned" by the user and produce `Skipping existing` on write attempts. This is the user-protection mechanism — respect it. When you see `Skipping existing`, the user can add the MCP label in Designer if they want to allow AI edits.

Exception: Description fields are always writable, even on user-owned objects without the MCP label. If the user asks you to update a description on an object you otherwise can't edit, that specific field will still save.

`solution_id` on every write

Every `write_objects`, `delete_objects`, `rename_objects`, and `designer_action` call should include the `solution_id` from the most recent `open_solution` or `create_solution` response. This verifies the session is still connected to the expected solution.

Section 3 — The build loop

This cadence catches errors within the same turn they were introduced, before they accumulate.

1. **PLAN** — what am I adding/changing? What tags does it bind to?
2. **READ** — `get_objects(detail='full')` if modifying an existing display
3. **WRITE** — `write_objects(data=[...])`
4. **COMPILE CHECK** — `get_state(target='designer')`, look at `errorList`
5. **FIX** — if `errorList` non-empty, fix and go back to step 3
6. **VISUAL CHECKPOINT** — see "What errorList covers, and what it doesn't" below
7. **RESPOND** — summarize what was built to the user and move to the next request

Reading `errorList` and `readOnly` from `get_state`

`get_state` on a display returns compile errors as a list, and also reports whether the Designer is currently read-only:

```
{
  "readOnly": false,
  "readOnlyReason": null,
  "errorList": [
    {
      "ID": 0,
      "ErrorCode": "BC30456",
      "IsWarning": false,
      "Line": 8,
      "Column": -1,
      "Location": "Uid_41_TTextBlock_LinkedValue_e1",
      "ErrorText": "error BC30456: 'Status' is not a member of 'UserType'."
    }
  ]
}
```

The `Location` string tells you **which element** (by Uid) and **which property** (LinkedValue, Expression, etc.) has the problem. Use it to find and fix the exact property.

If `errorList` is absent or empty, the display compiled clean.

Also check `readOnly`. If it is `true`, a subsequent `write_objects` will fail — check `readOnlyReason` and surface it to the user before retrying. Common causes: runtime is running, another client has the display open for edit.

What errorList covers, and what it doesn't

`errorList` is ground truth for **compile correctness**: bindings resolve, required fields are present, the display will render without crashing. A clean `errorList` is necessary before you respond as finished.

`errorList` is blind to **visual correctness**. None of the following produce an error:

- Two elements overlapping at the same `Left`/`Top`
- A `TextBlock` whose `LinkedValue` clips because its rendered length exceeds `Width`
- An element placed outside its intended zone but still inside the display
- A pipe endpoint 15 px off the tank nozzle it was supposed to connect to
- A `RadialGauge` with `YMaxValue` below the tag's engineering-range maximum (pegs at max forever)
- Three 48 pt "hero" numbers on the same display, all screaming for attention
- A `Polygon` with valid `Points` but `Fill` equal to the background (invisible)
- A `MainPage` (or any landing display) that opens empty
- A detail panel that opens empty when no master row is selected (no empty-state placeholder)
- A page where every visible element is below the fold or off in a corner — nothing in the top half of the canvas

All of those compile clean. All are operator-rejected.

Visible-on-open scan (both paradigms)

Before any paradigm-specific procedure, mentally open the display from the operator's seat, top edge first. If there is nothing in the top half of the canvas — or if the only thing there is a placeholder waiting for a master selection that never came — that is a failure even on a clean `errorList`. Go back and add content, restructure, or wire an empty-state placeholder before continuing to the paradigm-specific checkpoint below. This catches the empty-`MainPag` failure and the empty-detail-panel failure before any screenshot or pixel-tweak.

Visual checkpoint — paradigm-specific

This step closes the gap `errorList` leaves open. The rules differ by paradigm because spatial correctness matters far more in Canvas than in Dashboard.

Canvas: spatial correctness is the whole point. Coordinate-based composition of pipes, vessels, zone layouts, and P&ID flow is the class of work where authoring blind gets it wrong the most. Take `get_screenshot(target='display', element='X')` at authoring checkpoints:

1. After zone scaffolding lands (background rectangles + zone titles) — verify the zones divide the canvas as planned.
2. After main equipment is placed (symbols, vessels, pipes) — verify elements landed inside their zones and pipes connect where they should.
3. Before responding as finished — final sanity check of the composed display.

Hard cap: 3 screenshots per display per session. If three checkpoints did not resolve the visual problem, a fourth will not either — the problem is judgment, not pixels. Hand back to the user with a summary of what was built and what looks wrong; ask them to open Designer and guide the next move.

