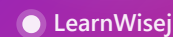


System.Drawing.Managed en Windows, Linux, macOS, iOS y Android

Guía de la lección · Guía del laboratorio / examen



Guía de la lección

Renderizado fuera de pantalla que funciona en todas partes: por qué System.Drawing.Common solo funciona en Windows en el .NET moderno, qué son realmente el paquete Managed.System.Drawing y el ensamblado System.Drawing.Managed, la matriz de fuentes y gráficos de Wisej.NET 4, una política de fuentes determinista, y cómo elegir entre él, ImageSharp, SkiaSharp, Aspose.Drawing y Microsoft.Maui.Graphics según el caso de uso. Qué significa esto para un desarrollador .NET con experiencia que ya construye pantallas Wisej.NET y ahora necesita gráficos personalizados, y cómo leer este módulo.



Enfoque de producción

Renderizado fuera de pantalla que funciona en todas partes: por qué System.Drawing.Common solo funciona en Windows en el .NET moderno, qué son realmente el paquete Managed.System.Drawing y el ensamblado System.Drawing.Managed, la matriz de fuentes y gráficos de Wisej.NET 4, una política de fuentes determinista, y cómo elegir entre él, ImageSharp, SkiaSharp, Aspose.Drawing y Microsoft.Maui.Graphics según el caso de uso.

1. El problema en producción

Todo lo que VisualOperationsStudio ha dibujado hasta ahora pertenece a una sesión viva: el indicador se redibuja porque un control lo ha pedido, y la topología en Canvas del Módulo 5 existe solo como comandos que el navegador ya ha ejecutado. Operaciones quiere ahora esa misma topología en un correo nocturno de incidencias y adjunta a un ticket, lo que significa una imagen sin ningún control detrás. Al mismo tiempo, la aplicación se está moviendo a un contenedor Linux, así que el código de dibujo tiene que seguir funcionando en un host sin GDI+ y sin `libgdiplus`.

2. Conceptos clave

Concepto	Por qué importa
<code>System.Drawing.Common</code>	El envoltorio de GDI+ de Microsoft, específico de Windows desde .NET 6; el conmutador para Unix desapareció en .NET 7, así que no es una opción para Linux, macOS, iOS ni Android.
<code>Managed.System.Drawing</code>	El identificador del paquete NuGet, invertido porque la raíz de paquetes <code>System.*</code> está reservada; Wisej.NET 4 ya lo referencia por ti.
<code>System.Drawing.Managed.dll</code>	El ensamblado que entrega ese paquete: la propia implementación gestionada, que se ve en un registro de compilación y no en tu código fuente.
El espacio de nombres <code>System.Drawing</code>	Sin cambios: <code>Bitmap</code> , <code>Graphics</code> , <code>Pen</code> y <code>Font</code> conservan sus nombres, así que el código de dibujo existente compila sin reescribirlo.
Motor de gráficos gestionado	Una implementación totalmente gestionada sobre un motor ImageSharp personalizado e integrado; en el .NET moderno evita GDI+, libgdiplus y las llamadas nativas.
<code>Graphics.FromImage</code>	El punto de entrada fuera de pantalla: una superficie de dibujo sobre un <code>Bitmap</code> que te pertenece, sin control, sin evento Paint y sin sesión.
<code>PrivateFontCollection</code>	Carga los archivos de fuente que distribuye la aplicación, de modo que la salida depende de lo que hayas desplegado y no de lo que resulte estar instalado en el host.

3. Desarrollo de la lección

Por qué System.Drawing.Common se detiene en la frontera de Windows

El .NET moderno cambió las premisas en torno a `System.Drawing`. Microsoft marcó `System.Drawing.Common` como específico de Windows a partir de .NET 6; el conmutador temporal de soporte para Unix que ofrecía esa versión se eliminó en .NET 7. La recomendación oficial para aplicaciones multiplataforma es pasarse directamente a otra biblioteca: SkiaSharp, ImageSharp, Aspose.Drawing o Microsoft.Maui.Graphics.

Esa recomendación es razonable para una base de código nueva y dolorosa para una existente. Wisej.NET tenía una versión concreta del problema: las aplicaciones empresariales y los controles de terceros están llenos de código de System.Drawing, y el propio framework necesita cargar imágenes y medir fuentes para poder disponer siquiera una pantalla. Reescribir el código de dibujo de todos los clientes no era viable, y un framework solo para Windows tampoco.

