Feladatmegoldások Fluent API segítségével 1:N

Vegyük a múltkor létrehozott könyves adatbázist, amelyben 2 tábla található: a **Konyv** és a **Szerzo**. A táblák között **1:N** kapcsolat van, **1** könyvnek csak **1** szerzője lehet, de **1** szerzőnek **N** db könyve is lehet. Hozzuk létre a következő beállításokat a **Fluent API** segítségével (eddig annotációval csináltuk):
(ne feledjük, hogy az **IDEGEN KULCSOKAT** az **EF** automatikusan felismeri, ha a property neve : „*HIVATKOZOTT TÁBLA NEVE + Id*” itt pl. **SzerzoId**, tehát erre **NEM KELLENE** külön **FluentAPI** beállítás, de mi itt gyakorlásból beállítjuk!)

```
0 references
public class Konyv
{
    0 references
    public int Id { get; set; }

    [Required]  // not null
    [StringLength(100)]  // max 100 karakter hosszú lehet
    0 references
    public string Cim { get; set; }

    // Külső kulcs az egyik módon használva
    [ForeignKey("Szerzo")]
    0 references
    public int SzerzoId { get; set; }

    [Required]
    [Range(1, 200)]
    0 references
    public int Helyezes { get; set; }

    // Navigációs tulajdonság, a segítségével a Könyv osztályban is megismerhetjük a Szerzőt!
    0 references
    public Szerzo Szerzo { get; set; }
}
```

Mielőtt részletesen tárgyalnánk **string** típus esetén a **KÖTELEZŐ** mezőket (**NOT NULL**), tudnunk kell, hogy az **EF Core 5** előtt a string alapból **nullable volt az adatbázisban** (**NULL** engedélyezett)!

Viszont az EF Core 5+ ; C# 8-tól létezik a **nullable reference types** (**NRT)**, amely engedélyezésével a string típus már önmagában is biztosítja a **NOT NULL**-t. Ha szeretném megengedni a **NULL**-t akkor **string?** a helyes típusmegjelölés vagy nem engedélyezem az **NRT**-t!.
Az **INT** típus alapból **BIZTOSÍTJA A NOT NULL**-t! Ha kell a **NULL**, akkor **int?**

FONTOS tudni, hogy a **string önmagában NEM biztosítja azt**, hogy az adatbázisban a mező **NOT NULL** lesz!!! A viselkedése az EF Core **nullability beállításaitól** függ (**NRT**), tehát akkor járunk el helyesen, ha explicit módon beállítjuk a **NOT NULL**-t!

Állítsuk be a **NOT NULL**-t a **Cím** mezőre. Az **AppDbContext** osztályban kell felülírni az **OnModelCreating** metódust.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    modelBuilder.Entity<Konyv>()
        .Property(x => x.Cim)
        .IsRequired();
}
```

Állítsuk be a **NOT NULL** mellé a **Cím** maximális hosszát is.

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

    modelBuilder.Entity<Konyv>()
        .Property(x => x.Cim)
        .HasMaxLength(100)
        .IsRequired();
}
```

Ezek mellé állítsuk be, hogy egyedi legyen a **Cím** (**UNIQUE**)

```
modelBuilder.Entity<Konyv>()
    .Property(x => x.Cim)
    .HasMaxLength(100)
    .IsRequired();
modelBuilder.Entity<Konyv>()
    .HasIndex(x => x.Cim)
    .IsUnique();
```

Tartományt megadni csak **Check constraint**-tel tudunk. Viszont ez **NEM** csak **VALIDÁCIÓ**, mint a **RANGE** esetében, hanem valóban érvényesül az adatbázison is. **AZ ANNOTÁCIÓK TÖBBSÉGE ÉRVÉNYESÜL AZ ADATBÁZISON, DE A RANGE NEM!** A **Range**-nél csak a **C#** felől nem tudtunk értéket adni a tartományon kívülről, **magán az adatbázison semmilyen korlátozás sem lesz**! **Fluent API**-val viszont **tényleges adatbázis-szintű védelmet hoztunk létre**, nem csak runtime validációt! **FONTOS**, hogy **MySQL** esetén legalább 8-as verzió kell! Az aktuális verziót a **SELECT version();** paranccsal tudjuk lekérdezni, míg azt megnézni, hogy tényleg rátette-e a megszorítást (a kipróbáláson kívül) a **SHOW CREATE TABLE** *tábla neve*; vagy a **SELECT * FROM** *information_schema.table_constraints* **WHERE** *table_schema* = *'adatbazis_neve'*; paranccsal tudjuk.

```
modelBuilder.Entity<Konyv>()
    .ToTable(x => x.HasCheckConstraint("CK_Helyezes", "Helyezes BETWEEN 1 AND 200"));
```