Dashboard: grid reflow is resolved at render time against a specific window size, and most Dashboard mistakes (wrong control choice for the cell purpose, missing `Cell.HeaderLink`, broken DataGrid detail wiring) are visible in the structure, not the pixels. Screenshots are generally unnecessary for Dashboard — rely on a clean `errorList` and a clear written summary to the user. If something looks wrong after the user previews it, fix and iterate; don't screenshot speculatively.

Not for iterative self-soothing. Do not take a screenshot after every write just to feel certain. The cost is real and the behavior encourages re-work spirals over clean planning. Screenshots go at checkpoints, not at every turn.

Write a one-paragraph summary when you respond, regardless of paradigm: which zones or cells exist, what lives in each, which tags drive what. The summary is what the user reads before they look at the display themselves — and it forces you to notice gaps ("I reserved a zone for alarms but never put an `AlarmViewer` in it") that `errorList` cannot.

Section 4 — Binding syntax

`@Tag.` is the only binding prefix you'll use in displays

- `@Tag.<path>` — the tag's Value. Do NOT append `.Value`; it's implicit (`@Tag.X` == `@Tag.X.Value`).
- `@Tag.<path>/Attr.<member>` — a UDT member on the tag.
- `@Tag.<path>.<udt_member>` — alternate dot syntax for UDT members.
- `@Dataset.Query.<QueryName>` — dataset query result (for DataGrid, ComboBox DataTable source).
- `@Client.Context.<property>` — session-scoped client context (AssetName, AssetPath, Theme).
- `@Client.<property>` — general client-scoped tags (`@Client.Theme`, `@Client.AlarmPage.Filter`).
- `@Now` — current time.

Never use `@Label.` in display-element bindings. `@Label.` is only for symbol-definition internals.

Tag paths must be FULL paths

Include asset folders in every tag reference: `Area1/Line1/Motor1.Speed`, not `Motor1.Speed`.

Inline interpolation — the composite LinkedValue pattern

The single most important text pattern. Instead of two TextBlocks (value + unit) side by side, use ONE with composite LinkedValue:

```
{
  "Type": "TextBlock",
  "LinkedValue": "Temperature: {@Tag.Reactor/Temperature_C} °C",
  "FontFamily": "Inter",
  "FontSize": 14,
  "Width": 220, "Height": 22
}
```

Patterns:

- `"Flow: {@Tag.X} GPM"` — label + value + unit
- `"{@Tag.X} NTU"` — value + unit
- `"pH: {@Tag.X}"` — label + value
- `"Operator: {@Tag.Shift/CurrentOperator}"` — label + bound string

Polygon / Polyline / Gridline / Spline silently skip on missing Points

If you write a Polygon without `Points`, nothing renders and no error is produced — the element is simply invisible.

```
// Silently invisible
{ "Type": "Polygon", "Left": 100, "Top": 100, "Width": 50, "Height": 50 }
```

// Renders correctly { "Type": "Polygon", "Left": 100, "Top": 100, "Width": 50, "Height": 50, "Points": "25,0 50,50 0,50" }

Polygon auto-closes (last point connects back to first). Polyline doesn't. For pipe runs use Gridline (constrains to horizontal/vertical segments — the P&ID convention).

Writer normalizations — what you write what you read back

The writer normalizes several shapes on save. When you later read the display back, you get the normalized form:

You wrote	Stored as
`{"FillTheme": "ElementBlue"}`	`{"Fill": "theme:ElementBlue"}`
`{"Type": "Cylinder", Left, Top, Width, Height}`	`{"Type": "Path", "Data": "M 0,20 C ..."}`
`{"Type": "Hexagon", ...}`	`{"Type": "Polygon", "Points": "50,0 100,25 ..."}`
`{"Type": "SvgGroup", "SvgContent": "<svg>..."}`	`{"Type": "ShapeGroup", "Children": [...]}`
`{"Pens": [{TrendPen}]}`	Flat array OR wrapper — writer accepts both

Consequence: when round-tripping a display for edits, don't expect to see `"Type": "Cylinder"` — you'll see a `"Path"` with auto-generated Data. Edit the Path, or add new Cylinders alongside (they normalize too).

Section 5 — Spacing and typography tokens

Spacing scale

Token	Pixels	Where
xs	4	Gap between label and value in a stacked pair
sm	8	Gap between sibling sub-elements within a card
md	16	Gap between sibling cards in a row
lg	24	Card interior padding (content vs card edge)
xl	32	Zone padding (zone background Rectangle vs content)
2xl	48	Major section separator
3xl	64	Display-level margins

Typography ramp

Use `FontFamily: "Inter"` (or the solution's chosen font) universally.

