Háttéralkalmazások

Objektum-relációs keretrendszerek: Entity Framework Core

Simon.Gabor@vik.bme.hu
Szabó Gábor munkája nyomán

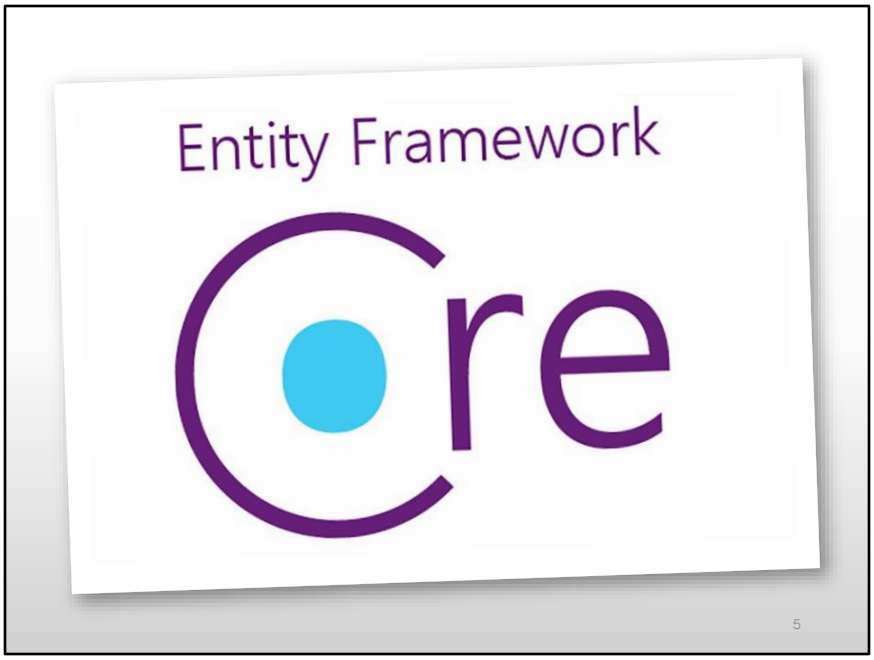
BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

A szerző elérhetősége:
Simon.Gabor@vik.bme.hu

Bevezetés

Az Entity Framework…

- …egy .NET alapú ORM keretrendszer
  - Entity Framework (1.0 - 6.x) – a .NET **Framework** eredeti objektum-relációs leképező keretrendszere
  - Entity Framework **Core** (EF Core 1.0 – 7.x) – a .NET Core/.NET 5, 6, 7 objektum-relációs leképező keretrendszere
    - Nyílt forráskódú (https://github.com/dotnet/efcore)
    - Alapkoncepciójában megegyezik, modernizált, vannak eltérések az eredeti EF-hez képest
    - Az tárgyban erről lesz csak szó
    - EF Core 6 a .NET 6 keretrendszerrel kompatibilis
  - https://learn.microsoft.com/en-us/ef/ef6/
  - https://learn.microsoft.com/en-us/ef/core/

