**Visual Studio Toolbox**

**Entity Framework Core**

How to use entity framework core

https://channel9.msdn.com/Shows/Visual-Studio-Toolbox/Entity-Framework-Core-Part-1

Part 1

**Overview**

What is EF?

EF – is an ORM (Object-relational mapper)

EF is less about talking to the database, and more about how you get to work with the data in your app.

What is an ORM?

ORM (mapper) converts what is good for the database (relational) into what is good for the application (object).

What our application needs and what is optimized for the database

**Relational**

* Databases tend to store their data very relationally (relational databases)
* Normalized (specialized) tables based on the data:
  + Customer table, Products table

**Object**

* But applications don’t work that way, we tend to work with domain objects (models)
* Looking at a customer, also want to their orders, and order details, and the products related

EF vs EF Core

* Complete rewrite of classic version
* Similar in name, goal, use
* Both use ADO.NET/ Core under the covers
  + Still using data providers, connections, command objects. It’s just abstracted away.

Data access can be thought of as plumbing, almost every application needs it. You want to streamline “data access”, to spend more time on other functionality, and less time on the plumbing.

EF is the plumbing that every application needs, but we don’t want to build.

The goal always is to spend less time writing plumbing code and more time writing business logic that’s useful for the app.

**Use**

Two different paradigms of using EF.

1st You have this existing database and you want to start using EF Core to query the data and to work with data, but how does that work? Because we only have the thing called “code first”.

Code first - really means code centric.

EF classic

* 4 ways of talking to a database
  + Designer from an existing database
  + Designer with no database
  + Code first with an existing database
  + Code first with no database

EF is very modular only pull down what you need.

* (Relational, Abstractions, Analyzers, Design, SqlServer, Tools)

NuGet Package Manager:

* Search for (Microsoft.EntityFrameworkCore)
* Click on (Microsoft.EntityFrameworkCore.SqlServer)

Starting with an existing database (AdventureWorks2016)

Create all objects that we need with the scaffolding process

**Scaffolding**

Run a command line interface process that will take the existing database and create all of the EF objects that we need to work with it.

The command line interface (CLI) is “the queen of the root”, you can do everything you need to do with .NET Core and EF Core from the command line. And you can do most of what you need to from Visual Studio. The command line comes first, and Visual Studio follows.

.NET Core global tool

Similar to .NET Classic, putting something into global assembly cache (GAC) stores. So you can access it from anywhere.

To run any of the command line tools in EF Core, you need to install the global tool for EF Core.

(dotnet ef dbcontext scaffold "server=(localdb)\mssqllocaldb;Database=Adventureworks2016;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -d -c AwDbContext --context-dir EfStructures -o Entities)

**DbContext**

Like the master control program from Tron. It is the heart and soul of how EF works.

It derives from DbContext, it contains a mechanism to tie into the database, to open connections, works very closely with the change tracker, it holds all the DbSets, and relationship with a database provider.

Provides a mechanism for shaping our database (Mapping).

* Table names
* Schema
* Datatype of a column
* Column name

When installing (Microsoft.EntityFrameworkCore.SqlServer) package, we added the SqlServer data provider into our installation.

**DbSet**

Specialized collections that wrap our tables.

Table per class.

Every table in the database is represented by a DbSet.

*This is the “Object Relational Mapping” objects mapped to tables in the database.*

A key tenant throughout all of .NET Core is dependency injection.

We actually configure connection string and other options, by injecting them into the DbContext.

**OnConfiguring**

Don’t use OnConfiguring.

What OnConfiguring does, is provide a fallback mechanism for if you haven’t configured (optionsBuilder.IsConfigured) class, it will use its hard coded connection string to configure it for you.

C Sharp objects are called “entities” typically.

**OnModelCreating**

Provides a deeper mechanism for shaping the database.

A migration is the reverse process of scaffolding.

**Change tracker**

Provides the power to using EF Core. The core of EF Core.

Anytime you start working with the DeSets in your code the changes are tracked, and that is used to build very efficient queries.

Part 2

Configuring a DbContext with dependency injection.

DbContext is the “master control program” of EF Core, your “general contractor in your project working with the data”.

The DbContext we have to configure,

DbContext doesn’t know its connection string, or what database provider to use.

How do we handle connection strings in the real world?

You wouldn’t put that type of information in your source code and checking it in.

If it’s not a production (only used in development) connection string, its ok to be check in.

With ASP.NET Core there’s a rich configuration ecosystem where based on the environment you’re running in (Development, Staging, Production) it’ll load a different configuration file.

appsettings.production.json file that would have all the production values in it developers wouldn’t have access to it, that would be handled by the “IT Pros”.

You could use Azure Key Vault, Azure Active Directory, and many other ways.

EF Core it never sees the production connection string.

Prior version of EF you had to have it in the app.config “web.config“ file.

Using EF Core in a WPF app as the data access layer, the connection string would actually come from a service and be injected into the app, not in the config file.

Change Tracker

Tracking for all those things that change, get added, deleted modified. So when we call save changes on the dbcontext, then it all wraps up for us. Creates the sql and does it thing.

Provides the most business value from using an ORM in your application.

If were not using an ORM, and were pulling data back (stored procedures, commands, connections, data adapters). So now you have data in your application, but somebody works with that data, and now we have to figure out how to persist those changes. Tracking what fields changed, records were added, records were deleted. All of the plumbing manually, writing all that sql very specific for those fields for those objects.

