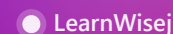


Fondamentaux de la validation dans Wisej.NET

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



Guide de la leçon

Ce que signifie la validation dans une interface Wisej.NET côté serveur : le cycle de vie de Validating et Validated, le rejet d'une saisie invalide avec e.Cancel, le comportement du focus selon AutoValidate, CausesValidation sur les boutons Cancel et ValidateChildren comme garde-fou de Save. 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

Ce que signifie la validation dans une interface Wisej.NET côté serveur : le cycle de vie de Validating et Validated, le rejet d'une saisie invalide avec e.Cancel, le comportement du focus selon AutoValidate, CausesValidation sur les boutons Cancel et ValidateChildren comme garde-fou de Save.

1. Le problème en production

ValidationClinic est l'application d'exemple que vous construisez vous-même au fil de ce cours : le Guide du lab / de l'examen de ce module explique précisément comment démarrer, et l'Exercice de code avec l'IA peut la faire assembler par un assistant. Son formulaire d'inscription client accepte actuellement n'importe quoi : un nom vide, une adresse e-mail sans @, ou un clic sur Save sur un formulaire que personne n'a rempli. Le premier réflexe de la plupart des débutants est une instruction `if` dans le bouton Save, mais l'utilisateur ne reçoit alors aucun retour pendant la saisie et Save se transforme en mur d'erreurs tout à la fin. Ce module place la validation là où elle doit se trouver dans un formulaire Wisej.NET : sur chaque éditeur quand l'utilisateur le quitte, puis une nouvelle fois quand il clique sur Save.

2. Concepts clés

Concept	Pourquoi c'est important
validation	La décision de savoir si une valeur est acceptable avant que l'application ne l'utilise, suivie d'un retour visible et, quand la valeur est incorrecte, d'un refus de continuer.
<code>Validating</code>	L'événement qu'un contrôle déclenche quand l'utilisateur tente de le quitter (ou quand le code demande une validation) ; son gestionnaire lit la valeur et c'est le seul endroit où l'on décide d'accepter ou de rejeter.
<code>CancelEventArgs.Cancel</code>	L'indicateur qu'un gestionnaire <code>Validating</code> met à <code>true</code> pour rejeter la valeur : le contrôle est considéré comme invalide, le focus peut y être retenu et <code>ValidateChildren()</code> signale l'échec.
<code>Validated</code>	L'événement déclenché uniquement après un <code>Validating</code> terminé sans annulation ; l'endroit pour les finitions, comme supprimer les espaces d'un texte, jamais pour enregistrer.
<code>AutoValidate</code>	Une propriété d'un formulaire, d'une page ou d'un contrôle utilisateur qui détermine ce qu'il advient du focus quand un contrôle enfant échoue à la validation.
<code>CausesValidation</code>	Un réglage propre à chaque contrôle ; mis à <code>false</code> sur un bouton comme Cancel, un clic sur ce bouton ne valide pas le champ que l'utilisateur quitte.
<code>ValidateChildren()</code>	Une méthode de conteneur qui lance la validation de ses contrôles enfants, y compris les champs que l'utilisateur n'a jamais visités, et renvoie <code>false</code> si un gestionnaire a annulé.

3. Déroulé de la leçon

La validation dans une interface côté serveur

Dans Wisej.NET, vous construisez un formulaire comme n'importe quelle classe C# : un `CustomerIntakeForm` avec des champs nommés `txtName`, `txtEmail` et `btnSave`. Cet objet formulaire et ses contrôles vivent sur le serveur, dans la session de l'utilisateur. Le navigateur les dessine et signale ce que fait l'utilisateur (saisie, clics, déplacement du focus), et vos gestionnaires d'événements s'exécutent comme du C# ordinaire sur le serveur. Ainsi, quand l'utilisateur appuie sur Tab pour quitter `txtEmail`, le navigateur informe le serveur que le focus se déplace, le serveur déclenche l'événement de validation de `txtEmail`, votre gestionnaire lit `txtEmail.Text`, et le résultat, par exemple une icône d'erreur ou un focus retenu, est renvoyé puis dessiné.

Cela a deux conséquences pratiques. La décision est prise par votre code serveur, donc le navigateur ne peut pas la contourner. Et chaque changement de focus avec validation est un aller-retour, donc un gestionnaire doit être rapide et prévisible : examiner ce seul champ, puis définir ou effacer un message. Interroger un service distant à chaque appui sur Tab rend le formulaire lent ; gardez ce type de vérification pour Save. Enfin, un message de validation est un texte produit écrit pour l'utilisateur, comme « Enter the customer's name. », et non un message d'exception.

Validating décide, Validated fait les finitions

Le cycle de vie suit un ordre fixe. L'utilisateur modifie un contrôle, puis tente de le quitter, de cliquer sur un bouton qui valide ou d'enregistrer le formulaire. Le contrôle déclenche `Validating`. Votre gestionnaire examine la valeur : si elle est invalide, il définit un message et annule ; si elle est valide, il efface tout ancien message. Ce n'est que si `Validating` n'a pas annulé que le contrôle déclenche `Validated`.

