

Egyszerű adatbázis létrehozása Entity Framework segítségével

Az Entity Framework (EF) a Microsoft által fejlesztett Object-Relational Mapper (ORM) .NET-hez. Ami azt jelenti, hogy az Entity Framework egy átalakító réteg az objektumorientált kód (C# osztályok) és a relációs adatbázis (táblák, rekordok, oszlopok) között.

Több koncepció létezik a műveletek irányát tekintve.

- **Már létezik adatbázis** és ehhez szeretnénk valamilyen kezelőfelületet biztosítani
- **Még nem létezik adatbázis** (zöld mezős beruházás), most hozzuk létre, de úgy, hogy a szerkezetét a későbbiekben még alakíthatjuk, adhatunk hozzá új táblákat, mezőket stb.
- **Még nem létezik adatbázis**, most hozzuk létre, de úgy, hogy a szerkezetét a későbbiekben **NEM tudjuk módosítani!** (egyszerűbb létrehozni, mint az előzőt, fejlesztéshez, teszteléshez használjuk, élesben nem!!!)

Először nézzük a legegyszerűbb esetet, amikor friss, nem módosítható adatbázist hozunk létre.

Először létre kell hozni a **táblának megfeleltetett osztályokat**. Elsőnek egy 1 táblás adatbázist hozunk létre (legegyszerűbb, könyveket tárolunk benne) Az osztály **PUBLIC** láthatóságú!

```
2 references
public class Konyv
{
    [Key]
    0 references
    public int Id { get; set; }
    1 reference
    public string Cim { get; set; }
    1 reference
    public string Szerzo { get; set; }
    1 reference
    public int Oldalszam { get; set; }

    0 references
    public Konyv(string cim, string szerzo, int oldalszam)
    {
        Cim = cim;
        Szerzo = szerzo;
        Oldalszam = oldalszam;
    }
}
```

A [Key] attribútum az Entity Frameworknek szóló jelzés (annotation). Azt jelenti, hogy az elsődleges kulcs a táblában az **Id** mező lesz! Kell hozzá a *System.ComponentModel.DataAnnotations* névtér is:

```
using System.ComponentModel.DataAnnotations;
```

Ha nem tesszük ki a [Key] attribútumot az EF akkor is elsődleges kulcsnak veszi az Id mezőt!!!

Tehát a lentebb látható osztály egyenértékű a fentebb látható osztállyal (és itt nem is kell a *System.ComponentModel.DataAnnotations* névtér!)

```
2 references
public class Konyv
{
    0 references
    public int Id { get; set; }
    1 reference
    public string Cim { get; set; }
    1 reference
    public string Szerzo { get; set; }
    1 reference
    public int Oldalszam { get; set; }

    0 references
    public Konyv(string cim, string szerzo, int oldalszam)
    {
        Cim = cim;
        Szerzo = szerzo;
        Oldalszam = oldalszam;
    }
}
```

Ha viszont **NEM Id** nevet adunk az elsődleges kulcsunknak, akkor jelölni **KELL**, hogy ez a mező lesz az elsődleges kulcs, ennek a jelölésnek az egyik módja a **[Key]** attribútum.

Kész a **Konyv** osztályunk, amiből tábla lesz az adatbázisban.

Most csináljuk meg azt az osztályt, amely az adatbázissal fog „beszélgetni”. Ez az EF egyik legfontosabb osztálya, összekapcsolja az alkalmazásunkat az adatbázissal! A neve megegyezés szerint vagy az adatbázis nevével kezdődik és **DbContext**-re végeződik vagy **AppDbContext**! Jelen esetben hozzuk létre a **KonyvDbContext** osztályt. Ez **MINDIG** a **DbContext** osztályból származik (onnan örököl), a **DbContext**-hez pedig kell az **Entity Framework Core** NuGet csomag is! A szükséges csomagokat telepíthetjük úgy is, hogy bemásoljuk a projektfájlba (**projektneve.csproj**) a megfelelő sorokat a következőkből:

```
<ItemGroup>
  <PackageReference Include="Microsoft.EntityFrameworkCore" Version="8.0.21" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite"
Version="8.0.21" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
Version="8.0.21" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="8.0.21">
    <PrivateAssets>all</PrivateAssets>
    <IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>
  </PackageReference>
  <PackageReference Include="MySql.EntityFrameworkCore" Version="8.0.20" />
</ItemGroup>
```

