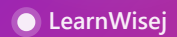


UX de validación en producción, proveedores de errores personalizados y proyecto final

Guía de la lección · Guía del laboratorio / examen



Guía de la lección

Unir todas las capas: un flujo de Save sencillo y repetible, un IErrorProvider personalizado que alimenta un panel de resumen, proveedores de iconos y de resumen trabajando en paralelo, diez reglas de UX de validación y el proyecto final ValidationClinic, que demuestra dónde debe ir cada comprobación. 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

Unir todas las capas: un flujo de Save sencillo y repetible, un IErrorProvider personalizado que alimenta un panel de resumen, proveedores de iconos y de resumen trabajando en paralelo, diez reglas de UX de validación y el proyecto final ValidationClinic, que demuestra dónde debe ir cada comprobación.

1. El problema en producción

Un usuario que rellena el formulario de alta de ValidationClinic nunca pregunta qué API ha encontrado un error; solo se fija en si la pantalla le dice qué está mal y le deja terminar. Después de seis módulos, la aplicación tiene controladores de campo, reglas del extensor, un servicio validador, modelos con anotaciones y reglas de cuadrícula, pero cada elemento se añadió por separado y todavía nada garantiza que todos se ejecuten, en el orden correcto, cada vez que se hace clic en Save. Este módulo une las capas en un flujo de Save predecible y en un único resumen visible, y después te pide demostrar el diseño en un proyecto final.

2. Conceptos clave

Concepto	Por qué importa
Flujo de Save	Una secuencia fija dentro de <code>btnSave_Click</code> que confirma las ediciones pendientes, ejecuta por turnos cada capa de validación y sale en la primera capa que falla, para que nada llegue al repositorio sin comprobar.
Salida anticipada	Cada etapa termina el controlador con <code>return</code> cuando encuentra errores, así que las etapas posteriores nunca se ejecutan sobre datos que una etapa anterior ya ha rechazado.
<code>IErrorProvider</code>	La pequeña interfaz a la que el sistema de validación de Wisej.NET envía los mensajes: <code>SetError(Control, string)</code> para establecer o borrar un mensaje y <code>GetError(Control)</code> para volver a leerlo.
<code>SummaryErrorProvider</code>	Tu propio <code>IErrorProvider</code> , que guarda un diccionario de pares control-mensaje y reescribe a partir de él una etiqueta de resumen cada vez que se establece o se borra un error.
Proveedores dobles	Un único método auxiliar <code>SetError</code> que escribe el mismo mensaje en el <code>ErrorProvider</code> integrado (el icono junto al campo) y en el proveedor de resumen (la lista de la parte superior).
Error del usuario frente a fallo del sistema	Un error que el usuario puede corregir, como un nombre duplicado, se convierte en un mensaje de campo; una excepción inesperada se registra para los desarrolladores y se muestra como un mensaje genérico y sereno.
Prueba de límites	Un valor de prueba situado justo en el borde de una regla, como una edad de 0 o 120, que detecta los errores por uno que los valores válidos y no válidos corrientes no detectan.

3. Desarrollo de la lección

Un botón Save deliberadamente aburrido

Un buen controlador de Save hace los mismos pasos en el mismo orden en cada clic y se detiene en cuanto algo va mal. En ValidationClinic el orden es: borrar el resumen, llamar a `bindingSource.EndEdit()` para que el valor que se sigue escribiendo llegue al modelo, llamar a `ValidateChildren()` para que se ejecuten las reglas de campo de todos los controles, ejecutar la comprobación de DataAnnotations con `Validator.TryValidateObject`, ejecutar `ContactValidator.Validate` para las reglas de negocio, comprobar que ninguna fila de la cuadrícula tiene todavía un error y, solo entonces, guardar. `EndEdit` va primero porque, de lo contrario, todas las comprobaciones del modelo posteriores examinarían datos obsoletos.

```

ResetSummary();
bindingSource.EndEdit();
if (!this.ValidateChildren()) return;

var model = (ContactEditModel)bindingSource.Current;
if (!ApplyAnnotationErrors(model)) return;

var businessErrors = contactValidator.Validate(model);
if (businessErrors.Count > 0) { ApplyBusinessErrors(businessErrors); return; }

if (GridHasErrors()) return;

try { SaveContact(model); MessageBox.Show("Saved.", MessageBoxIcon.Information); }
catch (DuplicateNameException) { SetError(txtName, "A contact with this name already
exists."); }

