Eötvös Loránd Tudományegyetem
Informatikai Kar

Webes alkalmazások fejlesztése

1. előadás

Objektumrelációs adatkezelés (Entity Framework Core)

Cserép Máté
mcserep@inf.elte.hu
https://mcserep.web.elte.hu

Objektumrelációs adatkezelés

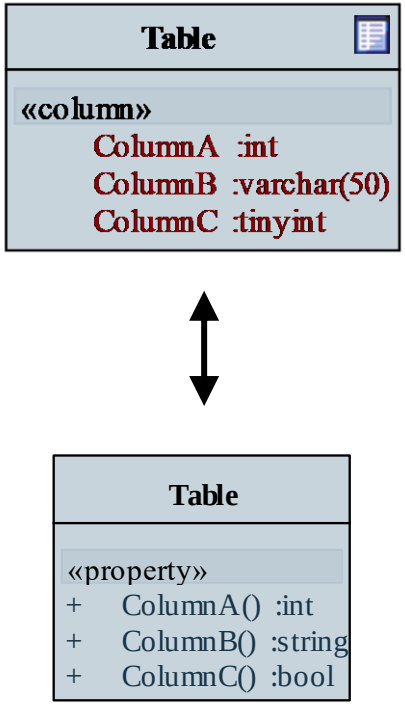
Adatkezelési megoldások

- Az adatbázisok kezelésének több módja adott a .NET Core keretrendszerben
  - *natív kapcsolat*: direkt SQL utasítások végrehajtása a fizikai adatbázison
  - *logikai relációs modell:* a fizikai adatbázis szerveződésének felépítése és adattárolás a memóriában
  - *entitás alapú objektumrelációs modell* (*Entity Framework*): az adatbázis-információk speciális, paraméterezhető leképezése objektumorientált szerkezetre

Objektumrelációs adatkezelés

Entitás alapú objektumrelációs modell

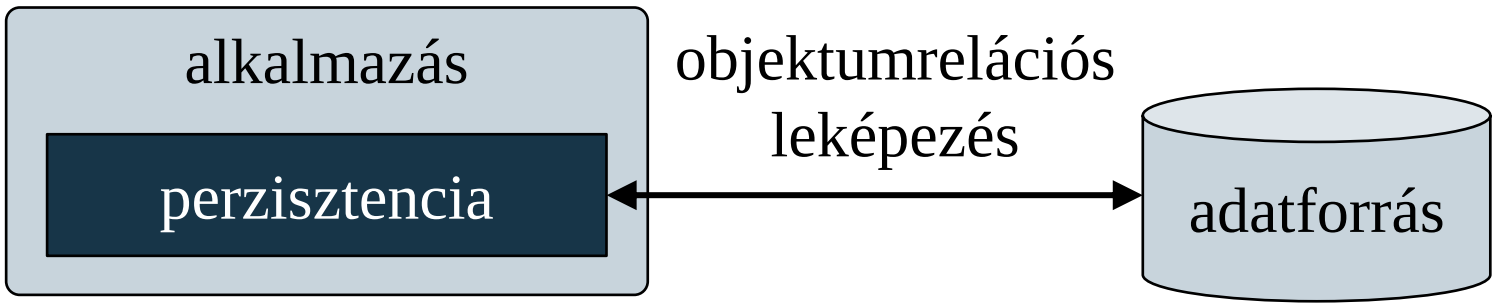
- Az adatkezelő programokat általában objektumorientáltan építjük fel, így célszerű, hogy az adatkezelés is így történjen
- A relációs adatbázisokban
  - az adatokat táblákba csoportosítjuk, amely meghatározza az adatok sémáját, felépítésének módját, azaz *típusát*
  - egy sor tárolja egy adott elem adatait, azaz a sor a típus *példánya*
- Ez a megfeleltetés könnyen átültethető objektumorientált környezetre, a sorok adják az objektumokat, a táblák az osztályokat



Objektumrelációs adatkezelés

Entitás alapú objektumrelációs modell

- A megfeleltetést *objektumrelációs leképezés*nek (*object-relational mapping*, *ORM*) nevezzük
  - magas szintű transzformációját adja az adatbázisnak, amely a programban könnyen használható
  - ugyanakkor szabályozza az adatok kezelésének módját
  - a létrejött osztályok csak adatokat tárolnak, műveleteket nem végeznek



