Entity Framework

Jánosi-Rancz Katalin Tünde

Sapientia EMTE
tsuto@ms.sapientia.ro

Adatmodell és üzleti logika

- az adatbázissémák nem mindig ideálisak az éppen készülő alkalmazások számára
  - meglévő sémákra kell épülniük az új alkalmazásoknak
  - alul/felülnormalizált sémák működési vagy teljesítményszempontok miatt
  - az alkalmazás és az adatbázis is fejlődik az idővel
- az adatbázisséma gyakran átitatja az egész alkalmazást
  - minden alkalmazás próbál számára logikus nézeteket létrehozni
    - tárolt eljárások, nézetek és (a leggyakrabban) ad hoc lekérdezések
  - a séma helyenként döntően visszahat az alkalmazás felépítésére

Alapprobléma

- a fogalmi szint (üzleti logika) a valóságot közvetlenül modellezi
- a logikai szint (adatbázis) tartalmilag azonos, de normalizált formában
- a fogalmi szint és a logikai szint közti különbség áthidalására lehetővé tesszük a kettő közti automatikus leképzést
- Adat != Objektum
  - ORM

Entity Framework - EF

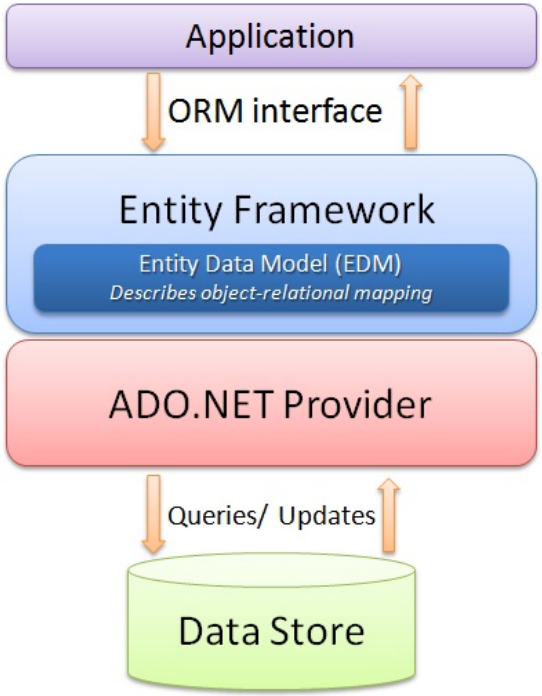
- `ORM` - Objektum-Relációs Leképezés (Object Relational Mapping) egy programozási technika adatok konvertálására, nem kompatibilis típusos rendszerek és objektumorientált programozási nyelvek között (Wikipedia)
- `EF` : az ADO.NET nyílt forráskódú adatelérési keretrendszere - ORM(.NET 3.5-től)
- `EF` célja:
  - lecsökkentse az alkalmazáson belüli objektum alapú adatábrázolás /tárolás és az adatbázison belüli relációs adattárolás közötti különbséget
  - lehetővé tegye, hogy objektumokkal és tulajdonságokkal dolgozzunk táblák helyett
  - lehetővé teszi a logikai (adatbázis) és a fogalmi (üzleti logika) modellek szétválasztását
  - függetleníti az alkalmazásunkat az adatbázismotortól

Entity Framework - EF

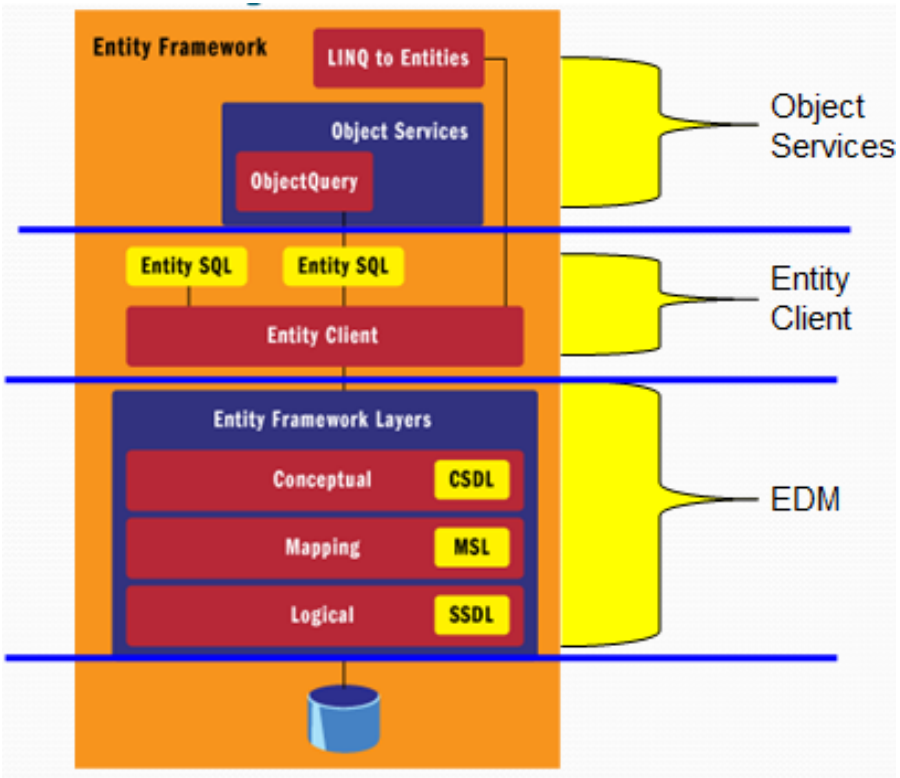
- automatikusan létrehozza az adatokat reprezentáló entitásokat és a köztük lévő kapcsolatokat
- gazdag lekérdezési lehetőségeket biztosít nyelvi támogatással
  - egyik lekérdező nyelve a `Linq`
  - a LINQ kéréseket SQL lekérdezéssé transzformálja
- kihívások: egységbezárás, interfészek, osztályok, származtatás, polimorfizmus, adattípusok közötti különbségek, stb.
- előnyök: könnyebb és/vagy absztraktabb kód

Entity Framework (EF) helye

- EF: egy ORM réteg
- ADO.NET data provider (adatszolgáltató): egy SW komponens, amely kölcsönhatásba lép az adatforrással (szöveges állomány, táblázatkezelő, ABKR, pl. Oracle DB)
- EDM: egy fogalom halmaz, amely leírja az adatok szerkezetét tekintettel a tárolt formájára
- ORM interfész

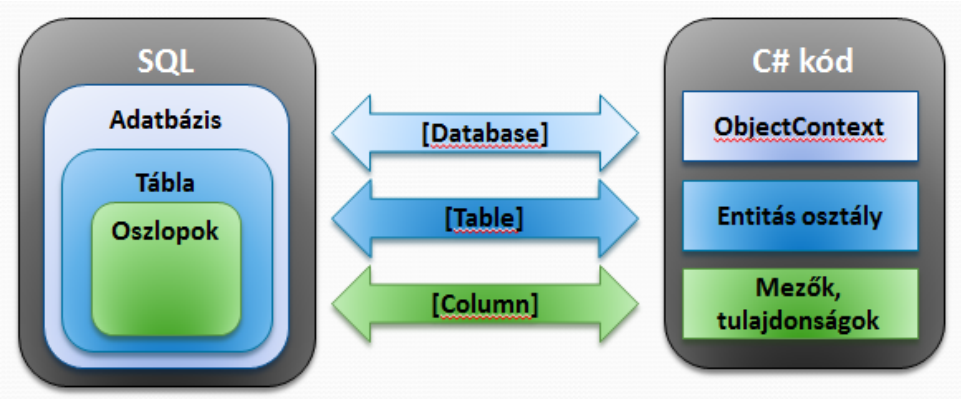


Az EF architektúrája



EF leképezés

- a teljes relációs adatbázist egy `ObjectContext` fogja reprezentálni
- öröklés többféleképpen (később)
- táblákból Entitás osztályok keletkeznek
  - egy táblából akár több entitás és fordítva
- oszlopokból pedig mezők és tulajdonságok
- sorokból lesznek `entitás` objektumok



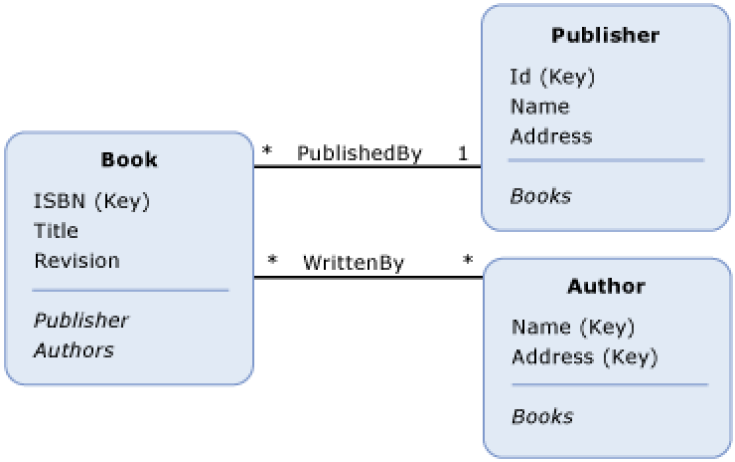
- minden alkalmazás saját nézetet kaphat ugyanahhoz az adatforráshoz
- a nézetek megvalósítása tisztán a kliensoldalon történik
- az adatbázisséma ”tiszta” marad, nincs adatbázis módosítás

EF leképezés -2

- nem kell feltétlenül olyan entitás modellt készíteni, amiben egy táblának egy osztály felel meg és fordítva
- még a név sem kell azonos legyen (loose coupling)
- az osztályba felvehetünk további, pl. számított tulajdonságokat

Entity Data Model

- leírja az **entitásokat** és a **köztük levő kapcsolatokat** CSDL-t (conceptual schema definition language) használva



koncepcionális modell diagramm

http://msdn.microsoft.com/en-US/data/jj652004

```
<Schema xmlns="http://schemas.microsoft.com/ado/2008/09/edm"
xmlns:cg="http://schemas.microsoft.com/ado/2006/04/codegeneration"
xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/EntityStoreSchemaGenerator"
    Namespace="BooksModel" Alias="Self">
        <EntityContainer Name="BooksContainer" >
            <EntitySet Name="Books" EntityType="BooksModel.Book" />
            <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
            <EntitySet Name="Authors" EntityType="BooksModel.Author" />
            <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
                <End Role="Book" EntitySet="Books" />
                <End Role="Publisher" EntitySet="Publishers" />
            </AssociationSet>
            <AssociationSet Name="WrittenBy" Association="BooksModel.WrittenBy">
                <End Role="Book" EntitySet="Books" />
                <End Role="Author" EntitySet="Authors" />
            </AssociationSet>
        </EntityContainer>
        <EntityType Name="Book">
            <Key>
                <PropertyRef Name="ISBN" />
            </Key>
            <Property Type="String" Name="ISBN" Nullable="false" />
            <Property Type="String" Name="Title" Nullable="false" />
            <Property Type="Decimal" Name="Revision" Nullable="false" Precision="29" Scale="29" />
            <NavigationProperty Name="Publisher" Relationship="BooksModel.PublishedBy"
```

