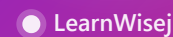


Validation programmatique et retour avec ErrorProvider

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



Guide de la leçon

Un seul ErrorProvider par formulaire, SetLastError et l'effacement des erreurs contrôle par contrôle, InvalidMessage pour un retour local simple, un petit type de résultat FieldValidation qui garde les règles hors de l'interface, et un récapitulatif de validation visible pour qu'aucune erreur ne dépende du survol d'une icône. 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

Un seul ErrorProvider par formulaire, SetLastError et l'effacement des erreurs contrôle par contrôle, InvalidMessage pour un retour local simple, un petit type de résultat FieldValidation qui garde les règles hors de l'interface, et un récapitulatif de validation visible pour qu'aucune erreur ne dépende du survol d'une icône.

1. Le problème en production

Le Module 1 a doté le formulaire d'inscription de ValidationClinic de deux gestionnaires `Validating`, et cela fonctionne pour deux champs. Un vrai formulaire de contact comporte un nom, une adresse e-mail, un numéro de téléphone, un code client et plus encore ; si chaque champ reçoit son propre gestionnaire écrit à la main, avec sa propre manière d'afficher un message, le formulaire devient vite incohérent : un champ affiche une icône, un autre rien du tout, un troisième laisse une ancienne erreur à l'écran alors que l'utilisateur l'a corrigée. Ce module remplace cette dérive par un seul modèle : de petites méthodes utilitaires décident de ce qui ne va pas, une méthode affiche le résultat, et un récapitulatif visible indique à chaque utilisateur ce qu'il doit corriger sans l'obliger à survoler quoi que ce soit.

2. Concepts clés

Concept	Pourquoi c'est important
<code>ErrorProvider</code>	Un composant extender qui dessine une icône d'erreur à côté de tout contrôle auquel vous attribuez un message et affiche ce message en infobulle ; une seule instance sert tout le formulaire.
<code>ContainerControl</code>	Le formulaire ou le contrôle utilisateur auquel appartient le provider ; le définir sur <code>this</code> indique au provider sur quelle surface dessiner ses icônes.
<code>setError(control, message)</code>	Affiche <code>message</code> à côté de <code>control</code> ; l'appeler avec une chaîne vide ou <code>null</code> retire l'erreur de ce seul contrôle.
<code>GetError</code> , <code>HasErrors</code> , <code>Clear()</code>	Lire le message actuel d'un contrôle, demander s'il reste une erreur sur le formulaire, ou retirer toutes les erreurs d'un coup quand le formulaire entier est réinitialisé.
<code>InvalidMessage</code>	Une propriété des contrôles d'édition qui prennent en charge l'interface <code>IValidation</code> ; le contrôle affiche lui-même le message dans son infobulle d'invalidité, sans provider distinct.
Record <code>FieldValidation</code>	Un tout petit type de résultat contenant le contrôle, un nom de champ et un message, où un message vide signifie valide ; il permet aux règles de renvoyer des réponses au lieu de modifier l'écran.
<code>validationSummaryLabel</code>	Un simple Label qui liste en texte tous les problèmes en cours, visible uniquement tant qu'il existe des erreurs, pour que le retour ne dépende jamais du fait de voir ou de survoler une icône.

3. Déroulé de la leçon

Afficher et effacer les erreurs de champ avec `ErrorProvider`

Créez un seul `ErrorProvider` pour le formulaire, soit en le déposant depuis la boîte à outils sur le designer, soit comme champ créé dans le code, et définissez son `ContainerControl` sur le formulaire. Deux réglages facultatifs le rendent plus reposant pour les utilisateurs : `BlinkStyle = ErrorBlinkStyle.NeverBlink` empêche l'icône de clignoter, et `SetIconAlignment(txtEmail, ErrorIconAlignment.MiddleRight)` avec `SetIconPadding(txtEmail, 4)` place l'icône juste à droite du champ, là où l'œil l'attend.

Afficher une erreur tient en un appel : `errorProvider.SetError(txtName, "Name is required.")`. Rappelez-vous que Wisej.NET exécute cet appel sur le serveur ; le navigateur reçoit simplement l'icône et l'infobulle mises à jour. Effacer, c'est le même appel avec une chaîne vide. Ce détail compte plus qu'il n'y paraît. Quand l'utilisateur corrige le nom, vous effacez l'erreur de `txtName` et rien d'autre. `Clear()` retire toutes les erreurs du formulaire ; réservez-le donc aux moments où le formulaire entier repart vraiment de zéro : chargement d'un autre contact, réinitialisation du formulaire, ou passe de validation complète qui réapplique immédiatement toutes les erreurs en cours. `GetError(control)` vous indique ce qu'affiche actuellement un contrôle, et `HasErrors` sert de vérification finale rapide avant que Save ne continue.

InvalidMessage ou SetError ?

