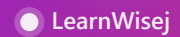


Eigene Widgets, Erweiterungen, Theming und Abschlussprojekt

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Native Controls im Vergleich zu Widgets, Komponenten, Extender Providern und Icon Packs; Packages, InitScript, WidgetEvent und die CallAsync-Anbindung des Widget-Controls; Theming und Dokumentation; das Abschlussprojekt Operations Console. Was das für Wisej.NET-Entwickler bedeutet – und wie Sie dieses Modul lesen.



Fokus: Produktivbetrieb

Native Controls im Vergleich zu Widgets, Komponenten, Extender Providern und Icon Packs; Packages, InitScript, WidgetEvent und die CallAsync-Anbindung des Widget-Controls; Theming und Dokumentation; das Abschlussprojekt Operations Console.

1. Das Problem im Produktivbetrieb

Früher oder später braucht ein Produkt etwas, das die native Bibliothek nicht mitbringt: eine spezialisierte Browserfunktion, eine JavaScript-Bibliothek eines Drittanbieters, eine eigene Visualisierung oder eine wiederverwendbare Integration. Die Versuchung ist groß, in JavaScript abzutauchen und dabei das Wisej.NET-Modell zu verlieren: Serverzustand, Theming, den Designer und einen Event-Fluss, dem ein anderer Entwickler folgen kann. Dieses Modul integriert ein Client-Widget, ohne auf irgendetwas davon zu verzichten, und schließt dann das Abschlussprojekt Operations Console ab.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
natives Control	Ein Control, das sowohl die Serverseite als auch sein Browser-Widget implementiert und daher bereits Designer-Unterstützung, Theming, Events und Serverzustand mitbringt; immer die erste Anlaufstelle.
Erweiterungsarten	Control, Widget, Component, Extender Provider und Icon Pack: Wer die Art benennt, verwechselt „wiederverwendbar“ nicht mehr mit „neues natives Control“.
Widget-Control	Ein universeller Client-Container, der JavaScript- und CSS-Pakete lädt, ein Initialisierungsskript ausführt und Events und Daten mit dem Servercode austauscht.
Packages und InitScript	Die Paketliste deklariert die Client-Dateien, von denen das Widget abhängt; das InitScript erzeugt das JavaScript-Objekt, verknüpft seine Client-Events und stellt nur die Funktionen bereit, die der Server braucht.
WidgetEvent	Das Server-Event, das ausgelöst wird, wenn Client-Code <code>fireWidgetEvent</code> aufruft; Typ und Daten-Payload kommen aus dem Browser und müssen wie jede andere Benutzereingabe validiert werden.
CallAsync / Eval	Die Richtung vom Server zum Client dieser Verbindung: eine vom Widget bereitgestellte Funktion aufrufen oder ein Skript auswerten, ohne C# zu verlassen.
Widget-Vertrag	Die dokumentierten Properties, Events, das Payload-Format, die Paketabhängigkeiten und Theming-Regeln, mit denen ein anderer Entwickler oder ein fundierter KI-Assistent das Widget pflegen kann.

3. Der rote Faden der Lektion

Zuerst nativ, dann fünf Arten von Erweiterungen

Die Control-Bibliothek ist jedes Mal die erste Anlaufstelle. Ein natives Control implementiert Server- und Clientseite gemeinsam und nimmt deshalb ganz selbstverständlich am Visual-Studio-Designer, an `Application.Theme`, am Event-Modell und am sessionbezogenen Serverzustand teil. Eigener Code bleibt dem vorbehalten, was die Bibliothek nicht abdeckt: einer Browserfunktion ohne native Entsprechung, einer JavaScript-Bibliothek eines Drittanbieters, einer maßgeschneiderten Visualisierung oder einer Integration, die Sie wiederverwenden möchten.

