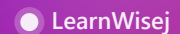


Créer votre propre pack d'icônes Wisej.NET

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



Guide de la leçon

La recette en quatre étapes derrière chaque pack d'icônes : une bibliothèque de classes, un répertoire Resources, des fichiers d'icônes marqués Embedded Resource et un catalogue statique de chaînes resource.wx qui offre à l'équipe l'IntelliSense au lieu de chemins magiques, ainsi que les pratiques de nommage, de versionnage, de recolorage et de remplacement qui gardent le pack maintenable. Ce que cela signifie pour un développeur qui connaît les bases de C# et a déjà travaillé avec les contrôles et le designer de Wisej.NET — et comment aborder ce module.



Objectif en production

La recette en quatre étapes derrière chaque pack d'icônes : une bibliothèque de classes, un répertoire Resources, des fichiers d'icônes marqués Embedded Resource et un catalogue statique de chaînes resource.wx qui offre à l'équipe l'IntelliSense au lieu de chemins magiques, ainsi que les pratiques de nommage, de versionnage, de recolorage et de remplacement qui gardent le pack maintenable.

1. Le problème en production

Au cinquième module, IconDesk affiche un logo embarqué, une image de thème et des icônes issues de deux packs officiels, et c'est là que le désordre commence : des chaînes de ressources apparaissent en ligne dans les gestionnaires d'événements, deux écrans orthographient différemment le même fichier, et personne ne sait dire quelles icônes le produit possède. Un pack personnalisé est le remède économique, et il achète une frontière : un assembly contient les visuels, un catalogue C# les nomme, et le reste de l'application ne saisit plus jamais un chemin.

2. Concepts clés

Concept	Pourquoi c'est important
assembly de ressources	Une simple bibliothèque de classes .NET dont l'unique rôle est de transporter des fichiers d'icônes en ressources embarquées ; elle n'a besoin d'aucune référence à l'application ni de code Wisej.NET propre.
la recette en quatre étapes	Créez une bibliothèque de classes, créez un répertoire nommé <code>Resources</code> , placez-y les fichiers d'icônes, définissez leur Build Action sur <code>Embedded Resource</code> . Tout ce qui suit relève du confort.
catalogue d'icônes	Une classe statique telle qu' <code>AppIcons</code> qui expose chaque icône sous forme de constante de chaîne nommée, si bien que le compilateur et l'IntelliSense remplacent les chemins copiés-collés.
nom sémantique contre nom de fichier	Le catalogue expose le vocabulaire de votre produit — enregistrer, supprimer, client, avertissement — pendant que le fichier garde un nom simple, si bien que le renommer change une ligne et non quarante écrans.
adresse de ressource qualifiée	Une icône dans un assembly distinct s'adresse par <code>resource.wx/</code> plus le nom de l'assembly plus le chemin de la ressource, et c'est pourquoi toutes les entrées du catalogue partagent une seule constante racine privée.
contrat de recolorage	Le suffixe <code>?color=</code> ne recolore que les visuels monochromes adaptés ; un SVG avec des valeurs <code>fill</code> codées en dur est la raison habituelle d'une icône qui refuse de changer de couleur.
jeux de thèmes et remplacements	Des packages parallèles partageant une même API de catalogue sur des visuels différents, plus des fichiers déployés qui priment sur une ressource embarquée, remplacent des icônes sans toucher au code de l'application.

3. Le fil de la leçon

Une bibliothèque de classes dont l'unique rôle est les icônes

Wisej.NET documente un pack d'icônes personnalisé en quatre étapes, et il vaut la peine de les lire littéralement, car on s'attend à quelque chose de plus solennel. Créez une bibliothèque de classes .NET. Créez-y un répertoire nommé `Resources`. Placez-y les fichiers d'icônes. Définissez leur Build Action sur `Embedded Resource`. C'est tout le mécanisme : le compilateur place les fichiers dans l'assembly, et le gestionnaire de ressources embarquées du Module 5 les sert. Dans le lab, la bibliothèque est `IconDesk.Icons`, elle cible un framework compatible avec IconDesk, et elle ne contient aucun code Wisej.NET.

Vous pouvez exprimer l'empaquetage dans le fichier projet avec un élément unique qui inclut tout ce qui se trouve sous `Resources`, mais Visual Studio écrit généralement ces métadonnées pour vous lorsque vous choisissez le Build Action. Ne faites pas les deux : un fichier déclaré deux fois est la raison courante pour laquelle un pack tout neuf compile mais produit des noms de ressources dupliqués ou manquants.

Après la première compilation, demandez une icône depuis l'application en cours d'exécution — le chemin de ressource que vous avez supposé est juste, ou il ne l'est pas.

Pourquoi un assembly plutôt qu'un dossier copié dans chaque solution ? Parce qu'un assembly est une unité : il peut être versionné, empaqueté en NuGet, partagé entre produits et remplacé en bloc, et sa frontière empêche les chemins d'icônes de fuir dans les écrans.

