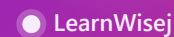


Concorrenza, transazioni, rilascio, diagnostica e capstone

Guida della lezione · Guida al lab / esame



Guida della lezione

Concorrenza ottimistica con RowVersion e DbUpdateConcurrencyException, un dialogo di conflitto con ricarica, sovrascrittura e unione, transazioni e strategie di esecuzione, configurazione e logging specifici per ambiente, una strategia di test e il capstone Support Desk. Che cosa significa per uno sviluppatore Wisej.NET che lavora con database reali — e come affrontare questo modulo.



Focus di produzione

Concorrenza ottimistica con RowVersion e DbUpdateConcurrencyException, un dialogo di conflitto con ricarica, sovrascrittura e unione, transazioni e strategie di esecuzione, configurazione e logging specifici per ambiente, una strategia di test e il capstone Support Desk.

1. Il problema in produzione

Due operatori aprono il ticket T-1042 in due sessioni del browser; uno lo chiude, l'altro ne alza la priorità, e il salvataggio che arriva per secondo decide che cosa ricorda il database. Bloccare le righe non è un'opzione in un'applicazione web, quindi la Support Desk Data Console deve lasciare lavorare entrambi gli utenti, rilevare la collisione al momento del salvataggio, spiegarla e lasciare decidere a una persona. Questo modulo copre anche transazioni, configurazione, migrazioni, logging, test e il capstone.

2. Concetti chiave

Concetto	Perché è importante
concorrenza ottimistica	Gli utenti modificano liberamente; al momento del salvataggio l'applicazione rileva che qualcun altro ha cambiato la riga dopo il caricamento, invece di bloccare le righe.
token RowVersion	Un token di concorrenza generato dal database; EF Core mette il suo valore originale nella clausola WHERE dell'UPDATE, così una riga modificata aggiorna zero righe.
OriginalValue	La versione caricata dall'editor, ripristinata sull'entità tracciata prima di <code>SaveChangesAsync</code> così EF Core confronta con ciò che l'utente ha visto.
DbUpdateConcurrencyException	Lanciata quando una riga attesa non è stata toccata; le sue <code>Entries</code> contengono i valori proposti, e <code>GetDatabaseValuesAsync</code> recupera ciò che il database contiene adesso.
dialogo di conflitto	Un form Wisej.NET che elenca campo, tuo valore, valore nel database e valore originale, e offre Reload, Overwrite quando la policy lo consente, oppure Merge.
strategia di esecuzione	Il meccanismo di retry di EF Core per gli errori transitori; una transazione esplicita deve girare al suo interno così un nuovo tentativo può ripetere l'intera unit of work.
script di migrazione idempotente	SQL generato da <code>dotnet ef migrations script --idempotent</code> che controlla lo storico delle migrazioni prima di ogni passo, così è sicuro su qualsiasi ambiente.

3. Il filo della lezione

Due operatori, un ticket

La concorrenza ottimistica lascia che ogni operatore modifichi il ticket T-1042 e controlla al momento del salvataggio se la riga ha ancora l'aspetto che aveva quando l'editor si è aperto. Il controllo è la colonna `RowVersion` configurata con `IsRowVersion` nel Modulo 2. Il database la cambia a ogni aggiornamento, e quando l'entità tracciata porta la versione originale, EF Core scrive `UPDATE ... WHERE Id = @id AND RowVersion = @original`. Se un'altra sessione ha salvato prima, la clausola WHERE non corrisponde a nulla, vengono toccate zero righe e `SaveChangesAsync` lancia `DbUpdateConcurrencyException`.

Il punto delicato è che l'editor non tiene viva alcuna entità tracciata tra caricamento e salvataggio: mappa una lettura senza tracking in `TicketEditModel`, e il `DbContext` che salva è un Context nuovo. Quindi la versione che l'utente ha visto viaggia con il modello di modifica come stato nascosto: una proprietà `RowVersion` mai collegata a un controllo. Prima di salvare un ticket esistente il form ricarica l'entità e ripristina il token con `db.Entry(ticket).Property(t => t.RowVersion).OriginalValue = model.RowVersion`. Senza quella riga il Context nuovo confronta con la versione che ha appena caricato, che è sempre quella corrente, e nessun conflitto viene mai rilevato.

