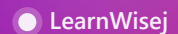


Ein eigenes Wisej.NET-Icon-Pack bauen

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Das vierstufige Rezept hinter jedem Icon-Pack: eine Klassenbibliothek, ein Verzeichnis Resources, Icon-Dateien mit Build Action Embedded Resource und ein statischer Katalog aus resource.wx-Strings, der dem Team IntelliSense statt magischer Pfade gibt – dazu die Konventionen für Benennung, Versionierung, Umfärben und Überschreiben, die das Pack wartbar halten. Was das für Entwicklerinnen und Entwickler bedeutet, die die Grundlagen von C# beherrschen und schon etwas mit den Steuerelementen und dem Designer von Wisej.NET gearbeitet haben – und wie Sie dieses Modul lesen.



Fokus: Produktivbetrieb

Das vierstufige Rezept hinter jedem Icon-Pack: eine Klassenbibliothek, ein Verzeichnis Resources, Icon-Dateien mit Build Action Embedded Resource und ein statischer Katalog aus resource.wx-Strings, der dem Team IntelliSense statt magischer Pfade gibt – dazu die Konventionen für Benennung, Versionierung, Umfärben und Überschreiben, die das Pack wartbar halten.

1. Das Problem im Produktivbetrieb

Beim fünften Modul zeigt IconDesk ein eingebettetes Logo, ein Theme-Bild und Icons aus zwei offiziellen Packs – und genau da beginnt das Durcheinander: Ressourcen-Strings stehen direkt in Event-Handletern, zwei Bildschirme schreiben dieselbe Datei unterschiedlich, und niemand kann sagen, welche Icons dem Produkt gehören. Ein eigenes Pack ist die günstige Lösung, und es bringt eine Grenze: Eine Assembly hält die Grafiken, ein C#-Katalog benennt sie, und der Rest der Anwendung tippt nie wieder einen Pfad.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
Ressourcen-Assembly	Eine schlichte .NET-Klassenbibliothek, deren einzige Aufgabe es ist, Icon-Dateien als eingebettete Ressourcen zu tragen; sie braucht keine Referenz auf die Anwendung und keinen eigenen Wisej.NET-Code.
Das vierstufige Rezept	Eine Klassenbibliothek erstellen, ein Verzeichnis namens <code>Resources</code> anlegen, die Icon-Dateien hineinlegen, ihre Build Action auf <code>Embedded Resource</code> setzen. Alles danach ist Komfort.
Icon-Katalog	Eine statische Klasse wie <code>AppIcons</code> , die jedes Icon als benannte String-Konstante bereitstellt, sodass Compiler und IntelliSense kopierte Pfade ersetzen.
Semantischer Name gegenüber Dateiname	Der Katalog zeigt das Vokabular Ihres Produkts – save, delete, customer, warning –, während die Datei einen einfachen Namen behält; sie umzubenennen ändert also eine Zeile und nicht vierzig Bildschirme.
Qualifizierte Ressourcenadresse	Ein Icon in einer eigenen Assembly wird als <code>resource.wx/</code> plus Assembly-Name plus Ressourcenpfad adressiert – deshalb teilen sich alle Katalogeinträge eine private Wurzelkonstante.
Vertrag beim Umfärben	Das Suffix <code>?color=</code> färbt nur geeignete einfarbige Grafiken um; ein SVG mit fest verdrahteten <code>fill</code> -Werten ist der übliche Grund, warum ein Icon seine Farbe nicht ändert.
Theme-Sätze und Überschreibungen	Parallele Pakete mit derselben Katalog-API über anderen Grafiken sowie ausgelieferte Dateien, die Vorrang vor einer eingebetteten Ressource haben, ersetzen Icons, ohne Anwendungscode anzufassen.

3. Der rote Faden der Lektion

Eine Klassenbibliothek, deren einzige Aufgabe Icons sind

Wisej.NET dokumentiert ein eigenes Icon-Pack als vier Schritte, und es lohnt sich, sie wörtlich zu lesen, weil man etwas Feierlicheres erwartet. Erstellen Sie eine .NET-Klassenbibliothek. Legen Sie darin ein Verzeichnis namens `Resources` an. Legen Sie die Icon-Dateien hinein. Setzen Sie ihre Build Action auf `Embedded Resource`. Das ist der gesamte Mechanismus: Der Compiler legt die Dateien in die Assembly, und der Handler für eingebettete Ressourcen aus Modul 5 liefert sie aus. Im Lab heißt die Bibliothek `IconDesk.Icons`, sie zielt auf ein zu IconDesk passendes Framework und enthält überhaupt keinen Wisej.NET-Code.