Objektumrelációs adatkezelés

Entity Framework Core

- Az *Entity Framework Core* valósítja meg az adatok platform-független, összetett, objektumrelációs leképezését
  - általában egy *entitás* egy tábla sorának objektumorientált reprezentációja, de ez tetszőlegesen variálható
  - az entitások között kapcsolatok állíthatóak fel, amely lehet asszociáció, vagy öröklődés
  - támogatja a nyelvbe ágyazott lekérdezéseket (LINQ), a dinamikus adatbetöltést, az aszinkron adatkezelést
  - használatához a **`Microsoft.EntityFrameworkCore`** és a specifikus NuGet csomagok projekthez rendelése szükséges (pl. **`Microsoft.EntityFrameworkCore.SqlServer`**)
    - névtere a **`Microsoft.EntityFrameworkCore`**

Objektumrelációs adatkezelés

Entity Framework Core

- modularizált felépítésének előnye, hogy csak a szükséges komponensek, *providerek* betöltése megvalósítható
- támogatottság (teljesség igénye nélkül):

Adatbázis	Entity Framework Core NuGet csomag
MSSQL	Microsoft.EntityFrameworkCore.SqlServer
SQLite	Microsoft.EntityFrameworkCore.Sqlite
MySQL / MariaDB	MySql.Data.EntityFrameworkCore Pomelo.EntityFrameworkCore.MySql
Oracle	Oracle.EntityFrameworkCore
PostgreSQL	Npgsql.EntityFrameworkCore.PostgreSQL

Objektumrelációs adatkezelés

ADO.NET Entity Framework

- Az *Entity Framework Core* elődje a .NET Frameworkben az *ADO.NET Entity Framework*
  - nem modularizált, valamint funkcionalitásban és az egyes típusokban és eljárások szignatúrájában is eltér
  - támogatott, de új fejlesztéseket már nem kap
  - szintén NuGet csomagként (**`EntityFramework`**) vehető használatba a projektünkben
    - névtere a **`System.Data.Entity`**
  - .NET Core-os projekt esetén figyeljünk, hogy az *Entity Framework Core*-t érdemes használni mindenképpen

Objektumrelációs adatkezelés

Entitás adatmodellek létrehozása

- A modell létrehozására három megközelítési mód áll rendelkezésünkre:
  - *adatbázis alapján* (*database first*): az adatbázis-szerkezet leképezése az entitás modellre (az adatbázis séma alapján generálódik a modell)
  - *tervezés alapján* (*model first*): a modellt manuálisan építjük fel és állítjuk be a kapcsolatokat (a modell alapján generálható az adatbázis séma)
  - *kód alapján* (*code first*): a modellt kódban hozzuk létre
- A modellben, illetve az adatbázis sémában történt változtatások szinkronizálhatóak, mindkettő könnyen módosítható

Objektumrelációs adatkezelés

Entitás adatmodellek létrehozása

- Pl. (adatbázis, vásárlók tábla):

```
create table Customer( -- tábla létrehozása
    -- tábla oszlopai
    Email VARCHAR(MAX) PRIMARY KEY,
        -- elsődleges kulcs
    Name VARCHAR(50),
    AddressId INTEGER,

    -- idegen kulcs
    CONSTRAINT CustomerToAddress
        FOREIGN KEY (AddressId)
        REFERENCES Address (Id)
);
```

Objektumrelációs adatkezelés

Entitás adatmodellek létrehozása

- Pl. (adatbázis, címek tábla):

```
create table Address( -- tábla létrehozása
    -- tábla oszlopai
    Id INTEGER PRIMARY KEY,
       -- elsődleges kulcs
    Country VARCHAR(50),
    City VARCHAR(50),
    Address VARCHAR(MAX),
    PostalCode VARCHAR(10)
);
```

Objektumrelációs adatkezelés

Entitás adatmodellek létrehozása

- Pl. (adatbázis, rendelések tábla):

```
create table Order( -- tábla létrehozása
    -- tábla oszlopai
    Id INTEGER PRIMARY KEY,
       -- elsődleges kulcs
    Content VARCHAR(MAX),
    Price FLOAT,
    CustomerEmail VARCHAR(MAX),

    -- idegen kulcs
    CONSTRAINT OrderToCustomer
       FOREIGN KEY (CustomerEmail)
       REFERENCES Customer (Email)
);
```