```

Los nombres de los métodos auxiliares (`ResetSummary` , `ApplyAnnotationErrors` , `GridHasErrors`) son pequeños métodos tuyos, no miembros del framework. El guardado en sí está dentro de `try / catch` , y el `catch (Exception ex)` final (no se muestra) escribe la excepción con `System.Diagnostics.Debug.WriteLine` y muestra un texto comprensible con `MessageBox.Show` . El flujo es aburrido a propósito: cuando un tester informa de que «Save ha dejado pasar un correo electrónico incorrecto», puedes señalar la etapa exacta que debería haberlo detenido.

Un panel de resumen mediante `IErrorProvider`

Al sistema de validación de Wisej.NET no le importa qué muestra sus mensajes, solo que el destino implemente `IErrorProvider` . El `ErrorProvider` integrado es una implementación y una `Label` puede ser otra. Tu `SummaryErrorProvider` es una tercera: su constructor recibe la etiqueta de resumen, `SetError` elimina la entrada del control cuando el mensaje está vacío o la guarda en caso contrario, después une los mensajes distintos en el `Text` de la etiqueta y establece `Visible` solo mientras exista al menos un error. `GetError` devuelve el mensaje guardado o una cadena vacía.

Lo conectas con dos líneas: crea el proveedor alrededor de `validationSummaryLabel` y después asigna `validation.ErrorProvider = summaryProvider` . A partir de ese momento, cada regla que ejecuta el extensor Validation, incluida tu `MinimumAgeValidationRule` del Módulo 4, notifica en el panel sin ningún código adicional en la propia regla.

Errores de campo, de cuadrícula y del modelo en un solo lugar

Ahora los errores llegan desde tres direcciones. Los errores de campo proceden de los controladores `Validating` y de las reglas del extensor. Los errores del modelo proceden de los nombres de miembro de `ValidationResult` y de los mensajes de `ContactValidator` , que asocias de nuevo a los controles con el mapa de campos del Módulo 5. Los errores de la cuadrícula viven en celdas y filas como `ErrorText` , establecidos en `CellValidating` , `ValidateRow` y `DataError` . El usuario no debería tener que saber cuál de ellos ha producido un mensaje.

La forma de conseguirlo es encaminar los errores del formulario y del modelo a través de un único método auxiliar `SetError(Control, string)` en el formulario, que llama a `errorProvider.SetError` para el icono y a `summaryProvider.SetError` para la lista. Los errores de la cuadrícula se quedan en la cuadrícula,

donde el usuario ve la celda en rojo, y la etapa de la cuadrícula del flujo añade una línea al resumen, como «Row 3: Closed contacts require a closed date.», para que el problema también sea visible en la parte superior. Borrar funciona igual a la inversa: un mensaje vacío elimina a la vez el icono y la línea del resumen.

Diez reglas para una validación que la gente pueda usar

La mayor parte de una buena UX de validación es disciplina, no código. Las diez reglas siguientes, agrupadas en seis líneas, son la lista de comprobación con la que se revisa el proyecto final:

- Coloca cada mensaje junto a su campo y repítelo en un resumen de todo el formulario.
- Escribe con un lenguaje claro que diga qué hacer a continuación, y nunca culpes al usuario.
- Nunca muestres trazas de pila, texto SQL ni excepciones de conversión.
- Nunca guardes datos no válidos y nunca atrapes al usuario: no bloques Cancel ni Close sin un motivo real.
- Mantén las comprobaciones lentas o remotas fuera de la validación en cada pulsación de tecla; ejecútalas al guardar.
- Ejecuta las mismas reglas del modelo desde el formulario, la cuadrícula y cualquier servicio, y prueba valores obligatorios, no válidos, válidos y límite.

