Skill Display Construction - Dashboard

Purpose

Dashboard displays are the paradigm for **data cards, not diagrams**. Rows and columns of gauges, trends, KPIs, tables. Every element lives in a `Cell`, and cells are arranged in a responsive grid that reflows on window resize.

Use Dashboard when:

- The display is primarily **data monitoring** (live values, trends, alarms)
- The layout is a **grid of cards** — fleet status, shift summary, KPI wall, operator console
- The screen must reflow across window sizes
- There's no spatial/physical relationship between equipment to preserve

Use Canvas (load **Skill Display Construction — Canvas** instead) when equipment positions relative to each other matter (pipe A connects to vessel B), you're building a P&ID, or you need animation dynamics.

Prerequisite: load **Skill Display Construction — Basics** first.

Section 1 — Dashboard mental model

A Dashboard is a grid of cells

The whole display is defined by:

1. A **DashboardDisplay** root with `Columns` and `Rows` arrays describing the grid track sizes
2. A **Cells** array, where each cell specifies `Row`, `Col`, optional `RowSpan`/`ColSpan`, a `Cell.HeaderLink` for the card title, and `Content` (the element that fills the cell)

```
{
  "Name": "OperationsOverview",
  "PanelType": "Dashboard",
  "DashboardDisplay": {
    "Columns": ["*", "*", "*"],
    "Rows": ["Auto", "*", "*"],
    "Cells": [
      { "Row": 0, "Col": 0, "ColSpan": 3, "Cell.HeaderLink": "Plant Overview", "Content": { /* header card */ } },
      { "Row": 1, "Col": 0, "Cell.HeaderLink": "Production Rate", "Content": { /* gauge */ } },
      { "Row": 1, "Col": 1, "Cell.HeaderLink": "Active Alarms", "Content": { /* alarm viewer */ } },
      { "Row": 1, "Col": 2, "Cell.HeaderLink": "Shift Output", "Content": { /* KPI */ } }
    ]
  }
}
```

Track sizing

The `Columns` and `Rows` arrays define track sizes. Each entry can be:

- `"*"` — equal share of remaining space (like CSS `1fr`)
- `"2*"` — twice the share (like CSS `2fr`)
- `"Auto"` — size to content
- `"240"` — fixed pixel width
- `"*,min=200"` — fractional with a minimum in pixels

Standard patterns:

Layout	Columns	Rows
3 equal columns, 2 rows	`["*","*","*"]`	`["*","*"]`
Header + 2 equal rows	`["*","*","*"]`	`["Auto","*","*"]`
Sidebar + main	`["240","*"]`	`["*"]`
Header + sidebar + main	`["240","*"]`	`["Auto","*"]`
4 KPIs across top, trend below	`["*","*","*","*"]`	`["Auto","*"]`
Operator wall (4×3)	`["*","*","*","*"]`	`["*","*","*"]`

Don't think in pixels — think in cells

In Canvas you place a gauge at `Left: 472, Top: 320`. In Dashboard you place a gauge at `Row: 1, Col: 2`. The engine handles pixel positioning, cell padding, and reflow.

This is a completely different mental model. If you find yourself calculating Left/Top/Width/Height for Dashboard elements, stop — you're building a Canvas display by accident.

Cells size to their content, not the other way around

Setting a row or column to `"*"` gives that track the space; it does not force the cell's content to fill it. A fixed-size gauge in a `"*"` row will sit at its native size inside a large cell, not stretch. For content that should fill the cell, use stretch-friendly controls (TrendChart, BarChart, AlarmViewer, DataGrid). Gauges are fixed size; place them in cells sized to match.

Dashboard-compatible controls only

Not every element type works inside a Dashboard cell. Call `list_elements('Dashboard')` for the authoritative compatibility list in the current release. Rule of thumb:

- **Works in Dashboard:** Interaction, Charts, Gauges, Viewer, Editors, TextBlock, Label.
- **Canvas-only:** shape primitives, first-class auto-shapes, and containers.

Dynamics work in Dashboard for visual controls (FillColorDynamic on a TextBlock, VisibilityDynamic on a chart) but the animation dynamics (Rotate, Scale, MoveDrag, Skew, Bargraph) are **Canvas-only** — they're silently ignored in Dashboard cells.

Section 1.5 — Responsive behavior (OnResize and the six layout locks)

A Dashboard grid already reflows: cells track their `Columns`/`Rows` sizing and re-pack on window resize. But a FrameworX solution is built from *several* displays that share one window — a Header, maybe a Footer, a Menu rail, and a content page docked into the middle. How each of those displays behaves when the window changes size is controlled by one display-level property, `OnResize`, plus a small set of per-element and per-display **locks**. Get these right and the whole layout breathes cleanly across a 4K control-room wall and a 1366-wide laptop; get them wrong and content compresses, clips, or floats.

