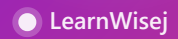


Concurrency, Transactions, Deployment, Diagnostics, and Capstone

Lesson Guide · Lab / Exam Guide



Lesson Guide

Optimistic concurrency with RowVersion and DbUpdateConcurrencyException, a conflict dialog with reload, overwrite and merge, transactions and execution strategies, environment-specific configuration and logging, a testing strategy, and the Support Desk capstone. What this means for a Wisej.NET developer working with real databases — and how to read this module.



Production focus

Optimistic concurrency with RowVersion and DbUpdateConcurrencyException, a conflict dialog with reload, overwrite and merge, transactions and execution strategies, environment-specific configuration and logging, a testing strategy, and the Support Desk capstone.

1. The production problem

Two operators open ticket T-1042 in two browser sessions; one closes it, the other raises its priority, and whichever save lands second decides what the database remembers. Locking rows is not an option in a web application, so the Support Desk Data Console must let both users work, detect the collision at save time, explain it and let a person decide. This module also covers transactions, configuration, migrations, logging, tests and the capstone.

2. Key concepts

Concept	Why it matters
optimistic concurrency	Users edit freely; the application detects at save time that someone else changed the row since it was loaded, instead of locking rows.
RowVersion token	A database-generated concurrency token; EF Core puts its original value in the UPDATE's WHERE clause, so a changed row updates zero rows.
OriginalValue	The version the editor loaded, restored onto the tracked entity before <code>SaveChangesAsync</code> so EF Core compares against what the user saw.
DbUpdateConcurrencyException	Thrown when an expected row was not affected; its <code>Entries</code> hold the proposed values, and <code>GetDatabaseValuesAsync</code> fetches what the database holds now.
conflict dialog	A Wisej.NET form listing field, your value, database value and original value, offering Reload, Overwrite when policy permits, or Merge.
execution strategy	EF Core's retry for transient failures; an explicit transaction must run inside it so a retry can replay the whole unit of work.
idempotent migration script	SQL from <code>dotnet ef migrations script --idempotent</code> that checks the migrations history before each step, so it is safe on any environment.

3. Lesson narrative

Two operators, one ticket

Optimistic concurrency lets every operator edit ticket T-1042 and checks at save time whether the row still looks the way it did when the editor opened. The check is the `RowVersion` column configured with `IsRowVersion` in Module 2. The database changes it on every update, and when the tracked entity carries the original version, EF Core writes `UPDATE ... WHERE Id = @id AND RowVersion = @original`. If another session saved first, the WHERE clause matches nothing, zero rows are affected, and `SaveChangesAsync` throws `DbUpdateConcurrencyException`.

The catch is that the editor keeps no tracked entity alive between load and save: it maps a no-tracking read into `TicketEditModel`, and the `DbContext` that saves is a fresh one. So the version the user saw travels with the edit model as hidden state: a `RowVersion` property that is never bound to a control. Before saving an existing ticket the form reloads the entity and restores the token with `db.Entry(ticket).Property(t => t.RowVersion).OriginalValue = model.RowVersion`. Without that line the fresh context compares against the version it just loaded, which is always current, and no conflict is ever detected.

When the exception arrives, turn it into business language. `ex.Entries` lists the failed entries; for each, `GetDatabaseValuesAsync` returns what the database holds now, or `null` when another user deleted the row, and comparing `CurrentValues` with the database values yields the conflict list:

```

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
}

```

A conflict dialog with Reload, Overwrite and Merge

The dialog explains what happened: someone changed this ticket while you were editing it. It shows, per field, your value, the current database value and, where it helps, the original value. **Reload** discards the local edits: `ReloadAsync` on each entry, or running `LoadEditorAsync` again, replaces the model with the database values and the new `RowVersion`. **Overwrite** saves the user's values anyway by setting `OriginalValue` to the database version and calling `SaveChangesAsync` again; it is offered only when policy permits, for example to a supervisor. **Merge** keeps some values from each side, then saves on top of the current version.

Whatever the choice, refresh the grid afterwards so the visible state matches the database; the no-tracking ticket browser will not update itself. The catch block sits in `SaveAsync` next to the `DbUpdateException` handler, and the loading guard keeps `_saving` true while the dialog is open.



Common mistake: silently overwriting another user's change

Catching `DbUpdateConcurrencyException` and saving again with the database `RowVersion` reopens the ticket a colleague just closed. Overwrite is a policy decision in the dialog, never an automatic retry.

