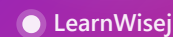


System.Drawing.Managed su Windows, Linux, macOS, iOS e Android

Guida della lezione · Guida al lab / esame



Guida della lezione

Rendering off-screen che gira ovunque: perché System.Drawing.Common è solo per Windows sulle versioni moderne di .NET, che cosa sono davvero il package Managed.System.Drawing e l'assembly System.Drawing.Managed, la matrice di font e grafica di Wisej.NET 4, una politica deterministica sui font e come scegliere, caso per caso, fra questa strada, ImageSharp, SkiaSharp, Aspose.Drawing e Microsoft.Maui.Graphics. Che cosa significa per uno sviluppatore .NET esperto che già costruisce schermate Wisej.NET e ora ha bisogno di grafica personalizzata — e come affrontare questo modulo.



Focus di produzione

Rendering off-screen che gira ovunque: perché System.Drawing.Common è solo per Windows sulle versioni moderne di .NET, che cosa sono davvero il package Managed.System.Drawing e l'assembly System.Drawing.Managed, la matrice di font e grafica di Wisej.NET 4, una politica deterministica sui font e come scegliere, caso per caso, fra questa strada, ImageSharp, SkiaSharp, Aspose.Drawing e Microsoft.Maui.Graphics.

1. Il problema in produzione

Tutto quello che VisualOperationsStudio ha disegnato finora appartiene a una sessione viva: l'indicatore si ridisegna perché un controllo l'ha chiesto, e la topologia sul Canvas del Modulo 5 esiste soltanto come comandi che il browser ha già eseguito. Ora operations vuole quella stessa topologia dentro una e-mail notturna sugli incidenti e allegata a un ticket, il che significa un'immagine senza alcun controllo dietro. Nel frattempo l'applicazione si sta spostando in un container Linux, quindi il codice di disegno deve continuare a funzionare su un host senza GDI+ e senza `libgdiplus`.

2. Concetti chiave

Concetto	Perché è importante
<code>System.Drawing.Common</code>	Il wrapper di GDI+ di Microsoft, specifico per Windows a partire da .NET 6; l'interruttore per Unix è sparito in .NET 7, quindi non è un'opzione per Linux, macOS, iOS o Android.
<code>Managed.System.Drawing</code>	L'id del package NuGet, invertito perché la radice <code>System.*</code> dei package è riservata; Wisej.NET 4 lo riferenzia già al posto tuo.
<code>System.Drawing.Managed.dll</code>	L'assembly consegnato da quel package — l'implementazione managed vera e propria, che vedi in un log di build più che nel tuo sorgente.
Il namespace <code>System.Drawing</code>	Invariato: <code>Bitmap</code> , <code>Graphics</code> , <code>Pen</code> e <code>Font</code> mantengono i loro nomi, quindi il codice di disegno esistente compila senza riscritture.
Motore grafico managed	Un'implementazione puramente managed su un motore ImageSharp personalizzato e integrato; sulle versioni moderne di .NET evita GDI+, libgdiplus e le chiamate native.
<code>Graphics.FromImage</code>	Il punto di ingresso off-screen: una superficie di disegno sopra un <code>Bitmap</code> che possiedi, senza controllo, senza evento Paint e senza sessione.
<code>PrivateFontCollection</code>	Carica i file di font distribuiti con l'applicazione, così l'output dipende da ciò che hai rilasciato e non da ciò che capita sia installato sull'host.

3. Il filo della lezione

Perché System.Drawing.Common si ferma al confine di Windows

Le versioni moderne di .NET hanno cambiato i presupposti attorno a `System.Drawing`. Microsoft ha marcato `System.Drawing.Common` come specifico per Windows a partire da .NET 6; l'interruttore temporaneo di supporto a Unix che quella release offriva è stato rimosso in .NET 7. L'indicazione ufficiale per le applicazioni multiplatforma è passare del tutto a un'altra libreria — SkiaSharp, ImageSharp, Aspose.Drawing o Microsoft.Maui.Graphics.

È un'indicazione ragionevole per una base di codice nuova e dolorosa per una esistente. Wisej.NET aveva una versione tutta sua del problema: le applicazioni enterprise e i controlli di terze parti sono pieni di codice System.Drawing, e il framework stesso ha bisogno di caricare immagini e misurare font anche solo per disporre una schermata. Riscrivere il codice di disegno di ogni cliente non era praticabile, e nemmeno lo era un framework solo per Windows.