Qué es realmente Managed.System.Drawing

Wisej.NET 4 introdujo `System.Drawing.Managed`, una implementación gestionada y multiplataforma del conocido espacio de nombres `System.Drawing`. Tres nombres viajan juntos. El **paquete** de NuGet es `Managed.System.Drawing`, en ese orden porque la raíz de paquetes `System.*` está reservada. El

ensamblado que entrega es `System.Drawing.Managed.dll`. El **espacio de nombres** contra el que escribe tu código sigue siendo `System.Drawing`. Nada de esto es el `System.Drawing.Common` de Microsoft, y confundir ambos es el error más caro de este módulo.

La documentación de Wisej describe la implementación como totalmente gestionada, construida sobre un motor ImageSharp personalizado e integrado. En el .NET moderno eso significa nada de GDI+, nada de libgdiplus y nada de llamadas nativas específicas de plataforma. En .NET Framework es híbrida a propósito: la medición de fuentes usa la ruta gestionada mientras que la capa de imagen puede seguir envolviendo el GDI+ nativo por compatibilidad y rendimiento. El objetivo es la compatibilidad de API con el System.Drawing clásico más un comportamiento predecible de imágenes y fuentes entre plataformas, no una API nueva que aprender.

Dibujar sin un control: Bitmap, Graphics y un PNG

El `TopologyImageRenderer` del laboratorio tiene la forma que debería tener todo renderizador fuera de pantalla: entran un modelo corriente y un tamaño de salida, salen bytes. Crea su propio `Bitmap`, obtiene una superficie con `Graphics.FromImage`, la limpia, dibuja aristas, nodos, etiquetas y una leyenda con `Pen`, `SolidBrush`, `GraphicsPath` y `Font`, y después guarda en un `MemoryStream` con `ImageFormat.Png`. Como nunca toca un control de Wisej, esa misma clase se ejecuta desde una página, desde una prueba unitaria o desde un servicio en segundo plano.

```
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 propiedad de los recursos es la regla que mantiene esto a salvo. Libera todos los `Bitmap`, `Graphics`, `Image`, `Pen`, `Brush`, `Font`, `GraphicsPath`, `Matrix` y flujos que el propio renderizador haya creado, y deja en paz los objetos de quien lo llama: justo lo contrario que en un controlador de Paint, donde el `Graphics` te lo prestan. El tamaño también importa: un mapa de bits sin comprimir de cuatro bytes por píxel y 4000×3000 necesita unos 48 MB solo para su búfer de píxeles, así que dibuja en el tamaño que la salida necesite de verdad.

Las fuentes son la otra mitad del determinismo, y las métricas de fuente entre plataformas son una de las razones más fuertes por las que Wisej.NET usa la biblioteca gestionada. `Font`, `FontFamily` y `PrivateFontCollection` están todas disponibles, y una `PrivateFontCollection` carga los archivos de fuente que distribuye la aplicación. Antes de escribir `ResolveFont`, fija la política: qué familias son

obligatorias, si están instaladas en todas partes o se distribuyen, cuál es la alternativa, si la licencia permite incrustarlas en el servidor y si el navegador necesita la misma fuente.

La matriz de plataformas de Wisej.NET 4 y la regla del espacio de nombres

La documentación de gráficos gestionados ofrece una pequeña matriz conceptual que conviene memorizar, porque explica comportamientos que de otro modo te resultarán sorprendentes en una sesión del diseñador:

- **Windows con .NET Framework:** métricas de fuente gestionadas, gráficos GDI+.
- **.NET moderno, todas las plataformas:** fuentes gestionadas, gráficos gestionados.
- **Compilación del diseñador de Visual Studio sobre .NET Core:** fuentes gestionadas, gráficos GDI+.

Entre los destinos de despliegue recomendados están `net8.0` para despliegues de servidor en Windows, Linux y macOS, además de `net8.0-ios`, `net8.0-android`, `net8.0-maccatalyst` y el destino Hybrid de Windows; los TFM más nuevos funcionan allí donde la versión de Wisej los admite. El número no es la lección. La lección es no forzar un sufijo `-windows` en un despliegue de servidor multiplataforma salvo que lo exija una dependencia genuinamente específica de Windows, porque esa única modificación acaba en silencio con la portabilidad que acabas de pagar.

Las colisiones de espacios de nombres son el peligro asociado. Ambas implementaciones publican tipos en el mismo espacio de nombres, así que un proyecto que añade su propia referencia a

