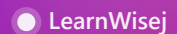


Validación enlazada a datos y errores del modelo

Guía de la lección · Guía del laboratorio / examen



Guía de la lección

Validar el modelo, no solo los controles: `BindingSource` y `EndEdit`, `DataAnnotations` ejecutadas con `Validator.TryValidateObject`, la asociación de `MemberNames` con los controles, `IDataErrorInfo` con un `ErrorProvider` enlazado a datos, `BindToDataAndErrors` y el orden seguro de guardado. Qué significa esto para un desarrollador que conoce C# básico y empieza a desarrollar formularios con Wisej.NET, y cómo leer este módulo.



Enfoque de producción

Validar el modelo, no solo los controles: `BindingSource` y `EndEdit`, `DataAnnotations` ejecutadas con `Validator.TryValidateObject`, la asociación de `MemberNames` con los controles, `IDataErrorInfo` con un `ErrorProvider` enlazado a datos, `BindToDataAndErrors` y el orden seguro de guardado.

1. El problema en producción

Todo lo que `ValidationClinic` comprueba hasta ahora vive en el formulario de alta: un controlador `Validating` aquí, una regla del extensor `Validation` allá, una llamada a `ContactValidator` en `btnSave_Click`. Pero un contacto no solo llega a través de ese formulario. Puede editarse en la cuadrícula de contactos, importarse desde un archivo o crearse desde un servicio, y ninguno de esos caminos ejecuta los controladores del formulario. Este módulo traslada las reglas básicas al propio `ContactEditModel`, para que los datos lleven su propia definición de «válido» y todas las pantallas puedan hacerles la misma pregunta.

2. Conceptos clave

Concepto	Por qué importa
BindingSource	Un componente que se sitúa entre los controles y tu objeto: su <code>DataSource</code> es el modelo, y cada control se enlaza a una propiedad a través de él en lugar de copiarse a mano.
EndEdit()	Lleva al modelo el valor que el usuario todavía está editando en el control, para que la validación lea lo que hay en pantalla y no el último valor confirmado.
DataAnnotations	Atributos como <code>[Required]</code> , <code>[StringLength]</code> , <code>[EmailAddress]</code> y <code>[Range]</code> colocados en las propiedades del modelo, cada uno con un <code>ErrorMessage</code> escrito para el usuario.
Validator.TryValidateObject	El método de .NET que ejecuta esos atributos sobre un objeto y rellena una lista de objetos <code>ValidationResult</code> ; no sabe nada de controles ni de Wisej.NET.
MemberNames	Los nombres de las propiedades a las que pertenece un <code>ValidationResult</code> , que son el único vínculo entre un mensaje de error y el control que debe mostrarlo.
IDataErrorInfo	Una interfaz que permite a un modelo responder «¿qué le pasa a esta propiedad?» mediante un índice, para que la capa de enlace pida los errores en lugar de que tu código los empuje.
ErrorProvider enlazado a datos	Un <code>ErrorProvider</code> cuyo <code>DataSource</code> es el mismo origen de enlace que el de los controles, de modo que muestra los errores del modelo junto a los controles enlazados sin ninguna llamada a <code>SetError</code> .

3. Desarrollo de la lección

Enlaza el formulario y después valida el modelo

El enlace a datos conecta una propiedad de un control con una propiedad del modelo. Crea un `BindingSource`, establece su `DataSource` en un `ContactEditModel` y llama a `txtName.DataBindings.Add("Text", contactBindingSource, nameof(ContactEditModel.Name))` para cada campo. A partir de ese momento, el formulario ya no copia texto en un objeto al hacer clic en Save; el modelo son los datos del formulario, y `contactBindingSource.Current` te lo devuelve.

Hay un hueco en el que casi todos los principiantes caen. El enlace no actualiza el modelo con cada pulsación de tecla. Si el usuario escribe un correo electrónico y hace clic en Save enseguida, el texto nuevo puede seguir en el TextBox mientras `Email` en el modelo conserva el valor antiguo. Validar el modelo en ese momento comprueba datos que el usuario ya ha sustituido. La solución es una sola llamada: `contactBindingSource.EndEdit()` termina la edición actual y lleva los valores escritos al modelo. Haz que sea la primera línea de cada camino de guardado, antes de que cualquier regla lea el modelo.

DataAnnotations y Validator.TryValidateObject

Las DataAnnotations colocan reglas sencillas justo al lado de los datos que protegen. En `ContactEditModel`, `Name` recibe `[Required]` y `[StringLength(80)]`, `Email` recibe `[Required]` y

`[EmailAddress]`, y `Age` recibe `[Range(0, 120)]`. Cada atributo lleva un `ErrorMessage` como «Enter a valid email address.». Trata esos mensajes como texto del producto que leerá el usuario, no como notas para desarrolladores.