Che cos'è davvero Managed.System.Drawing

Wisej.NET 4 ha introdotto `System.Drawing.Managed`, un'implementazione managed e multiplatforma del familiare namespace `System.Drawing`. Tre nomi viaggiano insieme. Il **package** NuGet è `Managed.System.Drawing`, in quest'ordine perché la radice `System.*` dei package è riservata. L'**assembly** che consegna è `System.Drawing.Managed.dll`. Il **namespace** contro cui scrivi il tuo codice resta

`System.Drawing`. Nulla di tutto questo è il `System.Drawing.Common` di Microsoft, e confondere i due è l'errore più costoso di questo modulo.

La documentazione di Wisej descrive l'implementazione come puramente managed, costruita su un motore ImageSharp personalizzato e integrato. Sulle versioni moderne di .NET questo significa niente GDI+, niente libgdiplus e nessuna chiamata nativa specifica della piattaforma. Su .NET Framework è ibrida di proposito: la misurazione dei font passa per la strada managed, mentre il livello di imaging può ancora avvolgere GDI+ nativo per compatibilità e prestazioni. L'obiettivo è la compatibilità di API con il classico System.Drawing più un comportamento prevedibile di immagini e font su tutte le piattaforme, non una nuova API da imparare.

Disegnare senza un controllo: Bitmap, Graphics e un PNG

Il `TopologyImageRenderer` del lab ha la forma che ogni renderer off-screen dovrebbe avere: in ingresso un modello semplice e una dimensione di output, in uscita dei byte. Crea il proprio `Bitmap`, ne ottiene una superficie con `Graphics.FromImage`, la pulisce, disegna archi, nodi, etichette e una legenda con `Pen`, `SolidBrush`, `GraphicsPath` e `Font`, poi salva in un `MemoryStream` con `ImageFormat.Png`. Poiché non tocca mai un controllo Wisej, la stessa classe gira da una pagina, da uno unit test o da un servizio in background.

```
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();
}
```

La proprietà degli oggetti è la regola che tiene tutto questo al sicuro. Fai il dispose di ogni `Bitmap`, `Graphics`, `Image`, `Pen`, `Brush`, `Font`, `GraphicsPath`, `Matrix` e stream creato dal renderer stesso, e lascia in pace gli oggetti del chiamante — l'opposto di un handler di Paint, dove il `Graphics` ti viene prestato. Conta anche la dimensione: un bitmap non compresso a quattro byte per pixel di 4000×3000 richiede all'incirca 48 MB solo per il buffer dei pixel, quindi rendi alla dimensione che l'output richiede davvero.

I font sono l'altra metà del determinismo, e le metriche dei font multiplatforma sono una delle ragioni più forti per cui Wisej.NET usa la libreria managed. `Font`, `FontFamily` e `PrivateFontCollection` sono tutti disponibili, e una `PrivateFontCollection` carica i file di font distribuiti con l'applicazione. Prima di scrivere `ResolveFont`, fissa la politica: quali famiglie sono obbligatorie, se sono installate ovunque oppure distribuite con l'applicazione, qual è il fallback, se la licenza consente l'incorporazione lato server e se al browser serve lo stesso font.

La matrice delle piattaforme di Wisej.NET 4 e la regola sul namespace

La documentazione sulla grafica managed offre una piccola matrice concettuale che vale la pena memorizzare, perché spiega comportamenti che altrimenti in una sessione del designer ti sorprenderebbero:

- **Windows .NET Framework** — metriche dei font managed, grafica GDI+.
- **.NET moderno, tutte le piattaforme** — font managed, grafica managed.
- **Build .NET Core del Visual Studio Designer** — font managed, grafica GDI+.

Fra i target di deployment raccomandati ci sono `net8.0` per i rilasci server su Windows, Linux e macOS, più `net8.0-ios`, `net8.0-android`, `net8.0-maccatalyst` e il target Windows Hybrid; i TFM più recenti funzionano dove la versione di Wisej li supporta. Il numero non è la lezione. La lezione è non forzare un suffisso `-windows` su un deployment server multiplatforma, a meno che non lo imponga una dipendenza davvero specifica di Windows, perché quell'unica modifica pone fine in silenzio alla portabilità che hai appena pagato.

Le collisioni di namespace sono il rischio gemello. Entrambe le implementazioni pubblicano tipi nello stesso namespace, quindi un progetto che aggiunge un proprio riferimento a `System.Drawing.Common` accanto a quello managed di Wisej sta chiedendo al compilatore di scegliere fra due tipi `Bitmap`. Rimuovilo quando non è necessario; quando è un package di terze parti a trascinarsi dietro una versione in conflitto, usa l'aliasing esplicito del package come descrive la documentazione di Wisej. Sii altrettanto onesto sui limiti: la documentazione sui concetti segnala lacune attorno ad alcuni effetti di imaging avanzati e alle operazioni sulle region, e metafile, handle di device context e contesti di stampa non si mappano affatto su un'implementazione puramente managed. Isola e riprogetta il codice costruito su handle Win32, interop HDC o stampa, invece di dare per scontato che viaggi.