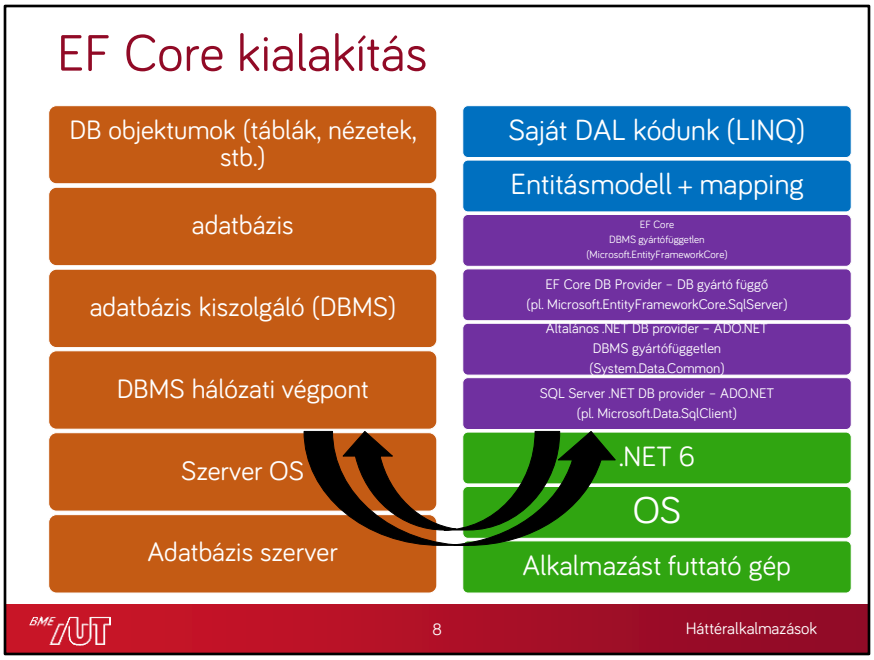

Entity Framework
Core
5

EF Core célja

- Fejlesztési időben
  - támogassa az adatbázis objektumok és a nekik megfelelő OO entitásmodell kialakítását, összerendelését
- Futási időben
  - az entitásmodellen végzett, C#-ban megírt műveleteket adatbázis műveletekre (SQL) fordítsa le
    - a C# kód lehetőleg a sima memóriabeli adatokat (listák, tömbök) kezelő kódhoz hasonló lehessen (pl. LINQ)
  - hajtsa végre a lefordított műveleteket, ne kelljen adatbázisnyelvű kódot írni

Előfeltételek - ismeretek

- SQL, adatbázis alapfogalmak
- Objektum-relációs megfeleltetés (ORM)
  - Táblák = osztályok
  - Sorok = objektumok
  - Oszlopok = tulajdonságok
  - Külső kulcsok = referenciák
- LINQ (EVIP tárgy anyaga)



Kék: saját kód
Lila: NuGet csomag

Alternatívák

C#-ból adatbázis művelet végrehajtása

- Adatbáziskezelés alacsonyabb szinten
  - > EF Core helyett a .NET DB provider (ADO.NET) műveleteit hívjuk közvetlenül
  - > Kisebb overhead – jobb teljesítmény
- Mikro ORM-ek, pl. Dapper, PetaPOCO
  - > Kevesebb szolgáltatás
  - > Kevesebb (fekete) mágia
  - > Egyszerűbb használatbavétel
  - > Kisebb overhead – jobb teljesítmény
- Alternatív .NET ORM-ek
  - > Pl. NHibernate

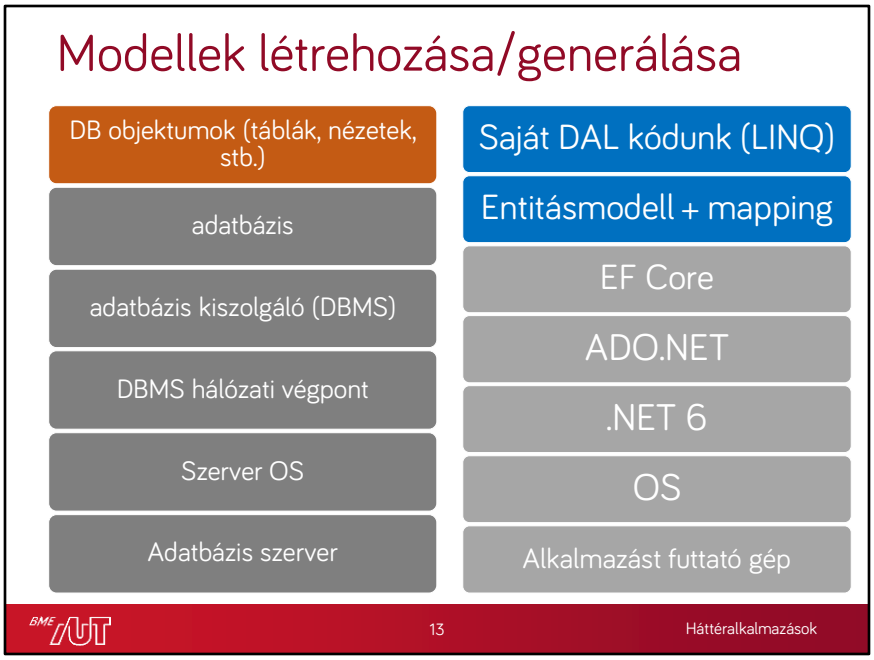
Beizzítás

1. Adatbázis előkészítés
   > Adatbáziskiszolgáló telepítése, konfigurálása
   > (Üres) adatbázis létrehozása
2. .NET 6 telepítése
3. Projekt létrehozás
   > Függőségek telepítése (NuGet)
   - EF Core csomagok, DB Provider csomagok
   - Általános és a megcélzott adatbázis típusa szerinti
   - Elég csak a EF Core DB specifikus csomagot telepíteni, függőségként települnek a nem DB specifikus komponensek is
4. Modellek és mapping létrehozása/generálása

