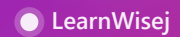


System.Drawing.Managed unter Windows, Linux, macOS, iOS und Android

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Off-Screen-Rendering, das überall läuft: warum System.Drawing.Common im modernen .NET nur unter Windows funktioniert, was das Paket Managed.System.Drawing und die Assembly System.Drawing.Managed tatsächlich sind, die Schrift- und Grafikmatrix von Wisej.NET 4, eine deterministische Schriftenstrategie und die Wahl zwischen dieser Lösung, ImageSharp, SkiaSharp, Aspose.Drawing und Microsoft.Maui.Graphics je nach Anwendungsfall. Was das für erfahrene .NET-Entwickler bedeutet, die bereits Wisej.NET-Oberflächen bauen und nun eigene Grafiken brauchen – und wie Sie dieses Modul lesen.



Fokus: Produktivbetrieb

Off-Screen-Rendering, das überall läuft: warum System.Drawing.Common im modernen .NET nur unter Windows funktioniert, was das Paket Managed.System.Drawing und die Assembly System.Drawing.Managed tatsächlich sind, die Schrift- und Grafikmatrix von Wisej.NET 4, eine deterministische Schriftenstrategie und die Wahl zwischen dieser Lösung, ImageSharp, SkiaSharp, Aspose.Drawing und Microsoft.Maui.Graphics je nach Anwendungsfall.

1. Das Problem im Produktivbetrieb

Alles, was VisualOperationsStudio bisher gezeichnet hat, gehört zu einer lebenden Sitzung: Das Gauge zeichnet sich neu, weil ein Control darum gebeten hat, und die Canvas-Topologie aus Modul 5 existiert nur als Befehle, die der Browser bereits ausgeführt hat. Der Betrieb möchte dieselbe Topologie nun in einer nächtlichen Störungs-E-Mail und als Anhang an einem Ticket – und das bedeutet ein Bild ohne Control dahinter. Gleichzeitig zieht die Anwendung in einen Linux-Container um, der Zeichencode muss also auf einem Host weiterlaufen, auf dem es weder GDI+ noch `libgdiplus` gibt.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
<code>System.Drawing.Common</code>	Microsofts GDI+-Hülle, ab .NET 6 Windows-spezifisch; der Unix-Schalter fiel mit .NET 7 weg, also ist sie für Linux, macOS, iOS oder Android keine Option.
<code>Managed.System.Drawing</code>	Die NuGet-Paket-ID, umgedreht, weil der Paketstamm <code>System.*</code> reserviert ist; Wisej.NET 4 referenziert sie bereits für Sie.
<code>System.Drawing.Managed.dll</code>	Die Assembly, die dieses Paket ausliefert — die verwaltete Implementierung selbst, die Ihnen eher im Buildprotokoll als in Ihrem Quellcode begegnet.
Der Namespace <code>System.Drawing</code>	Unverändert: <code>Bitmap</code> , <code>Graphics</code> , <code>Pen</code> und <code>Font</code> behalten ihre Namen, sodass vorhandener Zeichencode ohne Umschreiben kompiliert.
Verwaltete Grafik-Engine	Eine rein verwaltete Implementierung auf einer angepassten, integrierten ImageSharp-Engine; im modernen .NET vermeidet sie GDI+, libgdiplus und native Aufrufe.
<code>Graphics.FromImage</code>	Der Einstiegspunkt fürs Off-Screen-Rendering: eine Zeichenoberfläche über einer <code>Bitmap</code> , die Ihnen gehört – ohne Control, ohne Paint-Event, ohne Sitzung.
<code>PrivateFontCollection</code>	Lädt Schriftdateien, die die Anwendung mitbringt, sodass die Ausgabe davon abhängt, was Sie ausgeliefert haben, und nicht davon, was zufällig auf dem Host installiert ist.

3. Der rote Faden der Lektion

Warum System.Drawing.Common an der Windows-Grenze endet

Das moderne .NET hat die Annahmen rund um `System.Drawing` verändert. Microsoft hat `System.Drawing.Common` ab .NET 6 als Windows-spezifisch gekennzeichnet; der befristete Schalter für Unix-Unterstützung aus jenem Release wurde in .NET 7 entfernt. Die offizielle Empfehlung für plattformübergreifende Anwendungen lautet, gleich auf eine andere Bibliothek umzusteigen — SkiaSharp, ImageSharp, Aspose.Drawing oder Microsoft.Maui.Graphics.

