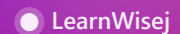


Validation, ErrorProvider et retours utilisateur

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



Guide de la leçon

Les trois niveaux de validation, les DataAnnotations sur les modèles d'édition, un service TicketValidator qui renvoie des messages par champ et un récapitulatif, ErrorProvider et le récapitulatif affiché dans Wisej.NET, et DbUpdateException comme dernier filet de sécurité. Ce que cela signifie pour un développeur Wisej.NET qui travaille avec de vraies bases de données — et comment aborder ce module.



Objectif production

Les trois niveaux de validation, les DataAnnotations sur les modèles d'édition, un service TicketValidator qui renvoie des messages par champ et un récapitulatif, ErrorProvider et le récapitulatif affiché dans Wisej.NET, et DbUpdateException comme dernier filet de sécurité.

1. Le problème en production

L'éditeur de tickets du Module 4 enregistre tout ce que l'utilisateur a saisi. Un titre vide, un ticket sans client ou un ticket fermé dont l'échéance tombe la semaine prochaine passent tous par

`SaveChangesAsync` et ne sont arrêtés, au mieux, que par la base de données, qui répond par une exception du fournisseur qu'un agent du support ne peut pas lire. Les utilisateurs ont besoin de messages immédiats et compréhensibles avant que la base ne rejette un enregistrement, et l'application a besoin d'une dernière ligne de défense qui reste en place quand l'interface est contournée par un import, un second écran ou un bug.

2. Concepts clés

Concept	Pourquoi c'est important
trois niveaux de validation	La validation de l'interface donne un retour immédiat, la validation du domaine impose les règles qui doivent être vraies partout, et les contraintes de la base garantissent la cohérence finale ; chaque niveau intercepte ce que le précédent a laissé passer.
DataAnnotations sur le modèle d'édition	Des attributs comme <code>[Required]</code> et <code>[StringLength]</code> sur <code>TicketEditModel</code> décrivent les règles de saisie en un seul endroit déclaratif, séparé de l'entité EF Core.
validation explicite	<code>Validator.TryValidateObject</code> exécute les annotations à la demande ; EF Core ne les exécute pas pour vous avant <code>SaveChangesAsync</code> , donc le formulaire doit l'appeler.
TicketValidator et ValidationMessage	Un service qui renvoie des éléments <code>ValidationMessage(FieldName, Message)</code> , pour que l'ensemble de règles soit testable et que le formulaire se contente d'associer les messages aux contrôles.
ErrorProvider	L'extender Wisej.NET qui affiche une icône et une info-bulle à côté du contrôle invalide via <code>SetError</code> , et qui peut aussi être lié aux erreurs de données de l'élément courant.
récapitulatif de validation	Un libellé ou un <code>AlertBox</code> qui liste les erreurs entre champs n'appartenant à aucun contrôle en particulier, comme un ticket fermé avec une échéance future.
DbUpdateException	L'exception levée par EF Core quand la base rejette la modification ; c'est le dernier filet de sécurité, et elle doit être traduite en message compréhensible, jamais affichée brute.

3. Le fil de la leçon

Trois niveaux, une seule autorité finale

Une application Wisej.NET et EF Core de production valide à trois endroits. Le niveau de l'interface intercepte les erreurs évidentes tant que l'utilisateur est encore sur l'écran : un titre manquant, une échéance hors de la plage autorisée, un client jamais sélectionné. Le niveau du domaine porte les règles métier qui doivent être vraies quel que soit l'écran, l'import ou la tâche en arrière-plan qui crée l'enregistrement. Le niveau de la base de données, avec ses colonnes NOT NULL, ses index uniques, ses clés étrangères et le jeton `RowVersion` du Module 2, fait autorité en dernier ressort pour la cohérence.

Ces niveaux ne sont pas interchangeables. Le retour de l'interface est une courtoisie qui accélère le cas courant ; la base de données reste le seul endroit où l'unicité ou l'intégrité référentielle peuvent être garanties face à des utilisateurs concurrents. L'éditeur du Support Desk valide donc avant d'enregistrer, tout en s'attendant à ce que la base dise non de temps en temps.

