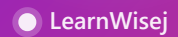


Widgets personnalisés, extensions, thèmes et projet de synthèse

Guide de la leçon · Guide du lab / de l'examen



Guide de la leçon

Contrôles natifs ou widgets, composants, extender providers et packs d'icônes ; les packages du contrôle Widget, InitScript, WidgetEvent et la passerelle CallAsync ; thèmes et documentation ; le projet de synthèse Operations Console. Ce que cela signifie pour un développeur Wisej.NET — et comment aborder ce module.



Objectif en production

Contrôles natifs ou widgets, composants, extender providers et packs d'icônes ; les packages du contrôle Widget, InitScript, WidgetEvent et la passerelle CallAsync ; thèmes et documentation ; le projet de synthèse Operations Console.

1. Le problème en production

Tôt ou tard, un produit a besoin de quelque chose que la bibliothèque native ne fournit pas : une capacité spécialisée du navigateur, une bibliothèque JavaScript tierce, une visualisation sur mesure ou une intégration réutilisable. La tentation est de basculer dans le JavaScript et de perdre le modèle Wisej.NET : l'état serveur, les thèmes, le designer et un flux d'événements qu'un autre développeur peut suivre. Ce module intègre un widget client sans renoncer à rien de tout cela, puis termine le projet de synthèse Operations Console.

2. Concepts clés

Concept	Pourquoi c'est important
contrôle natif	Un contrôle qui implémente à la fois le côté serveur et son widget navigateur, et dispose donc déjà du support au design, des thèmes, des événements et de l'état serveur ; toujours le premier recours.
types d'extension	Control, Widget, Component, Extender Provider et Icon Pack : nommer le type évite de confondre « réutilisable » et « nouveau contrôle natif ».
contrôle Widget	Un conteneur client généraliste qui charge des packages JavaScript et CSS, exécute un script d'initialisation et échange événements et données avec le code serveur.
Packages et InitScript	La liste des packages déclare les fichiers client dont dépend le widget ; l'InitScript crée l'objet JavaScript, branche ses événements client et n'expose que les fonctions dont le serveur a besoin.
WidgetEvent	L'événement serveur déclenché quand le code client appelle <code>fireWidgetEvent</code> ; son type et son payload de données arrivent du navigateur et doivent être validés comme toute autre saisie utilisateur.
CallAsync / Eval	La moitié serveur vers client de la passerelle : appeler une fonction exposée par le widget, ou évaluer un script, sans quitter le C#.
contrat du widget	Les propriétés, événements, format du payload, dépendances de packages et règles de thème documentés qui permettent à un autre développeur, ou à un assistant IA correctement informé, de maintenir le widget.

3. Le fil de la leçon

Le natif d'abord, puis cinq types d'extension

La bibliothèque de contrôles est le premier recours, à chaque fois. Un contrôle natif implémente ensemble le côté serveur et le côté client, c'est pourquoi il participe gratuitement au designer de Visual Studio, à `Application.Theme`, au modèle d'événements et à l'état serveur propre à chaque session. Le code personnalisé est réservé à ce que la bibliothèque ne couvre pas : une capacité du navigateur sans équivalent natif, une bibliothèque JavaScript tierce, une visualisation sur mesure ou une intégration que vous souhaitez réutiliser.

Quand le code personnalisé se justifie, nommez le type d'extension que vous construisez. Un **Control** est un contrôle natif complet avec une classe serveur et un widget client. Un **Widget** est le conteneur client généraliste pour un widget JavaScript tiers ou personnalisé. Un **Component** peut exister uniquement sur le serveur ou aussi sur le client. Un **Extender Provider** ajoute des propriétés et un comportement à des contrôles existants, comme le font `ToolTip` ou `ErrorProvider`. Un **Icon Pack** regroupe des ressources visuelles pour le projet et le designer. Cette taxonomie évite la confusion de conception : tout élément réutilisable n'est pas un `UserControl`, et toute intégration JavaScript n'exige pas un nouveau contrôle natif.

Pour Operations Console, les règles de décision sont courtes. Utilisez d'abord les contrôles natifs. Utilisez un `UserControl` pour une composition réutilisable côté serveur, comme l'éditeur de client du Module 2. Utilisez `Widget` pour une interface JavaScript tierce qui peut vivre dans un conteneur Wisej.NET. Ne construisez une extension complète que si l'intégration doit être réutilisée entre applications et se présenter comme un composant à part entière.