Állítsuk be az elsődleges kulcsot:

```
modelBuilder.Entity<Konyv>()
    .HasKey(x => x.Id);
```

És akkor már csak a két tábla összekapcsolása maradt hátra. **FONTOS**, hogy a kapcsolatot a **„many" (N) oldalról** általában „egyszerűbb" konfigurálni **(**itt a **Konyv** irányából) és mindig a **navigációs property**-et használjuk az összekapcsolásra.

```
modelBuilder.Entity<Konyv>()
    .HasOne(k => k.Szerzo)
    .WithMany(s => s.Konyvek)
    .HasForeignKey(k => k.SzerzoId);
```

Egy **Könyvnek** egy **Szerzője** van: `.HasOne(k => k.Szerzo)`

Egy **Szerzőnek** sok **Könyve** lehet: `.WithMany(s => s.Konyvek)`

Az **idegen kulcs** (**FK**) a **Konyv** osztályban a **SzerzoId**. `.HasForeignKey(k => k.SzerzoId);`

A **Szerzo** irányából így néz ki a táblák összekapcsolása (pont ugyanazt csinálja, mint az előző, bármelyik jó megoldást ad!):

```
modelBuilder.Entity<Szerzo>()
    .HasMany(s => s.Konyvek)
    .WithOne(k => k.Szerzo)
    .HasForeignKey(k => k.SzerzoId);
```

Egy **Szerzőnek** sok **Könyve** lehet:

```
.HasMany(s => s.Konyvek)
```

Egy **Könyvnek** egy **Szerzője** van

```
.WithOne(k => k.Szerzo)
```

Az idegen kulcs a **Konyv**-ben van (**SzerzoId**)

```
.HasForeignKey(k => k.SzerzoId);
```

Nem marad más hátra, migráljunk és nézzük meg, hogy jól dolgoztunk-e?!

Ha netán hiba lépne fel a migrálás során, akkor a hibáról szóló információkat a **View – Output** útvonalon érhetjük el!

Gyakorlásnak készítsük el a **Könyv – Szerző** táblákat tartalmazó adatbázis **Fluent API** segítségével.

Ha pedig az ment, akkor készítsük el a következő adatbázist **Fluent API** segítségével.

Az adatbázis felépítése a következő:

szakma

id	Szöveg, a szakma azonosítója, PK
szakmaNev	Szöveg, a szakma magyar megnevezése *NOT NULL, UNIQUE*

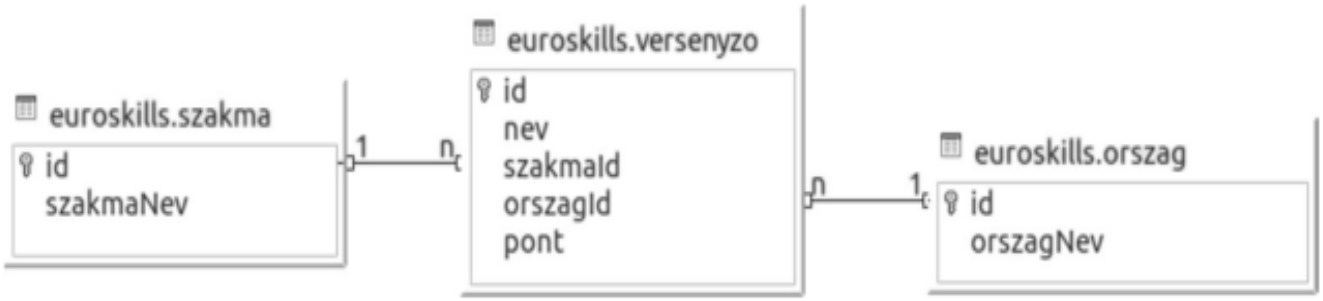
orszag

id	Szöveg, az ország azonosítója, PK
orszagNev	Szöveg, az ország magyar megnevezése *NOT NULL, UNIQUE*

