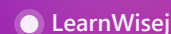


Crea tu propio pack de iconos de Wisej.NET

Guía de la lección · Guía del laboratorio / examen



Guía de la lección

La receta de cuatro pasos que hay detrás de todo pack de iconos: una biblioteca de clases, un directorio Resources, archivos de icono marcados como Embedded Resource y un catálogo estático de cadenas resource.wx que le da al equipo IntelliSense en lugar de rutas mágicas, más las prácticas de nomenclatura, versionado, recolorado y sobrescritura que mantienen el pack mantenible. Qué significa esto para un desarrollador que conoce C# básico y ya ha trabajado algo con los controles y el diseñador de Wisej.NET — y cómo leer este módulo.



Enfoque de producción

La receta de cuatro pasos que hay detrás de todo pack de iconos: una biblioteca de clases, un directorio Resources, archivos de icono marcados como Embedded Resource y un catálogo estático de cadenas resource.wx que le da al equipo IntelliSense en lugar de rutas mágicas, más las prácticas de nomenclatura, versionado, recolorado y sobrescritura que mantienen el pack mantenible.

1. El problema en producción

Para el quinto módulo IconDesk muestra un logotipo incrustado, una imagen de tema e iconos de dos packs oficiales, y ahí es donde empieza el desorden: aparecen cadenas de recurso en línea dentro de los manejadores de eventos, dos pantallas escriben de forma distinta el mismo archivo y nadie sabe decir qué iconos son propiedad del producto. Un pack propio es la solución barata, y compra una frontera: un ensamblado guarda la ilustración, un catálogo de C# la nombra y el resto de la aplicación no vuelve a escribir una ruta.

2. Conceptos clave

Concepto	Por qué importa
Ensamblado de recursos	Una biblioteca de clases .NET corriente cuyo único trabajo es llevar archivos de icono como recursos incrustados; no necesita referenciar la aplicación ni tener código propio de Wisej.NET.
La receta de cuatro pasos	Crea una biblioteca de clases, crea un directorio llamado <code>Resources</code> , pon dentro los archivos de icono y establece su Build Action en <code>Embedded Resource</code> . Todo lo que viene después es comodidad.
Catálogo de iconos	Una clase estática como <code>AppIcons</code> que expone cada icono como una constante de cadena con nombre, de modo que el compilador e IntelliSense sustituyen a las rutas copiadas y pegadas.
Nombre semántico frente a nombre de archivo	El catálogo expone el vocabulario de tu producto —guardar, eliminar, cliente, advertencia— mientras el archivo conserva un nombre sencillo, así que renombrarlo cambia una línea y no cuarenta pantallas.
Dirección de recurso calificada	Un icono de otro ensamblado se direcciona como <code>resource.wx/</code> más el nombre del ensamblado más la ruta del recurso, y por eso todas las entradas del catálogo comparten una constante raíz privada.
Contrato de recolorado	El sufijo <code>?color=</code> solo recolora ilustraciones monocromas adecuadas; un SVG con valores de <code>fill</code> fijados en el archivo es la razón habitual de que un icono se niegue a cambiar de color.
Conjuntos temáticos y sobrescrituras	Paquetes paralelos que comparten una misma API de catálogo sobre ilustraciones distintas, más archivos desplegados que tienen prioridad sobre un recurso incrustado, permiten sustituir iconos sin tocar el código de la aplicación.

3. Desarrollo de la lección

Una biblioteca de clases cuyo único trabajo son los iconos

Wisej.NET documenta un pack de iconos propio en cuatro pasos, y merece la pena leerlos al pie de la letra porque la gente espera algo más ceremonioso. Crea una biblioteca de clases .NET. Crea dentro un directorio llamado `Resources`. Pon ahí los archivos de icono. Establece su Build Action en `Embedded Resource`. Ese es todo el mecanismo: el compilador coloca los archivos dentro del ensamblado y el controlador de recursos incrustados del Módulo 5 los sirve. En el laboratorio la biblioteca es `IconDesk.Icons`, tiene como destino un framework compatible con IconDesk y no contiene absolutamente nada de código de Wisej.NET.

Puedes expresar el empaquetado en el archivo del proyecto con un elemento que incluya todo lo que hay bajo `Resources`, pero Visual Studio suele escribir esos metadatos por ti cuando eliges la Build Action. No hagas las dos cosas: un archivo declarado dos veces es la razón habitual de que un pack recién creado compile pero produzca nombres de recurso duplicados o ausentes. Después de la primera compilación, pide un icono desde la aplicación en ejecución: la ruta de recurso que supusiste es correcta o no lo es.

¿Por qué un ensamblado y no una carpeta copiada en cada solución? Porque un ensamblado es una unidad: se puede versionar, empaquetar como NuGet, compartir entre productos y sustituir por completo, y su frontera impide que las rutas de iconos se filtren a las pantallas.

