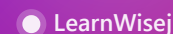


Binding bidirezionale ed editor CRUD

Guida della lezione · Guida al lab / esame



Guida della lezione

DataBindings e BindingSource negli editor, EndEdit, CancelEdit, AddNew e RemoveCurrent, binding diretto alle entità contro binding a un modello di modifica, e flussi di salvataggio, annullamento, eliminazione e ricaricamento che terminano in SaveChangesAsync. Che cosa significa per uno sviluppatore Wisej.NET che lavora con database reali — e come affrontare questo modulo.



Focus di produzione

DataBindings e BindingSource negli editor, EndEdit, CancelEdit, AddNew e RemoveCurrent, binding diretto alle entità contro binding a un modello di modifica, e flussi di salvataggio, annullamento, eliminazione e ricaricamento che terminano in SaveChangesAsync.

1. Il problema in produzione

Gli operatori di Support Desk vogliono schermate Add Ticket ed Edit Ticket che si comportino come form desktop: un dialogo modale, i pulsanti Save, Cancel e Delete, e una griglia che mostra la modifica nell'istante in cui il dialogo si chiude. Il binding bidirezionale rende quella schermata rapida da costruire, ma Save è un'operazione deliberata, non un effetto collaterale della digitazione. Se l'editor legge un valore collegato troppo presto, salva una riga della griglia senza tracking come se fosse tracciata, o lascia che un doppio click invii due volte, il form sembra comunque corretto mentre perde o duplica dati in silenzio. Questo modulo rende espliciti salvataggio, annullamento, eliminazione e ricaricamento, così l'editor dà la sensazione di un form desktop e segue comunque regole di accesso ai dati sicure per il web.

2. Concetti chiave

Concetto	Perché è importante
binding bidirezionale	Una proprietà di un controllo come <code>TextBox.Text</code> è collegata a una proprietà dell'elemento corrente, così il controllo mostra il modello e il modello riceve ciò che l'utente ha digitato.
DataSourceUpdateMode	L'argomento di <code>DataBindings.Add</code> che decide quando un controllo spinge il suo valore nel modello: alla validazione (quando il focus esce) oppure a ogni cambio di proprietà.
EndEdit e CancelEdit	<code>BindingSource.EndEdit</code> conferma nell'elemento corrente le modifiche in sospeso dei controlli; <code>CancelEdit</code> le scarta. Save inizia sempre con EndEdit.
AddNew e RemoveCurrent	Operazioni di lista sulla BindingSource che creano un nuovo elemento corrente o rimuovono quello selezionato dalla lista collegata; nessuna delle due tocca il database finché non salvi.
binding diretto alle entità	Collegare i controlli direttamente a un'entità EF Core tracciata, così il change tracker sa già che cosa è cambiato. Produttivo per form piccoli, di breve durata e con un Context privato.
modello di modifica	Una classe solo per l'interfaccia come <code>TicketEditModel</code> che viene collegata, validata e poi mappata sull'entità, dando a validazione, formattazione e filtri di sicurezza un confine tutto loro.
salvataggio ed eliminazione con Context nuovo	Ogni Save e ogni Delete crea il proprio DbContext dalla factory, carica l'entità per chiave, la modifica e chiama <code>SaveChangesAsync</code> , così la fonte di verità è il database, non la riga della griglia.

3. Il filo della lezione

Collegare l'editor: DataBindings e BindingSource

Un editor tipico si apre assegnando un modello di modifica o un'entità a una `BindingSource` e collegando ogni controllo a una proprietà dell'elemento corrente. Nel `TicketEditorForm` la casella del titolo collega `Text` a `Title`, la `ComboBox` del cliente collega `SelectedValue` a `CustomerId`, il selettore della data di scadenza collega `Value` a `DueDate` e una `CheckBox` collega `Checked` a un flag booleano. Ogni chiamata a `DataBindings.Add` indica la proprietà del controllo, la BindingSource, il nome del membro (usa `nameof` così una rinomina rompe la compilazione, non la schermata), abilita la formattazione e sceglie un `DataSourceUpdateMode`. `OnValidation` è adatto a testi e date perché il valore passa quando il focus esce dal controllo; `OnPropertyChanged` è adatto ai lookup, dove la scelta deve arrivare subito al modello.

Per i lookup l'ordine conta. Imposta `DisplayMember` e `ValueMember` e carica le liste di clienti, agenti e categorie prima che il modello di modifica diventi il `DataSource`, altrimenti la chiave esterna collegata non ha nulla da selezionare. `LoadEditorAsync` costruisce il modello: un `TicketEditModel` vuoto con i

valori predefiniti per un nuovo ticket, oppure una lettura senza tracking mappata nel modello, compresa la `RowVersion` che servirà più avanti al controllo di concorrenza.



Errore tipico

Leggere `TextBox.Text` o il modello prima di `EndEdit` mentre la `BindingSource` ha modifiche in sospeso. L'ultimo campo in cui l'utente ha scritto non è ancora confermato, quindi `Save` scrive il valore precedente.