Für eine neue Codebasis ist diese Empfehlung vernünftig und für eine vorhandene schmerzhaft. Wisej.NET hatte eine besondere Ausprägung des Problems: Unternehmensanwendungen und Fremd-Controls stecken voller System.Drawing-Code, und das Framework selbst braucht das Laden von Bildern und die Schriftvermessung, um eine Oberfläche überhaupt anzuordnen. Den Zeichencode jedes Kunden umzuschreiben war ebenso wenig machbar wie ein Framework nur für Windows.

Was Managed.System.Drawing tatsächlich ist

Wisej.NET 4 hat `System.Drawing.Managed` eingeführt, eine verwaltete, plattformübergreifende Implementierung des vertrauten Namespace `System.Drawing`. Drei Namen reisen gemeinsam. Das NuGet-Paket heißt `Managed.System.Drawing`, in dieser Reihenfolge, weil der Paketstamm `System.*` reserviert ist. Die **Assembly**, die es ausliefert, heißt `System.Drawing.Managed.dll`. Der **Namespace**,

gegen den Ihr Code geschrieben wird, ist weiterhin `System.Drawing`. Nichts davon ist Microsofts `System.Drawing.Common`, und die beiden zu verwechseln ist der teuerste Fehler in diesem Modul.

Die Wisej-Dokumentation beschreibt die Implementierung als rein verwaltet, aufgebaut auf einer angepassten und integrierten ImageSharp-Engine. Im modernen .NET heißt das: kein GDI+, kein libgdiplus, keine plattformspezifischen nativen Aufrufe. Auf .NET Framework ist sie bewusst hybrid: Die Schriftvermessung nutzt den verwalteten Weg, während die Bildschicht aus Kompatibilitäts- und Leistungsgründen weiterhin natives GDI+ umhüllen kann. Ziel ist API-Kompatibilität zum klassischen System.Drawing plus vorhersagbares plattformübergreifendes Bild- und Schriftverhalten – keine neue API, die man lernen muss.

Zeichnen ohne Control: Bitmap, Graphics und ein PNG

Der `TopologyImageRenderer` aus dem Lab hat genau die Form, die jeder Off-Screen-Renderer haben sollte: ein schlichtes Modell und eine Ausgabegröße hinein, Bytes heraus. Er erzeugt seine eigene `Bitmap`, bezieht eine Oberfläche mit `Graphics.FromImage`, löscht sie, zeichnet Kanten, Knoten, Beschriftungen und eine Legende mit `Pen`, `SolidBrush`, `GraphicsPath` und `Font` und speichert dann mit `ImageFormat.Png` in einen `MemoryStream`. Weil er nie ein Wisej-Control anfasst, läuft dieselbe Klasse aus einer Seite, einem Unit-Test oder einem Hintergrunddienst.

```
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();
}
```

Der Besitz ist die Regel, die das sicher hält. Geben Sie jede `Bitmap`, jedes `Graphics`, `Image`, jeden `Pen`, `Brush`, `Font`, `GraphicsPath`, jede `Matrix` und jeden Stream frei, den der Renderer selbst erzeugt hat, und lassen Sie die Objekte des Aufrufers in Ruhe — das Gegenteil eines Paint-Handlers, in dem Ihnen das `Graphics` geliehen wird. Auch die Größe zählt: Eine unkomprimierte Bitmap mit vier Byte pro Pixel und 4000×3000 Pixeln braucht allein für ihren Puffer rund 48 MB – rendern Sie also in der Größe, die die Ausgabe wirklich braucht.

Schriften sind die andere Hälfte der Determiniertheit, und plattformübergreifende Schriftmetriken sind einer der stärksten Gründe dafür, dass Wisej.NET die verwaltete Bibliothek überhaupt einsetzt. `Font`, `FontFamily` und `PrivateFontCollection` stehen alle zur Verfügung, und eine `PrivateFontCollection` lädt Schriftdateien, die die Anwendung mitliefert. Bevor Sie `ResolveFont` schreiben, klären Sie die Strategie: welche Familien erforderlich sind, ob sie überall installiert oder mitgeliefert werden, wie der

Rückfall aussieht, ob die Lizenz das Einbetten auf dem Server erlaubt und ob der Browser dieselbe Schrift braucht.

