**ORM - Object Relational Mapping**

Applikasiyalarda olan datalar fiziki olaraq database’lərdə saxlanılır. Beləliklə yazılan app ilə database arasında daim bir bağlantı üzərindən əlaqə saxlanılır. Çünki app’in əldə etdiyi xarici datalar, database’də saxlanıla bilsin və ya istənilən zaman o datalar üzərində müəyyən əməliyyatlar aparıla bilsin. Bu səbəbdən app ilə database arasında dediyimiz bağlantı yaradılır və o bağlantı üzərindən data transferi həyata keçir. Yazılan kodlar, bu bağlantı üzərindən database’ə anlayacağı dildən sorğular göndərir və database də bu sorğulara əsasən müəyyən datalar üzərində əməliyyatlar həyata keçirir. Bu sorğular əsasən SQL dilində olur. ORM’ə olan ehtiyac elə burada yaranır.

Ado.Net vasitəsi ilə yazılmış app’lərin daxilində məlum olduğu kimi sql sorğuları yer almaqdadır. Kodların arasında yer alan bu Query’lər isə ümumi olaraq kodun çirklənməsinə səbəb olur. Bu şəkildə yazılan kodlar SQL’ə olan bağlılığı artırmaqda, idarə oluna bilməsini isə çətinləşdirməkdədir. Əlavə olaraq komandamıza gələcək olan yeni üzvümüz yazılan bu kodları başa düşməklə yanaşı həmçinin SQL dilini bilməli, onu anlamalı və tətbiq etməyi bacarmalıdır. Bu isə əlavə resursların israfına gətirib çıxarır.

Uzun sözün qısası, kod daxilində SQL kodları yazmaq və Database’dən qayıdan cavabı manual olaraq parse etmək böyük proyektlərdə böyük problemlərə səbəb ola bilər. Bu problemlər aşağıdaki şəkildədir.

-Kodun çirklənməsinə səbəb olur.

-Development və support resursu daha çox olacaq bir proyektə səbəb olacaq.

-Database’ə bağlılıq yaradır.

-Kompleks sorğuları manual olaraq yazmaq lazım olur.

-Developer tərəfindən SQL məsuliyyəti də gözlənilir.

-Database sorğuları nəticəsində gələn dataları manual olaraq parse edilir. Sorğu zamanı table və ya column kimi asılılıqlar olduğu kimi gələn dataların da eyni şəkildə asılılıqları olacaqdır.

-Database daxilində baş vermiş dəyişikliklərə əsasən kod təkrardan review edilməlidir.

-Kodun inkşafı zamanı bütün database əməliyyatları, yazılan proqramlaşdırma dilinin və OOP’in avantajını istifadə etmədən icra edilir.

**ORM nədir?**

Software və database arasındaki bağlantı üzərindən sorğular vasitəsi ilə data transferini OOP istifadə edərək təmin olunmasını və beləliklə kodun da, developer’in də SQL’ə asılılığı olmadan sürətli və asanlıqla əməliyyatlar həyata keçirə biləcəyi bir anlayışdır.

**Entity Framework**

Migration,MigrationBuilder,CustomSQL

Demək olar ki, Database’lə bağlı bütün əməliyyatları C# kodları vasitəsi ilə həyata keçirə bilirik. Lakin bir sıra məqamlarda, zamandan qazanmaq və kodları gələcək yönümlü yazmaq üçün yenə də SQL query’lərdən istifadə edirik.

Bu proses Migration yaradan zaman icra edilir. Ilk öncə boş migration yaratmalı oluruq. Hər migration class’ı, Migration class’dan miras alır və müvafiq olaraq update-migration və remove-migration’a uyğun olaraq Up və Down method’ları generate olunur. Hər bir method MigrationBuilder type parameter qəbul edir. Hər iki method daxilində bu parametr’ə aid olan .Sql(string query) method’u işə salaraq yazmış olduğumuz query icra edilir.

**View in Entity Framework**

Entity Framework vasitəsi ilə biz həmçinin View yaradıb, View’dan gələn dataları qəbul edə bilərik. Bunun üçün ilk öncə boş migration yaradılmalıdır. Daha sonra Up method daxilində icra etməli olduğumuz query’ni migrationBuilder.Sql(query); method’u vasitəsi ilə işə icra edirik. Down method daxilində isə drop əməliyyatı üçün lazım olan query yazılır və bu query də yuxarıda yazdığımız method ilə icra edilir.

Daha sonra yerinə yetirməli olduğumuz bir sıra əməliyyatlar da mövcuddur. Bildiyimiz kimi Entity Framework vasitəsi ilə hər hansısa SQL table yaratmaq üçün DbSet<TEntity> property’ə ehtiyac duyulur. View üçün də bu proses keçərlidir.

Əlavə olaraq FluentApi vasitəsi ilə yaranacaq olan SQL obyektinin table yox, view olduğunu göstərməliyik. Context class’ın OnModelCreating(ModelBuilder modelBuilder) method’u daxilində modelBuilder.Entity<ViewEntity>().HasNoKey().ToView(“viewName”) vasitəsi ilə

və ya

FluentApi’ın separate class halında yəni IEntityTypeConfiguration<TEntity> interface’dən miras alan class’ın Configure(EntityTypeBuilder<TEntity> builder) method’u daxilində builder.HasNoKey().ToView(“viewName”) method işə salınaraq hər iki yol ilə də View create ola bilər.

