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.

### 30-DEC-2022

https://itnext.io/create-react-typescript-project-with-webpack-and-babel-2431cac8cf5b

Very useful for building a toolchain.

Typescript build only would be nice but not possible – uses webpack and babel to create the build pipeline.

Use the extension WebPack Task Runner.

Use Task Runner Explorer to add these bindings:

Run – Development -> After Build

Watch – Development -> Project Open

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.

Note: for production and pwa install – these changes need to be made:

1 – index.cshtml – add an extra service worker to the service worker list.

2 – in the root service worker which does the offline caching – set the devmode flag to false.

### 03-JAN-23

Add routing. The app routes (should not be too many) populate the app menu.

Removed the service worker part – for development makes life difficult with updates.

Webpack-watch works with hot reloading so this is fine for normal development.

### 04-JAN-23

Fix issue with routing.

If the user navigates to a client-side route, a not found response happens. Added a fallback url and a new Pages route. This is necessary as the fallback route cannot be the same as ‘Index’.

Using partials should help maintain these pages which are identical.

Added new extension to run tasks for npm (like test).

So there are 3 extensions – all used by Task Runner (use the window Task Runner Explorer).

WebCompiler 2022+(for compilerconfig.json). This compiles the sass files for bootstrap (and anything else). Can also be used to minify files outside of the typescript build pipeline.

WebPack Task Runner (for webpack.config.js). Sets up the webpack pipeline and allows for rapid development by compiling after changes.

Npm Task Runner (for package.json). Runs anything for Node which cannot be covered by anything else. In our case, it’s for running test scripts.

Note: Using Jest as a test environment did not work – lots of config issues. Have reverted to the example in the VS template project which uses cross-env to run tests.

Example tests are there.

Make the product catalogue next.

In dev mode – the urls for the api and host are not on local host.

This means that the api and app cannot talk to each other because it is considered a cors request by some browsers.

Even subdomains (api.localhost, app.localhost) would be considered cors. This means that the host for the app needs to pass-through requests to the api.

There would also be problems authenticating.

The api needs to be called via proxy methods from the app host. This is the easiest way to get access to it.

It may also make securing it easier. Eg: use identity server to protect the api. The only client will be the app so managing keys will be easier. Identity will be managed in the app using the default providers.

This would replace using identity and identity server used together (jwt) which seems slow and flaky on my machine.

Next: write out the services and create the components to display products and categories.

### 06-JAN-2023

Adding product viewer with pager and sort.

Note: the pager needs work.

The urls are correct and will work directly. However, this needs to be overridden and float the click back upto the parent container so the page does not do a full redraw. (event.preventdefault).

This should test events and changing state of the parent component.

See the docs for instructions.

### 09-JAN-2023

Sorted the pager issues and bubbling events. There is a bit of extra work to make it work for typescript. The method to push the event up has to be a member of the props object if using strongly typed props. Once that is in place, the process needs attention, making sure the correct method is called (props.eventName in the child component).

Next – add a sort component.

### 10-JAN-2023

Sort component added

### 11-JAN-2023

Search component added.

### 12-JAN-2023

Added very basic seo items (metas, robots.txt, sitemap.xml). Not complete but a starting point.

Very basic tests for component added.

### 13-JAN-2023

More time spent on testing. It is not easy to setup or use. Needs lots of additional components.

The articles below may be useful. The react and jest docs do not seem to work as expected.

Testing needs to be easy – otherwise it will not get done to a good standard and there will be bugs. It also needs to be time-efficient, if it takes too long to write a test, manual testing will take place which does not scale well over time.

<https://www.smashingmagazine.com/2020/06/practical-guide-testing-react-applications-jest/>

<https://www.freecodecamp.org/news/testing-react-hooks/>

### 16-JAN-23

There is a lot of bad documentation about testing (indicates that not a lot gets done).

Seems to be an issue caused by upgrading to v18 react.

Use testing-library/react and an act() fixes the problem.

Tests act on components after they have been rendered.

No further interaction can take place.

Data sources can be mocked so there is some test data.

Events can be raised although it is not possible to change state in the app or pass properties back as there is no re-rendering.

It seems ok. A bit basic. Should be enough to work on. It may be possible to create elaborate tests for forms and form events.

The test framework should get easier and more robust as more complex items get added.

### 17-JAN-23

Added product view as razor page – this is a bit clunky going back and forward between app and page. The only advantage is that the product page is accessible by crawlers.

Added razor page category page. This one will be easier to crawl as it will go through all the content. Categories -> Category -> Products -> Product. The pages are not complex – it is just a demo site. It’s the functionality which is important. Server-side rendering is still an important part of web for seo and non-javascript enabled browsers.

### 19-Jan-23

Replaced ‘test’ component with ‘categories’.

Merged react branch back to main for the next steps…

Next – Reporting would be good. This would need the same things as below (identity) and is more challenging than a shopping basket (has more design patterns and complexity).

1. Add identity & identity server
2. Add reporting framework
3. Update github pages – this needs to describe this project and needs a tidy up – it is quite old now.

### 20-Jan-23

Adding identity. This is scaffolded code with good documentation.

The problem with client-side frameworks is handling authorisation and securing apis.

There may be a cookie or headers required to authenticate with an api.

Identity Server is expensive. Today it is $1500USD per annum. This is quite a lot of cash each year on top of other hosting and licensing costs.

The cost puts this tech out-of-reach for anything other than large national and multinational companies.

The other consideration is that it is not a good idea to implement your own security. It is best for security experts to recommend an approach and keep up to date with the recommendations.

Another option for companies would be to implement an sso provider across the whole estate. This would be more cost effective when there are multiple assets to secure. It also changes the problem from ‘how do I secure this resource’ to ‘how do I implement our SSO provider on this resource’.

Again, implementing an SSO provider in an organisation will be easier for larger companies.

There is a need for a good, affordable (preferably cost-free) implementation for securing apis when the estate is small.

These documents may be useful:

<https://cheatsheetseries.owasp.org/cheatsheets/DotNet_Security_Cheat_Sheet.html>

<https://jasonwatmore.com/post/2022/01/24/net-6-jwt-authentication-with-refresh-tokens-tutorial-with-example-api>

Use CORS to communicate with the api (it will be secured later)

<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>

### 23-JAN-23

Updated the content security policy and cors policy so they use configured values.

Steps:

2 – work out the paths for jwt tokens

3 – implement identity and add the jwt items and front-end client. Ideally, this should be transparent so the scaffolded code in the ‘react with identity’ project will work. The only change is the implementation of jwt tokens on the server-side.

Add data protection using one of the web-farm methods (See data protection methods)

<https://learn.microsoft.com/en-us/aspnet/core/security/data-protection/configuration/overview?view=aspnetcore-7.0>

### 27-Jan-23

Identity is in an acceptable state for most kinds of site. Needs UI reworking for style.

IdentityServer has been added to authorise with the api.

### 30-JAN-23

Integrated Identity with Identity Server so they work together. It is quite easy at face value.

It does appear to be good value – only took a few mins to setup and means identity can be centralised across several applications. This will save time and effort securing applications when there are several microservices.

Steps:

4 – 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.

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