El catálogo es lo que tu equipo usa de verdad

Un ensamblado de recursos por sí solo sigue siendo un saco de cadenas, y las cadenas no sobreviven a una base de código que crece. La segunda mitad de un pack de verdad es un catálogo estático: una clase, una constante raíz privada y una constante pública por icono. Quien lo use escribe entonces un nombre que el editor puede completar y que un revisor puede leer.

```
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";
```

Lee con atención la constante raíz, porque es la parte que la gente hace mal: el controlador `resource.wx`, después el **nombre del ensamblado** y después la ruta del recurso dentro de ese ensamblado. La ruta exacta depende de cómo incruste tu biblioteca sus archivos, así que verifícala contra la aplicación en ejecución en lugar de tratar una raíz de ejemplo como si fuera mágica. Una vez confirmada, vive en una única constante privada.

La otra mitad del diseño del catálogo es la selección. Un pack no es una carpeta con todos los iconos que alguien se descargó; es el vocabulario de tu producto: acciones, entidades, estados. Mantén sencillos los nombres de archivo: cuando diseño sustituya `save.svg` por un glifo redibujado, la entrada del catálogo no se mueve.

Usar el pack desde el diseñador y desde C#

Referencia el proyecto de la biblioteca de clases desde IconDesk y *compila la solución*. Esa compilación es el paso que la gente se salta, y es la razón de que a veces el Image Selector parezca vacío: el diseñador descubre los recursos de packs de iconos a partir de los ensamblados que referencia el proyecto. Una vez que los tiene, tu pack aparece junto a los oficiales, y elegir un icono escribe en `Designer.cs` la misma cadena de origen que produce tu constante del catálogo: abre una vez el archivo generado y compara las dos.

Usa ambas vías de forma deliberada. El diseñador es el adecuado para los iconos de un control que colocas visualmente; el catálogo es el adecuado para todo lo que se decide en tiempo de ejecución, como un glifo de estado elegido a partir de un enum o una barra de herramientas construida por código. Ambos

acaban siendo una cadena de origen de imagen que el navegador resuelve y cachea icono a icono, y ninguno es un `System.Drawing.Image`: un SVG sigue siendo un SVG hasta llegar al navegador, y por eso se mantiene nítido a 16 y a 48 píxeles.

El recolorado merece una prueba y no una suposición. El sufijo `?color=` recolora ilustraciones dibujadas para ser recoloradas: un trazado de un solo color sin un `fill` o un `stroke` fijados que le peleen. En el laboratorio verificas tres iconos con un color hexadecimal, un nombre de color del tema y ambos temas. Un icono que se niega suele necesitar que se le quite el atributo `fill` de su SVG, y a una marca corporativa hay que dejarla en paz: sus colores son la marca, no una elección de estilo.

Reglas que mantienen vivo el pack después de publicarlo

Todo lo anterior es una tarde de trabajo. Lo que decide si el pack sigue ayudando dentro de dos años son las reglas que escribas ahora. La nomenclatura va primero: un concepto significa un icono, el uso de mayúsculas es coherente, los view box están normalizados y cada icono tiene su licencia registrada. Los conceptos duplicados —dos glifos de búsqueda de dos diseñadores— son la forma en que un pack seleccionado vuelve a convertirse en una carpeta.

El versionado se deriva de tratar el catálogo como la API pública. Añadir una constante es aditivo. Sustituir la ilustración que hay detrás de una constante existente es un cambio visual que debe constar en las notas de la versión aunque no falle ninguna compilación. Eliminar o renombrar una constante es disruptivo, así que márcala como obsoleta en lugar de borrarla. Las sobrescrituras te dan dos palancas documentadas y una arquitectónica: un origen que omite el calificador de ensamblado permite a Wisej.NET preferir un archivo desplegado coincidente, una ruta totalmente calificada se puede sobrescribir reproduciendo la estructura de directorios del despliegue, y un paquete paralelo —un conjunto de alto contraste o el de un cliente— distribuye los mismos nombres de catálogo sobre otra ilustración.

- Cada icono tiene un registro de licencia y un view box normalizado.
- El catálogo es el contrato; la ilustración puede cambiar, los nombres no pueden desaparecer en silencio.
- Una página de ejemplo renderiza todo el pack en ambos temas antes de etiquetar una versión.



Nombra primero el mecanismo

Antes de asignar cualquier imagen, di qué mecanismo estás usando: un objeto `System.Drawing.Image` que el servidor renderiza como PNG, una cadena de origen de imagen que resuelve el navegador, una imagen de tema con nombre, un recurso incrustado servido por `resource.wx` o una fuente de iconos dibujada como HTML. Asigna una sola propiedad de imagen por control para que las demás no puedan restablecerla en silencio, mantén la cadena de origen con significado en lugar de una ruta mágica pegada, prefiere SVG para los iconos de interfaz para que se mantengan nítidos en cualquier tamaño y escala, y dale una alternativa textual a todo icono que transmita significado.