Un panel de resumen también importa para la accesibilidad: un error que solo aparece cuando alguien pasa el ratón por un icono es invisible para los usuarios de teclado y fácil de pasar por alto para todos. Un mensaje al que un lector de pantalla puede llegar como texto forma parte del diseño, no es decoración.

El proyecto final ValidationClinic

El proyecto final no es una aplicación grande. Es la prueba de que sabes dónde debe ir cada comprobación. ValidationClinic debe tener un nombre y un correo electrónico obligatorios, un `ErrorProvider` con un resumen visible, el extensor Validation con al menos tres reglas integradas, una clase `ValidationRule` personalizada, un servicio validador usado por Save, validación del modelo con DataAnnotations o `IDataErrorInfo`, validación de celdas de correo electrónico y edad más un controlador `DataError` en la cuadrícula, el resumen con `IErrorProvider` personalizado y un breve README que explique el diseño.

Después lo demuestras con un guion breve y fijo: sal de un campo no válido, haz clic en Save y observa cómo `ValidateChildren` lo detiene, muestra un error del modelo que solo aparece después de `EndEdit`, escribe un correo electrónico incorrecto en una celda de la cuadrícula, señala el panel de resumen que lo enumera todo y, por último, corrige cada problema y guarda correctamente. Escribe primero la tabla de pruebas del README, con valores obligatorios, no válidos, válidos y límite para cada regla, y la demostración se convierte en ejecutar tus propias pruebas en voz alta.



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:

- ✓ Construir un flujo de Save completo para un formulario Wisej.NET de nivel principiante.
- ✓ Crear un IErrorProvider personalizado para un panel de resumen.
- ✓ Combinar errores de campo, errores de cuadrícula y errores del modelo.
- ✓ Diseñar una UX de validación clara, accesible y comprobable.
- ✓ Completar la aplicación del proyecto final del curso.

Guía del laboratorio / examen

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



Laboratorio práctico

Completa el proyecto final de ValidationClinic. Escribe `btnSave_Click` como un flujo fijo: restablece el resumen, llama a `bindingSource.EndEdit()`, detente si `ValidateChildren()` falla, detente si falla la comprobación de `DataAnnotations`, detente si `ContactValidator` devuelve errores de negocio, detente si la cuadrícula de contactos todavía tiene errores de fila y solo entonces guarda dentro de un `try/catch` que convierta un nombre duplicado en un error de campo y cualquier otra excepción en un mensaje comprensible. Crea `SummaryErrorProvider` implementando `IErrorProvider`, asígnalo a `validation.ErrorProvider` y encamina los errores a través de un método auxiliar `SetError` que escriba tanto en el `ErrorProvider` de iconos como en el proveedor de resumen. Prueba valores obligatorios, no válidos, válidos y límite, y escribe un breve README que explique el diseño de la validación.

Punto de partida

ValidationClinic tal como lo dejaste después del Módulo 6: el formulario de alta tiene controladores de campo, reglas del extensor, `MinimumAgeValidationRule`, un servicio `ContactValidator` y un `ContactEditModel` enlazado mediante un `BindingSource`, y la cuadrícula de contactos valida sus celdas de correo electrónico, edad y fecha de cierre. Cada capa funciona por separado, pero `Save` todavía no las ejecuta todas en orden, y los errores aparecen en lugares distintos según la capa que los haya detectado.

