A leggyakrabban használt EF Core Fluent API parancsok

A **Fluent API** az Entity Framework Core-ban (és más .NET könyvtárakban) egy **kód alapú konfigurációs módszer**, amelyben **láncolt metódushívásokkal** állítható be az **entitások**, **kapcsolatok**, **property**-k viselkedése. Azért nevezik „*folyékony*” stílusnak, mert minden metódushívás egy olyan objektummal tér vissza, amelyen ismét lehet metódushívásokat alkalmazni, így lesz belőle láncolt, könnyen olvasható és értelmezhető utasítássorozat.

Eddig ezeket a beállításokat az **Annotációk** segítségével oldottuk meg:
(`using System.ComponentModel.DataAnnotations névtérből pl. [Key]`)

Csakhogy az **Annotációkkal NEM LEHET MINDEN FONTOS** beállítást szabályozni! (újonnan belekerült ugyan a **UNIQUE** tulajdonság szabályozása is
`[Index(nameof(Mezo), IsUnique = true)]`
de pl. *összetett kulcsot* **NEM LEHET** beállítani annotációval!) => Meg kell ismerkednünk a **Fluent API** -val.

Nézzük, miket tudunk beállítani a **Fluent API** -val (spoiler: mindent):

- táblanevek
- kulcsok
- relációk
- oszlopok tulajdonságai
- törlési szabályok
- indexek
- seed adatok

Ráadásul „erősebb”, mint a **DataAnnotation** és nagyobb projekteknél **CSAK** ezt használják!

Nézzünk pár általános példát, a konkrét megvalósítást a többi dokumentumban mutatom!

1.1. Tábla nevének beállítása:

```
modelBuilder.Entity<User>()
    .ToTable("users");
```

1.2. Elsődleges kulcs (PK)

```
modelBuilder.Entity<User>()
    .HasKey(u => u.Id);
```

1.3. Kompozit kulcs (ez azt jelenti, hogy nem egy oszlop önmagában a kulcs, hanem több)

```
modelBuilder.Entity<SzineszFilm>()
    .HasKey(szf => new { szf.SzineszId, szf.FilmId });
```

2. Property konfiguráció (oszlopok)

2.1. Oszlop neve

```
modelBuilder.Entity<User>()
    .Property(u => u.Email)
    .HasColumnName("email_address");
```

2.2. Típus meghatározása

```
.Property(u => u.Price)
.HasColumnType("decimal(10,2)");
```

2.3. Max hossz (string)

```
.Property(u => u.Name)
.HasMaxLength(100);
```

2.4. Kötelező mező (NOT NULL)

```
.Property(u => u.Name)
.IsRequired();
```

2.5. Egyedi mező (UNIQUE)

```
.HasIndex(u => u.Email)
.IsUnique();
```

2.6. Alapértelmezett érték

```
.Property(u => u.IsActive)
.HasDefaultValue(true);
```

2.7. Alapértelmezett SQL érték

```
.Property(u => u.CreatedAt)
.HasDefaultValueSql("GETDATE()");
```

3. Táblák közötti kapcsolatok

3.1. Klasszikus 1:N kapcsolat (1 user – N orders)

```
modelBuilder.Entity<Order>()
    .HasOne(o => o.User)
    .WithMany(u => u.Orders)
    .HasForeignKey(o => o.UserId);
```

A **User** táblát ez a sor kezeli le:
.HasOne(o => o.User)

Az **Orders** táblát és azt, hogy több rekord is tartozhat egy userhez ez a sor kezeli le:
.WithMany(u => u.Orders)

Az idegen kulcs az **Orders** táblában (*mindig az N ágban van az idegen kulcs*!!! (**FK**))
.HasForeignKey(o => o.UserId);

3.2. Egy-az-egyhez (1:1) kapcsolat

```
modelBuilder.Entity<User>()
    .HasOne(u => u.Profile)
    .WithOne(p => p.User)
    .HasForeignKey<UserProfile>(p => p.UserId);
```

3.3. Sok-a-sokhoz (N:M)

3.3.1. Kapcsolótábla osztállyal

```
modelBuilder.Entity<SzineszFilm>()
    .HasKey(szf => new { szf.SzineszId, szf.FilmId });
```

3.3.2. Kapcsolótábla osztály nélkül (UsingEntity) Csak megemlítjük, a lényege annyi, hogy a kapcsolótáblát automatikusan az **EF** hozza létre, egyszerű, gyors, de csak erősen korlátozottan használható!

```
modelBuilder.Entity<Szinesz>()
    .HasMany(sz => sz.Szineszek)
    .WithMany(f => f.Filmek)
    .UsingEntity(j => j.ToTable("szinesz_film"));
```

4. Indexek

4.1. Egyszerű index

```
.HasIndex(u => u.Email);
```

4.2. Kompozit index

```
.HasIndex(u => new { u.FirstName, u.LastName });
```

4.3. Egyedi index (régen csak ezzel állíthattuk a **UNIQUE tulajdonságot)**

```
.HasIndex(u => u.Email).IsUnique();
```

5. Konfigurálás a két oldalon

5.1. Affluent chaining — teljes kapcsolat konfiguráció

```
modelBuilder.Entity<Comment>()
    .HasOne(c => c.Post)
    .WithMany(p => p.Comments)
    .HasForeignKey(c => c.PostId)
    .OnDelete(DeleteBehavior.Cascade);
```

6. Törlési viselkedés

Az **EF Core** törlési viselkedések szabálya (DeleteBehavior) azt határozza meg, mi történjen a kapcsolódó rekordokkal, amikor egy entitást törölsz, amelyhez más entitások kapcsolódnak. FK használata esetén számít!

`.OnDelete(DeleteBehavior.Restrict);`
Nem engedi törölni, ha vannak kapcsolódó rekordok.

`.OnDelete(DeleteBehavior.NoAction);`
az adatbázisra bízza a viselkedést (általában Restrict)

`.OnDelete(DeleteBehavior.SetNull);`
A kapcsolódó rekordok **megmaradnak**, de a FK mező **NULL-ra áll**, ha a parent törlődik.

`.OnDelete(DeleteBehavior.Cascade);`
A kapcsolódó rekordokat is törli.

7. Check constraint

```
modelBuilder.Entity<Product>()
    .HasCheckConstraint("CK_Product_Price", "Price > 0");
```

8. Seed Data

8.1. Entitás seed

```
modelBuilder.Entity<User>().HasData(
    new User { Id = 1, Name = "Dezső" },
    new User { Id = 2, Name = "Rozál" }
    );
```

8.2. Middle table seed

```
.UsingEntity(j =>
    j.HasData(new { UsersId = 1, RolesId = 2 })
    );
```