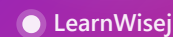


System.Drawing.Managed sur Windows, Linux, macOS, iOS et Android

Guide de la leçon · Guide du lab / de l'examen



Guide de la leçon

Un rendu hors écran qui tourne partout : pourquoi System.Drawing.Common est réservé à Windows sur .NET moderne, ce que sont réellement le package Managed.System.Drawing et l'assembly System.Drawing.Managed, la matrice des polices et des graphiques de Wisej.NET 4, une politique de polices déterministe, et le choix entre cette solution, ImageSharp, SkiaSharp, Aspose.Drawing et Microsoft.Maui.Graphics selon le cas d'usage. Ce que cela signifie pour un développeur .NET expérimenté qui construit déjà des écrans Wisej.NET et a désormais besoin de graphiques personnalisés — et comment aborder ce module.



Objectif en production

Un rendu hors écran qui tourne partout : pourquoi System.Drawing.Common est réservé à Windows sur .NET moderne, ce que sont réellement le package Managed.System.Drawing et l'assembly System.Drawing.Managed, la matrice des polices et des graphiques de Wisej.NET 4, une politique de polices déterministe, et le choix entre cette solution, ImageSharp, SkiaSharp, Aspose.Drawing et Microsoft.Maui.Graphics selon le cas d'usage.

1. Le problème en production

Tout ce que VisualOperationsStudio a dessiné jusqu'ici appartient à une session vivante : la jauge se redessine parce qu'un contrôle l'a demandé, et la topologie Canvas du Module 5 n'existe que sous forme de commandes que le navigateur a déjà exécutées. L'exploitation veut désormais cette même topologie dans un courriel d'incident nocturne et jointe à un ticket, c'est-à-dire une image sans aucun contrôle derrière elle. Dans le même temps, l'application migre vers un conteneur Linux : le code de dessin doit donc continuer à fonctionner sur une machine sans GDI+ et sans `libgdiplus`.

2. Concepts clés

Concept	Pourquoi c'est important
<code>System.Drawing.Common</code>	Le wrapper GDI+ de Microsoft, spécifique à Windows depuis .NET 6 ; le commutateur Unix a disparu dans .NET 7, ce n'est donc pas une option pour Linux, macOS, iOS ou Android.
<code>Managed.System.Drawing</code>	L'identifiant du package NuGet, inversé parce que la racine de packages <code>System.*</code> est réservée ; Wisej.NET 4 le référence déjà pour vous.
<code>System.Drawing.Managed.dll</code>	L'assembly que ce package livre — l'implémentation managée elle-même, que l'on voit dans un journal de compilation plutôt que dans vos sources.
L'espace de noms <code>System.Drawing</code>	Inchangé : <code>Bitmap</code> , <code>Graphics</code> , <code>Pen</code> et <code>Font</code> gardent leurs noms, si bien que le code de dessin existant compile sans réécriture.
Moteur graphique managé	Une implémentation purement managée sur un moteur ImageSharp personnalisé et intégré ; sur .NET moderne, elle évite GDI+, libgdiplus et les appels natifs.
<code>Graphics.FromImage</code>	Le point d'entrée hors écran : une surface de dessin au-dessus d'un <code>Bitmap</code> qui vous appartient, sans contrôle, sans événement Paint et sans session.
<code>PrivateFontCollection</code>	Charge les fichiers de police que l'application distribue, pour que la sortie dépende de ce que vous avez déployé et non de ce qui se trouve installé sur la machine.

3. Le fil de la leçon

Pourquoi System.Drawing.Common s'arrête à la frontière Windows

.NET moderne a changé les hypothèses autour de `System.Drawing`. Microsoft a marqué `System.Drawing.Common` comme spécifique à Windows à partir de .NET 6 ; le commutateur temporaire de prise en charge d'Unix proposé par cette version a été supprimé dans .NET 7. La consigne officielle pour les applications multiplateformes est de passer à une autre bibliothèque — SkiaSharp, ImageSharp, Aspose.Drawing ou Microsoft.Maui.Graphics.

Cette consigne est raisonnable pour un code neuf et douloureuse pour un code existant. Wisej.NET avait une version bien précise du problème : les applications d'entreprise et les contrôles tiers regorgent de code `System.Drawing`, et le framework lui-même a besoin du chargement d'images et de la mesure des polices ne serait-ce que pour disposer un écran. Réécrire le code de dessin de chaque client n'était pas envisageable, et un framework limité à Windows non plus.

