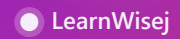


Production Drawing Architecture, Performance and Capstone

Lesson Guide · Lab / Exam Guide



Lesson Guide

Making drawing code shippable: one visual model with a pure geometry layer and separate renderers per surface, instrumentation for render time and payload size, profiling a painted path against an off-screen path, resize, theme, empty-data and missing-font behaviour, an accessible textual equivalent, and the Visual Operations Studio capstone. What this means for an experienced .NET developer who already builds Wisej.NET screens and now needs custom graphics — and how to read this module.



Production focus

Making drawing code shippable: one visual model with a pure geometry layer and separate renderers per surface, instrumentation for render time and payload size, profiling a painted path against an off-screen path, resize, theme, empty-data and missing-font behaviour, an accessible textual equivalent, and the Visual Operations Studio capstone.

1. The production problem

VisualOperationsStudio now owns four rendering surfaces, and each was built in its own module with its own private idea of what a machine's health means. That is how a drawing subsystem rots: somebody moves a warning threshold, three surfaces agree, and the exported PNG quietly contradicts the screen it came from. This module collapses those four interpretations into one model behind one geometry layer, then proves with measurements, deliberate failure paths and a semantic equivalent that the result survives real data, real platforms and real users.

2. Key concepts

Concept	Why it matters
Visual model	Semantic state only: readings, captions, nodes and edges, trend series, severity as a domain choice and the shared selection, so no surface needs a copy.
Geometry layer	Mostly pure maths turning model state plus the current surface size into rectangles, sweep angles, label positions and hit regions, testable with no control present.
Surface adapter	One renderer per surface rather than a common interface: <code>System.Drawing.Graphics</code> and <code>Wisej.Web.Canvas</code> keep their own primitives above a shared geometry.
Performance budget	A number agreed before tuning, such as a painted status control under 5 ms and a full Canvas redraw inside roughly 16 to 50 ms.
Render instrumentation	Recorded elapsed time, bitmap dimensions, drawn object count and payload growth per surface, so an optimisation is argued from a before and an after.
Golden-image test	A fixed-input off-screen render compared with a stored reference using a tolerance, because anti-aliasing and encoders differ across platforms.
Semantic equivalent	The label, <code>AccessibleDescription</code> , tooltip or companion list carrying every essential value the graphics show, so nothing important is pixels-only.

3. Lesson narrative

One model, one geometry layer, a renderer per surface

The reusable design has three layers. The visual model holds semantic state and nothing else. The geometry layer converts that state and the current viewport dimensions into rectangles, points, paths, label positions and hit regions. The renderer converts geometry into a surface, and only there does anything know whether it is talking to a `Graphics` object, a `Wisej.Web.Canvas` or an HTML fragment. One topology model can then be shown interactively on Canvas, exported to PNG through `System.Drawing.Managed`, and listed for accessibility without the domain rules existing three times.

Resist unifying the renderers themselves: their semantics genuinely differ, and an interface limited to what both can do costs more than it saves unless the visual is trivial. Share the model and the geometry instead. For VisualOperationsStudio the restructuring is mostly relocation: the gauge maths from Module 2 and the world-coordinate conversions from Module 5 move into `Geometry`, while the `Paint` handler, the `CellPaint` handler, the Canvas `Redraw` method and the off-screen renderer become thin adapters. Selection moves onto the model too, which is what lets a grid row click retarget the gauge and select the same Canvas node.

Budgets first, then instrumentation

Decide the numbers before you tune. Sensible starting budgets: a painted status control under 5 ms of server render at normal size, a user-painted grid column with no visible scrolling hitch at representative

row counts, a full Canvas redraw inside roughly 16 to 50 ms, and an export that may take longer because it is an explicit command but must not block unrelated sessions. What you record differs per surface:

- Server `Paint` and `CellPaint`: handler elapsed time, allocation rate, temporary image size, response payload growth and repaints per workflow.
- Canvas: drawing commands per redraw, scene payload size, redraw frequency, pointer round-trip latency and the browser's own cost on a large scene.
- Off-screen `System.Drawing.Managed`: bitmap dimensions and pixel memory, encode time, font-loading behaviour, output size and visual differences per platform.

```
var watch = System.Diagnostics.Stopwatch.StartNew();
using (var bitmap = new Bitmap(width, height))
using (var g = Graphics.FromImage(bitmap))
{
    _topologyRenderer.Render(g, scene, layout);
    watch.Stop();
    _metrics.Record("topology.export", watch.Elapsed,
        bitmap.Width, bitmap.Height, layout.VisibleNodes.Count);
}
```

With recording in place, profile one server-painted path against one off-screen path using the CPU Usage and allocation tools, plus microbenchmarks around the pure geometry. A handler that is fast for one cell is not automatically fast for a thousand, per-cell object graphs are the usual culprit, and concurrent sessions belong in the measurement rather than after it.

