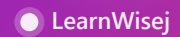


Classes de widgets réutilisables et propriétés adaptées au Designer

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



Guide de la leçon

Passer d'instances ponctuelles de Widget à des classes serveur réutilisables, avec des propriétés typées et une configuration maîtrisée. Ce que cela signifie pour un développeur qui intègre avec Wisej.NET — et comment aborder ce module.



Objectif production

Passer d'instances ponctuelles de Widget à des classes serveur réutilisables, avec des propriétés typées et une configuration maîtrisée.

1. Le problème d'intégration

Le prototype fonctionne, mais le même widget sera utilisé sur de nombreux écrans. Recopier Packages, Options et InitScript dans chaque page crée un risque de maintenance.

2. Concepts clés

Concept	Comment l'utiliser
Propriétés typées	Exposer des propriétés .NET stables plutôt que les options brutes de la bibliothèque tierce.
Configuration des ressources	Garder la configuration des packages et des scripts à l'intérieur de la classe wrapper.

3. Leçon de production

Une classe de widget réutilisable est l'étape naturelle qui suit un prototype réussi. Le développeur d'application ne devrait pas avoir à se souvenir, pour chaque écran, de l'ordre des scripts tiers, du code d'initialisation exact ou des noms bruts des options. La classe wrapper centralise ces décisions. La classe serveur dérive toujours de `Wisej.Web.Widget`, mais sa surface publique devient une API .NET. Par exemple, une jauge peut exposer `Value`, `Minimum`, `Maximum`, `Label` et `AnimationEnabled` au lieu d'obliger chaque développeur à modifier un objet Options dynamique. La classe peut conserver une porte de sortie pour les options avancées, mais le chemin courant doit être typé, découvrable et adapté au Designer. C'est aussi là que le nommage, les valeurs par défaut, les catégories, les descriptions et les convertisseurs de type prennent de l'importance. Le wrapper n'est plus un simple hôte de script : il devient une partie du framework applicatif utilisé par d'autres développeurs.

Une classe qui possède sa configuration

Le lab refactorise le prototype en `IntegrationLab.Controls.SimpleGauge`, une classe qui dérive de `Wisej.Web.Widget`. Son constructeur, ou sa surcharge d'initialisation, affecte les Packages, l'InitScript et les Options par défaut qui étaient auparavant saisis sur chaque page. Un écran qui a besoin d'une jauge déclare désormais un champ `SimpleGauge`, le dépose depuis la boîte à outils et ne voit jamais une ligne de JavaScript.

L'InitScript lui-même passe dans une ressource incorporée ou une constante de la classe, pour être versionné avec le code C# qui en dépend. Les chemins des packages tiers deviennent des constantes regroupées en un seul endroit. Quand la bibliothèque tierce publie une nouvelle version, une seule classe change, et chaque écran en profite au build suivant.

Une classe wrapper est aussi le premier endroit où un registre de décisions devient du code. Le nom de la classe, son espace de noms et son commentaire de synthèse indiquent à quoi sert le widget et quelle bibliothèque tierce il encapsule, c'est-à-dire ce qu'un développeur doit savoir en premier dans deux ans.

Masquer les portes de sortie

Dès que la classe possède la configuration, les membres bruts deviennent dangereux. Un développeur qui modifie Packages sur une instance casse cette instance et aucune autre, ce qui est le type de bug le plus difficile à trouver. Le wrapper masque ou protège donc Packages, InitScript et l'objet Options brut, afin qu'ils n'apparaissent pas dans la fenêtre Propriétés et ne puissent pas être modifiés à la légère depuis le code applicatif.

En pratique, cela signifie redéfinir ces membres en `protected`, ou les marquer pour que le Designer ne les affiche ni ne les sérialise. La surface publique de la classe se limite alors aux propriétés typées et aux événements qu'elle expose délibérément. Les scénarios avancés peuvent toujours passer par une méthode `protected` qui permet à une classe dérivée d'étendre les options, ce qui laisse la porte de sortie ouverte à ceux qui en comprennent les conséquences.

L'auto-vérification du lab demande quels membres doivent être masqués aux utilisateurs ordinaires du wrapper. La réponse : tout membre dont un mauvais usage pourrait casser le widget sans bruit.

Des propriétés typées qui pilotent Options

Une propriété typée est un petit morceau de C# qui lit et écrit dans l'objet Options tout en présentant un type .NET normal. `Value` est un double avec contrôle de plage, `Minimum` et `Maximum` sont des doubles validés l'un par rapport à l'autre, et `Caption` est une chaîne. Chaque setter affecte l'option en camelCase, et Wisej.NET rend la modification et appelle `update` côté client exactement comme pour le prototype.