The `OnResize` decision tree

Set `OnResize` once per display, by what the display *is*:

Display kind	`OnResize`	Then also…
Dashboard / data page with a prominent element that should grow (DataGrid, TrendChart, AlarmViewer)	`Responsive`	Set the element locks `IsWidthAlign` + `IsHeightAlign` on that prominent element (see below). A full-width band element gets `IsWidthAlign` only; a bottom-pinned input row gets `IsTopAlign` + `IsWidthAlign`; a right-edge control gets `IsLeftAlign`; small top-left labels and nav icons get no locks.
Process diagram / drawing (P&ID, equipment layout)	`StretchUniform`	Nothing — the whole drawing scales as one, preserving aspect ratio.
Viewer / text page (text-led page where everything should scale together)	`ResizeChildren`	Nothing — children scale with the container.
Header / Footer band	`Responsive`	Set the display lock `LockedHeight` to pin the band to its native pixel height. Match the Header's native `Width` to the Layout `Width` so it does not compress horizontally.
Menu / SubMenu rail	`Responsive`	Set the display lock `LockedWidth` to pin the rail to its native pixel width.
Popup / Dialog / Form	`NoAction`	Nothing — the modal keeps its authored size.

Docked Header / Footer / Menu are `Responsive`, not `NoAction`. The intuition that a fixed band should be "frozen" (`NoAction`) is wrong for a *docked* region: `NoAction` stops the band participating in the layout at all, while the demo-proven pattern is `Responsive` *plus* the matching dimension lock — `LockedHeight` for a Header/Footer, `LockedWidth` for a Menu. That keeps the band's pinned dimension fixed while the free dimension tracks the window. (`NoAction` is still correct for a Popup/Dialog/Form, which is *not* docked.)

Two different knobs — do not conflate them. Setting the Header region to span the full width of the layout (the region's `HorizontalAlign = Stretch`) is the layout-docking knob. It is *not* the same as `OnResize = StretchUniform`. One controls how the region docks into the layout; the other controls how the display's contents behave on resize.

The six layout locks

The locks come in two levels. You author them as **flat properties** — on the element, or on the display — and the platform translates them into the underlying resize behavior. **You never write the raw WPF attached-property XAML**; just set the flat boolean.

Element-level locks (4) — set on an individual element inside a `Responsive` display. They are only honored when the display's `OnResize` is `Responsive`; on any other `OnResize` they are ignored.

Property	Effect on resize
`IsWidthAlign`	**Stretch** the element's width with the container.
`IsHeightAlign`	**Stretch** the element's height with the container.
`IsLeftAlign`	**Move** the element to track the container's right edge (keeps its right-side gap constant).
`IsTopAlign`	**Move** the element to track the container's bottom edge (keeps its bottom gap constant).

So `IsWidthAlign`/`IsHeightAlign` *grow* the element; `IsLeftAlign`/`IsTopAlign` *reposition* it. A DataGrid or TrendChart that should fill its area gets both `IsWidthAlign` + `IsHeightAlign`. A toolbar pinned to the bottom gets `IsTopAlign` + `IsWidthAlign` (it rides the bottom edge and stretches across). A small label in the top-left corner gets no locks — it stays put at its native size.

Display-level locks (2) — set on the display itself, to pin a docked region to native pixels under `Responsive`:

Property	Use on	Effect
`LockedHeight`	A docked Header or Footer band	Pins the band's height to its native pixels; the width still tracks the window.
`LockedWidth`	A docked Menu or SubMenu rail	Pins the rail's width to its native pixels; the height still tracks the window.

Worked examples (Asset Monitor Demo)

The shipped Asset Monitor demo is the canonical reference for every case above:

Display	Native size	`OnResize` + locks
Header	1546×60	`Responsive` + `LockedHeight`; native Width matched to the layout Width (1546) so there is no horizontal compression.
HeaderPad	1366×40	`Responsive` + `LockedHeight`.
HeaderMobile	400×40	`NoAction` + `LockedWidth` + `LockedHeight` (a fixed-size mobile band).
MenuTree	180×628	`Responsive` + `LockedWidth`. The AssetsTree inside carries `IsWidthAlign` + `IsHeightAlign`; the three bottom TextBoxes carry `IsTopAlign` + `IsWidthAlign`.
Area (a Responsive Canvas)	—	`Responsive`; the DataGrid carries `IsWidthAlign` + `IsHeightAlign`; a FlowPanel carries `IsWidthAlign`; the top-left label TextBoxes carry no locks.
Home / OEEDashboard (Responsive Dashboards)	—	`Responsive` with Star-sized grid columns + a GridSplitter + two ChildDisplays. **No element locks** — the grid does the reflow.
OEE_Left / OEE_Right	—	`StretchUniform` (fixed-aspect card panels living inside the dashboard cells).
Site detail pages (Dallas, Detroit, Houston, Europe)	—	`ResizeChildren`.