View’dan gələn datalar, Table’larda olduğu kimi context obyekti vasitəsi ilə Entity adıdan müraciət edilir.

**Owned type**

**Database First**

Database first anlayışı o deməkdir ki, bizim daha öncədən Databazamız dizayn edilib və table’lar artıq yaradılıb. Hazır olan table’ları biz EntityFramework vasitəsi ilə modelləşdirməli və entity’lərimizi əldə etməliyik. Bunun üçün bir sıra Package’lərə ehtiyac vardır.

**Packages For Database First**

Package Manager Console(PMC) istifadə ediriksə,

Microsoft.EntityFrameworkCore.Tools və Microsoft.EntityFrameworkCore.[Provider] package’ləri proyektə əlavə edilməlidir.

Dotnet CLI istifadə ediriksə,

Microsoft.EntityFrameworkCore.Design və Microsoft.EntityFrameworkCore.[Provider] package’ləri proyektə əlavə edilməlidir. Əlavə olaraq CLI istifadə edə bilmək üçün bu command line’dan istifadə etmək lazımdır. dotnet tool install --global dotnet-ef

**Scaffolding**

PMC vasitəsi ilə bu proses aşağıdaki command vasitəsilə olur:

Scaffold-DbContext ‘connectionString’ Microsoft.EntityFrameworkCore.[Provider]

Dotnet CLI vasitəsi ilə bu proses aşağıdaki command vasitəsilə olur:

dotnet ef dbcontext scaffold ‘connectionString’ Microsoft.EntityFrameworkCore.[Provider]

**Xüsusi Table’ların gətirilməsi**

Bu kodları run etdikdən sonra default olaraq database daxilində yerləşən bütün table’lar modelləşdirilir və Project’ə add olunur. Lakin biz sadəcə seçili table’ların modelləşdirilməsini və əlavə edilməsini istəyiriksə aşağıdaki ardıcıllığı izləməliyik:

PMC vasitəsi ilə bu proses aşağıdaki command vasitəsilə olur:

Scaffold-DbContext ‘connectionString’ Microsoft.EntityFrameworkCore.[Provider] -Tables TableName1, …, TableNameN

Dotnet CLI vasitəsi ilə bu proses aşağıdaki command vasitəsilə olur:

dotnet ef dbcontext scaffold ‘connectionString’ Microsoft.EntityFrameworkCore.[Provider] --table TableName1, …, --table TableNameN

**DbContext adının dəyişdirilməsi**

Scaffold vasitəsi ilə əldə etdiyimiz DbContext class’ı Database adı ilə və Context sonluğu ilə yaranır. Biz bu adı dəyişdirmək istəsək,

PMC vasitəsi ilə bu proses aşağıdaki command vasitəsilə olur:

Scaffold-DbContext ‘connectionString’ Microsoft.EntityFrameworkCore.[Provider] -Context ContextName

Dotnet CLI vasitəsi ilə bu proses aşağıdaki command vasitəsilə olur:

dotnet ef dbcontext scaffold ‘connectionString’ Microsoft.EntityFrameworkCore.[Provider] --context ContextName

**Path və Namespace**

Əldə olunan Entity’lər və DbContext class’ı, default olaraq proyektin özünə modellənir və namespace’ini istifadə edir. Əgər əldə olunan Entity’lərin və DbContext class’ın başqa bir path’də yer almağını istəsək,

**Path**

PMC vasitəsi ilə bu proses aşağıdaki command vasitəsilə olur:

Scaffold-DbContext ‘connectionString’ Microsoft.EntityFrameworkCore.[Provider] -ContextDir FolderForContextClass -OutputDir FolderForEntities

Dotnet CLI vasitəsi ilə bu proses aşağıdaki command vasitəsilə olur:

dotnet ef dbcontext scaffold ‘connectionString’ Microsoft.EntityFrameworkCore.[Provider] --context-dir FolderForContextClass --output-dir FolderForEntities

**Namespace**

PMC vasitəsi ilə bu proses aşağıdaki command vasitəsilə olur:

Scaffold-DbContext ‘connectionString’ Microsoft.EntityFrameworkCore.[Provider] -Namespace NamespaceForEntities --ContextNamespace NamespaceForContext

Dotnet CLI vasitəsi ilə bu proses aşağıdaki command vasitəsilə olur:

dotnet ef dbcontext scaffold ‘connectionString’ Microsoft.EntityFrameworkCore.[Provider] --namespace NamespaceForEntities --context-namespace NamespaceForContext

**Force**

Database’də baş verən dəyişiklikləri project’ə yansıtmaq üçün Scaffold prosesini təkrarlamaq lazımdır. Lakin bu zaman Entity’lərin zatən yarandığı ilə bağlı error alacayıq. Error almadan dəyişiklikləri əldə etmək üçün Scaffold prosesinin sonunda Force keyword’dən istifadə etmək kifayətdir.

PMC vasitəsi ilə bu proses aşağıdaki command vasitəsilə olur:

Scaffold-DbContext ‘connectionString’ Microsoft.EntityFrameworkCore.[Provider] -Force

Dotnet CLI vasitəsi ilə bu proses aşağıdaki command vasitəsilə olur:

dotnet ef dbcontext scaffold ‘connectionString’ Microsoft.EntityFrameworkCore.[Provider] -- force

**Migration**

Code tərəfdə Entity’lər üzərində edilən dəyişikliklər Database tərəfdə də həmçinin icra edilməsi üçün Migration’dan istifadə edilir. Migration Package Manager Console və ya Dotnet CLI vasitəsi ilə yaranır.