koncepcionális modell CSDL-el

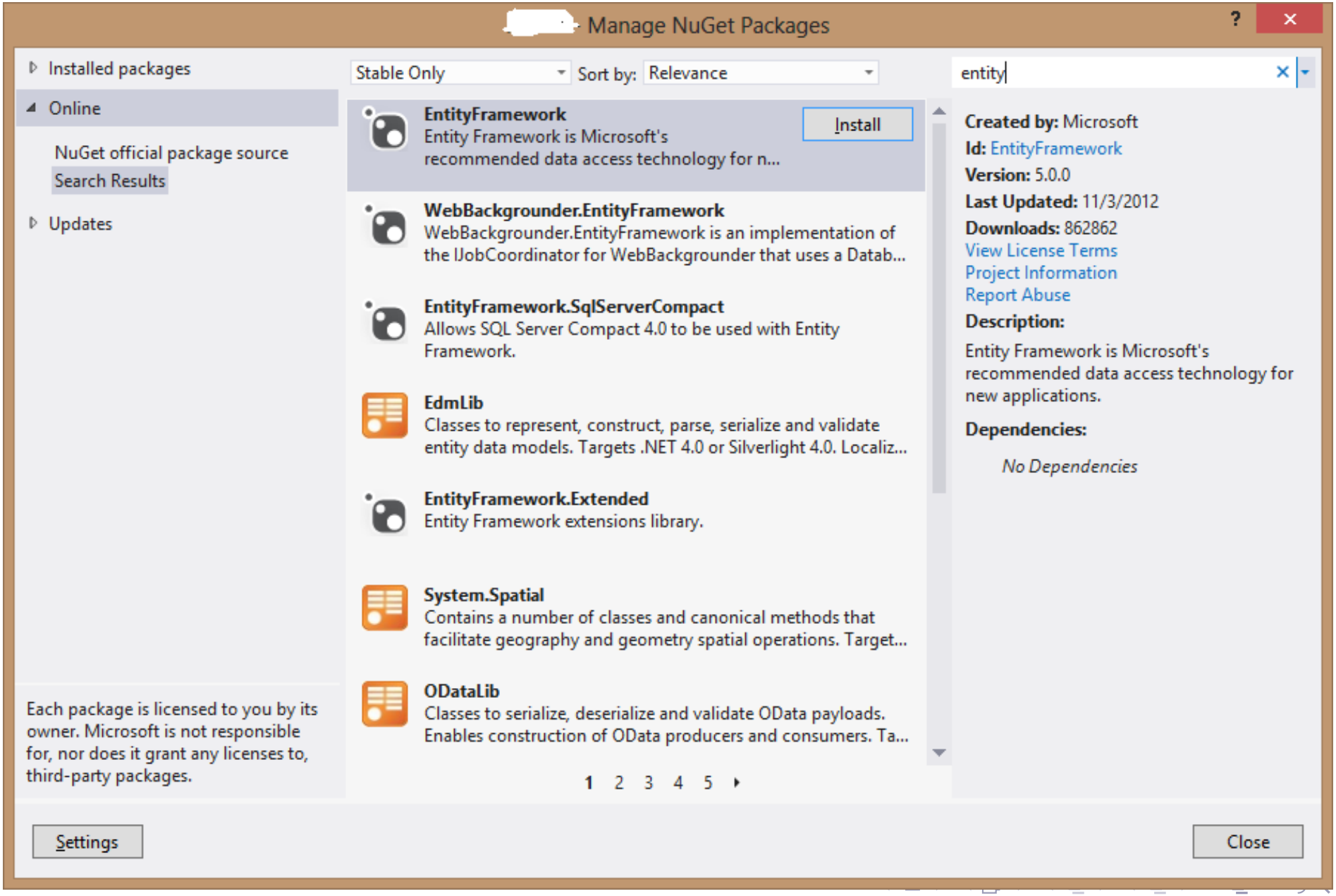
EDMX

- Az Edmx fájl tartalmazza az EF metaadat meghatározásokat (CSDL/MSL/SSDL)
- XML fájlok, ezek létrehozhatóak és szerkeszthetőek kézzel
- .CSDL: az alkalmazás fogalmi szintjét írja le
  - a rendszer ebből fogja generálni az elérő osztályokat
  - az alkalmazásunkban csak ezt a sémát kell ismernünk az adateléréshez
  - a séma a mi igényeinkhez igazítható, és nem kell ragaszkodnunk az adatbázis szerkezethez
- .SSDL: magát az adatbázist írja le, a PK/FK relációkkal együtt
- .MSL: a kettő közötti (CSDL <–> SSDL) leképzést írja le

Adatbázis

- az EDM nem rendelkezik közvetlen ismeretekkel az adatbázismotorról
- a konkrét adatbázismotor típusa (Oracle, MSSQL stb.) (elvileg) nincs közvetlen hatással az EDM működésére
- az entitásokon végzett műveleteket a `provider` fordítja le az adatbázismotor műveleteire
- támogatott szolgáltatók:
  - SQL Server
  - Oracle
  - MySQL
  - Stb.
- `provider`: híd az alkalmazás és az adatforrás között, ezen keresztül mozognak az adatok az alkalmazás és az adatbázis között
  - Oracle Data Provider
    - http://www.oracle.com/technetwork/topics/dotnet/utilsoft-086879.html
    - ODTwithODAC121024.zip

EF installálás



Adatelérés

- a lekérdezéseket az adatmodellen fogalmazzuk meg
- lekérdezés a modellen –> adatbázis-specifikus lekérdezés
- a lekérdezések lefordulnak és az adattárban hajtódnak végre
  - nincs lokális végrehajtás
- a lekérdezés eredményei entitásokként materializálódnak
- Lehetőségek
  - LINQ to Entities
  - Entity SQL + Object Services
  - Entity SQL + Entity Client

LINQ to Entities

- LINQ to Entities - SQL-ből megszokott szintaszis
- IntelliSense támogatás
- a lekérdezések Entity SQL lekérdezésekként értelmeződnek

```
var courses = from course in context.Courses
              where course.Title.StartsWith(”C”)
              orderby course.Title ascending
              select new
              {
                    Title = course.Title,
                    Location = course.Location
              };
```

Entity SQL

- T-SQL -szerű lekérdezési nyelv
- EF az Entity SQL-t adatbázis specifikus lekérdezésre fordítja

```
var qStr = @"SELECT VALUE c
             FROM SchoolEntities.Courses AS c
             WHERE c.Title='Calculus'";
var courses = context.CreateQuery<Course>(qStr);
```

Entity Client lekérdezés

- EntityDataReader-t használ

```
using (var conn = new EntityConnection(”name=ProgrammingEFDB1Entities”))
{
    conn.Open();
    var qStr = ”SELECT VALUE c FROM SchoolEntities.Courses AS c ”;
    var cmd = conn.CreateCommand();
    cmd.CommandText = qStr;
    using (var rdr = cmd.ExecuteReader(CommandBehavior.SequentialAccess))
    {
        while (rdr.Read())
    {
    Console.WriteLine(rdr.GetString(1));
    }
    }
}
```

Lekérdezési módok összehasonlítása

	LINQ to Entities	Entity SQL with Object Services	Entity SQL with Entity Client
LINQ support	√	–	–
IntelliSense	√	–	–
Model dynamic queries	–	√	√
Return type	Objects	Objects or DbDataRecords	DbDataReader
Performance	√	√√√	√√

Melyiket használjuk?

- Entity SQL + Entity Client
  - az adatokat sorban (streamelve) szeretnénk lekérdezni
  - meglévő alkalmazások átírásakor
- Entity SQL + Object Services
  - LINQ-nél nagyobb kifejezőerőre van szükség
  - dinamikusan összeállított lekérdezésekre van szükség
  - a teljesítmény nagyon fontos
- LINQ to Entities
  - minden más esetben

LINQ to Entities - Adatbetöltés

Adatbetöltés - Loading Entities

- a kapcsolodó entitásokat (gyerek vagy unoka entitásokat) a relációkon keresztül töltjük le
- 3 féle betöltés:
  - `Mohó` betöltés (Eager loading)
  - `Explicit` betöltés (Explicit loading)
  - `Lusta` betöltés (Lazy loading)

Mohó betöltés - Eager loading

- egyetlen lekérdezéssel mindent
- egy bizonyos entitás lekérésével betölti a kapcsolodó entitásokat
- letöltést követően az adatok és a részletek azonnal elérhetőek lesznek
- Include() metódus használatával érjük el (System.Data.Entity névtér)
- hátránya: nagy mennyiségű adatot tölt le egyszerre

```
var blogs1 = context.Blogs
                        .Include(”Posts”).ToList();
// vagy Lambda kif-el
var blogs2 = context.Blogs
                        .Include(b => b.Posts).ToList();
```

Mohó betöltés - Példák

- extra where feltétel

```
var blog1 = context.Blogs
                        .Where(b => b.Name == ”ADO.NET Blog”)
                        .Include(”Posts”).FirstOrDefault();
```

- több szintű betöltés (unokát is)

```
var blogs1 = context.Blogs
                         .Include(”Posts.Comments”).ToList();
// vagy
var blogs2 = context.Blogs
                        .Include(b => b.Posts.Select(p => p.Comments))
                        .ToList();
```

NB: context.Database.Log = Console.Write; //kiirja a generált sql utasítást

Lusta vagy késleltetett betöltés - Lazy loading

- $\simeq$ default
- csak akkor töltsük be a kapcsolódó entitásokat, amikor ténlyeg szükség van rá
  - csak azokat, amiket pl. a felhasználó kinyit
- nem akkor kerül végrehajtásra, amikor elkészítjük, hanem akkor, amikor hozzáférünk, pl. bejárjuk
- kivétel:
  - Count, First, ToArray, ToList, ToDictionary, ToLookup
- virtual módosítót használ a kapcsolatban lévő propertykre

```
// Blog.cs-ben
 public virtual ICollection<Post> Posts { get; set; }
```

```
IList<Blog> blogList = ctx.Blogs.ToList<Blog>();
Blog blg = blogList[0]; //csak amelyiket lenyitották pl.
    foreach (var i in blg.Posts)
        {
            Console.WriteLine(i.Name);
        }
```

Lazy loading kikapcsolása

- lekérhetjük a teljes AB-t ha nem vagyunk elég figyelemesek
- kikapcsolás:
  - elhagyjuk a virtual kulcsszót

```
public ICollection<Post> Posts { get; set; }
```

  - kikapcsolható a contextben levő összes entitásra

```
public class BloggingContext : DbContext
{
    public BloggingContext()
    {
        this.Configuration.LazyLoadingEnabled = false;
    }
}
```

Explicit loading

- ha a Lazy Loading le van tiltva, még akkor is lehet késleltetve betölteni egy explicit `Load()` hívással
  - Entry(...).Reference(...).Load() vagy Entry(...)..Collection(...).Load()

```
var post = context.Posts.Find(2);
context.Entry(post).Reference(p => p.Blog).Load(); //bizonyos post blogja
//vagy
context.Entry(post).Reference("Blog").Load();//bizonyos post blogja

var blog = context.Blogs.Find(1);
context.Entry(blog).Collection(p => p.Posts).Load(); //bizonyos blog
     postjai
//vagy
context.Entry(blog).Collection("Posts").Load();//bizonyos blog postjai
```

NB: `Reference` -t használunk ha egyetlen entitás tartozik hozzá és `Collection` -t ha egy egész gyűjtemény

Explicit loading -2

- Query() metódus biztosítja a hozzáférést a kapcsolodó entitásokhoz

```
context.Entry(blog)
        .Collection(”Posts”)
        .Query()
        .Where(p => p.Tags.Contains(”entity-framework”) //postok Tag-
            jei, amelyek...
        .Load();
```

- ha nem akarjuk betölteni csak megszámolni a kapcsolodó entitásokat, akkor elhagyjuk a Load()-ot

```
var postCount = context.Entry(blog)
                       .Collection(b => b.Posts)
                       .Query()
                       .Count();
```

AB műveletek - Entitások létrehozása

```
Product p1 = new Product
{
      ProductName = ”Új termék”,
      CategoryID = 1,
      SupplierID = 4
};
//hozzáadjuk a DataContext-hez
ctx.Products.Add (p1);

//ténylegesen ekkor kerül bele az adatbázisba
ctx.SaveChanges();
```

- Entitás csatolása
  - EntityCollection.Add(...)
  - ObjectContext.AddTo(...)
- az előbbi példában lehetett volna használni a context AddToProducts() függvényét is

Entitások módosítása

```
//kikeressük az adatbázisból
Product modProduct = (from p in ctx.Products
                      where p.ProductName == ”Új termék”
                      select p).Single();

//megváltoztatjuk a nevét
modProduct.ProductName = ”Módosított termék”;

//változások mentése
ctx.SaveChanges();
```

Entitások törlése

- csak betöltött entitást tudunk törölni

```
Product delProduct = (from p in ctx.Products
                      where p.ProductName == ”Módosított termék”
                      select p).Single();

//törlés a DataContext-bol
ctx.Products.Remove(delProduct);

//változások mentése
ctx.SaveChanges();
```