Wenn eigener Code gerechtfertigt ist, benennen Sie die Art der Erweiterung, die Sie bauen. Ein **Control** ist ein vollständiges natives Control mit einer Serverklasse und einem Client-Widget. Ein **Widget** ist der universelle Client-Container für ein JavaScript-Widget eines Drittanbieters oder ein eigenes. Eine **Component** kann nur auf dem Server oder auch auf dem Client leben. Ein **Extender Provider** fügt vorhandenen Controls Properties und Verhalten hinzu, so wie `ToolTip` oder `ErrorProvider`. Ein **Icon Pack** verpackt visuelle Ressourcen für Projekt und Designer. Diese Einteilung verhindert Verwirrung im Design: Nicht alles Wiederverwendbare ist ein `UserControl`, und nicht jede JavaScript-Integration braucht ein neues natives Control.

Für die Operations Console sind die Entscheidungsregeln kurz. Verwenden Sie zuerst native Controls. Verwenden Sie ein `UserControl` für eine wiederverwendbare serverseitige Komposition wie den Kundeneditor aus Modul 2. Verwenden Sie `Widget` für JavaScript-Oberflächen von Drittanbietern, die in einem Wisej.NET-Container leben können. Bauen Sie eine vollständige Erweiterung nur, wenn die Integration anwendungsübergreifend wiederverwendet und als vollwertige Komponente erscheinen soll.

Die Widget-Anbindung in beide Richtungen

Das `Widget` -Control erledigt drei Dinge. Seine `Packages` -Collection lädt die JavaScript- und CSS-Dateien, die die Client-Bibliothek braucht, in der richtigen Reihenfolge. Sein `InitScript` läuft, sobald die Pakete vorhanden sind: `this.init` erzeugt das Client-Objekt innerhalb von `this.container`, verknüpft die eigenen Events der Bibliothek und stellt die wenigen Funktionen bereit, die der Server aufrufen darf. Schließlich transportiert es Nachrichten: Client-Code ruft `this.fireWidgetEvent("ratingChanged", { value: 4 })` auf, und der Server empfängt ein `WidgetEvent`, dessen `WidgetEventArgs` den `Type` und die `Data` -Payload bereitstellen; der Servercode antwortet mit `CallAsync`, um eine Client-Funktion aufzurufen, oder mit `Eval`, um ein Skript auszuführen.

Das Bewertungs-Widget aus dem Lab zeigt den kompletten Hin- und Rückweg: Ein Klick auf einen Stern löst das Event aus, der Handler validiert den Wert, speichert ihn über einen Service und bestätigt ihn dem Widget:

```
private async void ratingWidget_WidgetEvent(object sender, WidgetEventArgs e)
{
    if (e.Type != "ratingChanged") return;

    // Die Payload kommt aus dem Browser: vor dem Vertrauen validieren.
    int value = Convert.ToInt32(e.Data.value);
    if (value < 1 || value > 5) return;

    await _ratings.SaveAsync(_customerId, value);
    await this.ratingWidget.CallAsync("setSaved", value);
    MessageBox.Show($"Rating {value} saved.");
}
```

Nutzen Sie die Verbindung mit Bedacht. JavaScript bleibt auf reine Browseraufgaben konzentriert: Rendering, Gesten, Animation. Geschäftsregeln bleiben auf dem Server, wo sie abgesichert, getestet und mit jedem Bildschirm geteilt werden; duplizieren Sie eine Regel in JavaScript nur als bewusste clientseitige Optimierung, niemals als einzige Kopie. Ein klassischer Fehler: Innerhalb eines Bibliotheks-Callbacks ist `this` das Objekt der Bibliothek, nicht das Widget; halten Sie daher `var me = this` in `init` fest und rufen Sie `me.fireWidgetEvent` auf.

Ein Vertrag, den Sie debuggen, gestalten und dokumentieren können

Ein robuster Wrapper hat einen Vertrag. Öffentliche serverseitige Properties entsprechen sinnvoller Konfiguration, nicht jeder Option, die die Bibliothek anbietet. Events stehen für fachlich relevante Benutzeraktionen: `ratingChanged`, nicht `mouseup`. Umfassen Payloads mehr als ein oder zwei Werte, definieren Sie ein DTO oder eine klare JSON-Struktur; der Handler validiert und normalisiert die Daten vor der Verwendung, weil jeder mit Entwicklerwerkzeugen eine Browser-Payload verändern kann.