Role	FontSize	When
Hero	26–32	Display title, single-metric hero number
H1	20–22	Main section headings
H2	16	Sub-section headings, card titles
Body	13–14	Bound values, primary text
Meta	10–11	Labels, units, captions
Micro	9	Timestamps, very-low-priority meta

Minimum sizes (hard floors)

Element	Min	Recommended
Button	100×32	130×40
TextBox / NumericTextBox	120×28	160×32
ComboBox	160×28	200×32
Slider	200×28	260×32
ToggleSwitch	60×28	80×32
CircularGauge / RadialGauge	150×150	180×180
SemiCircle	200×120	240×140
LinearGauge (horizontal)	260×80	300×100
CenterValue	120×120	140×140
TrendChart	400×200	500×300
BarChart	300×200	400×240
PieChart	200×200	240×240
AlarmViewer	400×220	600×240
AssetsTree	200×300	240×400
DataGrid	400×200	600×300
Wizard symbol (TANK/PUMP/etc.)	60×60	80×80

Color hex fallbacks (for process meaning only)

Role	Hex
Heat / reaction	`#FFEF4444` (red)
Cold / water	`#FF38BDF8` (cyan)
Running / active	`#FF34D399` (green)
Alarm / warning	`#FFF59E0B` (amber)
Accent / link / data	`#FF38BDF8` or theme `AccentBrush`
Text on dark	`#FFF3F4F6` or theme `TextForeground`
Meta text	`#FF64748B` or theme `TextSubtleForeground`

Section 6 — Element types: discover at runtime

There are many element types — Shapes, Container, Interaction, Gauges, Charts, Viewer, Editors, IndustrialIcons, and the Dashboard-specific Cell. The catalogue also evolves between releases.

Rather than rely on a static list that drifts, always discover at runtime:

- `list_elements()` — every element type, grouped by category
- `list_elements('<ElementName>')` — the schema for one specific element (valid properties, defaults, notes)
- `list_elements('Wizard')` — the 5 Wizard symbols
- `list_elements('Library/HMI')` or subpath — Library symbols (see §7)
- `list_elements('ThemeColors')` — brush catalogue
- `list_elements('ThemeNames')` — theme-pair catalogue

Canvas and Dashboard skills cover which element types fit each paradigm and how to use them in context.

Section 7 — Symbols (all three sources)

Every symbol uses `Type: "Symbol"`. Not `Type: "Pump"`, not `Type: "Valve"`. One element type, many `SymbolName` values.

```
{
"Type": "Symbol",
"SymbolName": "Wizard/PUMP",
"Left": 400, "Top": 300,
"Width": 80, "Height": 80,
"SymbolLabels": [
{ "Type": "SymbolLabel", "Key": "State", "LabelName": "State", "LabelValue": "@Tag.Pump1/Running", "FieldType":
"Expression" },
{ "Type": "SymbolLabel", "Key": "RPM",   "LabelName": "RPM",   "LabelValue": "@Tag.Pump1/Speed",   "FieldType":
"Expression" }
]
}
```

The 5 Wizard symbols

The complete Wizard catalog is **5 symbols**:

SymbolName	What it is	Typical SymbolLabels
`Wizard/BLOWER`	Industrial blower/fan	State, RPM
`Wizard/MOTOR`	Electric motor	State, RPM
`Wizard/PUMP`	Pump (styles selectable in Designer)	State, RPM
`Wizard/TANK`	Storage tank	Level, Alarm
`Wizard/VALVE`	Valve (various styles)	State, Position

When to reach for a Wizard. When the equipment in the spec is a fan/blower, a rotating drive, a liquid mover, a vessel with level dynamics, or a flow-control element, the corresponding Wizard is almost always the right first move — they ship with state dynamics already wired and avoid the cost of composing the same shape from primitives. The Wizard catalog stops at five canonical types on purpose; for anything outside (compressor, heat exchanger, conveyor, sensor, instrument), fall through to Solution / Library/HPG / Library/HMI in that order. Wizard styling variants (orientation, foot detail, port direction) are user-controlled in Designer via the symbol's configuration button, not by editing the placed element's geometry.

Always treat `list_elements('Wizard')` as the authoritative runtime catalog — if an entry does not appear there, do not attempt to use it.

Library symbols (~1600)

Call `list_elements('Library/HMI')` or specific subfolders (`Library/HMI/Pumps`, `Library/HMI/Valves`, etc.) to discover. **Results are limited to 50 per call** — when you see `truncated: true`, drill into specific subfolders to see complete listings. Library symbols **auto-import** on first reference:

```
{ "Type": "Symbol", "SymbolName": "Library/HMI/Pumps/CentrifugalPump", "Left": 200, "Top": 200, "Width": 120,
"Height": 80 }
```

Solution symbols (your own)

User-created symbols in the current solution use `Solution/` prefix:

```
{ "Type": "Symbol", "SymbolName": "Solution/MyCustomTank", "Left": 100, "Top": 100, "Width": 80, "Height": 100 }
```

Sizing

All symbols scale to any proportional size. Default is 65×65 — use 40×40 for compact, 80×80 for medium, 120×120 for large. Maintain aspect ratio; don't stretch asymmetrically.