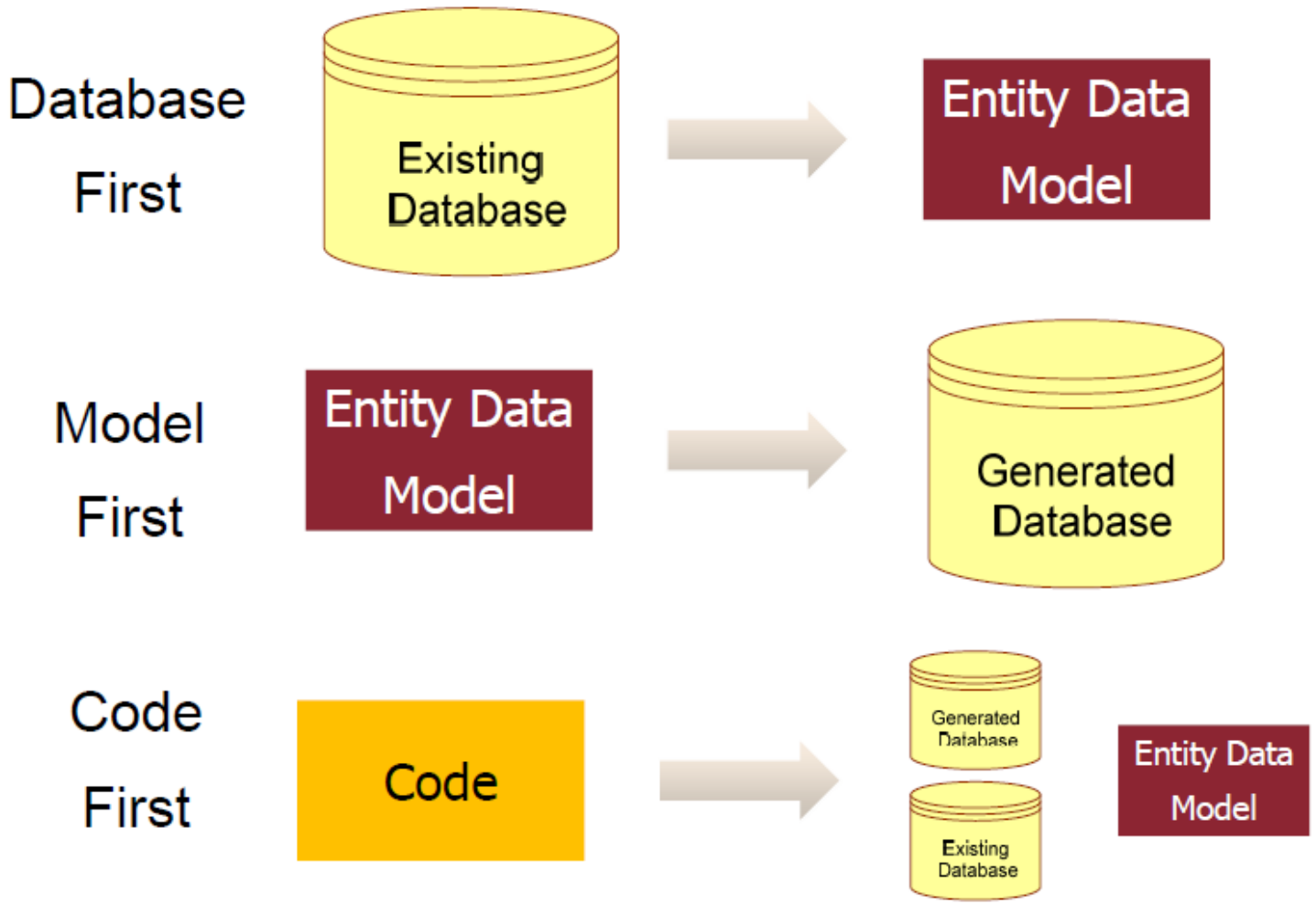
Változáskövetés

- az entitások változásait az `ObjectContext` tartja nyílván.
- az ObjectContext referenciát tart minden példányosított entitásra
- az adatváltozások visszaírása az adatbázisba a `SaveChanges` metódussal történik
- ADO.NET optimista ütközéskezelés
- a módosítások egy tranzakción belül kerülnek végrehajtásra

```
context.SaveChanges();

OR

// megmondja az EF-nak, hogy hajtsa végre a szükséges AB parancsokat, de
     jegyezze meg a változásokat, hogy hiba esetén visszapörgetheto legyen
context.SaveChanges(false);

//ha idáig eljutottunk akkor minden rendben
scope.Complete();

//ha idáig eljutottunk akkor minden változtatást elfogadunk
context.AcceptAllChanges();
```

Entity Framework - Modellezés iránya

Modellezés iránya



Modellezés iránya -2

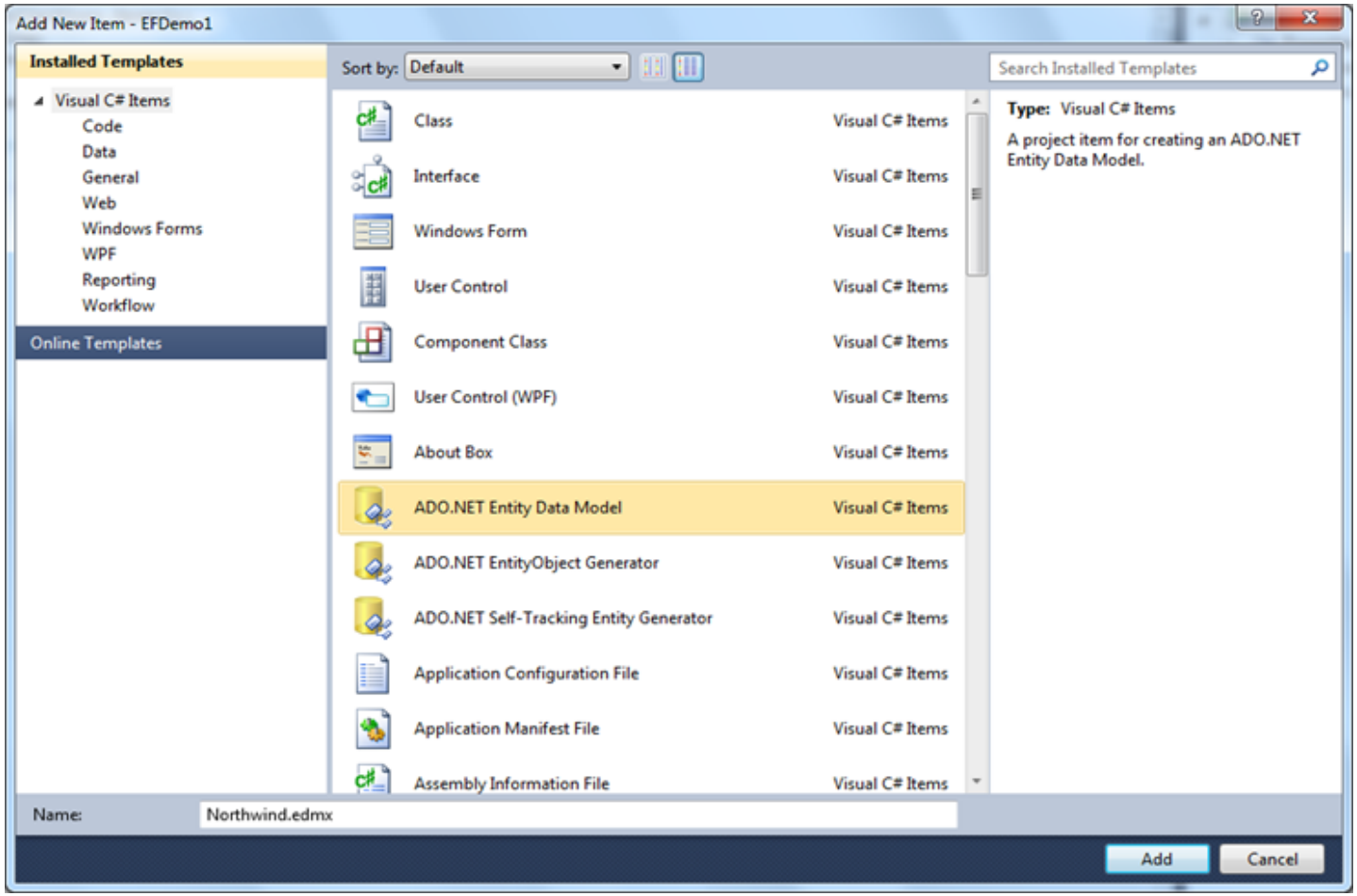
1. Code first
   - teljes kontroll a kód felett
   - általános elvárás, hogy ne foglalkozz az AB-al
   - a Code First API-k létrehozzák az AB-t az entitás osztályok és konfiguráció alapján
2. Database first
   - a relációs adatbázis már létezik, ebből hozunk létre entitás osztályokat
   - jó választás ha van egy DBA által megtervezett AB-unk, vagy egy létező AB-unk
   - EF létrehozza neked az entitásokat
   - az AB-on végezhetünk manuális módosításokat
3. Model first
   - előbb létrehozzuk az entitás modellt, majd ebből generálunk adatbázist
   - népszerű a designereknek (= nem szeret sem kódot, sem SQL-t írni)
   - az AB script és az osztály template scriptje is kigenerálódik

