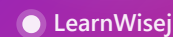


UX de validation en production, error providers personnalisés et projet de synthèse

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



Guide de la leçon

Assembler toutes les couches : un pipeline de Save volontairement banal et reproductible, un `ILoggerProvider` personnalisé qui alimente un panneau récapitulatif, des providers d'icônes et de récapitulatif côte à côte, dix règles d'UX de validation, et le projet de synthèse `ValidationClinic` qui prouve où chaque vérification a sa place. Ce que cela signifie pour un développeur qui connaît les bases de C# et débute dans le développement de formulaires Wisej.NET — et comment aborder ce module.



Objectif production

Assembler toutes les couches : un pipeline de Save volontairement banal et reproductible, un `ILoggerProvider` personnalisé qui alimente un panneau récapitulatif, des providers d'icônes et de récapitulatif côte à côte, dix règles d'UX de validation, et le projet de synthèse `ValidationClinic` qui prouve où chaque vérification a sa place.

1. Le problème en production

Un utilisateur qui remplit le formulaire d'inscription de `ValidationClinic` ne se demande jamais quelle API a trouvé une erreur ; il remarque seulement si l'écran lui dit ce qui ne va pas et le laisse terminer. Après six modules, l'application dispose de gestionnaires de champ, de règles d'extender, d'un service de validation, de modèles annotés et de règles de grille, mais chacun a été ajouté séparément, et rien ne garantit encore qu'ils s'exécutent tous, dans le bon ordre, à chaque clic sur Save. Ce module réunit les couches dans un pipeline de Save prévisible et un récapitulatif visible, puis vous demande de prouver la conception dans un projet de synthèse.

2. Concepts clés

Concept	Pourquoi c'est important
Pipeline de Save	Une séquence fixe dans <code>btnSave_Click</code> qui valide les modifications en attente, exécute tour à tour chaque couche de validation et sort à la première couche qui échoue, pour que rien n'atteigne le repository sans vérification.
Sortie anticipée	Chaque étape termine le gestionnaire avec <code>return</code> quand elle trouve des erreurs, pour que les étapes suivantes ne s'exécutent jamais sur des données qu'une étape précédente a déjà rejetées.
<code>IErrorProvider</code>	La petite interface à laquelle le système de validation de Wisej.NET envoie ses messages : <code>SetError(Control, string)</code> pour définir ou effacer un message et <code>GetError(Control)</code> pour le relire.
<code>SummaryErrorProvider</code>	Votre propre <code>IErrorProvider</code> , qui tient un dictionnaire de paires contrôle-message et réécrit à partir de celui-ci un libellé récapitulatif chaque fois qu'une erreur est définie ou effacée.
Double provider	Un unique utilitaire <code>SetError</code> qui écrit le même message dans l' <code>ErrorProvider</code> intégré (l'icône à côté du champ) et dans le provider récapitulatif (la liste en haut).
Erreur utilisateur ou panne système	Une erreur que l'utilisateur peut corriger, comme un nom en double, devient un message de champ ; une exception inattendue est journalisée pour les développeurs et affichée sous forme de message générique et posé.
Test aux limites	Une valeur de test située exactement à la limite d'une règle, comme un âge de 0 ou de 120, qui détecte les erreurs de décalage d'une unité que les valeurs valides et invalides ordinaires laissent passer.

3. Déroulé de la leçon

Un bouton Save volontairement banal

Un bon gestionnaire de Save effectue les mêmes étapes dans le même ordre à chaque clic et s'arrête dès que quelque chose ne va pas. Dans ValidationClinic, l'ordre est le suivant : effacer le récapitulatif, appeler `bindingSource.EndEdit()` pour que la valeur en cours de saisie soit poussée dans le modèle, appeler `ValidateChildren()` pour que les règles de champ de chaque contrôle s'exécutent, lancer la vérification DataAnnotations avec `Validator.TryValidateObject`, exécuter `ContactValidator.Validate` pour les règles métier, vérifier qu'aucune ligne de la grille ne porte encore d'erreur, et seulement ensuite enregistrer. `EndEdit` vient en premier, car sinon chaque vérification du modèle qui suit examinerait des données périmées.

```

ResetSummary();
bindingSource.EndEdit();
if (!this.ValidateChildren()) return;

var model = (ContactEditModel)bindingSource.Current;
if (!ApplyAnnotationErrors(model)) return;

var businessErrors = contactValidator.Validate(model);
if (businessErrors.Count > 0) { ApplyBusinessErrors(businessErrors); return; }

if (GridHasErrors()) return;

try { SaveContact(model); MessageBox.Show("Saved.", MessageBoxIcon.Information); }
catch (DuplicateNameException) { SetError(txtName, "A contact with this name already exists."); }