versenyzo

id	Egész szám, a versenyző azonosítója, PK
nev	Szöveg, a versenyző neve *NOT NULL*
szakmaId	Szöveg, a versenyző szakmájának azonosítója, FK
orszagId	Szöveg, a versenyző országának azonosítója, FK
pont	Egész szám, a versenyző által elért pontszám *0-800 közzé eső szám*

Az elsődleges kulcsokat **PK**-val, az idegenkulcsokat **FK**-val jelöltük! Az adattáblák közti kapcsolatokat az alábbi ábra mutatja:



Ha a következő oldalon található mintának megfelelően hozzuk létre az osztályokat, akkor mindenben megfeleltünk az **EF** előírásainak, ami azt jelenti, hogy **Fluent API NÉLKÜL, AUTOMATIKUSAN JÓL** hozza létre a táblákat! Helyes **PK** és **FK** beállításokkal, összekapcsolásokkal! Szóval alapból használjuk ezt a szintaxist, de most gyakorlunk, így állítsuk be: ha töröljük az országot az **Orszag** táblából, akkor az ország minden versenyzője törlődjön a **Versenyzo** táblából! Ehhez már kell a **Fluent API**!

```
public class Versenyzo
{
    0 references
    public int Id { get; set; }
    2 references
    public string Nev { get; set; } = string.Empty;
    0 references
    public int Pont { get; set; }
    3 references
    public Szakma Szakma { get; set; } = null!;
    3 references
    public Orszag Orszag { get; set; } = null!;
    2 references
    public string SzakmaId { get; set; } = string.Empty;
    1 reference
    public string OrszagId { get; set; } = string.Empty;
}
5 references
public class Szakma
{
    1 reference
    public string Id { get; set; } = string.Empty;
    3 references
    public string SzakmaNev { get; set; } = string.Empty;
    1 reference
    public ICollection<Versenyzo> Versenyzok { get; set; } = new List<Versenyzo>();
}
5 references
public class Orszag
{
    1 reference
    public string Id { get; set; } = string.Empty;
    3 references
    public string OrszagNev { get; set; } = string.Empty;
    1 reference
    public ICollection<Versenyzo> Versenyzok { get; set; } = new List<Versenyzo>();
}
```

Itt ez a sor magyarázatra szorul. :

```
public ICollection<Versenyzo> Versenyzok { get; set; } = new List<Versenyzo>();
```

A magyarázatot a következő oldalon találod, bár azt egy másik projekthez csináltam, de szerintem így is érthető!

A **navigációs property EF**-ben azért **ICollection** (vagy más gyűjteménytípus), mert **egy színészhez több film tartozhat**, illetve **egy filmhez több színész.**

EF Core az **ICollection<T>**-t támogatja legstabilabban és legkompatibilisebben! **MINDIG EZT HASZNÁLJUK!**

Az EF Core a következő interfészekkel tud automatikusan dolgozni:

- ICollection<T>
- IList<T>
- IEnumerable<T> (de ez read-only jellegű)

Ezekből mindig az ICollection<T>-t válasszuk, mert:

- van **Add(), Remove(), Clear()** művelete
- nem túl specifikus (mint a **List<T>**)
- megfelel az **EF** belső proxynak, trackelésnek

Ez kicsit mélyebb víz, de meg kell említenem, hogy az **EF Core** bizonyos funkciókhoz (*lazy loading, change tracking, proxyzás*) gyakran **lecseréli** a kollekciót egy saját típusra. Ezt **List<T>** esetén **NEM** tudná megtenni!

Valójában a **List<T>** egy konkrét implementáció, míg az **ICollection<T>** csak egy szerződés.

