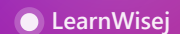


JavaScript-Integration mit dem Widget

Native Controls oder das Widget · Sauberes C# und Widget-Sicherheit



Native Controls oder das Widget

Wann Sie native Wisej.NET-Controls und wann ein eigenes JavaScript-Widget verwenden, die JavaScript-Datei, die das Widget erzeugt (ein vollständiges init/update-Beispiel), und wie Sie es aus C# über Packages, InitScript und Options anbinden.



Lernziel der Lektion

Entscheiden Sie, wann eine Seite native Wisej.NET-Controls verwenden sollte und wann ein eigenes JavaScript-**Widget** besser passt – schreiben Sie dann die JavaScript-Datei, die das Widget *ist* (mit ihren Funktionen `init` und `update`), und verbinden Sie diese Datei über Packages, InitScript und Options mit C#.

1. Zuerst native Wisej.NET-Controls verwenden

Die meisten Geschäftsanwendungsseiten sollten mit den integrierten Wisej.NET-Controls beginnen – Labels, Textfelder, Buttons, Listen, Grids, Panels, Tabs, Dialoge und Menüs. Sie sind leichter zu warten und zu debuggen und passen bereits zum Event-Modell von Wisej.NET. Ein **Widget** ist für die Fälle gedacht, in denen Sie eine browserseitige visuelle Funktion brauchen, die normale Controls nicht sauber abbilden.

Anforderung	Empfohlene Wahl	Begründung
Ticket-Formular, Einstellungsseite, normale Buttons	Native Wisej.NET-Controls	Schnell erstellt und für Junior-Entwickler leicht verständlich.
Ticketliste oder Geschäftstabelle	DataGridView oder native Listen-/Grid-Controls	Funktioniert direkt mit C#-Data-Binding und Auswahl-Events.
Individuelles Diagramm, Anzeige, Karte, Zeitleiste oder spezielle Visualisierung	Widget	JavaScript-Bibliotheken eignen sich für spezialisierte Visualisierungen oft besser.
Einmaliger optischer Effekt ohne fachlichen Nutzen	Vermeiden	Holen Sie sich kein Wartungsrisiko ins Haus, nur damit die Oberfläche auffälliger wirkt.

2. Wie sich ein Widget in eine Wisej.NET-Anwendung einfügt

Halten Sie die normale Geschäftslogik in C#. Legen Sie *nur* die visuelle JavaScript-Integration in den Widget-Wrapper. Die Verbindung zwischen beiden sollte schmal und bewusst gestaltet sein.

Wisej.NET-Seite → C#-Controls und Layout
Widget-Wrapper → lädt JS/CSS und stellt Options bereit
JavaScript-UI → Diagramm / Statusanzeige / Zeitleiste
Server-Events → Aktualisierung, Status und Datenänderungen

Die Wisej.NET-Seite und die Services verantworten weiterhin die Geschäftsdaten; das Widget verantwortet ein kleines browserseitiges visuelles Element. Senden Sie nur die Daten, die das Widget braucht, hören Sie nur auf Events, die wirklich wichtig sind, und lassen Sie den Rest der App normales C# bleiben.

3. JavaScript und CSS laden

Eine Integration im Widget-Stil braucht in der Regel Script-Dateien, Stylesheet-Dateien und Konfigurationsdaten. Das Script kann eine lokale Datei oder eine vertrauenswürdige Bibliothek eines Drittanbieters sein; das CSS steuert das Aussehen; die Konfiguration sagt dem Widget, was es zeichnen soll.

Teil	Was er tut	Regel für Einsteiger
JavaScript-Datei	Erzeugt oder aktualisiert das individuelle visuelle Element.	Klein halten, klar benennen und auf das Widget beschränken.
CSS-Datei	Gestaltet das Widget-Element.	Styles vermeiden, die versehentlich die ganze App verändern.
Konfiguration / Options	Übergibt Werte aus C# an das Widget.	Einfache Daten senden: Zahlen, Beschriftungen und Status-Strings.
Events / Callbacks	Lassen das Widget dem Server mitteilen, dass etwas passiert ist.	Nur Events verdrahten, die einen echten Arbeitsablauf unterstützen.