Le catalogue est ce que votre équipe utilise réellement

Un assembly de ressources seul reste un sac de chaînes, et les chaînes ne survivent pas à un code qui grossit. La seconde moitié d'un vrai pack est un catalogue statique : une classe, une constante racine privée, une constante publique par icône. Les appelants écrivent alors un nom que l'éditeur sait compléter et qu'un relecteur sait lire.

```
namespace IconDesk.Icons;

public static class AppIcons
{
    private const string Root = "resource.wx/IconDesk.Icons/";

    public const string Save = Root + "save.svg";
    public const string Delete = Root + "delete.svg";
    public const string Success = Root + "status-success.svg";
}

// callers never type a path again
btnSave.ImageSource = AppIcons.Save;
btnDelete.ImageSource = AppIcons.Delete + "?color=error";
```

Lisez attentivement la constante racine, car c'est la partie que l'on se trompe le plus souvent : le gestionnaire `resource.wx`, puis le **nom de l'assembly**, puis le chemin de la ressource dans cet assembly. Le chemin exact dépend de la façon dont votre bibliothèque embarque ses fichiers : vérifiez-le donc contre l'application en cours d'exécution au lieu de traiter une racine d'exemple comme une formule magique. Une fois confirmée, elle vit dans une seule constante privée.

L'autre moitié de la conception d'un catalogue est la sélection. Un pack n'est pas un dossier de toutes les icônes que quelqu'un a téléchargées ; c'est le vocabulaire de votre produit — actions, entités, états. Gardez des noms de fichiers simples : quand le design remplace `save.svg` par un glyphe redessiné, l'entrée du catalogue ne bouge pas.

Utiliser le pack depuis le designer et depuis C#

Référez le projet de bibliothèque de classes depuis IconDesk et *compilez la solution*. Cette compilation est l'étape que l'on saute, et c'est pourquoi l'Image Selector paraît parfois vide : le designer découvre les ressources de packs d'icônes dans les assemblies que le projet référence. Une fois qu'il les a, votre pack apparaît à côté des packs officiels, et choisir une icône écrit dans `Designer.cs` la même chaîne de source que produit votre constante de catalogue — ouvrez une fois le fichier généré et comparez les deux.

Utilisez les deux voies délibérément. Le designer convient aux icônes d'un contrôle que vous posez visuellement ; le catalogue convient à tout ce qui se décide à l'exécution, comme un glyphe d'état choisi

d'après une énumération ou une barre d'outils construite en code. Les deux aboutissent à une chaîne de source d'image que le navigateur résout et met en cache une icône à la fois, et aucune n'est un `System.Drawing.Image` — un SVG reste un SVG jusqu'au navigateur, et c'est pourquoi il reste net à 16 comme à 48 pixels.

Le recolorage mérite un test plutôt qu'une hypothèse. Le suffixe `?color=` recolore les visuels dessinés pour être recolorés : un tracé d'une seule couleur sans valeur `fill` ou `stroke` codée en dur qui viendrait s'y opposer. Dans le lab, vous vérifiez trois icônes avec une couleur hexadécimale, un nom de couleur du thème et les deux thèmes. Une icône qui refuse a généralement besoin qu'on retire l'attribut fill de son SVG, et une marque doit être laissée telle quelle : ses couleurs sont la marque, pas un choix de style.

Des règles qui gardent le pack vivant après la sortie

Tout ce qui précède représente un après-midi de travail. Ce qui décide si le pack aidera encore dans deux ans, ce sont les règles que vous écrivez maintenant. Le nommage vient en premier : un concept vaut une icône, la casse est cohérente, les view box sont normalisées, et chaque icône a une licence consignée. Les concepts en double — deux glyphes de recherche venus de deux designers — sont la façon dont un pack sélectionné redevient un dossier.

Le versionnage découle du fait de traiter le catalogue comme l'API publique. Ajouter une constante est additif. Remplacer le visuel derrière une constante existante est un changement visuel qui a sa place dans les notes de version, même si rien ne cesse de compiler. Supprimer ou renommer une constante est cassant : dépréciez plutôt que de supprimer. Les remplacements vous donnent deux leviers documentés et un levier architectural : une source qui omet le qualificateur d'assembly permet à Wisej.NET de préférer un fichier déployé correspondant, un chemin entièrement qualifié peut être remplacé en reproduisant la structure de répertoires du déploiement, et un package parallèle — un jeu à contraste élevé ou un jeu client — livre les mêmes noms de catalogue sur des visuels différents.

- Chaque icône a une licence consignée et une view box normalisée.
- Le catalogue est le contrat ; les visuels peuvent changer, les noms ne peuvent pas disparaître en silence.
- Une page d'exemple rend tout le pack dans les deux thèmes avant que vous ne taguiez une version.



Nommez d'abord le mécanisme

Avant d'affecter la moindre image, dites quel mécanisme vous utilisez : un objet `System.Drawing.Image` que le serveur rend en PNG, une chaîne de source d'image que le navigateur résout, une image de thème nommée, une ressource embarquée `resource.wx`, ou une police d'icônes dessinée en HTML. N'affectez qu'une seule propriété d'image par contrôle pour que les autres ne puissent pas la réinitialiser en silence, gardez une chaîne de source parlante plutôt qu'un chemin magique collé, préférez le SVG pour les icônes d'interface afin qu'elles restent nettes à toute taille et à toute échelle, et donnez à chaque icône porteuse de sens un équivalent textuel.



