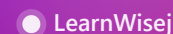


Validation liée aux données et erreurs du modèle

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



Guide de la leçon

Valider le modèle, pas seulement les contrôles : BindingSource et EndEdit, les DataAnnotations exécutées avec Validator.TryValidateObject, l'association des MemberNames aux contrôles, IDataErrorInfo avec un ErrorProvider lié aux données, BindToDataAndErrors et l'ordre d'enregistrement sûr. 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

Valider le modèle, pas seulement les contrôles : BindingSource et EndEdit, les DataAnnotations exécutées avec Validator.TryValidateObject, l'association des MemberNames aux contrôles, IDataErrorInfo avec un ErrorProvider lié aux données, BindToDataAndErrors et l'ordre d'enregistrement sûr.

1. Le problème en production

Tout ce que ValidationClinic vérifie jusqu'ici vit dans le formulaire d'inscription : un gestionnaire `Validating` ici, une règle sur l'extender `Validation` là, un appel à `ContactValidator` dans `btnSave_Click`. Mais un contact n'arrive pas uniquement par ce formulaire. Il peut être modifié dans la grille des contacts, importé depuis un fichier ou créé par un service, et aucun de ces chemins n'exécute les gestionnaires du formulaire. Ce module déplace les règles de base sur le `ContactEditModel` lui-même, pour que les données portent leur propre définition de « valide » et que chaque écran puisse leur poser la même question.

2. Concepts clés

Concept	Pourquoi c'est important
BindingSource	Un composant qui se place entre les contrôles et votre objet : son <code>DataSource</code> est le modèle, et chaque contrôle se lie à une propriété par son intermédiaire au lieu d'être recopié à la main.
EndEdit()	Pousse la valeur que l'utilisateur est encore en train de modifier du contrôle vers le modèle, pour que la validation lise ce qui est à l'écran plutôt que la dernière valeur validée.
DataAnnotations	Des attributs comme <code>[Required]</code> , <code>[StringLength]</code> , <code>[EmailAddress]</code> et <code>[Range]</code> placés sur les propriétés du modèle, chacun avec un <code>ErrorMessage</code> rédigé pour l'utilisateur.
Validator.TryValidateObject	La méthode .NET qui applique ces attributs à un objet et remplit une liste d'objets <code>ValidationResult</code> ; elle ne sait rien des contrôles ni de Wisej.NET.
MemberNames	Les noms des propriétés auxquelles appartient un <code>ValidationResult</code> , seul lien entre un message d'erreur et le contrôle qui doit l'afficher.
IDataErrorInfo	Une interface qui permet à un modèle de répondre à « qu'est-ce qui ne va pas avec cette propriété ? » au moyen d'un indexeur, pour que la couche de liaison demande les erreurs au lieu que votre code les pousse.
ErrorProvider lié aux données	Un <code>ErrorProvider</code> dont le <code>DataSource</code> est la même binding source que celle des contrôles, si bien qu'il affiche les erreurs du modèle à côté des contrôles liés sans aucun appel à <code>SetError</code> .

3. Déroulé de la leçon

Lier le formulaire, puis valider le modèle

La liaison de données (data binding) relie une propriété de contrôle à une propriété du modèle. Créez un `BindingSource`, définissez son `DataSource` sur un `ContactEditModel`, et appelez `txtName.DataBindings.Add("Text", contactBindingSource, nameof(ContactEditModel.Name))` pour chaque champ. Dès lors, le formulaire ne recopie plus le texte dans un objet au clic sur Save ; le modèle est la donnée du formulaire, et `contactBindingSource.Current` vous le rend.

Il existe un piège dans lequel un débutant tombe presque toujours. La liaison ne met pas à jour le modèle à chaque frappe. Si l'utilisateur tape une adresse e-mail et clique immédiatement sur Save, le nouveau texte peut encore se trouver dans la TextBox alors que la propriété `Email` du modèle contient l'ancienne valeur. Valider le modèle à ce moment-là vérifie des données que l'utilisateur a déjà remplacées. La correction tient en un appel : `contactBindingSource.EndEdit()` termine la modification en cours et pousse les valeurs saisies dans le modèle. Faites-en la première ligne de chaque chemin d'enregistrement, avant qu'une règle ne lise le modèle.

DataAnnotations et Validator.TryValidateObject

Les DataAnnotations placent des règles simples juste à côté des données qu'elles protègent. Sur `ContactEditModel`, `Name` reçoit `[Required]` et `[StringLength(80)]`, `Email` reçoit `[Required]` et `[EmailAddress]`, et `Age` reçoit `[Range(0, 120)]`. Chaque attribut porte un `ErrorMessage` comme « Enter a valid email address. » Traitez ces messages comme des textes produit que l'utilisateur lira, pas comme des notes de développeur.