```

Les noms d'utilitaires utilisés ici (`ResetSummary` , `ApplyAnnotationErrors` , `GridHasErrors`) sont vos propres petites méthodes, pas des membres du framework. L'enregistrement lui-même se trouve dans un `try / catch` , et le `catch (Exception ex)` final (non montré) écrit l'exception avec `System.Diagnostics.Debug.WriteLine` et affiche un texte compréhensible avec `MessageBox.Show` . Le pipeline est banal à dessein : quand un testeur signale « Save a laissé passer une adresse e-mail incorrecte », vous pouvez désigner l'étape exacte qui aurait dû l'arrêter.

Un panneau récapitulatif via IErrorProvider

Le système de validation de Wisej.NET se moque de ce qui affiche ses messages ; il exige seulement que la cible implémente `IErrorProvider` . L' `ErrorProvider` intégré en est une implémentation et un `Label` peut en être une autre. Votre `SummaryErrorProvider` en est une troisième : son constructeur reçoit le libellé récapitulatif, `SetError` retire l'entrée du contrôle quand le message est vide ou l'enregistre sinon, puis regroupe les messages distincts dans le `Text` du libellé et ne définit `Visible` que tant qu'au moins une erreur existe. `GetError` renvoie le message enregistré ou une chaîne vide.

Vous le connectez en deux lignes : créez le provider autour de `validationSummaryLabel` , puis affectez `validation.ErrorProvider = summaryProvider` . Dès cet instant, chaque règle exécutée par l'extender Validation, y compris votre `MinimumAgeValidationRule` du Module 4, remonte dans le panneau sans aucun code supplémentaire dans la règle elle-même.

Erreurs de champ, de grille et de modèle au même endroit

Les erreurs arrivent désormais de trois directions. Les erreurs de champ viennent des gestionnaires `Validating` et des règles de l'extender. Les erreurs de modèle viennent des noms de membres des `ValidationResult` et des messages de `ContactValidator` , que vous associez aux contrôles avec la table de correspondance des champs du Module 5. Les erreurs de grille vivent sur les cellules et les lignes sous forme d' `ErrorText` , défini dans `CellValidating` , `ValidateRow` et `DataError` . L'utilisateur ne devrait pas avoir à savoir laquelle de ces sources a produit un message.

Pour y parvenir, faites passer les erreurs du formulaire et du modèle par un seul utilitaire `SetError(Control, string)` dans le formulaire, qui appelle `errorProvider.SetError` pour l'icône et `summaryProvider.SetError` pour la liste. Les erreurs de grille restent sur la grille, où l'utilisateur voit la

cellule rouge, et l'étape grille du pipeline ajoute une ligne au récapitulatif, par exemple « Row 3: Closed contacts require a closed date. », pour que le problème soit aussi visible en haut. L'effacement fonctionne de la même manière en sens inverse : un message vide retire à la fois l'icône et la ligne du récapitulatif.

Dix règles pour une validation vraiment utilisable

Une bonne UX de validation relève surtout de la discipline, pas du code. Les dix règles ci-dessous, regroupées en six lignes, forment la checklist qui sert à évaluer le projet de synthèse :

- Placez chaque message à côté de son champ, et répétez-le dans un récapitulatif pour tout le formulaire.
- Écrivez en langage simple ce qu'il faut faire ensuite, et ne blâmez jamais l'utilisateur.
- N'affichez jamais de piles d'appels, de texte SQL ni d'exceptions de conversion.
- N'enregistrez jamais de données invalides, et ne piègez jamais l'utilisateur : ne bloquez pas Cancel ou Close sans vraie raison.
- Tenez les vérifications lentes ou distantes à l'écart de la validation à chaque frappe ; exécutez-les sur Save.
- Appliquez les mêmes règles de modèle depuis le formulaire, la grille et n'importe quel service, et testez des valeurs obligatoires, invalides, valides et aux limites.

Un panneau récapitulatif compte aussi pour l'accessibilité : une erreur qui n'apparaît qu'au survol d'une icône est invisible pour les utilisateurs au clavier et facile à manquer pour tout le monde. Un message qu'un lecteur d'écran peut atteindre sous forme de texte fait partie de la conception, ce n'est pas de la décoration.

