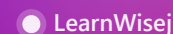


Intégration JavaScript avec Widget

Contrôles natifs ou Widget · C# propre et sécurité des widgets



Contrôles natifs ou Widget

Quand utiliser les contrôles natifs de Wisej.NET ou un Widget JavaScript personnalisé, le fichier JavaScript qui crée le widget (un exemple init/update complet) et comment le relier depuis C# via Packages, InitScript et Options.



Objectif de la leçon

Déterminer quand un écran doit utiliser les contrôles natifs de Wisej.NET et quand un **Widget** JavaScript personnalisé est plus adapté — puis écrire le fichier JavaScript qui est le widget (ses fonctions `init` et `update`) et relier ce fichier à C# via Packages, InitScript et Options.

1. Utiliser d'abord les contrôles natifs de Wisej.NET

La plupart des écrans métier doivent commencer avec les contrôles intégrés de Wisej.NET — libellés, zones de texte, boutons, listes, grilles, panneaux, onglets, boîtes de dialogue et menus. Ils sont plus faciles à maintenir et à déboguer, et s'intègrent déjà au modèle événementiel de Wisej.NET. Un **Widget** est réservé aux cas où vous avez besoin d'une fonctionnalité visuelle côté navigateur que les contrôles ordinaires ne fournissent pas proprement.

Besoin	Choix recommandé	Raison
Formulaire de ticket, écran de paramètres, boutons ordinaires	Contrôles natifs de Wisej.NET	Rapides à construire et faciles à comprendre pour des développeurs juniors.
Liste de tickets ou tableau métier	DataGridView ou contrôles natifs de liste/grille	Fonctionne directement avec la liaison de données C# et les événements de sélection.
Graphique, jauge, carte, frise chronologique ou visualisation spécifique	Widget	Les bibliothèques JavaScript sont souvent meilleures pour les visuels spécialisés.
Effet visuel ponctuel sans valeur métier	À éviter	N'ajoutez pas de risque de maintenance uniquement pour rendre l'interface tape-à-l'œil.

2. La place d'un Widget dans une application Wisej.NET

Gardez la logique métier ordinaire en C#. Ne placez dans l'enveloppe du Widget *que* l'intégration JavaScript visuelle. La passerelle entre les deux doit être étroite et délibérée.

Page Wisej.NET	→ contrôles C# et mise en page
Enveloppe du Widget	→ charge JS/CSS et expose les options
Interface JavaScript	→ graphique / jauge d'état / frise chronologique
Événements serveur	→ rafraîchissement, état et mises à jour des données

La page Wisej.NET et les services restent propriétaires des données métier ; le Widget possède un petit élément visuel côté navigateur. N'envoyez que les données dont le widget a besoin, n'écoutez que les événements qui comptent et gardez le reste de l'application en C# ordinaire.

3. Charger JavaScript et CSS

Une intégration de type Widget a généralement besoin de fichiers de script, de fichiers de style et de données de configuration. Le script peut être un fichier local ou une bibliothèque tierce de confiance ; le CSS contrôle l'apparence ; la configuration indique au widget ce qu'il doit dessiner.

Élément	Ce qu'il fait	Règle pour débutants
Fichier JavaScript	Crée ou met à jour l'élément visuel personnalisé.	Gardez-le petit, bien nommé et centré sur le widget.
Fichier CSS	Met en forme l'élément du widget.	Évitez les styles qui modifient par accident toute l'application.
Configuration / options	Transmet des valeurs de C# au widget.	Envoyez des données simples : nombres, libellés et chaînes d'état.
Événements / callbacks	Permettent au widget de signaler au serveur que quelque chose s'est produit.	Ne branchez que les événements qui servent un vrai workflow.

