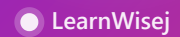


Validación programática y avisos con ErrorProvider

Guía de la lección · Guía del laboratorio / examen



Guía de la lección

Un ErrorProvider por formulario, SetLastError y el borrado de errores por control, InvalidMessage para avisos locales sencillos, un pequeño tipo de resultado FieldValidation que mantiene las reglas fuera de la interfaz y un resumen de validación visible para que ningún error dependa de pasar el ratón sobre un icono. 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

Un ErrorProvider por formulario, SetLastError y el borrado de errores por control, InvalidMessage para avisos locales sencillos, un pequeño tipo de resultado FieldValidation que mantiene las reglas fuera de la interfaz y un resumen de validación visible para que ningún error dependa de pasar el ratón sobre un icono.

1. El problema en producción

El Módulo 1 dio al formulario de alta de ValidationClinic dos controladores `Validating`, y eso funciona con dos campos. Un formulario de contacto real tiene un nombre, un correo electrónico, un número de teléfono, un código de cliente y más, y si cada campo tiene su propio controlador escrito a mano con su propia forma de mostrar un mensaje, el formulario se vuelve incoherente enseguida: un campo muestra un icono, otro no muestra nada y un tercero deja un error antiguo en pantalla después de que el usuario lo haya corregido. Este módulo sustituye esa deriva por un único patrón: pequeños métodos auxiliares deciden qué está mal, un método muestra el resultado y un resumen visible indica a cada usuario qué debe corregir sin obligarle a pasar el ratón por encima de nada.

2. Conceptos clave

Concepto	Por qué importa
<code>ErrorProvider</code>	Un componente extensor que dibuja un icono de error junto a cualquier control al que le des un mensaje y muestra ese mensaje como tooltip; una sola instancia sirve para todo el formulario.
<code>ContainerControl</code>	El formulario o control de usuario al que pertenece el proveedor; establecerlo en <code>this</code> indica al proveedor sobre qué superficie se dibujan sus iconos.
<code>setError(control, message)</code>	Muestra <code>message</code> junto a <code>control</code> ; llamarlo con una cadena vacía o con <code>null</code> elimina el error solo de ese control.
<code>GetError</code> , <code>HasErrors</code> , <code>Clear()</code>	Leen el mensaje actual de un control, preguntan si queda algún error en el formulario o eliminan todos los errores a la vez cuando se restablece todo el formulario.
<code>InvalidMessage</code>	Una propiedad de los controles de edición que admiten la interfaz <code>IValidation</code> ; el propio control muestra el mensaje como tooltip de valor no válido, sin que intervenga ningún proveedor aparte.
record <code>FieldValidation</code>	Un tipo de resultado mínimo que contiene el control, un nombre de campo y un mensaje, donde un mensaje vacío significa válido; permite que las reglas devuelvan respuestas en lugar de cambiar la pantalla.
<code>validationSummaryLabel</code>	Una Label sencilla que enumera como texto todos los problemas actuales, visible solo mientras haya errores, para que el aviso nunca dependa de ver un icono o pasar el ratón por encima.

3. Desarrollo de la lección

Mostrar y borrar errores de campo con ErrorProvider

Crea un `ErrorProvider` para el formulario, ya sea arrastrándolo desde el Cuadro de herramientas al diseñador o como un campo creado en código, y establece su `ContainerControl` en el formulario. Dos ajustes opcionales lo hacen más tranquilo para los usuarios: `BlinkStyle = ErrorBlinkStyle.NeverBlink` evita que el icono parpadee, y `SetIconAlignment(txtEmail, ErrorIconAlignment.MiddleRight)` junto con `SetIconPadding(txtEmail, 4)` coloca el icono justo a la derecha del campo, donde la vista lo espera.

Mostrar un error es una sola llamada: `errorProvider.SetError(txtName, "Name is required.")` . Recuerda que Wisej.NET ejecuta esto en el servidor; el navegador simplemente recibe el icono y el tooltip actualizados. Borrarlo es la misma llamada con una cadena vacía. Ese detalle importa más de lo que parece. Cuando el usuario corrige el nombre, borras el error de `txtName` y nada más. `Clear()` elimina todos los errores del formulario, así que resérvalo para los momentos en que todo el formulario vuelve a empezar de verdad: cargar otro contacto, restablecer el formulario o iniciar una pasada de validación completa que vuelve a aplicar de inmediato todos los errores actuales. `GetError(control)` te dice qué

muestra en ese momento un control concreto, y `HasErrors` es una comprobación final rápida antes de que Save continúe.