Los atributos no hacen nada por sí solos; algo tiene que ejecutarlos. Crea una `List<ValidationResult>` y un `ValidationContext` para el modelo, y después llama a `Validator.TryValidateObject(model, context, results, validateAllProperties: true)`. Devuelve `false` cuando algo ha fallado y añade un resultado por cada regla incumplida. No omitas el último argumento: sin `validateAllProperties: true` solo se comprueba `[Required]`, y un correo electrónico mal formado o una edad de 300 pasan sin avisar. Ten en cuenta también que `[EmailAddress]` y `[Range]` consideran válido un valor null, y precisamente por eso la regla de obligatoriedad es un atributo aparte.

IDataErrorInfo y el ErrorProvider enlazado a datos

`IDataErrorInfo` invierte la relación. En lugar de que tu código ejecute un validador y empuje mensajes a los controles, el modelo responde a preguntas. Tiene dos miembros: `Error`, un mensaje sobre el objeto en su conjunto (a menudo una cadena vacía), y el indizador `this[string columnName]`, que devuelve el mensaje de una propiedad o una cadena vacía cuando esa propiedad está bien. En el segundo modelo del laboratorio, el indizador devuelve «Name is required.» cuando `columnName` es `Name` y el nombre está vacío.

La interfaz resulta útil cuando un `ErrorProvider` está conectado a los mismos datos. Establece `errorProvider.ContainerControl` en el formulario y `errorProvider.DataSource` en `contactBindingSource`. El proveedor pregunta entonces al elemento actual por el error de cada propiedad enlazada y muestra cada mensaje junto al control enlazado a ella. No aparece ninguna llamada a `SetError` en tu código. .NET también tiene `INotifyDataErrorInfo`, que puede notificar varios errores por propiedad y cambiarlos de forma asíncrona, pero para un formulario de principiante `IDataErrorInfo` es el punto de partida más sencillo y mejor soportado.

BindToDataAndErrors en tiempo de ejecución

Un proveedor enlazado a datos necesita que dos ajustes coincidan: `DataSource`, el objeto o la lista, y `DataMember`, la ruta dentro de él. En el diseñador estableces ambos una vez y no vuelves a tocarlos. En tiempo de ejecución, por ejemplo cuando `ValidationClinic` cambia el editor del modelo con anotaciones al modelo `IDataErrorInfo`, establecerlos uno tras otro deja un momento en el que el origen nuevo está emparejado con el miembro antiguo, y el proveedor puede intentar enlazarse a algo que no existe.

`BindToDataAndErrors(source, member)` establece ambos en una sola llamada, así que el proveedor solo ve parejas que coinciden. Cuando el objeto completo son los datos, pasa `string.Empty` como miembro: `errorProvider.BindToDataAndErrors(contactBindingSource, string.Empty)`. Usa las dos propiedades en el diseñador, y el método siempre que el origen cambie con el formulario abierto.

Asocia los errores del modelo a los controles y al resumen

`Validator.TryValidateObject` no sabe nada de Wisej.NET a propósito, así que el formulario debe traducir sus resultados. Construye un mapa de campos en el constructor, un `Dictionary<string, Control>` de `nameof(ContactEditModel.Email)` a `txtEmail`, etc., y después recorre los resultados: para cada nombre de `result.MemberNames`, busca el control y llama a `errorProvider.SetError(control, result.ErrorMessage)`. Por último, une todos los mensajes en `validationSummaryLabel` y muéstrala,

porque un icono solo pasa desapercibido con facilidad. Un resultado cuyo miembro no está en el mapa sigue llegando al resumen, así que nunca se pierde ningún error. El mapa tiene la misma forma que `ApplyModelValidation` del Módulo 4, y por eso ambos pueden compartir un mismo resumen.

Con todas las capas en su sitio, el controlador de Save sigue un orden fijo: terminar la edición, validar los controles hijos, validar el modelo, ejecutar las reglas entre campos, guardar solo si nada ha fallado y capturar aparte las excepciones del sistema para que una caída de la base de datos nunca parezca un error de escritura.

```
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
}
```



Regla de las capas de validación

Nombra la capa antes de escribir la comprobación. La validación de campo protege la edición y se ejecuta cuando el usuario sale de un control; la validación del modelo protege los datos y ejecuta las mismas reglas desde el formulario, la cuadrícula y cualquier servicio; la validación al guardar protege la base de datos y vuelve a ejecutar todas las capas antes de escribir nada. Muestra cada error cerca de su campo y en un resumen visible, con un lenguaje claro que diga qué hacer a continuación, y mantén los errores que el usuario puede corregir separados de los fallos del sistema.