Côté C#, ces éléments correspondent à trois membres du contrôle **Widget** : **Packages** (les fichiers de script et de feuille de style à charger, dans l'ordre), **InitScript** (le JavaScript qui crée le visuel) et **Options** (l'objet de données sérialisé vers le navigateur). La suite de cette leçon construit chacun d'eux pour une petite *jauge d'état* — le même widget que celui que vous construirez dans le lab.

4. Écrire le fichier JavaScript — le widget lui-même

C'est dans le fichier JavaScript que le visuel est réellement créé. Wisej.NET exécute votre **InitScript** une fois dans le navigateur, avec **this** désignant l'objet widget côté client, et attend que vous y définissiez **deux fonctions** :

Fonction	Quand Wisej.NET l'appelle	Ce que vous y faites
<code>this.init(options)</code>	Une fois, après le chargement des Packages et la création de l'élément DOM du widget.	Créer le visuel dans <code>this.container</code> et dessiner les premières valeurs à partir de <code>options</code> .
<code>this.update(options, old)</code>	Chaque fois que C# modifie <code>Options</code> et appelle <code>Update()</code> .	Redessiner uniquement ce qui a changé. Sans elle, il faudrait recréer le widget à chaque changement d'options.

Trois noms à retenir dans ce fichier : `this.container` est l'élément que Wisej.NET réserve à votre contenu (créez votre balisage à l'intérieur, ne remplacez jamais la racine du widget) ; `options` est l'objet JavaScript simple que C# a affecté à `Options`, avec des noms de propriétés en camelCase (`Percent` devient `percent`) ; et `this.fireWidgetEvent(name, data)` renvoie un petit événement au serveur, qui arrive sous la forme de l'événement C# `WidgetEvent`.

Widgets/statusGauge.js — un fichier d'exemple complet (aucune bibliothèque tierce nécessaire)

```
// Wisej.NET Widget InitScript for a simple status gauge.
// Runs in the browser. "this" is the client-side widget object.
// Options from C# arrive as: { percent: 0-100, label: "text", status: "ok|warn|error" }

// Called ONCE, after Packages have loaded and the widget element exists.
this.init = function (options) {
    // Keep a reference to the widget: inside DOM callbacks "this" changes.
    var me = this;

    // Build the visual inside the container Wisej.NET reserves for us.
    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>';

    // Cache the elements we will update later.
    me.fillEl = me.container.querySelector('.lw-gauge-fill');
    me.labelEl = me.container.querySelector('.lw-gauge-label');

    // Wire ONE event that supports a real workflow: click → tell the server.
    me.container.addEventListener('click', function () {
        // Small payload only. Arrives in C# as WidgetEvent with e.Type == "gaugeClick".
        me.fireWidgetEvent('gaugeClick', { percent: me.lastPercent || 0 });
    });

    // First draw with the values C# sent.
    me.render(options);
};

// Called EVERY TIME C# changes Options and calls widStatus.Update().
this.update = function (options, old) {
    // Redraw only if something we show actually changed.
    if (!old || options.percent !== old.percent ||
```

```

        options.label === old.label || options.status === old.status) {
            this.render(options);
        }
    };

    // Optional helper: a named function the server can call with widStatus.Call("pulse").
    this.pulse = function () {
        var el = this.container.querySelector('.lw-gauge');
        if (!el) return;
        el.classList.remove('lw-gauge-pulse');
        void el.offsetWidth; // restart the CSS animation
        el.classList.add('lw-gauge-pulse');
    };

    // Our own drawing routine: pure display logic, no business rules.
    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 + '%';

        // Status only picks a colour class; C# decided WHICH status applies.
        this.fillEl.className = 'lw-gauge-fill lw-gauge-' + (options.status || 'ok');
    };

```

Widgets/statusGauge.css — limité aux noms de classes propres au widget

```

.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; } }

```



Pourquoi `var me = this` ?

