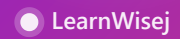


Modeling, DbContext Configuration, and Migrations

Lesson Guide · Lab / Exam Guide



Lesson Guide

Entities shaped for data-bound screens, Fluent API for keys, lengths, indexes, delete behaviour and RowVersion, migrations as reviewed source with scripts and bundles, and the design-time factory. What this means for a Wisej.NET developer working with real databases — and how to read this module.



Production focus

Entities shaped for data-bound screens, Fluent API for keys, lengths, indexes, delete behaviour and RowVersion, migrations as reviewed source with scripts and bundles, and the design-time factory.

1. The production problem

The Support Desk Data Console will show tickets in a pageable grid, edit them in a modal form and let several agents work the same queue at once, and each screen is only as good as the model underneath it. A filter with no matching index becomes a table scan at fifty thousand tickets, two agents saving the same ticket silently lose one agent's work unless the row carries a concurrency token, and a schema patched by hand breaks the next deployment. The model is the contract between the UI and the database, so this module builds it explicitly and versions it as source.

2. Key concepts

Concept	Why it matters
binding-friendly entity	An entity with a predictable key, stable display fields, explicit navigation properties and small lookup tables, so a grid, a <code>ComboBox</code> and an editor can all bind to it.
TicketListItem view model	A flat projection of only the columns the grid shows, so the grid never receives the whole Ticket graph and stays independent from the editing unit of work.
Fluent API configuration	Mapping rules written in <code>OnModelCreating</code> (required strings, lengths, indexes, delete behaviour, precision, defaults) so persistence rules live next to the <code>DbContext</code> .
RowVersion concurrency token	A database-generated byte array marked <code>IsRowVersion</code> ; EF Core adds its original value to the UPDATE's WHERE clause, so a stale save updates zero rows and throws <code>DbUpdateConcurrencyException</code> .
delete behaviour	The <code>OnDelete</code> rule of a relationship (Restrict, SetNull, Cascade) that decides whether deleting a customer is refused, unassigns tickets, or removes comments too.
migration	A C# class plus the model snapshot describing one reviewed schema change, committed to source control and applied in order to bring any database up to the model.
design-time factory	An <code>IDesignTimeDbContextFactory</code> that tells the <code>dotnet ef</code> tools how to construct the context without running the Wisej.NET application startup.

3. Lesson narrative

Entities shaped for master-detail screens

Data-bound UIs work best when the model has predictable keys, stable display fields, explicit relationships and small lookup tables. In the Support Desk Data Console the `Ticket` entity carries `Customer`, `Agent`, `Category`, `Status`, `Priority`, `DueDate`, `Title`, `Description`, `RowVersion`, `CreatedAt` and `UpdatedAt`. `Customer`, `Agent` and `Category` are small lookup entities, each with a `Tickets` collection, and `TicketComment` is the detail collection under a ticket.

`CustomerId` and `CategoryId` are required integers because a ticket without them makes no sense on any screen; `AgentId` is a nullable `int?` because a new ticket exists before anyone is assigned, so the agent `ComboBox` can show "Unassigned" without a sentinel row. A grid does not need every property, so the course also defines `TicketListItem` view models (number, title, customer, status, due date, agent, last update) for the read-only grid, while an editor binds directly to a tracked `Ticket` or to an edit model mapped back to the entity.

Fluent API first for production models

EF Core conventions discover keys and foreign keys from names, but a production application should explicitly configure everything that affects schema and behaviour: required strings, maximum lengths, the indexes the search filters use, delete behaviour, decimal precision, default values and concurrency tokens. Data annotations are fine for simple projects; the Fluent API keeps persistence rules close to the

