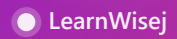


System.Drawing.Managed Across Windows, Linux, macOS, iOS and Android

Lesson Guide · Lab / Exam Guide



Lesson Guide

Off-screen rendering that runs everywhere: why System.Drawing.Common is Windows-only on modern .NET, what the Managed.System.Drawing package and System.Drawing.Managed assembly actually are, the Wisej.NET 4 font and graphics matrix, deterministic font policy, and choosing between it, ImageSharp, SkiaSharp, Aspose.Drawing and Microsoft.Maui.Graphics by use case. 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

Off-screen rendering that runs everywhere: why System.Drawing.Common is Windows-only on modern .NET, what the Managed.System.Drawing package and System.Drawing.Managed assembly actually are, the Wisej.NET 4 font and graphics matrix, deterministic font policy, and choosing between it, ImageSharp, SkiaSharp, Aspose.Drawing and Microsoft.Maui.Graphics by use case.

1. The production problem

Everything VisualOperationsStudio has drawn so far belongs to a live session: the gauge repaints because a control asked for it, and the Canvas topology from Module 5 exists only as commands the browser has already executed. Operations now wants that same topology in a nightly incident e-mail and attached to a ticket, which means an image with no control behind it. At the same time the application is moving into a Linux container, so the drawing code has to keep working on a host with no GDI+ and no `libgdiplus`.

2. Key concepts

Concept	Why it matters
<code>System.Drawing.Common</code>	Microsoft's GDI+ wrapper, Windows-specific from .NET 6; the Unix switch went in .NET 7, so it is no option for Linux, macOS, iOS or Android.
<code>Managed.System.Drawing</code>	The NuGet package id, reversed because the <code>System.*</code> package root is reserved; Wisej.NET 4 already references it for you.
<code>System.Drawing.Managed.dll</code>	The assembly that package delivers — the managed implementation itself, seen in a build log rather than in your source.
The <code>System.Drawing</code> namespace	Unchanged: <code>Bitmap</code> , <code>Graphics</code> , <code>Pen</code> and <code>Font</code> keep their names, so existing drawing code compiles without a rewrite.
Managed graphics engine	A pure managed implementation on a customized, integrated ImageSharp engine; on modern .NET it avoids GDI+, libgdiplus and native calls.
<code>Graphics.FromImage</code>	The off-screen entry point: a drawing surface over a <code>Bitmap</code> you own, with no control, no Paint event and no session.
<code>PrivateFontCollection</code>	Loads font files the application ships, so output depends on what you deployed, not on what happens to be installed on the host.

3. Lesson narrative

Why System.Drawing.Common stops at the Windows boundary

Modern .NET changed the assumptions around `System.Drawing`. Microsoft marked `System.Drawing.Common` as Windows-specific beginning with .NET 6; the temporary Unix-support switch that release offered was removed in .NET 7. The official guidance for cross-platform applications is to move to a different library altogether — SkiaSharp, ImageSharp, Aspose.Drawing or Microsoft.Maui.Graphics.

That guidance is reasonable for a new codebase and painful for an existing one. Wisej.NET had a specific version of the problem: enterprise applications and third-party controls are full of `System.Drawing` code, and the framework itself needs image loading and font measurement to lay out a screen at all. Rewriting every customer's drawing code was not viable, and neither was a Windows-only framework.

What Managed.System.Drawing actually is

Wisej.NET 4 introduced `System.Drawing.Managed`, a managed, cross-platform implementation of the familiar `System.Drawing` namespace. Three names travel together. The NuGet **package** is `Managed.System.Drawing`, that way round because the `System.*` package root is reserved. The **assembly** it delivers is `System.Drawing.Managed.dll`. The **namespace** your code writes against is still `System.Drawing`. None of this is Microsoft's `System.Drawing.Common`, and confusing the two is the most expensive mistake in this module.

The Wisej documentation describes the implementation as pure managed, built on a customized and integrated ImageSharp engine. On modern .NET that means no GDI+, no libgdiplus and no platform-

specific native calls. On .NET Framework it is deliberately hybrid: font measurement uses the managed path while the imaging layer can still wrap native GDI+ for compatibility and performance. The goal is API compatibility with classic System.Drawing plus predictable cross-platform image and font behaviour, not a new API to learn.