Das Debugging von Widgets beginnt in der Browserkonsole, und die Wisej.NET-Dokumentation sagt das ausdrücklich. Wenn ein Widget leer erscheint oder ein Event nie ankommt, prüfen Sie zuerst die Konsole, dann Ladereihenfolge der Pakete, Ressourcenpfade und Initialisierungszeitpunkt, und schließlich, ob das Widget nach einer Aktualisierung oder Layoutänderung neu erstellt wird – was bedeutet, dass `init` gefahrlos erneut laufen können muss. Skripte, die ins Start-HTML gelegt werden, können für Wisej.NET-Widgets zu früh laufen; eingebettete Ressourcen, `Packages`, `InitScript` und Aufrufe auf Control-Ebene sind die sichereren Muster.

Theming und responsives Verhalten sind der letzte Feinschliff. Wenn das Widget zur Konsole passen muss, entscheiden Sie bewusst: Style-Properties bereitstellen, auf Theme-Appearances abbilden, die Extender StyleSheet oder Animation verwenden oder CSS im Widget-Paket mitliefern. Codieren Sie Anwendungsfarben nicht fest im JavaScript, es sei denn, das Widget ist bewusst unabhängig. Wenden Sie wie bei der Shell responsive Properties pro Client-Profil an, sorgen Sie, wo praktikabel, für barrierefreie Beschriftungen und Tastaturbedienung, und schreiben Sie die Notiz zur KI-Fundierung: wofür das Widget gedacht ist, warum kein natives Control passte und wo der Vertrag dokumentiert ist.

- Paketabhängigkeiten und Ladereihenfolge sind dokumentiert.
- Properties, Events und das Payload-Format stehen direkt neben dem Code.
- Geschäftsregeln werden serverseitig durchgesetzt, und die Browserkonsole ist sauber.



Typische Fehler

Standard-Controls durch JavaScript ersetzen, weil es modern wirkt. Geschäftsregeln in Client-JavaScript legen, wo sie schwerer abzusichern und zu testen sind. Die Neuinitialisierung des Widgets nach einer Aktualisierung oder Layoutänderung nicht behandeln. Vergessen, die Paketabhängigkeiten und den Event-Vertrag zu dokumentieren. Ein eigenes Widget bauen, das sonst niemand im Team pflegen kann.

Die Operations Console fertigstellen

Das Abschlussprojekt vereint den Kurs in einer einzigen Anwendung, `OperationsConsole`: die navigierbare Shell (Module 1 und 3), den validierten Kundeneditor (Modul 2), den Explorer mit TreeView und ListView (Modul 4), die Orders- `DataGridView` (Modul 5), den Dashboard- und Inhaltsbildschirm (Modul 6) und den Bildschirm mit dem eigenen Widget oder der Erweiterung aus diesem Modul. Jeder Bildschirm wird an den Fragen gemessen, die die Labs gestellt haben: Welches Control leistet die wichtigste Arbeit, wo liegt der Zustand, was passiert bei zwei Sessions oder zehnfacher Datenmenge, und welcher Code gehört in ein wiederverwendbares `UserControl` oder einen Service?

Das Bewertungsschema gewichtet Control-Auswahl und Daten-Controls mit je 20 % sowie Layout und Wiederverwendung, Validierung und Feedback, den Vertrag des eigenen Widgets und Feinschliff und Dokumentation mit je 15 %. Bereiten Sie eine fünfminütige Demo nach dem Skript für das Abschlussprojekt vor und schreiben Sie `CapstoneNotes.md` nach der Vorlage im Arbeitsheft: verwendete Controls und warum, Annahmen zur Datenmenge, Validierungs- und Layout-Entscheidungen, ausgelagerter wiederverwendbarer Code und eine Sache, die Sie für den Produktivbetrieb verbessern würden. Der Code sollte für einen anderen Entwickler verständlich und von einem KI-Assistenten erklärbar sein, der auf Ihren Notizen aufbaut.