Az `ICollection<T>` egy **interfész**:

- csak előírja, hogy a kollekció tudjon **Add(), Remove(), Count** stb.-t
- **NEM** szabja meg, milyen konkrét adatstruktúra legyen mögötte

Ezzel szemben a `List<T>`:

- fixen egy dinamikus tömb-alapú lista
- nem helyettesíthető más típussal
- nem cserélhető le futásidőben **EF** által

Standard, bevált gyakorlat, hogy:

- **Property** = interfész **(ICollection)**
- Konstruktorban vagy setterben így hozzuk létre → **List<T>**

```
//ha ezt írjuk le:
3 references
public ICollection<Film> Filmek { get; set; }

//akkor később teljes szabadságunk van, ez pl. mind jó:
0 references
public Film()
{
    Filmek = new List<Film>();
    Filmek = new HashSet<Film>();
    Filmek = new ObservableCollection<Film>();
}
```

Ha sikerült beállítani a Fluent API-t és jó volt a migráció, akkor már csak fel kell tölteni az adatokat az adatbázisba. Itt fontos a feltöltés sorrendje, először azokat a táblákat kell, amelyeknek **NINCS FÜGGŐSÉGE**! Tehát az **Orszag** és a **Szakma** először, utána a **Versenyzo.** Erről csak csináltam egy kis képet. Vedd észre, hogy itt a **Szakma** és az **Orszag** objektum példányosításakor **NEM** adtuk meg a **navigációs property** listát, mert azt már létrehoztuk az osztályban!

```
List<Versenyzo> v = new List<Versenyzo>();
List<Orszag> o = new List<Orszag>();
List<Szakma> sz = new List<Szakma>();
var ctx = new AppDbContext();
foreach (var i in File.ReadAllLines("szakma.csv").Skip(1))
{
    string[] resz = i.Split(';');
    Szakma x = new Szakma { Id = resz[0], SzakmaNev = resz[1] };
    sz.Add(x);

}
foreach (var i in File.ReadAllLines("orszag.csv").Skip(1))
{
    string[] resz = i.Split(';');
    Orszag x = new Orszag { Id = resz[0], OrszagNev = resz[1] };
    o.Add(x);
}
foreach (var i in File.ReadAllLines("versenyzo.csv").Skip(1))
{
    string[] resz = i.Split(';');
    Versenyzo x = new Versenyzo
    {
        Id = int.Parse(resz[0]),
        Nev = resz[1],
        SzakmaId = resz[2],
        OrszagId = resz[3],
        Pont = int.Parse(resz[4])
    };
    v.Add(x);
}

ctx.Orszagok.AddRange(o);
ctx.Szakmak.AddRange(sz);
ctx.Versenyzok.AddRange(v);
ctx.SaveChanges();
```

És eljött a pillanat, hogy továbblépjünk!

A feltöltés és mentés után van egy kész adatbázisunk, amit már nem létrehozni szeretnénk, hanem lekérdezni. A lekérdezéshez kellenek a **navigációs property**-k, amiket a múltkor **MANUÁLISAN** töltöttünk fel pusztán a jobb megértés miatt. Ha még nem érted, akkor menj vissza és nézd át újra!!!

Ha érted, akkor nosza, lépjünk tova. 😉

A **navigációs property**-k feltöltését az **EF** automatikusan is végre tudja hajtani! Erre a folyamatra két különböző módszert nézünk meg. Mindkettő nagyon fontos! Ezt is meg kell értened a továbblépéshez!

A dolog lényege annyi, hogy az **EF Core**-ban alapból a **kapcsolódó táblák NEM töltődnek be automatikusan**. Ezeket külön be kell tölteni, ha adatokat szeretnénk belőlük kinyerni. A betöltésnek két különböző módja van. Az elsőnél betöltjük az egész táblát, ez lesz az **INCLUDE**, a másodiknál csak a számunkra szükséges mezők értékét töltjük be, ez lesz a **JOIN + DTO** módszer.

Nézzük először az **INCLUDE**-t.