Auf der C#-Seite entsprechen diese Teile drei Mitgliedern des Controls **Widget** : **Packages** (die zu ladenden Script- und Stylesheet-Dateien, in der richtigen Reihenfolge), **InitScript** (das JavaScript, das die Visualisierung erzeugt) und **Options** (das Datenobjekt, das serialisiert an den Browser geht). Im weiteren Verlauf dieser Lektüre bauen wir jeden dieser Teile für eine kleine *Statusanzeige* auf – dasselbe Widget, das Sie im Lab erstellen.

4. Die JavaScript-Datei schreiben – das eigentliche Widget

In der JavaScript-Datei entsteht die Visualisierung tatsächlich. Wisej.NET führt Ihr **InitScript** einmal im Browser aus, wobei **this** auf das clientseitige Widget-Objekt zeigt, und erwartet, dass Sie darauf **zwei Funktionen** definieren:

Funktion	Wann Wisej.NET sie aufruft	Was Sie darin tun
<code>this.init(options)</code>	Einmal, nachdem die Packages geladen sind und das DOM-Element des Widgets existiert.	Die Visualisierung in <code>this.container</code> erzeugen und die ersten Werte aus <code>options</code> zeichnen.
<code>this.update(options, old)</code>	Jedes Mal, wenn C# <code>Options</code> ändert und <code>Update()</code> aufruft.	Nur das neu zeichnen, was sich geändert hat. Ohne diese Funktion müsste das Widget bei geänderten Options neu erzeugt werden.

Drei Namen sollten Sie sich für diese Datei merken: `this.container` ist das Element, das Wisej.NET für Ihren Inhalt reserviert (erzeugen Sie Ihr Markup *darin* und ersetzen Sie niemals das eigene Wurzelement des Widgets); `options` ist das einfache JavaScript-Objekt, das C# an `Options` zugewiesen hat, mit Eigenschaftsnamen in camelCase (aus `Percent` wird `percent`); und `this.fireWidgetEvent(name, data)` sendet ein kleines Event an den Server zurück, das dort als C#- `WidgetEvent` ankommt.

Widgets/statusGauge.js – eine vollständige Beispieldatei (ohne Bibliothek eines Drittanbieters)

```
// Wisej.NET-Widget-InitScript für eine einfache Statusanzeige.
// Läuft im Browser. "this" ist das clientseitige Widget-Objekt.
// Options aus C# kommen so an: { percent: 0-100, label: "text", status: "ok|warn|error" }

// Wird EINMAL aufgerufen, nachdem die Packages geladen sind und das Widget-Element existiert.
this.init = function (options) {
    // Referenz auf das Widget merken: In DOM-Callbacks ändert sich "this".
    var me = this;

    // Die Visualisierung in dem Container aufbauen, den Wisej.NET für uns reserviert.
    me.container.innerHTML =
        '<div class="lw-gauge">' +
        '  <div class="lw-gauge-track"><div class="lw-gauge-fill"></div></div>' +
        '  <div class="lw-gauge-label"></div>' +
        '</div>';

    // Die Elemente zwischenspeichern, die wir später aktualisieren.
    me.fillEl = me.container.querySelector('.lw-gauge-fill');
    me.labelEl = me.container.querySelector('.lw-gauge-label');

    // EIN Event verdrahten, das einen echten Ablauf unterstützt: Klick → Server benachrichtigen.
    me.container.addEventListener('click', function () {
        // Nur eine kleine Nutzlast. Kommt in C# als WidgetEvent mit e.Type == "gaugeClick" an.
        me.fireWidgetEvent('gaugeClick', { percent: me.lastPercent || 0 });
    });

    // Erstes Zeichnen mit den Werten, die C# gesendet hat.
    me.render(options);
};

// Wird JEDES MAL aufgerufen, wenn C# Options ändert und widStatus.Update() aufruft.
```

```

this.update = function (options, old) {
    // Nur neu zeichnen, wenn sich etwas Angezeigtes tatsächlich geändert hat.
    if (!old || options.percent !== old.percent ||
        options.label !== old.label || options.status !== old.status) {
        this.render(options);
    }
};

// Optionale Hilfsfunktion: eine benannte Funktion, die der Server mit
widStatus.Call("pulse") aufrufen kann.
this.pulse = function () {
    var el = this.container.querySelector('.lw-gauge');
    if (!el) return;
    el.classList.remove('lw-gauge-pulse');
    void el.offsetWidth; // CSS-Animation neu starten
    el.classList.add('lw-gauge-pulse');
};

// Unsere eigene Zeichenroutine: reine Anzeigelogik, keine Geschäftsregeln.
this.render = function (options) {
    options = options || {};
    var percent = Math.max(0, Math.min(100, Number(options.percent) || 0));
    this.lastPercent = percent;

    this.fillEl.style.width = percent + '%';
    this.labelEl.textContent = (options.label || 'Status') + ' – ' + percent + '%';

    // Der Status wählt nur eine Farbklasse; WELCHER Status gilt, hat C# entschieden.
    this.fillEl.className = 'lw-gauge-fill lw-gauge-' + (options.status || 'ok');
};