Sie können die Paketierung in der Projektdatei mit einem Eintrag ausdrücken, der alles unter `Resources` einschließt, aber Visual Studio schreibt diese Metadaten meist selbst, wenn Sie die Build Action wählen. Tun Sie nicht beides: Eine doppelt deklarierte Datei ist der häufige Grund, warum ein frisches Pack zwar baut, aber doppelte oder fehlende Ressourcennamen erzeugt. Fordern Sie nach dem ersten Build ein Icon aus der laufenden Anwendung an – der Ressourcenpfad, den Sie angenommen haben, stimmt entweder oder nicht.

Warum eine Assembly statt eines Ordners, der in jede Projektmappe kopiert wird? Weil eine Assembly eine Einheit ist: Sie lässt sich versionieren, als NuGet paketieren, zwischen Produkten teilen und komplett ersetzen – und ihre Grenze verhindert, dass Icon-Pfade in die Bildschirme durchsickern.

Der Katalog ist das, was Ihr Team tatsächlich benutzt

Eine Ressourcen-Assembly allein ist immer noch eine Sammlung von Strings, und Strings überleben eine wachsende Codebasis nicht. Die zweite Hälfte eines echten Packs ist ein statischer Katalog: eine Klasse, eine private Wurzelkonstante, eine öffentliche Konstante pro Icon. Aufrufer schreiben dann einen Namen, den der Editor vervollständigen und ein Reviewer lesen kann.

```
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";
}

// Aufrufer tippen nie wieder einen Pfad
btnSave.ImageSource = AppIcons.Save;
btnDelete.ImageSource = AppIcons.Delete + "?color=error";
```

Lesen Sie die Wurzelkonstante genau, denn das ist der Teil, den man falsch macht: der Handler `resource.wx`, dann der **Assembly-Name**, dann der Ressourcenpfad innerhalb dieser Assembly. Der genaue Pfad hängt davon ab, wie Ihre Bibliothek ihre Dateien einbettet – prüfen Sie ihn also gegen die laufende Anwendung, statt eine Beispielwurzel für gesetzt zu halten. Einmal bestätigt, lebt er in einer einzigen privaten Konstante.

Die andere Hälfte des Katalogentwurfs ist Kuratierung. Ein Pack ist kein Ordner mit jedem Icon, das irgendwer heruntergeladen hat; es ist das Vokabular Ihres Produkts – Aktionen, Entitäten, Zustände. Halten Sie die Dateinamen einfach: Wenn das Design `save.svg` durch ein neu gezeichnetes Glyph ersetzt, bewegt sich der Katalogeintrag nicht.

Das Pack aus dem Designer und aus C# verwenden

Referenzieren Sie das Klassenbibliotheksprojekt aus IconDesk und *bauen Sie die Projektmappe*. Dieser Build ist der Schritt, den man überspringt, und er ist der Grund, warum der Image Selector manchmal leer wirkt: Der Designer findet Icon-Pack-Ressourcen in den Assemblys, die das Projekt referenziert. Sobald er sie hat, erscheint Ihr Pack neben den offiziellen, und ein Icon auszuwählen schreibt denselben Quell-String in `Designer.cs`, den auch Ihre Katalogkonstante liefert – öffnen Sie die generierte Datei einmal und vergleichen Sie beides.

Verwenden Sie beide Wege bewusst. Der Designer passt für Icons auf einem Steuerelement, das Sie visuell platzieren; der Katalog passt für alles, was zur Laufzeit entschieden wird, etwa ein aus einem Enum gewähltes Status-Glyph oder eine im Code gebaute Toolbar. Beides endet als Bildquellen-String, den der

Browser auflöst und Icon für Icon cacht, und keines ist ein `System.Drawing.Image` – ein SVG bleibt bis in den Browser ein SVG, weshalb es bei 16 wie bei 48 Pixeln scharf bleibt.

Umfärben verdient einen Test statt einer Annahme. Das Suffix `?color=` färbt Grafiken um, die zum Umfärben gezeichnet wurden: ein einfarbiger Pfad ohne fest verdrahtetes `fill` oder `stroke`, das dagegenhält. Im Lab prüfen Sie drei Icons gegen eine Hex-Farbe, einen Theme-Farbnamen und beide Themes. Ein Icon, das sich verweigert, braucht meist ein aus seinem SVG entferntes fill-Attribut, und eine Wortmarke sollte unangetastet bleiben: Ihre Farben sind die Marke und keine Stilfrage.