¿InvalidMessage o SetError?

Wisej.NET te ofrece una segunda forma, más ligera, de dar avisos. Los controles de edición que implementan `IValidation` tienen una propiedad `InvalidMessage`. Establécela dentro del controlador `Validating` del control cuando el valor sea incorrecto, vuelve a dejarla como cadena vacía cuando el valor sea correcto, y el control mostrará el mensaje en su propio tooltip de valor no válido. Sin proveedor, sin colocar iconos, sin nada compartido.

Eso hace que `InvalidMessage` encaje bien en un formulario muy sencillo o en un campo cuyo aviso deba quedarse en local. `SetError` encaja mejor cuando un formulario tiene varios campos, porque así todos los errores se ven igual, se pueden volver a leer con `GetError` y se pueden contar con `HasErrors`. En el laboratorio usas ambos a propósito: el campo del código de cliente usa `InvalidMessage`, todos los demás campos pasan por el `ErrorProvider`, y escribes una breve nota comparando lo que ve el usuario.

Una etiqueta de resumen que todos pueden leer

Un icono de error es pequeño, es solo un color y una forma, y su texto solo aparece cuando el puntero se detiene sobre él. Un usuario de teclado, un usuario con pantalla táctil o un usuario con baja visión pueden pasarlo por alto por completo. Por eso cada formulario de este curso muestra también sus errores como texto. La versión más sencilla es una Label llamada `validationSummaryLabel` colocada encima del botón Save, con una línea por problema, como *Email: Enter a valid email address*.

Dos reglas mantienen el resumen fiable. Solo está visible mientras exista al menos un error, así que nunca se queda un recuadro rojo vacío. Y se reconstruye a partir de los mismos resultados que establecen los iconos, así que los iconos y el resumen nunca pueden contradecirse. Un resumen visible también hace que el formulario sea más fácil de probar: puedes comprobar el texto de la etiqueta en lugar de pasar el ratón por los iconos.

Un método auxiliar para varios campos

El paso clave de este módulo es separar decidir de mostrar. Cada regla se convierte en un método pequeño, `ValidateName`, `ValidateEmail` o `ValidatePhone`, que lee un valor y devuelve un record `FieldValidation`. Nunca llama a `SetError` y nunca toca una Label. Después, un único método `Apply` recibe todos los resultados y hace el trabajo de visualización en un solo lugar.

```

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);
}

```

Fíjate en que `Apply` llama a `SetError` para cada resultado, sea válido o no. Un resultado válido tiene un mensaje vacío, así que pasarlo borra el icono de ese control. Así es exactamente como corregir un campo: borra solo el error de ese campo: los demás resultados siguen llevando sus mensajes, así que sus iconos se mantienen. El botón Save llama a `Apply(ValidateName(), ValidateEmail(), ValidatePhone())` y sale antes si recibe `false`.

Avisos de campo frente a avisos de reglas de negocio

No todos los mensajes van junto a un campo. Los *avisos de campo* tratan del valor de un control: está vacío, no tiene @, el número de teléfono es demasiado corto. El usuario puede corregirlo sin salir de ese control, así que el aviso va junto al control y en el resumen. Los *avisos de reglas de negocio* tratan de lo que significan los datos: ya existe un contacto con este nombre, o dos fechas no encajan entre sí. Esas reglas suelen necesitar otros campos o datos almacenados, y normalmente se ejecutan al guardar y no cuando el usuario sale de un campo con Tab.

Mantén separados ambos tipos en tu código y en tu redacción. Los métodos auxiliares de campo siguen siendo pequeños y rápidos. Una regla de negocio puede seguir señalando el control más relevante con `SetError`, pero su mensaje explica la regla con un lenguaje claro y aparece en el resumen para que el usuario entienda por qué Save se ha detenido. Ninguno de los dos es una excepción: un correo electrónico incorrecto es un error esperado con un mensaje normal del producto, mientras que una conexión a la base de datos perdida es un fallo del sistema que se gestiona aparte. Los módulos siguientes trasladan las reglas de negocio a un servicio reutilizable; este módulo crea el hábito.



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:

- ✓ Usar `ErrorProvider` para mostrar y borrar errores a nivel de campo.
- ✓ Comprender la diferencia entre `InvalidMessage` y `ErrorProvider.SetError`.
- ✓ Crear una etiqueta de resumen de validación para la accesibilidad.
- ✓ Validar varios campos desde un único método auxiliar.
- ✓ Separar los avisos de campo de los avisos de reglas de negocio.