```

Widgets/statusGauge.css – auf die eigenen Klassennamen des Widgets begrenzt

```

.lw-gauge      { font: 13px/1.4 inherit; padding: 8px; }
.lw-gauge-track { height: 14px; border-radius: 7px; background: #e6e6ee; overflow: hidden; }
.lw-gauge-fill  { height: 100%; width: 0; border-radius: 7px; transition: width .4s ease; }
.lw-gauge-ok    { background: #1f9d57; }
.lw-gauge-warn  { background: #e8a317; }
.lw-gauge-error { background: #d64545; }
.lw-gauge-label { margin-top: 6px; color: #444; }
.lw-gauge-pulse { animation: lwPulse .6s ease; }
@keyframes lwPulse { 50% { opacity: .5; } }

```



Warum var me = this?

Innerhalb von `init` ist `this` das Wisej.NET-Widget. In einem Browser-Callback wie dem Click-Listener ist `this` dagegen das Element, das das Event ausgelöst hat – `this.fireWidgetEvent` wäre dort also undefiniert. Speichern Sie das Widget deshalb zuerst in `me` und verwenden Sie `me` in jedem Callback. Beachten Sie außerdem, dass die Datei nie selbst entscheidet, ob ein Wert „warn“ oder „error“ ist – C# legt den Status fest, und das JavaScript wählt nur eine Farbe dafür aus.

Versehen Sie Ihre CSS-Klassennamen mit einem Präfix (hier `lw-gauge-...`), damit das Stylesheet nicht versehentlich den Rest der Anwendung umgestaltet. Halten Sie die Datei frei von Geschäftsregeln, Secrets und Server-URLs – jeder Entwickler sollte sie als „zeichne diese Zahlen“ lesen können.