http://stackoverflow.com/questions/5446316/code-first-vs-model-database-first

EF - Database first

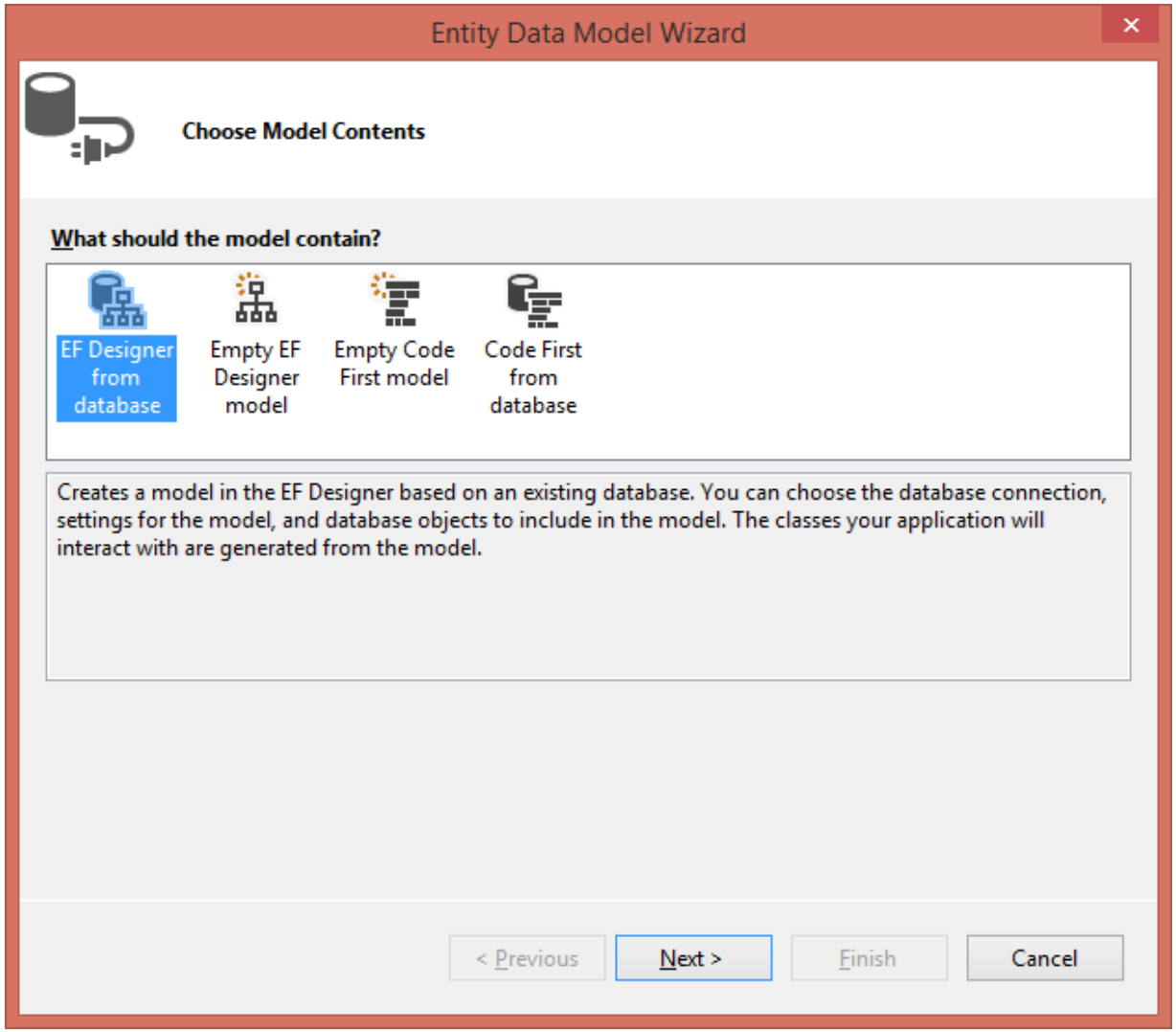
EF - Database first

- Jobb klikk a projektre –> Add –> New Item –> ADO.NET Entity Data Model



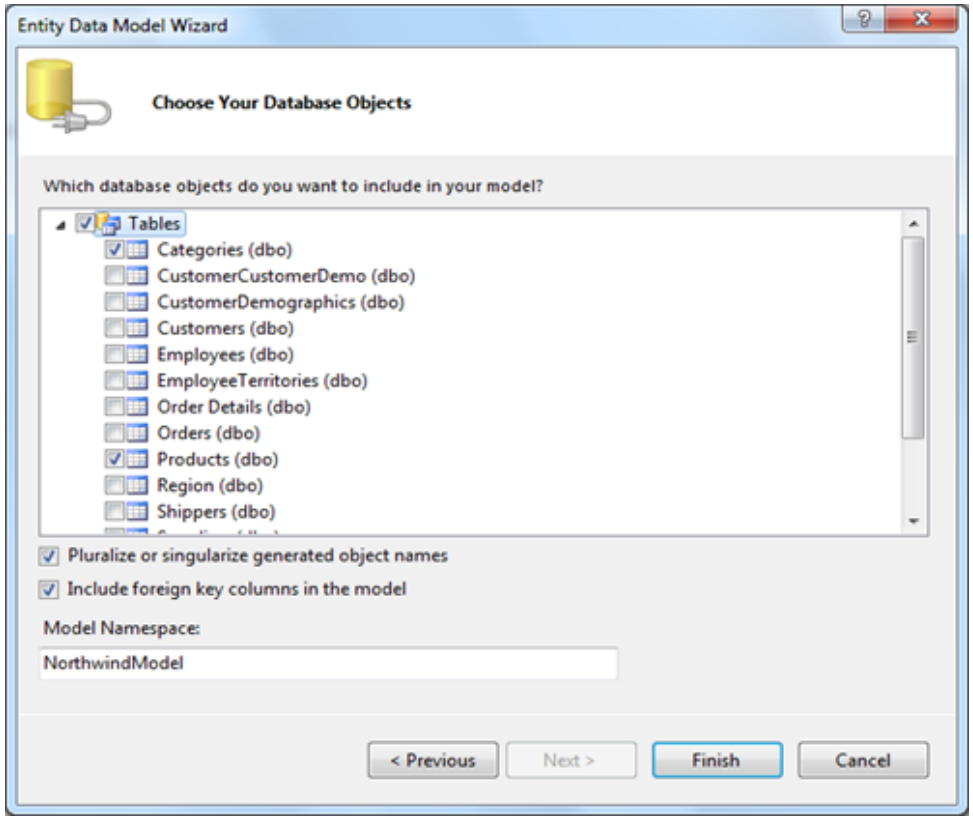
EF - Database first -2

- válasszuk a EF designer from database-t



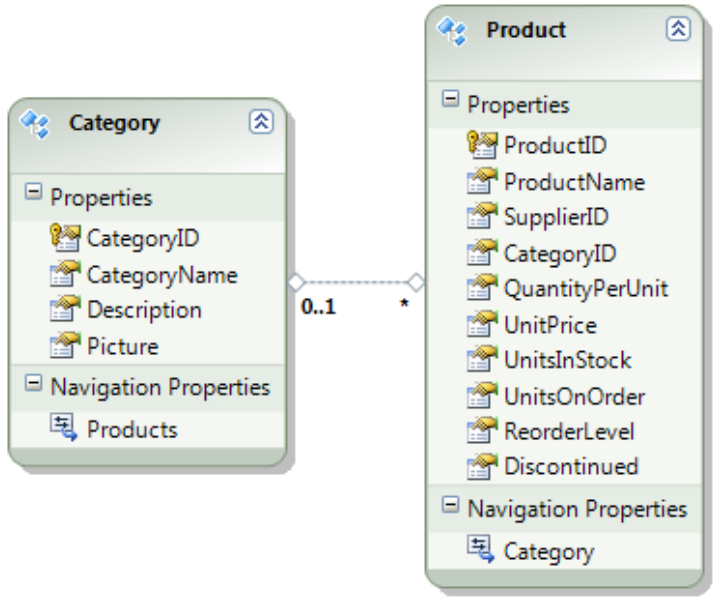
EF - Database first -3

- válasszuk ki az adatbázis kapcsolatot –> Next
- válasszuk ki azokat a táblákat, amiket használni szeretnénk
- egyes szám többes szám átalakítás



Mi jött létre?

- legenerálta az Context-et és az entitásokat
- itt a context osztályunk az ObjectContext-től örököl!
- a Solution Explorer-ben létrejött egy .edmx fájl
- a .designer.cs állományban vannak a generált C# osztályok
- a referenciák (References) közé felvenni a System.Data.Entity.dll-t (VS 2010), EntityFramework.dll-t
- konfigurációs állományba (web.config vagy app.config desktop alkalmazásnál) felvette a connection string-et



Entity Framework Code First

Code First - egy rövid példa

Lépések:

1. létrehozni a modellt
2. létrehozni a context-et
3. olvasni és írni az adatokat

`http://msdn.microsoft.com/en-us/data/jj193542.aspx`

1. A modell létrehozása

Blog 1 — * Post

```
public class Blog {
    public int BlogId { get; set; }
    public string Name { get; set; }
    public virtual List<Post> Posts { get; set; }
}

public class Post {
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public int BlogId { get; set; }
    public virtual Blog Blog { get; set; }
}
```

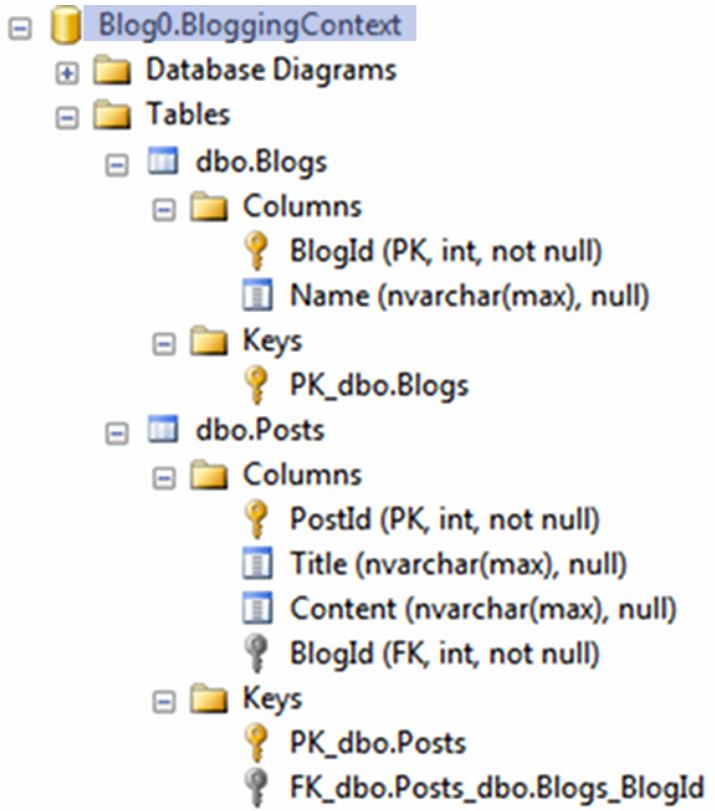
2. A Context létrehozása

- bizonyosodjunk meg, hogy az Entity Framework fel van installálva (via NuGet)
- adjuk hozzá a projekthez referenciaként az EF assembly-t

```
using System.Data.Entity;
public class BloggingContext : DbContext {
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
}
```

3. Az adatok írása és olvasása

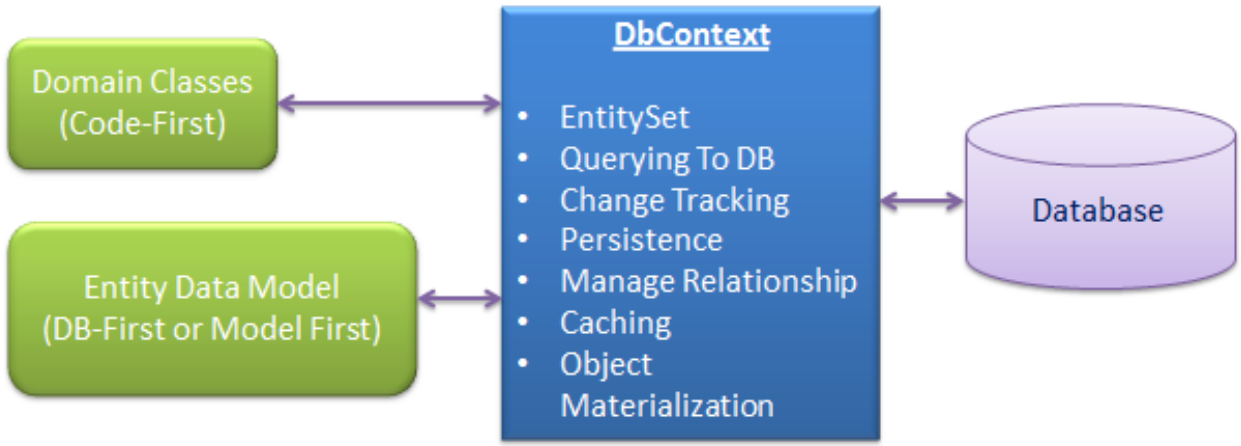
```
using (var db = new BloggingContext()) {
  Console.Write(”Enter new blog name:”);
  var name = Console.ReadLine();

  var blog = new Blog { Name = name };
  db.Blogs.Add(blog);
  db.SaveChanges();

  foreach (var item in db.Blogs)
    Console.WriteLine(item.Name);
}
```



BlogPost.cs

DbContext, DbSet

- EF 4.1 előtt ObjectContext volt
- a `DbContext` híd az entitás osztályok és az AB között
- `DbContext` nyilvántartja az entitásokat, hogy a rajtuk végzett változtatásokat le tudja küldeni az adatbázisba



- Object Materialization: DbContext a nyers adattáblát átkonvertálja entitás objektummá
- minden kéréshez hozzunk létre új DBContextet, mindig `using` - gal használjuk!
- `DbSet` egy adott típusú entitás gyűjtemény

Entitás állapotok

- **Added**: az entitás része a contextnek, de még nem létezik az AB-ban
- **Unchanged**: az entitás része a contextnek és létezik az AB-ban, és a tulajdonságának értéke nem változott az AB-beli értéktől
- **Modified**: az entitás része a contextnek és létezik az AB-ban, és 1/több tulajdonságának értéke megváltozott az AB-beli értéktől
- **Deleted**: az entitás része a contextnek és létezik az AB-ban, de törlésre volt jelölve az AB-beli értéke, amint a SaveChanges meghívódik
- **Detached**: az entitás nem része a contextnek
- az entitás állapota lekérdezhető/módosítható `DbEntityEntry`

```
dbContext.Entry(myBlog).State = System.Data.Entity.EntityState.Deleted;
```

`EntityStates`

Adatbázis kapcsolat

1. paraméter élkül: AB létrejön `Namespace.ContextClass` patternt használva

```
public class BloggingContext: DbContext {
  public BloggingContext(): base() { }
}
```

2. “Name” paraméter: Az AB ezzel a névvel jön létre

```
public class BloggingContext: DbContext {
  public BloggingContext(): base(”MyBloggingContext”) { }
}
```

3. ConnectionStringName: az AB a `web.config` vagy `app.config` connection stringje alapján jön létre

```
public class BloggingContext: DbContext {
  public BloggingContext(string connStr): base(connStr) { }
}
//...
string connStr = ConfigurationManager.
                 ConnectionStrings[”BloggingContext”].
                 ConnectionString;
```

```
<connectionStrings>
  <add name=”BlogginContext”
    connectionString=”Data Source=.;Initial Catalog=MyFabulousDB;Integrated
         Security=true”
    providerName=”System.Data.SqlClient”
  />
</connectionStrings>
```

Adatbázis inicializáló stratégiák

1. `CreateDatabaseIfNotExists` (default)
2. `DropCreateDatabaseIfModelChanges`
3. `DropCreateDatabaseAlways`
4. testreszabott Adatbázis inicializáló

Example:

```
public class BloggingContext: DbContext  {
  public BloggingContext()  {
    Database.SetInitializer<BloggingContext>(
      new CreateDatabaseIfNotExists<BloggingContext>()
    );
...
```

- Adatbázis inicializáló kikapcsolása:

```
Database.SetInitializer<BloggingContext>(null)
```

- **NB**. az Adatbázis inicializáló beállítható a web.config/app.config-ból

Seed a Database

- beszúrhatunk adatokat az inicializáló folyamat alatt (pl. első bejelentkezéshez szükséges infókat)
- ehhez egy egyéni inicializálót kell használni, felülírt `Seed` metódussal