Ce qu'est réellement Managed.System.Drawing

Wisej.NET 4 a introduit `System.Drawing.Managed`, une implémentation managée et multiplateforme du familier espace de noms `System.Drawing`. Trois noms voyagent ensemble. Le **package** NuGet est `Managed.System.Drawing`, dans cet ordre parce que la racine de packages `System.*` est réservée. L'**assembly** qu'il livre est `System.Drawing.Managed.dll`. L'**espace de noms** sur lequel votre code s'appuie

reste `System.Drawing`. Rien de tout cela n'est le `System.Drawing.Common` de Microsoft, et confondre les deux est l'erreur la plus coûteuse de ce module.

La documentation Wisej décrit l'implémentation comme purement managée, bâtie sur un moteur ImageSharp personnalisé et intégré. Sur .NET moderne, cela signifie ni GDI+, ni libgdiplus, ni appels natifs spécifiques à une plateforme. Sur .NET Framework, elle est volontairement hybride : la mesure des polices emprunte le chemin managé tandis que la couche d'imagerie peut encore encapsuler GDI+ natif, pour la compatibilité et les performances. Le but est la compatibilité d'API avec le System.Drawing classique, plus un comportement prévisible des images et des polices d'une plateforme à l'autre, et non une nouvelle API à apprendre.