When we pull something from the database were loading it into the DbSet. When pulling it into the DbSet it gets added to the change tracker. The change tracker wraps a proxy around it, and says I’m going to track this particular instance. If you change a value, it knows what the current value is of that object and also knows what the original value was. The value when it got loaded into the DbSet. Then the DbContext “the general contractor” will hear you ask for the changes to be saved. SaveChanges method is the trigger point for the DbContext to figure out whats going on, and persist the data. The change tracker then reports to the DbContext with all the objects it tracking, all the ones that have changed, and what has been changed. The DbContext takes that information, and talks to the database provider, and creates the sql necessary to persist that data to the database. All done auto-magically. Now all the changes are wrapped up in a sql call.

The SaveChanges method by default is transactional.

Calling SaveChanges opens a transaction, the sql call gets executed, it checks to make sure everything worked, if anything failed it will roll back the entire transaction, and if everything worked it will commit the transaction. Then any server-side properties (row version, id) will get populated on those objects, in the change tracker and the DbSets. Now we have an updated set of values, and our in-memory objects matching those database records is updated to match.

We can turn off change tracking when we do a query (to help with memory pressure). Good for read-only data. Turn off for get, turn on for post.

Query types, or items that don’t not have a primary key. In prior versions of EF Core, we had a DbQueryType, what we have now is a specialized DbSet. Where we till it doesn’t have a primary key, and we don’t have to call AsNoTracking on it, it automatically knows not to add it to the change tracker. Useful for views, stored procedures, or doing a FromSql call to pull back data where the linq maybe difficult write but you want to do these joins.

Part 3

Query execution

There are certain triggers that will have a query execute against the database, it doesn’t happen immediately

How query execution happens

When we create a query in Linq in EF Core nothing actually happens at that time, were just building up an IQueryable object.

AsQueryable method says at some point in time you’re going to run a query against this. When the query executes is when it gets iterated over, if you call ToList on it, or if you bind it to something like a data grid.

When want to make sure were controlling the timing of when the query executes. The most important part is when it executes, or it can really crush your performance.

The term query is overloaded

You have linq query which is defined by the IQueryable, which is like if you’re doing linq to objects. Its linq with some extensions for EF Core. When you go to execute that query, the dbcontext then sends that linq statement to the database provider for the linq translation engine to turn that in to sql to execute in the database.

Same linq, but with extension built in for EF Core.

Like linq to object, you can add wheres, or multiple where clauses (dynamic search screen, and want to allow the user to pick different fields that limit the search – instead of a huge if/ switch statement) chaining is like an AND. Cant chain with an OR.

Ordering is done with OrderBy, and with multiple using ThenBy. Also can use OrderByDescending. Once you add an OrderBy statement into it returns an IOrderedQueryable, as oppose to an IQueryable. But you can just use var. For the most part you don’t care about the type, you really just want the data, you want to bind it to something (view model, grid).

Var was really designed for linq.

You can build up queries dynamically

CSharp like functions:

* .Contains
* .Like
* .IsDate
* .Sum
* .Count
* .Average
* .Max
* .Min
* .Any
* .All

It may not be clear at design time what works or what doesn’t work, so you should have sometime of testing (integration testing “calling the database”)

In EF 5 there will be an extension method off of your query that will output the string.

Another option is with logging turned on EF Core will log the sql that is built at run time, when that sql is executed. So you can see the query in the log.

Or in SSMS turn on profiler and see what comes across. This only works if it can be translated.

Find is a special function, it first looks in the change tracker to see if you already have this object in a DbSet, and if you do it return the object you have. If it doesn’t find it already retrieved it will go retrieve it. Have to get the order right. Just a shortcut for doing a where clause with firstordefault.

FirstOrDefault says get me the first one that matches this or if none of them match then return null. There is also a First, but if it doesn’t find one then throw an exception.

SingleOrDefault is not recommended. What it does server side is select top 2, pull those two records back if there are two records, and if its more than one it’ll throw an exception. Doing the checking for the single on the client, and pulling back two records from the server.

Paging, using skip and offset in sql.

Skip, will skip a certain number of records, and then Take the number of records that you want.

Part 4

Related Data

Very seldom in an application do you need just a single table.

We need an ORM, because databases are optimized one way (based on tables), and applications are optimized another way (domain objects). And we map between them.

To do a join in EF Core is use the Include keyword (left outer join)

Implicitly loading means were doing the includes, and we want it all in one fell swoop (good for a webserver to keep the chattiness down)

Explicitly load means loading them a piece at a time. (could be used for a WPF app)

Projections takes little bits of pieces of the data coming back, and create new object instances from that data

Anonymous objects can be beneficial, but what’s better is a view model

View model (view entity) its parts of different models squished together to be more beneficial for the consumer

A model for the view. Models representing the data objects “tables”.

A view is taking bits and pieces and of those (models) and putting them together as a convent transport mechanism for the required data for the consumer

IQueryable is used by EF Core to setup something that will eventually query the database

You can use linq queries against collections and enumerables and lists, but that isn’t going to query the database. A ICollection with a ToList is not going to go through EF Core

Part 5

When you call save changes on a dbcontext it is automatically executing in a transaction, but we can also enlist our on transaction and use that instead of the implicit transaction that gets created.

Nothing happens until save changes is called.

Save changes will save everything that’s in the change tracker, and marked as has some change. And all the different changes all happen in the transaction. You could also create your own transaction

In the web world at the end of a get request the dbcontext gets recycled.

In the WPF world where were always connected, you don’t recycle the dbcontext on every call.

Which way you delete depends on whether youre in a web world or a WPF style world.