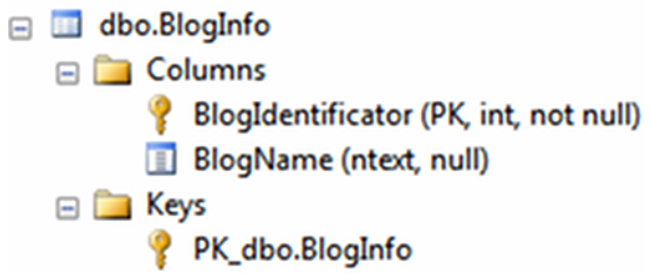
```
//BloggingContext.cs-ben:
public BloggingContext() {
  Database.SetInitializer<BloggingContext>(new BloggingDBInitializer());
}

//BloggingDBInitializer.cs-ben:
public class BloggingDBInitializer :
  DropCreateDatabaseAlways<BloggingContext> {
  protected override void Seed(BloggingContext context) {
    Post p1 = new Post { Content = ”content1”, Title = ”title1” };
    Post p2 = new Post { Content = ”content2”, Title = ”title2” };
    Blog b = new Blog { Name = ”blog1”, Posts = new List<Post> { p1, p2 } };
    context.Blogs.Add(b);
    base.Seed(context);
  }
}
```

Az entitás osztályok konfigurálása

- EF Code First egy bizonyos konvenciót használ az ORM leképezésre (elérhető `DbModelBuilder.Conventions`)
- ha az entitás osztályok nem követik az egyezményt akkor finomítható a beilleszkedése
- 2 lehetőség: DataAnnotation és Fluent API

Az entitás osztályok konfigurálása DataAnnotation-el

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

[Table(”BlogInfo”)]
public class Blog {
  [Key]
  public int BlogIdentificator {get; set;}

  [Column(”BlogName”, TypeName = ”ntext”)]
  [MaxLength(20)]
  public string Name {get; set;}

  [NotMapped] //ne legyen eltárolva
  int? SomeProperty {get; set}

  public virtual List<Post> Posts {get; set;}
}
```



További DataAnnotation-ok

Validációs atribútumok

- `Required`
- `MinLength, MaxLength`
- `StringLength`

Adatbázis séma atribútumok

- `Table, Column`
- `Key, ForeignKey`
- `NotMapped`
- `ComplexType`
- `Timestamp`
- `DatabaseGenerated`

Az entitás osztályok konfigurálása Fluent API-t használva

- bizonyos funkciókat nem lehet kifejezni DataAnnotation-el
- felül kell írni a `DbContext.OnModelCreating` metódust, hogy használhassunk Fluent API-t

```
//in BloggingContext.cs
protected override void OnModelCreating(DbModelBuilder modelBuilder) {
          modelBuilder.Entity<Blog>().ToTable(”BlogInfo”)
                      .HasKey(b => b.BlogIdentificator)
                      .Ignore(b => b.SomeCalculatedProperty);
          base.OnModelCreating(modelBuilder);
}
```

Kapcsolat típusok

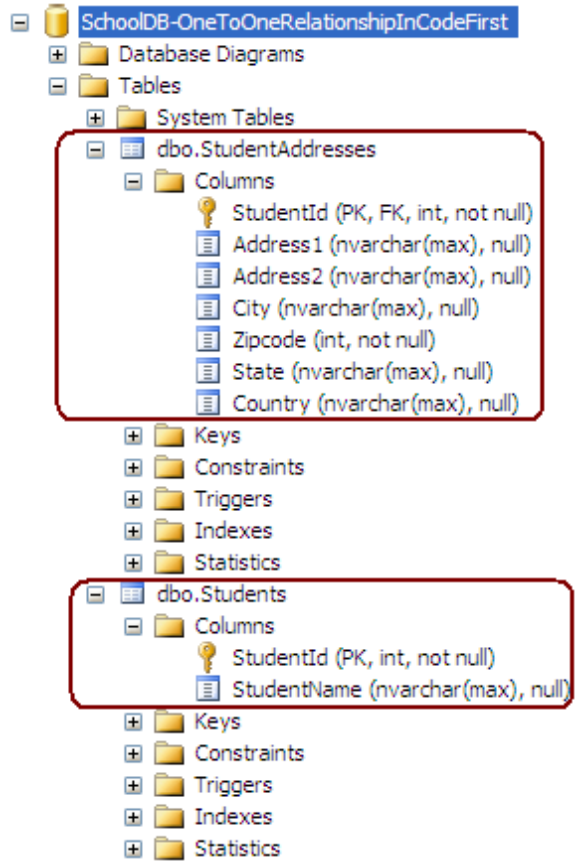
1–1 Kapcsolat

1-1 kapcsolat (DataAnnotation)

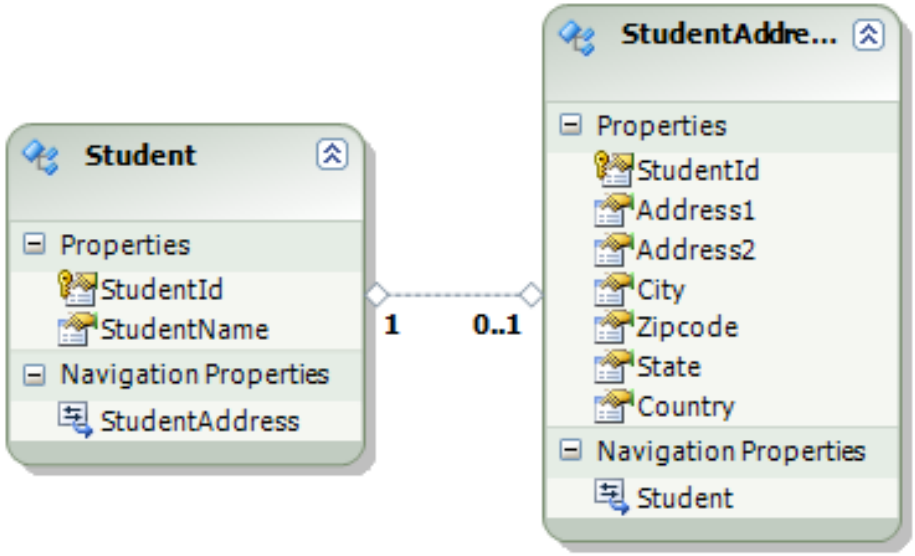
- `Student` — `StudentAddress` : 1–1

```
public class Student
{
  public int StudentId { get; set; }
  public string StudentName { get; set; }

 public virtual StudentAddress StudentAddress{get;set;}
}
```

```
public class StudentAddress
{
    [Key, ForeignKey("Student")]
    public int StudentId { get; set; }

    public string Address1 { get; set; }
    public string Address2 { get; set; }
    public string City { get; set; }
    public int Zipcode { get; set; }
    public string State { get; set; }
    public string Country { get; set; }

    public virtual Student Student { get; set; }
}
```



1-1 EDM



Mi a különbség a E-K diagram és az entitás modell között?

1–0..1 Kapcsolat (Fluent API)

- `Person` — `Passport`: 1–0..1
- `Person`: a **fő** komponense a kapcsolatnak
- `Passport`: a **függő** része a kapcsolatnak

```
class Person {
  public int Id { get; set;}
  public virtual Passport Passport {get; set;}
}
class Passport {
  public int Id { get; set;}
  public virtual Person Owner {get; set;}
}
```

- az első `Add` művelet: InvalidOperationException: Unable to determine the principal end of an association
- megoldás:
  `modelBuilder.Entity<Passport>().HasRequired(pp => pp.Owner);`

`PersonPassport project`

1–1 Kapcsolat (Fluent API)

- `Person` — `BirthCertificate` : 1–1
- `Person` : a **fő** komponense a kapcsolatnak
- `BirthCertificate` : a **függő** része a kapcsolatnak

```
class Person {
  public int Id { get; set;}
  public virtual BirthCertificate BirthCertificate {get; set;}
}
class BirthCertificate {
  public int Id { get; set;}
  public virtual Person Owner {get; set;}
}
...
modelBuilder.Entity<Person>()
            .HasRequired(p => p.BirthCertificate)
            .WithRequiredPrincipal(c => c.Owner);
```

- NB. az ABKR megszorításai miatt, 1–1 kapcsolatból is 1–0..1 lesz még akkor is ha hozzáadjuk a:

```
modelBuilder.Entity<BirthCertificate>().HasRequired(c => c.Owner);
```

Virtual kulcsszó

- a **virtual** kulcsszó a lazy loading-ot/change tracking-et teszi lehetővé
- **virtual** kulcsszó = felülírható a származtatott osztályban, ezzel speciálisabb viselkedést elérve
- lazy loading = ha a navigációs tulajdonság virtual, akkor az EF futásidőben létrehoz egy új osztályt az eredeti osztályból származtatva és ezt használja helyette. Ez az osztály fogja tartalmazni a navigációs logikát
- Lazy loading egy komplex folyamat, amely előírja, hogy az adatok hogyan jöjjenek az AB-ból

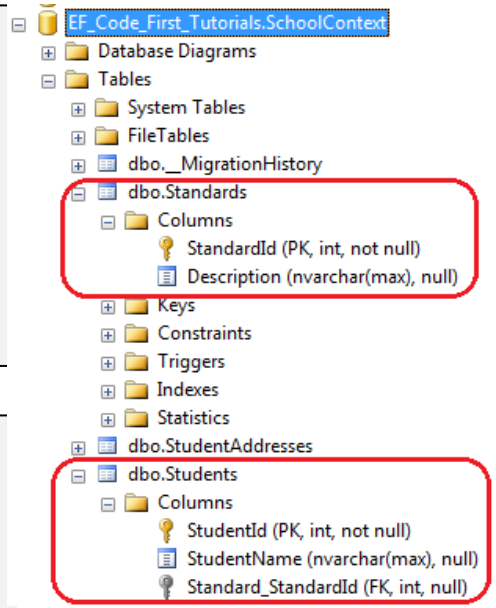
1–N Kapcsolat

1–N Kapcsolat (DataAnnotation)

- `Student` — `Standard` : 1–N

```
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }

    public virtual Standard Standard { get; set; }
}
```

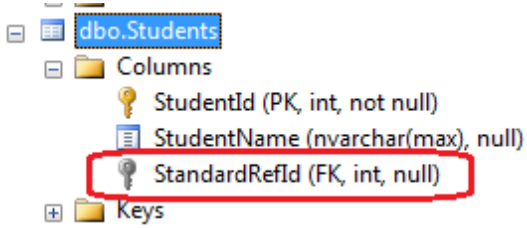
```
public class Standard
{
    public Standard()
    {
        Students = new List<Student>();
    }
    public int StandardId { get; set; }
    public string Description { get; set; }

public virtual ICollection<Student> Students{get;set;}
}
```



FK tulajdonság a modelben

- ajánlott külön propertyként felvenni a külső kulcsot

```
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }

     // FK hozzáadva
    public int StdandardRefId { get; set; }

    [ForeignKey(”StandardRefId”)]
    public virtual Standard Standard {get;set;}
}
```




```
public class Standard
{   public Standard()
    {
        Students = new List<Student>();
    }
    public int StandardId { get; set; }
    public string Description { get; set; }
public virtual ICollection<Student> Students{get;set;}
}
```

Foreign Key asszociáció

- ha a FK része a modelnek: **foreign key association**
- az 1–1 vagy 1–0..1 kapcsolatokban nincs külön FK oszlop, a PK külső kulcsként is viselkedik `PersonPassport`
- létrehozhatunk/változtathatunk egy kapcsolatot módosítva a külső kulcsot

```
public class Blog {
  public int Id { get; set; }
  public string Name { get; set; }
  public virtual ICollection<Post> Posts { get; set; }
}
public class Post {
  public int Id { get; set; }
  public string Title { get; set; }
  public int? BlogId { get; set; }          //FK hozzáadva
  public virtual Blog Blog { get; set; }
}
...
Blog blog = new Blog { Name = ”my fantastic blog” };
Post post = new Post { Title = ”title”, Content = ”content”};
ctx.Blogs.Add(blog);
ctx.Posts.Add(post);
ctx.SaveChanges();
post.BlogId = blog.Id;              //a lényeg itt történik
ctx.SaveChanges();
Console.WriteLine(”post belongs to {0}”, post.Blog.Name);
```

`ModifyingForeignKey`

Független asszociáció

- ha a FK nem része a modelnek: **independent association**
- a navigációs property-t használhatjuk, hogy hozzáadjunk/töröljünk/módosítsunk egy entitást

```
//pl:
course.Department = null; //letöröl egy FK kapcsolatot
//NB. a FK mezo nullable kell legyen
```

Összetett PK és FK