**Add Migration**

Migration’lar Add Migration prosesi ilə yaradılır. Bilməli olduğumuz məqam budur ki, Entity tərəfdə edilən dəyişikliklər mütləq olaraq Migration vasitəsi ilə Database’ə göndərilməlidir. Add Migration prosesi aşağıdaki kimidir.

PCM vasitəsi ilə Migration yaranması prosesi: add-migration “migrationName”

Dotnet CLI vasitəsi ilə Migration yaranması prosesi: dotnet ef migrations add “migrationName”

Beləliklə default olaraq Migrations folder yaranır, daxilində abstract olan Migration Class’dan miras alan Partial Class(lar) yaranır və bu class daxilində override edilmiş Up və Down method’ları yer alır. Əgər Migration Class’lar, Migrations folder daxilində yox, başqa bir path’də yaransın istəyiriksə bunun üçün aşağıdaki prosesi icra etməliyik.

PCM vasitəsi ilə Migration əlavə olunma prosesi: add-migration “migrationName” -OutputDir “Path”

Dotnet CLI vasitəsi ilə Migration əlavə olunma prosesi: dotnet ef migrations add “migrationName” --output-dir “Path”

**Get Migration**

Add olunan bütün Migration’ları görmək üçün Get Migration’dan istifadə edilir.

PCM vasitəsi ilə Migration əlavə olunma prosesi: get-migration

Dotnet CLI vasitəsi ilə Migration əlavə olunma prosesi: dotnet ef migrations list

**Up-Down Methods**

Up method, bəhs etdiyimiz dəyişikliklərin Database’də də icra edilməsi üçün olan method’dur.

Down method’u isə icra edilən dəyişikliyin bir əvvəlki halına qaytarılması üçündür.

Bir sıra situasiyalarda boş Migration yaradaraq Up method’u Customize edəcəyik. Lakin Down method’u işə salınmadan və Remove Migration prosesi icra edilmədən heç bir şəkildə özümüz migration’ı manual olaraq silməməliyik.

**Remove Migration**

Yaradılan(Add olunan) Migration’ın Database tərəfdə icra edilmədən öncə yəni Update prosesi baş vermədən öncə Remove Migration vasitəsi ilə bu Migration silinə bilər. Remove prosesi hər zaman ən sonuncu Migration’ı silir. Əgər bütün Migration’lar Update olunubsa yəni Update olunmamış Migration yoxdursa Remove prosesi işə düşməyəcək. Remove Migration prosesi aşağıdaki şəkildə olacaq.

PCM vasitəsi ilə Migration əlavə olunma prosesi: remove-migration

Dotnet CLI vasitəsi ilə Migration əlavə olunma prosesi: dotnet ef migrations remove

**Update Database**

Yaradılan (Add olunan) Migration’ın Database tərəfdə icra edilməsi (Migrate) üçün Update Migration prosesi icra edilməlidir. Bu zaman Up method’u işə düşəcək. Update prosesi aşağıdaki şəkildə olacaq.

PCM vasitəsi ilə Migration əlavə olunma prosesi: update-database

Dotnet CLI vasitəsi ilə Migration əlavə olunma prosesi: dotnet ef database update

Update prosesindən sonra Database tərəfdə həmçinin EFMigrationHistory table’ı da yaranacaq və bütün Migration’ların adı orada yer alacaq. Biz Update Database prosesi ilə Migration’ları geri ala bilərik yəni Database’in daha əvvəlki vəziyyətinə qayıda bilərik. Bu zaman geriyə almaq istədiyimiz Migration’a qədər olan hər bir Migration daxilində Down method’u işə düşəcək və Database tərəfində dəyişikliklər icra edilərək əvvəlki halına qayıdacaq. Bu proses aşağıdaki şəkildə olacaq.

PCM vasitəsi ilə Migration əlavə olunma prosesi: update-database “migrationName”

Dotnet CLI vasitəsi ilə Migration əlavə olunma prosesi: dotnet ef database update “migrationName”

Bu proses icra olunduqdan sonra artıq istəyə uyğun olaraq hazırdaki versiyaya qədər olan Migration’ları silmək mümkün olacaq. Əgər biz bütün icra olunan Migration’ları silmək istəsək yəni Database’i başlanğıc halına gətirmək istəsək [migrationName] əvəzinə 0 yazacayıq. Bununla da Database tərəfdə yaranan bütün Table’lar silinəcək.

**Migrate**

Migration’ları sadəcə Package Manager Console və ya Dotnet CLI vasitəsi ilə yox, kod vasitəsi ilə də Migrate yəni icra edə bilərik. Bu proses bizə device üzərində ilk dəfə run olacaq project’in avtomatik olaraq Database tərəfdə bütün table’larının yaranması üçün lazım olur. Migrate method’u işə düşdükdən sonra bütün Migrationlar avtomatik olaraq Migrate ediləcək və Database’də yerini alacaq. Bu prosesin icrası isə aşağıdaki şəkildədir.