Gewohnheit für die Control-Bibliothek

Wählen Sie das einfachste native Control, das die Aufgabe des Benutzers abbildet, bevor Sie eigenen Code schreiben. Native Controls bringen Designer-Unterstützung, Theming, Events und Serverzustand bereits mit; greifen Sie als Nächstes zu einem Extender oder einem UserControl und bauen Sie ein JavaScript-Widget erst, wenn kein natives Control passt.



Lernziele

Am Ende dieses Moduls können Sie:

- ✓ den Unterschied zwischen nativen Controls, Widgets, Komponenten, Extender Providern und Icon Packs erklären.
- ✓ mit dem Widget-Control JavaScript-/CSS-Pakete laden, ein Client-Widget initialisieren und Events und Daten mit dem Servercode austauschen.
- ✓ Theming, responsive Properties, Hinweise zur Barrierefreiheit und Notizen zur KI-Fundierung auf ein eigenes Control-Paket anwenden.
- ✓ das Abschlussprojekt Operations Console mit nativen Controls und mindestens einem eigenen Widget oder einer Erweiterung fertigstellen und präsentieren.

Lab- und Prüfungsleitfaden

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



Praktisches Lab

Kapseln Sie ein kleines JavaScript-Bewertungs- oder Anzeige-Widget mit dem Widget-Control oder konfigurieren Sie ein Wisej.NET-Erweiterungspaket: Laden Sie seine JavaScript- und CSS-Pakete, lösen Sie mit dem Widget-Event-Muster vom Client aus ein serverseitiges Event aus, rufen Sie mit CallAsync oder Eval vom Server aus eine clientseitige Funktion auf, passen Sie das Theme an die Konsole an, teilen Sie dann am vollständigen Abschlussprojekt Operations Console und bereiten Sie eine 5-minütige Demo vor.

Ausgangspunkt

Öffnen Sie die Projektmappe `OperationsConsole`, die Sie durch die Module 1 bis 6 mitgenommen haben: Shell, Kundeneditor, Dokument-Explorer, Orders-Grid und Dashboard existieren bereits, und der Tab Widgets ist noch die Platzhalterseite aus Modul 1. Legen Sie die Arbeit dieses Moduls in einen eigenen Ordner oder Namespace, damit das Abschlussprojekt sie wiederverwenden kann.

Vorgeschlagenes Vorgehen

- ✓ Entscheiden und notieren Sie, warum kein natives Control zu der Bewertung oder Anzeige passt, die Sie hinzufügen; passt doch ein natives Control, verwenden Sie es und wählen Sie ein anderes Widget als Ziel.
- ✓ Ziehen Sie ein `Widget`-Control auf die Seite Widgets, nehmen Sie die JavaScript- und CSS-Dateien in Ladereihenfolge in seine `Packages`-Collection auf und schreiben Sie ein `InitScript`, dessen `init` das Client-Objekt innerhalb von `this.container` erzeugt und nach einer Aktualisierung gefahrlos erneut laufen kann.
- ✓ Halten Sie im Client-Callback `var me = this` fest und rufen Sie `me.fireWidgetEvent("ratingChanged", { value })` auf; behandeln Sie auf dem Server `WidgetEvent`, prüfen Sie `e.Type`, validieren und normalisieren Sie `e.Data` und speichern Sie dann über einen Service.
- ✓ Bestätigen Sie dem Browser das Ergebnis mit `CallAsync("setSaved", value)` oder `Eval`, geben Sie dem Benutzer einen Toast oder eine MessageBox und gestalten Sie das Widget über Theme-Appearances, paketierte CSS oder den StyleSheet-Extender statt über in JavaScript fest codierte Farben.
- ✓ Schreiben Sie den Vertrag direkt neben den Code (Pakete, Properties, Events, Payload-Format, Hinweise zum Debugging), gehen Sie dann jeden Bildschirm des Abschlussprojekts anhand des Bewertungsschemas durch und proben Sie das Skript für die fünfminütige Demo.

