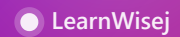


Building Your Own Wisej.NET Icon Pack

Lesson Guide · Lab / Exam Guide



Lesson Guide

The four-step recipe behind every icon pack: a class library, a Resources directory, icon files marked Embedded Resource, and a static catalog of resource.wx strings that gives the team IntelliSense instead of magic paths, plus the naming, versioning, recoloring and override practices that keep the pack maintainable. What this means for a developer who knows basic C# and has done some Wisej.NET control and designer work — and how to read this module.



Production focus

The four-step recipe behind every icon pack: a class library, a Resources directory, icon files marked Embedded Resource, and a static catalog of resource.wx strings that gives the team IntelliSense instead of magic paths, plus the naming, versioning, recoloring and override practices that keep the pack maintainable.

1. The production problem

By the fifth module IconDesk shows an embedded logo, a theme image and icons from two official packs, and that is when the mess begins: resource strings appear inline in event handlers, two screens spell the same file differently, and nobody can say which icons the product owns. A custom pack is the cheap fix, and it buys a boundary: one assembly holds the artwork, one C# catalog names it, and the rest of the application never types a path again.

2. Key concepts

Concept	Why it matters
Resource assembly	A plain .NET class library whose only job is to carry icon files as embedded resources; it needs no reference to the application and no Wisej.NET code of its own.
The four-step recipe	Create a class library, create a directory named <code>Resources</code> , put the icon files in it, set their Build Action to <code>Embedded Resource</code> . Everything after that is convenience.
Icon catalog	A static class such as <code>AppIcons</code> that exposes each icon as a named constant string, so the compiler and IntelliSense replace copy-and-paste paths.
Semantic name versus file name	The catalog exposes your product's vocabulary — save, delete, customer, warning — while the file keeps a simple name, so renaming it changes one line, not forty screens.
Qualified resource address	An icon in a separate assembly is addressed as <code>resource.wx/</code> plus the assembly name plus the resource path, which is why every catalog entry shares one private root constant.
Recoloring contract	The <code>?color=</code> suffix only recolors suitable monochrome artwork; an SVG with hard-coded <code>fill</code> values is the usual reason an icon refuses to change color.
Theme sets and overrides	Parallel packages sharing one catalog API over different artwork, plus deployed files that take precedence over an embedded resource, replace icons without touching application code.

3. Lesson narrative

A class library whose only job is icons

Wisej.NET documents a custom icon pack as four steps, and it is worth reading them literally because people expect something more ceremonious. Create a .NET class library. Create a directory in it named `Resources`. Put the icon files there. Set their Build Action to `Embedded Resource`. That is the entire mechanism: the compiler places the files inside the assembly, and the embedded-resource handler from Module 5 serves them. In the lab the library is `IconDesk.Icons`, it targets a framework compatible with IconDesk, and it contains no Wisej.NET code at all.

You can express the packaging in the project file with one item that includes everything under `Resources`, but Visual Studio usually writes that metadata for you when you pick the Build Action. Do not do both: a file declared twice is the common reason a fresh pack builds yet produces duplicate or missing resource names. After the first build, request one icon from the running application — the resource path you guessed is either right or it is not.

Why an assembly rather than a folder copied into every solution? Because an assembly is a unit: it can be versioned, packaged as NuGet, shared between products and replaced wholesale, and its boundary stops icon paths leaking into screens.

The catalog is what your team actually uses

A resource assembly on its own is still a bag of strings, and strings do not survive a growing codebase. The second half of a real pack is a static catalog: one class, one private root constant, one public constant per icon. Callers then write a name the editor can complete and a reviewer can read.

```
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";
```

Read the root constant carefully, because it is the part people get wrong: the `resource.wx` handler, then the **assembly name**, then the resource path inside that assembly. The exact path depends on how your library embeds its files, so verify it against the running application rather than treating a sample root as magic. Once confirmed, it lives in one private constant.

The other half of catalog design is curation. A pack is not a folder of every icon anyone downloaded; it is your product's vocabulary — actions, entities, statuses. Keep the file names simple: when design replaces `save.svg` with a redrawn glyph, the catalog entry does not move.

Using the pack from the designer and from C#

Reference the class-library project from IconDesk and *build the solution*. That build is the step people skip, and it is why the Image Selector sometimes looks empty: the designer discovers icon-pack resources from the assemblies the project references. Once it has them, your pack appears next to the official ones, and picking an icon writes the same source string into `Designer.cs` that your catalog constant produces — open the generated file once and compare the two.

Use both routes deliberately. The designer is right for icons on a control you lay out visually; the catalog is right for anything decided at runtime, such as a status glyph chosen from an enum or a toolbar built in code. Both end up as an image-source string the browser resolves and caches one icon at a time, and neither is a `System.Drawing.Image` — SVG stays an SVG all the way to the browser, which is why it stays sharp at 16 and at 48 pixels.