DB Providers

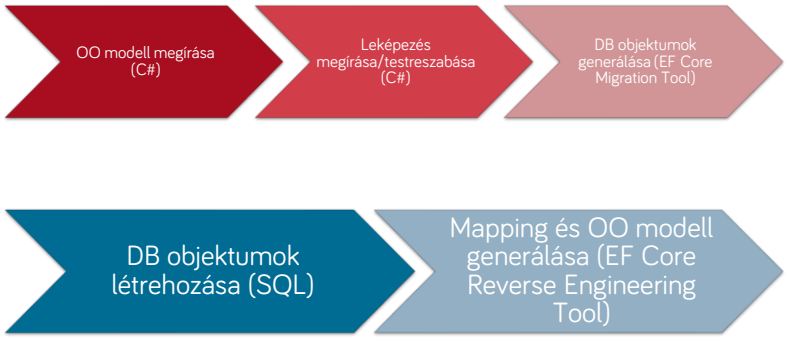
- Az alkalmazásunk adott adatbázismotorral fog kommunikálni, ettől is függhet, hogyan definiáljuk a modellünket
- Microsoft által karbantartott, ingyenes (Microsoft.EntityFrameworkCore.X csomagok):
  - MS SQL Server (Express, LocalDB) 2012+: X = SqlServer
  - SQLite 3.7+: X = Sqlite
  - In-memory (teszteléshez): X = InMemory
  - Azure Cosmos DB SQL API: X = Cosmos
- Egyéb, nem Microsoft által karbantartottak is léteznek:
  - MySQL, MariaDB, Oracle, Firebird, Db2, Informix, Teradata, Access, SQL Server Compact, OpenEdge, File
  - Figyeljünk rá, hogy a megfelelő EF verziót támogatja-e
  - Fizetős is lehet
- https://learn.microsoft.com/en-us/ef/core/providers

Modellezés



EF Core, ADO.NET összevonva-egyszerűsítve jelenik meg itt már

Code-First vs DB-First



Entitásmodell (C#)

- Egyszerű .NET osztályokkal fogalmazzuk meg a modellünket, amik az adatbázisban levő egyes táblákat és köztük levő kapcsolatokat reprezentálják
  - > általában csak property-k
    - egyszerű adattárolásra beépített típusú (string, int, stb.) property-k
    - kapcsolatokhoz másik entitástípus típusú property
  - > Konstruktor a kötelező, de C# null értéket felvenni képes típusú mezők inicializációjához
- A kialakításához nincs szükség semmilyen (EF) függőségre

Dog.cs

```
namespace EFCoreDemo.Entities;

/// <summary>
/// A kutyákat tároló táblát reprezentáló osztály.
/// </summary>
public class Dog
{
    /// <summary>
    /// Az adatbázis által automatikusan generált azonosító.
    /// </summary>
    public int Id { get; set; }

    /// <summary>
    /// A kutya neve (adatbázisban NVARCHAR(max) típus lesz).
    /// </summary>
    public string Name { get; set; }

    /// <summary>
    /// A kutya születési dátuma (ha ismert).
    /// </summary>
    public DateTime? BirthDate { get; set; }

    public override string ToString() =>
        $"({Id}) {Name}{(BirthDate != null ? $" [D.o.B: {BirthDate}]" : "")}";

    /// <summary>
    /// Konstruktor a nem nullozható mezők inicializációjára. Fontos a paraméter neve!
    /// </summary>
    public Dog(string name)
    {
        Name=name;
    }
}
```

Kapcsolatok

- Tegyük fel, hogy egy kutyának gazdái lehetnek
  - Ez egy több-többes kapcsolat a Dogs és People tábla között.
  - Relációs adatbázisban több-többes kapcsolatokat 2 (vagy több) 1-többes kapcsolattal ábrázolunk (a „függő” entitás oldalára kerül a külső kulcs)
    - Extra táblát veszünk fel (kapcsolótábla)
  - Megj: „buta” (csak a két külső kulcsot tartalmazó kapcsolótáblás) kapcsolatokat EF Core 5.0 óta másképp is modellezhetünk, de kevésbé robusztus megoldás

A tulajdonlás entitás

```
namespace EFCoreDemo.Entities
{
    public class DogOwnership
    {
        /// <summary>
        /// A tulajdonlás azonosítója.
        /// </summary>
        public int Id { get; set; }

        /// <summary>
        /// A tulajdonolt kutya azonosítója.
        /// </summary>
        public int DogId { get; set; }

        /// <summary>
        /// A tulajdonolt kutyára mutató referencia, "navigation property".
        /// </summary>
        public Dog Dog { get; set; } = null!

        /// <summary>
        /// A tulajdonolt gazdi azonosítója.
        /// </summary>
        public int PersonId { get; set; }

        /// <summary>
        /// A tulajdonolt gazdira mutató navigation property.
        /// </summary>
        public Person Person { get; set; } = null!
    }
}
```

A referenciák másik oldala

```
using System.Collections.Generic;

namespace EFCoreDemo.Entities
{
    public class Dog
    {
        /* ... */

        /// <summary>
        /// A kutyához tartozó tulajdonlások.
        /// </summary>
        public ICollection<DogOwnership> DogOwnerships { get; } = List<DogOwnership>();

        /* ... */
    }

    /// <summary>
    /// A személy táblát reprezentáló entitás.
    /// </summary>
    public class Person
    {
        /// <summary>
        /// A személy azonosítója.
        /// </summary>
        public int Id { get; set; }

        /// <summary>
        /// A személy neve.
        /// </summary>
        public string Name { get; set; }

        /// <summary>
        /// A személyhez tartozó tulajdonlások.
        /// </summary>
        public ICollection<DogOwnership> DogOwnerships { get; } = List<DogOwnership>();

        /* ...konstruktor a Name beállítására… */

    }
}
```

