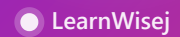


Wiederverwendbare Widget-Klassen und designerfreundliche Eigenschaften

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Von einmaligen Widget-Instanzen zu wiederverwendbaren serverseitigen Klassen mit typisierten Eigenschaften und kontrolliertem Setup übergehen. Was das für Entwickler bedeutet, die mit Wisej.NET integrieren – und wie Sie dieses Modul lesen sollten.



Fokus: Produktivbetrieb

Von einmaligen Widget-Instanzen zu wiederverwendbaren serverseitigen Klassen mit typisierten Eigenschaften und kontrolliertem Setup übergehen.

1. Das Integrationsproblem

Der Prototyp funktioniert, doch dasselbe Widget wird auf vielen Masken eingesetzt. Packages, Options und InitScript in jede Seite zu kopieren, schafft ein Wartungsrisiko.

2. Zentrale Konzepte

Konzept	So setzen Sie es ein
Typisierte Eigenschaften	Stabile .NET-Eigenschaften statt roher Anbieteroptionen bereitstellen.
Ressourcen-Setup	Das Setup von Packages und Skripten innerhalb der Wrapper-Klasse halten.

3. Lektion aus der Praxis

Eine wiederverwendbare Widget-Klasse ist der natürliche nächste Schritt nach einem erfolgreichen Prototyp. Anwendungsentwickler sollten sich nicht für jede Maske die Skriptreihenfolge des Anbieters, den genauen Initialisierungscode oder die rohen Optionsnamen merken müssen. Die Wrapper-Klasse bündelt diese Entscheidungen an einer Stelle. Die Serverklasse leitet weiterhin von `Wisej.Web.Widget` ab, doch ihre öffentliche Oberfläche wird zu einer .NET-API. Eine Gauge kann zum Beispiel `Value`, `Minimum`, `Maximum`, `Label` und `AnimationEnabled` bereitstellen, statt von jedem Entwickler zu verlangen, ein dynamisches Options-Objekt zu bearbeiten. Die Klasse kann für fortgeschrittene Optionen weiterhin einen Notausgang anbieten, der übliche Weg sollte aber typisiert, auffindbar und designerfreundlich sein. Hier kommt es auch auf Benennung, Standardwerte, Kategorien, Beschreibungen und Typkonverter an. Der Wrapper ist kein schneller Skript-Host mehr. Er wird Teil des Anwendungsframeworks, das andere Entwickler nutzen.

Eine Klasse, die ihr Setup selbst verantwortet

Im Lab wird der Prototyp per Refactoring zu `IntegrationLab.Controls.SimpleGauge`, einer Klasse, die von `Wisej.Web.Widget` ableitet. Ihr Konstruktor oder ihre Initialisierungsüberschreibung weist die Packages, das InitScript und die Standard-Options zu, die bisher auf jeder Seite eingetippt wurden. Eine Maske, die eine Gauge braucht, deklariert jetzt ein `SimpleGauge`-Feld, zieht es aus der Toolbox und sieht keine einzige Zeile JavaScript.

Das InitScript selbst wandert in eine eingebettete Ressource oder eine Konstante der Klasse und wird so zusammen mit dem C#-Code versioniert, der davon abhängt. Die Paketpfade des Anbieters werden zu Konstanten an einer einzigen Stelle. Veröffentlicht der Anbieter eine neue Version, ändert sich eine Klasse, und jede Maske übernimmt die Änderung beim nächsten Build.

Eine Wrapper-Klasse ist auch der erste Ort, an dem aus einem Entscheidungsprotokoll Code wird. Klassenname, Namespace und Summary-Kommentar sagen, wofür das Widget gedacht ist und welche Anbieterbibliothek es kapselt – genau das, was ein Entwickler in zwei Jahren als Erstes wissen muss.

Die Notausgänge verbergen

Sobald die Klasse ihr Setup selbst verantwortet, werden die rohen Member gefährlich. Wer Packages an einer Instanz bearbeitet, beschädigt genau diese Instanz und keine andere – die am schwersten zu findende Art von Fehler. Der Wrapper verbirgt oder schützt deshalb Packages, InitScript und das rohe Options-Objekt, sodass sie nicht im Eigenschaftenfenster erscheinen und nicht nebenbei aus dem Anwendungscode geändert werden können.

In der Praxis heißt das, die Member als `protected` zu überschreiben oder so zu kennzeichnen, dass der Designer sie weder anzeigt noch serialisiert. Die öffentliche Oberfläche der Klasse besteht dann nur noch aus den typisierten Eigenschaften und Events, die sie bewusst bereitstellt. Fortgeschrittene Szenarien lassen sich weiterhin über eine `protected`-Methode bedienen, mit der eine abgeleitete Klasse die Optionen erweitern kann; so bleibt der Notausgang für diejenigen erreichbar, die die Folgen verstehen.

Der Selbstcheck im Lab fragt, welche Member vor normalen Verwendern des Wrappers verborgen werden sollten. Die Antwort: jeder Member, dessen falsche Verwendung das Widget unbemerkt beschädigen könnte.

Typisierte Eigenschaften, die Options steuern

Eine typisierte Eigenschaft ist ein kleines Stück C#, das aus dem Options-Objekt liest und hineinschreibt und dabei einen normalen .NET-Typ präsentiert. `Value` ist ein `double` mit Bereichsprüfung, `Minimum` und `Maximum` sind `doubles`, die sich gegenseitig validieren, und `Caption` ist ein `string`. Jeder Setter weist die Option in camelCase zu, und Wisej.NET rendert die Änderung und ruft auf dem Client `update` auf – genau wie beim Prototyp.