Geforderte Ergebnisse

- ✓ Bildschirm auf Basis eines Widgets oder einer Erweiterung mit geladenen JavaScript- und CSS-Paketen
- ✓ Event vom Client zum Server über das Widget-Event-Muster oder ein natives Erweiterungs-Event

- ✓ Aufruf vom Server zum Client über CallAsync, Eval oder die API der Erweiterung
- ✓ Dokumentierter Client/Server-Vertrag: Pakete, Properties, Events und Payload-Format
- ✓ Ausgefeiltes Abschlussprojekt Operations Console mit einem Skript für eine 5-minütige Demo

Wiederholungsfragen

- ✓ Was würde passieren, wenn ein Benutzer die `ratingChanged` -Payload in den Entwicklerwerkzeugen des Browsers verändert, bevor sie Ihren Handler erreicht, und welche Zeile Ihres Codes hält das auf?
- ✓ Ihr Widget erscheint leer, nachdem der Tab Widgets aus- und wieder eingeblendet wurde: Welche drei Dinge prüfen Sie in welcher Reihenfolge, und warum ist das Start-HTML nicht die Lösung?
- ✓ Welche Regel Ihres Widgets könnten Sie als Optimierung nach JavaScript verlagern, und welche darf den Server niemals verlassen?

Abgabe des Abschlussprojekts

Das Lab dieses Moduls ist zugleich die Abgabe des Abschlussprojekts. Die Operations Console muss sechs Bildschirme enthalten: die Anwendungs-Shell mit Navigations-, Befehls- und Statusbereich; den validierten Kunden- oder Kontaktditor; den TreeView/ListView-Explorer mit stabilen IDs und einer Strategie für verzögertes Laden oder Virtualisierung; die Orders-DataGridView mit expliziten Spalten, Formatierung und einer Strategie für große Datenmengen; den Dashboard- und Inhaltsbildschirm mit ChartJS und einer Dokumentvorschau; und den Bildschirm mit dem eigenen Widget oder der Erweiterung aus diesem Lab. Reichen Sie das Quellprojekt, Screenshots jedes Bildschirms, eine nach der Vorlage im Arbeitsheft geschriebene `CapstoneNotes.md` und eine fünfminütige Demo (live oder aufgezeichnet) ein, die diesem Skript folgt:

- ✓ Erklären Sie die Control-Architektur der Anwendung und zeigen Sie dann den Editor und sein Validierungsverhalten.
- ✓ Zeigen Sie die Navigation in Baum und Liste mit Laden der Details, dann das Grid und wie es mit großen Datenmengen umgeht.
- ✓ Zeigen Sie die Dashboard- und Inhalts-Controls, dann das eigene Widget oder die Erweiterung einschließlich der Event-Kommunikation in beide Richtungen.
- ✓ Erklären Sie, was Ihr KI-Assistent antworten sollte, wenn er gefragt wird, warum jedes Control gewählt wurde.

Bewertungsbereich	Gewichtung	So sieht „hervorragend“ aus
Control-Auswahl	20 %	Native Controls, Extender, UserControls, Widgets und Erweiterungen werden angemessen und mit klarer Begründung eingesetzt.
Layout und Wiederverwendung	15 %	Container, Docking, Anchoring, Splitter, Tabs und UserControls werden sauber eingesetzt.
Daten-Controls	20 %	Muster für TreeView, ListView und DataGridView mit korrekter Identität, korrektem Binding und einer Strategie für große Datenmengen.
Validierung und Feedback	15 %	Editor-Properties, ErrorProvider und Benachrichtigungen werden klar und konsistent eingesetzt.
Eigenes Widget oder Erweiterung	15 %	Ein dokumentierter Client/Server-Vertrag, behandelte Events, keine falsch platzierte Geschäftslogik.
Feinschliff und Dokumentation	15 %	Theme, responsives Verhalten, Kommentare und Notizen zur KI-Fundierung sind auf den Produktivbetrieb ausgerichtet.



Gewohnheit für die Control-Bibliothek

Wählen Sie das einfachste native Control, das die Aufgabe des Benutzers abbildet, bevor Sie eigenen Code schreiben. Native Controls bringen Designer-Unterstützung, Theming, Events und Serverzustand bereits mit; greifen Sie als Nächstes zu einem Extender oder einem UserControl und bauen Sie ein JavaScript-Widget erst, wenn kein natives Control passt.