Kapcsolatok (1-N)

- Függő oldal (Gyerek)
  - Idegen kulcs oszlopnak megfelelő property, a típusa a hivatkozott típus kulcsának típusa
  - És egy kapcsolódó entitás típusú property, ún. **navigációs property**
    - Nagyon kényelmes használni lekérdezésekben
    - Gyakori, hogy nem töltődik be és null értékű, erre fel kell készülnünk!
    - Akkor is lehet null értékű, ha az adatbázisban kötelező a kapcsolat!
    - Ha kötelező a kapcsolat, legyen a típus nem nullozható és inicializáljuk null-ra (nem elírás!)
- Szülő oldal (Principal)
  - Nincs adatbázis oszlop, így csak navigációs property van
  - A navigációs property kollekció típusú
    - Setter nem kell, inicializáljuk üres listával

DbContext

- Az entitásmodellünket *DbContext* leszármazottban fogjuk össze
- Ehhez már kell az EF NuGet csomag függőség
- Az adatbázist reprezentálja a programunk számára
- Tipikus kialakítása
  - Konstruktor, inicializáció (OnConfiguring fv.)
    - Kapcsolódási adatok kezelése – connection string megadása
    - Naplózás, stb.
  - Táblánként egy-egy DbSet<Entitás> property-t szoktunk felvenni
    - nem kötelező, de az EF-et segíti az entitástípusok felderítésében
  - Mapping kód (OnModelCreating fv.)
    - nem csak itt lehet megadni
    - nem kötelező írnunk mapping/leképezési kódot

OnConfiguring, OnModelCreating az ősosztály virtuális függvényei, ezeket felüldefiniáljuk

DogFarmDbContext

```
using EFCoreDemo.Entities;
using Microsoft.EntityFrameworkCore;

namespace DogFarm
{
    /// <summary>
    /// Az adatbázis(kapcsolato)t, -műveleteket reprezentáló osztály.
    /// </summary>
    public class DogFarmDbContext : DbContext
    {
        /// <summary>
        /// Az opciók határozzák meg például, hol található az adatbázis
        /// (pl. connectionString).
        /// Érdemes nem beégetni az osztályhoz, hogy tesztelhető maradjon.
        /// </summary>
        public DogFarmDbContext()
            : base(new DbContextOptionsBuilder().UseSqlServer(
                "Server=(localdb)\\mssqllocaldb;Database=DogFarmDB;Trusted_Connection=True;"
                ).Options) { }

        /// <summary>
        /// A kutyák tábláját reprezentáló tulajdonság.
        /// </summary>
        public DbSet<Dog> Dogs => Set<Dog>();
    }
}
```

EF Core Tools

- Parancssori eszközök
- Gép szinten vagy projekt szinten telepíthető (Microsoft.EntityFrameworkCore.Tools NuGet csomag)
- Visual Studio-ban ajánlott Package Manager Console-ból (PMC) használni, de sima parancssorból (powershell) is használható
- EF Core Migrations
  - Meglévő entitásmodell alapján a relációs modell létrehozása
  - De ennél jóval többet is tud
    - Entitásmodell változások alapján generált adatbázismodell változáscsomagok nyilvántartása a forráskód részeként (ún. migrációk)
    - Migrációk alapján adatbázismodell megváltoztatása
  - A projekt teljes életciklusán át támogatja az adatmodell változások érvényesítését, a két modell szinkronban tartását
- EF Core Reverse Engineering Tool
  - Meglévő relációs modell alapján entitásmodell létrehozása
  - Ahhoz hasonló kód, amit enitásmodellel kezdve írnunk kellett volna
  - Általában csak a projekt elején használjuk, a későbbi változások átvezetése nehézkes

EF Core Migrations

- Migráció: a relációs modell változásának leírása C# kódban az előző migrációhoz képest
  - Pl. hozz létre egy új oszlopot *DateOfBirth* névvel, *datetime2* típussal a *Dog* táblában
  - Neve kell legyen, névvel tudunk rá hivatkozni
- Adattartalom kezelést is támogat, pl. sorok beszúrását, törlését
- Migrációk gyűjteménye: általában egy könyvtárban csomó C# kódfájl
- Szinte teljesen EF Core tools parancssori eszközökkel kezeljük, csak apró módosításokat csinálunk néha kézileg
  - A parancsok a projektünket futtatják, használják
- A migrációk egymásra épülnek!
- Egy migráció a visszavonás logikát is tartalmazza: hogyan lehet visszatérni az előző migráció állapotához
  - Pl. Dobd el a *DateOfBirth* oszlopot a *Dog* táblában
- Az adatbázisban érvényesített migrációkat magában az adatbázisban is jegyzőkönyvezi egy technikai táblában