`DbContext` and avoids mixing UI validation with database mapping. The heart of `SupportDeskContext.OnModelCreating` :

```
modelBuilder.Entity<Ticket>(b =>
{
    b.Property(x => x.Title).HasMaxLength(180).IsRequired();
    b.Property(x => x.Status).HasMaxLength(30).IsRequired();
    b.Property(x => x.RowVersion).IsRowVersion();
    b.HasIndex(x => new { x.Status, x.DueDate });
    b.HasIndex(x => x.CustomerId);
    b.HasIndex(x => x.UpdatedAt);
    b.HasOne(x => x.Customer).WithMany(c => c.Tickets)
        .HasForeignKey(x => x.CustomerId).OnDelete(DeleteBehavior.Restrict);
    b.HasOne(x => x.Agent).WithMany(a => a.Tickets)
        .HasForeignKey(x => x.AgentId).OnDelete(DeleteBehavior.SetNull);
});
```

Every line is a decision the ticket browser depends on. The composite index on `Status` and `DueDate` serves the queue that filters by status and sorts by due date, `CustomerId` serves the customer filter and `UpdatedAt` the "recently changed" sort. `Restrict` means a customer or category with tickets cannot be deleted by accident, `SetNull` means a departing agent simply unassigns their tickets, and comments use `Cascade`. A money-like column would get `HasPrecision(18, 2)` here, and a default status of "Open" is set here rather than in form code.

The concurrency token matters most in a multi-user Wisej.NET application. A `byte[] RowVersion` configured with `IsRowVersion` becomes a SQL Server `rowversion` column the database changes on every write, and EF Core includes the original value in the update condition: `UPDATE ... WHERE Id = @p0 AND RowVersion = @p1`. If another user changed the row after it was loaded, the update affects zero rows and `SaveChangesAsync` throws `DbUpdateConcurrencyException`; the UI can then show the current database values and offer reload, overwrite or merge. SQLite has no `rowversion` type: there the property is still a concurrency token, but a trigger or code must bump it.



Common mistake: annotations for everything

Decorating every entity with `[Required]`, `[MaxLength]` and `[Timestamp]` when the Fluent API would be clearer, so the entity is both persistence mapping and UI validation contract.



Common mistake: no concurrency token on multi-user screens

Leaving `RowVersion` out because "the grid is read-only". The editor is not, and without a token the second agent to save silently overwrites the first.

Migrations as reviewed source

Migrations are source code. During development, add one whenever the model changes with `dotnet ef migrations add InitialCreate --project SupportDesk.Data --startup-project SupportDesk.Web`, review the generated `Up` and `Down` operations, then apply it with `dotnet ef database update`. Review catches a rename scaffolded as a drop-and-add column that loses data, a `Cascade` where you meant

`Restrict`, a missing index, or a `rowversion` column that never appeared. The migration and the model snapshot are committed together.

Production is different. Avoid unreviewed automatic schema changes on application startup in multi-instance environments: two instances starting at once race to apply the same migration. Prefer a reviewed SQL script from `dotnet ef migrations script --idempotent` when database approval and rollback planning are required, or a migration bundle from `dotnet ef migrations bundle`, a self-contained executable a release pipeline runs before switching traffic.

- Connection strings externalised, never in source.
- Every migration reviewed; script or bundle generated and kept with the release.
- Multi-instance startup migration risk addressed in writing.

The `dotnet ef` tools must be able to construct `SupportDeskContext` at design time, and a design-time factory is the most explicit way to let them. `SupportDeskDesignTimeFactory` implements `IDesignTimeDbContextFactory<SupportDeskContext>`; its `CreateDbContext` builds the options with the application's provider and a safe local connection (LocalDB's `SupportDeskDev`, trusted connection, no password) and returns a new context. It matters because the Wisej.NET startup includes UI framework setup that should not run during migration scaffolding; with the factory, the tools never touch it.



Common mistake: migrations as generated files that never need review

Treating the Migrations folder like build output. A migration is a schema change somebody will run against production; review it like any other commit.