Wisej.NET vous offre une seconde manière, plus légère, de donner un retour. Les contrôles d'édition qui implémentent `Invalidation` possèdent une propriété `InvalidMessage`. Définissez-la dans le gestionnaire `Validating` du contrôle quand la valeur est incorrecte, remettez-la à une chaîne vide quand la valeur est correcte, et le contrôle affiche le message dans sa propre infobulle d'invalidité. Pas de provider, pas de placement d'icône, rien de partagé.

`InvalidMessage` convient donc bien à un formulaire très simple ou à un champ dont le retour doit rester local. `SetError` est préférable quand un formulaire comporte plusieurs champs, car toutes les erreurs ont alors le même aspect, peuvent être relues avec `GetError` et comptées avec `HasErrors`. Dans le lab, vous utilisez les deux volontairement : le champ du code client utilise `InvalidMessage`, tous les autres champs passent par l' `ErrorProvider`, et vous rédigez une courte note comparant ce que voit l'utilisateur.

Un libellé récapitulatif lisible par tous

Une icône d'erreur est petite, elle n'est qu'une couleur et une forme, et son texte n'apparaît que lorsque le pointeur s'y arrête. Un utilisateur au clavier, un utilisateur sur écran tactile ou un utilisateur malvoyant peut la manquer complètement. C'est pourquoi chaque formulaire de ce cours affiche aussi ses erreurs en texte simple. La version la plus simple est un Label nommé `validationSummaryLabel`, placé au-dessus du bouton Save, avec une ligne par problème, par exemple *Email: Enter a valid email address*.

Deux règles gardent le récapitulatif fiable. Il n'est visible que tant qu'au moins une erreur existe, pour qu'aucun cadre rouge vide ne traîne à l'écran. Et il est reconstruit à partir des mêmes résultats que ceux qui définissent les icônes, pour que les icônes et le récapitulatif ne puissent jamais se contredire. Un récapitulatif visible rend aussi le formulaire plus facile à tester : vous pouvez vérifier le texte du libellé au lieu de survoler les icônes.

Une seule méthode utilitaire pour plusieurs champs

L'étape clé de ce module consiste à séparer la décision de l'affichage. Chaque règle devient une petite méthode, `ValidateName`, `ValidateEmail` ou `ValidatePhone`, qui lit une valeur et renvoie un record `FieldValidation`. Elle n'appelle jamais `SetError` et ne touche jamais à un Label. Une seule méthode `Apply` prend ensuite tous les résultats et effectue l'affichage en un seul endroit.

```
public sealed record FieldValidation(Control Control, string FieldName, string Message)
{
    public bool IsValid => string.IsNullOrEmpty(Message);
}

private bool Apply(params FieldValidation[] results)
{
    foreach (var result in results)
        errorProvider.SetError(result.Control, result.Message);
    validationSummaryLabel.Text = string.Join("\n", results
        .Where(r => !r.IsValid).Select(r => $"{r.FieldName}: {r.Message}"));
    validationSummaryLabel.Visible = results.Any(r => !r.IsValid);
    return results.All(r => r.IsValid);
}
```

Remarquez qu' `Apply` appelle `SetError` pour chaque résultat, valide ou non. Un résultat valide porte un message vide, donc le transmettre efface l'icône de ce contrôle. C'est exactement ainsi que corriger un champ n'efface que l'erreur de ce champ : les autres résultats portent toujours leurs messages, donc leurs icônes restent. Le bouton Save appelle `Apply(ValidateName(), ValidateEmail(), ValidatePhone())` et sort immédiatement s'il obtient `false`.

Retour sur les champs et retour sur les règles métier

Tous les messages n'ont pas leur place à côté d'un champ. Le *retour sur un champ* concerne la valeur d'un contrôle : elle est vide, elle n'a pas de @, le numéro de téléphone est trop court. L'utilisateur peut corriger sans quitter ce contrôle, donc le message va à côté du contrôle et dans le récapitulatif. Le *retour sur une règle métier* concerne le sens des données : un contact portant ce nom existe déjà, ou deux dates ne concordent pas. Ces règles ont souvent besoin d'autres champs ou de données enregistrées, et elles s'exécutent généralement sur Save plutôt que lorsque l'utilisateur quitte un champ avec Tab.

Séparez les deux dans votre code comme dans votre formulation. Les utilitaires de champ restent petits et rapides. Une règle métier peut tout de même désigner le contrôle le plus pertinent avec `SetError`, mais son message explique la règle en langage simple et apparaît dans le récapitulatif, pour que l'utilisateur comprenne pourquoi Save s'est arrêté. Aucun des deux n'est une exception : une adresse e-mail incorrecte est une erreur attendue avec un message produit normal, alors qu'une connexion perdue à la base de données est une panne système traitée à part. Les modules suivants déplacent les règles métier dans un service réutilisable ; ce module installe le réflexe.



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 :

- ✓ Utiliser `ErrorProvider` pour afficher et effacer des erreurs au niveau des champs.
- ✓ Comprendre la différence entre `InvalidMessage` et `ErrorProvider.SetError`.
- ✓ Créer un libellé récapitulatif de validation pour l'accessibilité.
- ✓ Valider plusieurs champs à partir d'une seule méthode utilitaire.
- ✓ Séparer le retour sur les champs du retour sur les règles métier.