EF Core Migrations főbb műveletek

- Zárójelben a Nuget Package Manager Console parancsok
- Generálj a projektben lévő aktuális entitásmodell alapján migrációt (**Add-Migration**)
- Legutóbbi migráció törlése a kódból (**Remove-Migration**)
- Adatbázismodell módosítása (**Update-Database**)
  - Adatbáziskapcsolat szükséges!
  - Magasszintű adatbázis jogosultság szükséges!
  - Célmigráció nevét lehet megadni, alapértelmezett a legutóbbi migráció
  - Összes migráció futtatása, ami még nem volt érvényesítve
  - Összes migráció futtatása, de csak X nevű migrációig
  - Összes migráció visszavonása (adatbázis kiürítése), célmigráció neve: „0”
  - Összes migráció visszavonása X nevű migrációig
- Adatbázismodell módosító szkript generálása (**Script-Migration**)
  - Adatbáziskapcsolat nem szükséges!
  - A kezdő és cél migráció is megadandó
  - Valakinek a szkriptet futtatni kell (tipikusan DB adminisztrátor)

EF Core Reverse Engineering

- Enitásmodellt generál meglévő adatbázis-modellből (**Scaffold-DbContext**)
  - Adatbáziskapcsolat
  - Generál: entitiás osztályok + DbContext + mapping kód
- Akkor jó, ha még nincs entitásmodellünk
  - Összefésülésre nincs támogatás ☹
- Általában egyszer használjuk, utána áttérünk a Migrations alapú változáskövetésre

Mapping / leképezés

- Melyik entitásmodell-elem melyik, milyen relációsmodell-elemnek felel meg
  - > Lásd ORM-ek általános logikája
- A modell létrehozásakor nem adtunk meg explicit hivatkozásokat, a táblák, külső kulcsok neveit, hogy melyik navigációs tulajdonság melyik kulcshoz tartozik, mik az egyedi értékek stb.
- Ilyenkor az EF konvenciók alapján „kitalál” egy leképezést
- Ha valamelyik része nem tetszik, kódban felülbírálhatjuk, kiegészíthetjük

Konvenciók

- Néhány hasznos beépített konvenció:
  - `Id` vagy `{TEntity}Id`: autoinkrementált/adatbázis által generált kulcs (GUID, int/long)
  - `{TEntity}Id` vagy `{TEntity}{PKProperty}Id`: automatikus idegen kulcs + index a hivatkozott entitásra
  - `ICollection<TEntity>` típusú tulajdonság: függő entitások (FK/navigation property) a függő entitásban
  - A `DbContext` megfelelő `DbSet` tulajdonsága vagy az entitástípus neve a DB tábla neve
  - Az entitástípus tulajdonsága a DB tábla mezőjének neve
  - Automatikus nullabilitás `?`-es típusoknál
  - Automatikus DB típusmegfeleltetés (pl. `string` → `NVARCHAR(max)`)
  - Automatikus kaszkád törlés nem nullozható kapcsolatokon

Egyedi modellkonfiguráció

- Ha az automatikus megoldástól el szeretnénk térni, ezek a lehetőségeink:
  - Attribútumok használata
  - Fluent API használata
  - (EF Core 7-től testre szabhatjuk a konvenciókat)

Egyedi modellkonfiguráció – attribútumok

```
public class Dog
{
    // Megadhatjuk, hogy az oszlopot az adatbázis generálja, számítja, vagy kézzel adjuk meg.
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    [Key] // Az entitás kulcsa
    public int Id { get; set; }

    [StringLength(32)] // A kutya neve maximum 32 karakter hosszú lehet, így NVARCHAR(32) típust kap.
    [Column("DogName")] // Az adatbázis oszlop nevét adhatjuk vele meg.
    public string Name { get; set; }

    [Required] // Az oszlop/tulajdonság kötelező (de a modellben null értéket is felvehet majd).
    public DateTime? BirthDate { get; set; }

    [InverseProperty(nameof(DogOwnership.Dog))] // A navigáció másik oldalát jelezzük vele
    public ICollection<DogOwnership> DogOwnerships { get; set; }

    [NotMapped] // Ez a tulajdonság nem lesz az adatbázistábla része.
    public int Barks { get; set; }

    [Timestamp] // Ez a tulajdonság fogja jelezni az adatsor verzióját, ami minden módosításkor frissül.
    [ConcurrencyCheck] // Ezt az oszlopot konkurenciakezeléshez fogjuk alkalmazni.
    public byte[] RowVersion { get; set; }

    // Konstruktor

}
```