Il ritorno operativo su Linux è concreto: libgdiplus smette di essere un prerequisito del container e sparisce un'intera classe di guasti da dipendenze native — anche se i font li distribuisce comunque e la memoria per i bitmap grandi la collaudi comunque. Su iOS e Android, dove `System.Drawing.Common` non è mai stato supportato, è la libreria managed a rendere praticabile un codice di disegno condiviso per Wisej.NET Hybrid; verifica su dispositivi reali la disponibilità dei font, i percorsi dei file, gli encoder delle immagini e la pressione sulla memoria.

Scegliere in base al carico di lavoro, non alla popolarità

Il vantaggio di `System.Drawing.Managed` dentro una soluzione Wisej.NET è compatibilità e integrazione, non superiorità. È la risposta giusta quando stai riusando codice `System.Drawing`, alimentando handler di Paint e di CellPaint o generando immagini sul server a partire da dati di business. ImageSharp è una moderna pipeline managed di elaborazione delle immagini, con un'ottima gestione dei formati e un modello di licenza da esaminare. SkiaSharp porta una capacità 2D ampia e ad alte prestazioni e gli asset nativi di Skia. Aspose.Drawing offre una compatibilità managed simile a GDI+ come prodotto commerciale. Microsoft.Maui.Graphics è un'astrazione di disegno per basi di codice MAUI, non un sostituto diretto di `System.Drawing`.

Qui dentro sta anche un'altra decisione, perché il rendering off-screen è il punto in cui arrivano immagini dall'esterno. Tratta un'immagine non fidata come input non fidato: limita la dimensione dell'upload, le dimensioni in pixel e il numero di pixel decodificati, restringi i formati accettati, decodifica le immagini

grandi lontano dai percorsi critici per la richiesta, e non lasciare mai che un nome di file fornito dall'utente esca dal percorso di archiviazione che controlli.



La regola di scegliere prima la superficie

Scegli la superficie di rendering prima di scrivere una sola chiamata di disegno. Chiediti dove vengono prodotti i pixel e che cosa attraversa la rete: l'handler di Paint di un controllo gira sul server e spedisce un'immagine, CellPaint fa lo stesso dentro una singola cella di griglia, il Canvas invia comandi di disegno che il browser esegue e non conserva nulla attraverso un resize, e System.Drawing.Managed produce byte di immagine senza alcun controllo. Tieni il modello semantico in oggetti C#, rendi ogni renderer una funzione deterministica di quel modello, invalida solo quando è cambiata una dipendenza visuale e lascia sempre un equivalente testuale di tutto ciò che è importante ed esiste solo come pixel.



Obiettivi di apprendimento

Alla fine di questo modulo sarai in grado di:

- ✓ Spiegare perché System.Drawing.Common è solo per Windows sulle versioni moderne di .NET.
- ✓ Comprendere l'architettura e il modello di package di System.Drawing.Managed.
- ✓ Usare Bitmap, Graphics, Font, Image, brush, pen, path e il salvataggio delle immagini su tutte le piattaforme.
- ✓ Comprendere la matrice delle piattaforme di Wisej.NET 4 e le indicazioni sulle collisioni di namespace.
- ✓ Confrontare System.Drawing.Managed con ImageSharp, SkiaSharp, Aspose.Drawing e Microsoft.Maui.Graphics in base al caso d'uso anziché alla popolarità.

Guida al lab / esame

Che cosa costruirai nel lab pratico, l'approccio suggerito e i deliverable richiesti.

🎯 Lab pratico

Aggiungi a VisualOperationsStudio un `TopologyImageRenderer` che trasforma il modello di topologia in un PNG senza toccare un solo controllo Wisej. Dagli un metodo `Render` che riceve la scena e una dimensione di output, crea il `Bitmap`, ne ottiene il `Graphics` con `Graphics.FromImage`, disegna nodi, archi, etichette e una legenda con pen, brush e un `GraphicsPath`, e fa il dispose di tutto ciò che possiede lasciando in pace gli oggetti del chiamante. Risolvi il suo `Font` attraverso un metodo `ResolveFont` che prova una famiglia fornita dall'applicazione e ripiega su una sicuramente presente, invece di dare per scontato che un font di sistema esista; salva in un `MemoryStream` come PNG e passa i byte alla pagina per il download. Poi controlla i riferimenti del progetto per confermare che non stai aggiungendo `System.Drawing.Common` accanto all'implementazione managed di Wisej, esegui o prepara il renderer su Windows e in un container Linux, e metti per iscritto come si comporta lo stesso codice sui target iOS, Android e macOS Hybrid.