La passerelle Widget dans les deux sens

Le contrôle `Widget` fait trois choses. Sa collection `Packages` charge, dans l'ordre, les fichiers JavaScript et CSS dont la bibliothèque client a besoin. Son `InitScript` s'exécute une fois les packages présents : `this.init` crée l'objet client dans `this.container`, branche les événements propres à la bibliothèque et expose les quelques fonctions que le serveur peut appeler. Enfin, il transporte les messages : le code client appelle `this.fireWidgetEvent("ratingChanged", { value: 4 })` et le serveur reçoit un `WidgetEvent` dont les `WidgetEventArgs` exposent le `Type` et le payload `Data` ; le code serveur répond avec `CallAsync` pour invoquer une fonction client, ou `Eval` pour exécuter un script.

Le widget de notation du lab illustre l'aller-retour : un clic sur une étoile déclenche l'événement, le gestionnaire valide la valeur, l'enregistre via un service et confirme au widget :

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

    // The payload came from the browser: validate before trusting it.
    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.");
}
```

Utilisez la passerelle avec prudence. Le JavaScript reste concentré sur le travail propre au navigateur : rendu, gestes, animation. Les règles métier restent sur le serveur, où elles sont sécurisées, testées et partagées avec chaque écran ; n'en dupliquez une en JavaScript que comme optimisation délibérée côté client, jamais comme unique exemplaire. Un bug classique : dans un callback de la bibliothèque, `this` désigne l'objet de la bibliothèque et non le widget, donc capturez `var me = this` dans `init` et appelez `me.fireWidgetEvent`.

Un contrat que l'on peut déboguer, thématiser et documenter

Un wrapper robuste a un contrat. Les propriétés publiques côté serveur correspondent à une configuration pertinente, pas à chaque option exposée par la bibliothèque. Les événements représentent des actions utilisateur pertinentes pour le métier : `ratingChanged`, pas `mouseup`. Si les payloads dépassent une ou deux valeurs, définissez un DTO ou une forme JSON claire ; le gestionnaire valide et normalise les données avant de les utiliser, car un payload du navigateur peut être modifié par n'importe qui disposant des outils de développement.

Le débogage des widgets commence dans la console du navigateur, et la documentation Wisej.NET le dit explicitement. Quand un widget s'affiche vide ou qu'un événement n'arrive jamais, vérifiez d'abord la console, puis l'ordre de chargement des packages, les chemins des ressources et le moment de l'initialisation, et si le widget est recréé après une actualisation ou un changement de mise en page, ce qui signifie que `init` doit pouvoir être réexécutée sans risque. Les scripts placés dans le HTML de démarrage peuvent s'exécuter trop tôt pour les widgets Wisej.NET ; les ressources incorporées, `Packages`, `InitScript` et les appels au niveau du contrôle sont des modèles plus sûrs.

Les thèmes et le comportement responsive constituent la touche finale. Si le widget doit s'accorder à la console, décidez délibérément : exposer des propriétés de style, s'appuyer sur les apparences du thème, utiliser les extenders StyleSheet ou Animation, ou livrer du CSS dans le package du widget. Ne codez pas en dur les couleurs de l'application dans le JavaScript, sauf si le widget est volontairement indépendant. Appliquez des propriétés responsive par profil client comme vous l'avez fait pour le shell, fournissez des libellés accessibles et un comportement clavier quand c'est possible, et rédigez la note de contexte pour l'IA : à quoi sert le widget, pourquoi aucun contrôle natif ne convenait et où se trouve le contrat.

- Les dépendances de packages et l'ordre de chargement sont documentés.
- Les propriétés, les événements et le format du payload sont consignés à côté du code.
- Les règles métier sont appliquées côté serveur et la console du navigateur est propre.



Pièges courants

Remplacer des contrôles standard par du JavaScript parce que cela semble moderne. Placer des règles métier dans le JavaScript client, où elles sont plus difficiles à sécuriser et à tester. Ne pas gérer la réinitialisation du widget après une actualisation ou un changement de mise en page. Oublier de documenter les dépendances de packages et le contrat d'événements. Construire un widget personnalisé que personne d'autre dans l'équipe ne peut maintenir.