Objetivos de aprendizaje

Al final de este módulo serás capaz de:

- ✓ Enlazar controles a un modelo y validar el modelo antes de guardar.
- ✓ Usar `DataAnnotations` con `Validator.TryValidateObject`.
- ✓ Comprender `IDataErrorInfo` y el comportamiento de un `ErrorProvider` enlazado a datos.
- ✓ Usar `BindToDataAndErrors` al cambiar el origen de datos de un proveedor en tiempo de ejecución.
- ✓ Asociar los errores del modelo a los controles y resúmenes de Wisej.NET.

Guía del laboratorio / examen

Qué vas a construir en el laboratorio práctico, el enfoque sugerido y los entregables requeridos.



Laboratorio práctico

Enlaza el formulario de alta de ValidationClinic a un ContactEditModel mediante un contactBindingSource y añade las anotaciones Required, StringLength, EmailAddress y Range a sus propiedades Name, Email y Age. En btnSave_Click llama primero a contactBindingSource.EndEdit(), ejecuta Validator.TryValidateObject con validateAllProperties establecido en true y asocia el nombre de miembro de cada ValidationResult a su control mediante un mapa de campos y el resumen de validación. Después implementa IDataErrorInfo en un segundo modelo, dirige el ErrorProvider al origen de enlace con BindToDataAndErrors y muestra cómo aparecen los mismos errores a través del enlace a datos sin ninguna llamada a SetLastError escrita a mano.

Punto de partida

ValidationClinic después del Módulo 4: el formulario de alta tiene reglas de campo, un servicio `ContactValidator`, `ApplyModelValidation` y una `MinimumAgeValidationRule`, pero sus controles no están enlazados a nada, así que `btnSave_Click` sigue leyendo los valores directamente de `txtName`, `txtEmail` y `txtAge`.

Enfoque sugerido

- ✓ Crea `ContactEditModel` con `Name`, `Email` y un `Age` que admita null, añade las cuatro anotaciones con textos `ErrorMessage` claros y enlaza los tres TextBox al modelo con `DataBindings.Add("Text", contactBindingSource, ...)`.
- ✓ Antes de escribir ninguna validación, demuestra el hueco de la edición: escribe un correo electrónico, haz clic en Save y registra `((ContactEditModel)contactBindingSource.Current).Email` con y sin una llamada a `contactBindingSource.EndEdit()` delante.
- ✓ Escribe `ValidateModel` alrededor de `Validator.TryValidateObject(model, context, results, validateAllProperties: true)`; después construye `_fieldMap` en el constructor y un método `ApplyAnnotationErrors` que establezca cada error de control y rellene `validationSummaryLabel`.
- ✓ Reescribe `btnSave_Click` en el orden seguro: `EndEdit`, `ValidateChildren`, anotaciones, el `ContactValidator` del Módulo 4, guardado y un `catch` aparte que muestre un error del sistema comprensible.
- ✓ Añade un segundo modelo que implemente `IDataErrorInfo`, cambia el origen de enlace a este modelo en tiempo de ejecución, llama a `errorProvider.BindToDataAndErrors(contactBindingSource, string.Empty)` y vacía el nombre para ver cómo aparece el error sin ningún `SetError`.

Entregables requeridos

- ✓ ContactEditModel con DataAnnotations enlazado al formulario mediante un BindingSource

- ✓ Controlador de Save que llama a EndEdit antes de ejecutar Validator.TryValidateObject
- ✓ Mapa de campos que convierte los nombres de miembro de ValidationResult en errores de control y líneas del resumen
- ✓ Modelo IDataErrorInfo cuyos errores aparecen a través de un ErrorProvider enlazado a datos
- ✓ Llamada a BindToDataAndErrors usada cuando el origen de datos del proveedor cambia en tiempo de ejecución

Preguntas de repaso

- ✓ Un tester escribe un nuevo correo electrónico y pulsa Intro para guardar, y lo que se comprueba y almacena es el correo electrónico antiguo y válido. ¿Qué línea falta y por qué el formulario parece correcto en pantalla mientras el modelo está mal?
- ✓ Quitas `validateAllProperties: true` y todos los guardados siguen «funcionando». ¿Cuáles de tus cuatro anotaciones han dejado de ejecutarse y qué datos incorrectos pueden llegar ahora al repositorio?
- ✓ El servicio de importación crea contactos sin abrir el formulario. ¿Qué reglas de este módulo siguen protegiendo ese camino y cuáles solo se ejecutan porque un usuario ha hecho clic en Save?



Regla de las capas de validación

Nombra la capa antes de escribir la comprobación. La validación de campo protege la edición y se ejecuta cuando el usuario sale de un control; la validación del modelo protege los datos y ejecuta las mismas reglas desde el formulario, la cuadrícula y cualquier servicio; la validación al guardar protege la base de datos y vuelve a ejecutar todas las capas antes de escribir nada. Muestra cada error cerca de su campo y en un resumen visible, con un lenguaje claro que diga qué hacer a continuación, y mantén los errores que el usuario puede corregir separados de los fallos del sistema.