Recoloring deserves a test rather than an assumption. The `?color=` suffix recolors artwork drawn to be recolored: a single-color path with no hard-coded `fill` or `stroke` fighting it. In the lab you verify three icons against a hex color, a theme color name and both themes. An icon that refuses usually needs the fill attribute stripped from its SVG, and a brand mark should be left alone: its colors are the brand, not a style choice.

Rules that keep the pack alive after release

Everything above is one afternoon of work. What decides whether the pack still helps in two years is the rules you write down now. Naming comes first: one concept means one icon, casing is consistent, view boxes are normalized, and every icon has a recorded license. Duplicate concepts — two search glyphs from two designers — are how a curated pack turns back into a folder.

Versioning follows from treating the catalog as the public API. Adding a constant is additive. Replacing the artwork behind an existing constant is a visual change that belongs in the release notes even though nothing fails to compile. Removing or renaming a constant is breaking, so deprecate rather than delete. Overrides give you two documented levers and one architectural one: a source that omits the assembly qualifier lets Wisej.NET prefer a matching deployed file, a fully qualified path can be overridden by matching the deployment directory structure, and a parallel package — a high-contrast or customer set — ships the same catalog names over different artwork.

- Every icon has a license record and a normalized view box.
- The catalog is the contract; artwork may change, names may not disappear quietly.
- A sample page renders the whole pack in both themes before you tag a version.



Name the mechanism first

Before you assign any image, say which mechanism you are using: a `System.Drawing.Image` object the server renders as PNG, an image-source string the browser resolves, a named theme image, a `resource.wx` embedded resource, or an icon font drawn as HTML. Assign one image property per control so the others cannot quietly reset it, keep the source string meaningful instead of a pasted magic path, prefer SVG for interface icons so they stay sharp at every size and scale, and give every icon that carries meaning a text alternative.



Learning objectives

By the end of this module you'll be able to:

- ✓ Create a class library that packages application icons as embedded resources.
- ✓ Expose stable icon source constants/properties for IntelliSense and code reuse.
- ✓ Use the custom pack from the designer and from C#.
- ✓ Design icon naming, versioning, theming and override practices for a maintainable product.

Lab / Exam Guide

What you'll build in the hands-on lab, the suggested approach, and the required deliverables.



Hands-on lab

Create IconDesk.Icons as a class library beside the application, put at least twelve SVG icons in a Resources directory and mark them Embedded Resource. Add a static Applcons catalog that exposes each icon as a named resource.wx string so callers never type a path, reference the library from IconDesk, then assign some icons through the designer and others from the catalog in C#. Verify the colour suffix on three icons and write down the SVG cleanup any of them needed. Finish with the naming and versioning rules that decide how a new icon enters the pack.

Starting point

Open the IconDesk solution as Module 5 left it: the application already embeds its own images in a `Resources` folder and resolves them through `resource.wx`, and two official icon packs are referenced from Module 4. Nothing in the solution is a reusable pack yet, so every custom source string is still typed by hand on the page that needs it.

Suggested approach

- ✓ Add a class library named `IconDesk.Icons` to the solution, targeting a framework compatible with IconDesk, and give it a `Resources` directory holding at least twelve curated SVG icons.
- ✓ Set every icon file's Build Action to `Embedded Resource`, build the library once, and confirm the resource path of one icon from the running application before you write any catalog entry.
- ✓ Write the static `AppIcons` class with one private root constant and one public constant per icon, named for what the icon means rather than for the file it came from.
- ✓ Reference the library from IconDesk, rebuild the solution, then assign some icons through the designer Image Selector and the rest from `AppIcons` in C# on a page that shows the whole pack.
- ✓ Append the colour suffix to three icons, test a hex colour and a theme colour name in both a light and a dark theme, and fix or exempt any artwork that refuses to recolour.

Required deliverables

- ✓ IconDesk.Icons class library with at least twelve SVG files in a Resources directory, all Embedded Resource
- ✓ Static Applcons catalog exposing every icon as a named resource.wx string
- ✓ IconDesk page assigning some icons through the designer and others from the Applcons catalog
- ✓ Three icons verified with the colour suffix plus a note on the SVG cleanup they needed
- ✓ Naming and versioning rules describing how a new icon enters the pack and how an application overrides one

Student review questions

- ✓ Your pack builds and the designer shows nothing in the Image Selector. Which step of the recipe would you check first, and what evidence would tell you the resources really are in the assembly?
- ✓ Two of your three test icons recolour and the third stays grey. What is in that SVG, and when would you leave an icon un-recolourable on purpose?
- ✓ A customer wants three icons replaced in their deployment only. Which of the override routes would you choose, and what does that choice cost you at the next pack version?



Name the mechanism first

Before you assign any image, say which mechanism you are using: a `System.Drawing.Image` object the server renders as PNG, an image-source string the browser resolves, a named theme image, a `resource.wx` embedded resource, or an icon font drawn as HTML. Assign one image property per control so the others cannot quietly reset it, keep the source string meaningful instead of a pasted magic path, prefer SVG for interface icons so they stay sharp at every size and scale, and give every icon that carries meaning a text alternative.