Terminer Operations Console

Le projet de synthèse réunit le cours dans une seule application, `OperationsConsole` : le shell navigable (Modules 1 et 3), l'éditeur de client validé (Module 2), l'explorateur TreeView et ListView (Module 4), le `DataGridView` Orders (Module 5), l'écran tableau de bord et contenu (Module 6), et l'écran de widget personnalisé ou d'extension de ce module. Chaque écran est évalué au regard des questions posées dans les labs : quel contrôle fait le travail le plus important, où réside l'état, que se passe-t-il avec deux sessions ou dix fois plus de données, et quel code relève d'un `UserControl` ou d'un service réutilisable.

La grille d'évaluation attribue 20 % chacun à la sélection des contrôles et aux contrôles de données, et 15 % chacun à la mise en page et la réutilisation, à la validation et au retour utilisateur, au contrat du widget personnalisé, et à la finition et la documentation. Préparez une démo de cinq minutes qui suit le script du projet de synthèse, et rédigez `CapstoneNotes.md` à partir du modèle du cahier d'exercices : contrôles utilisés et pourquoi, hypothèses de volume de données, choix de validation et de mise en page, code réutilisable extrait, et une chose que vous amélioreriez en production. Le code doit être compréhensible par un autre développeur et explicable par un assistant IA s'appuyant sur vos notes.



Réflexe bibliothèque de contrôles

Choisissez le contrôle natif le plus simple qui exprime la tâche de l'utilisateur avant d'écrire du code personnalisé. Les contrôles natifs offrent déjà le support au design, les thèmes, les événements et l'état serveur ; tournez-vous ensuite vers un extender ou un UserControl, et ne créez un widget JavaScript que si aucun contrôle natif ne convient.



Objectifs d'apprentissage

À la fin de ce module, vous saurez :

- ✓ Expliquer la différence entre contrôles natifs, widgets, composants, extender providers et packs d'icônes.
- ✓ Utiliser le contrôle Widget pour charger des packages JavaScript/CSS, initialiser un widget client et échanger événements et données avec le code serveur.
- ✓ Appliquer thèmes, propriétés responsive, notes d'accessibilité et notes de contexte pour l'IA à un package de contrôle personnalisé.
- ✓ Terminer et présenter le projet de synthèse Operations Console en utilisant des contrôles natifs et au moins un widget personnalisé ou une extension.

Guide du lab / de l'examen

Ce que vous allez construire dans le lab pratique, l'approche suggérée et les livrables attendus.



Lab pratique

Encapsulez un petit widget JavaScript de notation ou d'indicateur avec le contrôle Widget, ou configurez un package d'extension Wisej.NET : chargez ses packages JavaScript et CSS, déclenchez un événement côté serveur depuis le client avec le modèle d'événement de widget, appelez une fonction côté client depuis le serveur avec CallAsync ou Eval, appliquez-lui un thème cohérent avec la console, puis peaufinez le projet de synthèse Operations Console complet et préparez une démo de 5 minutes.

Point de départ

Ouvrez la solution `OperationsConsole` que vous faites évoluer depuis les Modules 1 à 6 : le shell, l'éditeur de client, l'explorateur de documents, la grille Orders et le tableau de bord existent tous, et l'onglet Widgets est encore la page provisoire du Module 1. Gardez le travail de ce module dans son propre dossier ou namespace pour que le projet de synthèse puisse le réutiliser.

Approche suggérée

- ✓ Décidez, et notez par écrit, pourquoi aucun contrôle natif ne convient à la notation ou à l'indicateur que vous ajoutez ; si un contrôle natif convient, utilisez-le et choisissez une autre cible de widget.
- ✓ Déposez un contrôle `Widget` sur la page Widgets, ajoutez les fichiers JavaScript et CSS à sa collection `Packages` dans l'ordre de chargement, et écrivez un `InitScript` dont la fonction `init` crée l'objet client dans `this.container` et peut être réexécutée sans risque après une actualisation.
- ✓ Dans le callback client, capturez `var me = this` et appelez `me.fireWidgetEvent("ratingChanged", { value })` ; côté serveur, gérez `WidgetEvent`, vérifiez `e.Type`, validez et normalisez `e.Data`, puis enregistrez via un service.