Attribute machen die Eigenschaften designerfreundlich. Ein `DefaultValue`-Attribut teilt dem Designer mit, was er nicht serialisieren muss, eine Kategorie gruppiert die Eigenschaften im Eigenschaftenfenster, und eine Beschreibung erklärt jede einzelne im Hilfebereich. Eine Eigenschaft, deren Rendering teuer ist – etwa weil sie den Anbieter zum Neuaufbau zwingt –, sollte das in ihrer Beschreibung sagen und die Option nur zuweisen, wenn sich der Wert tatsächlich geändert hat.

- Welche Eigenschaftsänderungen müssen eine Aktualisierung auf dem Client erzwingen, und welche sind nach dem Erzeugen schreibgeschützt?
- Welche Werte werden im Setter validiert, und welche Ausnahme löst ein ungültiger Wert aus?
- Welche Standardwerte entsprechen denen des Anbieters, damit eine frische Instanz sinnvoll gerendert wird?

Belange des Wrappers und Belange der Anwendung

Der Wrapper ist für den Anbieter zuständig: seine Dateien laden, seine Instanz erzeugen und freigeben, Optionen übersetzen und Events weiterleiten. Die Anwendung ist für die Bedeutung zuständig: welchen Messwert die Gauge anzeigt, welcher Schwellenwert relevant ist und was beim Überschreiten passiert. Ein Wrapper mit einem Datenbankaufruf oder einer Geschäftsregel hat diese Grenze überschritten – ebenso eine Seite, die Initialisierungscode des Anbieters enthält.

Die Demoseite im Lab ist der Beweis. Sie platziert eine `SimpleGauge`, bindet `Value` an einen Messwert aus einem Service und behandelt ein .NET-Event. Auf der Seite gibt es kein InitScript und im Code der Seite keinen Anbieternamen. Würde die Anbieterbibliothek morgen ersetzt, bliebe die Seite unverändert.

Ein nützlicher Grenztest: Stellen Sie sich vor, Sie liefern den Wrapper als NuGet-Paket an ein anderes Team aus. Alles, was dieses Team innerhalb des Pakets bearbeiten müsste, gehört stattdessen in die Anwendung.



Gewohnheit für den Produktivbetrieb

Halten Sie klar getrennt, welchen Zustand Wisej.NET besitzt und wie sich das Client-Widget verhält. Dokumentieren Sie den Client-Server-Vertrag, bevor Sie sich im Produktivbetrieb auf die Integration verlassen.



Typischer Fehler

Demo-JavaScript in Masken kopieren, ohne Lebenszyklus, Ressourcen, Event-Verträge, Aufräumen oder Sicherheitsgrenzen zu dokumentieren.



Lernziele

Am Ende dieses Moduls können Sie:

- ✓ Eine Klasse erstellen, die von `Wisej.Web.Widget` abgeleitet ist.
- ✓ Packages, InitScript und die rohen Options bei Bedarf verbergen oder schützen.
- ✓ Typisierte .NET-Eigenschaften bereitstellen, die das Options-Objekt sicher aktualisieren.
- ✓ Das Ressourcen-Setup des Anbieters von der Verwendung in der Anwendung trennen.

Lab- und Prüfungsleitfaden

Was Sie im praktischen Lab erstellen, der empfohlene Arbeitsablauf, die geforderten Ergebnisse und der Selbstcheck vor der Abgabe.



Praktisches Lab

Überführen Sie den Gauge-Prototyp per Refactoring in `IntegrationLab.Controls.SimpleGauge``, mit typisierten Eigenschaften und standardmäßiger Ressourcenregistrierung.

Empfohlener Arbeitsablauf

- ✓ Öffnen Sie das vorhandene Projekt des Wisej.NET-Integrationskurses und starten Sie es lokal.
- ✓ Ermitteln Sie die Server-Komponente, das Client-Widget, die Ressourcendatei oder den Endpunkt, um die es in diesem Modul geht.
- ✓ Nehmen Sie jeweils nur eine kleine Änderung vor und testen Sie sie im Browser, bevor Sie weitermachen.
- ✓ Halten Sie anbieterspezifisches JavaScript im Adapter oder Wrapper, statt es über mehrere Seiten zu verstreuen.
- ✓ Passen Sie die UI so an, dass Erfolg, Fehler, Fortschritt oder Änderungen in der Kommunikation klar erkennbar sind.
- ✓ Halten Sie mit Screenshots oder Notizen fest, dass die Integration funktioniert, und begründen Sie die Designentscheidung.

Geforderte Ergebnisse

- ✓ Wiederverwendbare, von Widget abgeleitete Klasse
- ✓ Typisierte Eigenschaften mit Standardwerten
- ✓ Demoseite, die die Klasse ohne eigenes InitScript auf der Seite verwendet

Selbstcheck vor der Abgabe

- ✓ Welche Member sollten vor normalen Verwenden des Wrappers verborgen werden?
- ✓ Welche Eigenschaftsänderungen sollten eine Aktualisierung auf dem Client erzwingen?
- ✓ Was gehört in den Wrapper und was in die Anwendung?