Dessiner sans contrôle : Bitmap, Graphics et un PNG

Le `TopologyImageRenderer` du lab a la forme que devrait avoir tout moteur de rendu hors écran : un simple modèle et une taille de sortie en entrée, des octets en sortie. Il crée son propre `Bitmap`, obtient une surface avec `Graphics.FromImage`, l'efface, dessine les arêtes, les nœuds, les libellés et une légende avec `Pen`, `SolidBrush`, `GraphicsPath` et `Font`, puis enregistre dans un `MemoryStream` avec `ImageFormat.Png`. Comme il ne touche jamais à un contrôle Wisej, la même classe s'exécute depuis une page, un test unitaire ou un service d'arrière-plan.

```
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 propriété des ressources est la règle qui garde tout cela sûr. Libérez chaque `Bitmap`, `Graphics`, `Image`, `Pen`, `Brush`, `Font`, `GraphicsPath`, `Matrix` et flux que le moteur de rendu a lui-même créé, et laissez tranquilles les objets de l'appelant — l'exact contraire d'un gestionnaire Paint, où le `Graphics` vous est prêté. La taille compte aussi : un bitmap non compressé de 4000×3000 à quatre octets par pixel réclame environ 48 Mo rien que pour son tampon de pixels ; rendez donc à la taille dont la sortie a réellement besoin.

Les polices sont l'autre moitié du déterminisme, et les métriques de police multiplateformes comptent parmi les raisons les plus fortes pour lesquelles Wisej.NET utilise la bibliothèque managée. `Font`, `FontFamily` et `PrivateFontCollection` sont tous disponibles, et une `PrivateFontCollection` charge les fichiers de police que l'application distribue. Avant d'écrire `ResolveFont`, fixez la politique : quelles familles sont exigées, si elles sont installées partout ou embarquées, quel est le repli, si la licence autorise l'incorporation côté serveur, et si le navigateur a besoin de la même police.

La matrice des plateformes de Wisej.NET 4 et la règle des espaces de noms

La documentation sur les graphiques managés fournit une petite matrice conceptuelle qu'il vaut la peine de retenir, car elle explique des comportements que vous trouveriez sinon surprenants en session de conception :

- **Windows .NET Framework** — métriques de police managées, graphiques GDI+.
- **.NET moderne, toutes plateformes** — polices managées, graphiques managés.
- **Compilation .NET Core du concepteur Visual Studio** — polices managées, graphiques GDI+.

Les cibles de déploiement recommandées incluent `net8.0` pour les déploiements serveur Windows, Linux et macOS, plus `net8.0-ios`, `net8.0-android`, `net8.0-maccatalyst` et la cible Windows Hybrid ; des TFM plus récents fonctionnent là où la version de Wisej les prend en charge. Le numéro n'est pas la leçon. La leçon, c'est de ne pas imposer un suffixe `-windows` à un déploiement serveur multiplateforme, sauf si une dépendance réellement spécifique à Windows l'exige, car cette seule modification met discrètement fin à la portabilité que vous venez de payer.

Les collisions d'espaces de noms sont le risque associé. Les deux implémentations publient des types dans le même espace de noms : un projet qui ajoute sa propre référence à `System.Drawing.Common` à côté de celle, managée, de Wisej demande au compilateur de choisir entre deux types `Bitmap`. Supprimez-la lorsqu'elle est inutile ; lorsqu'un package tiers entraîne une version conflictuelle, utilisez l'alias de package explicite décrit par la documentation Wisej. Soyez tout aussi honnête sur les limites : la documentation conceptuelle signale des manques autour de certains effets d'imagerie avancés et des opérations sur les régions, et les métafichiers, les handles de contexte de périphérique et les contextes d'imprimante ne se transposent pas du tout à une implémentation purement managée. Isolez et repensez le code bâti sur des handles Win32, de l'interopérabilité HDC ou de l'impression, au lieu de supposer qu'il voyage.

Le gain opérationnel sous Linux est concret : libgdiplus cesse d'être un prérequis du conteneur et toute une classe de défaillances de dépendances natives disparaît — même si vous embarquez toujours les polices et testez toujours la mémoire des grands bitmaps. Sur iOS et Android, où `System.Drawing.Common` n'a jamais été pris en charge, c'est la bibliothèque managée qui rend praticable le partage de code de dessin pour Wisej.NET Hybrid ; vérifiez la disponibilité des polices, les chemins de fichiers, les encodeurs d'images et la pression mémoire sur de vrais appareils.

Choisir selon la charge de travail, pas selon la popularité

L'avantage de `System.Drawing.Managed` dans une solution Wisej.NET, c'est la compatibilité et l'intégration, pas la supériorité. C'est la bonne réponse lorsque vous réutilisez du code `System.Drawing`, alimentez des gestionnaires `Paint` et `CellPaint`, ou générez des images serveur à partir de données métier. `ImageSharp` est un pipeline managé et moderne de traitement d'images, avec une solide gestion des formats et un modèle de licence à examiner. `SkiaSharp` apporte une capacité 2D large et très performante, avec des composants natifs Skia. `Aspose.Drawing` offre une compatibilité managée de type GDI+ sous forme de produit commercial. `Microsoft.Maui.Graphics` est une abstraction de dessin pour les bases de code MAUI, pas un remplacement direct de `System.Drawing`.

Une décision de plus a sa place ici, car le rendu hors écran est l'endroit où les images arrivent de l'extérieur. Traitez une image non fiable comme une entrée non fiable : bornez la taille de l'envoi, les dimensions et le nombre de pixels décodés, restreignez les formats acceptés, décodez les grandes images

hors des chemins critiques d'une requête, et ne laissez jamais un nom de fichier fourni par l'utilisateur s'échapper du dossier de stockage que vous contrôlez.



Réflexe dessin : la surface d'abord

Choisissez la surface de rendu avant d'écrire le moindre appel de dessin. Demandez-vous où les pixels sont produits et ce qui transite sur le réseau : un gestionnaire Paint de contrôle s'exécute sur le serveur et expédie une image, CellPaint fait de même à l'intérieur d'une cellule de grille, Canvas envoie des commandes de dessin que le navigateur exécute et ne conserve rien après un redimensionnement, et System.Drawing.Managed produit des octets d'image sans aucun contrôle. Gardez le modèle sémantique dans des objets C#, faites de chaque moteur de rendu une fonction déterministe de ce modèle, n'invalidiez que lorsqu'une dépendance visuelle a changé, et laissez toujours un équivalent textuel de tout ce qui compte et n'existe que sous forme de pixels.