Drawing without a control: Bitmap, Graphics and a PNG

The lab's `TopologyImageRenderer` is the shape every off-screen renderer should have: a plain model and an output size in, bytes out. It creates its own `Bitmap`, obtains a surface with `Graphics.FromImage`, clears it, draws edges, nodes, labels and a legend with `Pen`, `SolidBrush`, `GraphicsPath` and `Font`, then saves to a `MemoryStream` with `ImageFormat.Png`. Because it never touches a Wisej control, the same class runs from a page, a unit test or a background service.

```
public byte[] Render(TopologyScene scene, int width, int height)
{
    using var bitmap = new Bitmap(width, height);
    using var g = Graphics.FromImage(bitmap);
    using var font = ResolveFont(11f);
    g.Clear(Color.White);

    foreach (var edge in scene.Edges) DrawEdge(g, edge);
    foreach (var node in scene.Nodes) DrawNode(g, node, font);

    using var stream = new MemoryStream();
    bitmap.Save(stream, ImageFormat.Png);
    return stream.ToArray();
}
```

Ownership is the rule that keeps this safe. Dispose every `Bitmap`, `Graphics`, `Image`, `Pen`, `Brush`, `Font`, `GraphicsPath`, `Matrix` and stream the renderer itself created, and leave the caller's objects alone — the opposite of a Paint handler, where the `Graphics` is lent to you. Size matters too: an uncompressed four-byte-per-pixel bitmap of 4000×3000 needs roughly 48 MB for its pixel buffer alone, so render at the size the output actually needs.

Fonts are the other half of determinism, and cross-platform font metrics are one of the strongest reasons Wisej.NET uses the managed library at all. `Font`, `FontFamily` and `PrivateFontCollection` are all available, and a `PrivateFontCollection` loads font files the application ships. Before writing `ResolveFont`, settle the policy: which families are required, whether they are installed everywhere or shipped, what the fallback is, whether the licence permits server embedding, and whether the browser needs the same font.

The Wisej.NET 4 platform matrix and the namespace rule

The managed-graphics documentation gives a small conceptual matrix worth memorising, because it explains behaviour you will otherwise find surprising in a designer session:

- **Windows .NET Framework** — managed font metrics, GDI+ graphics.
- **Modern .NET, all platforms** — managed fonts, managed graphics.
- **.NET Core Visual Studio Designer build** — managed fonts, GDI+ graphics.

Recommended deployment targets include `net8.0` for Windows, Linux and macOS server deployments, plus `net8.0-ios`, `net8.0-android`, `net8.0-maccatalyst` and the Windows Hybrid target; newer TFMs work where the Wisej version supports them. The number is not the lesson. The lesson is not to force a `-windows` suffix onto a cross-platform server deployment unless a genuinely Windows-specific dependency requires it, because that one edit quietly ends the portability you just paid for.

Namespace collisions are the matching hazard. Both implementations publish types in the same namespace, so a project that adds its own `System.Drawing.Common` reference beside Wisej's managed one is asking the compiler to choose between two `Bitmap` types. Remove it when it is unnecessary; when a third-party package drags in a conflicting version, use explicit package aliasing as the Wisej documentation describes. Be equally honest about the limits: the concepts documentation calls out gaps around some advanced imaging effects and region operations, and metafiles, device-context handles and printer contexts do not map to a pure managed implementation at all. Isolate and redesign code built on Win32 handles, HDC interop or printing rather than assuming it travels.

The operational payoff on Linux is concrete: libgdiplus stops being a container prerequisite and a class of native dependency failures disappears — though you still ship the fonts and still test memory for large bitmaps. On iOS and Android, where `System.Drawing.Common` was never supported at all, the managed library is what makes shared drawing code practical for Wisej.NET Hybrid; verify font availability, file paths, image encoders and memory pressure on real devices.

Choosing by workload, not by popularity

`System.Drawing.Managed`'s advantage inside a Wisej.NET solution is compatibility and integration, not superiority. It is the right answer when you are reusing `System.Drawing` code, feeding `Paint` and `CellPaint` handlers, or generating server images from business data. `ImageSharp` is a modern managed image-processing pipeline with strong format handling and a licensing model to review. `SkiaSharp` brings broad, high-performance 2D capability and native Skia assets. `Aspose.Drawing` offers managed GDI+-like compatibility as a commercial product. `Microsoft.Maui.Graphics` is a drawing abstraction for MAUI codebases, not a `System.Drawing` drop-in.