Objektumrelációs adatkezelés

Entitás adatmodellek létrehozása

- Pl. (kód, vásárlók típus):

```
class Customer // entitástípus létrehozása
{
    [Key] // elsődleges kulcs
    public String Email { get; set; }

    [StringLength(50)] // megszorítás
    public String Name { get; set; }

    [ForeignKey("AddressId")] // idegen kulcs
    public Address Address { get; set; }

    public ICollection<Order> Orders { get; set; }
}
```

Objektumrelációs adatkezelés

Entitás adatmodellek létrehozása

- Pl. (kód, címek típus):

```
class Address // entitástípus létrehozása
{
    [Key] // elsődleges kulcs
    public Int32 Id { get; set; }

    [StringLength(50)] // megszorítás
    public String Country { get; set; }
    [StringLength(50)]
    public String City { get; set; }
    public String Address { get; set; }
    [StringLength(10)]
    public String PostalCode { get; set; }
}
```

Objektumrelációs adatkezelés

Entitás adatmodellek létrehozása

- Pl. (kód, rendelések típus):

```
class Order // entitástípus létrehozása
{
    [Key] // elsődleges kulcs
    public Int32 Id { get; set; }

    public String Content { get; set; }

    public Single Price { get; set; }

    [ForeignKey("CustomerEmail")] // idegen kulcs
    public Customer Customer { get; set; }
}
```

Objektumrelációs adatkezelés

Entitás adatmodellek használata

- Az entitásokat egy adatbázis modell (**`DbContext`**) felügyeli, amelyben eltároljuk az adatbázis táblákat (**`DbSet`**)
  - egy aszinkron modellt biztosít, a változtatások csak külön hívásra (**`SaveChanges`**) mentődnek az adatbázisba
  - pl.:

```
public class SalesContext : DbContext {
    // kezelő létrehozása
    public DbSet<Customer> Customers {
        get; set;
    }
        // adatbázisbeli tábla
    …
}
```

Objektumrelációs adatkezelés

Entitás adatmodellek használata

- Az adattábla (**`DbSet`**) biztosítja lekérdezések futtatását, adatok kezelését
  - létrehozás (**`Create`**), hozzáadás (**`Add`**, **`Attach`**), keresés (**`Find`**), módosítás, törlés (**`Remove`**)
  - az adatokat és a lekérdezéseket lusta módon kezeli
    - az adatok csak lekérdezés hatására töltődnek a memóriába, de betölthetjük őket előre (**`Load`**)
    - a LINQ lekérdezések átalakulnak SQL utasítássá, és közvetlenül az adatbázison futnak

Objektumrelációs adatkezelés

Entitás adatmodellek használata

- Pl.:

```
SalesContext db = new SalesContext();
Customer customer =
    db.Customers.FirstOrDefault(cust =>
        cust.Email == "mcserep@inf.elte.hu");
    // LINQ lekérdezés
if (customer == null)
{
    customer = new Customer {
                    Name = "Cserép Máté",
                    Email = "mcserep@inf.elte.hu" };
    db.Customers.Add(customer);
        // entitás létrehozása és felvétele
    db.SaveChanges(); // változások elmentése
}
```

Objektumrelációs adatkezelés

Csatolt adatok betöltése

- Egy tábla nem tárolja a csatolt adatokat, azok betöltése végezhető mohó (*eager*), explicit vagy lusta (*lazy*) módon
  - Mohó betöltéssel (*eager loading*) az **`Include`** eljárással végezhető, eredményeként az eredeti lekérdezéssel együtt tölthetőek be a csatolt adatok:

```
SalesContext db = new SalesContext();
var customers = db.Customers
  .Where(cust => cust.Email.EndsWith("elte.hu")
  .Include(cust => cust.Address);
```

Objektumrelációs adatkezelés

Csatolt adatok betöltése

- Explicit betöltéssel (*explicit loading*) egy már betöltött entitás objektum csatolt adatait kérhetjük le explicit módon, a **`DbContext`** osztály **`Entry`** metódusával:

```
var customer = db.Customers.Single(
    cust => cust.Email = "mcserep@gmail.com");
db.Entry(customer)
    .Collection(cust => cust.Orders)
    .Load();
```