Objectifs d'apprentissage

À la fin de ce module, vous saurez :

- ✓ Expliquer pourquoi System.Drawing.Common est réservé à Windows sur .NET moderne.
- ✓ Comprendre l'architecture et le modèle de packages de System.Drawing.Managed.
- ✓ Utiliser Bitmap, Graphics, Font, Image, brushes, pens, paths et l'enregistrement d'images de façon multiplateforme.
- ✓ Comprendre la matrice des plateformes de Wisej.NET 4 et les consignes sur les collisions d'espaces de noms.
- ✓ Comparer System.Drawing.Managed à ImageSharp, SkiaSharp, Aspose.Drawing et Microsoft.Maui.Graphics selon le cas d'usage plutôt que selon la popularité.

Guide du lab / de l'examen

Ce que vous allez construire dans le lab pratique, l'approche suggérée et les livrables attendus.



Lab pratique

Ajoutez à VisualOperationsStudio un `TopologyImageRenderer` qui transforme le modèle de topologie en PNG sans toucher à un seul contrôle Wisej. Donnez-lui une méthode `Render` qui prend la scène et une taille de sortie, crée le `Bitmap`, obtient son `Graphics` avec `Graphics.FromImage`, dessine les nœuds, les arêtes, les libellés et une légende avec des pens, des brushes et un `GraphicsPath`, et libère tout ce qu'il possède sans toucher aux objets de l'appelant. Résolvez sa `Font` via une méthode `ResolveFont` qui essaie une famille fournie par l'application et retombe sur une police connue comme présente, au lieu de supposer qu'une police système existe ; enregistrez le résultat en PNG dans un `MemoryStream` et confiez les octets à la page pour le téléchargement. Vérifiez ensuite les références du projet pour confirmer que vous n'ajoutez pas `System.Drawing.Common` à côté de l'implémentation managée de Wisej, exécutez ou préparez le renderer sous Windows et dans un conteneur Linux, et rédigez le comportement du même code sur les cibles iOS, Android et macOS Hybrid.

Point de départ

VisualOperationsStudio après le Module 5 : la page de topologie possède un `TopologyScene` avec des objets `NodeModel` et `EdgeModel` en coordonnées monde, et `RenderScene` le dessine sur un `Wisej.Web.Canvas` que le navigateur repeint à partir des commandes. Rien dans le projet ne sait produire un fichier image, et chaque ligne de code de dessin écrite jusqu'ici a besoin d'un contrôle vivant et d'une session vivante pour s'exécuter.

Approche suggérée

- ✓ Créez `TopologyImageRenderer` dans un dossier qui ne comporte aucune directive `using` `Wisej.Web`, et donnez-lui une seule méthode publique qui prend un `TopologyScene` plus une largeur et une hauteur et renvoie un `byte[]`. Si vous ne pouvez pas l'appeler depuis un simple test console, elle est encore couplée à l'interface.
- ✓ Dans `Render`, créez le `Bitmap`, récupérez sa surface avec `Graphics.FromImage`, effacez-la, puis dessinez les arêtes, les formes des nœuds, les libellés et une légende avec `Pen`, `SolidBrush`, `GraphicsPath` et `DrawString`. Enveloppez chaque objet créé par la méthode dans un `using`, et passez la scène en paramètre plutôt que d'aller chercher l'état de la page.
- ✓ Écrivez `ResolveFont` avec une chaîne explicite : essayez la famille que l'application distribue via une `PrivateFontCollection`, puis une famille que vous savez présente sur toutes les cibles, et notez laquelle a été utilisée pour que la sortie reste explicable. Ne laissez jamais le renderer lever une exception parce qu'une machine n'a pas une police.
- ✓ Enregistrez le bitmap dans un `MemoryStream` avec `ImageFormat.Png`, renvoyez `stream.ToArray()`, et depuis la page confiez ces octets à `Application.Download` derrière un bouton Export. Chronométrez un rendu et notez le nombre d'octets, car ces deux chiffres reviennent au Module 7.

- ✓ Ouvrez le fichier projet et confirmez que c'est bien `Managed.System.Drawing` qui est en jeu et qu'aucune référence directe à `System.Drawing.Common` ne l'accompagne, vérifiez que le framework cible n'a pas été figé sur une variante `-windows`, puis exécutez l'exportateur sous Windows et de nouveau dans un conteneur Linux et comparez les deux PNG pixel par pixel.

Livrables attendus

- ✓ `TopologyImageRenderer` sans aucune dépendance à un contrôle Wisej, qui rend via `Graphics.FromImage` dans un `Bitmap`
- ✓ Méthode `ResolveFont` avec une chaîne de repli explicite au lieu d'une police système supposée présente
- ✓ PNG écrit dans un `MemoryStream` et téléchargé depuis la page, chaque objet graphique possédé étant libéré
- ✓ Vérification des références confirmant que `Managed.System.Drawing` est utilisé et que `System.Drawing.Common` n'est pas ajouté à côté
- ✓ `CrossPlatform.md` avec les preuves sous Windows et en conteneur Linux, et les implications pour iOS, Android et macOS

Questions de révision

- ✓ Un collègue affirme que le renderer prouve que l'application est multiplateforme puisque le code compile sous Linux. Que devriez-vous réellement observer dans le conteneur avant d'accepter cette affirmation, et quelle partie de la sortie risque le plus de différer ?
- ✓ Votre renderer libère le `Bitmap` qu'il a créé. Pourquoi libérer le `Graphics` dans le gestionnaire `Paint` d'un contrôle serait-il un bug, et quelle règle couvre les deux cas en une seule phrase ?
- ✓ Quelqu'un propose de remplacer ce renderer par `SkiaSharp` parce qu'il obtient de meilleurs résultats en banc d'essai. Que devriez-vous savoir de la charge de travail avant d'accepter, et que perd l'équipe en abandonnant l'API `System.Drawing` ?



Réflexe dessin : la surface d'abord

Choisissez la surface de rendu avant d'écrire le moindre appel de dessin. Demandez-vous où les pixels sont produits et ce qui transite sur le réseau : un gestionnaire `Paint` de contrôle s'exécute sur le serveur et expédie une image, `CellPaint` fait de même à l'intérieur d'une cellule de grille, `Canvas` envoie des commandes de dessin que le navigateur exécute et ne conserve rien après un redimensionnement, et `System.Drawing.Managed` produit des octets d'image sans aucun contrôle. Gardez le modèle sémantique dans des objets C#, faites de chaque moteur de rendu une fonction déterministe de ce modèle, n'invalidiez que lorsqu'une dépendance visuelle a changé, et laissez toujours un équivalent textuel de tout ce qui compte et n'existe que sous forme de pixels.