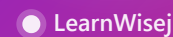


Real Data Architecture: EF Core, Transactions, Repositories, and Commands

Lesson Guide · Lab / Exam Guide



Lesson Guide

Production data access, transactions, service boundaries, command/query split. What this means for an advanced Wisej.NET team — and how to read this module.



Production focus

Production data access, transactions, service boundaries, command/query split.

1. The production problem

Intermediate labs can use fake services. Advanced applications must connect to real data stores without turning screens into SQL clients or leaking persistence concerns into the UI.

2. Key concepts

Concept	Why it matters
EF Core boundary	The line between UI code and Entity Framework Core. Screens call services and commands; they never hold a DbContext or write LINQ against entities.
DbContext lifetime	How long a DbContext instance lives. Short, per-operation contexts avoid stale tracking, memory growth, and cross-session leaks in a long-lived app.
repository vs query service	Repositories load and save aggregates for commands, while query services return read-only projections shaped for screens. Keeping them separate keeps both simple.
transaction boundary	The unit of database work that must succeed or fail together. It belongs in a service or command, not spread across UI event handlers.
command object	A small class that carries the input for one business operation and is executed by a handler, so the operation can be validated, logged, and tested independently.
unit of work	Coordinates all changes in one operation and commits them together, giving a single place to manage transactions and change tracking.
error mapping	Translating database exceptions such as constraint violations and concurrency failures into meaningful application errors the UI can present safely.

3. Lesson narrative

The UI should not know the database

A Wisej.NET form can be rich and stateful without knowing how records are stored. Advanced code maps UI intent into commands and lets application services validate, authorize, persist, and audit the result.

Choose the simplest data boundary that survives change

Repositories, query services, and command handlers are not mandatory ceremony. They are useful when they isolate a database, simplify tests, or prevent UI code from spreading queries everywhere. Use boundaries because they solve a concrete problem.

A data boundary sized for the application

A line-of-business application needs one clear place where the database begins. On one side are forms, pages, and the application services they call. On the other side are Entity Framework Core contexts, entities, and queries. The lab's boundary diagram draws that line and lists what crosses it: commands going in, results and read models coming out, and nothing else. No form holds a DbContext, and no entity is bound to a grid.

The shape of the boundary depends on the application's size. A small application can put queries and commands in a handful of application services that use EF Core directly. A larger one separates repositories, which load and save aggregates for commands, from query services, which return read-only projections for screens. The rule is to add a layer when it isolates something that changes, simplifies a test, or stops queries spreading through the code, and not before.

Whatever the shape, the boundary is documented once and enforced in review. A pull request that adds a LINQ query to a form is rejected on sight.

DbContext lifetime, transactions, and commands

A DbContext is a unit of work with a change tracker, and its lifetime should match the operation it serves. In a long-lived Wisej.NET session, a context stored in a form field lives for hours, accumulates tracked entities, serves stale data, and cannot safely be shared with a background task. The lab's lifetime decision is therefore short: one context per command or per query, created when the operation starts and disposed when it ends.

A transaction boundary wraps the writes that must succeed or fail together. Approving a work order updates the record, writes an audit entry, and queues a notification; the first two belong in one transaction, and the third is queued only after it commits. That boundary lives in the command handler, where the whole operation is visible, not in the form and not in the repository.

A command object such as `ApproveWorkOrderCommand` carries the input for that operation and is executed by a handler that returns a `CommandResult`. Because the command is a plain object, the handler can be tested with an in-memory database and no form at all, which is the lab's first review question.

Thin handlers, real services

The Approve button's click handler in the lab is one line: it builds a command from the current context and the selected work order, hands it to the service, and shows the result. That is the target shape for every UI event handler. The handler translates a gesture into an intent and displays an outcome. It contains no

validation beyond what the screen needs to enable a button, no query, no transaction, and no knowledge of what a database is.

Everything else moves into services and handlers that can be reviewed without opening the Designer. A validator checks the command against business rules. A repository loads the aggregate. The handler applies the transition and saves. An audit writer records what happened. Each piece can be tested in isolation, reused by an import job or a WebMethod, and secured once.

The test for a handler is whether it could be deleted and rewritten by someone who has never seen the database. If not, persistence has leaked into the UI.

Mapping database errors into useful messages

The database reports failures in its own vocabulary: a unique constraint violation, a foreign key failure, a concurrency exception, a timeout, a connection loss. None of those should reach the user as text, and none of them should reach the audit log as a raw stack trace. The error mapping table in the lab lists each database failure the application can produce and maps it to three things: a named result code, a user message, and an audit entry.

A duplicate work order number becomes a result that says the number is already in use, shown next to the field, and logged as a validation failure. A concurrency exception becomes the conflict flow from Module 3. A timeout becomes a message that the operation could not be completed with a correlation ID for support, and a log entry with the query name and duration. The user gets something they can act on; support gets something they can search for.

Mapping happens in the command handler, at the boundary, so that every caller receives the same result codes whether it is a form, a batch, or a job.



Advanced production habit

Do not stop when the screen looks correct. Explain state ownership, security implications, failure paths, and production behavior before considering the module complete.



Learning objectives

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

- ✓ Design a data access boundary appropriate for Wisej.NET LOB applications.
- ✓ Use DbContext lifetime, transactions, and command objects safely.
- ✓ Keep UI event handlers out of persistence logic.
- ✓ Map database errors into useful user and audit messages.

Lab / Exam Guide

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



Hands-on lab

Replace fake WorkOrderService persistence with an EF Core-backed implementation or a production-shaped repository abstraction. Implement Create, Update, Approve, Search, and Audit operations with transaction boundaries and mapped errors.

Suggested approach

- ✓ Read the production problem and identify the architecture risk before touching the UI.
- ✓ Create or update the relevant service, command, model, or component boundary first.
- ✓ Build the Wisej.NET screen or component using the established project standards.
- ✓ Add a failure-path demonstration, not only the happy path.
- ✓ Record what changed in the learning log and update the relevant checklist.

Required deliverables

- ✓ Data access boundary diagram
- ✓ DbContext lifetime decision
- ✓ Command and result classes
- ✓ Transaction example
- ✓ Error mapping table

Student review questions

- ✓ Can a save be tested without creating a Form?
- ✓ What is inside the transaction?
- ✓ How are concurrency and unique key errors surfaced?



Advanced production habit

Do not stop when the screen looks correct. Explain state ownership, security implications, failure paths, and production behavior before considering the module complete.