Dans `init`, `this` est le widget Wisej.NET. Dans un callback du navigateur comme l'écouteur de clic, `this` est l'élément qui a déclenché l'événement, si bien que `this.fireWidgetEvent` serait indéfini. Capturez d'abord le widget dans `me` et utilisez `me` dans chaque callback. Remarquez aussi que le fichier ne décide jamais si une valeur est « warn » ou « error » — C# décide de l'état et le JavaScript se contente de lui associer une couleur.

Préfixez vos noms de classes CSS (ici `lw-gauge-...`) pour que la feuille de style ne puisse pas modifier par accident le style du reste de l'application. Gardez le fichier exempt de règles métier, de secrets et d'URL de serveur — n'importe quel développeur doit pouvoir le lire comme « dessine ces nombres ».

5. Relier le fichier à C#

Ajoutez un contrôle **Widget** à la page (nommez-le **widStatus**), puis branchez les deux fichiers et les premières options dans le constructeur de la page ou dans le gestionnaire **Load** . Le CSS va dans **Packages** ; le fichier JavaScript devient l'**InitScript** ; les données vont dans **Options**.

```
// C# - StatusPage.cs (both files live under the web project's /Widgets folder)
private void StatusPage_Load(object sender, EventArgs e)
{
    // 1. Stylesheet (and any third-party scripts) load in list order, once per session.
    widStatus.Packages.Add(new Package {
        Name     = "statusGauge-css",
        Source   = "Widgets/statusGauge.css"
    });

    // 2. The JavaScript file IS the widget: load it as the InitScript.
    //     (You can also paste the file into the InitScript property in the designer.)
    widStatus.InitScript = File.ReadAllText(Application.MapPath("Widgets/statusGauge.js"));

    // 3. First options - safe display values only, prepared by a C# service.
    var data = statusService.GetStatus();
    widStatus.Options = new { percent = data.Percent, label = data.Label, status =
data.Status };

    // 4. Listen only for the event we wired in the JS file.
    widStatus.WidgetEvent += widStatus_WidgetEvent;
}

private void btnRefresh_Click(object sender, EventArgs e)
{
    var data = statusService.GetStatus();           // C# owns the data and the rules
    widStatus.Options = new { percent = data.Percent, label = data.Label, status =
data.Status };
    widStatus.Update();                             // runs this.update(options, old) in the
browser
    widStatus.Call("pulse");                         // runs this.pulse() in the browser
    AddLog("Widget refreshed with server data");
}

private void widStatus_WidgetEvent(object sender, WidgetEventArgs e)
{
    if (e.Type == "gaugeClick")
    {
        // e.Data is the small object the JS sent: { percent: 42 }
        int percent = e.Data.percent;
        AddLog($"Gauge clicked at {percent}%");
    }
}
```

Observez comment les deux côtés se correspondent : **Options = new { percent, label, status }** en C# est exactement l'objet **options** que lit le JavaScript ; **Update()** déclenche **this.update** ; **Call("pulse")** exécute **this.pulse** ; et **fireWidgetEvent('gaugeClick', ...)** arrive sous la forme de

`WidgetEvent` avec `e.Type == "gaugeClick"`. Définissez les options *avant* d'appeler `Update()` ou `Call()` — Wisej.NET les envoie au navigateur en un seul lot, appliqué dans l'ordre.

6. Ce qui se passe à l'exécution

1. La page s'affiche et le navigateur crée l'élément du widget.
2. Wisej.NET charge chaque entrée de **Packages**, dans l'ordre (ici le CSS).
3. Il exécute l'**InitScript**, qui définit `init`, `update`, `pulse` et `render` sur le widget.
4. Il appelle `init(options)` avec les **Options** sérialisées — la jauge apparaît.
5. Chaque fois que C# définit `Options` et appelle `Update()`, le navigateur exécute `update(options, old)`.
6. Un clic dans le navigateur déclenche `gaugeClick`; C# le traite dans `WidgetEvent` comme n'importe quel clic de bouton.