Egyedi modellkonfiguráció – Fluent API

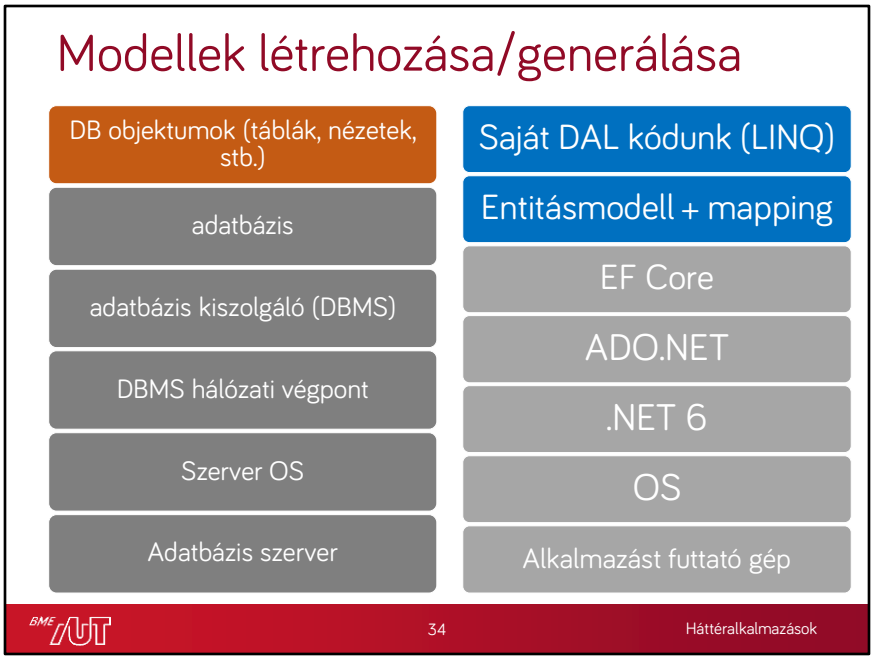
- Az attribútumokkal néhány metaadatot beállíthatunk, de nem teljeskörűen
- Érdemes ehelyett a Fluent API-t használni, ami a teljes konfigurációt elérhetővé teszi számunkra.

Egyedi modellkonfiguráció – Fluent API

```
public class DogFarmDbContext : DbContext
{
    /* ... */
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Dog>(dog => // A dog entitást konfiguráljuk.
        {
            dog.HasKey(d => d.Id); // Az entitás kulcsa.

            dog.Property(d => d.Id)
                .UseIdentityColumn(); // Az oszlopot az adatbázis generálja, számítja, vagy kézzel adjuk meg.

            dog.Property(d => d.Name)
                .HasColumnName("DogName") // Az adatbázis oszlop nevét adhatjuk vele meg.
                .HasMaxLength(32); // A kutya neve maximum 32 karakter hosszú lehet, így NVARCHAR(32) típust kap.

            dog.Property(d => d.BirthDate)
                .IsRequired(); // Az oszlop/tulajdonság kötelező (de a modellben null értéket is felvehet majd).

            dog.Property(d => d.RowVersion)
                .IsRowVersion() // Az adatsor verziója, ami minden módosításkor frissül.
                .IsConcurrencyToken(); // Ezt az oszlopot konkurenciakezeléshez fogjuk alkalmazni.

            dog.Ignore(d => d.Barks); // Ez a tulajdonság nem lesz az adatbázistábla része.

            dog.HasMany(p => p.DogOwnerships) // A navigáció másik oldalát jelezzük vele...
                .WithOne(o => o.Dog) // ...megadva, hogy melyik tulajdonság ez a kapcsolat a másik oldalon...
                .HasForeignKey(o => o.DogId) // ...mit használunk a külső kulcs tárolására...
                .HasPrincipalKey(d => d.Id); // ...és melyik mezőre mutat itt a külső kulcs.
        });
    }
}
```