Itt a 8-as .NET-et használó projektekhez van a minta! A 10-el egyelőre bánunk óvatosan! A **Tools** csomag a migrációhoz kell, míg az **SqlServer** végű az **MSSQL**-hez

```
namespace Konyedb1
{
  0 references
  public class KonyvDbContext:DbContext
  {
    ...
  }
}
```

Ebben az osztályban hozzuk létre a táblákat a **DbSet<T>** gyűjtemény segítségével, amely az adatbázis egy tábláját képviseli az alkalmazásban. Valójában a **DbSet** egy tábla logikai leképezése (mapping) az adatbázisból, a típusparamétere **<T>** pedig az adott entitásosztály, ami egy sor (rekord) szerkezetét írja le. Itt így valósítjuk meg:

```
0 references
public class KonyvDbContext:DbContext
{
  0 references
  public DbSet<Konyv> Konyvek { get; set; }
```

- Az adatbázisban lesz egy **Konyvek** nevű tábla
- minden sora egy **Konyv** objektumnak felel meg
- A **Konyvek** property-n keresztül tudjuk lekérdezni, beszúrni, módosítani vagy törölni az adatokat.

Azért jó ez nekünk, mert a **DbSet<T>** ugyanúgy viselkedik, mint egy lista **List<T>** (használhatunk **LINQ-t**), **de mögötte adatbázis műveletek állnak**.

Ha készen vannak a táblák, akkor hozunk lére egy konstruktort az osztályban, ami létrehozza az adatbázist a megfelelő táblákkal, ha még nem létezett az adatbázis! Ha létezik, akkor nem csinál semmit!

```
public class KonyvDbContext : DbContext
{
    0 references
    public DbSet<Konyv> Konyvek { get; set; }
    0 references
    public KonyvDbContext()
    {
        this.Database.EnsureCreated();
    }
}
```

Ezután adjuk hozzá a kapcsolati sztringet és döntsük el, hogy milyen SQL-ben szeretnénk adatbázist létrehozni?! Az adatbázismotor kiválasztása után telepíteni kell a megfelelő csomagot.

MySQL esetén vagy a *MySql.EntityFrameworkCore* vagy a *Pomelo.EntityFrameworkCore.MySql*

MSSQL esetén a *Microsoft.EntityFrameworkCore.SqlServer*

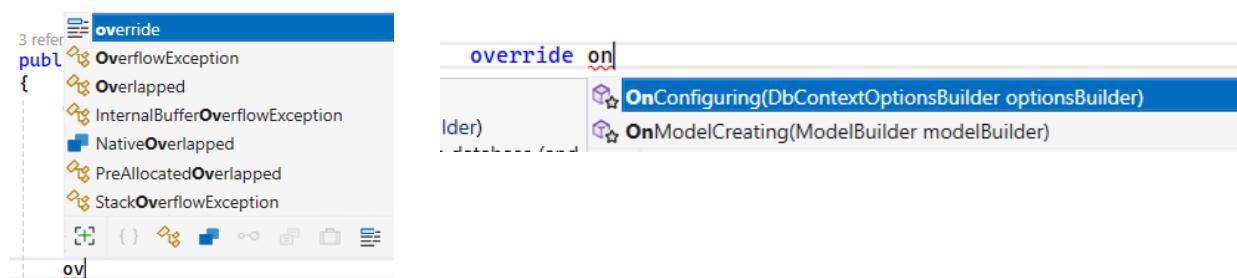
SQLite esetén a *Microsoft.EntityFrameworkCore.SQLite* stb...

Ezeket **provizereknek** hívjuk. Ők azok a komponensek, amik „megtanítják” az EF Core-t arra, hogyan kommunikáljon az adott adatbázissal (SQL Server, SQLite, PostgreSQL, MySQL stb.).

A provider fő feladatai:

- a LINQ-lekérdezések ből SQL-t generáljon az adott adatbázis dialektusában,
- az adatbázis-kapcsolatot kezelje,
- az adatbázis műveleteket (CREATE, INSERT, UPDATE, DELETE, SELECT) végrehajtsa,
- az EF Core modelltáblákat leképezze az adatbázisra.