Notice the pattern in the two Responsive Dashboards (Home, OEEDashboard): when the grid itself carries the reflow — Star-sized columns, a splitter, ChildDisplays — the cells need **no element locks at all**. Element locks are for a `Responsive` display whose content is positioned rather than grid-tracked (the "Responsive Canvas" case, like Area above).

The Home pattern — a Dashboard with two ChildDisplays

The landing page of a solution is, by convention, a Dashboard named **Home** that hosts **two ChildDisplays** in a responsive grid — one for navigation /network, one for features/KPIs — separated by a GridSplitter. In the demo, `Home` embeds `Home_Network` and `Home_Features`. This is the highest-leverage screen to get right: it is the operator's first impression and the preview customers see. Build it as a `Responsive` Dashboard with Star columns, a GridSplitter, and two `ChildDisplay` cells — no element locks needed.

Keep MainPage in place — never rename to "consolidate." The landing page in this pattern is named `Home`; the template's `MainPage` (Display ID 0) is a *separate*, reserved page — the Solution Center preview anchor. Build `Home`, point the Startup layout's Content region at it, and leave `MainPage` in place (a static placard is enough for the thumbnail). Do **not** rename `MainPage` out of the way, or rename `Home` onto it: renaming the ID-0 page builds green but silently detaches the preview anchor. An orphaned-looking `MainPage` beside a wired `Home` is the intended state — not a cleanup target. See Displays List Reference § Special Pages.

Mobile: split a Canvas page into two child displays

When a content page must also render well on a phone, do not try to make one Canvas reflow to both form factors. Split it into **two child displays, each its own Canvas** — one composed for the wide desktop layout, one for the narrow mobile layout — and select between them per client. (The demo's `HeaderMobile` band, sized 400×40 with `NoAction` + both display locks, is the header half of this same mobile-split idea.)

Section 2 — Standard grid recipes

3×2 KPI wall (the most common)

Six data cards in a 3-column × 2-row grid:

```
{ "Name": "PlantKPIs", "PanelType": "Dashboard", "DashboardDisplay": { "Columns": ["*", "*", "*"], "Rows": ["*", "*"], "Cells": [ { "Row": 0, "Col": 0, "Cell.HeaderLink": "Production Rate", "Content": { "Type": "CenterValue", "LinkedValue": "@Tag.Plant/ProductionRate", "CenterTextFormat": "N1", "AccentTextLink": "units/hr" } }, { "Row": 0, "Col": 1, "Cell.HeaderLink": "Active Alarms", "Content": { "Type": "CenterValue", "LinkedValue": "@Tag.Plant/AlarmCount", "CenterTextFormat": "N0", "AccentTextLink": "alarms" } }, { "Row": 0, "Col": 2, "Cell.HeaderLink": "Operator on Duty", "Content": { "Type": "TextBlock", "LinkedValue": "{@Tag.Shift/CurrentOperator}", "FontSize": 22, "FontWeight": "Bold" } }, { "Row": 1, "Col": 0, "Cell.HeaderLink": "Temp Trend", "Content": { "Type": "TrendChart", "Duration": "5m", "Pens": [{"Type":"TrendPen","LinkedValue":"@Tag.Plant/AvgTemp","Stroke":"#FFEF4444"}] } }, { "Row": 1, "Col": 1, "Cell.HeaderLink": "Pressure Trend", "Content": { "Type": "TrendChart", "Duration": "5m", "Pens": [{"Type":"TrendPen","LinkedValue":"@Tag.Plant/AvgPressure","Stroke":"#FF38BDF8"}] } }, { "Row": 1, "Col": 2, "Cell.HeaderLink": "Shift Output", "Content": { "Type": "BarChart", "LinkedValue": "@Tag.Shift/OutputByHour" } } ] } }
```

4×3 operator wall

Twelve cells. Full-screen control-room overview with one card per reactor / line / zone:

```
"Columns": ["*","*","*","*"], "Rows": ["*","*","*"]
```

2-column detail (list detail)

