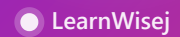


JavaScript-Grundlagen für zuverlässige Integrationen

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Die JavaScript-Muster beherrschen, die bei Integrationen von Drittanbietern immer wieder auftauchen. Was das für Entwickler bedeutet, die mit Wisej.NET integrieren – und wie Sie dieses Modul lesen sollten.



Fokus: Produktivbetrieb

Die JavaScript-Muster beherrschen, die bei Integrationen von Drittanbietern immer wieder auftauchen.

1. Das Integrationsproblem

Die meisten Integrationsfehler sind schlichte JavaScript-Fehler: verlorener Kontext, falsches Container-Element, falsche Ladereihenfolge der Skripte, asynchrones Timing oder ein API-Aufruf des Anbieters, bevor das Widget überhaupt existiert.

2. Zentrale Konzepte

Konzept	So setzen Sie es ein
Kontext	Das richtige Objekt in Callbacks und asynchronem Code erhalten.
Ladereihenfolge	Sicherstellen, dass Skripte, CSS und Anbieterbibliotheken geladen sind, bevor sie verwendet werden.

3. Lektion aus der Praxis

Wer mit Wisej.NET integriert, braucht genug JavaScript-Kenntnisse, um die Beispiele des Anbieters zu lesen und in einen Adapter zu übersetzen. Dazu muss aus einem C#-Entwickler kein Spezialist für Frontend-Frameworks werden. Nötig ist aber ein sorgfältiger Umgang mit Kontext, Callbacks, DOM-Elementen und asynchroner Ausführung. Der häufigste Fehler ist die Annahme, dass `this` in einem Callback des Anbieters noch auf das Wisej.NET-Widget verweist. Drittanbieter-Bibliotheken bewahren Ihren Objektkontext nur selten. Speichern Sie die Referenz in einer Closure, verwenden Sie `bind`, oder leiten Sie den Callback über eine Methode mit dem richtigen Kontext. Wisej.NET-Widgets erhalten den Kontext in ihren eigenen Listnern, doch Anbieterbibliotheken liegen außerhalb dieses Vertrags. Der zweithäufigste Fehler ist das falsche Element. Manche Widgets wollen ein generisches `div`. Andere wollen ein `input`, eine `textarea`, ein `canvas`, eine SVG-Wurzel oder ein bestimmtes, per ID identifiziertes Element. Das Wisej.NET-Element `this.container` ist der sichere Ort, um das Objekt des Anbieters zu hosten.

Das richtige this behalten

In den Funktionen eines InitScripts ist `this` das Wisej.NET-Client-Widget, daher stehen `this.container` und `this.fireWidgetEvent` zur Verfügung. In einem Callback des Anbieters ist `this` das, was der Anbieter festgelegt hat – meist seine eigene Instanz oder das DOM-Element, das das Event ausgelöst hat. Code, der so geschrieben ist, als wäre das Widget noch im Gültigkeitsbereich, kompiliert und läuft – und scheitert beim ersten Callback mit einem Fehler wegen einer undefinierten Methode.

```
init: function (options) {  
    var me = this;  
    me.gauge = new VendorGauge(me.container, options);  
    me.gauge.on('change', function (v) {  
        me.fireWidgetEvent('valueChanged', { value: v });  
    });  
}
```

Das Widget vor dem Callback in einer lokalen Variablen festzuhalten, ist die einfachste Lösung und im Review am leichtesten zu lesen. Ein Aufruf von `bind(this)` auf dem Callback erzielt dasselbe Ergebnis, wenn die Funktion an anderer Stelle deklariert ist. In beiden Fällen wird die Instanz des Anbieters auf dem Widget gespeichert, hier als `me.gauge`, damit `update` und das Freigeben sie später finden. Für asynchronen Code gilt dieselbe Regel: Ein Timer, ein Promise oder ein fetch-Callback läuft erst, nachdem der ursprüngliche Aufruf zurückgekehrt ist, und nur eine festgehaltene Referenz überdauert bis zu diesem Moment.

Das richtige Element – und nur dieses

Jede Anbieterbibliothek dokumentiert, was sie übergeben bekommen möchte. Manche akzeptieren jedes Blockelement und bauen ihre eigenen Kindelemente auf. Andere verlangen ein `input`, um es zu erweitern, ein `canvas`, um darauf zu zeichnen, eine `svg`-Wurzel oder ein Element mit einer bestimmten ID, auf das ihr CSS abzielt. Bekommt eine Diagrammbibliothek ein `div`, obwohl sie ein `canvas` erwartet, entsteht ein leerer Kasten ohne jede Fehlermeldung. Deshalb beginnt das Lab mit einem Nachweis in reinem HTML, bevor auch nur eine Zeile Wisej.NET-Code existiert.

Der sichere Host ist immer `this.container`, das Element, das das Wisej.NET-Widget für Inhalte reserviert. Braucht der Anbieter ein bestimmtes Kindelement, erzeugen Sie es innerhalb des Containers – etwa, indem Sie das innere HTML des Containers auf ein leeres `canvas` oder `input` setzen und dieses Kindelement dann an den Anbieter übergeben. Was niemals passieren darf: das eigene Wurzelement des Widgets verändern, seine Klassen ändern oder es ersetzen, denn das Framework verlässt sich bei Layout, Sichtbarkeit und Freigabe auf dieses Element.