Le projet de synthèse ValidationClinic

Le projet de synthèse n'est pas une grosse application. C'est la preuve que vous savez où chaque vérification a sa place. ValidationClinic doit avoir un nom et une adresse e-mail obligatoires, un `ErrorProvider` avec un récapitulatif visible, l'extender Validation avec au moins trois règles intégrées, une classe `ValidationRule` personnalisée, un service de validation utilisé par Save, une validation du modèle par DataAnnotations ou `IDataErrorInfo`, une validation des cellules e-mail et âge ainsi qu'un gestionnaire `DataError` sur la grille, le récapitulatif `IErrorProvider` personnalisé et un court README expliquant la conception.

Vous en faites ensuite la démonstration selon un script court et fixe : quittez un champ invalide, cliquez sur Save et regardez `ValidateChildren` l'arrêter, montrez une erreur de modèle qui n'apparaît qu'après `EndEdit`, tapez une adresse e-mail incorrecte dans une cellule de la grille, désignez le panneau récapitulatif qui liste tout, puis corrigez chaque problème et enregistrez avec succès. Rédigez d'abord le tableau de tests du README, avec des valeurs obligatoires, invalides, valides et aux limites pour chaque règle, et la démonstration revient à exécuter vos propres tests à voix haute.



Réflexe des couches de validation

Nommez la couche avant d'écrire la vérification. La validation des champs protège la saisie et s'exécute quand l'utilisateur quitte un contrôle ; la validation du modèle protège les données et applique les mêmes règles depuis le formulaire, la grille et n'importe quel service ; la validation à l'enregistrement protège la base de données et réexécute toutes les couches avant que quoi que ce soit ne soit écrit. Affichez chaque erreur près de son champ et dans un récapitulatif visible, dans un langage simple qui dit quoi faire ensuite, et séparez les erreurs que l'utilisateur peut corriger des pannes système.



Objectifs d'apprentissage

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

- ✓ Construire un pipeline de Save complet pour un formulaire Wisej.NET de débutant.
- ✓ Créer un IErrorProvider personnalisé pour un panneau récapitulatif.
- ✓ Combiner les erreurs de champ, les erreurs de grille et les erreurs de modèle.
- ✓ Concevoir une UX de validation claire, accessible et testable.
- ✓ Terminer l'application du projet de synthèse du cours.

Guide du lab / de l'examen

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

Lab pratique

Terminez le projet de synthèse ValidationClinic. Écrivez `btnSave_Click` comme un pipeline fixe : réinitialisez le récapitulatif, appelez `bindingSource.EndEdit()`, arrêtez-vous si `ValidateChildren()` échoue, arrêtez-vous si la vérification `DataAnnotations` échoue, arrêtez-vous si `ContactValidator` renvoie des erreurs métier, arrêtez-vous si la grille des contacts porte encore des erreurs de ligne, et seulement ensuite enregistrez dans un `try/catch` qui transforme un nom en double en erreur de champ et toute autre exception en message compréhensible. Créez `SummaryErrorProvider` en implémentant `IErrorProvider`, affectez-le à `validation.ErrorProvider`, et faites passer les erreurs par un utilitaire `SetError` qui écrit à la fois dans l'`IErrorProvider` à icônes et dans le provider récapitulatif. Testez des valeurs obligatoires, invalides, valides et aux limites, et rédigez un court README qui explique la conception de la validation.

Point de départ

ValidationClinic tel que vous l'avez laissé après le Module 6 : le formulaire d'inscription possède des gestionnaires de champ, des règles d'étendre, `MinimumAgeValidationRule`, un service `ContactValidator` et un `ContactEditModel` lié via un `BindingSource`, et la grille des contacts valide ses cellules e-mail, âge et date de clôture. Chaque couche fonctionne isolément, mais `Save` ne les exécute pas encore toutes dans l'ordre, et les erreurs apparaissent à des endroits différents selon la couche qui les a trouvées.