Annoter le modèle d'édition, puis le valider explicitement

Le Module 4 a introduit `TicketEditModel` comme l'objet auquel l'éditeur se lie. Ce module le décore avec `System.ComponentModel.DataAnnotations` : `[Required, StringLength(180)]` sur `Title`, `[StringLength(4000)]` sur `Description`, `[Required]` sur `Status`, `Priority`, `CustomerId` et `CategoryId`. Les clés étrangères sont volontairement des `int?` nullables : une `ComboBox` sans sélection

produit null, et c'est null que `[Required]` rejette, alors qu'un `int` non nullable passerait silencieusement avec la valeur zéro.

Les annotations décrivent des règles ; encore faut-il que quelque chose les exécute.

`Validator.TryValidateObject` avec `validateAllProperties: true` évalue chaque attribut et remplit une liste d'éléments `ValidationResult`. Le lab encapsule cet appel dans un service `TicketValidator` qui aplatit les résultats en records `ValidationMessage` et ajoute les règles que les annotations ne peuvent pas exprimer. La règle entre champs en est l'exemple type : un ticket fermé ne peut pas avoir une échéance future.

```
public IReadOnlyList<ValidationMessage> Validate(TicketEditModel model)
{
    var results = new List<ValidationResult>();
    var context = new ValidationContext(model);
    Validator.TryValidateObject(model, context, results, validateAllProperties: true);

    var messages = results
        .SelectMany(r => r.MemberNames.DefaultIfEmpty(null),
            (r, member) => new ValidationMessage(member, r.ErrorMessage ?? "Invalid
value."))
        .ToList();

    if (model.Status == "Closed" && model.DueDate?.Date > DateTime.Today)
        messages.Add(new ValidationMessage(nameof(model.DueDate), "Closed tickets cannot
have a future due date."));
    return messages;
}
```

Comme le validateur est un service simple sans référence à `Wisej.Web`, les cinq cas de test négatifs demandés par le lab s'exécutent dans le projet Tests sans formulaire, sans session et sans base de données.

Pourquoi les attributs de mappage ne sont pas un pipeline de validation

EF Core comprend lui aussi `[Required]` et `[MaxLength]`, mais sur une entité ce sont des instructions de mappage : elles décident si une colonne est nullable et quelle est sa largeur. EF Core n'effectue pas de passe complète de validation des DataAnnotations avant `SaveChangesAsync`. Un titre de 300 caractères sur une entité mappée à 180 n'est pas intercepté dans .NET ; il est envoyé à la base, qui le tronque ou lève une erreur selon le fournisseur. Gardez la forme de persistance dans la Fluent API de `SupportDeskContext`, gardez les règles de saisie sur le modèle d'édition, et appelez vous-même le validateur.

Le test pratique est simple : si une règle a besoin d'un message destiné à un humain, elle appartient au modèle d'édition et à `TicketValidator`. Si c'est un fait concernant la colonne, elle appartient à la configuration du modèle et à la migration.

ErrorProvider pour les champs, un récapitulatif pour les règles entre champs

L' `ErrorProvider` de Wisej.NET attache une icône rouge avec une info-bulle à n'importe quel contrôle. L'éditeur en garde une seule instance, créée pour le formulaire, et suit une séquence fixe dans `SaveAsync` : appeler `Clear()` pour faire disparaître les icônes de l'exécution précédente, appeler

`EndEdit` sur la `BindingSource` pour que le modèle contienne la saisie de l'utilisateur, exécuter `TicketValidator.Validate`, puis parcourir les messages. Un message dont le `FieldName` vaut `Title`, `CustomerId`, `CategoryId` ou `DueDate` devient `errorProvider.SetError(control, message)` sur la `TextBox`, la `ComboBox` ou le `DateTimePicker` correspondant. Le provider peut aussi être lié à l'élément courant d'une source de données, pour qu'un modèle qui signale ses propres erreurs marque automatiquement les contrôles, et l'extender Validation couvre les règles simples par contrôle sans code. Les règles entre champs n'ont pas de place unique ; chaque message est donc aussi regroupé dans un libellé récapitulatif, ou affiché dans un `AlertBox` quand le formulaire est chargé. Save reste désactivé tant que le modèle est invalide et est réactivé dans `finally`, et la méthode sort avant la création de tout `DbContext`. Si la validation échoue, EF Core n'est pas appelé.