Two test matrices: platforms and browsers

Off-screen rendering must be verified on every platform you promise: a Windows modern .NET server, a Linux container, macOS for development and test where supported, and iOS or Android Hybrid if you ship them. Automate text metrics, required fonts, transparent images, gradients, anti-aliasing and image saving, backed by golden reference outputs compared with a tolerance rather than byte equality, because a different encoder or half a pixel of anti-aliasing is a platform difference and not a regression.

Canvas needs a second matrix, because the browser does the rendering: verify current Chromium, Firefox and Safari, watching fonts, compositing, shadows and touch gestures. Resize everything while testing, since `Redraw` must reconstruct the whole scene from the model and a `Paint` handler must recompute geometry from the current width and height. Layer the suite as the pack recommends: unit tests for normalisation, value-to-angle mapping, hit testing, zoom transforms and culling; golden-image tests for the off-screen renderer; integration tests for page render, redraw after resize, grid scrolling and theme changes; manual passes for touch, keyboard, screen readers, browser zoom and high-DPI.

Degrade instead of throwing, and never ship pixels-only

Draw the bad states on purpose. A missing image becomes a placeholder glyph plus a log entry, a missing font resolves through a defined fallback chain, empty data becomes a neutral "No data" visual, an invalid number is clamped or rejected before rendering, and an export failure surfaces as a message the user can act on instead of a blank image. Theming deserves the same care: avoid hard-coding a white background

and black text, accept colours as public properties or resolve them from the application's theme, and test a dark theme as well as a light one.

Every essential painted value also needs a semantic twin. A gauge gets a visible label such as "CPU 72%" together with an `AccessibleDescription`. A sparkline gets accessible text along the lines of "Trend: up 12% over 7 days". A Canvas topology gets a companion `TreeView` or `ListView` with keyboard selection. Severity expressed as colour needs a shape or a word beside it, because red and green alone do not reach everyone.

The Visual Operations Studio capstone

The capstone is one page with four coordinated views: the painted `TelemetryGauge` for the selected asset, the operations grid with its user-painted Health and Trend columns, the Canvas topology with selection, drag, pan and zoom, and an Export command producing a PNG through `System.Drawing.Managed`. Selection is shared, and the export is generated from the visual model by a `System.Drawing` renderer, never by trying to screenshot the browser.

Alongside the running application you submit the evidence: where each renderer executes and what crosses the network, proof that no per-session scene state sits in a process-wide static, your disposal and font policy, measurements for two rendering paths with one optimisation you can defend, a Windows plus Linux container verification plan with the iOS, Android and macOS implications, and the accessible alternative. Then walk the final design checklist and answer yes to each line, including whether the visual rebuilds from model state, whether the surface suits the update frequency, and whether high-frequency animation stays on the client.



Surface-first drawing habit

Choose the rendering surface before you write a single drawing call. Ask where the pixels are produced and what crosses the network: a control Paint handler runs on the server and ships an image, `CellPaint` does the same inside one grid cell, Canvas sends drawing commands the browser executes and keeps nothing across a resize, and `System.Drawing.Managed` produces image bytes with no control at all. Keep the semantic model in C# objects, make every renderer a deterministic function of that model, invalidate only when a visual dependency changed, and always leave a textual equivalent of anything important that exists only as pixels.



Learning objectives

By the end of this module you'll be able to:

- ✓ Build a shared visual model with separate `System.Drawing` and Canvas renderers.
- ✓ Profile drawing CPU, allocations, image sizes, grid scrolling and session behavior.
- ✓ Test cross-platform rendering and browser redraw behavior.
- ✓ Design accessibility, export, theming, responsive layout and error handling into custom visuals.
- ✓ Complete the Visual Operations Studio capstone.

Lab / Exam Guide

What you'll build in the hands-on lab, the suggested approach, and the required deliverables.



Hands-on lab

Complete the VisualOperationsStudio capstone by putting one model behind all four surfaces. Restructure the project into Models, Geometry, Renderers and Wisej adapter folders so the painted gauge, the user-painted grid cells, the Canvas topology and the PNG export all read the same OperationsModel and share the pure geometry layer, with no business state duplicated per renderer. Add a RenderMetrics helper that records render duration, output bitmap size and the number of scene objects drawn, and use it to compare the server-painted gauge against the off-screen renderer before and after one optimisation you choose and justify. Exercise the failure paths — empty data, a zero-sized surface, an invalid range, a missing font — so each degrades to a readable placeholder rather than an exception, add an accessible table that lists every value the graphics show, and write the capstone notes covering the architecture, the browser/server boundary and the test matrix.