Punto di partenza

VisualOperationsStudio dopo il Modulo 5: la pagina di topologia possiede una `TopologyScene` con oggetti `NodeModel` ed `EdgeModel` in coordinate mondo, e `RenderScene` la disegna su un `Wisej.Web.Canvas` che il browser ridisegna a partire dai comandi. Nel progetto non c'è nulla in grado di produrre un file immagine, e ogni riga di codice di disegno scritta finora ha bisogno di un controllo vivo e di una sessione viva per essere eseguita.

Approccio suggerito

- ✓ Crea `TopologyImageRenderer` in una cartella che non abbia alcuna using directive di `Wisej.Web`, e dagli un solo metodo pubblico che riceve una `TopologyScene` più una larghezza e un'altezza e restituisce `byte[]`. Se non riesci a chiamarlo da un semplice test da console, è ancora accoppiato alla UI.
- ✓ Dentro `Render` crea il `Bitmap`, prendine la superficie con `Graphics.FromImage`, puliscila, poi disegna archi, forme dei nodi, etichette e una legenda usando `Pen`, `SolidBrush`, `GraphicsPath` e `DrawString`. Racchiudi in un `using` ogni oggetto creato dal metodo, e passa la scena come parametro invece di andare a pescare lo stato della pagina.
- ✓ Scrivi `ResolveFont` con una catena esplicita: prova la famiglia che l'applicazione distribuisce attraverso una `PrivateFontCollection`, poi una famiglia che sai essere presente su ogni target, e registra quale è stata usata così che l'output sia spiegabile. Non lasciare mai che il renderer sollevi un'eccezione perché su un host manca un font.
- ✓ Salva il bitmap in un `MemoryStream` con `ImageFormat.Png`, restituisci `stream.ToArray()` e dalla pagina passa quei byte ad `Application.Download` dietro un pulsante Export. Cronometra un rendering e annota il numero di byte, perché tutti e due i numeri tornano nel Modulo 7.
- ✓ Apri il file di progetto e conferma che in gioco ci sia `Managed.System.Drawing` e che non ci sia accanto alcun riferimento diretto a `System.Drawing.Common`, controlla che il target framework

non sia stato fissato a una variante `-windows`, poi esegui l'esportatore su Windows e di nuovo in un container Linux e confronta i due PNG pixel per pixel.

Deliverable richiesti

- ✓ `TopologyImageRenderer` senza alcuna dipendenza da un controllo Wisej, che rende tramite `Graphics.FromImage` dentro un `Bitmap`
- ✓ Metodo `ResolveFont` con una catena di fallback esplicita invece di un font di sistema dato per scontato
- ✓ PNG scritto in un `MemoryStream` e scaricato dalla pagina, con il `dispose` di ogni oggetto grafico posseduto
- ✓ Verifica dei riferimenti che conferma l'uso di `Managed.System.Drawing` e l'assenza di `System.Drawing.Common` affiancato
- ✓ `CrossPlatform.md` con le prove su Windows e in container Linux e le implicazioni per iOS, Android e macOS

Domande di autovalutazione

- ✓ Un collega sostiene che il renderer dimostra che l'applicazione è multiplatforma perché il codice compila su Linux. Che cosa dovresti osservare davvero nel container prima di accettare quell'affermazione, e quale parte dell'output è più probabile che differisca?
- ✓ Il tuo renderer fa il `dispose` del `Bitmap` che ha creato. Perché fare il `dispose` del `Graphics` dentro l'handler di `Paint` di un controllo sarebbe un bug, e quale regola copre entrambi i casi in una sola frase?
- ✓ Qualcuno propone di sostituire questo renderer con `SkiaSharp` perché nei benchmark è più veloce. Che cosa dovresti sapere del carico di lavoro prima di dire di sì, e che cosa perde il team abbandonando l'API `System.Drawing`?



La regola di scegliere prima la superficie

Scegli la superficie di rendering prima di scrivere una sola chiamata di disegno. Chiediti dove vengono prodotti i pixel e che cosa attraversa la rete: l'handler di `Paint` di un controllo gira sul server e spedisce un'immagine, `CellPaint` fa lo stesso dentro una singola cella di griglia, il `Canvas` invia comandi di disegno che il browser esegue e non conserva nulla attraverso un `resize`, e `System.Drawing.Managed` produce byte di immagine senza alcun controllo. Tieni il modello semantico in oggetti C#, rendi ogni renderer una funzione deterministica di quel modello, invalida solo quando è cambiata una dipendenza visuale e lascia sempre un equivalente testuale di tutto ciò che è importante ed esiste solo come pixel.