EndEdit, CancelEdit, AddNew e RemoveCurrent

Quando l'utente clicca `Save`, la sequenza è fissa: `ValidateChildren` perché giri ogni gestore `Validating`, `BindingSource.EndEdit` perché l'ultimo valore in sospeso di un controllo arrivi nell'elemento corrente, validazione di dominio sul modello, copia dei valori approvati sull'entità EF Core se usi un modello di modifica, poi `SaveChangesAsync`. Leggi il modello tramite `BindingSource.Current`, non dai controlli.

Gli altri membri della `BindingSource` gestiscono l'elemento corrente, non il database. `CancelEdit` scarta le modifiche in sospeso sull'elemento corrente quando l'elemento supporta la semantica di oggetto modificabile; un modello semplice viene invece ricaricato, oppure il form viene eliminato. `AddNew` crea un nuovo elemento del tipo della lista e lo rende corrente, così un flusso `Add Ticket` che condivide la lista con una griglia può partire da lì. `RemoveCurrent` toglie l'elemento selezionato dalla lista collegata così la griglia si aggiorna all'istante. Nessuno dei due ha parlato con EF Core; una riga rimossa ricompare alla ricerca successiva a meno che un `Context` nuovo non la elimini.

Flussi di salvataggio, annullamento, eliminazione e ricaricamento

Il metodo di salvataggio deve impedire i click duplicati, concludere le modifiche di binding, validare, creare un `DbContext`, caricare l'entità o crearne una nuova, mappare i campi, chiamare `SaveChangesAsync`, e chiudere o aggiornare l'editor. Proteggi il metodo con un flag di salvataggio e un pulsante `Save` disabilitato, ripristina entrambi nel `finally`, e tieni breve il percorso di successo:

```
editBindingSource.EndEdit();
var model = (TicketEditModel)editBindingSource.Current!;
var errors = Validator.Validate(model);
if (errors.Count > 0) { ShowValidation(errors); return; }

await using var db = await DbFactory.CreateDbContextAsync();
var ticket = model.Id == 0
    ? db.Tickets.Add(new Ticket { Number = await NextNumberAsync(db) }).Entity
    : await db.Tickets.SingleAsync(t => t.Id == model.Id);
ticket.Title = model.Title.Trim();
ticket.CustomerId = model.CustomerId!.Value;
ticket.DueDate = model.DueDate;
await db.SaveChangesAsync();
DialogResult = DialogResult.OK; Close();
```

Il percorso di errore è esplicito. Intercetta prima `DbUpdateConcurrencyException`: un altro operatore ha salvato lo stesso ticket e l'utente ha bisogno di un dialogo di conflitto, non di un errore generico. Intercetta `DbUpdateException` separatamente: il database ha rifiutato la modifica (un lookup mancante, un numero

duplicato), quindi registra il dettaglio sul server e mostra un `AlertBox` comprensibile. Cancel è il percorso più semplice: il Context è stato eliminato alla fine di `LoadEditorAsync`, quindi chiudere con `DialogResult.Cancel` non salva nulla. Il ricaricamento spetta al padre: quando `ShowDialog` restituisce `DialogResult.OK`, il browser dei ticket riesegue la sua ricerca.

Delete non deve basarsi solo sulla riga selezionata nella griglia. Chiedi conferma, crea un Context nuovo, carica l'entità per chiave, controlla le regole di business come "i ticket con commenti non possono essere eliminati", rimuovila e chiama `SaveChangesAsync`. Se il caricamento restituisce null, qualcun altro l'ha già eliminata: mostra un messaggio comprensibile e aggiorna la lista. Dove conta la tracciabilità, preferisci un'eliminazione logica che conserva la riga.



Errore tipico

Salvare elementi di lista senza tracking come se fossero entità tracciate. Un `TicketListItem` della griglia è stato proiettato con `AsNoTracking`; modificarlo e chiamare `SaveChangesAsync` su un Context nuovo non scrive nulla. Carica invece l'entità per chiave nel Context di salvataggio.



Errore tipico

Non proteggere Save dai doppi click. Due chiamate `SaveAsync` sovrapposte creano due ticket o si contendono la stessa riga; il flag di salvataggio più un pulsante disabilitato, ripristinati nel `finally`, fanno parte della correttezza, non della rifinitura.

Binding diretto all'entità o modello di modifica?

Il binding diretto alle entità è produttivo per le schermate semplici perché il change tracking di EF Core conosce già i valori originali e correnti, quindi Save è poco più di `SaveChangesAsync`. Usalo quando il form modifica un solo aggregato, il Context è privato del form, il form viene eliminato presto e l'utente può annullare eliminando o ricaricando. Non collegarti mai direttamente a un elemento della griglia senza tracking aspettandoti che EF Core salvi le modifiche: nessuno lo sta tracciando.

Un modello di modifica aggiunge un confine deliberato tra interfaccia e persistenza. È la scelta migliore per validazione, formattazione, valori calcolati da mostrare, filtri di sicurezza e aggiornamenti parziali. Il flusso è: carica l'entità tracciata, mappala su `TicketEditModel`, collega il modello, validalo, poi ricopia sull'entità tracciata solo i valori approvati. Il modello può lasciare completamente fuori `CreatedAt` e `Number`, così un utente non può modificare un campo che la schermata non intendeva esporre, e un validatore che riceve un modello semplice non ha bisogno di un database per essere testato. Scegli in base al profilo di rischio: un form privato, di breve durata e con un solo aggregato può collegarsi all'entità; tutto ciò che ha regole, ruoli o aggiornamenti parziali riceve un modello.