Transactions and execution strategies

`SaveChangesAsync` is already transactional for the changes in that call: when the editor adds a `Ticket` and its first `TicketComment` in one context and saves once, both rows are written or neither is. Explicit transactions are needed only when several operations must succeed or fail together, for instance closing a ticket and archiving its comments in two calls; then `db.Database.BeginTransactionAsync` wraps the unit.

Connection resiliency adds a twist. An execution strategy, enabled with `EnableRetryOnFailure`, retries transient failures automatically, but only for operations it owns. A transaction begun outside the strategy cannot be replayed, so EF Core refuses to start one while retries are enabled. Obtain the strategy from `db.Database.CreateExecutionStrategy()` and pass the whole unit, begin and commit included, to `ExecuteAsync`; a retry may run it twice, so it must be safe to repeat.

Migrations, configuration and logging

Migrations are source code, reviewed like any other. For production, generate a script with `dotnet ef migrations script --idempotent`, review the SQL and run it as a release step, or, where approved, ship a `dotnet ef migrations bundle`. The design-time factory keeps the tools independent of Wisej.NET startup, and the Web project must not call `Migrate()` on startup in a multi-instance environment.



Common mistake: applying migrations automatically from every production instance

Two servers starting at the same moment race to alter the same tables, and there is no reviewed script to roll back from. Schema changes are a release step, not an application feature.

Connection strings and secrets come from environment-specific configuration, secret stores or deployment settings, never from source code: the development string lives in `appsettings.Development.json` or user secrets, and the host injects the rest. The same discipline protects the logs. `EnableSensitiveDataLogging` writes parameter values into the SQL output, so it becomes conditional on the environment being Development. In production, set the `Microsoft.EntityFrameworkCore.Database.Command` category to Warning and log enough to diagnose a failure without writing personal data or database credentials.



Common mistake: enabling sensitive-data logging in production

Ticket titles, customer names and credentials end up in log files many people can read. Tie the setting to the Development environment check, not to a flag someone might flip on a live server.

Tests, AI grounding and the capstone

Test the query services separately from the Wisej.NET UI: `SearchTicketsAsync` and `TicketValidator` have no dependency on a form. Use SQLite for integration tests that need relational behaviour; use the EF Core InMemory provider only for simple tests where constraints and SQL translation do not matter. Test the validators, the mapping, the concurrency path with two contexts, and the migration script against an empty database.

The course ships an AI grounding pack. A course-specific assistant answers from this material first, cites the module section or source map entry it relies on, and says so when unsure about provider-specific behaviour instead of inventing it. It recommends short-lived contexts, `BindingSource` as the UI bridge, explicit validation and async loading guards unless there is a valid reason to choose otherwise; your capstone note is its last grounding chunk.

The capstone is the Support Desk Data Console itself: the projected, paged ticket browser bound through a `BindingSource`; lookup ComboBoxes; the modal editor with an edit model and `ErrorProvider` validation; async load, save and delete behind loading guards; RowVersion concurrency with the conflict dialog; a migration script or bundle plan with configuration notes; and the grounding note. The rubric weighs architecture, binding and EF Core correctness at 20% each, validation UX and performance at 15% each, and deployment and diagnostics at 10%.



Data-binding habit

Keep the two lifetimes apart. Wisej.NET forms, BindingSources and the selected row live for the session; a DbContext lives for one operation. Create it from the factory, query or save, dispose it, and never let it become session state.



Learning objectives

By the end of this module you'll be able to:

- ✓ Handle optimistic concurrency conflicts in a user-friendly Wisej.NET dialog.
- ✓ Use transactions and execution strategies appropriately.
- ✓ Prepare migration/deployment workflows and environment-specific configuration.
- ✓ Add logging, diagnostics, test strategy, and AI assistant grounding material.

Lab / Exam Guide

What you'll build in the hands-on lab, the suggested approach, and the required deliverables.



Hands-on lab

Make the Support Desk Data Console safe for multiple users: add `RowVersion` to the edit model and its hidden state, set `OriginalValue` before saving an existing entity, reproduce a conflict by editing the same ticket in two browser sessions, catch `DbUpdateConcurrencyException`, build a conflict list of proposed and database values, add `Reload` and `Overwrite` paths according to policy, generate an idempotent SQL migration script with deployment notes and environment-specific connection strings, keep sensitive-data logging off outside development, then complete the capstone review.