MyDbContext context = new();

await context.Database.MigrateAsync()

**On Configuring**

EntityFrameworkCore.Context class daxilində yer alan virtual OnConfiguring method vasitəsi ilə biz Context class üçün müəyyən konfiqurasiyalar həyata keçirə bilərik. Bu konfiqurasiyalara daxildir:

Provider(yəni hansı Database tipinə qoşulacayıq məs: MSSQL, Oracle və s.) , Connection String(qoşulacağımız Database tipinə uyğun qoşulmaq üçün lazım olan text) və s.

public class AppDbContext:DbContext

{

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(connectionStr);

}

}

**Entity**

Başlanğıc olaraq boş bir Entity class’ı yaratmışıqsa və DbSet property olaraq Table yaranması üçün qeyd etmişiksə, add migration prosesi zamanı error alacayıq. Çünki EntityFrameworkCore bizə hər bir Entity’in primary key’ə sahib olmalı olduğunu deyir.

**Create**

Database’ə hər hansısa bir datanı əlavə etmək üçün Add(Entity) və ya AddAsync(Entity) method’larından istifadə edilir. 2 şəkildə data əlavə etmək mümkündür. İlk olaraq bunun üçün bizə Context class’dan instance lazımdır. Birinci yol type safety olaraq Context obyektinin DbSet<TEntity> property’ə bu method’lar tətbiq edilməlidir. İkinci yol isə bir başa Context obyektinə bu method’lar tətbiq edilir, göndərilən obyektə əsasən type müəyyən edilir. Method’lar aşağıdaki şəkildə tətbiq edilir:

MyDbContext context = new();

MyEntity myEntity = new();

context.Add(myEntity); və ya await context.AddAsync(myEntity);

context.MyEntities.Add(myEntity); və ya await context.MyEntities.AddAsync(myEntity);

await context.SaveChangesAsync();

SaveChanges method’u tətbiq olunmazsa Entity tərəfdə əlavə edilən və hazırlanan query execute olunmayacaq və bu səbəblə database’də biz bu əlavə olunmuş data’nı görməyəcəyik. Bu səbəbdən sonda SaveChanges method’u mütləq tətbiq edilməlidir.

**SaveChanges() & SaveChangesAsync()**

Create, Update və Delete əməliyyatlarının icra edilməsi üçün yəni query’lərin execute olunması üçün SaveChanges() və ya SaveChangesAsync() method’ları istifadə edilir. SaveChanges vasitəsi ilə bu action’lar transaction olaraq Database’ə yansıyır. SaveChanges vasitəsi ilə hər hansısa bir query execute edilən zaman uğursuzluqla nəticələnərsə, transaction daxilində olan heç bir əməliyyat Database’ə yansımaz yəni Rollback baş verər. Hər bir SaveChanges method’u yeni bir transaction yaradır. Və transaction da böyük resurs xərclədiyi üçün bu prosesi daha səmərəli şəkildə istifadə etməliyik. Method’lar aşağıdaki şəkildə tətbiq edilir:

context.SaveChanges(); və ya await context.SaveChangesAsync();

**Id**

Add prosesindən və SaveChanges prosesindən sonra Database’ə Add olunan data üçün Id avtomatik olaraq generate olacaq. SaveChanges prosesi başa çatdıqdan sonra biz Add etdiyimiz entity object’in Id’ni get edən zaman generate olunmuş Id’ni əldə edə bilərik.

**Update**

Database’də hansısa bir datanı dəyişmək üçün Update(Entity) method’dan istifadə edilir. Bu method ChangeTracker tərəfindən izlənilməyən Entity obyekti üzərində çalışır. Entity obyekti mütləq Id dəyərinə sahib olmalıdır əks təqdirdə Update prosesin hansı Database obyekti üzərində gedəcəyi bilinməyəck. Update method’u tətbiq etmək istədiyimiz Entity obyekti ChangeTracker tərəfindən izlənirsə bu zaman Update method’u çalışan zaman runtime error’la qarşılaşacağıq.

MyDbContext context = new();

MyEntity myEntity = new();

context.Update(myEntity); & context.MyEntities.Update(myEntity);

Əgər Entity obyekti ChangeTracker tərəfindən izlənilərsə bu zaman obyekt üzərində edilən hər hansısa bir dəyişiklik EntityFrameworkCore tərəfindən hər hansısa bir update prosesin həyata keçiriləcəyini anlayır və bu dəyişikliklərin database tərəfdə də update olunması üçün sadəcə SaveChanges() method’u işə salınır və beləliklə Database’ə bu dəyişiklik tətbiq olunur.

Eyni anda bir neçə data Update olunursa hər update prosesindən sonra SaveChanges method’u işə salınması Update olunan data sayı qədər transaction yaranmasına səbəb olacaq və bu bizə böyük resurs itkisinə gətirib çıxaracaq. Bunun qarşısını almaq üçün isə bütün Update prosesləri bitdiksən sonra yekun bir transaction yaratmaq və bu şəkildə execute etmək lazımdır.

**Delete**

Database’də hər hansısa bir data silmək üçün Remove(Entity) method’dan istifadə edilir. Bu prosesdən sonra Entity’in State’i Deleted olaraq dəyişəcək. Remove method’u vasitəsi ilə hazırlanan query’in execute olunması üçün hər zamanki kimi SaveChanges method’u işə salınmalıdır. Change Tracker tərəfindən izlənilməyən obyektləri Database’dən silmək üçün mütləq şəkildə ID qeyd edilməlidir.