Guide du lab / de l'examen

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



Lab pratique

Transformez le formulaire de contact de ValidationClinic en formulaire à validation programmatique. Créez un seul `ErrorProvider` pour le formulaire avec `ContainerControl` défini, `NeverBlink` et une icône alignée à droite, écrivez un record `FieldValidation` ainsi que des utilitaires `ValidateName`, `ValidateEmail` et `ValidatePhone` qui renvoient un message au lieu de toucher à l'interface, et une méthode `Apply` qui appelle `SetError` pour chaque résultat et remplit un `validationSummaryLabel` visible uniquement tant qu'il existe des erreurs. Utilisez `InvalidMessage` sur le champ du code client pour comparer les deux styles de retour, et assurez-vous que corriger un champ n'efface que l'erreur de ce champ.

Point de départ

Vous ouvrez le projet `ValidationClinic` que vous avez construit au Module 1 — ou, si vous arrivez ici directement, la version du Module 1 disponible dans le [dépôt d'exemples public](#). Le formulaire d'inscription client comporte déjà `txtName`, `txtEmail`, un bouton `Save` et un bouton `Cancel`, avec les gestionnaires `Validating` et le garde-fou `ValidateChildren()` écrits la dernière fois ; dans ce lab, vous ajoutez `txtPhone`, `txtCustomerCode` et un Label récapitulatif, et vous déplacez les vérifications dans des méthodes utilitaires.

Approche suggérée

- ✓ Ajoutez un `ErrorProvider` au formulaire et, dans une méthode `SetupValidation` appelée depuis le constructeur, définissez `ContainerControl = this`, `BlinkStyle = ErrorBlinkStyle.NeverBlink` et `SetIconAlignment(..., ErrorIconAlignment.MiddleRight)` avec un petit `SetIconPadding` pour chaque champ.
- ✓ Déclarez le record `FieldValidation`, puis écrivez `ValidateName`, `ValidateEmail` et `ValidatePhone` pour que chacune supprime les espaces de la valeur, la vérifie et renvoie un record avec un message en langage simple, ou un message vide quand la valeur est correcte. Aucune d'elles ne doit appeler `SetError`.
- ✓ Placez un Label nommé `validationSummaryLabel` au-dessus des boutons avec `Visible = false`, et écrivez `Apply(params FieldValidation[] results)` pour qu'elle définisse ou efface l'erreur de chaque contrôle, remplisse le libellé avec une ligne par problème, n'affiche le libellé que tant qu'il existe des problèmes et renvoie si tout est valide.
- ✓ Appelez `Apply(ValidateName(), ValidateEmail(), ValidatePhone())` au début de `btnSave_Click` et sortez quand le résultat est `false`. Faites en sorte que le gestionnaire `Validating` de chaque champ définisse ou efface uniquement l'erreur de son propre contrôle à partir du résultat de son utilitaire et, tant que le récapitulatif est visible après un `Save` échoué, relancez l' `Apply` complet pour que la liste se réduise à mesure que les champs sont corrigés.
- ✓ Dans `txtCustomerCode_Validating`, utilisez `InvalidMessage` au lieu du provider, puis exécutez le formulaire avec plusieurs valeurs invalides, corrigez un champ à la fois et rédigez une courte note sur la différence entre les deux styles de retour pour l'utilisateur.

Livrables attendus

- ✓ ErrorProvider au niveau du formulaire avec ContainerControl, BlinkStyle et alignement des icônes configurés
- ✓ Record FieldValidation et méthodes utilitaires ValidateName, ValidateEmail et ValidatePhone
- ✓ Méthode Apply qui définit ou efface l'erreur de chaque contrôle et renvoie si tous les champs sont valides
- ✓ validationSummaryLabel listant tous les problèmes en cours et masqué quand il n'y en a aucun
- ✓ Un champ utilisant InvalidMessage au lieu d'ErrorProvider, avec une note comparant les deux

Questions de révision

- ✓ Le gestionnaire `txtName_Validating` d'un collègue appelle `errorProvider.Clear()` dès que le nom devient valide. Qu'est-ce qui ne va pas pour un utilisateur qui a aussi saisi une adresse e-mail invalide, et qu'est-ce que le gestionnaire devrait appeler à la place ?
- ✓ Le champ du code client affiche son message via `InvalidMessage`. Que devrait savoir le libellé récapitulatif sur ce champ pour le lister lui aussi, et qu'est-ce que cela vous apprend sur les cas où un provider partagé est le meilleur choix ?
- ✓ Un utilisateur saisit une adresse e-mail bien formée qui appartient déjà à un autre contact. S'agit-il d'un retour sur un champ ou sur une règle métier, quand la vérification doit-elle s'exécuter, et où le message doit-il apparaître ?



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.