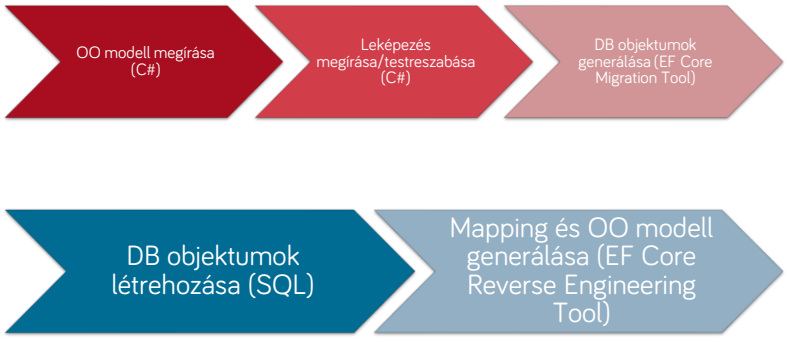
Egyedi modellkonfiguráció – Fluent API

- Sokkal több lehetőség:
  - Kapcsolatok
  - Alternatív kulcsok definíciója
  - Nézetek, egyedi eljárások definíciója
  - Ősfeltöltés (seed) támogatása, adatok betöltése
  - Adatbázis-specifikus beállítások
    - Pl. SQL Server memóriabeli optimalizációk
  - Annotációk és kommentek hozzáfűzése az adatbázishoz
  - Explicit DB adattípus megadás
  - Kétirányú átalakítás (pl. enum <-> string)
  - Alapértékek konfigurációja
  - Hi-Lo támogatás
  - ...és még sok más... (https://learn.microsoft.com/en-us/ef/core/modeling/)



EF Core, ADO.NET összevonva-egyszerűsítve jelenik meg itt már

Code-First vs DB-First



Adatmódosító műveletek

DbContext entitásnyilvántartás

- Minden adatbázis művelet (lekérdezés, hozzáadás, törlés, módosítás - CRUD) egy DbContext példányon keresztül történik
- A DbContext példány a nyilvántartást vezet entitáspéldányokról
- A DbContext példány létrejöttekor üres a nyilvántartás
- Nyilvántartja az entitáspéldány állapotát (törölt, hozzáadott, változott), és az eredeti (bekerüléskori) adattartalmát
  - Ez utóbbi alapján ki lehet számolni, hogy egy entitás változott-e
- Bekerül egy példány
  - Implicit, ha a DbContext példányon keresztül lekérdeztük adatbázisból
  - Explicit, ha függvény meghívásával regisztráljuk
    - Add – regisztrál és hozzáadott állapotúként jelöl
    - Remove - regisztrál és törölt állapotúként jelöl
    - Attach – csak regisztrál
- DbContext.ChangeTracker property-n keresztül érhető el a nyilvántartás
- Egy entitásról nyilvántartott adatok: DbContext.Entry(entitás)
- A SaveChanges függvény használja, innen tudja, hogy melyik entitással mit kell csinálni

Létrehozás

```
private static void AddDogToDatabase(Dog newDog)
{
    using var dbContext = new DogFarmDbContext();
    dbContext.Dogs.Add(newDog);
    dbContext.SaveChanges();
}
```

- A `using` kulcsszó segítségével példányosított `IDisposable` objektumunkon (`DbContext`) a blokkon kívül érve automatikusan meghívódik a `.Dispose()` metódus, mely bontja az adatbáziskapcsolatot
- A `SaveChanges()` hívás a detektált módosításokat (ide értve a létrehozást és törlést) elküldi SQL formájában az adatbázisnak, melyet tranzakcióban futtat
- Az entitás elsőleges kulcs property-jét nem szoktuk beállítani, általában nincs értelme (adatbázis osztja ki az értéket)

Módosítás

```
private static void UpdateDogBirthDate(int dogId, DateTime? birthDate)
{
    using var dbContext = new DogFarmDbContext();
    var dog = dbContext.Dogs.Find(dogId);
    dog.BirthDate = birthDate;
    dbContext.SaveChanges();
}
```

- A módosítandó elemet előbb le kell kérdezni vagy explicit nyilvántartásba kell vetetni (Attach/Update) a Dbcontext-tel
- Ekkor a *DbContext* eltárolja az ekkori állapotát
- Szabadon módosíthatjuk az entitásokat C# kóddal
- A SaveChanges híváskor összeveti az eltárolt állapotot az aktuálissal

Törlés

```
private static void DeleteDog(int dogId)
{
    using var dbContext = new DogFarmDbContext();
    var dog = dbContext.Dogs.Find(dogId);
    dbContext.Dogs.Remove(dog);
    dbContext.SaveChanges();
}
```

- Szintén példányra van előbb szükség
- A Remove hívás
  - nyilvántartásba veszi a dog példányt, bár a lekérdezés miatt ezesetben már nem kell
  - megjelöli törlendőként

Lekérdezések

Legegyszerűbb(?) lekérdezés

```
private static void PrintDogs()
{
    using var dbContext = new DogFarmDbContext();
    List<Dog> dogs = dbContext.Dogs.ToList();
    if (dogs.Count == 0)
    {
        Console.WriteLine("No dogs are stored in the database.");
    }
    else
    {
        foreach (var dog in dogs)
            Console.WriteLine(dog);
    }
    Console.WriteLine();
}
```

Összetett lekérdezések

- **DbSet<T>**-ből (pl. DbContext.Dogs) kiinduló LINQ kifejezések, függvényhívási láncok
- Az `IQueryable<T>` (amilyen a `DbSet<T>` is) minden LINQ operátort támogat
  - IQueryable<T> az IEnumerable<T> leszármazottja
  - Képes a LINQ kifejezéseket SQL-lé alakítani
- Gyakorlatilag minden LINQ kódunk lefordul
  - DE! Nem mindegyik fog lefutni. Nem minden LINQ kifejezés alakítható SQL-lé!
  - Mindig próbáljuk ki, hogy kivételt kapunk-e?