One more decision belongs here, because off-screen rendering is where images arrive from outside. Treat an untrusted image as untrusted input: bound the upload size, the dimensions and the decoded pixel count, restrict accepted formats, decode large images away from request-critical paths, and never let a user-supplied filename escape the storage path you control.



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:

- ✓ Explain why `System.Drawing.Common` is Windows-only on modern .NET.
- ✓ Understand the architecture and package model of `System.Drawing.Managed`.
- ✓ Use `Bitmap`, `Graphics`, `Font`, `Image`, brushes, pens, paths and image saving cross-platform.
- ✓ Understand Wisej.NET 4's platform matrix and namespace-collision guidance.
- ✓ Compare `System.Drawing.Managed` with `ImageSharp`, `SkiaSharp`, `Aspose.Drawing` and `Microsoft.Maui.Graphics` by use case rather than popularity.

Lab / Exam Guide

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

Hands-on lab

Add a `TopologyImageRenderer` to `VisualOperationsStudio` that turns the topology model into a PNG without touching a `Wisej` control. Give it a `Render` method taking the scene and an output size, create the `Bitmap`, obtain its `Graphics` with `Graphics.FromImage`, draw nodes, edges, labels and a legend with pens, brushes and a `GraphicsPath`, and dispose everything it owns while leaving the caller's objects alone. Resolve its `Font` through a `ResolveFont` method that tries an application-supplied family and falls back to a known-present one instead of assuming a system font exists, save to a `MemoryStream` as PNG and hand the bytes to the page for download. Then check the project references to confirm you are not adding `System.Drawing.Common` alongside `Wisej`'s managed implementation, run or stage the renderer on Windows and in a Linux container, and write up how the same code behaves on iOS, Android and macOS Hybrid targets.

Starting point

`VisualOperationsStudio` after Module 5: the topology page owns a `TopologyScene` with `NodeModel` and `EdgeModel` objects in world coordinates, and `RenderScene` draws it onto a `Wisej.Web.Canvas` that the browser repaints from commands. Nothing in the project can produce an image file, and every line of drawing code written so far needs a live control and a live session to run.

Suggested approach

- ✓ Create `TopologyImageRenderer` in a folder that has no `Wisej.Web` using directive at all, and give it one public method that takes a `TopologyScene` plus a width and height and returns `byte[]`. If you cannot call it from a plain console test, it is still coupled to the UI.
- ✓ Inside `Render`, create the `Bitmap`, take its surface with `Graphics.FromImage`, clear it, then draw edges, node shapes, labels and a legend using `Pen`, `SolidBrush`, `GraphicsPath` and `DrawString`. Wrap every object the method creates in `using`, and pass the scene in rather than reaching for page state.
- ✓ Write `ResolveFont` with an explicit chain: try the family the application ships through a `PrivateFontCollection`, then a family you know is present on every target, and record which one was used so the output is explainable. Never let the renderer throw because a host is missing a font.
- ✓ Save the bitmap to a `MemoryStream` with `ImageFormat.Png`, return `stream.ToArray()`, and from the page hand those bytes to `Application.Download` behind an Export button. Time one render and note the byte count, because both numbers come back in Module 7.
- ✓ Open the project file and confirm `Managed.System.Drawing` is what is in play and no direct `System.Drawing.Common` reference sits beside it, check that the target framework has not been pinned to a `-windows` variant, then run the exporter on Windows and again in a Linux container and compare the two PNGs pixel for pixel.

Required deliverables

- ✓ TopologyImageRenderer with no dependency on a Wisej control, rendering through Graphics.FromImage into a Bitmap
- ✓ ResolveFont method with an explicit fallback chain instead of an assumed system font
- ✓ PNG written to a MemoryStream and downloaded from the page, with every owned graphics object disposed
- ✓ Reference check confirming Managed.System.Drawing is in use and System.Drawing.Common is not being added alongside it
- ✓ CrossPlatform.md with Windows plus Linux container evidence and the iOS, Android and macOS implications

Student review questions

- ✓ A colleague says the renderer proves the app is cross-platform because the code compiles on Linux. What would you actually have to observe in the container before you accepted that claim, and which part of the output is most likely to differ?
- ✓ Your renderer disposes the `Bitmap` it created. Why would disposing the `Graphics` inside a control's Paint handler be a bug, and what rule covers both cases in one sentence?
- ✓ Someone proposes replacing this renderer with SkiaSharp because it benchmarks faster. What would you need to know about the workload before agreeing, and what does the team lose by leaving the System.Drawing API behind?



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.