Starting point

Open the Support Desk Data Console you have carried through Modules 1 to 6: `SupportDeskContext` already configures `RowVersion` with `IsRowVersion`, the paged ticket browser is a no-tracking projection, and `TicketEditorForm` saves a `TicketEditModel` through `TicketValidator`, an `ErrorProvider` and a `DbUpdateException` handler. What is missing is the concurrency path: the edit model carries no version, `SaveAsync` has no catch for `DbUpdateConcurrencyException`, and there is no production migration script.

Suggested approach

- ✓ Add a `byte[] RowVersion` property to `TicketEditModel`, fill it in `LoadEditorAsync`, keep it as hidden state that no control binds to, and in `SaveAsync` restore it with `db.Entry(ticket).Property(t => t.RowVersion).OriginalValue = model.RowVersion` before mapping the other fields.
- ✓ Reproduce the conflict before you handle it: open the application in two browser sessions (a second browser profile or a private window, so it is a different Wisej.NET session), open ticket T-1042 in both, save the first, then save the second and confirm in the debugger that `SaveChangesAsync` throws `DbUpdateConcurrencyException`.
- ✓ Catch the exception in `SaveAsync`, build the `ConflictField` list from `ex.Entries` with `GetDatabaseValuesAsync` (treating a null result as "deleted by another user"), and show it in a `ConflictDialog` form whose `DataGridView` lists field, your value, database value and original value.
- ✓ Implement `Reload` with `ReloadAsync` or a fresh `LoadEditorAsync`, implement `Overwrite` by setting `OriginalValue` to the database version and saving again, and show `Overwrite` only when the current user's role permits it; after either path refresh the parent grid so it matches the database.
- ✓ Run `dotnet ef migrations script --idempotent` into `artifacts/sql`, write the deployment notes with environment-specific connection strings sourced from configuration or a secret store, guard `EnableSensitiveDataLogging` with the Development environment check, configure the logging categories, then walk the capstone rubric and complete the checklists.

Required deliverables

- ✓ RowVersion carried in the edit model and set as OriginalValue before saving
- ✓ Reproducible concurrency conflict caught as DbUpdateConcurrencyException
- ✓ Conflict dialog listing proposed and database values with Reload and Overwrite paths
- ✓ Idempotent SQL migration script with deployment notes and environment-specific connection strings
- ✓ Sensitive-data logging disabled outside development and the capstone review completed

Student review questions

- ✓ If you left out the `OriginalValue` line, what would the two-session test show, and why would it look like success while silently losing the first operator's change?
- ✓ Which resolution paths would you offer an ordinary agent and which a supervisor, and what must happen to the grid and the `RowVersion` in the edit model after each path?
- ✓ Three production instances start after a deployment. In your design, which of them applies the migration and when, and what would go wrong if `Migrate()` ran in each instance's startup?

Capstone hand-in

This module's lab is also the capstone submission. The Support Desk Data Console must demonstrate that you can wire entities to the UI with two-way binding, validation, async loads and multi-user safety, and it must include these features:

- ✓ Ticket browser with filters, paging and no-tracking projected list items, bound to a `DataGridView` through a `BindingSource`.
- ✓ Lookup ComboBoxes for Customer, Agent, Category, Status and Priority.
- ✓ Modal add/edit form using an edit model and a `BindingSource`, with explicit validation through `ErrorProvider` and summary feedback.
- ✓ Async load, save and delete operations with loading guards.
- ✓ RowVersion optimistic concurrency with the conflict dialog.
- ✓ Migration script or bundle plan, environment-specific configuration notes, and a short AI assistant grounding note that explains the implemented patterns.

Submit the source repository or a ZIP; screenshots of the grid, the editor, a validation error and the concurrency conflict; the migration files and the generated production SQL script; a one-page architecture note; and the completed checklist files.

Rubric area	Weight	Excellent looks like
Architecture	20%	Clear service boundary, factory usage, no shared mutable state.
Binding	20%	BindingSource, DTO or edit model, EndEdit and grid refresh all correct.
EF Core correctness	20%	Async, tracking versus no-tracking, migrations and concurrency handled.
Validation UX	15%	Field and summary errors clear and reusable.
Performance	15%	Server-side filtering, paging and projection with measured notes.
Deployment and diagnostics	10%	Safe configuration, logging and a migration plan.



Data-binding habit

Keep the two lifetimes apart. Wisej.NET forms, BindingSources and the selected row live for the session; a DbContext lives for one operation. Create it from the factory, query or save, dispose it, and never let it become session state.