`System.Drawing.Common` junto a la gestionada de Wisej está pidiendo al compilador que elija entre dos tipos `Bitmap`. Quítala cuando sea innecesaria; cuando un paquete de terceros arrastre una versión conflictiva, usa alias de paquete explícitos como describe la documentación de Wisej. Sé igual de honesto sobre los límites: la documentación de conceptos señala carencias en algunos efectos de imagen avanzados y en las operaciones con regiones, y los metarchivos, los identificadores de contexto de dispositivo y los contextos de impresora no se corresponden en absoluto con una implementación totalmente gestionada. Aísla y rediseña el código construido sobre identificadores de Win32, interoperabilidad con HDC o impresión, en lugar de dar por supuesto que viaja.

La recompensa operativa en Linux es concreta: `libgdiplus` deja de ser un requisito previo del contenedor y desaparece toda una clase de fallos de dependencias nativas, aunque sigues distribuyendo las fuentes y sigues probando la memoria con mapas de bits grandes. En iOS y Android, donde `System.Drawing.Common` nunca tuvo soporte, la biblioteca gestionada es lo que hace práctico el código de dibujo compartido para Wisej.NET Hybrid; verifica en dispositivos reales la disponibilidad de fuentes, las rutas de archivo, los codificadores de imagen y la presión de memoria.

Elegir por carga de trabajo, no por popularidad

La ventaja de `System.Drawing.Managed` dentro de una solución Wisej.NET es la compatibilidad y la integración, no la superioridad. Es la respuesta correcta cuando reutilizas código de `System.Drawing`, cuando alimentas controladores de Paint y CellPaint o cuando generas imágenes en el servidor a partir de datos de negocio. `ImageSharp` es una canalización gestionada y moderna de procesamiento de imágenes, con un buen manejo de formatos y un modelo de licencia que conviene revisar. `SkiaSharp` aporta una capacidad 2D amplia y de alto rendimiento, con recursos nativos de Skia. `Aspose.Drawing` ofrece

compatibilidad gestionada al estilo de GDI+ como producto comercial. Microsoft.Maui.Graphics es una abstracción de dibujo para bases de código MAUI, no un reemplazo directo de System.Drawing.

Aquí corresponde una decisión más, porque el renderizado fuera de pantalla es el punto por el que llegan imágenes del exterior. Trata una imagen no confiable como entrada no confiable: acota el tamaño de la subida, las dimensiones y el número de píxeles decodificados, restringe los formatos aceptados, decodifica las imágenes grandes fuera de las rutas críticas de las peticiones y no dejes nunca que un nombre de archivo suministrado por el usuario se escape de la ruta de almacenamiento que tú controlas.



Regla de dibujo: primero la superficie

Elige la superficie de renderizado antes de escribir una sola llamada de dibujo. Pregúntate dónde se producen los píxeles y qué cruza la red: un controlador de Paint de un control se ejecuta en el servidor y envía una imagen, CellPaint hace lo mismo dentro de una celda de cuadrícula, Canvas envía comandos de dibujo que ejecuta el navegador y no conserva nada tras un redimensionado, y System.Drawing.Managed produce bytes de imagen sin ningún control de por medio. Mantén el modelo semántico en objetos C#, haz que cada renderizador sea una función determinista de ese modelo, invalida solo cuando haya cambiado una dependencia visual, y deja siempre un equivalente textual de todo lo importante que exista únicamente como píxeles.



Objetivos de aprendizaje

Al final de este módulo podrás:

- ✓ Explicar por qué System.Drawing.Common solo funciona en Windows en el .NET moderno.
- ✓ Entender la arquitectura y el modelo de paquetes de System.Drawing.Managed.
- ✓ Usar Bitmap, Graphics, Font, Image, brushes, pens, paths y el guardado de imágenes de forma multiplataforma.
- ✓ Entender la matriz de plataformas de Wisej.NET 4 y las indicaciones sobre colisiones de espacios de nombres.
- ✓ Comparar System.Drawing.Managed con ImageSharp, SkiaSharp, Aspose.Drawing y Microsoft.Maui.Graphics según el caso de uso y no según su popularidad.

Guía del laboratorio / examen

Lo que construirás en el laboratorio práctico, el enfoque sugerido y los entregables requeridos.



Laboratorio práctico

Añade a VisualOperationsStudio un `TopologyImageRenderer` que convierta el modelo de topología en un PNG sin tocar ningún control de Wisej. Dale un método `Render` que reciba la escena y un tamaño de salida, cree el `Bitmap`, obtenga su `Graphics` con `Graphics.FromImage`, dibuje nodos, aristas, etiquetas y una leyenda con pens, brushes y un `GraphicsPath`, y libere todo lo que le pertenece sin tocar los objetos de quien lo llama. Resuelve su `Font` mediante un método `ResolveFont` que pruebe una familia proporcionada por la aplicación y recurra a otra que sepas presente, en lugar de dar por hecho que existe una fuente del sistema; guarda el resultado como PNG en un `MemoryStream` y entrega los bytes a la página para su descarga. Después revisa las referencias del proyecto para confirmar que no estás añadiendo `System.Drawing.Common` junto a la implementación gestionada de Wisej, ejecuta o prepara el renderizador en Windows y en un contenedor Linux, y documenta cómo se comporta el mismo código en destinos iOS, Android y macOS Hybrid.