Schema constraints are not UI validation

`HasMaxLength(180).IsRequired()` on `Title` produces a `NOT NULL nvarchar(180)` column, and violating it produces a `DbUpdateException` after a round-trip, carrying a provider message no user should read. That is the final safety layer, not the first. The friendly "Enter a title" belongs to the editor's `ErrorProvider` and the `TicketValidator` Module 5 adds, as does "closed tickets cannot have a future due date", which no column constraint can express. Keep the layers separate on purpose: the Fluent API declares what the database must guarantee (nullability, length, the unique index on the ticket number, foreign keys, the concurrency token); `DataAnnotations` and validators on the edit model declare what the user must be told.

Seed data follows the same discipline. Fixed lookup rows such as categories can go through `HasData` so they travel with the migration; the fifty tickets are better produced by a development-only seeding method that creates one context from the factory, checks whether tickets already exist, adds the rows, calls `SaveChangesAsync` and disposes the context. Behind a "Seed development data" button with a loading guard and a `try / catch`, it is where the Module 1 count query first returns more than zero.



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:

- ✓ Model entities and relationships for master-detail UI.
- ✓ Use Fluent API for keys, indexes, required fields, length, decimal precision, delete behavior, and row version.
- ✓ Create migrations and understand when to use scripts, bundles, and design-time factories.
- ✓ Separate database schema constraints from UI/domain validation.

Lab / Exam Guide

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



Hands-on lab

Create the Customer, Agent, Category, Ticket and TicketComment entities for the Support Desk Data Console, configure relationships and delete behaviours with the Fluent API, configure RowVersion with IsRowVersion, add indexes for Status, DueDate, CustomerId and UpdatedAt, add a design-time factory, create the InitialCreate migration, and seed five customers, three agents, categories and at least fifty tickets into a local SQLite or SQL Server LocalDB database.

Starting point

Open the Support Desk Data Console solution from Module 1: `SupportDesk.Web` (the Wisej.NET project), `SupportDesk.Data` with an empty `SupportDeskContext` registered through `AddDbContextFactory`, `SupportDesk.Services` and `SupportDesk.Tests`. The Module 1 page still runs its async count query against a database with no tables, so the first thing you will see when this lab is done is that count turning into a real number.

Suggested approach

- ✓ Write the five entity classes in `SupportDesk.Data` first, with integer keys, the navigation properties in both directions, a nullable `AgentId` and a `byte[] RowVersion` on `Ticket`; keep them free of mapping attributes.
- ✓ Configure everything that touches the schema in `OnModelCreating`: required strings and lengths, `IsRowVersion`, the four planned indexes, and `OnDelete` per relationship (Restrict for Customer and Category, SetNull for Agent, Cascade for comments).
- ✓ Add `SupportDeskDesignTimeFactory` with the same provider the application uses and a local connection that contains no password, then run `dotnet ef migrations add InitialCreate` from the solution root.
- ✓ Read the generated migration before applying it: check the `rowversion` column, the index names, the `ON DELETE` clauses and the snapshot, then run `dotnet ef database update`.
- ✓ Write a development-only seeding method that creates one context from the factory, checks whether tickets already exist, adds the customers, agents, categories and tickets, saves and disposes, and call it from a Seed button behind a loading guard and a `try / catch`.

Required deliverables

- ✓ Customer, Agent, Category, Ticket and TicketComment entities with relationships and delete behaviours
- ✓ RowVersion configured with IsRowVersion
- ✓ Indexes for Status, DueDate, CustomerId and UpdatedAt
- ✓ Design-time factory and the InitialCreate migration
- ✓ Seed data: five customers, three agents, categories and at least fifty tickets

Student review questions

- ✓ You delete a customer who still has open tickets. What happens with the delete behaviour you configured, and what would have happened with Cascade?
- ✓ Two agents load ticket SD-1042, and both click Save a second apart. Which save throws, what does the UPDATE statement's WHERE clause contain, and what does the user see?
- ✓ Why does `IsRequired()` on `Title` not give the user an "Enter a title" message, and where will that message come from later in the course?



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.