```
public class Passport
{
    [Key]
    [Column(Order=1)]
    public int PassportNumber { get; set; }
    [Key]
    [Column(Order = 2)]
    public string IssuingCountry { get; set; }
    public DateTime Issued { get; set; }
    public DateTime Expires { get; set; }
}
```

```
public class PassportStamp
{
    [Key]
    public int StampId { get; set; }
    public DateTime Stamped { get; set; }
    public string StampingCountry { get; set; }

    [ForeignKey(”Passport”)]
    [Column(Order = 1)]
    public int PassportNumber { get; set; }

    [ForeignKey(”Passport”)]
    [Column(Order = 2)]
    public string IssuingCountry { get; set; }

    public Passport Passport { get; set; }
}
```

Összetett FK - (Fluent API)

```
class Course {
  public int Id { get; set; }
  public string Name { get; set; }
  public virtual Department Department { get; set; }
}

class Department {
  public string School { get; set; }
  public string Specialization { get; set; }
  public ICollection<Course> Courses { get; set; }
}

...
modelBuilder.Entity<Department>()
  .HasKey(d => new { d.School, d.Specialization });

modelBuilder.Entity<Course>()
  .HasRequired(c => c.Department)
  .WithMany(d => d.Courses);
```

CompositeKeys

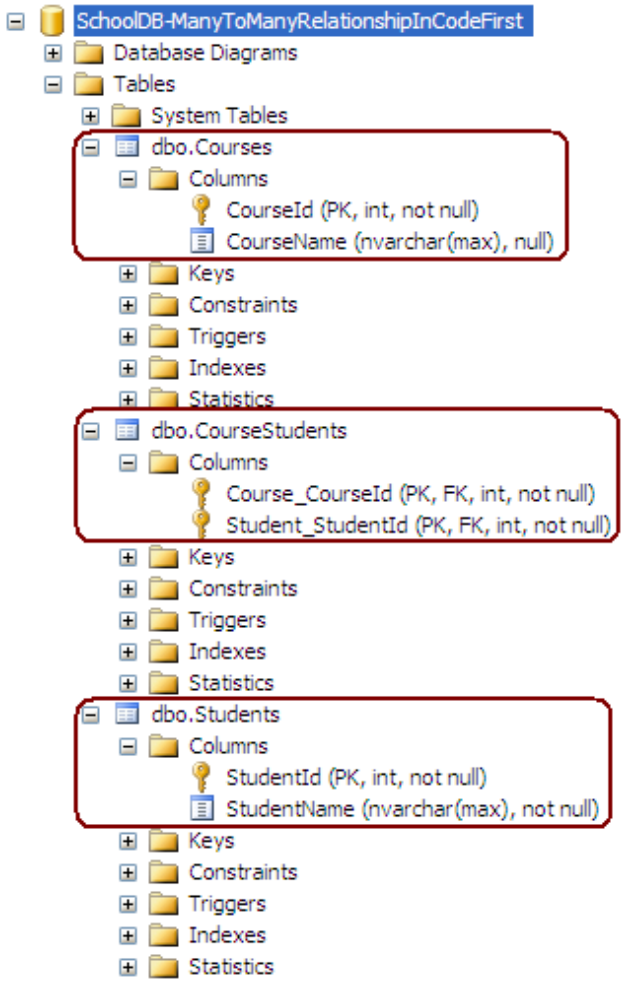
N–N Kapcsolat

N–N kapcsolat (DataAnnotation)

- `Student` — `Course` : N–N

```
public class Student
{   public Student()
    {
         this.Courses = new HashSet<Course>(); //new
              List<Course>(); is helyes, de így
              hatékonyabb
    }
    public int StudentId { get; set; }
    [Required]
    public string StudentName { get; set; }

 public virtual ICollection<Course> Courses{get;set;}
}
```

```
public class Course
{   public Course()
    {
        this.Students = new HashSet<Student>();
    }
    public int CourseId { get; set; }
    public string CourseName { get; set; }

 public virtual ICollection<Student> Students{get;set;}
}
```



N–N kapcsolat (Fluent API)

```
class Instructor {
   public int Id { get; set; }
   public virtual ICollection<Course> Courses { get; set; }
 }
 class Course {
   public int Id { get; set; }
   public virtual ICollection<Instructor> Instructors { get; set; }
 }
 ...
 modelBuilder.Entity<Instructor>().HasMany(i => i.Courses).WithMany(c => c.
      Instructors);
```

- a köztes tábla automatikusan generálódik

`CourseInstructors`

Cascade Delete

konvenció:

- ha egy függő entitás FK-e **nem nullable**, akkor a cascade delete lesz beállítva

```
Person john = new Person { Name = ”John” };
john.BirthCertificate = new BirthCertificate {Owner = john, SerialNr =
    ”00”};
ctx.Persons.Add(john);
ctx.SaveChanges();

//ha John-t töröljük, az ő certificate-je is törlodik
Console.WriteLine(ctx.BirthCertificates.Count()); //1
ctx.Persons.Remove(john);
ctx.SaveChanges();
Console.WriteLine(ctx.BirthCertificates.Count()); //0
```

- ha egy függő entitás FK-e **nullable**, akkor Code First nem használ cascade delete-t a kapcsolatban. Amikor az apa táblában törlünk a fiú táblában null-ra állítódik

Explicit Cascade Delete -Fluent Api

- használjunk `WillCascadeOnDelete` -t, hogy felülírjuk a default viselkedést

```
class Person {
  public int Id { get; set; }
  public string Name { get; set; }
  public virtual Car Car { get; set; }
}
class Car {
  public int Id { get; set; }
  public virtual Person Owner { get; set; }
}
...
modelBuilder.Entity<Person>()
 .HasOptional(p => p.Car)
 .WithOptionalPrincipal(c => c.Owner)
 .WillCascadeOnDelete(true);
```

`CascadeDelete`

Adatbázis migrálás

Migrálás

- EF 4.3+ támogat egy új AB inicializálást, `MigrateDatabaseToLatestVersion`
- ez automatikusan updateli az AB sémát ha a model változik, anélkül hogy elveszítenénk a létező adatokat
- 2 típusú migrálás: automatikus és kódfüggő

Automatikus migrálás

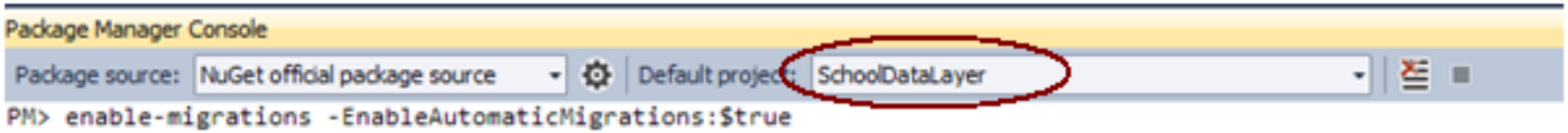
- futtasuk a következő parancsot a Tools –> Library Package Manager –> Package Manager Console:
  `enable-migrations -EnableAutomaticMigration:$true`
- ez létrehozza a `Migrations.Configuration` osztályt
- emelett az AB inicializálót is kell konfigurálni a context létrehozásánál:

```
Database.SetInitializer(
  new MigrateDatabaseToLatestVersion<MyDbContext,
    Migrations.Configuration>());
```

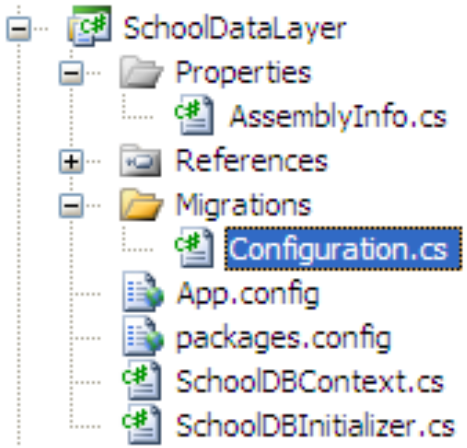
- amíg nincs adatvesztés, addig az autómatikus migrálás jól működik (pl. egy tulajdonságot törlünk egy entitásból)
- ha az adatvesztés megengedett AutomaticMigrationDataLossAllowed = true, AutomaticMigrationsEnabled = true

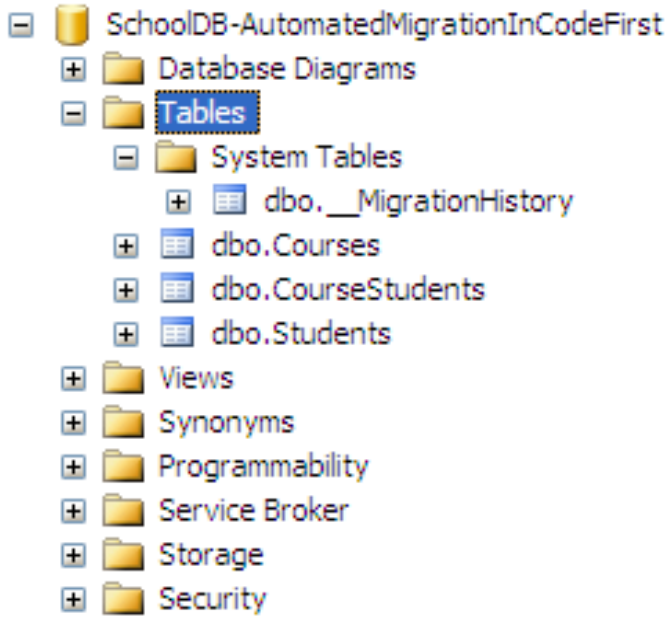
`AutomatedMigration + DB migration history`

Migrálás



forrás:





http://www.entityframeworktutorial.net/code-first/automated-migration-in-code-first.aspx

Kódfüggő Migrálás

Szükséges lépések:

- a Package Manager Console-ban (PMC):

  `enable-migrations`

- állítsuk be az AB inicializálót

```
Database.SetInitializer(
  new MigrateDatabaseToLatestVersion<MyDbContext,
    Migrations.Configuration>());
```

- a PMC-ben

  `add-migration ”Initial-DB-Schema”`

- AB módosítása: PMC-ben

  `update-database`

- AB módosítások visszapörgetése:

  `update-database -TargetMigration:”Initial-DB-Schema”`

`CodeBasedMigration`

Öröklődési stratégiák

Öröklődési stratégiák

1. Öröklési hierarchia egy táblába (table per hierarchy (TPH)) – egy teljes öröklési fa minden eleme ugyanabba a táblába kerül; Minden entitástípusnál meg van adva egy feltétel, mely alapján a típus beazonosítható (discriminator oszlop, pl.: discriminator = ”Person”)
2. Minden típus saját táblába (table per type (TPT)) – alapértelmezett. Egy tábla minden osztálynak
3. Minden valós osztály saját táblába (table per concrete type (TPC)) – ezzel nem foglalkozunk

Öröklődési stratégiák: öröklési hierarchia egy táblába

```
public abstract class BillingDetail {
  public int Id { get; set; }
  public string Owner { get; set; }
  public string Number { get; set; }
}
public class BankAccount : BillingDetail {
  public string BankName { get; set; }
  public string Swift { get; set; }
}
public class CreditCard : BillingDetail {
  public int CardType { get; set; }
  public string ExpireDate { get; set; }
}
public class TPHContext : DbContext {
  public DbSet<BillingDetail> BillingDetails { get; set; }
}
...
using (var ctx = new TPHContext()) {
  var bankacc = new BankAccount{BankName = ”otp”, Owner = ”joe”, ...
  var ccard = new CreditCard{CardType = 22, Owner = ”mary”, ...
  ctx.BillingDetails.Add(bankacc);
  ctx.BillingDetails.Add(ccard);
  ctx.SaveChanges();
}
```

Öröklődési stratégiák: öröklési hierarchia egy táblába- TPH -2

Az eredmény:

Id	Owner	Number	BankName	Swift	CardType	ExpireDate	Discriminator
1	joe	11	otp	00	NULL	NULL	BankAccount
2	mary	22	NULL	NULL	33	2044	CreditCard

Megjegyezni:

- a “Discriminator” oszlop belsőleg az EF használja
- jó teljesítményt nyújt
- az alosztályok tulajdonságainak az oszlopa nullable lesz az AB-ban, függetlenül a NOT NULL megszorítástól
- TPH megsérti a 3NF szabályait (minden a kulcstól függjön)

`TablePerHierarchy`

Öröklődési stratégiák: Minden típus saját táblába -TPT

```
...
  modelBuilder.Entity<BankAccount>().ToTable(”BankAccounts”);
  modelBuilder.Entity<CreditCard>().ToTable(”CreditCards”);
```

Az eredmény:

```
Table: BillingDetails | Table: BankAccount   | Table: CreditCard
----------------------+----------------------+--------------------------
Id  Owner  Number     | Id  BankName  Swift | Id  CardType  ExpireDate
1   joe     11        | 1   otp       00    | 2   33        2044
2   mary    22        |                     |
```

Megjegyezni:

- osztott PK vannak használva
- az SQL séma normalizált
- komplex osztályhierarchiával a teljesítmény szegényesebb, mivel Joinok szükségesek több osztály között is

`TablePerType`

Komplex típusok

```
public class User {
    public int UserId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Address Address { get; set; }
}
public class Address {
    public string Street { get; set; }      //NINCS ID
    public string City { get; set; }        //az Address
    public string PostalCode { get; set; }  //osztályban
}
```

```
public class EntityMappingContext : DbContext
{
    public DbSet<User> Users { get; set; }
}
```

- ha a PK nem vezethető le és nincs specifikálva, akkor a típus **komplex típus**-ként regisztrálódik
- a komplex típusok mindig kötelezőek (pl. a `User` `Address`-e nem lehet null)

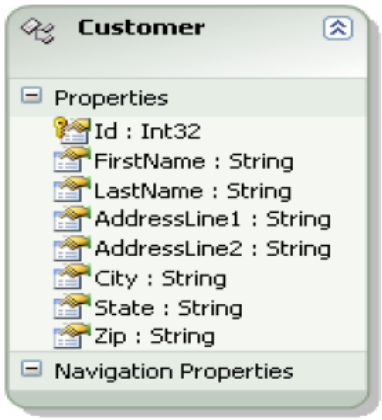
```
UserId|FirstName|LastName|Address_Street|Address_City|Address_PostalCode
------+---------+--------+--------------+------------+------------------
 1    | John    | Luca   | 5th Avenue   | New York   | ZX80
```

ComplexType

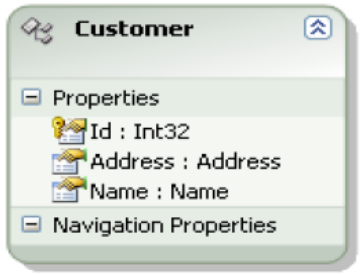
Komplex típusok -2

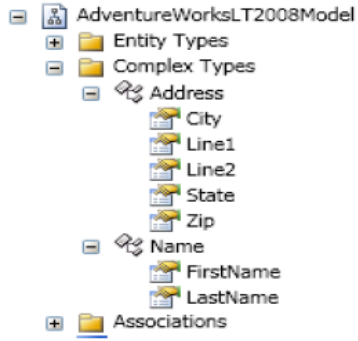
- több skalár (vagy összetett) típusú érték vonható össze egyetlen komplex típusba
  - például cím = ország+város+utca+...
  - összetett típus tartalmazhat másik összetett típust
- az összetett típusok nem lehetnek üresek (null)
- ha az összetett tulajdonságon belül bármelyik skalár érték megváltozik, az egész komplex property megváltozottá válik

Komplex típusok -3



VS.



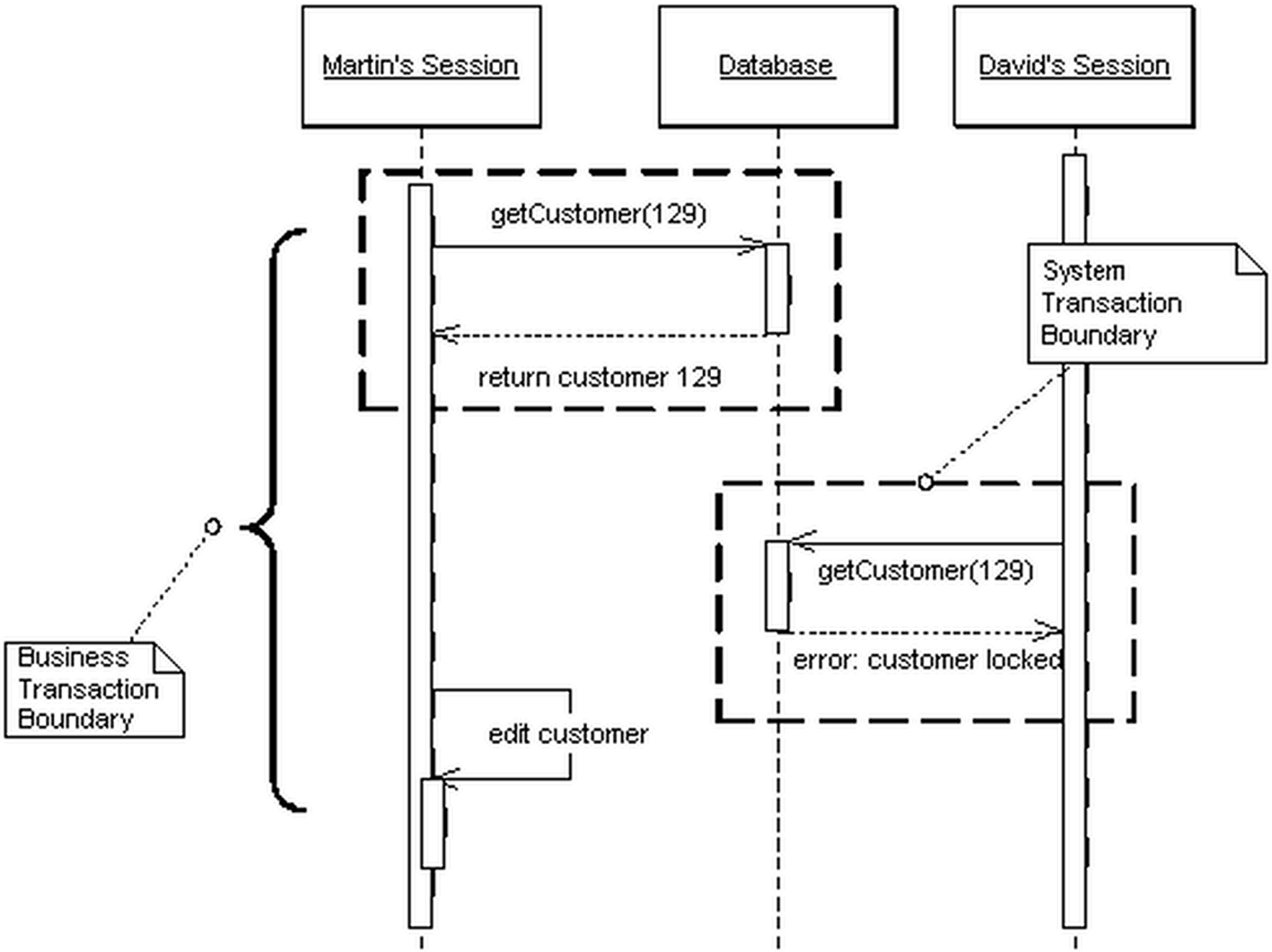


1) 1:1 leképzés
2) Túlzsúfolt

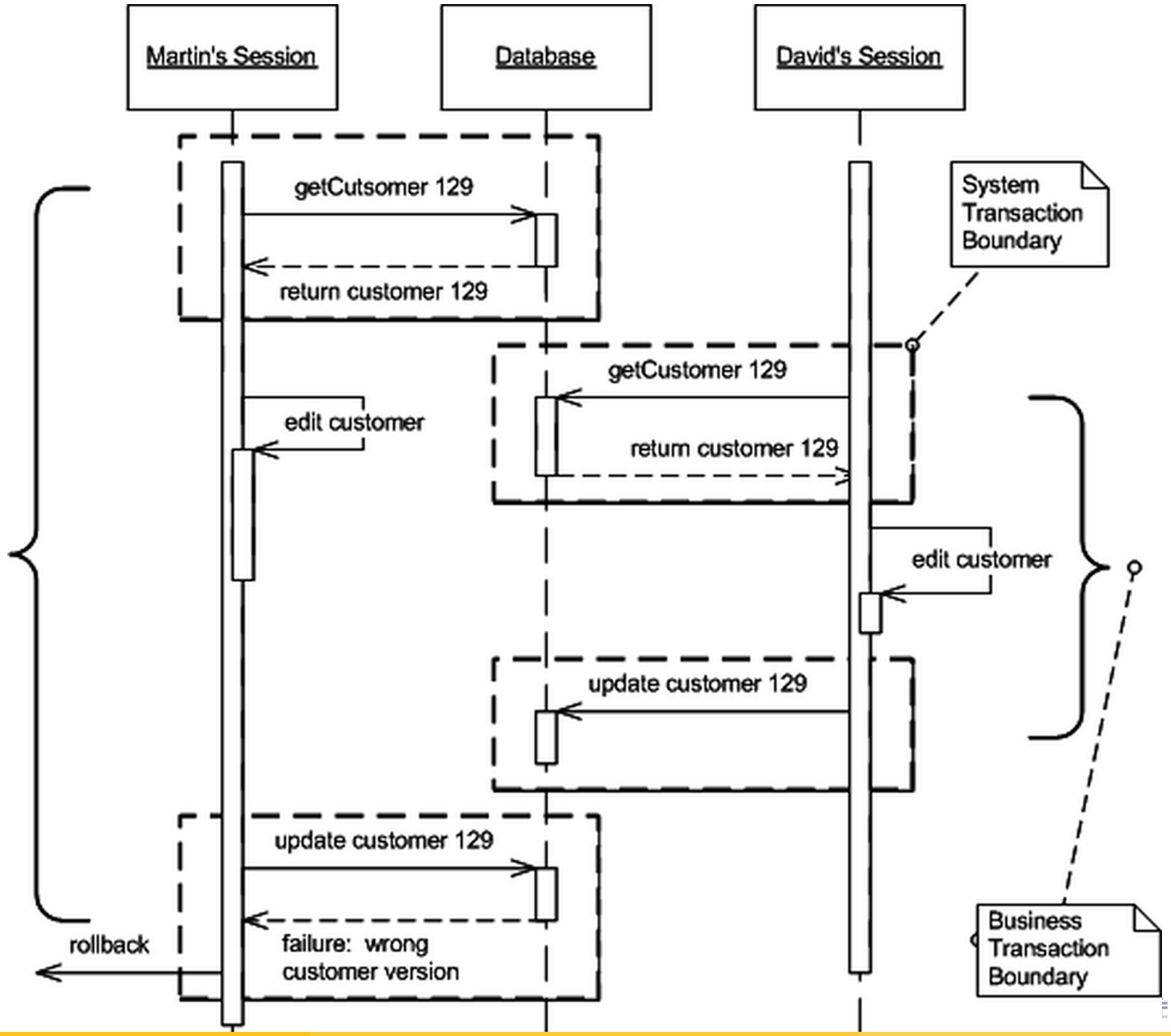
1) Komplex leképzés
2) Rendezett

AB konkurencia: pesszimista zárolás

Megakadályozza konkurens tranzakciók közötti konfliktust, azzal, hogy nem enged csak egyetlen tranzakciót egyidőben hozzáférni az adatokhoz



AB konkurencia: optimista zárolás



Konkurencia Entity Framework-ban

- EF defualtként optimista zárolást használ
- szükséges elemek:

```
public class BankAccount {
  public int Id { get; set; }
  public string Owner { get; set; }
  public decimal Balance { get; set; }
  public byte[] Timestamp { get; set; } //1. hozzáadni Timestamp mezot
}
...
modelBuilder.Entity<BankAccount>()      //2. megmondani az EF-nak, hogy
      Timestamp
  .Property(e => e.Timestamp)           // lesz használva
  .IsRowVersion();                      // a row versioning-hez
```

- RowVersion automatikusan lesz hozzáadva és módosítva Insert/Update műveleteknél
- konkurencia esetén EF érzékelni fogja és kivételt dob `DbUpdateConcurrencyException`

`OptimisticConcurrency`

Tranzakció támogatás (EF6+)

- explicit tranzakció rollback támogatással
- `DbContextTransaction` osztályt kell használni
- fontos metódusok: `Commit(), Rollback()`

```
using (var ctx = new XDbContext()) {
    using (var trans = ctx.Database.BeginTransaction()) {
    try {
        ...
        ctx.SaveChanges();
        ...
        ctx.SaveChanges();
        ...
        transaction.Commit();
    } catch (Exception e) {
        transaction.Rollback();
        ...
    }
}}
```

`EF6TransactionsDemo`

Teljesítmény: IQueryable vs IEnumerable

IEnumerable is a collection of objects in memory that you can enumerate
IQueryable is an expression tree, with ability to enumerate over the final outcome

- Mi a különbség?