Quando arriva l'eccezione, traducila in linguaggio di business. `ex.Entries` elenca le entry fallite; per ciascuna, `GetDatabaseValuesAsync` restituisce ciò che il database contiene adesso, oppure `null` quando un altro utente ha eliminato la riga, e il confronto tra `CurrentValues` e i valori del database produce l'elenco dei conflitti:

```
catch (DbUpdateConcurrencyException ex)
{
    var conflicts = new List<ConflictField>();
    foreach (var entry in ex.Entries)
    {
        var dbValues = await entry.GetDatabaseValuesAsync();
        if (dbValues is null) { conflicts.Add(new("Ticket", "your edit", "deleted"));
        continue; }
        foreach (var p in entry.Metadata.GetProperties())
            if (!Equals(entry.CurrentValues[p], dbValues[p]))
                conflicts.Add(new(p.Name, entry.CurrentValues[p], dbValues[p]));
    }
    await ShowConflictDialogAsync(conflicts, ex);    // Reload · Overwrite · Merge
}
```

Un dialogo di conflitto con Reload, Overwrite e Merge

Il dialogo spiega che cosa è successo: qualcuno ha modificato questo ticket mentre lo stavi modificando tu. Mostra, per ogni campo, il tuo valore, il valore corrente nel database e, dove aiuta, il valore originale.

Reload scarta le modifiche locali: `ReloadAsync` su ogni entry, oppure una nuova esecuzione di `LoadEditorAsync`, sostituisce il modello con i valori del database e la nuova `RowVersion`. **Overwrite** salva comunque i valori dell'utente impostando `OriginalValue` alla versione del database e chiamando di nuovo `SaveChangesAsync`; viene offerto solo quando la policy lo consente, per esempio a un supervisore. **Merge** tiene alcuni valori da ciascuna parte, poi salva sopra la versione corrente.

Qualunque sia la scelta, aggiorna poi la griglia così lo stato visibile corrisponde al database; il browser dei ticket senza tracking non si aggiornerà da solo. Il blocco catch sta in `SaveAsync` accanto al gestore di `DbUpdateException`, e la protezione di caricamento tiene `_saving` a true mentre il dialogo è aperto.



Errore tipico: sovrascrivere in silenzio la modifica di un altro utente

Intercettare `DbUpdateConcurrencyException` e salvare di nuovo con la `RowVersion` del database riapre il ticket che un collega ha appena chiuso. Overwrite è una decisione di policy nel dialogo, mai un nuovo tentativo automatico.

Transazioni e strategie di esecuzione

`SaveChangesAsync` è già transazionale per le modifiche di quella chiamata: quando l'editor aggiunge un `Ticket` e il suo primo `TicketComment` in un Context e salva una volta, vengono scritte entrambe le righe o nessuna. Le transazioni esplicite servono solo quando più operazioni devono riuscire o fallire insieme, per esempio chiudere un ticket e archiviarne i commenti in due chiamate; allora `db.Database.BeginTransactionAsync` avvolge l'unità.

La resilienza delle connessioni aggiunge una complicazione. Una strategia di esecuzione, abilitata con `EnableRetryOnFailure`, ritenta automaticamente gli errori transitori, ma solo per le operazioni che gestisce lei. Una transazione avviata fuori dalla strategia non può essere ripetuta, quindi EF Core si rifiuta di avviarne una mentre i retry sono abilitati. Ottieni la strategia da `db.Database.CreateExecutionStrategy()` e passa l'intera unità, inizio e commit compresi, a `ExecuteAsync`; un nuovo tentativo può eseguirla due volte, quindi deve essere sicuro di ripeterla.

Migrazioni, configurazione e logging

Le migrazioni sono codice sorgente, riviste come qualsiasi altro. Per la produzione genera uno script con `dotnet ef migrations script --idempotent`, rivedi l'SQL ed eseguilo come passo del rilascio, oppure, dove approvato, distribuisce un `dotnet ef migrations bundle`. La factory di design time tiene gli strumenti indipendenti dallo startup di Wisej.NET, e il progetto Web non deve chiamare `Migrate()` all'avvio in un ambiente con più istanze.



Errore tipico: applicare le migrazioni automaticamente da ogni istanza di produzione

Due server che partono nello stesso momento fanno a gara per modificare le stesse tabelle, e non c'è uno script rivisto da cui fare rollback. Le modifiche di schema sono un passo del rilascio, non una funzionalità dell'applicazione.

Stringhe di connessione e segreti arrivano da configurazione specifica per ambiente, archivi di segreti o impostazioni di rilascio, mai dal codice sorgente: la stringa di sviluppo vive in `appsettings.Development.json` o negli user secrets, e il resto lo inietta l'host. La stessa disciplina protegge i log. `EnableSensitiveDataLogging` scrive i valori dei parametri nell'output SQL, quindi diventa condizionato al fatto che l'ambiente sia Development. In produzione imposta la categoria `Microsoft.EntityFrameworkCore.Database.Command` a Warning e registra quanto basta per diagnosticare un errore senza scrivere dati personali o credenziali del database.