Séparez bien les rôles. `Validating` est l'endroit où la décision est prise. `Validated` sert au nettoyage d'une valeur déjà acceptée, par exemple supprimer les espaces qu'un utilisateur a tapés autour d'un nom. N'enregistrez jamais depuis `Validated` : un utilisateur peut valider le même champ de nombreuses fois avant de cliquer sur Save. Vous pouvez attacher les deux gestionnaires depuis la liste des événements de la fenêtre Propriétés de Visual Studio, ou dans le constructeur avec `+=`.

```
private void txtName_Validating(object sender, CancelEventArgs e)
{
    if (string.IsNullOrEmpty(txtName.Text))
    {
        e.Cancel = true;
        errorProvider.SetError(txtName, "Enter the customer's name.");
        return;
    }
    errorProvider.SetError(txtName, "");
}

private void txtName_Validated(object sender, EventArgs e)
{
    txtName.Text = txtName.Text.Trim();
}
```

Rejeter une valeur avec e.Cancel

Un gestionnaire `Validating` reçoit un `CancelEventArgs` comme second paramètre. Définir `e.Cancel = true` constitue le rejet ; le message de l' `ErrorProvider` en est l'explication, et il vous faut les deux. Un message sans `e.Cancel` est le bug de débutant le plus courant : le champ a l'air invalide, mais la valeur est acceptée, le focus passe au champ suivant et `ValidateChildren()` indique plus tard que tout va bien. Annuler sans message pose le problème inverse : l'utilisateur est bloqué et ne sait pas pourquoi.

Pour le message, ajoutez un `ErrorProvider` au formulaire et définissez son `ContainerControl` sur le formulaire. `errorProvider.SetError(txtName, "Enter the customer's name.")` affiche à côté du contrôle une icône d'erreur portant ce texte, et `SetError(txtName, "")` la retire. La branche qui efface compte : sans elle, un champ corrigé continue d'afficher son ancienne erreur. Le Module 2 étudie `ErrorProvider` en détail. Par ailleurs, ne levez pas d'exceptions pour des erreurs ordinaires comme un @ manquant. Les exceptions sont réservées aux pannes exceptionnelles, comme un repository injoignable.

Choisir un mode AutoValidate

Ce qu'il advient du focus quand un gestionnaire annule est déterminé par la propriété `AutoValidate` du conteneur : le formulaire, la page ou le contrôle utilisateur qui contient les éditeurs. Il existe quatre possibilités :

- `EnablePreventFocusChange` : le focus reste dans le champ invalide jusqu'à ce qu'il soit corrigé.

- `EnableAllowFocusChange` : le focus peut se déplacer et l'erreur reste visible ; Save refuse toujours.
- `Disable` : rien n'est validé lors d'un changement de focus ; la validation ne s'exécute que lorsque le code appelle `Validate()` ou `ValidateChildren()`.
- `Inherit` : le conteneur utilise le réglage de son parent.

Le formulaire du lab a deux champs obligatoires, donc `EnablePreventFocusChange` est un choix clair et adapté aux débutants. Il est aussi strict : si le nom est vide et que l'utilisateur clique sur Cancel, la TextBox invalide peut refuser de céder le focus et Cancel semble cassé. La solution est

`btnCancel.CausesValidation = false`. Cliquer sur un contrôle dont `CausesValidation` vaut `false` ne valide pas le champ que l'on quitte, donc le gestionnaire de Cancel s'exécute et l'utilisateur peut toujours sortir. Save conserve la valeur par défaut `true`. Votre note de lab consigne le mode choisi et la raison pour laquelle Cancel fonctionne toujours.

ValidateChildren() comme garde-fou de Save

La validation des champs ne s'exécute que pour les champs que l'utilisateur quitte réellement. Un utilisateur peut taper un nom et cliquer sur Save sans jamais entrer dans `txtEmail`, si bien que son gestionnaire `Validating` ne s'est jamais exécuté. C'est pourquoi la première instruction de `btnSave_Click` est `if (!this.ValidateChildren()) return;`. `ValidateChildren()` déclenche la validation des contrôles enfants du formulaire, visités ou non, affiche tous les messages définis par leurs gestionnaires et renvoie `false` si l'un d'eux a annulé. Une surcharge accepte des `ValidationConstraints` pour limiter les enfants vérifiés, par exemple aux seuls contrôles activés, sélectionnables, arrêts de tabulation ou enfants directs, ce qui est utile quand une partie du formulaire est masquée ou désactivée.

`Validate()` est l'outil plus restreint : il valide le contrôle qui perd le focus, ce qui est utile quand `AutoValidate` vaut `Disable` et que vous voulez vérifier uniquement le champ courant. Une fois `ValidateChildren()` passé, placez l'enregistrement proprement dit dans un `try / catch`, pour qu'une écriture échouée affiche un message système compréhensible et ne soit jamais confondue avec une erreur de saisie corrigable. La validation des champs protège la saisie ; la validation à l'enregistrement protège l'enregistrement. Il vous faut les deux.