Starting point

VisualOperationsStudio as you left it after Module 6: a `TelemetryGauge` control with its own gauge maths, an operations grid with `UserPaint` Health and Trend columns, a Canvas topology editor holding a per-session scene with hit testing, pan and zoom, and a `TopologyImageRenderer` that writes a PNG through `Graphics.FromImage`. Every surface works, but each keeps its own copy of thresholds, colours and node positions, nothing is measured, and a missing font or an empty data set still reaches the drawing code as an exception.

Suggested approach

- ✓ Create the `Models`, `Geometry`, `Renderers` and adapter folders first, then move one surface at a time onto a single `OperationsModel` that owns the assets, their readings, the topology nodes and edges, the severity thresholds and the shared selection. Delete each renderer's private copy as you go, and keep the scene state on the session rather than in a static field.
- ✓ Lift the coordinate maths out of the four renderers into the geometry layer so it takes model state plus a size and returns ring bounds, sweep angles, cell rectangles, node rectangles and label positions. Unit-test value-to-angle mapping, normalisation of an inverted range, hit testing and visible-region culling there, where no Wisej control is needed.
- ✓ Write `RenderMetrics` with a `Record` method taking a surface name, the elapsed `Stopwatch` time, the output width and height and the drawn object count, then call it from the gauge `Paint` adapter and from the off-screen renderer. Capture a baseline, apply one optimisation you can defend, and record the same numbers again.
- ✓ Add the degraded paths: return a neutral "No data" visual when the model is empty, return immediately on a zero-sized surface, clamp or reject an invalid range before drawing, and resolve fonts through a fallback chain so a missing family produces a readable placeholder rather than an exception. Wrap the export in `try / catch` so a failure reaches the user as a message instead of a blank image.

- ✓ Build the accessible table that lists every value the graphics show, give the gauge a visible reading plus an `AccessibleDescription`, then write `CapstoneNotes.md` with the layer diagram, where each renderer executes and what crosses the network, the test matrix for Windows, Linux containers and the Hybrid targets, and the before-and-after measurements.

Before you call the capstone finished, run the demo script end to end: change one telemetry value and watch all four surfaces agree, resize or recreate the Canvas and show that `Redraw` reconstructs it, scroll the grid and explain the user-painted versus HTML trade-off, drag and zoom a topology node and explain the inverse-coordinate hit testing, export a PNG and say out loud where `System.Drawing.Managed` is running, then present the measurements and the optimisation boundary you chose. The architecture note also owes one rejected alternative per surface, so be ready to say why the gauge is painted rather than themed, why the grid column is user-painted rather than HTML, why the topology is Canvas rather than a server-painted image, and why the export is off-screen rather than a browser screenshot.

Required deliverables

- ✓ Models, Geometry, Renderers and adapter layers with one `OperationsModel` driving all four surfaces
- ✓ `RenderMetrics` helper recording render duration, bitmap size and drawn object count per surface
- ✓ Before-and-after measurements for one optimisation, with the reasoning for choosing it
- ✓ Failure paths for empty data, zero size, invalid range and a missing font that degrade instead of throwing
- ✓ `CapstoneNotes.md` with the architecture, the server/browser boundary, the test matrix and an accessible data table

Student review questions

- ✓ Your geometry layer takes the model and a size and returns shapes, yet one teammate wants it to return a `GraphicsPath` so the gauge renderer gets shorter. What does that cost the Canvas renderer and the unit tests, and where would you draw the line instead?
- ✓ The export PNG and the on-screen Canvas disagree by a few pixels on the Linux build agent but match on Windows. Which measurements and which kind of test would tell you whether that is a defect or an expected platform difference?
- ✓ A reviewer says the accessible table is redundant because every gauge already has a tooltip. Explain what the table gives a keyboard-only or screen-reader user that the tooltip does not, and what that implies for the colour-only severity chips in the grid.



Surface-first drawing habit

Choose the rendering surface before you write a single drawing call. Ask where the pixels are produced and what crosses the network: a control Paint handler runs on the server and ships an image, CellPaint does the same inside one grid cell, Canvas sends drawing commands the browser executes and keeps nothing across a resize, and System.Drawing.Managed produces image bytes with no control at all. Keep the semantic model in C# objects, make every renderer a deterministic function of that model, invalidate only when a visual dependency changed, and always leave a textual equivalent of anything important that exists only as pixels.