Die Plattformmatrix von Wisej.NET 4 und die Namespace-Regel

Die Dokumentation zu verwalteten Grafiken liefert eine kleine begriffliche Matrix, die man sich merken sollte, weil sie Verhalten erklärt, das einem in einer Designer-Sitzung sonst überraschend vorkommt:

- **Windows .NET Framework** — verwaltete Schriftmetriken, GDI+-Grafik.
- **Modernes .NET, alle Plattformen** — verwaltete Schriften, verwaltete Grafik.
- **.NET-Core-Build des Visual Studio Designers** — verwaltete Schriften, GDI+-Grafik.

Zu den empfohlenen Zielplattformen gehören `net8.0` für Server-Deployments unter Windows, Linux und macOS sowie `net8.0-ios`, `net8.0-android`, `net8.0-maccatalyst` und das Windows-Hybrid-Ziel; neuere TFMs funktionieren, wo die Wisej-Version sie unterstützt. Die Zahl ist nicht die Lektion. Die Lektion lautet: Erzwingen Sie kein Suffix `-windows` an einem plattformübergreifenden Server-Deployment, solange nicht eine wirklich Windows-spezifische Abhängigkeit es verlangt – denn diese eine Änderung beendet still die Portabilität, für die Sie gerade bezahlt haben.

Namespace-Konflikte sind die dazu passende Gefahr. Beide Implementierungen veröffentlichen Typen im selben Namespace, also verlangt ein Projekt, das neben der verwalteten Variante von Wisej einen eigenen Verweis auf `System.Drawing.Common` hinzufügt, vom Compiler eine Wahl zwischen zwei `Bitmap`-Typen. Entfernen Sie ihn, wenn er unnötig ist; zieht ein Fremdpaket eine konfliktträchtige Version herein, verwenden Sie ausdrückliches Package Aliasing, wie es die Wisej-Dokumentation beschreibt. Seien Sie ebenso ehrlich über die Grenzen: Die Konzeptdokumentation nennt Lücken bei einigen fortgeschrittenen Bildeffekten und Region-Operationen, und Metadateien, Gerätekontext-Handles und Druckerkontexte lassen sich auf eine rein verwaltete Implementierung überhaupt nicht abbilden. Isolieren und überarbeiten Sie Code, der auf Win32-Handles, HDC-Interop oder Druck aufbaut, statt anzunehmen, dass er mitreist.

Der betriebliche Gewinn unter Linux ist greifbar: libgdiplus ist keine Voraussetzung für den Container mehr, und eine ganze Klasse nativer Abhängigkeitsfehler verschwindet — auch wenn Sie die Schriften weiterhin mitliefern und bei großen Bitmaps weiterhin den Speicher testen. Auf iOS und Android, wo `System.Drawing.Common` nie unterstützt wurde, macht die verwaltete Bibliothek gemeinsamen Zeichencode für Wisej.NET Hybrid überhaupt erst praktikabel; prüfen Sie Schriftverfügbarkeit, Dateipfade, Bildkodierer und Speicherdruck auf echten Geräten.

Nach Aufgabe wählen, nicht nach Beliebtheit

Der Vorteil von `System.Drawing.Managed` innerhalb einer Wisej.NET-Lösung liegt in Kompatibilität und Integration, nicht in Überlegenheit. Es ist die richtige Antwort, wenn Sie `System.Drawing`-Code wiederverwenden, `Paint`- und `CellPaint`-Handler versorgen oder Serverbilder aus Geschäftsdaten erzeugen. `ImageSharp` ist eine moderne verwaltete Bildverarbeitungs-Pipeline mit starker Formatunterstützung und einem Lizenzmodell, das man prüfen sollte. `SkiaSharp` bringt breite, leistungsstarke 2D-Fähigkeiten und native Skia-Bestandteile mit. `Aspose.Drawing` bietet als kommerzielles Produkt verwaltete GDI+-ähnliche Kompatibilität. `Microsoft.Maui.Graphics` ist eine Zeichenabstraktion für MAUI-Codebasen und kein Ersatz für `System.Drawing`.

Eine weitere Entscheidung gehört hierher, denn beim Off-Screen-Rendering kommen Bilder von außen herein. Behandeln Sie ein nicht vertrauenswürdige Bild als nicht vertrauenswürdige Eingabe: Begrenzen Sie Uploadgröße, Abmessungen und dekodierte Pixelanzahl, schränken Sie die akzeptierten Formate ein, dekodieren Sie große Bilder abseits der zeitkritischen Anfragepfade und lassen Sie einen vom Benutzer gelieferten Dateinamen niemals aus dem Speicherpfad ausbrechen, den Sie kontrollieren.