DbUpdateException, le dernier filet de sécurité

Même un modèle entièrement validé peut être rejeté. L'exemple du lab est un numéro de ticket en double : deux agents créent des tickets à la même seconde, l'index unique sur `Number` se déclenche pour le second enregistrement, et EF Core lève `DbUpdateException`. La même exception couvre les violations de clé étrangère quand une catégorie a été supprimée pendant que l'éditeur était ouvert, ainsi que les erreurs de colonnes obligatoires dont l'interface n'avait pas connaissance.

Le gestionnaire fait trois choses. Il journalise l'exception complète, avec l'erreur interne du fournisseur, sur le serveur via le logger de l'application. Il montre à l'utilisateur une seule phrase simple, par exemple « The ticket could not be saved because the database rejected the change. Please verify required fields and lookup values. » Et quand une liste de référence peut être obsolète, il recharge les listes pour que l'utilisateur puisse choisir une valeur valide. Il n'affiche jamais l'instruction SQL, le nom de la contrainte ni quoi que ce soit qui pourrait contenir une chaîne de connexion. `DbUpdateConcurrencyException` dérive de `DbUpdateException` : interceptez-la donc en premier ; le Module 7 lui donne sa propre boîte de dialogue.



Erreurs courantes

Supposer qu'EF Core exécutera automatiquement toute la validation des DataAnnotations avant `SaveChanges` ; il ne fait que les mapper. Afficher aux utilisateurs du SQL brut ou des erreurs du fournisseur, ce qui divulgue des détails du schéma et parfois des informations de connexion. Placer des règles métier réutilisables uniquement dans les gestionnaires de boutons, où elles ne peuvent pas être testées et manquent silencieusement au prochain écran qui enregistre un ticket.



Réflexe de liaison de données

Séparez les deux durées de vie. Les formulaires Wisej.NET, les BindingSources et la ligne sélectionnée vivent pendant la session ; un DbContext vit le temps d'une opération. Créez-le depuis la fabrique, interrogez ou enregistrez, libérez-le, et ne le laissez jamais devenir un état de session.



Objectifs d'apprentissage

À la fin de ce module, vous saurez :

- ✓ Utiliser `System.ComponentModel.DataAnnotations` ou des validateurs personnalisés sur les modèles d'édition.
- ✓ Afficher les erreurs de validation avec `ErrorProvider`, l'étendre `Validation`, des libellés et des messages récapitulatifs.
- ✓ Comprendre que les attributs de mappage d'EF Core ne constituent pas un système complet de validation automatique de l'interface.
- ✓ Gérer les contraintes de base de données et `DbUpdateException` comme dernier filet de sécurité.

Guide du lab / de l'examen

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

🎯 Lab pratique

Ajoutez la validation à l'éditeur de tickets du Support Desk : des `DataAnnotations` sur `TicketEditModel`, un `TicketValidator` qui renvoie des erreurs par champ et un récapitulatif, un `ErrorProvider` effacé avant chaque exécution et renseigné pour `Title`, `Customer`, `Category` et `DueDate`, un libellé récapitulatif pour les règles entre champs, `Save` désactivé tant que le modèle est invalide, un gestionnaire de `DbUpdateException` qui affiche un message compréhensible pour un numéro de ticket en double, et au moins cinq cas de test négatifs.