Itt a navigációs property-k kézzel történő feltöltése nélkül adjuk ki a következő utasítást:

```
var ctx = new AppDbContext();
var v = ctx.Versenyzok;
```

Ez létrehoz egy objektumot az **AppDbContext** osztályból, ezen keresztül érjük el az adatbázist. A **v** objektum (ami egy gyűjtemény) a **Versenyzo táblából** tartalmaz adatokat, de a **KAPCSOLÓDÓ** táblákból csak **NULL** értékek lesznek benne! Tehát ez az utasítás kiírja az első versenyző nevét:

```
Console.WriteLine(v.ElementAt(0).Nev);
```

Azért **ElementAt**, mert **NEM ALAKÍTJUK LISTÁVÁ A V**-t!!! Ugyanis ha listát csinálsz belőle, akkor letölti az adatokat az adatbázisból a memóriába és ettől kezdve **NEM** az adatbázis dolgozik, hanem a **C#**, ami sok adat esetén nagyon lassú lesz és feleslegesen pocsékolja az erőforrásokat!!!

Ott tartottunk, hogy nem tölti be a kapcsolódó táblákból az adatokat. Például:

```
var ctx = new AppDbContext();
var v = ctx.Versenyzok;
if(v.ElementAt(0).Szakma?.SzakmaNev==null)
{
    Console.WriteLine("Szopi");
}
```

Ez simán kiírja, hogy Szopi, hisz a **Szakma** property **NEM** lett feltöltve adatokkal (**NULL**)! Erre azért van egy gyengusoknak szóló megoldás, a **LAZY LOADING**, de annak annyira kevés előnye van, mint amennyire sok hátránya, így csak megemlítettük, de nem foglalkozunk vele!

Nézzük inkább az **INCLUDE**-ot. Az **Include** utasítja az EF-et, hogy *hozza magával a kapcsolódó entitásokat is* egy SQL JOIN-nel. És ettől kezdve a navigációs propertynek automatikusan feltöltött értéke lesz, nem nekem kell kézzel állítgatni! A **Versenyzőben** ugye van egy **Szakma** és egy **Orszag** nevű navigációs property. Itt azt adtam meg, hogy az *INCLUDE* ezeket töltse fel a megfelelő táblából:

```
var ctx = new AppDbContext();

var v = ctx.Versenyzok
        .Include(x => x.Szakma)
        .Include(x => x.Orszag);

Console.WriteLine(v.ElementAt(0).Szakma.SzakmaNev);
Console.WriteLine(v.ElementAt(0).Orszag.OrszagNev);
```

Így már kiíródik az első versenyző szakmaneve és országa, mert feltöltöttük a navigációs propertyket! És simán kigyűjthetjük a kőművesek neveit, hisz a **FOREACH** végig tud menni a gyűjteményen is és a **LINQ** is könnyedén kezeli, **NEM KELL A LIST**! (többször nem írom le) 😊

```
var ctx = new AppDbContext();

var v = ctx.Versenyzok
        .Include(x => x.Szakma)
        .Include(x => x.Orszag);

foreach (var i in v.Where(x => x.Szakma.SzakmaNev == "Komuves"))
{
    Console.WriteLine($"{i.Nev} - {i.SzakmaId} - {i.Orszag.OrszagNev}");
}
```

Természetesen **Include**-ni lehet másik irányból is, csak nem lesz egyszerűbb, így nem is érdemes, de megmutatom elrettentésnek, valamint **HA MEGÉRTED, AKKOR INNENTŐL SÉTAGALOPP AZ EGÉSZ!**
Az **Orszag** táblának a navigációs propertyje a **Versenyzok** nevű lista. Ezt feltöltöd **Include**-al a **Versenyzo** osztályból majd utána a **Versenyzo** osztályban lévő **Szakma** nevű navigációs property-t (ami egy objektum a **Szakma** osztályból) töltöd fel a **Szakma** osztályból, de az egy szinttel lejjebb van, ezért kell a **ThenInculde**-t használni. Remélem azért érhető vagyok. 😊

```
var ctx = new AppDbContext();

var v = ctx.Versenyzok
        .Include(x => x.Szakma)
        .Include(x => x.Orszag);

foreach (var i in v.Where(x => x.Szakma.SzakmaNev == "Komuves"))
{
    Console.WriteLine($"{i.Nev} - {i.SzakmaId} - {i.Orszag.OrszagNev}");
}

var o = ctx.Orszagok
    .Include(x => x.Versenyzok)
        .ThenInclude(y => y.Szakma);

var vankomuves = o.Where(x => x.Versenyzok.Any(v => v.Szakma.SzakmaNev == "Komuves"));

foreach (var orszag in vankomuves)
{
    foreach (var r in orszag.Versenyzok.Where(r => r.Szakma.SzakmaNev == "Komuves"))
    {
        Console.WriteLine($"{r.Nev} - {r.SzakmaId} - {orszag.OrszagNev})");
    }
}
```