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 :

- ✓ Expliquer ce que signifie la validation dans une interface Wisej.NET côté serveur.
- ✓ Gérer Validating et Validated sur les contrôles d'édition.
- ✓ Annuler une saisie invalide en définissant CancelEventArgs.Cancel.
- ✓ Choisir un mode AutoValidate pour un formulaire ou un contrôle utilisateur.
- ✓ Appeler Validate et ValidateChildren au bon moment.

Guide du lab / de l'examen

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



Lab pratique

Créez le formulaire d'inscription client de ValidationClinic avec les TextBox txtName et txtEmail, un bouton Save et un bouton Cancel. Attachez des gestionnaires Validating qui définissent e.Cancel = true et un message ErrorProvider quand le nom est vide ou que l'adresse e-mail ne contient pas de @, effacent le message quand la valeur est valide, et suppriment les espaces du nom dans son gestionnaire Validated. Définissez le mode AutoValidate du formulaire, assurez-vous que Cancel ne déclenche pas de validation pour que l'utilisateur puisse toujours sortir, et appelez ValidateChildren() au début de btnSave_Click pour que Save s'arrête dès qu'un champ est invalide. Terminez par une courte note de lab qui explique le comportement du focus que vous avez choisi.

Point de départ

ValidationClinic est l'application que vous construisez vous-même, module après module : rien n'est fourni en téléchargement et aucune solution de départ n'est installée pour vous. Créez dans Visual Studio un nouveau projet Wisej.NET 4 nommé **ValidationClinic**, donnez-lui une page principale, une grille de contacts modifiable et un repository de contacts simple en mémoire derrière cette grille, et ne validez encore rien nulle part ; chaque module suivant étend ce même projet. Deux raccourcis si vous préférez ne pas assembler la coquille à la main : la section **Exercice de code avec l'IA** de ce module remet à ChatGPT ou Claude un prompt prêt à l'emploi qui construit toute la solution à partir de la spécification du module, et notre propre version terminée de chaque module de ce cours se trouve dans le dépôt d'exemples public **LearnWisej-Samples**, avec un dossier par module : **Module 1** contient cette coquille et le corrigé de ce lab. Dans ce lab, vous ajoutez le formulaire d'inscription client et lui donnez ses deux premières couches de validation : la validation des champs sur chaque éditeur et la validation à l'enregistrement sur le bouton Save.

Approche suggérée

- ✓ Ajoutez un **CustomerIntakeForm** et utilisez le designer pour placer **txtName**, **txtEmail**, **btnSave**, **btnCancel** et un libellé **lblStatus**. Déposez un **ErrorProvider** sur le formulaire et définissez son **ContainerControl** sur le formulaire.
- ✓ Gérez **txtName.Validating** et **txtEmail.Validating**. Dans chacun, testez la valeur ; quand elle est incorrecte, définissez **e.Cancel = true** et appelez **errorProvider.SetError** avec un message en langage simple, et quand elle est correcte, appelez **SetError** avec une chaîne vide.
- ✓ Gérez **txtName.Validated** et supprimez-y les espaces du texte, pour que la valeur ne soit nettoyée qu'après avoir été acceptée.
- ✓ Définissez l'**AutoValidate** du formulaire, puis **btnCancel.CausesValidation = false**. Exécutez le formulaire, laissez le nom vide et cliquez sur Cancel pour prouver que l'utilisateur peut toujours sortir.
- ✓ Faites de **if (!this.ValidateChildren()) return;** la première ligne de **btnSave_Click**, puis enregistrez dans un **try / catch**. Testez Save avec un champ dans lequel vous n'avez jamais

cliqué, et rédigez la note de lab.

Livrables attendus

- ✓ Formulaire d'inscription client avec txtName et txtEmail, un bouton Save et un bouton Cancel
- ✓ Gestionnaires Validating qui définissent e.Cancel = true et un message d'erreur pour un nom vide et une adresse e-mail invalide
- ✓ Gestionnaire Validated qui supprime les espaces du nom accepté
- ✓ Gestionnaire de Save qui appelle ValidateChildren() et s'arrête quand la méthode renvoie false
- ✓ Note de lab indiquant le mode AutoValidate choisi et pourquoi Cancel fonctionne toujours sur un formulaire invalide

Questions de révision

- ✓ Vous retirez `e.Cancel = true` de `txtEmail_Validating` mais conservez l'appel à `SetError`. Décrivez précisément ce que voit l'utilisateur quand il quitte avec Tab le champ contenant « maria.example.com », puis clique sur Save.
- ✓ Parmi vos vérifications, lesquelles s'exécutent pendant la saisie, et lesquelles uniquement parce que Save a appelé `ValidateChildren()` ? Donnez une action utilisateur qui atteint Save avec un champ dont le gestionnaire ne s'est jamais exécuté.
- ✓ Vous passez le formulaire de `EnablePreventFocusChange` à `EnableAllowFocusChange`. Qu'est-ce qui change pour l'utilisateur, qu'est-ce qui reste identique, et `btnCancel.CausesValidation = false` est-il toujours utile ?



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.