Les attributs ne font rien par eux-mêmes ; quelque chose doit les exécuter. Créez une `List<ValidationResult>` et un `ValidationContext` pour le modèle, puis appelez `Validator.TryValidateObject(model, context, results, validateAllProperties: true)`. La méthode renvoie `false` quand quelque chose a échoué et ajoute un résultat par règle enfreinte. N'omettez pas le dernier argument : sans `validateAllProperties: true`, seul `[Required]` est vérifié, et une adresse e-mail mal formée ou un âge de 300 passent sans bruit. Notez aussi que `[EmailAddress]` et `[Range]` considèrent une valeur null comme valide, et c'est précisément pourquoi la règle d'obligation est un attribut distinct.

IDataErrorInfo et l'ErrorProvider lié aux données

`IDataErrorInfo` inverse la relation. Au lieu que votre code exécute un validateur et pousse des messages sur les contrôles, c'est le modèle qui répond aux questions. L'interface a deux membres : `Error`, un message sur l'objet dans son ensemble (souvent une chaîne vide), et l'indexeur `this[string columnName]`, qui renvoie le message d'une propriété ou une chaîne vide quand cette propriété est correcte. Dans le second modèle du lab, l'indexeur renvoie « Name is required. » quand `columnName` vaut `Name` et que le nom est vide.

L'interface devient utile quand un `ErrorProvider` est connecté aux mêmes données. Définissez `errorProvider.ContainerControl` sur le formulaire et `errorProvider.DataSource` sur `contactBindingSource`. Le provider demande alors à l'élément courant l'erreur de chaque propriété liée et affiche chaque message à côté du contrôle qui y est lié. Aucun appel à `SetError` n'apparaît dans votre code. .NET propose aussi `INotifyDataErrorInfo`, qui peut signaler plusieurs erreurs par propriété et les modifier de manière asynchrone, mais pour un formulaire de débutant, `IDataErrorInfo` est le point de départ le plus simple et le mieux pris en charge.

BindToDataAndErrors à l'exécution

Un provider lié aux données a besoin que deux réglages concordent : `DataSource`, l'objet ou la liste, et `DataMember`, le chemin à l'intérieur. Dans le designer, vous définissez les deux une fois pour toutes. À l'exécution, par exemple quand ValidationClinic fait passer l'éditeur du modèle annoté au modèle `IDataErrorInfo`, les définir l'un après l'autre crée un instant où la nouvelle source est associée à l'ancien membre, et le provider peut tenter de se lier à quelque chose qui n'existe pas.

`BindToDataAndErrors(source, member)` définit les deux en un seul appel, si bien que le provider ne voit jamais qu'une paire cohérente. Quand l'objet entier constitue les données, passez `string.Empty` comme membre : `errorProvider.BindToDataAndErrors(contactBindingSource, string.Empty)`. Utilisez les deux propriétés dans le designer, et la méthode chaque fois que la source change alors que le formulaire est ouvert.

Associer les erreurs du modèle aux contrôles et au récapitulatif

`Validator.TryValidateObject` ne sait volontairement rien de Wisej.NET ; le formulaire doit donc traduire ses résultats. Construisez une table de correspondance des champs dans le constructeur, un `Dictionary<string, Control>` de `nameof(ContactEditModel.Email)` vers `txtEmail` et ainsi de suite, puis parcourez les résultats : pour chaque nom de `result.MemberNames`, recherchez le contrôle et appelez `errorProvider.SetError(control, result.ErrorMessage)`. Enfin, regroupez tous les messages dans `validationSummaryLabel` et affichez-le, car une icône seule passe facilement inaperçue. Un résultat dont le membre ne figure pas dans la table atteint tout de même le récapitulatif, donc aucune erreur n'est jamais perdue. La table a la même forme qu' `ApplyModelValidation` du Module 4, et c'est pourquoi les deux peuvent partager un même récapitulatif.

Une fois toutes les couches en place, le gestionnaire d'enregistrement suit un ordre fixe : terminer la modification, valider les contrôles enfants, valider le modèle, exécuter les règles inter-champs, enregistrer seulement si rien n'a échoué, et intercepter séparément les exceptions système pour qu'une panne de base de données ne ressemble jamais à une faute de frappe.

```
private void btnSave_Click(object sender, EventArgs e)
{
    contactBindingSource.EndEdit();           // 1. typed values into the model
    if (!ValidateChildren()) return;           // 2. field validation
    var model = (ContactEditModel)contactBindingSource.Current;
    if (!ApplyAnnotationErrors(model)) return; // 3. DataAnnotations
    if (!ApplyModelValidation(model)) return;  // 4. ContactValidator, cross-field
    try { _repository.Save(model); ShowSaved(); } // 5. only now
    catch (Exception ex) { ShowSystemError(ex); } // 6. not a validation message
}
```



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 :

- ✓ Lier des contrôles à un modèle et valider le modèle avant l'enregistrement.
- ✓ Utiliser les DataAnnotations avec `Validator.TryValidateObject`.
- ✓ Comprendre `IDataErrorInfo` et le comportement d'un `ErrorProvider` lié aux données.
- ✓ Utiliser `BindToDataAndErrors` pour changer la source de données d'un provider à l'exécution.
- ✓ Associer les erreurs du modèle aux contrôles et aux récapitulatifs Wisej.NET.