MyDbContext context = new();

MyEntity myEntity = new();

context.Remove(myEntity); & context.MyEntities.Remove(myEntity);

Biz həmçinin EntityState vasitəsi ilə də Database’dən silmə prosesini həyata keçirə bilərik. Bunun üçün Entity’in State’i Deleted olaraq dəyişdirilib ardıyca SaveChanges method’u işə salınmalıdır. Beləliklə, Entity State vasitəsi ilə Remove prosesi həyata keçəcək.

Eyni anda birdən çox data Remove olarsa, Update’də qeyd etdiyimiz proses icra edilməlidir.

**State**

Entity obyektinin vəziyyətini ifadə edən reference State(EntityState) adlanır. Vəziyyəti dedikdə, biz əlimizdə olan Entity obyektinin, Database’dən əldə olunan obyekt olub-olmamasını, yenicə create, update, delete olunub-olunmamasını barədə məlumatı əldə edə bilərik. Context class’dan alınan instance vasitəsi ilə biz Entity’lərin State’lərini əldə edə bilərik. Bu proses ChangeTracker vasitəsi ilə işləyir. ChangeTracker bizə Entity’lərin database tərəfindən izlənməsini və database’də bu object varsa onun code tərəfdəki cari qarşılaşdırılmasını tapmağa kömək edir. Əlimizdə olan Entity object’in State’i aşağıdaki şəkildə əldə edilir.

EntityState state = context.Entry(Entity).State

State’lər aşağıdaki şəkildə olur:

Detached - Entity obyektinin State’i Detached olarsa bu o deməkdir ki, Change Tracker tərəfindən bu object izlənilirmir.

Unchanged - Entity obyektinin State’i Unchanged olarsa bu o deməkdir ki, Change Tracker tərəfindən bu object izlənilir lakin üzərində heç bir dəyişiklik aparılmayıb.

Added - Entity obyektinin State’i Added olarsa bu o deməkdir ki, bu data Add method’u vasitəsi ilə Database’ə əlavə olunması üçün query halına salınmışdır və artıq Change Tracker tərəfindən bu object izlənilir. Save Changes prosesindən sonra Database’ə bu data əlavə olunacaq və Entity obyektinin State’i Unchanged olaraq dəyişəcək.

Modified - Entity obyektinin State’i Modified olarsa bu o deməkdir ki, bu data üzərində Update prosesi gerçəkləştirilmişdir yəni Change Tracker tərəfində izlənilən bu Database object’i code tərəfindəki versiyası ilə üst-üstdə düşmədiyi üçün üzərində dəyişiklik edilmişdir. Save Changes prosesindən sonra Database tərəfdə bu dəyişiklik icra olunacaq və Entity obyektinin State’i Unchanged olaraq dəyişəcək.

Deleted - Entity obyektinin State’i Deleted olarsa bu o deməkdir ki, bu data üzərində Remove prosesi gerçəkləştirilmişdir yəni Change Tracker tərəfindən izlənilən bu Database object’i code tərəfindən silinmişdir. Save Changes prosesindən sonra Database tərəfdə bu remove icra olunacaq.

**Change Tracker**

Change Tracker, context obyekti üzərindən gələn dataların izlənilməsindən cavabdeh olan bir mexanizmdir. Context obyekti vasitəsi ilə gələn datalar, default olaraq hər zaman Change Tracker mexanimi vasitəsilə izlənilir. Gələn dataları loop’a salıb hər bir datanın State’ni kontrol etsək görərik ki hər biri Unchanged’dir.

Əgər biz gələn dataların Change Tracker tərəfindən izlənilməsinin qarşısını almaq istəyiriksə bunun üçün sorğu zamanı AsNoTracking() method istifadə edilməlidir. Bu zaman hər bir datanın State’i Detached olacaqdır. Və entity’lər üzərində edilən dəyişikliklər heç biri Change Tracker mexanizmi tərəfindən izlənilməyəcək.

IEnumerable entries = context.ChangeTracker.Entries<TEntity>(); - entries daxilində yer alan hər bir obyekt vasitəsi ilə biz entry.Entity property ilə dataları, entry.State vasitəsi ilə isə vəziyyətini əldə edə bilərik.

**Select**

Database’dəki table daxilində yer alan bütün dataları əldə etmək üçün müəyyən Select sorğuları həyata keçirmək lazımdır. Bunun üçün LINQ’dan istifadə edəcəyik. LINQ, method və query olaraq 2 yerə bölünür.

**ToList() & ToListAsync()**

List olaraq dataları əldə etmənin ən bəsit yolu context obyekti vasitəsi ilə DbSet<TEntity> olaraq təyin olunan property’ə müraciət edilb ToList() və ya ToListAsync() method’u tətbiq edilərək Select sorğusunu execute etməkdir. Burada IQueryable olan sorğu ToList vasətisə ilə execute olacaq.

**Method syntax**

MyDbContext context = new();

List<MyEntity> myEntityList= context.MyEntites.ToList()

və ya

List<MyEntity> myEntityList= await context.MyEntites.ToListAsync()

**Query Syntax**

MyDbContext context = new();

List<MyEntity> myEntityList = (from parameter in context.Entities

select parameter).ToList();

və ya

List<MyEntity> myEntityList = await (from parameter in context.Entities

select parameter).ToListAsync();

**Where()**

Yazılan sorğuya Where şərti əlavə etmək üçün istifadə edilən method’dur. Entity qəbul edən və geriyə şərtə uyğun boolean dəyər qaytaran Expression qəbul edir. Beləliklə şərtimizi sorğuya əlavə edir. Where method’u geriyə IQueryable qaytarır və bu da o deməkdir ki, sorğu execute edilməlidir.