Ha telepítettük a megfelelő NuGet csomagot, akkor a **KonyvDbContext** osztályban felül kell írni az űrosztály (**DbContext**) egy metódusát. Úgy könnyű megjegyezni, hogy *on*-al kezdődik:



A felülírandó metódus a kapcsolati sztring megadásához az **OnConfiguring!** A kapcsolati string lokális szerver esetén nem tartalmazza a portszámot és a jelszó a Xampp alapértelmezése szerint **NINCS** beállítva a **ROOT** felhasználóhoz. **ILYET SOHASE CSINÁLJUNK ÉLESBEN!!! Mindig legyen jelszó és ne a root-tal lépjünk be!!!**

MySQL esetén így néz ki, ha a *MySql.EntityFrameworkCore*-t használjuk:

```
3 references
public class KonyvDbContext : DbContext
{
    0 references
    public DbSet<Konyv> Konyvek { get; set; }
    1 reference
    public KonyvDbContext()
    {
        this.Database.EnsureCreated();
    }

    0 references
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseMySQL("server = localhost; database = uj_db_nev;uid=root;password = ''");
    }
}
```

És így, ha a *Pomelo.EntityFrameworkCore.MySql*-t (különbség az előzőhöz képest: MySQL <=> MySQL)

```
var connectionstring = "server = localhost; database = uj_db_nev;uid=root;password = ''";
var serverversion = new MySqlServerVersion(new Version(10, 4, 32)); // ha te adod meg kézzel
// MySQL-ben lekérdezheted a SELECT version(); parancssal!
optionsBuilder.UseMySQL(connectionstring, serverversion);
```

A MySQL server verziószáma a nyelvjárás pontos ismerete miatt kell. De használhatsz automatikus felismerést is:

```
optionsBuilder.UseMySQL("server = localhost; database = uj_db_nev;uid=root;password = ''",
    ServerVersion.AutoDetect("server = localhost; database = uj_db_nev;uid=root;password = ''"));
```

MSSQL esetén így néz ki (csak a sztring brutál, azt majd órán mutatom)

optionsBuilder.UseSqlServer("Data Source":

SQLite esetén így néz ki:

optionsBuilder.UseSqlite("Data Source=konyvek111.db");

Ha ezzel készen vagyunk és már égünk a vágytól, hogy legyen adatbázisunk, akkor nosza. A főprogramban létre kell hoznunk egy példányt a **KonyvDbContext** osztályból => lefut a konstruktorra => létrehozza az üres adatbázist, ha még nem létezett! És kész!

```
namespace Konydb1
{
    0 references
    internal class Program
    {
        This method has 0 reference(s).
        0 references
        static void Main(string[] args)
        {
            KonyvDbContext ctx = new KonyvDbContext();
        }
    }
}
```

Vigyünk adatokat a friss adatbázisunkba. Mivel egyetlen tábla van benne, ezért ez a parancs így végrehajtódik, de jobb, ha megszokjuk, hogy mindenki kiírja a tábla nevét is, amibe adatokat akarunk vinni, valamint fontos a végén a mentés:

```

static void Main(string[] args)
{
    KonyvDbContext ctx = new KonyvDbContext();
    ctx.Add(new Konyv("Vuk", "Fekete István", 215));
    ctx.Add(new Konyv("Kele", "Fekete István", 355));
    ctx.Add(new Konyv("Egri csillagok", "Gárdonyi Géza", 587));
    ctx.SaveChanges();
}

```

Lefut, de nem igazán jó!

A jó megoldás:

```

KonyvDbContext ctx = new KonyvDbContext();
ctx.Konyvek.Add(new Konyv("Vuk", "Fekete István", 215));
ctx.Konyvek.Add(new Konyv("Kele", "Fekete István", 355));
ctx.Konyvek.Add(new Konyv("Egri csillagok", "Gárdonyi Géza", 587));
ctx.SaveChanges();

```

Kérdezzük le a 300 oldalnál vastagabb könyveket:

```

var vastag = ctx.Konyvek.Where(x => x.Oldalszam > 300).Select(x=>x.Cim);
Console.WriteLine(String.Join(", ", vastag));

```

Figyeljünk arra, hogy ahányszor futtatjuk így a programunkat, annyiszor adja hozzá a műveket!!!