5. Die Datei mit C# verbinden

Fügen Sie der Seite ein `Widget`-Control hinzu (nennen Sie es `widStatus`) und verdrahten Sie dann die beiden Dateien und die ersten Options im Konstruktor oder im `Load`-Handler der Seite. Das CSS gehört in **Packages**; die JavaScript-Datei wird zum **InitScript**; die Daten kommen in **Options**.

```
// C# – StatusPage.cs (beide Dateien liegen im Ordner /Widgets des Webprojekts)
private void StatusPage_Load(object sender, EventArgs e)
{
    // 1. Stylesheet (und eventuelle Scripts von Drittanbietern) werden in Listenreihenfolge
    // geladen, einmal pro Session.
    widStatus.Packages.Add(new Package {
        Name = "statusGauge-css",
        Source = "Widgets/statusGauge.css"
    });

    // 2. Die JavaScript-Datei IST das Widget: als InitScript laden.
    // (Sie können die Datei auch im Designer in die Eigenschaft InitScript einfügen.)
    widStatus.InitScript = File.ReadAllText(Application.MapPath("Widgets/statusGauge.js"));

    // 3. Erste Options – nur unbedenkliche Anzeigewerte, vorbereitet von einem C#-Service.
    var data = statusService.GetStatus();
    widStatus.Options = new { percent = data.Percent, label = data.Label, status =
    data.Status };

    // 4. Nur auf das Event hören, das wir in der JS-Datei verdrahtet haben.
    widStatus.WidgetEvent += widStatus_WidgetEvent;
}

private void btnRefresh_Click(object sender, EventArgs e)
{
    var data = statusService.GetStatus(); // C# verantwortet Daten und Regeln
    widStatus.Options = new { percent = data.Percent, label = data.Label, status =
    data.Status };
    widStatus.Update(); // führt this.update(options, old) im
    // Browser aus
    widStatus.Call("pulse"); // führt this.pulse() im Browser aus
    AddLog("Widget refreshed with server data");
}

private void widStatus_WidgetEvent(object sender, WidgetEventArgs e)
{
    if (e.Type == "gaugeClick")
    {
        // e.Data ist das kleine Objekt, das das JS gesendet hat: { percent: 42 }
        int percent = e.Data.percent;
        AddLog($"Gauge clicked at {percent}%");
    }
}
```

```
}  
}
```

Achten Sie darauf, wie beide Seiten zusammenpassen: `Options = new { percent, label, status }` in C# ist genau das `options` -Objekt, das das JavaScript liest; `Update()` löst `this.update` aus; `Call("pulse")` führt `this.pulse` aus; und `fireWidgetEvent('gaugeClick', ...)` kommt als `WidgetEvent` mit `e.Type == "gaugeClick"` an. Setzen Sie die Options, *bevor* Sie `Update()` oder `Call()` aufrufen – Wisej.NET sendet alles gebündelt an den Browser und wendet es in dieser Reihenfolge an.

6. Was zur Laufzeit passiert

1. Die Seite wird gerendert, und der Browser erzeugt das Element des Widgets.
2. Wisej.NET lädt jeden Eintrag in **Packages** der Reihe nach (hier das CSS).
3. Es führt das **InitScript** aus, das `init`, `update`, `pulse` und `render` auf dem Widget definiert.
4. Es ruft `init(options)` mit den serialisierten **Options** auf – die Anzeige erscheint.
5. Jedes Mal, wenn C# `Options` setzt und `Update()` aufruft, führt der Browser `update(options, old)` aus.
6. Ein Klick im Browser löst `gaugeClick` aus; C# behandelt es in `WidgetEvent` wie jeden anderen Button-Klick.



Wenn das Widget leer bleibt

Öffnen Sie die Entwicklertools des Browsers. Ein Fehler beim Laden, der ein undefiniertes Symbol nennt, deutet auf ein Problem mit der Reihenfolge der Packages hin; „fireWidgetEvent is not a function“ in einem Callback bedeutet ein verlorenes `this` (verwenden Sie `me`); ein leerer Container ohne Fehler heißt meist, dass `init` die benötigte Option nie erhalten hat – geben Sie `options` am Anfang von `init` im Log aus, um zu sehen, was C# tatsächlich gesendet hat.



Sie sind bereit für die nächste Lektüre

Sie können zwischen nativen Controls und einem Widget wählen, die JavaScript-Datei des Widgets mit `init` und `update` schreiben und sie aus C# über Packages, `InitScript`, `Options`, `Update()`, `Call()` und `WidgetEvent` anbinden. Als Nächstes: wie der Rest der App sauberes C# bleibt und welche Sicherheitsrisiken Scripts von Drittanbietern mit sich bringen.