Errore tipico: abilitare il logging dei dati sensibili in produzione

Titoli dei ticket, nomi dei clienti e credenziali finiscono in file di log che molte persone possono leggere. Lega l'impostazione al controllo sull'ambiente Development, non a un flag che qualcuno potrebbe attivare su un server in produzione.

Test, grounding dell'IA e il capstone

Testa i servizi di query separatamente dall'interfaccia Wisej.NET: `SearchTicketsAsync` e `TicketValidator` non dipendono da alcun form. Usa SQLite per i test di integrazione che richiedono un comportamento relazionale; usa il provider InMemory di EF Core solo per test semplici in cui vincoli e traduzione SQL non contano. Testa i validatori, la mappatura, il percorso di concorrenza con due Context e lo script di migrazione su un database vuoto.

Il corso include un pacchetto di grounding per l'IA. Un assistente specifico del corso risponde innanzitutto a partire da questo materiale, cita la sezione del modulo o la voce della mappa delle fonti su cui si basa, e lo dichiara quando non è sicuro di un comportamento specifico del provider invece di inventarlo. Raccomanda Context di breve durata, `BindingSource` come collegamento verso l'interfaccia, validazione

esplicita e protezioni di caricamento asincrone, a meno che non ci sia un motivo valido per scegliere diversamente; la tua nota del capstone è il suo ultimo blocco di grounding.

Il capstone è la Support Desk Data Console stessa: il browser dei ticket proiettato e paginato collegato tramite una `BindingSource`; le ComboBox di lookup; l'editor modale con un modello di modifica e la validazione con `ErrorProvider`; caricamento, salvataggio ed eliminazione asincroni dietro protezioni di caricamento; la concorrenza con RowVersion e il dialogo di conflitto; uno script di migrazione o un piano con bundle con note di configurazione; e la nota di grounding. La rubrica pesa architettura, binding e correttezza di EF Core al 20% ciascuno, UX della validazione e prestazioni al 15% ciascuno, e rilascio e diagnostica al 10%.



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:

- ✓ Gestire i conflitti di concorrenza ottimistica in un dialogo Wisej.NET comprensibile per l'utente.
- ✓ Usare in modo appropriato transazioni e strategie di esecuzione.
- ✓ Preparare flussi di migrazione e rilascio e una configurazione specifica per ambiente.
- ✓ Aggiungere logging, diagnostica, strategia di test e materiale di grounding per l'assistente IA.

Guida al lab / esame

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



Lab pratico

Rendi la Support Desk Data Console sicura per più utenti: aggiungi RowVersion al modello di modifica e al suo stato nascosto, imposta OriginalValue prima di salvare un'entità esistente, riproduci un conflitto modificando lo stesso ticket in due sessioni del browser, intercetta DbUpdateConcurrencyException, costruisci un elenco dei conflitti con i valori proposti e quelli del database, aggiungi i percorsi Reload e Overwrite secondo la policy, genera uno script SQL di migrazione idempotente con note di rilascio e stringhe di connessione specifiche per ambiente, tieni disattivato il logging dei dati sensibili fuori dallo sviluppo, poi completa la revisione del capstone.

Punto di partenza

Apri la Support Desk Data Console che hai portato avanti nei Moduli da 1 a 6: `SupportDeskContext` configura già `RowVersion` con `IsRowVersion`, il browser dei ticket paginato è una proiezione senza tracking, e `TicketEditorForm` salva un `TicketEditModel` passando per `TicketValidator`, un `ErrorProvider` e un gestore di `DbUpdateException`. Manca il percorso di concorrenza: il modello di modifica non porta alcuna versione, `SaveAsync` non ha un catch per `DbUpdateConcurrencyException`, e non esiste uno script di migrazione per la produzione.