Enfoque sugerido

- ✓ Escribe primero `SummaryErrorProvider`: una clase que implemente `IErrorProvider`, reciba la `Label` del resumen en su constructor, guarde pares control-mensaje, elimine una entrada cuando el mensaje esté vacío y reescriba el texto y la visibilidad de la etiqueta en cada `SetError`. Implementa `GetError` para que devuelva una cadena vacía en un control sin error.
- ✓ Crea el proveedor alrededor de `validationSummaryLabel`, asigna `validation.ErrorProvider = summaryProvider` y añade un método auxiliar `SetError(Control, string)` a nivel de formulario que llame a `errorProvider.SetError` y a `summaryProvider.SetError` con el mismo mensaje. Sustituye por el método auxiliar las llamadas directas a `SetError` de tu código de asociación del modelo y de negocio.
- ✓ Reescribe `btnSave_Click` como el flujo fijo: restablece el resumen, `bindingSource.EndEdit()`, `ValidateChildren()`, la comprobación `Validator.TryValidateObject`, `ContactValidator.Validate` y, después, una comprobación de la cuadrícula que omita la fila nueva de marcador de posición e informe de cualquier fila que todavía tenga `ErrorText`. Cada etapa sale en cuanto falla.
- ✓ Envuelve solo el guardado en `try / catch`: captura primero `DuplicateNameException` y conviértela en un error en `txtName` mediante tu método auxiliar; después captura `Exception`, regístrala con `System.Diagnostics.Debug.WriteLine` y muestra un texto comprensible con `MessageBox.Show`. Si todo va bien, llama a `AlertBox.Show("Saved.", MessageBoxIcon.Information)`.

- ✓ Escribe el README: un breve párrafo por capa (de campo, del modelo y al guardar), una tabla de casos de prueba con valores obligatorios, no válidos, válidos y límite para cada regla, y la lista de comprobación del proyecto final. Después ensaya el guion de la demostración hasta que puedas ejecutarlo en cinco minutos.

El proyecto final se revisa frente a diez funcionalidades obligatorias: nombre y correo electrónico del cliente obligatorios; un `ErrorProvider` con un resumen visible; el extensor Validation con al menos tres reglas integradas; una clase `ValidationRule` personalizada; un servicio validador usado por Save; validación del modelo con DataAnnotations o `IDataErrorInfo`; validación de celdas de correo electrónico y edad en la cuadrícula; un controlador `DataError` en la cuadrícula; el resumen con `IErrorProvider` personalizado; y un breve README. La demostración sigue un orden fijo: muestra un campo no válido, muestra Save detenido por `ValidateChildren`, muestra un error del modelo que aparece después de `EndEdit`, muestra la validación de celdas de la cuadrícula, muestra el proveedor de resumen enumerándolo todo y, por último, corrige los datos y guarda correctamente. Antes de entregar, comprueba tus pantallas frente a las diez reglas de UX de la lección: mensajes junto al campo y en un resumen, lenguaje claro que diga qué hacer, sin culpar al usuario, sin trazas de pila, sin guardados no válidos, Cancel y Close nunca bloqueados sin motivo, sin comprobaciones remotas en cada pulsación de tecla, las mismas reglas del modelo en todas partes y los cuatro tipos de valores de prueba.

Entregables requeridos

- ✓ Flujo de Save que ejecuta EndEdit, ValidateChildren y las comprobaciones del modelo, de negocio y de la cuadrícula antes de guardar
- ✓ Clase SummaryErrorProvider que implementa SetError y GetError de IErrorProvider
- ✓ Método auxiliar SetError que escribe tanto en el ErrorProvider de iconos como en el proveedor de resumen
- ✓ try/catch alrededor del guardado que separa un error de nombre duplicado de los fallos del sistema
- ✓ README con el diseño de la validación y casos de prueba para valores obligatorios, no válidos, válidos y límite

Preguntas de repaso

- ✓ Un tester escribe un nuevo correo electrónico en el formulario y hace clic en Save inmediatamente; se guarda el correo electrónico antiguo. ¿Qué etapa del flujo falta o está mal colocada, y por qué importa su posición?
- ✓ Tu `SummaryErrorProvider` muestra «Email is required.» dos veces después de que el usuario corrija el campo y lo vuelva a estropear. ¿Qué parte de `SetError` debería evitarlo y de qué depende?
- ✓ El repositorio lanza una excepción con un mensaje de la base de datos cuando se cae la red. Describe paso a paso qué ve el usuario y qué puede encontrar después el desarrollador, y explica por qué esto es distinto del caso del nombre duplicado.



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.