Sauberes C# und Widget-Sicherheit

Halten Sie Geschäftsregeln in C# (nicht in JavaScript), kapseln Sie Integrationscode und erkennen Sie die Sicherheitsrisiken von Skripten Dritter.



Lernziel der Lektion

Halten Sie Geschäftsregeln in C# (nicht in JavaScript), kapseln Sie den Integrationscode, damit er sich nicht verstreut, und erkennen Sie die Sicherheits- und Wartungsrisiken, die Scripts von Drittanbietern in einer Geschäftsanwendung mit sich bringen.

1. Der Rest der App bleibt sauberes C#

Das Widget sollte nicht zu einer versteckten zweiten Anwendung innerhalb Ihrer Wisej.NET-App werden. Legen Sie keine Geschäftsregeln, Ticket-Berechnungen, Berechtigungen oder Datenbanklogik in JavaScript ab – behalten Sie diese in C#-Services und übergeben Sie dem Widget nur die fertigen Anzeigedaten.

Schlechtes Muster	Besseres Muster
Geschäftsregeln in einen großen JavaScript-Block schreiben.	Fachliche Werte in einem C#-Service berechnen und dann die Anzeigewerte an das Widget senden.
Beliebige Scripts von unbekannten Websites laden.	Vertrauenswürdige Bibliotheken verwenden, versionieren und dokumentieren, warum sie gebraucht werden.
Jede Seite ihre eigenen Script-Strings von Hand zusammenbauen lassen.	Den Integrationscode in einer wiederverwendbaren Widget-Hilfsklasse / einem Control kapseln.
Secrets oder interne Serverdaten an JavaScript weitergeben.	Nur unbedenkliche Anzeigedaten an den Browser senden.

2. Sicherheits- und Wartungsrisiken

Scripts von Drittanbietern bringen zusätzliche Möglichkeiten, aber auch Risiken – sie können das Verhalten der Seite verändern, nach einem Update der Bibliothek nicht mehr funktionieren, mit Styles kollidieren oder Daten preisgeben, wenn die falschen Informationen an den Browser gesendet werden. Behandeln Sie jedes externe Script als Abhängigkeit, die geprüft und gepflegt werden muss.

- Verwenden Sie vertrauenswürdige Bibliotheken; kopieren Sie keine unbekannten Scripts ins Projekt.
- Bevorzugen Sie lokale, versionierte Dateien, wenn Sie ein vorhersehbares Verhalten brauchen.
- Senden Sie **niemals** API-Keys, Passwörter, Connection Strings oder private interne Daten an das Browser-Widget.
- Dokumentieren Sie, was das Script tut und welche Seite es verwendet.
- Testen Sie die Seite nach Theme-Wechseln, Layout-Änderungen und Browser-Updates.



Geben Sie dem Browser niemals Secrets preis

Alles, was an das Widget gesendet wird, landet im Browser. Bereiten Sie unbedenkliche Anzeigedaten in C# vor und senden Sie nur das, was das Widget zum Zeichnen braucht.

3. Das Einsteigermuster zum Merken

Native Controls sind der Standard. Das Widget ist für spezielle Visualisierungen da. C# verantwortet Daten und Regeln; JavaScript verantwortet das visuelle Rendern. Die Integrationsgrenze sollte klein, klar benannt und leicht zu testen sein.



Sie sind bereit für das Lab

Als Nächstes: Sehen Sie sich den Video-Walkthrough zum Widget an, bestehen Sie den Wissenstest und erstellen Sie dann ein kleines JavaScript-Status- bzw. Diagramm-Widget – gespeist mit unbedenklichen Anzeigedaten aus einem C#-Service, aktualisiert über ein Server-Event und mit einer Datenkarte, um die Werte zu überprüfen.