Binding via SymbolLabels — never via direct properties

SymbolLabels is the ONLY way to push data into a symbol:

- `Key` must match a label key defined inside the symbol (case-sensitive).
- `LabelValue` uses `@Tag.X` / `@Client.Context.X` / `@Now` / composite strings.
- `FieldType: "Expression"` is the default and handles tag paths and expressions.
- **Never use `@Label.X`** in display-element SymbolLabels. `@Label.` is a symbol-definition internal only.

Section 8 — Layouts and asset navigation (brief)

Layouts

Layout regions: Header, Footer, Menu, Submenu, Content. The **Startup** layout defines which display loads into each region.

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

Asset navigation

For plant-wide navigation with dynamic content:

- Layout with Header + AssetTree (left menu) + Content region
- User selects an asset `Client.Context` updates content displays react
- **Static binding:** `@Tag.Area1/Line1/State`
- **Dynamic binding:** `Asset(Client.Context.AssetPath + "State1")`

A single display template can show data for whichever asset the operator selects. See the Dashboard skill for the AssetsTree + ChildDisplay master-detail pattern.

Section 9 — CodeBehind (brief pointer)

CodeBehind is client-side C# or VB.NET code embedded in each display. Lifecycle methods: `DisplayOpening()`, `DisplayIsOpen()`, `DisplayClosing()`, `DialogOnOK()`.

The `Contents` field format starts with `CSharp\r\n` or `VBdotNet\r\n` before the code. Full CodeBehind guidance lives in the **Skill Scripts Expressions** skill.

Section 10 — Common pitfalls

Mistake	Fix
Responding as finished on empty `errorList` alone	`errorList` covers compile correctness only. Run the paradigm-specific visual checkpoint in §3 before responding as finished.
Taking screenshots after every write	Checkpoints, not reassurance. Canvas: up to 3 per display. Dashboard: generally unnecessary.
Losing track of the element list across turns	Keep the display's element list in working memory as you add/modify/remove across the session. An element you forgot to drop will still render; `errorList` will not catch it.
Ignoring `readOnly: true` on `get_state`	Check `readOnly` / `readOnlyReason` before writing. Common: runtime running, display open for edit elsewhere.
Hardcoding hex without thinking about theme switching	Use `*Theme` properties; reserve hex for process-meaning
Polygon / Gridline without Points	Always include `Points` — min 3 for Polygon, 2 for Polyline/Gridline
Using `ObjectName` instead of `Name`	Field is always `Name`
Using `DisplaysDraw` as table_type	Visual editor UI, not a writable table. Use `DisplaysList`
Omitting `PanelType`	Required — `"Canvas"` or `"Dashboard"`
Wrapping the envelope in `JsonFormat`	Properties go at top level, no wrapper
Partial write on an existing display	Always read-modify-write the complete document
Using `@Label.X` in a display-element binding	`@Label.` is for symbol internals only — use `@Tag.`
Setting colors without clearing theme	Set value AND clear theme: `{Fill: '#FF3498DB', FillTheme: ''}`
Calling `get_screenshot` with `name=`	The parameter is `element=`. `name=` is silently ignored and the call falls back to the current UI view.
Relying on a static element/symbol list	Always call `list_elements()` / `list_dynamics()` to get the authoritative catalog at runtime

Section 11 — Quick reference

```
// Session startup
get_table_schema('DisplaysList')
list_elements('ThemeColors')
list_elements('ThemeNames')
```

// Before first use of a type this session list_elements('<ElementName>') list_dynamics('<DynamicName>')

// Read-modify-write an existing display get_objects('DisplaysList', names=['X'], detail='full') write_objects(table_type='DisplaysList', data=[...]) get_state (target='designer') // verify errorList empty AND readOnly false

// Visual checkpoint (see Section 3) // Canvas: up to 3 per display (after zones, after equipment, before done) // Dashboard: generally unnecessary get_screenshot(target='display', element='X')

Display envelope template:

```
{
"Name": "MyDisplay",
"PanelType": "Canvas",
"DisplayMode": "Page",
"Navigate": "true",
"Size": "1600 x 900",
"OnResize": "StretchFill",
"Width": 1600,
"Height": 900,
"Background": "theme:PageBackground",
"Elements": []
}
```

Next skills

After the basics are internalized, load the paradigm-specific skill:

- **Skill Display Construction — Canvas** — absolute-positioned displays for HMI, P&ID, process overviews, equipment layouts
- **Skill Display Construction — Dashboard** — grid-based displays for data monitoring, KPI walls, operator consoles

Both assume this Basics skill is loaded.