Hər zamanki kimi ToList method’u istifadə edərək və ya foreach döngüsünə salaraq sorğunu execute edirik.

**Method syntax**

MyDbContext context = new();

List<MyEntity> myEntityList= context.MyEntites.Where(e => e.Id >=10).ToList()

**Query Syntax**

MyContext context = new();

List<MyEntity> myEntityList= await (from ent in context.MyEntites

where ent.Id > 10 && ent.Id<0

select ent).ToListAsync();

**OrderBy()**

Yazılan sorğuya OrdeyBy Ascending(artan sıra yəni a-z, 1-n) tətbiq etmək üçün yəni dataları sıralamaq üçün istifadə olunan method’dur. Lambda expression vasitəsi ilə hansı column üzərindən sıralama aparılacaqsa onu qeyd edirik.

**Method syntax**

MyDbContext context = new();

List<MyEntity> myEntityList = context.MyEntites

.Where(e => e.Id >=10).OrderBy(e=>e.Name).ToList();

**Query Syntax**

MyContext context = new();

List<MyEntity> myEntityList= await (from ent in context.MyEntites

where ent.Id > 10 && ent.Id<0

orderby e.Name Ascending

select ent).ToListAsync();

Query syntax daxilində Ascending yazmasaq da zatən default olaraq artan sıra ilə sıralayacaq.

**ThenBy()**

Yazılan sorğuda bir neçə column’a OrdeyBy Ascending(artan sıra yəni a-z, 1-n) tətbiq etmək üçün yəni dataları sıralamaq üçün istifadə olunan method’dur. OrderBy method’dan sonra Lambda expression vasitəsi ilə hansı column üzərindən sıralama aparılacaqsa onu qeyd edirik.

**Method syntax**

MyDbContext context = new();

List<MyEntity> myEntityList = context.MyEntites

.Where(e => e.Id >=10).OrderBy(e=>e.Name).ThenBy(e=>e.Surname).ThenBy(e=>e.Id)

.ToList();

**Query Syntax**

MyContext context = new();

List<MyEntity> myEntityList= await (from ent in context.MyEntites

where ent.Id > 10 && ent.Id<0

orderby e.Name ,e.Surname, e.Id

select ent).ToListAsync();

**OrderByDescending()**

Yazılan sorğuya OrdeyBy Descending(azalan sıra yəni z-a, n-1) tətbiq etmək üçün yəni dataları sıralamaq üçün istifadə olunan method’dur. Lambda expression vasitəsi ilə hansı column üzərindən sıralama aparılacaqsa onu qeyd edirik.

**Method syntax**

MyDbContext context = new();

List<MyEntity> myEntityList = context.MyEntites

.Where(e => e.Id >=10).OrderByDescending(e=>e.Name).ToList();

**Query Syntax**

MyContext context = new();

List<MyEntity> myEntityList= await (from ent in context.MyEntites

where ent.Id > 10 && ent.Id<0

orderby e.Name descending

select ent).ToListAsync();

**ThenByDescending()**

Yazılan sorğuda bir neçə column’a OrdeyBy Descending(azalan sıra yəni z-a, n-1) tətbiq etmək üçün yəni dataları sıralamaq(defautl olaraq Ascending) üçün istifadə olunan method’dur. OrderByDescending method’dan sonra Lambda expression vasitəsi ilə hansı column üzərindən sıralama aparılacaqsa onu qeyd edirik.

**Method syntax**

MyDbContext context = new();

List<MyEntity> myEntityList = context.MyEntites

.Where(e => e.Id >=10).OrderBy(e=>e.Name).ThenBy(e=>e.Surname).ThenBy(e=>e.Id)

.ToList();

**Query Syntax**

MyContext context = new();

List<MyEntity> myEntityList= await (from ent in context.MyEntites

where ent.Id > 10 && ent.Id<0

orderby e.Name descending ,e.Surname descending, e.Id

select ent).ToListAsync();

**OrderBy, ThenBy, OrderByDescending, ThenByDescending**

Biz OrderBy method’dan sonra ThenBy istifadə etməyə məcbur deyilik. Yəni istəyə uyğun olaraq OrderBy’dan sonra həm ThenBy həm də ThenByDescending istifadə edə bilərik. Eynisi OrderByDescending üçün də keçərlidir.

**IQueryable**

EF Core vasitəsi ilə yazılmış sorğunun execute edilməmiş halı IQueryable adlanır.

Sorğu mərhələsində olan dataları(yəni IQueryable) ToList() method’u ilə execute edib, IEnumerable’dan miras alan List<TEntity> şəklində əldə edirik yəni dataları sorğu mərhələsindən In Memory mərhələsinə alıb dataları əldə edirik.

Lakin bununla yanaşı IQueryable(sorğu) mərhələsində olan dataları Foreach vasitəsi ilə də execute edərək əldə edə bilərik.

Bilmək lazımdır ki, IQueryable olaraq yazılan kod yəni Method və ya Query syntax olaraq yazılan LINQ sorğular, yazıldığı yerdə generate edilmir, yalnız execute prosesi zamanı generate edilir və execute olur.