Regeln, die das Pack nach dem Release am Leben halten

Alles bisher Beschriebene ist ein Nachmittag Arbeit. Ob das Pack in zwei Jahren noch hilft, entscheiden die Regeln, die Sie jetzt aufschreiben. Zuerst die Benennung: Ein Konzept bedeutet ein Icon, die Groß- und Kleinschreibung ist einheitlich, die View-Boxen sind normalisiert, und für jedes Icon ist die Lizenz festgehalten. Doppelte Konzepte – zwei Such-Glyphen von zwei Designern – sind der Weg, auf dem ein kuratiertes Pack wieder zum Ordner wird.

Die Versionierung folgt daraus, den Katalog als öffentliche API zu behandeln. Eine Konstante hinzuzufügen ist additiv. Die Grafik hinter einer bestehenden Konstante auszutauschen ist eine visuelle Änderung, die in die Release Notes gehört, auch wenn nichts die Kompilierung bricht. Eine Konstante zu entfernen oder umzubenennen ist ein Breaking Change – markieren Sie sie also als veraltet, statt sie zu löschen.

Überschreibungen geben Ihnen zwei dokumentierte Hebel und einen architektonischen: Eine Quelle ohne Assembly-Qualifizierer lässt Wisej.NET eine passende ausgelieferte Datei bevorzugen, ein voll qualifizierter Pfad lässt sich überschreiben, indem die Verzeichnisstruktur im Deployment nachgebildet wird, und ein paralleles Paket – ein kontrastreicher oder kundenspezifischer Satz – liefert dieselben Katalognamen über anderen Grafiken.

- Für jedes Icon gibt es einen Lizenznachweis und eine normalisierte View-Box.
- Der Katalog ist der Vertrag; Grafiken dürfen sich ändern, Namen dürfen nicht still verschwinden.
- Eine Beispielseite zeigt das gesamte Pack in beiden Themes, bevor Sie eine Version taggen.



Zuerst den Mechanismus benennen

Bevor Sie ein Bild zuweisen, sagen Sie, welchen Mechanismus Sie verwenden: ein `System.Drawing.Image`-Objekt, das der Server als PNG rendert, einen Bildquellen-String, den der Browser auflöst, ein benanntes Theme-Bild, eine über `resource.wx` eingebettete Ressource oder einen Icon-Font, der als HTML gezeichnet wird. Weisen Sie pro Steuerelement genau eine Bildeigenschaft zu, damit die anderen sie nicht still zurücksetzen, halten Sie den Quell-String aussagekräftig statt als eingefügten magischen Pfad, bevorzugen Sie SVG für Oberflächen-Icons, damit sie in jeder Größe und Skalierung scharf bleiben, und geben Sie jedem bedeutungstragenden Icon eine Textalternative.



Lernziele

Am Ende dieses Moduls können Sie:

- ✓ Eine Klassenbibliothek erstellen, die Anwendungs-Icons als eingebettete Ressourcen paketierte.

- ✓ Stabile Konstanten bzw. Eigenschaften für Icon-Quellen bereitstellen, für IntelliSense und Wiederverwendung im Code.
- ✓ Das eigene Pack aus dem Designer und aus C# verwenden.
- ✓ Konventionen für Icon-Benennung, Versionierung, Theming und Übersreibungen entwerfen, die ein Produkt wartbar halten.

Lab- und Prüfungsleitfaden

Was Sie im praktischen Lab bauen, das vorgeschlagene Vorgehen und die geforderten Ergebnisse.



Praktisches Lab

Erstellen Sie IconDesk.Icons als Klassenbibliothek neben der Anwendung, legen Sie mindestens zwölf SVG-Icons in ein Verzeichnis Resources und markieren Sie sie als Embedded Resource. Fügen Sie einen statischen Katalog AppIcons hinzu, der jedes Icon als benannten resource.wx-String bereitstellt, sodass niemand mehr einen Pfad tippt, referenzieren Sie die Bibliothek aus IconDesk und weisen Sie dann einige Icons über den Designer und andere in C# aus dem Katalog zu. Prüfen Sie das Farbsuffix an drei Icons und notieren Sie, welche SVG-Bereinigung sie dafür gebraucht haben. Schließen Sie mit den Regeln für Benennung und Versionierung ab, die bestimmen, wie ein neues Icon ins Pack kommt.