Asset tree on the left, detail panel on the right:

```
{ "DashboardDisplay": { "Columns": ["240", "*"], "Rows": ["Auto", "*"], "Cells": [ { "Row": 0, "Col": 0, "ColSpan": 2, "Cell.HeaderLink": "Production Area A", "Content": { "Type": "TextBlock", "LinkedValue": "{@Tag.Site/Name} — {@Server.DateTimeInfo.Now}" } }, { "Row": 1, "Col": 0, "Cell.HeaderLink": "Equipment", "Content": { "Type": "AssetsTree" } }, { "Row": 1, "Col": 1, "Cell.HeaderLink": "{@Client.Context.AssetName}", "Content": { "Type": "ChildDisplay", "DisplayLink": "EquipmentDetail" } } ] } }
```

The AssetsTree has all its bindings pre-wired to `@Client.Context.*` by default — drop it in and navigation "just works."

ColSpan / RowSpan for asymmetric layouts

```
{ "Row": 0, "Col": 0, "ColSpan": 2, ... } // cell spans columns 0 and 1 { "Row": 1, "Col": 0, "RowSpan": 2, ... } // cell spans rows 1 and 2
```

Use ColSpan on a header card to stretch it across all grid columns. Use RowSpan when you want a tall element (AlarmViewer, AssetsTree) next to shorter cards.

Section 3 — Controls by cell purpose

Cell purpose	Best control	Why
Display title / section header	`TextBlock` with composite LinkedValue	`"{@Tag.Site/Name} — {@Server.DateTimeInfo.Now}"` in one element
Single-value KPI (numeric)	`CenterValue`	Pre-styled big-number-plus-unit tile
Single-value KPI (string)	`TextBlock` with `FontSize` 24+	Simpler than CenterValue for strings
Time-series trend	`TrendChart`	The workhorse
Tag value dial	`RadialGauge`	Gauge with threshold bands
Tag value bar (linear)	`LinearGauge`	For bar-style KPIs with setpoint pointer
Cumulative count / meter	`DigitalGauge` or `DigitalMeter`	Specialty numeric readouts
Categorical comparison	`BarChart`	e.g., output by shift
Current alarms	`AlarmViewer`	Pre-wired to Client.AlarmPage context
Asset hierarchy navigation	`AssetsTree`	Pre-wired to Client.Context
Dropdown selector	`ComboBox` (DataTable-backed for FK)	Zero-script FK lookup
Data table / list	`DataGrid`	The list in listdetail

Document viewer (SOP, work order)	`PdfViewer`	URL-bound inline PDF
Fleet/plant map	`MapsOSM`	Lat/long-positioned markers
Embedded another display	`ChildDisplay`	Reusable detail panels
Multiple panels user switches	`TabControl`	Manual click-to-switch
Slideshow of status boards	`Carousel`	Auto-cycle with AutoCycleLink
Collapsible panel	`Expander`	Click to expand advanced settings
Card-per-item layout	`FlowPanel`	N items from a data source, one template

The `Cell.HeaderLink` pattern

Every cell should have a `Cell.HeaderLink`. The value is either:

- A static string: `"Production Rate"`
- A bound value: `"@Tag.CurrentAsset/Name"`
- A composite: `"Reactor R-101: {@Tag.R101/Status}"`

The header renders as a small title strip at the top of the card, in the theme's card-header style. Leaving it empty gives you a headerless card — useful for the header / status row banding the top of the display, but unusual elsewhere.

Section 4 — The TrendChart recipe

Most dashboard cells end up being trend charts. The canonical setup:

```
{ "Type": "TrendChart", "Duration": "5m", "YMinValue": 0, "YMaxValue": 100, "YLabels": 5, "XGridLines": 6, "YGridLines": 5, "LegendPlacement": "BottomPanel", "VerticalCursor": true, "BackgroundTheme": "ControlBackground", "ForegroundTheme": "TextForeground", "BorderBrushTheme": "DefaultBorder", "Pens": [ { "Type": "TrendPen", "LinkedValue": "@Tag.Reactor/Temperature_C", "PenLabel": "Temperature", "Stroke": "#FFEF4444", "StrokeThickness": 2 }, { "Type": "TrendPen", "LinkedValue": "@Tag.Reactor/Setpoint", "PenLabel": "Setpoint", "Stroke": "#FF34D399", "StrokeThickness": 1 } ] }
```

Pens accepts BOTH forms

- Flat array: `"Pens": [ {...}, {...} ]` prefer this for readability
- Wrapper object: `"Pens": { "Type": "TrendPenList", "Children": [ {...} ] }` older form, also accepted