Guide du lab / de l'examen

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



Lab pratique

Liez le formulaire d'inscription de `ValidationClinic` à un `ContactEditModel` via un `contactBindingSource` et ajoutez les annotations `Required`, `StringLength`, `EmailAddress` et `Range` à ses propriétés `Name`, `Email` et `Age`. Dans `btnSave_Click`, appelez d'abord `contactBindingSource.EndEdit()`, exécutez `Validator.TryValidateObject` avec `validateAllProperties` à `true`, et associez chaque nom de membre des `ValidationResult` à son contrôle via une table de correspondance des champs et le récapitulatif de validation. Implémentez ensuite `IDataErrorInfo` sur un second modèle, faites pointer l'`ErrorProvider` vers la binding source avec `BindToDataAndErrors`, et montrez les mêmes erreurs apparaître par liaison de données sans appel à `SetError` écrit à la main.

Point de départ

`ValidationClinic` après le Module 4 : le formulaire d'inscription possède des règles de champ, un service `ContactValidator`, `ApplyModelValidation` et une `MinimumAgeValidationRule`, mais ses contrôles ne sont liés à rien, si bien que `btnSave_Click` lit encore les valeurs directement dans `txtName`, `txtEmail` et `txtAge`.

Approche suggérée

- ✓ Créez `ContactEditModel` avec `Name`, `Email` et un `Age` nullable, ajoutez les quatre annotations avec des textes `ErrorMessage` en langage simple, et liez-y les trois `TextBox` avec `DataBindings.Add("Text", contactBindingSource, ...)`.
- ✓ Avant d'écrire la moindre validation, mettez en évidence le décalage de la modification : tapez une adresse e-mail, cliquez sur `Save` et journalisez `((ContactEditModel)contactBindingSource.Current).Email` avec et sans un appel à `contactBindingSource.EndEdit()` juste avant.
- ✓ Écrivez `ValidateModel` autour de `Validator.TryValidateObject(model, context, results, validateAllProperties: true)`, puis construisez `_fieldMap` dans le constructeur et une méthode `ApplyAnnotationErrors` qui définit l'erreur de chaque contrôle et remplit `validationSummaryLabel`.
- ✓ Réécrivez `btnSave_Click` dans l'ordre sûr : `EndEdit`, `ValidateChildren`, annotations, le `ContactValidator` du Module 4, enregistrement, et un `catch` distinct qui affiche une erreur système compréhensible.
- ✓ Ajoutez un second modèle qui implémente `IDataErrorInfo`, basculez la binding source sur ce modèle à l'exécution, appelez `errorProvider.BindToDataAndErrors(contactBindingSource, string.Empty)`, et videz le nom pour voir l'erreur apparaître sans aucun `SetError`.

Livrables attendus

- ✓ `ContactEditModel` avec `DataAnnotations`, lié au formulaire via un `BindingSource`
- ✓ Gestionnaire de `Save` qui appelle `EndEdit` avant d'exécuter `Validator.TryValidateObject`

- ✓ Table de correspondance des champs qui transforme les noms de membres des ValidationResult en erreurs de contrôle et en lignes du récapitulatif
- ✓ Modèle IDataErrorInfo dont les erreurs apparaissent via un ErrorProvider lié aux données
- ✓ Appel à BindToDataAndErrors utilisé quand la source de données du provider change à l'exécution

Questions de révision

- ✓ Un testeur tape une nouvelle adresse e-mail et appuie sur Entrée pour enregistrer, et c'est l'ancienne adresse, valide, qui est vérifiée et stockée. Quelle ligne manque, et pourquoi le formulaire paraît-il correct à l'écran alors que le modèle est faux ?
- ✓ Vous retirez `validateAllProperties: true` et chaque enregistrement « fonctionne » encore. Laquelle de vos quatre annotations a cessé de s'exécuter, et quelles données incorrectes peuvent désormais atteindre le repository ?
- ✓ Le service d'import crée des contacts sans ouvrir le formulaire. Quelles règles de ce module protègent encore ce chemin, et lesquelles ne s'exécutent que parce qu'un utilisateur a cliqué sur Save ?



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.