**IEnumerable**

EF Core vasitəsi ilə generate edilmiş sorğunun execute edilmiş və dataların In Memory’ə yüklənmiş halı IEnumerable adlanır.

**Deferred Execution**

Bildiyimiz kimi sorğular execute olunmayanadək generate olunmur. Bu o deməkdir ki, IQueryable olaraq yazılan sorğu execute edilənədək bu sorğuda yer alan datalarda hər hansısa bir dəyişiklik edildiyi halda sorğu generate edilib execute olan zaman datanın yeni dəyəri nəzərə alınacaq. Bu proses də elə məhz Deferred Execution adlanır.

MyContext context = new();

int productId = 5;

IQueryable<Product> products = from product in context.Products

where product.Id> productId

select product;

productId = 10;

Yuxarıdaki kod nümunəsində yer alan sorğuda productId dəyişənin dəyərində dəyişiklik olduğu halda, sorğu hələ də execute olunmadığı üçün sorğu daxilində artıq dəyişənin yeni dəyəri nəzərə alınacaq.

List<Product> productsList = products.ToList(); yəni ToList vasitəsi ilə və ya foreach vasitəsi ilə sorğu generate olan zaman productId = 10 olaraq generate ediləcək.

**Single() & SingleOrDefault()**

Sorğu nəticəsində şərtə uyğun olan yeganə datanın gəlməsini üçün Single və ya SingleOrDefault method’larından birini istifadə edirik. Hər iki method Async halına sahibdirlər.

Single method tətbiq olunan zaman sorğu nəticəsində gələn yeganə datanı return edir, əgər şərtə uyğun data gəlməzsə(Exception message: Sequence contains no elements) və ya birdən çox data gələrsə(Exception message: Sequence contains more than one elements) bu zaman Exception throw edəcək.

SingleOrDefault method tətbiq olunan zaman sorğu nəticəsində gələn yeganə datanı return edir, əgər şərtə uyğun birdən çox data gələrsə(Exception message: Sequence contains more than one elements) bu zaman Exception throw edəcək, heç bir data gəlməzsə geriyə default dəyər yəni null qaytaracaq.

Exception hallarının yaşanmaması üçün şərtə uyğun olan datanın yeganə olduğuna əmin olduğumuz zaman Single və ya SingleOrDefault method’ları istifadə etməliyik.

**First() & FirstOrDefault()**

Sorğu nəticəsində şərtə uyğun olan datalardan ilkinin gətirilməsi üçün First və ya FirstOrDefault method’larından birini istifadə edirik. Hər iki method Async halına sahibdirlər.

First method tətbiq olunan zaman sorğu nəticəsində gələn datalardan ilkini return edir, əgər şərtə uyğun data gəlməzsə(Exception message: Sequence contains no elements) bu zaman Exception throw edəcək.

FirstOrDefault method tətbiq olunan zaman sorğu nəticəsində gələn datalardan ilkini return edir, əgər şərtə uyğun heç bir data gəlməzsə geriyə default dəyər yəni null qaytaracaq.

**Last & LastOrDefault**

Sorğu nəticəsində şərtə uyğun olan datalardan sonuncusunun gətirilməsi üçün Last və ya LastOrDefault method’larından birini istifadə edirik. Hər iki method Async halına sahibdirlər. Last & LastOrDefault istifadə qaydası First & FirstOrDefault method’larından biraz fərqlidir. Method’lar istifadə olunmadan öncə mütləq sorğuya OrderBy method’u tətbiq edilməlidir sonra method’lardan biri istifadə oluna bilər. Qalan bütün davranışlar isə eynidir.

**Find**

Sorğu nəticəsində sürətli şəkildə Primary Key olaraq təyin olunan Id column’a görə axtarış nəticəsində yeganə datanın gətirilməsi üçün Find və ya FindAsync method’larından birini istifadə edirik. Göndərilən Id dəyərinə sahib data tapılmazsa geriyə default(null) dəyər return edir. Göndərilən Id dəyəri, Primary key column’un type’ı ilə üst-üstə düşməzsə exception baş verir. Async halına sahibdir.

public TEntity GetById(int id)

{

MyDbContext context = new();

MyEntity myEntity = context.MyEntites.Find(id);

}

və ya Primary key olaraq təyin olunan Id column’un type’ı Guid olarsa,

public TEntity GetById(Guid id)

{

MyDbContext context = new();

MyEntity myEntity = context.MyEntites.Find(id);

}

Find method’u vasitəsilə biz həmçinin Composite key’ə(Cross table) sahib olan table datalarını da əldə edə bilərik.

public TEntity GetById(int id1,int id2)

{

MyDbContext context = new();

MyEntity myEntity = context.MyEntites.Find(id1,id2);

}

**Find and Single, SingleOrDefault, First, FirstOrDefault**   
Find method’u sorğudan öncə hər zaman context’i control edir yəni əgər data InMemory’də yoxdursa o zaman sorğu generate olaraq Database’ə yönləndirilir. Yalnız Primary key column(ları) üzərində axtarış aparır və Where şərti ilə işlənmir.

Digərləri isə hər zaman sorğunu bir başa olaraq Database’ə yönləndirir. İstənilən column üzərində Where şərti ilə birgə axtarış aparmaq mümkündür.