Ez se túl jó hír. Egy lehetséges megoldás memóriakezeléssel:

```

using (var ctx = new KonyvDbContext())
{
    if (!ctx.Konyvek.Any(k => k.Cim == "Vuk"))
        ctx.Konyvek.Add(new Konyv("Vuk", "Fekete István", 215));

    if (!ctx.Konyvek.Any(k => k.Cim == "Kele"))
        ctx.Konyvek.Add(new Konyv("Kele", "Fekete István", 355));

    if (!ctx.Konyvek.Any(k => k.Cim == "Egri csillagok"))
        ctx.Konyvek.Add(new Konyv("Egri csillagok", "Gárdonyi Géza", 587));

    ctx.SaveChanges();

    var vastag = ctx.Konyvek.Where(x => x.Oldalszam > 300).Select(x => x.Cim);
    Console.WriteLine(String.Join(", ", vastag));
}

```

Adjunk hozzá egyszerre **SOK REKORDOT**, amelyek egy listában vannak:

ctx.Konyvek.AddRange(Lista neve) majd az adatbázis mentése

Töröljünk egy rekordot:

```

// Törlés
var torlendokonyv = ctx.Konyvek.FirstOrDefault(x => x.Cim == "Kele");
if (torlendokonyv != null)
{
    ctx.Konyvek.Remove(torlendokonyv);
    ctx.SaveChanges();
}

```

Töröljünk több rekordot:

```
// Több rekord törlése
var torlendoKonyvek = ctx.Konyvek
    .Where(k => k.Szerzo == "Fekete István")
    .ToList();
ctx.Konyvek.RemoveRange(torlendoKonyvek);
ctx.SaveChanges();
```

Nézzünk még át pár apró, de fontos dolgot.

Ha könyveket teszünk adatbázisba, akkor az elsődleges kulcs általában NEM szám, hanem valamilyen hosszú sztring. Ezt így oldjuk meg EF-ben:

```
public class Konyv
{
    [Key]
    public string Id { get; set; }

    [Required]
    public string Cim { get; set; }

    [MaxLength(50)]
    public string Szerzo { get; set; }

    [Range(1, int.MaxValue)]
    public int Oldalszam { get; set; }

    public Konyv(string cim, string szerzo, int oldalszam)
    {
        Id = Guid.NewGuid().ToString();
        Cim = cim;
        Szerzo = szerzo;
        Oldalszam = oldalszam;
    }
}
```

Fontos tudni, hogy az adatannotációk (*data annotations*) segítségével közvetlenül a model osztályban (itt a **Konyv**) meg tudjuk adni az adatbázisbeli szabályokat, korlátozásokat — pl. mi legyen kötelező, mennyi legyen a maximális hossz, mi az elsődleges kulcs stb.

Maga az annotáció attribútumot jelent, amit szögletes zárójelben [] írunk a property előre. Ezeket minden a *System.ComponentModel.DataAnnotations* névterből importáljuk, ahogy már fentebb említettem. Fontos tudni, hogy megszorításokat nem csak annotációk, hanem **Fluent API-k** segítségével is létrehozhatunk (később nézzük)

Pár fontos annotáció:

- **[Key]** Elsődleges kulcs (Primary Key)
- **[Required]** Kötelező mező (NOT NULL)
- **[MaxLength(n)]** Maximális hossz (varchar(n))
- **[MinLength(n)]** Minimum hossz (ellenőrzés)
- **[StringLength(n)]** Maximális hossz (ugyanaz, mint MaxLength, de EF + UI-hoz is jó)
- **[Range(min, max)]** Értéktartomány numerikus mezőknél
- **[ForeignKey("Nev")]** Idegen kulcs (kapcsolat másik táblához)
- **[Column("Nev", TypeName="varchar(100)")]** Oszlopnev és típuskényszer az adatbázisban
- **[Table("Nev")]** Tábla neve az adatbázisban
- **[Index(nameof(Mezo), IsUnique = true)]** Egyedi index létrehozása (EF Core 5+) Csak a *Microsoft.EntityFrameworkCore* névtér usingolása után működik!!!