Guía del laboratorio / examen

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



Laboratorio práctico

Convierte el formulario de contacto de ValidationClinic en un formulario de validación programática. Crea un `ErrorProvider` para el formulario con `ContainerControl` establecido, `NeverBlink` y el icono alineado a la derecha; escribe un record `FieldValidation` y los métodos auxiliares `ValidateName`, `ValidateEmail` y `ValidatePhone`, que devuelven un mensaje en lugar de tocar la interfaz, y un método `Apply` que llama a `SetError` para cada resultado y rellena un `validationSummaryLabel` visible solo mientras haya errores. Usa `InvalidMessage` en el campo del código de cliente para comparar los dos estilos de aviso, y asegúrate de que corregir un campo borra solo el error de ese campo.

Punto de partida

Abres el proyecto `ValidationClinic` que construiste en el Módulo 1 o, si te incorporas aquí, la versión del Módulo 1 del [repositorio público de ejemplos](#). El formulario de alta de clientes ya tiene `txtName`, `txtEmail`, un botón Save y un botón Cancel, con los controladores `Validating` y la protección `ValidateChildren()` que escribiste la última vez; en este laboratorio añades `txtPhone`, `txtCustomerCode` y una Label de resumen, y trasladas las comprobaciones a métodos auxiliares.

Enfoque sugerido

- ✓ Añade un `ErrorProvider` al formulario y, en un método `SetupValidation` llamado desde el constructor, establece `ContainerControl = this`, `BlinkStyle = ErrorBlinkStyle.NeverBlink` y `SetIconAlignment(..., ErrorIconAlignment.MiddleRight)` con un pequeño `SetIconPadding` para cada campo.
- ✓ Declara el record `FieldValidation` y después escribe `ValidateName`, `ValidateEmail` y `ValidatePhone` de modo que cada uno recorte el valor, lo compruebe y devuelva un record con un mensaje claro, o un mensaje vacío cuando el valor sea correcto. Ninguno de ellos puede llamar a `SetError`.
- ✓ Coloca una Label llamada `validationSummaryLabel` encima de los botones con `Visible = false`, y escribe `Apply(params FieldValidation[] results)` para que establezca o borre el error de cada control, rellene la etiqueta con una línea por problema, muestre la etiqueta solo mientras haya problemas y devuelva si todo es válido.
- ✓ Llama a `Apply(ValidateName(), ValidateEmail(), ValidatePhone())` al principio de `btnSave_Click` y sal cuando devuelva `false`. Haz que el controlador `Validating` de cada campo establezca o borre solo el error de su propio control a partir del resultado de su método auxiliar y, mientras el resumen esté visible tras un Save fallido, vuelve a ejecutar `Apply` completo para que la lista se reduzca a medida que se corrigen los campos.
- ✓ En `txtCustomerCode_Validating`, usa `InvalidMessage` en lugar del proveedor; después ejecuta el formulario con varios valores no válidos, corrige los campos de uno en uno y escribe una breve nota sobre en qué se diferencian los dos estilos de aviso para el usuario.

Entregables requeridos

- ✓ ErrorProvider a nivel de formulario con ContainerControl, BlinkStyle y alineación del icono configurados
- ✓ Record FieldValidation y métodos auxiliares ValidateName, ValidateEmail y ValidatePhone
- ✓ Método Apply que establece o borra el error de cada control y devuelve si todos los campos son válidos
- ✓ validationSummaryLabel que enumera todos los problemas actuales y se oculta cuando no hay ninguno
- ✓ Un campo que usa InvalidMessage en lugar de ErrorProvider, con una nota que compara ambos

Preguntas de repaso

- ✓ El controlador `txtName_Validating` de un compañero llama a `errorProvider.Clear()` en cuanto el nombre pasa a ser válido. ¿Qué falla para un usuario que también ha escrito un correo electrónico no válido, y a qué debería llamar el controlador en su lugar?
- ✓ El campo del código de cliente muestra su mensaje mediante `InvalidMessage`. ¿Qué necesitaría saber la etiqueta de resumen sobre ese campo para enumerarlo también, y qué te dice eso sobre cuándo es mejor opción un proveedor compartido?
- ✓ Un usuario escribe un correo electrónico bien formado que ya pertenece a otro contacto. ¿Es un aviso de campo o un aviso de regla de negocio, cuándo debe ejecutarse la comprobación y dónde debe aparecer el mensaje?



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.