Point de départ

Ouvrez la solution `SupportDesk` du Module 4 : le navigateur de tickets, les formulaires modaux `Add Ticket` et `Edit Ticket`, `TicketEditModel` et un `TicketEditorForm` avec une `BindingSource` et un `ErrorProvider` existent déjà, et `SaveAsync` contient toujours l'emplacement réservé de validation qui laisse passer n'importe quel modèle jusqu'à `SaveChangesAsync`. Le projet `Services` n'a pas encore de validateur et le projet `Tests` est vide.

Approche suggérée

- ✓ Décorez `TicketEditModel` dans le projet `Services` avec `[Required, StringLength(180)]` sur `Title`, `[StringLength(4000)]` sur `Description` et `[Required]` sur `Status`, `Priority`, `CustomerId` et `CategoryId` ; gardez les clés étrangères nullable pour qu'une liste de référence sans sélection soit détectée comme null.
- ✓ Ajoutez un record `ValidationMessage(FieldName, Message)` et un service `TicketValidator` dont la méthode `Validate` exécute `Validator.TryValidateObject` avec `validateAllProperties: true`, aplatit les résultats par nom de membre et ajoute la règle entre champs selon laquelle un ticket fermé ne peut pas avoir d'échéance future, avec un `FieldName` null ou `DueDate` comme champ ; enregistrez-le dans l'injection de dépendances de Microsoft à côté de la fabrique de Contexts.
- ✓ Dans `SaveAsync`, appelez `errorProvider.Clear()` et `editBindingSource.EndEdit()`, puis le validateur ; écrivez une méthode `ShowValidation` qui associe `Title`, `CustomerId`, `CategoryId` et `DueDate` à `SetError` sur la `TextBox`, la `ComboBox` et le `DateTimePicker` correspondants, regroupe tous les messages dans le libellé récapitulatif et sort avant la création de tout `DbContext`.
- ✓ Gardez `Save` désactivé tant que le modèle est invalide et pendant un enregistrement, et rétablissez-le dans `finally` ; ajoutez ensuite, sous le catch de concurrence, un `catch (DbUpdateException)` qui journalise l'exception complète côté serveur, affiche une seule phrase compréhensible via `AlertBox` et recharge les listes de référence quand une clé étrangère peut être obsolète.
- ✓ Dans le projet `Tests`, écrivez au moins cinq cas négatifs portant uniquement sur `TicketValidator` (titre vide, titre de plus de 180 caractères, client manquant, catégorie

manquante, ticket fermé avec une échéance future), puis reproduisez le numéro de ticket en double contre l'index unique et vérifiez que le message ne contient aucun SQL.

Livrables attendus

- ✓ DataAnnotations sur TicketEditModel et un TicketValidator qui renvoie des erreurs par champ et un récapitulatif
- ✓ ErrorProvider effacé avant chaque exécution et renseigné pour Title, Customer, Category et DueDate
- ✓ Libellé récapitulatif pour les règles entre champs et Save désactivé tant que le modèle est invalide
- ✓ DbUpdateException gérée avec un message compréhensible pour un numéro de ticket en double
- ✓ Au moins cinq cas de test négatifs

Questions de révision

- ✓ Si vous retiriez toutes les DataAnnotations de `TicketEditModel` en les gardant sur l'entité `Ticket`, lesquels de vos cinq tests négatifs échoueraient encore avant `SaveChangesAsync`, et pourquoi ?
- ✓ La règle des tickets fermés signale `DueDate` comme champ. Qu'est-ce qui changerait pour l'utilisateur si elle ne signalait aucun champ, et quand serait-ce le meilleur choix ?
- ✓ Deux agents créent un ticket à la même seconde et le second enregistrement lève `DbUpdateException`. Que voit le second agent, que consigne le serveur, et lequel des deux montreriez-vous à un client ?



Réflexe de liaison de données

Séparez les deux durées de vie. Les formulaires Wisej.NET, les BindingSources et la ligne sélectionnée vivent pendant la session ; un DbContext vit le temps d'une opération. Créez-le depuis la fabrique, interrogez ou enregistrez, libérez-le, et ne le laissez jamais devenir un état de session.