Adjunk valamilyen nevet az adatbázis tábláinak:

```
[Table("Dezso")]
5 references
public class Konyv
{
    1 reference
    public string Id { get; set; }
    6 references
    public string Cim { get; set; }
```

Szabályozhatjuk, hogy milyen hosszú lehet egy sztring típusú mező:

```
[StringLength(30)]
6 references
public string Cim { get; set; }
```

Megadhatjuk kötelezőnek a címet és minimum 2 karakter hosszúnak max. 50 karakteresnek így:

```
[Required]
[MinLength(2)]
[StringLength(50)]
6 references
public string Cim { get; set; }
```

vagy így (ezt használjuk inkább!)

```
[Required]
[StringLength(50, MinimumLength = 2)]
6 references
public string Cim { get; set; }
```

Adunk tartományt az oldalszámokra:

```
[Range(10, 5000)]
2 references
public int Oldalszam { get; set; }
```

FONTOS, hogy egy mező **NEM** lesz kötelező csak azért, mert van minimális hossz vagy tartomány beállítva rajta! És ez a **Range** korlátozás csak a **C#** felől jelent megkötést, innen nem tudunk az adatbázisba tölteni a tartományon kívüli értéket, az adatbázison magán semmilyen korlátozás nem jön létre ettől. Viszont **Fluent API**-val az adatbázison működő korlátozást is be lehet állítani! Ráadásul a **Fluent API** beállításai felülírják az annotációkat!!!

Végezetül nézzük meg az egyediséget (**UNIQUE**), ami sokáig csak **Fluent API**-val volt beállítható.

FONTOS, hogy ezt az attribútumot mindenkor az **OSZTÁLYRA** kell tenni és nem a **PROPERTY-re**!!!

```

[Table("Dezso")]
[Index(nameof(Cim), IsUnique = true)]
5 references
public class Konyv
{
    1 reference
    public string Id { get; set; }

    [Required]
    [StringLength(50, MinimumLength = 2)]
    7 references
    public string Cim { get; set; }

```

Több mezőt is egyedivé lehet tenni:

```

[Table("Dezso")]
[Index(nameof(Cim), nameof(Szerzo), IsUnique = true)]
5 references
public class Konyv
{
    1 reference
    public string Id { get; set; }

    [Required]
    [StringLength(50, MinimumLength = 2)]
    7 references
    public string Cim { get; set; }
    2 references
    public string Szerzo { get; set; }
    [Range(10, 5000)]
    2 references
    public int Oldalszam { get; set; }

```

Adjunk meg pár Seed adatot (az adatbázis létrehozásakor adunk pár rekordot a táblához, nem üres tábla jön létre, hanem mintaadatokat tartalmaz)

A kezdeti adatokat a **KonyvDbContext** osztályban tudjuk megadni, override-olnunk kell a *OnModelCreating* metódust. Onnan könnyű megjegyezni, hogy ez is **ON**-al kezdődik!

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
}

```

base.OnModelCreating(modelBuilder); - utasítást hagyjuk benne, **NE TÖRÖLJÜK KI!**

Itt kell majd megadni a **Fluent API** utasításokat is!!!

A seed adatok létrehozása:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    modelBuilder.Entity<Konyv>().HasData(
        new Konyv { Id = Guid.NewGuid().ToString(), Cim = "Vuk", Szerzo = "Fekete István", Oldalszam = 215 },
        new Konyv { Id = Guid.NewGuid().ToString(), Cim = "Kele", Szerzo = "Fekete István", Oldalszam = 355 },
        new Konyv { Id = Guid.NewGuid().ToString(), Cim = "Egri csillagok", Szerzo = "Gárdonyi Géza", Oldalszam = 587 }

    );
}
```

FONTOS dolgok:

- a **Konyv** osztályban kell egy **ÜRES** kondtruktor is, hogy működjön az objektum létrehozása így
- az **Id** mező **NEM HAGYHATÓ EL**, hiába adott az **AUTO_INCREMENT!** Ha számokat használtam volna (már nem volt kedvem átírni), akkor 1,2,3 lett volna ebben a sorrendben.
- nem annyira fontos, de érdemes megjegyezni: az **Id** szerint lesznek rendezve a rekordok, viszont az **Id** véletlenszerűen jön itt létre, ami azt jelenti, hogy az adatbázisban nem biztos az itt deklarált sorrend!