**Count**

Sorğu nəticəsində gələn dataların sayını əldə etmək üçün Count method’dan istifadə olunur. Count method’u həm IQueryable həm də IEnumerable’lara tətbiq edilə bilir. Lakin sorğu zamanı zamandan qənaət və əlavə dataların InMemory’ə yüklənməsinin qarşısını almaq üçün ən düzgün olan Count method’nu IQueryable’a tətbiq etməkdir. Count method’u tətbiq edildikdən sonra sorğu generate olub, execute olunacaq və geriyə int type dataların sayını qaytaracaq. Async halı mövcuddur.

MyDbContext context = new();

int elementsCount = context.MyEntites.ToList().Count(e=>e.Id > 100);

əvəzində,

MyDbContext context = new();

int elementsCount = context.MyEntites.Count(e=>e.Id > 100);

**LongCount**

Count method’dan fərqli olaraq geriyə Long dəyər qaytarır. Big dataların sayını əldə etmək üçün istifadə edilir. Async halı mövcuddur.

**Any**

Sorğu nəticəsində geriya datanın gəlib gəlməməsini bilmək üçün Any method’dan istifadə olunur. Geriyə bool tipində dəyər return edir. Əgər şərtə uyğun olaraq dəyər mövcuddursa bu zaman true, mövcud deyilsə false dəyər qaytaracaq. Async halı mövcuddur.

MyDbContext context = new();

bool elementIsExist = context.MyEntites.Where(e=>e.Id>100).Any();

və ya

MyDbContext context = new();

bool elementIsExist = context.MyEntites.Any(e=>e.Id>100);

IQueryable’da işləyəcəyimiz üçün hər iki halda da eyni query generate olunacaq. IEnumerable’da isə Any method istifadə etmək o qədər də düzgün deyil. Əgər bizə dataların özü lazım deyilsə, onları boş yerə InMemory’ə əlavə etmək düzgün deyil. Bir başa olaraq IQueryable zamanı method tətbiq olunmalıdır.

**Max**

Sorğu nəticəsində vermiş olduğumuz Column daxilində yer alan maximum dəyəri əldə etmək üçün Max method’dan istifadə olunur. Method daxilində göndərdiyimiz dəyərin tipinə uyğun olaraq o tipdə dəyəri return edir. Burada diqqət yetirilməli olan məqam, Max method’u yalnız numeric olan property'lərə(column’lara) tətbiq olunur.

**Min**

Sorğu nəticəsində vermiş olduğumuz Column daxilində yer alan minimum dəyəri əldə etmək üçün Min method’dan istifadə olunur. Method daxilində göndərdiyimiz dəyərin tipinə uyğun olaraq o tipdə dəyəri return edir. Burada diqqət yetirilməli olan məqam, Min method’u yalnız numeric olan property'lərə(column’lara) tətbiq olunur.

**Distinct**

Sorğu nəticəsində yalnız Unique dataları əldə etmək üçün Distinct method’dan istifadə olunur. Geriyə IQueryable qaytarır bu da o deməkdir ki, sorğu execute edilməlidir.

**All**

Sorğu nəticəsində əldə olunan dataların hamısının şərti ödəyib-ödəmədiyini bilmək üçün istifadə edilir. Müvafiq olaraq geriyə bool dəyər qayıdır. Async halı mövcuddur.

**Sum**

Sorğu nəticəsində vermiş olduğumuz Column daxilində yer alan dəyərlərin cəmini əldə etmək üçün istifadə olunur. Method daxilində göndərdiyimiz dəyərin tipinə uyğun olaraq o tipdə dəyəri return edir. Burada diqqət yetirilməli olan məqam, Sum method’u yalnız numeric olan property'lərə(column’lara) tətbiq olunur. Async halı mövcuddur.

**Average**

Sorğu nəticəsində vermiş olduğumuz Column daxilində yer alan dəyərlərin ədədi ortasını əldə etmək üçün istifadə olunur. Burada diqqət yetirilməli olan məqam, Average method’u yalnız numeric olan property'lərə(column’lara) tətbiq olunur. Async halı mövcuddur.

**Contains**

Sorğu nəticəsində SQL’dən bildiyimiz LIKE şərtini(%str%) tətbiq etmək üçün istifadə olunur. Where method istifadə olunaraq tətbiq edilir. Where method istifadə olunduğu üçün ToList() method vasitəsi ilə sorğumuz execute edilməlidir.

AppDbContext context = new AppDbContext();

List<MyEntity> myEntityList = context.MyEntites.Where(e => e.Name.Contains(“string”)).ToList();

**StartsWith**

Sorğu nəticəsində SQL’dən bildiyimiz LIKE şərtini(str%) tətbiq etmək üçün istifadə olunur. Where method istifadə olunaraq tətbiq edilir. Where method istifadə olunduğu üçün ToList() method vasitəsi ilə sorğumuz execute edilməlidir.

AppDbContext context = new AppDbContext();

List<MyEntity> myEntityList = context.MyEntites.Where(e => e.Name.StartsWith(“string”)).ToList();

**EndsWith**

Sorğu nəticəsində SQL’dən bildiyimiz LIKE şərtini(%str) tətbiq etmək üçün istifadə olunur. Where method istifadə olunaraq tətbiq edilir. Where method istifadə olunduğu üçün ToList() method vasitəsi ilə sorğumuz execute edilməlidir.

AppDbContext context = new AppDbContext();

List<MyEntity> myEntityList = context.MyEntites.Where(e => e.Name.EndsWith(“string”)).ToList();