Approche suggérée

- ✓ Écrivez d'abord `SummaryErrorProvider` : une classe qui implémente `IErrorProvider`, reçoit le `Label` récapitulatif dans son constructeur, enregistre des paires contrôle-message, retire une entrée quand le message est vide, et réécrit le texte et la visibilité du libellé à chaque `SetError`. Implémentez `GetError` pour qu'elle renvoie une chaîne vide pour un contrôle sans erreur.
- ✓ Créez le provider autour de `validationSummaryLabel`, affectez `validation.ErrorProvider = summaryProvider`, et ajoutez au formulaire un utilitaire `SetError(Control, string)` qui appelle `errorProvider.SetError` et `summaryProvider.SetError` avec le même message. Remplacez les appels directs à `SetError` dans votre code d'association du modèle et des règles métier par cet utilitaire.
- ✓ Réécrivez `btnSave_Click` comme le pipeline fixe : réinitialisation du récapitulatif, `bindingSource.EndEdit()`, `ValidateChildren()`, la vérification `Validator.TryValidateObject`, `ContactValidator.Validate`, puis une vérification de la grille qui ignore la nouvelle ligne de substitution et signale toute ligne portant encore un `ErrorText`. Chaque étape sort dès qu'elle échoue.
- ✓ N'entourez que l'enregistrement d'un `try / catch` : interceptez d'abord `DuplicateNameException` et transformez-la en erreur sur `txtName` via votre utilitaire, puis interceptez `Exception`, journalisez-la avec `System.Diagnostics.Debug.WriteLine` et affichez

un texte compréhensible avec `MessageBox.Show` . En cas de succès, appelez

`AlertBox.Show("Saved.", MessageBoxIcon.Information)` .

- ✓ Rédigez le README : un court paragraphe par couche (champ, modèle, enregistrement), un tableau de cas de test avec des valeurs obligatoires, invalides, valides et aux limites pour chaque règle, et la checklist du projet de synthèse. Répétez ensuite le script de démonstration jusqu'à pouvoir le dérouler en cinq minutes.

Le projet de synthèse est évalué sur dix fonctionnalités obligatoires : nom et adresse e-mail du client obligatoires ; un `ErrorProvider` avec un récapitulatif visible ; l'extender Validation avec au moins trois règles intégrées ; une classe `ValidationRule` personnalisée ; un service de validation utilisé par Save ; une validation du modèle par DataAnnotations ou `IDataErrorInfo` ; la validation des cellules e-mail et âge dans la grille ; un gestionnaire `DataError` sur la grille ; le récapitulatif `IErrorProvider` personnalisé ; et un court README. La démonstration suit un ordre fixe : montrer un champ invalide, montrer Save arrêté par `ValidateChildren` , montrer une erreur de modèle qui apparaît après `EndEdit` , montrer la validation des cellules de la grille, montrer le provider récapitulatif qui liste tout, puis corriger les données et enregistrer avec succès. Avant la remise, vérifiez vos écrans au regard des dix règles d'UX de la leçon : messages près du champ et dans un récapitulatif, langage simple qui dit quoi faire, aucun reproche, aucune pile d'appels, aucun enregistrement invalide, Cancel et Close jamais bloqués sans raison, aucune vérification distante à chaque frappe, les mêmes règles de modèle partout, et les quatre types de valeurs de test.

Livrables attendus

- ✓ Pipeline de Save qui exécute EndEdit, ValidateChildren, puis les vérifications du modèle, des règles métier et de la grille avant d'enregistrer
- ✓ Classe SummaryErrorProvider implémentant SetError et GetError d'IErrorProvider
- ✓ Utilitaire SetError qui écrit à la fois dans l'ErrorProvider à icônes et dans le provider récapitulatif
- ✓ try/catch autour de l'enregistrement qui distingue une erreur de nom en double des pannes système
- ✓ README avec la conception de la validation et des cas de test pour des valeurs obligatoires, invalides, valides et aux limites

Questions de révision

- ✓ Un testeur tape une nouvelle adresse e-mail dans le formulaire et clique immédiatement sur Save ; c'est l'ancienne adresse qui est enregistrée. Quelle étape du pipeline manque ou est mal placée, et pourquoi sa position compte-t-elle ?
- ✓ Votre `SummaryErrorProvider` affiche deux fois « Email is required. » après que l'utilisateur a corrigé puis à nouveau cassé le champ. Quelle partie de `SetError` devrait l'empêcher, et de quoi dépend-elle ?
- ✓ Le repository lève une exception contenant un message de base de données quand le réseau tombe. Décrivez ce que voit l'utilisateur et ce que le développeur peut retrouver ensuite, et expliquez en quoi cela diffère du cas du nom en double.



Réflexe des couches de validation

Nommez la couche avant d'écrire la vérification. La validation des champs protège la saisie et s'exécute quand l'utilisateur quitte un contrôle ; la validation du modèle protège les données et applique les mêmes règles depuis le formulaire, la grille et n'importe quel service ; la validation à l'enregistrement protège la base de données et réexécute toutes les couches avant que quoi que ce soit ne soit écrit. Affichez chaque erreur près de son champ et dans un récapitulatif visible, dans un langage simple qui dit quoi faire ensuite, et séparez les erreurs que l'utilisateur peut corriger des pannes système.