Project Journal

# Northwind 2025

Use the Northwind database, create an entity framework version. It is still a good example of a database.

All it needs to do is contain some data to hang a web api, mvc site and react web-app from.

Make it portable - a MsSql version of entity framework and in-memory (so it can be used for testing without connecting to anything).

Maybe – a version which connects to another database. This covers the ‘we want to save money so let’s use postgresql/mysql ‘.

## 01-DEC-2022

Create repo on github.

Clone locally:

git clone https://github.com/dsaunders531/Northwind-2025.git

Add this journal.

Add database creation scripts.

Create the database using the creation scripts.

Create 2 projects these are going to be the base database libraries

1 – Northwind.Context – for the base classes

2- Northwind.Context.MsSql – for SQL server things. This will also contain migration info.

Map the database using the package manager console command.

Scaffold-DbContext "Server=[Instance];Database=Northwind-2025;Trusted\_Connection=true;MultipleActiveResultSets=true;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -DataAnnotations -Context NorthwindContext

Install the nuget packages it asks for.

This copies the database structure except triggers, stored procedures & functions.

Views are copied. However, these are database specific. If we migrate to another database type, it may have no concept of a view. Anything associated with views will be removed. The functionality they provide will be mapped later as part of stored procedures.

Re-arrange the output so the context is in the Context folder.

The next steps are for creating a context which is not attached to SQL server.

This means I can create an in-memory, postgresql or whatever else I may need and the context is not tied to MsSql server.

Drag the models into Northwind.Context and change their namespace.

Analyse the output and remove any dependancies on SqlServer.

Move the views code out into its own interface.

The end result – A context which is not dependant on a database and an interface for services which we will need to implement in each database type.

Back in the Northwind.Context.MsSql project, change the connection string for the context and create a migration using package manager console:

Add-Migration InitialCreate

Next – I want to add the stored procs and views. I would implement the same for any database if it supports these kinds of functions. Otherwise, I would replicate the functionality using entity framework context.

The steps for this are:

1. Create an empty migration: Add-Migration StoredProcsAndViews
2. Copy out all the stored procs as text.

Note: The migration steps are only needed if you are planning to move the database. If you just want to put EF onto an existing database or use an in-memory database then these steps are not required.

While writing up the service to run stored procedures and views, I could see there would be a lot of repetition of the same commands – open connection, create a command and execute a data reader.

By moving this code out into individual command classes, I can base class and keep the quantity of typing (and possible errors) low.

It also adds 2 examples of patterns – the command pattern and façade pattern. The service provides a means of running the all commands from one object – the caller does not know how the commands are being run.

## 05-DEC-2022

The command pattern is implemented in a more traditional way now. There is an undo in the textbook definition. Note – implementing undo in a real-world app requires the state before the change to be kept somewhere. A transaction log should be ok. This does not need to be implemented at this time. The command method I am using is working well, there is less code and each class contains relevant steps only.

## 06-DEC-2022

Continue mapping out all the stored procs and views. It does take a while. It’s a bit of a manual process. It’s only typing though.

Migrating databases is a time consuming task – even this relatively small and simple one is taking time. If the sql commands already exist in apps – getting those organised takes time too.

It may be several days (or weeks for complex databases) before any new code is in a fit-to-run state – before any testing can take place.

Added stylecop. Best to do this early on any new project. Adding later is too painful. Use it early and there is less work to keep things clean.

### 07-DEC-2022

Added very basic integration tests for database and context.

Added in-memory dataset. This maintains its state in a file.

### 08-DEC-2022

Implement the service using the in-memory context. Complex LINQ queries involved!

The in-memory context saves it states in the users roaming profile.

### 12-DEC-2022

Carry on implementing the service for in-memory context.

It seems to take longer than writing stored procs which do the same thing! In the real-world, there would be fewer objects to work with. Reporting methods usually end up being stored procs because they are too slow to run. In this case, the in-memory dataset will be fine for what we need it for. The goal is to create a dataset to use in projects to try out new front-end tech.

### 13-DEC-2022

Fix the integration tests – Make sure the Northwind service EF version works without RTE.

### 14-DEC-2022

Add the migration program. Usual problems writing out json data from an Entity – recursive relations.

### 15-DEC-2022

Migration program works!

Added method to bring the data up-to-date. Only the Order table needed updating.

Change stored procs so they are not year dependant.

### 16-DEC-2022

Carry on changing methods for new interface items

Update the tests so they are not date dependant.

## 20-DEC-2022

Added the new functions to the Northwind service.

Add the api. Swagger documentation is in the .net 6 template!

### 21-DEC-2022

Add implementations for service in the controller.

Add missing tests placeholders for patterns

Add service for api to use

Add placeholder tests for controllers. This has changed from previous versions. See the notes in the test project. The only difference is how the server part is created.

It is a lot faster to do things once the ground work is done. Lots done today.

### 22-Dec-2022

Added more tests.

### 29-DEC-2022

Start building react tool chain in a new branch.

Using low-trust (minimal reliance on node.js). Since I use .net for server-side operations it does not make sense to use another server-side technology.

Typescript build only would be nice.

Use qunit for some client-side testing.

Use of node is unavoidable. If using webpack or any other build toolchain.

It is possible to minimise the packages by stating which ones will be used.

Toolchain seems to be:

Typescript (in /src) -> webpack run creates files in /dist -> use webcompiler extentension to minify the files and copy to wwwroot/js.

Pages need to refer to the files in wwwroot/js.

This might mean that debugging while running may not work as expected.

Extensions are needed:

WebPack Task Runner and WebComplier.  
Both of these make using the task runner easy. The task runner explorer allows the build chain to be configured and managed easily.

Steps:

1 – Build simple product viewer with react. This should establish a working toolchain.

2 – Add identity server and .net identity. If open id spec is simple, create it. This should save production costs as you have to pay to use identity server.

3 – Create simple shopping cart. Make sure the payment types part is extendable. Only implement one payment type (make something up). This will require authentication part-way through before the payment details are taken.

4 – Create order processing app. This needs to be completely authorised.