Remélem belátható, hogy az **1:N**-es kapcsolat esetén az **N** irányából érdemes **Inculde-ni!**

Viszont az Include sok tábla (5+) esetén már NEM JÓ! Minden táblából mindent betölt, holott lehet, hogy táblánként csak 1-1 mező értékére van szükségünk. Tovább kell lépni!

A továbblépést a **JOIN + DTO** jelenti.

A DTO, teljes nevén Data Transfer Object (adatátviteli objektum) valójában egy olyan **C# osztály**, amit **kizárólag adat lekérdezésére és továbbküldésére** használunk – *nem az adatbázist reprezentáló entitás*, nem kerül mentésre az EF-ben. A fő jellemzői:

- csak **adatot tartalmaz**, nincs logika
- **nincs navigációs property**
- **nincs EF Core által kezelt kapcsolata**
- csak azokat a mezőket tartalmazza, amelyekre tényleg szükségünk van
- gyorsabban töltődik be, mert az **EF** kisebb **SELECT**-et csinál
- viszont nem menthető az adatbázisba
- csak olvassuk
- **API** és **UI** számára ideális, a cégek többségénél csak ez használható **API**-hoz

Apró minta a DTO-ra:

```
// EF entitások:
0 references
public class Versenyzo1
{
    0 references
    public int Id { get; set; }
    0 references
    public string Nev { get; set; }
    0 references
    public int SzakmaId { get; set; }
    0 references
    public Szakma Szakma { get; set; }
}
0 references
public class Szakma1
{
    0 references
    public int Id { get; set; }
    0 references
    public string Nev { get; set; }
}

// példa DTO verzióra, csak az van benne, ami nekünk kell!!!
0 references
public class VersenyzoDTO
{
    0 references
    public int Id { get; set; }
    0 references
    public string Nev { get; set; }
    0 references
    public string SzakmaNev { get; set; }
}
```

Kapcsolt táblák esetén a **JOIN+DTO** megvalósításának szintaktikája erősen hasonlít az **SQL**-re. Összekapcsoljuk a táblákat, majd egy névtelen típusba kigyűjtjük a minket érdeklő adatokat.

```
var adatok =
    from x in ctx.Versenyzok
    join s in ctx.Szakmak on x.SzakmaId equals s.Id
    join y in ctx.Orszagok on x.OrszagId equals y.Id
    select new
    {
        Id = x.Id,
        Nev = x.Nev,
        Pont = x.Pont,
        Szakma = s.SzakmaNev,
        Orszag = y.OrszagNev,
    };

foreach (var i in adatok.OrderByDescending(x=>x.Pont).Take(10))
{
    Console.WriteLine($"{i.Nev} - {i.Szakma} - {i.Orszag} - {i.Pont}");
}
```

itt egy minta az Include és a Join+DTO -ra

Include:

```
var eredmeny = ctx.Versenyzok
    .Include(v => v.Szakma)
    .Include(v => v.Orszag)
    .Include(v => v.Eredmenyek)
        .ThenInclude(e => e.Verseny)
    .Include(v => v.Edzo)
        .ThenInclude(e => e.Helyszin);
```

Majdnem egyezik a kettő, de a lényeg szerintem átmegy.

Join + DTO:

```
var lista =
    from v in ctx.Versenyzok
    join s in ctx.Szakmak on v.SzakmaId equals s.Id
    join o in ctx.Orszagok on v.OrszagId equals o.Id
    join e in ctx.Edzo on v.EdzoId equals e.Id
    join h in ctx.Helyszin on e.HelyszinId equals h.Id
    select new
    {
        Id = v.Id,
        Nev = v.Nev,
        Pont = v.Pont,
        Szakma = s.Nev,
        Orszag = o.OrszagNev,
        Edzo = e.Nev,
        Helyszin = h.Nev
    };
```