LINQ

- Projekció:
```
Dogs.Select(d => d.Name)
```
- Csoportos projekció + lapítás:
```
.SelectMany(d => d.Ownerships)
```
- Szűrés:
```
.Where(d => d.Name.Length == 4)
```
- Csoportosítás, számolás:
```
.GroupBy(d =>
d.Ownerships.Count())
```
- Rendezés és ablakozás:
```
.OrderBy(d =>
d.Ownerships.Count())
    .ThenBy(d => d.BirthDate)
.Skip(20)
.Take(10)
```
- Első/legfeljebb első/egyetlen, legfeljebb egyetlen találat:
```
.Single(d => d.Name == "Bodri")
.SingleOrDefault(d => d.Name ==
"Bodri")
.First (d => d.Name == "Bodri")
.FirstOrDefault (d => d.Name ==
"Bodri")
```
- Bármely/mind:
```
.Any(d => !d.Ownerships.Any())
.Any(d => d.Ownerships
    .All(o => o.Person
                .Name == "Béla"))
```

Összetett lekérdezések

```
// Azok a nem árva kutyák, akik kizárólag Béla nevű ember tulajdonában vannak.
var q1 = dbContext.Dogs
                  .Where(d => d.DogOwnerships.Any() && d.DogOwnerships
                                                        .All(a => a.Person.Name == "Béla"));

// A tulajdonlások személyenként csoportosítva, ahol legalább 2 kutyája van egy embernek.
var q2 = dbContext.DogOwnerships
                  .GroupBy(o => o.Person)
                  .Where(g => g.Count() > 2);

// Az összes kutyanév (a duplikátumok kiszűrve), akiknek gazdája van.
var q3 = dbContext.People
                  .SelectMany(p => p.DogOwnerships
                                    .Select(o => o.Dog.Name))
                  .Distinct();

// Azok a gazdátlan kutyák, akik több, mint 5 évesek.
var fiveYearsAgo = new DateTime(DateTime.Now.Year - 5, DateTime.Now.Month, DateTime.Now.Day);
var q4 = dbContext.Dogs
                  .Where(d => d.BirthDate < fiveYearsAgo && !d.DogOwnerships
                                                             .Any());

// Az összes lehetséges gazdátlan kutya és kutyátlan ember párja (Descartes-szorzat).
var q5 = dbContext.Dogs
                  .Where(d => !d.DogOwnerships.Any())
                  .SelectMany(d => dbContext.People
                                            .Where(p => !p.DogOwnerships.Any())
                                            .Select(p => new { Dog = d, Person = p }));
```

Kiértékelés ideje

- Mikor hajtódik végre a lekérdezés?
  - Csak amikor fel kell használni az eredményt
  - Ciklusban (for, foreach) végigmegyünk az eredménykollekción
  - Azonnali végrehajtású LINQ operátorokat használunk, pl. ToList, ToArray, Single, Count
    https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/classification-of-standard-query-operators-by-manner-of-execution#classification-table

Kapcsolódó elemek betöltése

- Egy entitáspéldány lekérdezésekor a kapcsolódó elemek, a navigációs propertyk alapértelmezésben nem töltődnek fel, null értékűek maradnak
  - Kivéve, ha a kapcsolódó elem már korábban le lett kérdezve és a DbContext nyilvántartásában van
- Eager loading: betöltethetjük a lekérdezésben az *Include* operátorral
- Explicit loading: explicit külön függvényhívással DbContext.Entry(dog).Collection(d=>d.DogOwnerships).Load()
- Lazy loading: amikor a kód miatt szükség van a betöltésre, automatikusan lekérdezi az EF

Amiről nem volt szó…

- Hogyan működik belül az SQL-re fordítás
- Query syntax
- Lazy loading, proxy-k
- Kliensoldali kiértékelés
- Kapcsolótábla modellezés
- Kulcstalan entitások
- Birtokolt entitások
- Tárolt eljárások, nézetek
- Egyedi SQL futtatás migrációkor
- Párhuzamos lekérdezések
- Konkurenciakezelés
- NoTracking
- Aszinkron LINQ operátorok
- Darabolt lekérdezések
- Kötegelés
- Lekérdezés optimalizálás
- Gyakori hibák
- Leszármazás modellezés
- Tranzakciók
- Interceptorok
- Értékkonverziók
- Modellszintű szűrők
- Típusonkénti modellkonfiguráció
- Előre fordított lekérdezések
- … és még sok minden más…