La regola del data binding

Tieni separati i due cicli di vita. Form Wisej.NET, BindingSource e riga selezionata vivono per la sessione; un DbContext vive per un'operazione. Crealo dalla factory, interroga o salva, eliminalo, e non lasciarlo mai diventare stato di sessione.



Obiettivi di apprendimento

Alla fine di questo modulo sarai in grado di:

- ✓ Collegare i controlli di un editor alle proprietà del modello con DataBindings e BindingSource.
- ✓ Usare correttamente BindingSource.EndEdit, CancelEdit, AddNew, RemoveCurrent e la gestione dell'elemento corrente.
- ✓ Implementare i flussi di creazione, modifica, eliminazione, salvataggio, annullamento e ricaricamento.
- ✓ Scegliere tra binding diretto alle entità e binding a un modello di modifica in base al profilo di rischio del form.

Guida al lab / esame

Che cosa costruirai nel lab pratico, l'approccio suggerito e i deliverable richiesti.



Lab pratico

Costruisci i form modali Add Ticket ed Edit Ticket per la Support Desk Data Console: un `TicketEditModel`, un `TicketEditorForm` con una `BindingSource` e un `ErrorProvider`, proprietà di `TextBox`, `ComboBox`, `DateTimePicker` e `CheckBox` collegate al modello, `LoadEditorAsync` per ticket nuovi ed esistenti, `SaveAsync` che chiama `EndEdit`, un segnaposto per la validazione, la mappatura e `SaveChangesAsync`, `Delete` con conferma e un caricamento aggiornato dell'entità, e una griglia padre che si aggiorna dopo `DialogResult.OK`.

Punto di partenza

Apri la soluzione Support Desk Data Console del Modulo 3. La pagina del browser dei ticket cerca e pagina già le righe `TicketListItem` tramite una `BindingSource` in una `DataGridView`, le `ComboBox` di lookup di stato e cliente si caricano prima della prima ricerca, e `SupportDeskContext` è registrato tramite `AddDbContextFactory`. Per ora niente può aggiungere, modificare o eliminare un ticket.

Approccio suggerito

- ✓ Crea per primo `TicketEditModel` nel progetto Services, con le sole proprietà che la schermata può modificare più `Id` e `RowVersion`, e decidi quali campi dell'entità restano fuori.
- ✓ Aggiungi `TicketEditorForm` con un `IDbContextFactory<SupportDeskContext>` iniettato, una `BindingSource`, un `ErrorProvider` e i pulsanti Save, Cancel e Delete; carica i lookup prima di assegnare il modello.
- ✓ Collega ogni controllo con `DataBindings.Add` e un `DataSourceUpdateMode` esplicito, poi scrivi `LoadEditorAsync` sia per il caso del ticket nuovo sia per quello del ticket esistente.
- ✓ Scrivi `SaveAsync` come la pipeline della lettura (protezione, `EndEdit`, validazione, Context nuovo, caricamento o creazione, mappatura, `SaveChangesAsync`, chiusura) e `DeleteAsync` con una conferma e un caricamento aggiornato per chiave.
- ✓ Collega i pulsanti Add ed Edit del browser dei ticket a `ShowDialog` e riesegui la ricerca solo quando il risultato è `DialogResult.OK`; poi prova ogni percorso con una seconda sessione del browser aperta.

Deliverable richiesti

- ✓ `TicketEditModel` e `TicketEditorForm` con una `BindingSource` e un `ErrorProvider`
- ✓ Proprietà di `TextBox`, `ComboBox`, `DateTimePicker` e `CheckBox` collegate al modello di modifica
- ✓ `LoadEditorAsync` per ticket nuovi ed esistenti e `SaveAsync` con `EndEdit`, mappatura e `SaveChangesAsync`
- ✓ `Delete` con conferma, un caricamento aggiornato dell'entità e una gestione pulita dei record già eliminati
- ✓ Griglia padre aggiornata dopo `DialogResult.OK`, e Cancel che non salva nulla

Domande di revisione per lo studente

- ✓ Quale ciclo di vita hai scelto per il DbContext nell'editor, e che cosa dovrebbe essere vero del form perché un Context legato al form diventi accettabile?
- ✓ Quali valori sono collegati all'interfaccia, quali vivono solo nel modello di modifica e quali restano solo nel database, e perché `Number` non è modificabile?
- ✓ Quali modalità di errore hai gestito esplicitamente (doppio click, ticket già eliminato, rifiuto del database, modifica concorrente), e che cosa vede l'operatore in ciascun caso?



La regola del data binding

Tieni separati i due cicli di vita. Form Wisej.NET, BindingSource e riga selezionata vivono per la sessione; un DbContext vive per un'operazione. Crealo dalla factory, interroga o salva, eliminalo, e non lasciarlo mai diventare stato di sessione.