- ✓ Confirmez au navigateur avec `CallAsync("setSaved", value)` ou `Eval`, informez l'utilisateur par un Toast ou une MessageBox, et appliquez le thème du widget via les apparences du thème, du CSS packagé ou l'extender StyleSheet plutôt que des couleurs codées en dur dans le JavaScript.
- ✓ Rédigez le contrat à côté du code (packages, propriétés, événements, format du payload, notes de débogage), puis passez en revue chaque écran du projet de synthèse au regard de la grille d'évaluation et répétez le script de démo de cinq minutes.

Livrables attendus

- ✓ Écran basé sur un Widget ou une extension, avec ses packages JavaScript et CSS chargés
- ✓ Événement client vers serveur déclenché avec le modèle d'événement de widget ou un événement natif de l'extension
- ✓ Appel serveur vers client via CallAsync, Eval ou l'API de l'extension
- ✓ Contrat client/serveur documenté : packages, propriétés, événements et format du payload

- ✓ Projet de synthèse Operations Console peaufiné avec un script de démo de 5 minutes

Questions de révision

- ✓ Que se passerait-il si un utilisateur modifiait le payload `ratingChanged` dans les outils de développement du navigateur avant qu'il n'atteigne votre gestionnaire, et quelle ligne de votre code l'arrête ?
- ✓ Votre widget s'affiche vide après que l'onglet Widgets a été masqué puis réaffiché : quels trois points vérifiez-vous, dans quel ordre, et pourquoi le HTML de démarrage n'est-il pas la solution ?
- ✓ Quelle règle de votre widget pourriez-vous déplacer en JavaScript à titre d'optimisation, et laquelle ne doit jamais quitter le serveur ?

Remise du projet de synthèse

Le lab de ce module constitue aussi la remise du projet de synthèse. Operations Console doit comporter six écrans : le shell de l'application avec ses zones de navigation, de commandes et d'état ; l'éditeur de client ou de contact validé ; l'explorateur TreeView/ListView avec des ID stables et une stratégie de chargement différé ou virtuelle ; le DataGridView Orders avec des colonnes explicites, une mise en forme et une stratégie pour les gros volumes ; l'écran tableau de bord et contenu avec ChartJS et un aperçu de document ; et l'écran de Widget personnalisé ou d'extension de ce lab. Remettez le projet source, des captures d'écran de chaque écran, `CapstoneNotes.md` rédigé à partir du modèle du cahier d'exercices, et une démo de cinq minutes (en direct ou enregistrée) qui suit ce script :

- ✓ Expliquer l'architecture de l'application du point de vue de la bibliothèque de contrôles, puis montrer l'éditeur et son comportement de validation.
- ✓ Montrer la navigation dans l'arbre et la liste avec le chargement du détail, puis la grille et sa gestion des gros volumes.
- ✓ Montrer le tableau de bord et les contrôles de contenu, puis le widget personnalisé ou l'extension, y compris sa communication d'événements dans les deux sens.
- ✓ Expliquer ce que votre assistant IA devrait répondre si on lui demandait pourquoi chaque contrôle a été choisi.

Domaine évalué	Poids	À quoi ressemble l'excellence
Sélection des contrôles	20 %	Contrôles natifs, extenders, UserControls, widgets et extensions utilisés à bon escient, avec une justification claire.
Mise en page et réutilisation	15 %	Conteneurs, docking, ancrage, séparateurs, onglets et UserControls utilisés proprement.
Contrôles de données	20 %	Modèles TreeView, ListView et DataGridView avec une identité, une liaison et une stratégie pour les gros volumes correctes.
Validation et retour utilisateur	15 %	Propriétés des éditeurs, ErrorProvider et notifications utilisés de façon claire et cohérente.
Widget personnalisé ou extension	15 %	Un contrat client/serveur documenté, des événements gérés, aucune logique métier mal placée.
Finition et documentation	15 %	Thème, comportement responsive, commentaires et notes de contexte pour l'IA pensés pour la production.



Réflexe bibliothèque de contrôles

Choisissez le contrôle natif le plus simple qui exprime la tâche de l'utilisateur avant d'écrire du code personnalisé. Les contrôles natifs offrent déjà le support au design, les thèmes, les événements et l'état serveur ; tournez-vous ensuite vers un extender ou un UserControl, et ne créez un widget JavaScript que si aucun contrôle natif ne convient.