Ein schneller Review-Test: Durchsuchen Sie das InitScript nach allem, was dem Anbieter übergeben wird und nicht `this.container` ist. Jede andere Elementreferenz braucht eine schriftliche Begründung.

Abhängigkeitsreihenfolge für Skripte und Stile

Browserskripte werden in der Reihenfolge ausgeführt, in der sie geladen werden, und ein Plugin, das jQuery erweitert, wirft einen Fehler, wenn jQuery beim Ausführen des Plugins noch nicht global verfügbar ist. Dasselbe gilt für eine Diagrammbibliothek, die ihren Kern vor den Erweiterungsmodulen erwartet, und für ein CSS-Theme, das nach dem Basis-Stylesheet der Bibliothek geladen werden muss, damit seine

Regeln greifen. In einer eigenständigen HTML-Seite steuert der Entwickler diese Reihenfolge über die Abfolge der script-Tags. In Wisej.NET übernimmt die Packages-Liste des Widgets diese Rolle, und die Einträge werden in der aufgeführten Reihenfolge geladen.

Die Reihenfolge im Lab ist die übliche: zuerst jQuery, dann das Stylesheet des Anbieters, dann sein Skript und erst danach das InitScript, das sie verwendet. Eine falsche Reihenfolge scheitert auf verwirrende Weise. Der Fehler erscheint beim Laden in der Konsole, oft bevor überhaupt Widget-Code läuft, und nennt ein undefiniertes Symbol aus der Anbieterbibliothek statt etwas, das der Entwickler selbst geschrieben hat.

- Listen Sie jede Datei auf, die das Beispiel des Anbieters einbindet, in der Reihenfolge, in der es sie einbindet.
- Prüfen Sie, ob jQuery oder eine andere gemeinsam genutzte Bibliothek bereits von der Anwendung geladen wird, um keine zwei Kopien zu laden.
- Fixieren Sie die Versionen in den Paketpfaden, damit ein Upgrade eine bewusste Änderung ist.

Eingefügte Skripte debuggen

Ein InitScript wird zur Laufzeit in die Seite eingefügt; es erscheint daher nicht als Datei im Quellcode-Panel des Browsers, und Sie können keinen Breakpoint über seinen Namen setzen. Ein `sourceURL`-Kommentar am Ende des Skripts gibt ihm einen Namen, und die Entwicklertools führen es dann wie eine echte Datei auf. Eine `debugger`-Anweisung in `init` hält die Ausführung an, sobald die Tools geöffnet sind – der schnellste Weg, um `this`, das Optionsobjekt und den Container genau in dem Moment zu untersuchen, in dem das Objekt des Anbieters erzeugt wird.

Das zweite Werkzeug ist die Konsole. Wird die Instanz des Anbieters nach dem Erzeugen protokolliert, bestätigt das, dass sie existiert, und zeigt, welche Methoden sie bereitstellt. Ein Blick auf das Widget im Elements-Panel bestätigt, dass die Kindelemente des Anbieters innerhalb des Containers entstanden sind und nicht anderswo. Das Network-Panel bestätigt, dass jedes Package mit Status 200 und in der erwarteten Reihenfolge geladen wurde.

Die Debugging-Notizen aus dem Lab halten fest, wo die Instanz des Anbieters gespeichert ist und wie das überprüft wurde. Diese Notizen werden zur ersten Seite des Troubleshooting-Leitfadens, wenn sich dasselbe Widget im Produktivbetrieb danebenbenimmt.



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:

- ✓ Closures und bind einsetzen, um den Kontext in asynchronen Callbacks zu erhalten.
- ✓ Das richtige DOM-Element für ein Widget bestimmen und falsche Änderungen am Wurzelement des Wisej.NET-Widgets vermeiden.
- ✓ Die Abhängigkeitsreihenfolge von Skripten und CSS verstehen, besonders bei jQuery-basierten Bibliotheken.
- ✓ Die Entwicklertools des Browsers, sourceURL, die Konsole und debugger-Anweisungen wirkungsvoll nutzen.

Lab- und Prüfungsleitfaden

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



Praktisches Lab

Nehmen Sie ein reines JavaScript-Beispiel für ein Gauge- oder Knob-Widget und bringen Sie es zunächst in einer isolierten HTML-Seite zum Laufen. Schreiben Sie es anschließend als InitScript eines Wisej.NET-Widgets neu.

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

- ✓ Funktionierender Nachweis in reinem JavaScript
- ✓ Kontextsicheres Wisej.NET-InitScript
- ✓ Debugging-Notizen, die zeigen, wo die Widget-Instanz gespeichert wird

Selbstcheck vor der Abgabe

- ✓ Wovor schützt ``var me = this;``?
- ✓ Wann sollten Sie ``this.container.innerHTML`` verwenden?
- ✓ Warum ist ein Nachweis in reinem JS vor der Wisej.NET-Integration nützlich?