Gewohnheit: erst die Oberfläche, dann das Zeichnen

Wählen Sie die Rendering-Oberfläche, bevor Sie einen einzigen Zeichenaufruf schreiben. Fragen Sie, wo die Pixel entstehen und was über das Netzwerk geht: Der Paint-Handler eines Controls läuft auf dem Server und liefert ein Bild aus, CellPaint tut dasselbe innerhalb einer Grid-Zelle, Canvas schickt Zeichenbefehle, die der Browser ausführt, und behält über eine Größenänderung hinweg nichts, und System.Drawing.Managed erzeugt Bild-Bytes ganz ohne Control. Halten Sie das semantische Modell in C#-Objekten, machen Sie jeden Renderer zu einer deterministischen Funktion dieses Modells, invalidieren Sie nur, wenn sich eine visuelle Abhängigkeit geändert hat, und lassen Sie immer ein textliches Äquivalent für alles Wichtige zurück, das nur als Pixel existiert.



Lernziele

Am Ende dieses Moduls können Sie:

- ✓ Erklären, warum System.Drawing.Common im modernen .NET nur unter Windows funktioniert.
- ✓ Die Architektur und das Paketmodell von System.Drawing.Managed verstehen.
- ✓ Bitmap, Graphics, Font, Image, Brushes, Pens, Paths und das Speichern von Bildern plattformübergreifend einsetzen.
- ✓ Die Plattformmatrix von Wisej.NET 4 und die Hinweise zu Namespace-Konflikten verstehen.
- ✓ System.Drawing.Managed mit ImageSharp, SkiaSharp, Aspose.Drawing und Microsoft.Maui.Graphics nach Anwendungsfall statt nach Beliebtheit vergleichen.

Lab- und Prüfungsleitfaden

Was Sie im praktischen Lab bauen, das vorgeschlagene Vorgehen und die geforderten Ergebnisse.



Praktisches Lab

Ergänzen Sie VisualOperationsStudio um einen `TopologyImageRenderer`, der das Topologiemodell in ein PNG verwandelt, ohne ein einziges Wisej-Control anzufassen. Geben Sie ihm eine Render-Methode, die Szene und Ausgabegröße entgegennimmt, die Bitmap erzeugt, deren Graphics über `Graphics.FromImage` bezieht, Knoten, Kanten, Beschriftungen und eine Legende mit Pens, Brushes und einem `GraphicsPath` zeichnet und alles freigibt, was ihm gehört, während die Objekte des Aufrufers unangetastet bleiben. Lösen Sie seinen Font über eine Methode `ResolveFont` auf, die eine von der Anwendung mitgelieferte Schriftfamilie versucht und auf eine garantiert vorhandene zurückfällt, statt eine Systemschrift vorauszusetzen; speichern Sie als PNG in einen `MemoryStream` und übergeben Sie die Bytes zum Download an die Seite. Prüfen Sie anschließend die Projektverweise, um sicherzustellen, dass Sie `System.Drawing.Common` nicht neben der Managed-Implementierung von Wisej hinzufügen, führen Sie den Renderer unter Windows und in einem Linux-Container aus oder bereiten Sie das vor, und beschreiben Sie, wie sich derselbe Code auf iOS-, Android- und macOS-Hybrid-Zielen verhält.

Ausgangspunkt

VisualOperationsStudio nach Modul 5: Die Topologieseite besitzt eine `TopologyScene` mit `NodeModel` - und `EdgeModel` -Objekten in Weltkoordinaten, und `RenderScene` zeichnet sie auf eine `Wisej.Web.Canvas`, die der Browser aus Befehlen neu zeichnet. Nichts im Projekt kann eine Bilddatei erzeugen, und jede bisher geschriebene Zeile Zeichencode braucht zum Laufen ein lebendes Control und eine lebende Sitzung.