TrendPen properties

Property	Required	Notes
`LinkedValue`	?	Tag binding: `@Tag.Reactor/Temperature_C`
`PenLabel`	—	Legend text
`Stroke`	—	Line color hex. **Not theme-aware — hex only.**
`StrokeThickness`	—	Default 1; use 2 for emphasis, 3 for setpoint/limit
`YMin` / `YMax`	—	Per-pen Y scale (overrides chart scale)
`YAxisPosition`	—	`"Left"` (default) or `"Right"` for dual-axis plots
`Auto`	—	`true` for auto-scaling that pen
`Visible`	—	`true`/`false`

Duration defaults

Duration	When
`"1m"`	Demos — populates visibly in ~60s
`"5m"`	Default operator view — recent activity
`"15m"`	Short-term process monitoring
`"1h"` / `"4h"`	Shift-length trends

`"24h"` / `"7d"`	Production reports, historical review

LegendPlacement

`"BottomPanel"` (default), `"RightPanel"`, `"TopPanel"`, `"None"`. For narrow cells use `"RightPanel"`. For wide cells `"BottomPanel"`.

Chart control selection

Call `list_elements('Charts')` for the authoritative chart catalog in the current release. `TrendChart` is the right default for time-series data. For X-vs-Y scatter plots or correlation displays, verify the current element set before committing — the available chart types and their schemas evolve between releases.

Section 5 — The DataGrid listdetail pattern

The pattern in one sentence

Use a UserType-typed Client tag as the "selected row" carrier. The DataGrid pushes selected-row values into its fields, detail controls read from its fields via `@Tag.Selected<X>.Field`. Zero code-behind.

The setup

1. Create a Client-scope tag typed by your row's UDT. For a reactor list: `@Tag.SelectedReactor` with `UserType: "ReactorRow"`.
2. In the DataGrid, set `SelectedValuesLink` to `"@Tag.SelectedReactor"`. The DataGrid automatically pushes values from the currently-selected row into each matching field of the UDT.
3. Detail controls on the same display read `@Tag.SelectedReactor.Name`, `@Tag.SelectedReactor.Temperature`, etc. They update automatically as the operator clicks rows.

DataGrid definition

```
{ "Type": "DataGrid", "ItemsSource": "@Dataset.Query.ActiveReactors", "SelectedValuesLink": "@Tag.
SelectedReactor", "Columns": { "Type": "GridColumnList", "Children": [ { "Type": "GridColumn", "Title":
"Reactor", "FieldName": "Name", "Width": 120 }, { "Type": "GridColumn", "Title": "Temp (°C)", "FieldName":
"Temperature", "Width": 100 }, { "Type": "GridColumn", "Title": "Pressure", "FieldName": "Pressure", "Width": 100
}, { "Type": "GridColumn", "Title": "Status", "FieldName": "Status", "Width": 100 } ] }, "BackgroundTheme":
"ControlBackground", "ForegroundTheme": "TextForeground", "BorderBrushTheme": "DefaultBorder" }
```

The detail cells (sibling cells on the same display)

```
{
  "Row": 1, "Col": 1, "Cell.HeaderLink": "Temperature",
  "Content": {
    "Type": "CenterValue",
    "LinkedValue": "@Tag.SelectedReactor.Temperature",
    "CenterTextFormat": "N1",
    "AccentTextLink": "°C"
  }
},
{
  "Row": 1, "Col": 2, "Cell.HeaderLink": "Pressure",
  "Content": {
    "Type": "CenterValue",
    "LinkedValue": "@Tag.SelectedReactor.Pressure",
    "CenterTextFormat": "N1",
    "AccentTextLink": "bar"
  }
},
{
  "Row": 2, "Col": 1, "ColSpan": 2, "Cell.HeaderLink": "Status",
  "Content": {
    "Type": "TextBlock",
    "LinkedValue": "Status: {@Tag.SelectedReactor.Status}",
    "FontSize": 16
  }
}
```

When the operator clicks a row in the DataGrid, all three detail cells update simultaneously. No script, no event handler.

Requirements

- The UDT's member names must match the DataGrid column `FieldNames` (case-sensitive).
- The Client tag must be typed by that UDT.
- The dataset query must return rows whose column names match the UDT members.

Section 6 — ComboBox zero-script FK-lookup

```
{
  "Type": "ComboBox",
  "ItemsSourceType": "DataTable",
  "ItemsSourceLink": "@Dataset.Query.OperatorsList",
  "DisplayMember": "FullName",
  "SelectedValuePath": "OperatorID",
  "SelectedValueLink": "@Tag.Shift/CurrentOperatorId",
  "Foreground": "theme:TextForeground",
  "Background": "theme:ControlBackground",
  "BorderBrush": "theme:DefaultBorder"
}
```

When the operator picks "Maria Costa" from the dropdown, the OperatorID (say 42) lands in `@Tag.Shift/CurrentOperatorId` automatically. Other displays binding to that tag update immediately.

