Entity Framework Core:

Microsoft’s cross-platform data access framework for .Net

ORM = Object Relational Mapper

ORM are designed to reduce the friction between how data is structured in a relational database and how you define your classes

ORM allows us to express our queries using our classes

Domain : a specified sphere of activity or Knowledge

Dbset wraps the classes that EF Core will work with

Two Ways to Express LINQ Queries:

* LINQ Methods
* LINQ Operators

Queries are Composable:

* You don’t need to convey them in a single expression.
* You Can break them apart if you have a complicated expression to work out or even to defer query execution

Triggering Queries via Enumeration => looping but the database connection remains open during query enumeration and will stay open until the enumeration is complete and all of the results have been streamed back in some cases.

Good enumeration: Minimal effort on enumeration.

Bad enumeration: Lots of work for each result, Connection stays open until last result is fetched.

Execution: Smarter to get results first. (explicit execution list tolist).

In c#, a target-typed new expression can infer the type from the left side.

Lamda Expression (a => **a.FirstName == "O33li"**)

a => input parameter represents the entity type I am quering

=> => lambda operator token

a.FirstName == "O33li" => the expression that needs to be evaluated using the input parameter

**Parameterized Queries:** protect your database from SQl Injection attacks

Because it was hardcoded into the LINQ query. EF Core sees that the value didn’t just randomly come from some data entry somewhere and it is safe.

* Search value is Diretly in query
  + (.where (a => a.FirstName == “Josie”)
  + No parameter is created in SQL
* Search value in a variable
  + Var name =”Josie”
  + (.where (a => a.FirstName == name)
  + Parameter is Created in SQL

Finding an Entity Using its key Value:

* Dbset.find(keyValue)
* This is the only task that find can be used for
* Not a linq method
* Executes immediately
* If key is found in change tracker, avoids unneeded database query

If you many orderby expression linq will ignore all of them but not the last one , so you should use ThenBy()

Last methods require query to have an OrderBy() method otherwise will throw an exception.

Query translation happens in runtime.

**EF Core has two ways to Avoid tracking when it is a waste of Resources:**

No Track Queries and DbContexts

* AsNoTracking
  + Returns a query, not a Dbset and the Dbcontext doesn’t create entity entry objects to track the result of querys;
  + So you can’t
* In onConfiguring you can use .UseQueryTrackingBahaviorQueryTrackingBehavior.Notracking
  + All queries for this DbContext will default to no tracking

**DbContext represents a session with the database**

**DbContext.ChangeTracker Manages a collection of entityEntry Objects**

**EntityEntry state info each entity: CurrentValues,OriginalValues, State enum , Entity and more.**

Entity and the DbContext:

Entity is one of those overloaded terms that’s used throughout software.

In-Memory Objects;

Entites = In-Memory objects with key(Identity) properties that the DbContext and DbContext changer tracker is aware of

DbContext Contains EntityEntry objects with reference pointers to in-memory objects

Tracking and Saving WorkFlow:

EF Core Creates tracking objects for each entity

* Tracking states
  + Unchanged (Default)
  + Added
  + Modified
  + Deleted
* DbContext maintains EntityEntries
  + {‘dsadas’,Modified}
  + {‘dsadas’,Added}
  + {‘dsadas’, Added }
* On SaveChanges
  + DbContext updates State for tracked entities
  + Works with provider to compose SQL statements
  + Executes statements on database
* Return #Affected and new PKs
* Updates entity PKs and FKs
* Resets Stats Info

**Add from DbSet or DbContext**

* Track via DbSet
  + Dbset knows the type DbContext knows that it’s added
* Track via Dbcontext
  + Will discover type Knows its added

Until EF Core 7. EF used an extra Select to return the new Id, No it is a single Command = it is a big performance improvement.