Objetivos de aprendizaje

Al terminar este módulo serás capaz de:

- ✓ Crear una biblioteca de clases que empaquete los iconos de la aplicación como recursos incrustados.
- ✓ Exponer constantes o propiedades estables de origen de icono para IntelliSense y la reutilización de código.
- ✓ Usar el pack propio desde el diseñador y desde C#.
- ✓ Diseñar prácticas de nomenclatura, versionado, tematización y sobrescritura de iconos para un producto mantenible.

Guía del laboratorio / examen

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



Laboratorio práctico

Crea IconDesk.Icons como una biblioteca de clases junto a la aplicación, pon al menos doce iconos SVG en un directorio Resources y márcalos como Embedded Resource. Añade un catálogo estático Applcons que exponga cada icono como una cadena resource.wx con nombre para que quien lo use nunca escriba una ruta, referencia la biblioteca desde IconDesk y después asigna algunos iconos mediante el diseñador y otros desde el catálogo en C#. Verifica el sufijo de color en tres iconos y anota la limpieza del SVG que alguno de ellos necesitó. Termina con las reglas de nomenclatura y versionado que deciden cómo entra un icono nuevo en el pack.

Punto de partida

Abre la solución IconDesk tal como la dejó el Módulo 5: la aplicación ya incrusta sus propias imágenes en una carpeta `Resources` y las resuelve mediante `resource.wx`, y desde el Módulo 4 hay dos packs de iconos oficiales referenciados. Todavía no hay en la solución nada que sea un pack reutilizable, así que cada cadena de origen propia se sigue escribiendo a mano en la página que la necesita.

Enfoque sugerido

- ✓ Añade a la solución una biblioteca de clases llamada `IconDesk.Icons`, con un framework de destino compatible con IconDesk, y dale un directorio `Resources` con al menos doce iconos SVG seleccionados.
- ✓ Establece la Build Action de cada archivo de icono en `Embedded Resource`, compila la biblioteca una vez y confirma la ruta de recurso de un icono desde la aplicación en ejecución antes de escribir ninguna entrada del catálogo.
- ✓ Escribe la clase estática `AppIcons` con una constante raíz privada y una constante pública por icono, con el nombre de lo que el icono significa y no del archivo del que vino.
- ✓ Referencia la biblioteca desde IconDesk, recompila la solución y asigna después algunos iconos con el Image Selector del diseñador y el resto desde `AppIcons` en C#, en una página que muestre el pack entero.
- ✓ Añade el sufijo de color a tres iconos, prueba un color hexadecimal y un nombre de color del tema tanto en un tema claro como en uno oscuro, y corrige o exime a la ilustración que se niegue a recolorearse.

Entregables requeridos

- ✓ Biblioteca de clases IconDesk.Icons con al menos doce archivos SVG en un directorio Resources, todos como Embedded Resource
- ✓ Catálogo estático Applcons que expone cada icono como una cadena resource.wx con nombre
- ✓ Página de IconDesk que asigna algunos iconos mediante el diseñador y otros desde el catálogo Applcons

- ✓ Tres iconos verificados con el sufijo de color más una nota sobre la limpieza del SVG que necesitaron
- ✓ Reglas de nomenclatura y versionado que describen cómo entra un icono nuevo en el pack y cómo una aplicación sobrescribe uno

Preguntas de repaso

- ✓ Tu pack compila y el diseñador no muestra nada en el Image Selector. ¿Qué paso de la receta comprobarías primero y qué prueba te diría que los recursos están de verdad en el ensamblado?
- ✓ Dos de tus tres iconos de prueba se recolorean y el tercero se queda gris. ¿Qué hay en ese SVG y cuándo dejarías a propósito un icono sin posibilidad de recoloreado?
- ✓ Un cliente quiere sustituir tres iconos solo en su despliegue. ¿Cuál de las vías de sobrescritura elegirías y qué te cuesta esa elección en la siguiente versión del pack?



Nombra primero el mecanismo

Antes de asignar cualquier imagen, di qué mecanismo estás usando: un objeto `System.Drawing.Image` que el servidor renderiza como PNG, una cadena de origen de imagen que resuelve el navegador, una imagen de tema con nombre, un recurso incrustado servido por `resource.wx` o una fuente de iconos dibujada como HTML. Asigna una sola propiedad de imagen por control para que las demás no puedan restablecerla en silencio, mantén la cadena de origen con significado en lugar de una ruta mágica pegada, prefiere SVG para los iconos de interfaz para que se mantengan nítidos en cualquier tamaño y escala, y dale una alternativa textual a todo icono que transmita significado.