```
IQueryable<Customer> cq  = db.Customers.Where(c => c.City == ”Rome”);
IEnumerable<Customer> ce = db.Customers.Where(c => c.City == ”Rome”);
```

- ha tovább finomítjuk a `cq` -t akkor a lekérdezés az AB-ban lesz végrehajtva

```
var goldCustomers = cq.Where(c => c.IsGold);
```

- ha tovább finomítjuk a `ce` az eredeti lekérdezés az AB-ban lesz végrehajtva, és az extra szűrés a memóriában

```
var goldCustomers = ce.Where(c => c.IsGold);
```

- az utóbbi túl sok sor memóriába való beolvasását eredményezi
- IQueryable gyorsabb

Teljesítmény: IQueryable vs IEnumerable 2

ha csak a memórián belüli adathalmazzal dolgozunk akkor az IEnumerable egy jó választás, de ha adatbázisból akkor az IQueryable a jobb választás, mivel csökkenti a hálózati forgalmat, és kihasználja az SQL nyelv erejét

- Mi a különbség?
- használj `Skip` és `Take` metódusokat az `IQueryable` lapozásához

```
IEnumerable<Client> clients = db.Clients.Take(5).ToList();
    // Az összes klienset betölti, majd kivesz belole 5-öt

IQueryable<Client> sameClients = db.Clients.Take(5).ToList();
    // csak az elso 5 kliens töltodik be a memóriába
```

Teljesítmény 2

```
string city =  ”New York”;
List<School> schools = db.Schools.ToList();
List<School> newYorkSchools = schools.Where(s => s.City == city).ToList();
```

fenti helyett hatékonyabb:

```
List<School>  newYorkSchools = db.Schools.Where(s =>s.City == city).ToList();

//vagy ez is
IQueryable<School> schools = db.Schools;
List<School> newYorkSchools = schools.Where(s => s.City == city).ToList();
```

Teljesítmény 3

```
for (int i = 0; i < 2000; i++)
 {
    Pupil pupil = GetNewPupil();
    db.Pupils.Add(pupil);
 }
 db.SaveChanges();
```

- fenti helyett hatékonyabb:

```
var list =  new List<Pupil>();
for (int i =  0; i <  2000; i++)
{
     Pupil pupil = GetNewPupil();
     list.Add(pupil);
}
 db.Pupils.AddRange(list); // NB. EF 6-tól
db.SaveChanges();
```

Change Tracking kikapcsolása

- jelezhetjük, hogy nem áll szándékunkban módósítani a lekért adatokat

```
string city =  ”New York”;
List<School> schools = db.Schools
    .AsNoTracking()
    .Where(s => s.City == city)
    .Take(100)
    .ToList();
```

Change Tracking kikapcsolása 2

```
MyDbContext orderBO = new MyDbContext();
foreach (OrderIDsToProcess orderID in orderIDsToProcess)
    {
    var order = orderBO.Orders.FirstOrDefault(o=> o.OrderID == orderID.OrderID);
    orderID.CustomerID = order.CustomerID;
    }  // futásido 6 óra
```

```
foreach (OrderIDsToProcess orderID in orderIDsToProcess)
{
   MyDbContext orderBO = new MyDbContext(); //különbség itt
    var order = orderBO.Orders.FirstOrDefault(o=> o.OrderID == orderID.OrderID);
    orderID.CustomerID = order.CustomerID;
} // 2 perc
```

- Change Tracking kikapcsolása:

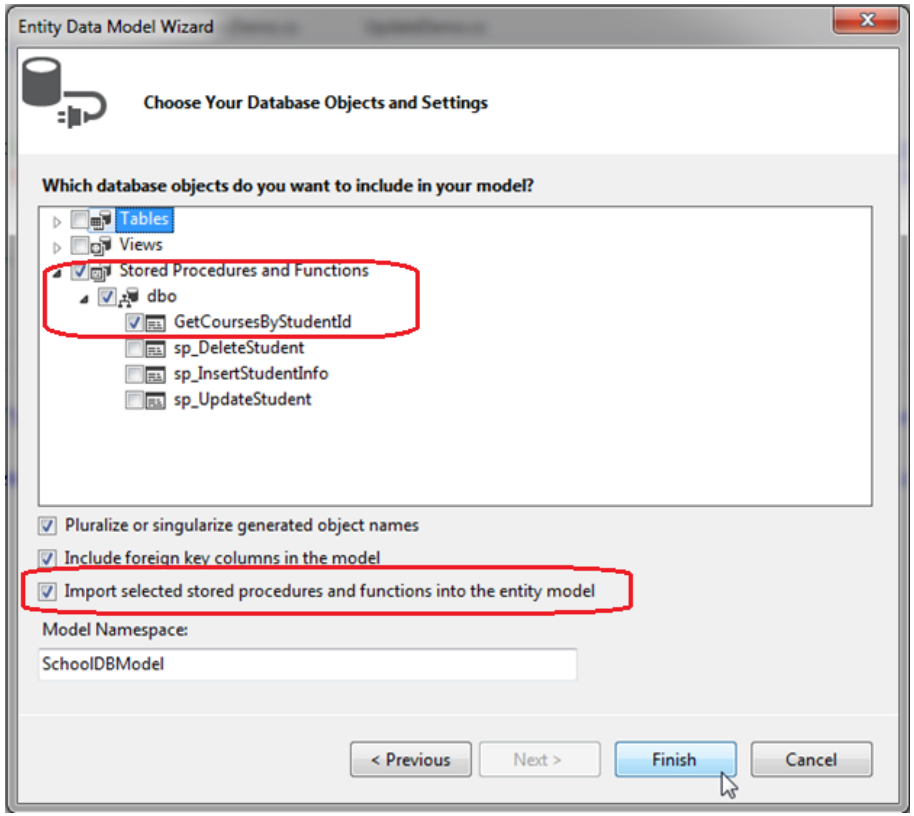
```
 MyDbContext orderBO = new MyDbContext();
orderBO.Configuration.AutoDetectChangesEnabled = false;
foreach (OrderIDsToProcess orderID in orderIDsToProcess)
{
    var order = orderBO.Orders.FirstOrDefault(o=> o.OrderID == orderID.OrderID);
    orderID.CustomerID = order.CustomerID;
} //40 sec
```

Tárolt eljárások

Tárolt eljárások

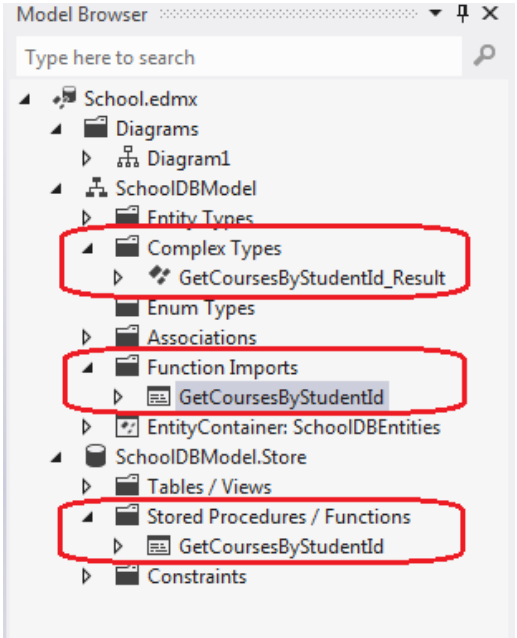
- A tárolt eljárást leképezhetjük
  - ObjectContext függvényeként
    - Entitásokat is adhat vissza

- 1. új ADO.Net Entity Data Model, EF Designer from database



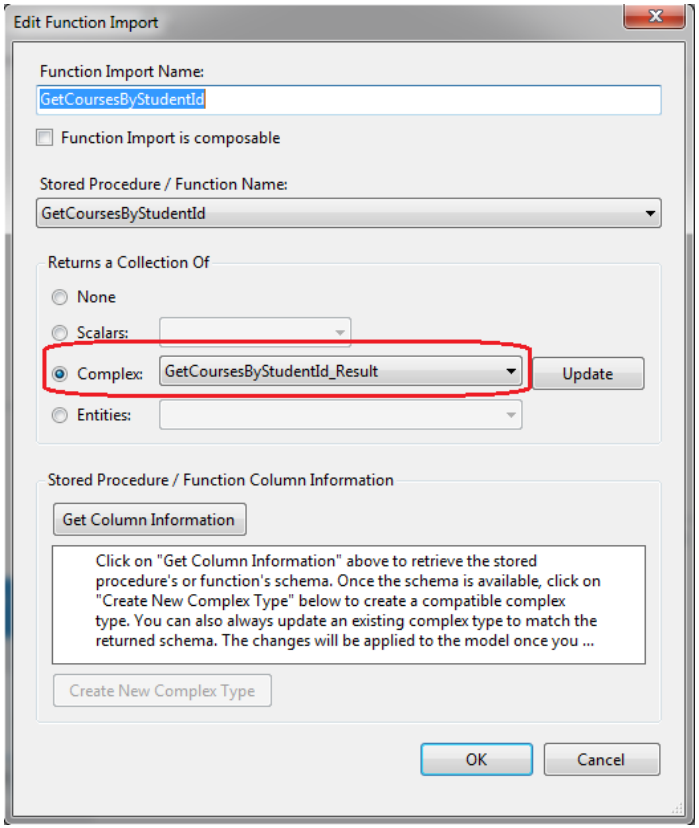
Tárolt eljárások -2

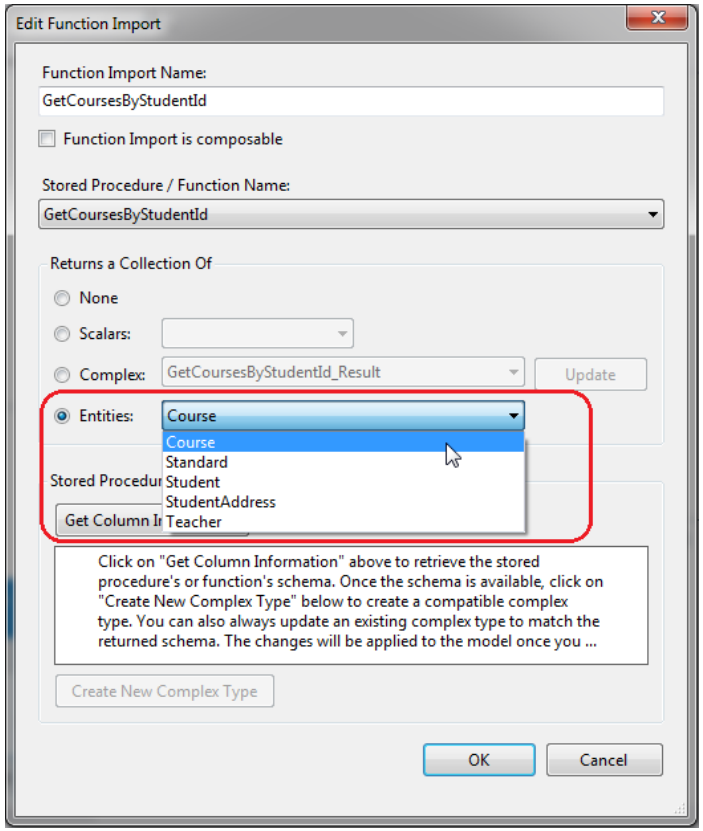
- A Model Browserben létrejön egy komplex típus



Tárolt eljárások -3

- kiválaszthatjuk a visszatérítési típust





Tárolt eljárások -4

- a Contextben létrejön a függvény

```
public partial class SchoolDBEntities : DbContext
{
    public SchoolDBEntities()
        : base("name=SchoolDBEntities")
    {
    }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)...

    public virtual DbSet<Course> Courses { get; set; }
    public virtual DbSet<Standard> Standards { get; set; }
    public virtual DbSet<Student> Students { get; set; }
    public virtual DbSet<StudentAddress> StudentAddresses { get; set; }
    public virtual DbSet<Teacher> Teachers { get; set; }

    public virtual ObjectResult<Course> GetCoursesByStudentId(Nullable<int> studentId)
    {
        var studentIdParameter = studentId.HasValue ?
            new ObjectParameter("StudentId", studentId) :
            new ObjectParameter("StudentId", typeof(int));

        return ((IObjectContextAdapter)this).ObjectContext.ExecuteFunction<Course>("GetCoursesByStudentId", studentIdParameter);
    }
```

- meghívás

```
using (var context = new SchoolDBEntities())
{
    var courses = context.GetCoursesByStudentId(1);

    foreach (Course cs in courses)
        Console.WriteLine(cs.CourseName);
}
```

Upcoming Entity Framework 7 Highlights

- Support for non-relational data stores and even in-memory data for testing.
- Support for machines and devices that don’t use the full .NET Framework. This means you can use EF7 on Linux and Macintosh machines that are running Mono