**How DbContext Discovers About Changes:**

* DbContext.ChangeTracker.DetectChanges()
  + Reads each object being tracked and updates state details in EntityEntry object
* DbContext.SaveChanges()
  + Always calls DetectChanges for you
* You Can Call Detect Changes() in your Code if needed

Both of these get disposed at the end of the method that instantiated them because them – because of the “using”

Deleting without Querying

1. Fake object with key property filled watch out for possible side effects
2. Stored procedure via raw sql for untracked data
3. Soft delete via Global Query Filters
4. Bulk Delete For untracked Data

Executing Directly in the Database on Untracked Data

* Delete(or update) a row when you only know its’ Id
* Does not interact with ChangeTracker
* Returns # of rows affected
* Dot not mix tracking and these Execute mehothd

get-help add-migration

Add-Migration Tasks For Model Changes:

* Read DbContext to determine data model
* Compare data model to snapshot
* Generate migration file to apply the deltas
* Create updated snapshot file

**Scripting Multiple Migrations:**

**Update-database checked history table in the database**

1. Script-migration

* Default: Scripts every migration

1. Script-migration -idempotent

* Scripts all migrations but check for each object first e.g., table already exists

1. Script-migration From To

* From Specifies last migration run, so start at the next one
* To: Final one to apply

Scaffolding Limitations:

* Updating model when database changes is not currently supported
* Transition to migrations is not pretty

Null! Is called a null forgiving operator.

By default, Scaffolding prepares your code for “lazy loading” related Data

Interpreting One-to-many Relationships:

* These are more fondly known as parent/child relationships
* Convention Over Configuration
  + It is paradigm in software which exists to reduce the number of decisions a developer needs to make.
  + Default behavior that can be overridden using configurations
* The list property or some type of collection is an entity is enough for EF core to understand that this is a one-to-many relationship

Shadow Properties

Properties that existing the data model but not the entity class. These can be inferred by EF Core or you can explicity define them in the DbContext.

**Configuring a Non-Conventional Foreign Key**

**Fk is tied** to a relationship, so you must first describe that relationship

EF Core Logging Capabilities:

EF Core Captures:

1. Sql
2. ChangeTracker activity
3. Interaction with database
4. Database transactions

EF Core specific configurations:

1. EnableDetailedErrors,EnableSensitveData
2. Filter based on message type(databases messages)
3. Even more detailed filtering

Object Graph:

A connected set of related objects

Eager Loading:

Include related objects in query

Query Projections:

Define the shape of query results

Lazy Loading

On the fly retrieval of data related to objects in memory

Enabling Lazy Loading:

* Every navigation property in every entity must be virtual
* Reference the Microsoft.EntityFramework.proxies package
* Use the proxy logic provided by that package (optionsBuilder.UseLazyLoadingProxies())

Explicit Loading

Explicity request related data for objects in memory

You can only load from single object

Query WorkFlow

1. Receives tabular results
2. Materializes results as objects
3. Add tracking details to Dbcontext instance
4. Dbcontext connets the relationship

Include is a method of Dbset and you can’t use Find with Include

**Performance Considerations with Include:**

1-composing many includes in one query could create performance issues. Monitor your queries!

2-include defaults to single SQL Command. Use AssplitQuery() to send multiple SQL Commands instead.

Ef Core can Only Track Entities Known By Dbcontext:

1-Anonymous types are not tracked

2-Entities that are properties of an anonymous type are tracked

3-but if it has properties that are recognized entities, those will be tracked

Attach starts tracking with state set to Unchanged

DbContext.Entry gives you a lot of fine-grained control over the change tracker

When a parent is deleted from the database all of children are deleted as well

(dependent data cannot be orphaned)

Four way To define Many-to-Many:

1. Skip Navigations

* Direct references from both ends
* Most Common

1. Skip with payload

* Allows database-generated data in extra colums

1. Explicit join Class

* Additional properties accessible via code

1. Uni-Directional Many-to-Many

* Relationship exists, but you only need to navigate from one end

DbContext must be able to identify a principal (“parent”) and a dependent (“child”)

Common Ways EF Core Identifies One-to-one:

1. Navigations on both ends with Fk in the dependent

* EF Core will recognize one-to-one and idenfity the dependent

1. Navigations on one end Fk on the other

* EF Core will recognize one-to-one and idenfity the dependent

1. Navigations on both ends

* EF Core requies a mapping to define principal/dependent