Si le widget reste vide

Ouvrez les outils de développement du navigateur. Une erreur au chargement mentionnant un symbole non défini signale un problème d'ordre des Packages ; « fireWidgetEvent is not a function » dans un callback signale un `this` perdu (utilisez `me`) ; un conteneur vide sans erreur signifie généralement que `init` n'a jamais reçu l'option dont il avait besoin — journalisez `options` au début de `init` pour voir ce que C# a réellement envoyé.



Vous êtes prêt pour la leçon suivante

Vous savez choisir entre contrôles natifs et Widget, écrire le fichier JavaScript du widget avec `init` et `update`, et le relier depuis C# via Packages, InitScript, Options, `Update()`, `Call()` et `WidgetEvent`. Ensuite : garder le reste de l'application en C# propre, et les risques de sécurité des scripts tiers.

C# propre et sécurité des widgets

Gardez les règles métier en C# (pas en JavaScript), encapsulez le code d'intégration et identifiez les risques de sécurité des scripts tiers.



Objectif de la leçon

Garder les règles métier en C# (pas en JavaScript), encapsuler le code d'intégration pour qu'il ne s'éparpille pas, et reconnaître les risques de sécurité et de maintenance liés à l'ajout de scripts tiers dans une application métier.

1. Garder le reste de l'application en C# propre

Le Widget ne doit pas devenir une seconde application cachée dans votre application Wisej.NET. Ne mettez pas de règles métier, de calculs sur les tickets, de permissions ou de logique de base de données en JavaScript — gardez-les dans des services C# et ne transmettez au widget que les données d'affichage finales.

Mauvais pattern	Meilleur pattern
Placer les règles métier dans un gros bloc JavaScript.	Calculer les valeurs métier dans un service C#, puis envoyer les valeurs d'affichage au Widget.
Charger des scripts quelconques depuis des sites inconnus.	Utiliser des bibliothèques de confiance, les versionner et documenter pourquoi elles sont nécessaires.
Laisser chaque page construire manuellement ses propres chaînes de script.	Encapsuler le code d'intégration dans un seul helper / contrôle de widget réutilisable.
Exposer des secrets ou des données internes du serveur au JavaScript.	N'envoyer au navigateur que des données d'affichage sans risque.

2. Risques de sécurité et de maintenance

Les scripts tiers apportent de la puissance, mais aussi des risques — ils peuvent modifier le comportement de la page, casser lors d'une mise à jour de la bibliothèque, entrer en conflit avec les styles ou exposer des données si de mauvaises informations sont envoyées au navigateur. Traitez chaque script externe comme une dépendance à examiner et à maintenir.

- Utilisez des bibliothèques de confiance ; évitez de copier-coller des scripts inconnus dans le projet.
- Préférez des fichiers locaux et versionnés quand vous avez besoin d'un comportement prévisible.
- N'envoyez **jamais** de clés d'API, de mots de passe, de chaînes de connexion ou de données internes privées au widget du navigateur.
- Documentez ce que fait le script et quel écran l'utilise.
- Testez l'écran après les changements de thème, de mise en page et les mises à jour du navigateur.



N'exposez jamais de secrets au navigateur

Tout ce qui est envoyé au widget arrive dans le navigateur. Préparez des données d'affichage sans risque en C# et n'envoyez que ce dont le widget a besoin pour dessiner.

3. Le pattern à retenir

Les contrôles natifs sont le choix par défaut. Le Widget est réservé aux visuels particuliers. C# possède les données et les règles ; JavaScript se charge du rendu visuel. La frontière d'intégration doit être réduite, nommée et facile à tester.



Vous êtes prêt pour le lab

Ensuite : regardez la vidéo pas à pas sur le widget, réussissez la vérification des connaissances, puis construisez un petit widget JavaScript d'état ou de graphique — alimenté en données d'affichage sans risque par un service C#, rafraîchi par un événement serveur, avec une carte de données pour vérifier les valeurs.