Ausgangspunkt

Öffnen Sie die Projektmappe IconDesk in dem Zustand, in dem Modul 5 sie hinterlassen hat: Die Anwendung bettet ihre eigenen Bilder bereits in einem Ordner **Resources** ein und löst sie über **resource.wx** auf, und aus Modul 4 sind zwei offizielle Icon-Packs referenziert. Noch ist nichts in der Projektmappe ein wiederverwendbares Pack, jeder eigene Quell-String wird also weiterhin von Hand auf der Seite getippt, die ihn braucht.

Vorgeschlagenes Vorgehen

- ✓ Fügen Sie der Projektmappe eine Klassenbibliothek namens **IconDesk.Icons** hinzu, die auf ein zu IconDesk passendes Framework zielt, und geben Sie ihr ein Verzeichnis **Resources** mit mindestens zwölf ausgewählten SVG-Icons.
- ✓ Setzen Sie die Build Action jeder Icon-Datei auf **Embedded Resource**, bauen Sie die Bibliothek einmal und prüfen Sie den Ressourcenpfad eines Icons aus der laufenden Anwendung, bevor Sie einen Katalogeintrag schreiben.
- ✓ Schreiben Sie die statische Klasse **AppIcons** mit einer privaten Wurzelkonstante und einer öffentlichen Konstante je Icon, benannt nach der Bedeutung des Icons statt nach der Datei, aus der es stammt.
- ✓ Referenzieren Sie die Bibliothek aus IconDesk, bauen Sie die Projektmappe neu und weisen Sie dann auf einer Seite, die das gesamte Pack zeigt, einige Icons über den Image Selector des Designers und die übrigen in C# aus **AppIcons** zu.
- ✓ Hängen Sie das Farbsuffix an drei Icons an, testen Sie eine Hex-Farbe und einen Theme-Farbnamen in einem hellen und einem dunklen Theme und korrigieren oder befreien Sie jede Grafik, die sich nicht umfärben lässt.

Geforderte Ergebnisse

- ✓ Klassenbibliothek IconDesk.Icons mit mindestens zwölf SVG-Dateien in einem Verzeichnis Resources, alle als Embedded Resource
- ✓ Statischer Katalog AppIcons, der jedes Icon als benannten resource.wx-String bereitstellt

- ✓ IconDesk-Seite, die einige Icons über den Designer und andere aus dem Katalog Applcons zuweist
- ✓ Drei mit dem Farbsuffix geprüfte Icons samt Notiz zur nötigen SVG-Bereinigung
- ✓ Regeln für Benennung und Versionierung, die beschreiben, wie ein neues Icon ins Pack kommt und wie eine Anwendung eines überschreibt

Wiederholungsfragen

- ✓ Ihr Pack baut, und der Designer zeigt im Image Selector nichts an. Welchen Schritt des Rezepts würden Sie zuerst prüfen, und welcher Beleg würde Ihnen sagen, dass die Ressourcen wirklich in der Assembly liegen?
- ✓ Zwei Ihrer drei Test-Icons lassen sich umfärben, das dritte bleibt grau. Was steckt in diesem SVG, und wann würden Sie ein Icon absichtlich nicht umfärbbar lassen?
- ✓ Ein Kunde möchte, dass nur in seinem Deployment drei Icons ersetzt werden. Welchen der Überschreibungswege würden Sie wählen, und was kostet Sie diese Wahl bei der nächsten Pack-Version?



Zuerst den Mechanismus benennen

Bevor Sie ein Bild zuweisen, sagen Sie, welchen Mechanismus Sie verwenden: ein `System.Drawing.Image`-Objekt, das der Server als PNG rendert, einen Bildquellen-String, den der Browser auflöst, ein benanntes Theme-Bild, eine über `resource.wx` eingebettete Ressource oder einen Icon-Font, der als HTML gezeichnet wird. Weisen Sie pro Steuerelement genau eine Bildeigenschaft zu, damit die anderen sie nicht still zurücksetzen, halten Sie den Quell-String aussagekräftig statt als eingefügten magischen Pfad, bevorzugen Sie SVG für Oberflächen-Icons, damit sie in jeder Größe und Skalierung scharf bleiben, und geben Sie jedem bedeutungstragenden Icon eine Textalternative.