Approccio suggerito

- ✓ Aggiungi una proprietà `byte[] RowVersion` a `TicketEditModel`, riempila in `LoadEditorAsync`, tienila come stato nascosto a cui nessun controllo è collegato, e in `SaveAsync` ripristinala con `db.Entry(ticket).Property(t => t.RowVersion).OriginalValue = model.RowVersion` prima di mappare gli altri campi.
- ✓ Riproduci il conflitto prima di gestirlo: apri l'applicazione in due sessioni del browser (un secondo profilo del browser o una finestra privata, così è una sessione Wisej.NET diversa), apri il ticket T-1042 in entrambe, salva nella prima, poi salva nella seconda e verifica nel debugger che `SaveChangesAsync` lanci `DbUpdateConcurrencyException`.
- ✓ Intercetta l'eccezione in `SaveAsync`, costruisci l'elenco di `ConflictField` da `ex.Entries` con `GetDatabaseValuesAsync` (trattando un risultato null come "eliminato da un altro utente") e mostralo in un form `ConflictDialog` la cui `DataGridView` elenca campo, tuo valore, valore nel database e valore originale.
- ✓ Implementa Reload con `ReloadAsync` o con un nuovo `LoadEditorAsync`, implementa Overwrite impostando `OriginalValue` alla versione del database e salvando di nuovo, e mostra Overwrite solo quando il ruolo dell'utente corrente lo consente; dopo ciascun percorso aggiorna la griglia padre così corrisponde al database.
- ✓ Esegui `dotnet ef migrations script --idempotent` in `artifacts/sql`, scrivi le note di rilascio con stringhe di connessione specifiche per ambiente prese dalla configurazione o da un archivio di segreti, proteggi `EnableSensitiveDataLogging` con il controllo sull'ambiente

Development, configura le categorie di logging, poi percorri la rubrica del capstone e completa le checklist.

Deliverable richiesti

- ✓ RowVersion portata nel modello di modifica e impostata come OriginalValue prima del salvataggio
- ✓ Conflitto di concorrenza riproducibile intercettato come DbUpdateConcurrencyException
- ✓ Dialogo di conflitto che elenca i valori proposti e quelli del database con i percorsi Reload e Overwrite
- ✓ Script SQL di migrazione idempotente con note di rilascio e stringhe di connessione specifiche per ambiente
- ✓ Logging dei dati sensibili disattivato fuori dallo sviluppo e revisione del capstone completata

Domande di revisione per lo studente

- ✓ Se tralasciassi la riga con `OriginalValue`, che cosa mostrerebbe il test con due sessioni, e perché sembrerebbe un successo mentre perde in silenzio la modifica del primo operatore?
- ✓ Quali percorsi di risoluzione offriresti a un agente ordinario e quali a un supervisore, e che cosa deve succedere alla griglia e alla `RowVersion` nel modello di modifica dopo ciascun percorso?
- ✓ Tre istanze di produzione partono dopo un rilascio. Nel tuo design, quale di esse applica la migrazione e quando, e che cosa andrebbe storto se `Migrate()` girasse nello startup di ogni istanza?

Consegna del capstone

Il lab di questo modulo è anche la consegna del capstone. La Support Desk Data Console deve dimostrare che sai collegare le entità all'interfaccia con binding bidirezionale, validazione, caricamenti asincroni e sicurezza multiutente, e deve includere queste funzionalità:

- ✓ Browser dei ticket con filtri, paginazione ed elementi di lista proiettati senza tracking, collegato a una `DataGridView` tramite una `BindingSource`.
- ✓ ComboBox di lookup per Customer, Agent, Category, Status e Priority.
- ✓ Form modale di aggiunta/modifica che usa un modello di modifica e una `BindingSource`, con validazione esplicita tramite `ErrorProvider` e feedback di riepilogo.
- ✓ Operazioni asincrone di caricamento, salvataggio ed eliminazione con protezioni di caricamento.
- ✓ Concorrenza ottimistica con RowVersion e il dialogo di conflitto.
- ✓ Script di migrazione o piano con bundle, note di configurazione specifiche per ambiente e una breve nota di grounding per l'assistente IA che spiega i pattern implementati.

Consegna il repository sorgente o uno ZIP; gli screenshot della griglia, dell'editor, di un errore di validazione e del conflitto di concorrenza; i file di migrazione e lo script SQL di produzione generato; una nota di architettura di una pagina; e i file delle checklist completati.

Area di valutazione	Peso	Che cosa significa "eccellente"
Architettura	20%	Confine di servizio chiaro, uso della factory, nessuno stato mutabile condiviso.
Binding	20%	BindingSource, DTO o modello di modifica, EndEdit e aggiornamento della griglia tutti corretti.
Correttezza di EF Core	20%	Async, tracking contro no-tracking, migrazioni e concorrenza gestiti.
UX della validazione	15%	Errori per campo e di riepilogo chiari e riutilizzabili.
Prestazioni	15%	Filtri, paginazione e proiezione lato server con note di misura.
Rilascio e diagnostica	10%	Configurazione sicura, logging e un piano di migrazione.



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.