Punto de partida

VisualOperationsStudio después del Módulo 5: la página de topología es dueña de un `TopologyScene` con objetos `NodeModel` y `EdgeModel` en coordenadas de mundo, y `RenderScene` lo dibuja sobre un `Wisej.Web.Canvas` que el navegador repinta a partir de comandos. Nada en el proyecto puede producir un archivo de imagen, y cada línea de código de dibujo escrita hasta ahora necesita un control vivo y una sesión viva para ejecutarse.

Enfoque sugerido

- ✓ Crea `TopologyImageRenderer` en una carpeta sin ninguna directiva `using` de `Wisej.Web`, y dale un único método público que reciba un `TopologyScene` más un ancho y un alto y devuelva `byte[]`. Si no puedes llamarlo desde una prueba de consola corriente, sigue acoplado a la interfaz.
- ✓ Dentro de `Render`, crea el `Bitmap`, toma su superficie con `Graphics.FromImage`, límpiala y después dibuja aristas, formas de nodo, etiquetas y una leyenda usando `Pen`, `SolidBrush`, `GraphicsPath` y `DrawString`. Envuelve en `using` todos los objetos que cree el método, y pásale la escena en lugar de echar mano del estado de la página.
- ✓ Escribe `ResolveFont` con una cadena explícita: prueba la familia que distribuye la aplicación mediante una `PrivateFontCollection`, después una familia que sepas presente en todos los destinos, y registra cuál se usó para que la salida sea explicable. No dejes nunca que el renderizador lance una excepción porque a un host le falte una fuente.
- ✓ Guarda el mapa de bits en un `MemoryStream` con `ImageFormat.Png`, devuelve `stream.ToArray()` y, desde la página, entrega esos bytes a `Application.Download` detrás de un botón Export. Cronometra un renderizado y anota el número de bytes, porque ambas cifras vuelven en el Módulo 7.

- ✓ Abre el archivo de proyecto y confirma que lo que está en juego es `Managed.System.Drawing` y que no hay una referencia directa a `System.Drawing.Common` a su lado; comprueba que el target framework no se haya fijado a una variante `-windows` y, después, ejecuta el exportador en Windows y de nuevo en un contenedor Linux y compara los dos PNG píxel a píxel.

Entregables requeridos

- ✓ `TopologyImageRenderer` sin dependencia de ningún control de Wisej, que dibuja con `Graphics.FromImage` sobre un `Bitmap`
- ✓ Método `ResolveFont` con una cadena de alternativas explícita en lugar de una fuente del sistema dada por supuesta
- ✓ PNG escrito en un `MemoryStream` y descargado desde la página, con todos los objetos gráficos propios liberados
- ✓ Comprobación de referencias que confirme que se usa `Managed.System.Drawing` y que `System.Drawing.Common` no se añade junto a él
- ✓ `CrossPlatform.md` con evidencias en Windows y en un contenedor Linux, y las implicaciones para iOS, Android y macOS

Preguntas de repaso

- ✓ Un compañero dice que el renderizador demuestra que la aplicación es multiplataforma porque el código compila en Linux. ¿Qué tendrías que observar realmente en el contenedor antes de aceptar esa afirmación, y qué parte de la salida es más probable que difiera?
- ✓ Tu renderizador libera el `Bitmap` que ha creado. ¿Por qué liberar el `Graphics` dentro del controlador de Paint de un control sería un error, y qué regla cubre ambos casos en una sola frase?
- ✓ Alguien propone sustituir este renderizador por `SkiaSharp` porque en las pruebas de rendimiento es más rápido. ¿Qué necesitarías saber sobre la carga de trabajo antes de aceptarlo, y qué pierde el equipo al abandonar la API de `System.Drawing`?



Regla de dibujo: primero la superficie

Elige la superficie de renderizado antes de escribir una sola llamada de dibujo. Pregúntate dónde se producen los píxeles y qué cruza la red: un controlador de Paint de un control se ejecuta en el servidor y envía una imagen, `CellPaint` hace lo mismo dentro de una celda de cuadrícula, `Canvas` envía comandos de dibujo que ejecuta el navegador y no conserva nada tras un redimensionado, y `System.Drawing.Managed` produce bytes de imagen sin ningún control de por medio. Mantén el modelo semántico en objetos C#, haz que cada renderizador sea una función determinista de ese modelo, invalida solo cuando haya cambiado una dependencia visual, y deja siempre un equivalente textual de todo lo importante que exista únicamente como píxeles.