Les attributs rendent les propriétés adaptées au Designer. Un attribut de valeur par défaut indique au Designer ce qu'il ne doit pas sérialiser, une catégorie regroupe les propriétés dans la fenêtre Propriétés, et une description explique chacune dans le volet d'aide. Une propriété coûteuse à rendre, par exemple une qui force la bibliothèque tierce à se reconstruire, doit le signaler dans sa description et n'affecter l'option que lorsque la valeur a réellement changé.

- Quels changements de propriété doivent forcer une mise à jour côté client, et lesquels sont en lecture seule après la création.
- Quelles valeurs sont validées dans le setter, et quelle exception lève une valeur incorrecte.

- Quelles valeurs par défaut correspondent à celles de la bibliothèque tierce, pour qu'une nouvelle instance s'affiche correctement.

Ce qui relève du wrapper, ce qui relève de l'application

Le wrapper est responsable de la bibliothèque tierce : charger ses fichiers, créer et libérer son instance, traduire les options et relayer les événements. L'application est responsable du sens : quel relevé affiche la jauge, quel seuil compte et ce qui se passe quand il est franchi. Un wrapper qui contient un appel à la base de données ou une règle métier a franchi cette limite, tout comme une page qui contient du code d'initialisation tiers.

La page de démonstration du lab en est la preuve. Elle place un `SimpleGauge`, lie `Value` à un relevé fourni par un service et gère un événement .NET. Il n'y a aucun `InitScript` sur la page et aucun nom de bibliothèque tierce dans son code. Si la bibliothèque tierce était remplacée demain, la page ne changerait pas.

Un bon test de frontière consiste à imaginer livrer le wrapper sous forme de package NuGet à une autre équipe. Tout ce qu'elle aurait besoin de modifier à l'intérieur du package appartient en réalité à l'application.



Réflexe de production

Gardez clairement séparées la propriété de l'état côté Wisej.NET et le comportement du widget client. Documentez le contrat client/serveur avant de vous appuyer sur l'intégration en production.



Erreur courante à éviter

Copier du JavaScript de démonstration dans les écrans sans documenter le cycle de vie, les ressources, les contrats d'événements, le nettoyage ni les frontières de sécurité.



Objectifs d'apprentissage

À la fin de ce module, vous serez capable de :

- ✓ Créer une classe qui dérive de `Wisej.Web.Widget`.
- ✓ Masquer ou protéger `Packages`, `InitScript` et les Options brutes lorsque c'est pertinent.
- ✓ Exposer des propriétés .NET typées qui mettent à jour l'objet `Options` en toute sécurité.
- ✓ Séparer la configuration des ressources tierces de l'usage applicatif.

Guide du lab / de l'examen

Ce que vous allez construire dans le lab pratique, la démarche recommandée, les livrables attendus et l'auto-vérification avant la remise.



Lab pratique

Refactorisez le prototype de jauge en `IntegrationLab.Controls.SimpleGauge``, avec des propriétés typées et un enregistrement des ressources par défaut.

Démarche recommandée

- ✓ Ouvrez le projet existant du cours d'intégration Wisej.NET et exécutez-le en local.
- ✓ Repérez le composant serveur, le widget client, le fichier de ressources ou le point de terminaison concernés par ce module.
- ✓ Faites une petite modification à la fois et testez-la dans le navigateur avant de passer à la suite.
- ✓ Gardez le JavaScript propre à la bibliothèque tierce dans l'adaptateur ou le wrapper au lieu de le disperser dans les pages.
- ✓ Mettez à jour l'interface pour que l'apprenant voie clairement le succès, l'échec, la progression ou les changements de communication.
- ✓ Prenez des captures d'écran ou des notes qui prouvent que l'intégration fonctionne et qui expliquent le choix de conception.

Livrables attendus

- ✓ Classe réutilisable dérivée de Widget
- ✓ Propriétés typées avec valeurs par défaut
- ✓ Page de démonstration utilisant la classe sans InitScript personnalisé sur la page

Auto-vérification avant la remise

- ✓ Quels membres faut-il masquer aux utilisateurs ordinaires du wrapper ?
- ✓ Quels changements de propriété doivent forcer une mise à jour côté client ?
- ✓ Qu'est-ce qui relève du wrapper et qu'est-ce qui relève de l'application ?