Objectifs d'apprentissage

À la fin de ce module, vous saurez :

- ✓ Créer une bibliothèque de classes qui empaquette les icônes de l'application en ressources embarquées.
- ✓ Exposer des constantes ou propriétés de source d'icônes stables pour l'IntelliSense et la réutilisation de code.
- ✓ Utiliser le pack personnalisé depuis le designer et depuis C#.
- ✓ Concevoir des pratiques de nommage, de versionnage, de thématisation et de remplacement d'icônes pour un produit maintenable.

Guide du lab / de l'examen

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



Lab pratique

Créez IconDesk.Icons comme bibliothèque de classes à côté de l'application, placez au moins douze icônes SVG dans un répertoire Resources et marquez-les Embedded Resource. Ajoutez un catalogue statique Applcons qui expose chaque icône sous forme de chaîne resource.wx nommée pour que les appelants n'aient jamais à saisir un chemin, référencez la bibliothèque depuis IconDesk, puis affectez certaines icônes via le designer et d'autres depuis le catalogue en C#. Vérifiez le suffixe de couleur sur trois icônes et notez le nettoyage SVG que certaines ont demandé. Terminez par les règles de nommage et de versionnage qui décident comment une nouvelle icône entre dans le pack.

Point de départ

Ouvrez la solution IconDesk telle que le Module 5 l'a laissée : l'application embarque déjà ses propres images dans un dossier **Resources** et les résout via **resource.wx**, et deux packs d'icônes officiels sont référencés depuis le Module 4. Rien dans la solution n'est encore un pack réutilisable : chaque chaîne de source personnalisée est donc toujours saisie à la main sur la page qui en a besoin.

Approche suggérée

- ✓ Ajoutez à la solution une bibliothèque de classes nommée **IconDesk.Icons**, ciblant un framework compatible avec IconDesk, et donnez-lui un répertoire **Resources** contenant au moins douze icônes SVG sélectionnées.
- ✓ Définissez le Build Action de chaque fichier d'icône sur **Embedded Resource**, compilez la bibliothèque une fois, et vérifiez le chemin de ressource d'une icône depuis l'application en cours d'exécution avant d'écrire la moindre entrée de catalogue.
- ✓ Écrivez la classe statique **AppIcons** avec une constante racine privée et une constante publique par icône, nommée d'après ce que l'icône signifie plutôt que d'après le fichier dont elle vient.
- ✓ Référencez la bibliothèque depuis IconDesk, recompilez la solution, puis affectez certaines icônes via l'Image Selector du designer et les autres depuis **AppIcons** en C# sur une page qui montre tout le pack.
- ✓ Ajoutez le suffixe de couleur à trois icônes, testez une couleur hexadécimale et un nom de couleur du thème dans un thème clair et un thème sombre, et corrigez ou dispensez tout visuel qui refuse le recolorage.

Livrables attendus

- ✓ Bibliothèque de classes IconDesk.Icons avec au moins douze fichiers SVG dans un répertoire Resources, tous en Embedded Resource
- ✓ Catalogue statique Applcons exposant chaque icône sous forme de chaîne resource.wx nommée
- ✓ Page IconDesk affectant certaines icônes via le designer et d'autres depuis le catalogue Applcons

- ✓ Trois icônes vérifiées avec le suffixe de couleur, avec une note sur le nettoyage SVG qu'elles ont demandé
- ✓ Règles de nommage et de versionnage décrivant comment une nouvelle icône entre dans le pack et comment une application en remplace une

Questions de révision

- ✓ Votre pack compile et le designer n'affiche rien dans l'Image Selector. Quelle étape de la recette vérifieriez-vous en premier, et quelle preuve vous dirait que les ressources sont réellement dans l'assembly ?
- ✓ Deux de vos trois icônes de test se recolorent et la troisième reste grise. Qu'y a-t-il dans ce SVG, et quand laisseriez-vous délibérément une icône non recolorable ?
- ✓ Un client veut faire remplacer trois icônes dans son déploiement uniquement. Laquelle des voies de remplacement choisiriez-vous, et que vous coûte ce choix à la prochaine version du pack ?



Nommez d'abord le mécanisme

Avant d'affecter la moindre image, dites quel mécanisme vous utilisez : un objet `System.Drawing.Image` que le serveur rend en PNG, une chaîne de source d'image que le navigateur résout, une image de thème nommée, une ressource embarquée `resource.wx`, ou une police d'icônes dessinée en HTML. N'affectez qu'une seule propriété d'image par contrôle pour que les autres ne puissent pas la réinitialiser en silence, gardez une chaîne de source parlante plutôt qu'un chemin magique collé, préférez le SVG pour les icônes d'interface afin qu'elles restent nettes à toute taille et à toute échelle, et donnez à chaque icône porteuse de sens un équivalent textuel.