Use this for: operator selection, product changeover, reactor mode select, batch selection — anywhere the user picks one row from a database-backed list.

ItemsSourceType options

- `"DataTable"` — the pattern above, dataset query as source
- `"Array"` — bound to an array tag
- `"StringTag"` — comma-separated string (legacy, avoid)
- `"Text"` — hardcoded comma-separated inline options

Section 7 — AlarmViewer (the default template is your friend)

The AlarmViewer default template ships pre-wired to `@Client.AlarmPage.*` context tags:

```
{ "Type": "AlarmViewer", "ShowRowSelectorPane": false, "BackgroundTheme": "ControlBackground", "ForegroundTheme":
"TextForeground" }
```

That's the entire cell content. All alarms, all columns, all features — wired.

Custom filtering

```
{ "Type": "AlarmViewer", "Filter": "Area = 'ReactorZone'", "ShowRowSelectorPane": false }
```

Filter syntax: `"Priority >= 2"`, `"Area = 'Tank1'"`, or a tag binding `"@Tag.AlarmFilter"`.

Custom columns

```
"Columns": { "Type": "GridColumnList", "Children": [ { "Type": "GridColumn", "Title": "Active", "FieldName":
"ActiveTime_Ticks", "Width": 140 }, { "Type": "GridColumn", "Title": "Tag", "FieldName": "TagName", "Width": 200
}, { "Type": "GridColumn", "Title": "Msg", "FieldName": "Message", "Width": 400 }, { "Type": "GridColumn",
"Title": "Pri", "FieldName": "Priority", "Width": 50 } ] }
```

Available field names: `AckStatus`, `ActiveTime_Ticks`, `TagName`, `Group`, `Value`, `ID`, `ItemName`, `State`, `AckRequired`, `Condition`, `SolutionName`, `Area`, `Priority`, `NormTime_Ticks`, `AckTime_Ticks`, `UserName`, `Message`, `Duration`, `Category`, `DateCreated_Ticks`, `AuxValue`, `AlarmLimit`, `PreviousValue`, `AuxValue2`, `AuxValue3`. Tick fields format as DateTime at render time.

Section 8 — AssetsTree + ChildDisplay navigation

The canonical asset-driven master-detail pattern. One "detail template" display shows whatever asset the operator picks in the tree.

The tree (drop it in, no config)

```
{ "Type": "AssetsTree" }
```

The default template already binds:

- `LinkedValue @Client.Context.AssetName` (output — which asset is selected)
- `AssetPathLink @Client.Context.AssetPath` (output — full path)

The detail panel — ChildDisplay

```
{ "Type": "ChildDisplay", "DisplayLink": "EquipmentDetailTemplate" }
```

Inside `EquipmentDetailTemplate`, every binding uses `Asset(@Client.Context.AssetPath + ".Property")` or direct `@Tag` paths constructed from the context. When the operator clicks a different tree node, the ChildDisplay re-renders with the new asset's data.

Dynamic ChildDisplay

```
{ "Type": "ChildDisplay", "DisplayLink": "@Tag.DetailDisplayName" }
```

A Script calculates which detail display to show based on the selected asset's type (Pump PumpDetail, Reactor ReactorDetail) and writes to `@Tag.DetailDisplayName`. The ChildDisplay swaps automatically.

Section 9 — Carousel and TabControl

Both accept the same structure: `HeaderElements` (navigation chrome) and `TabItems` (panels). Difference:

- **Carousel** — `AutoCycleLink: 5` for 5-second auto-cycling. Lobby displays, rotating KPI boards.
- **TabControl** — no auto-cycle; operator clicks tabs. Drill-down detail panels.

```
{
  "Type": "Carousel",
  "AutoCycleLink": 5,
  "TabItems": [
    {
      "Type": "TabItem",
      "IsSelected": true,
      "Header": { "Type": "TextBlock", "LinkedValue": "Production" },
      "Children": [ { "Type": "TrendChart", ... } ]
    },
    {
      "Type": "TabItem",
      "Header": { "Type": "TextBlock", "LinkedValue": "Quality" },
      "Children": [ { "Type": "BarChart", ... } ]
    },
    {
      "Type": "TabItem",
      "Header": { "Type": "TextBlock", "LinkedValue": "Alarms" },
      "Children": [ { "Type": "AlarmViewer" } ]
    }
  ]
}
```

Rules:

- Only ONE TabItem should have `IsSelected: true`
- `Children` is an array — can contain multiple elements per tab
- `AutoCycleLink` can be a number (seconds) OR a tag binding (`"@Tag.ShowroomCycleSeconds"`) for runtime-configurable cycling

Section 10 — Common KPI card recipes

Big-number CenterValue

```
{ "Type": "CenterValue", "LinkedValue": "@Tag.Plant/ProductionRate", "CenterTextFormat": "N1", "AccentTextLink":
"units/hr", "CenterFontSize": 48, "AccentFontSize": 14, "BackgroundTheme": "ControlBackground" }
```

`CenterTextFormat`: `"N0"` (integer), `"N1"` (1 decimal), `"N2"` (2 decimals), `"P1"` (percent 1 decimal). `AccentTextLink` is the unit or caption under /beside the main number.

Composite TextBlock (value + unit + context in one)

```
{ "Type": "TextBlock", "LinkedValue": "Throughput: {@Tag.Plant/ProductionRate} units/hr ({@Tag.Plant/TargetPct}%
of target)", "FontSize": 16, "FontWeight": "SemiBold", "ForegroundTheme": "TextForeground" }
```

Threshold-colored value

```
{ "Type": "TextBlock", "LinkedValue": "{@Tag.Plant/AvgTemp} °C", "FontSize": 32, "Dynamics": [ { "Type":
"TextColorDynamic", "LinkedValue": "@Tag.Plant/AvgTemp", "ChangeColorItems": [ { "Type": "ChangeColorItem",
"ChangeLimit": 0, "LimitColor": "#FF38BDF8" }, { "Type": "ChangeColorItem", "ChangeLimit": 75, "LimitColor":
"#FF34D399" }, { "Type": "ChangeColorItem", "ChangeLimit": 90, "LimitColor": "#FFF59E0B" }, { "Type":
"ChangeColorItem", "ChangeLimit": 100, "LimitColor": "#FFEF4444" } ] } ] }
```

Blue cold green normal amber warning red alarm.

Section 11 — Navigation on Dashboard displays

Same rule as Canvas: **page-to-page navigation belongs in the Header display**, not on content pages. Content pages get only in-page interactions.

Tab-bar header pattern

```
{ "Name": "Header", "PanelType": "Dashboard", "DashboardDisplay": { "Columns": ["Auto","Auto","Auto","Auto","*","
Auto"], "Rows": ["*"], "Cells": [ { "Row": 0, "Col": 0, "Content": { "Type": "Button", "LabelLink": "Overview",
"Dynamics": [{"Type":"ActionDynamic","MouseLeftButtonDown":{"Type":"DynamicActionInfo","ActionType":"
OpenDisplay","ObjectLink":"OperationsOverview"}}] } }, { "Row": 0, "Col": 1, "Content": { "Type": "Button",
"LabelLink": "Alarms", ... } }, { "Row": 0, "Col": 2, "Content": { "Type": "Button", "LabelLink": "Trends", ... }
}, { "Row": 0, "Col": 3, "Content": { "Type": "Button", "LabelLink": "Reports", ... } }, { "Row": 0, "Col": 5,
"Content": { "Type": "TextBlock", "LinkedValue": "Logged in: {@Client.Username}" } } ] } }
```

Minimum button size: 100×32. Recommended: 130×40 for touch-friendly control rooms.

Row-click drill-down

For DataGrids, a double-click to drill into detail:

```
"Dynamics": [ { "Type": "ActionDynamic", "MouseDoubleClick": { "Type": "DynamicActionInfo", "ActionType":
"OpenDisplay", "ObjectLink": "ReactorDetail" } } ]
```

The target display reads `@Tag.SelectedReactor.Name` (etc.) — so you get "double-click row open detail view of that row" with zero code.

Section 12 — Dashboard-specific quirks

1. **Dynamics that don't work in Dashboard.** Silently ignored: `RotateDynamic`, `ScaleDynamic`, `MoveDragDynamic`, `SkewDynamic`, `Bargraph Dynamic`. If you need a rotating element or a custom level-bar, move that cell's content to a Canvas display embedded via `ChildDisplay`, or pick a control with the behavior built in (LinearGauge for level, symbols for motor-running-spin).
2. **Cell headers use theme styles you can't directly override.** `Cell.HeaderLink` renders in a theme-defined style — you can't set `FontSize` or `Foreground` on the header itself from the cell. If you need a custom header, set `Cell.HeaderLink` to empty string and put a `TextBlock` as the first element inside the cell Content.
3. **ComboBox + DataTable source loads on first render.** Expect a brief empty state on display open. For critical selection flows, bind to `@Tag.SelectedValueLink` with a pre-populated default.
4. **ChildDisplay depth limit.** Don't nest ChildDisplays more than 2 levels deep. Beyond that, performance degrades and context-tag reuse becomes confusing.