Vorgeschlagenes Vorgehen

- ✓ Erstellen Sie `TopologyImageRenderer` in einem Ordner, der überhaupt keine using-Direktive auf `Wisej.Web` enthält, und geben Sie ihm eine öffentliche Methode, die eine `TopologyScene` plus Breite und Höhe entgegennimmt und `byte[]` zurückgibt. Wenn Sie sie nicht aus einem schlichten Konsolentest aufrufen können, hängt sie noch an der UI.
- ✓ Erzeugen Sie in `Render` die `Bitmap`, holen Sie ihre Oberfläche mit `Graphics.FromImage`, löschen Sie sie und zeichnen Sie dann Kanten, Knotenformen, Beschriftungen und eine Legende mit `Pen`, `SolidBrush`, `GraphicsPath` und `DrawString`. Setzen Sie jedes von der Methode erzeugte Objekt in ein `using` und übergeben Sie die Szene, statt auf den Zustand der Seite zuzugreifen.
- ✓ Schreiben Sie `ResolveFont` mit einer ausdrücklichen Kette: zuerst die Familie versuchen, die die Anwendung über eine `PrivateFontCollection` mitliefert, danach eine Familie, von der Sie wissen, dass sie auf jedem Ziel vorhanden ist, und festhalten, welche verwendet wurde, damit die Ausgabe erklärbar bleibt. Lassen Sie den Renderer niemals eine Exception auslösen, weil einem Host eine Schrift fehlt.
- ✓ Speichern Sie die Bitmap mit `ImageFormat.Png` in einen `MemoryStream`, geben Sie `stream.ToArray()` zurück und übergeben Sie diese Bytes von der Seite aus hinter einem

Export-Button an `Application.Download`. Stoppen Sie die Zeit eines Renderings und notieren Sie die Byteanzahl, denn beide Zahlen kommen in Modul 7 zurück.

- ✓ Öffnen Sie die Projektdatei und prüfen Sie, dass `Managed.System.Drawing` im Spiel ist und kein direkter Verweis auf `System.Drawing.Common` danebensteht, kontrollieren Sie, dass das Zielframework nicht auf eine `-windows`-Variante festgenagelt wurde, führen Sie den Export dann unter Windows und danach in einem Linux-Container aus und vergleichen Sie die beiden PNGs Pixel für Pixel.

Geforderte Ergebnisse

- ✓ TopologyImageRenderer ohne Abhängigkeit von einem Wisej-Control, der über `Graphics.FromImage` in eine Bitmap rendert
- ✓ Methode `ResolveFont` mit einer expliziten Fallback-Kette statt einer vorausgesetzten Systemschrift
- ✓ PNG in einen `MemoryStream` geschrieben und von der Seite heruntergeladen, wobei jedes eigene Grafikobjekt freigegeben wird
- ✓ Verweisprüfung, die bestätigt, dass `Managed.System.Drawing` verwendet und `System.Drawing.Common` nicht danebengestellt wird
- ✓ `CrossPlatform.md` mit Belegen von Windows und aus dem Linux-Container sowie den Folgen für iOS, Android und macOS

Wiederholungsfragen

- ✓ Ein Kollege sagt, der Renderer beweise die Plattformunabhängigkeit der Anwendung, weil der Code unter Linux kompiliert. Was müssten Sie im Container tatsächlich beobachten, bevor Sie diese Behauptung akzeptieren, und welcher Teil der Ausgabe weicht am wahrscheinlichsten ab?
- ✓ Ihr Renderer gibt die selbst erzeugte `Bitmap` frei. Warum wäre es ein Fehler, das `Graphics` im `Paint-Handler` eines Controls freizugeben, und welche Regel deckt beide Fälle in einem Satz ab?
- ✓ Jemand schlägt vor, diesen Renderer durch `SkiaSharp` zu ersetzen, weil es im Benchmark schneller ist. Was müssten Sie über die Aufgabe wissen, bevor Sie zustimmen, und was verliert das Team, wenn es die `System.Drawing-API` hinter sich lässt?



Gewohnheit: erst die Oberfläche, dann das Zeichnen

Wählen Sie die Rendering-Oberfläche, bevor Sie einen einzigen Zeichenaufruf schreiben. Fragen Sie, wo die Pixel entstehen und was über das Netzwerk geht: Der `Paint-Handler` eines Controls läuft auf dem Server und liefert ein Bild aus, `CellPaint` tut dasselbe innerhalb einer Grid-Zelle, `Canvas` schickt Zeichenbefehle, die der Browser ausführt, und behält über eine Größenänderung hinweg nichts, und `System.Drawing.Managed` erzeugt Bild-Bytes ganz ohne Control. Halten Sie das semantische Modell in C#-Objekten, machen Sie jeden Renderer zu einer deterministischen Funktion dieses Modells, invalidieren Sie nur, wenn sich eine visuelle Abhängigkeit geändert hat, und lassen Sie immer ein textliches Äquivalent für alles Wichtige zurück, das nur als Pixel existiert.