Section 13 — Dashboard checklist

- ? `PanelType: "Dashboard"` is set
- ? `OnResize` is set by display kind: `Responsive` for a data page whose prominent element should grow; `StretchUniform` for a process diagram/drawing; `ResizeChildren` for a viewer/text page; `Responsive` + `LockedHeight` for a docked Header/Footer (native Width matched to the Layout Width); `Responsive` + `LockedWidth` for a docked Menu/SubMenu rail; `NoAction` for a Popup/Dialog/Form (docked Header /Footer/Menu are NOT `NoAction`)
- ? On a `Responsive` display with positioned (non-grid) content, the prominent element carries the right element locks — `IsWidthAlign`/`IsHeightAlign` to stretch, `IsLeftAlign`/`IsTopAlign` to track the right/bottom edge; small top-left labels and nav icons carry none. A grid-driven Responsive Dashboard (Star columns + GridSplitter) needs no element locks
- ? Landing page follows the Home pattern — a `Responsive` Dashboard hosting two ChildDisplays (Star columns + GridSplitter)
- ? Any content page that must render on mobile is split into two child displays (one Canvas each), not one display forced to reflow to both form factors
- ? `DashboardDisplay` object includes `Columns`, `Rows`, `Cells`
- ? Every cell has `Row` and `Col`; `Cell.HeaderLink` is either a non-empty string or absent
- ? No Canvas-only element types (Rectangle, Ellipse, Polygon, Cylinder, ShapeGroup, SvgGroup, Group) in Content
- ? No Canvas-only dynamics (Rotate, Scale, MoveDrag, Skew, Bargraph) on cell content
- ? Text values 14 FontSize; big KPIs 22 FontSize
- ? ColSpan/RowSpan summed values don't exceed the grid
- ? AlarmViewer uses default template (no Columns override) unless you NEED custom columns
- ? DataGrid SelectedValuesLink UserType-typed Client tag detail cells binding chain verified
- ? ComboBox with DataTable source has `DisplayMember` AND `SelectedValuePath` AND `SelectedValueLink`
- ? TrendChart Pens are flat array form; each pen has at minimum `LinkedValue` + `Stroke`
- ? Header display owns all page-to-page navigation
- ? `get_state` after write shows `errorList` empty
- ? `get_display_tree(element=<DisplayName>)` returns — for each navigation Button or other control carrying a `LabelLink`, `labelResolved` matches the authored intent (no default caption left in place, no unexpected empty text)
- ? For every cell where a non-empty `Cell.HeaderLink` was authored, the resolved header text matches intent (the documented empty-string case for the headerless-card pattern is fine; the bug is the unintentionally-empty case)
- ? Resolved fill / foreground on every themed-color element — especially threshold-colored TextColorDynamic cells — falls in the theme palette for the current tag value (no hard-coded hex bleed-through, no resolution to a default outside the threshold set)

Section 14 — Quick reference

The 10 controls you'll actually use on most Dashboards

```
TextBlock // headers, composite KPIs, status text CenterValue // big-number KPI tiles TrendChart // all time-
series (the workhorse) BarChart // categorical comparison AlarmViewer // alarm list with default template
AssetsTree // plant navigation sidebar ChildDisplay // embedded detail panel DataGrid // list in list-to-detail
ComboBox // FK selector dropdown RadialGauge // circular gauge cells
```

Canonical dashboard envelope

```
{
  "Name": "MyDashboard",
  "PanelType": "Dashboard",
  "DashboardDisplay": {
    "Columns": ["*", "*", "*"],
    "Rows": ["Auto", "*"],
    "Cells": [
      { "Row": 0, "Col": 0, "ColSpan": 3, "Cell.HeaderLink": "Header",  "Content": { } },
      { "Row": 1, "Col": 0,               "Cell.HeaderLink": "Card 1",  "Content": { } },
      { "Row": 1, "Col": 1,               "Cell.HeaderLink": "Card 2",  "Content": { } },
      { "Row": 1, "Col": 2,               "Cell.HeaderLink": "Card 3",  "Content": { } }
    ]
  }
}
```

The minimal structural skeleton an AI session can paste-and-fill. Replace the empty `Content: { }` objects with the controls from Section 3 (TextBlock, CenterValue, TrendChart, AlarmViewer, AssetsTree, DataGrid, ComboBox, RadialGauge, ChildDisplay, BarChart). The header row spans all three columns; the body row gives equal width to three cards. Add more rows or vary track sizes via the patterns in §1.