Objektumrelációs adatkezelés

Csatolt adatok betöltése

- Lusta betöltéssel (*lazy loading*) az adatok akkor kerülnek betöltésre, amikor szükség van rájuk (amikor először kiértékelésre kerülnek)
  - Használatához telepítenünk kell a `Microsoft.EntityFrameworkCore.Proxies` NuGet csomagot és engedélyeznünk a lusta betöltést:

```
var contextOptions = new
  DbContextOptionsBuilder<SalesContext>()
    .UseLazyLoadingProxies()
    .UseSqlServer("...")
    .Options;
var db = new SalesContext(contextOptions);
```

Objektumrelációs adatkezelés

Entitás adatmodellek létrehozása

- A lustán betölteni kívánt navigációs tulajdonságokat virtuálisnak kell jelölnünk.

```
class Customer { // entitástípus létrehozása
    [Key] // elsődleges kulcs
    public String Email { get; set; }
    [StringLength(50)] // megszorítás
    public String Name { get; set; }
    [ForeignKey("AddressId")] // idegen kulcs
    public virtual Address Address { get; set; }
    public virtual ICollection<Order> Orders
    { get; set; }
}
```

Ilyenkor a helyettes tervezési minta (*proxy design pattern*) mentén egy leszármaztatott proxy típussal lesz helyettesítve.

Objektumrelációs adatkezelés

Entitás adatmodellek használata

- A lekérdezéseket előkészítve tárolhatjuk objektumként, amelyre az **`IQueryable<T>`** típus szolgál.
  - A lekérdezések így dinamikusan bővíthetőek és csak kérésre (pl. **`ToList()`** vagy **`Load()`**) vagy kiértékelésre (pl. **`foreach`**) kerülnek végrehajtásra.
  - Az adatbázis műveletek így tisztán objektum orientáltan, C# nyelven megfogalmazhatóak.
- Pl.:

```
IQueryable<Customer> query = db.Customers
    .Where(cust => cust.Email.EndsWith("elte.hu"))
    .Where(cust => cust.Orders.Count() > 0);
query = query.OrderBy(cust => cust.Name);
```

Objektumrelációs adatkezelés

Entitás adatmodellek használata

- Példa lusta kiértékelésre:

```
IQueryable<Customer> query1 = db.Customers
                        .Include(cust => cust.Address);
   // a megadott tulajdonságok (csatolt adatok)
   // is betöltésre kerülnek, hasonlóan
   // táblanévvel: .Include("Address")
IQueryable<Customer> query2 = query1
   .Where(cust => cust.Address.City == "Budapest")
   .OrderBy(cust => cust.Name);
   // a lekérdezés annak végrehajtása nélkül
   // tovább bővíthető

List<Customer> r = query2.ToList(); // kiértékelés
foreach(Customer cust in query2) {/* ... */ }
```

Objektumrelációs adatkezelés

Entitás adatmodellek használata

- Pl.:

```
Boolean anyBudapest1 = query1
    .Any(cust => cust.Address.City == "Budapest");
    // a lekérdezés az adatbázisban fut
Boolean anyBudapest2 = query1
    .Any(cust => cust.Address.City == "Budapest");
    // a lekérdezés továbbra is adatbázisban fut;
    // amennyiben közben változott az adatbázis
    // tartalma, az eredmény is eltérő lehet

query1.Load(); // adatok explicit betöltése
anyBudapest = query1
    .Any(cust => cust.Address.City == "Budapest");
    // a lekérdezés a memóriában fut
```

Objektumrelációs adatkezelés

Entitás adatmodellek használata

- SQL lekérdezés előállítása:

```
IQueryable<Customer> query = db.Customers
    .Where(cust => cust.Email.EndsWith("elte.hu"))
    .Where(cust => cust.Orders.Count() > 0)
    .OrderBy(cust => cust.Name);
Console.WriteLine(query.ToQueryString());
```

```
SELECT *
FROM Customers AS c
WHERE (c.Email LIKE '%elte.hu')
    AND ((SELECT COUNT(*)
          FROM Orders AS o
          WHERE c.Email = o.CustomerEmail)
         > 0)
ORDER BY c.Name
```

Objektumrelációs adatkezelés

Nullable típusok

- Mint azt korábban megismertük, a C# nyelv három típuskategóriát különböztet meg:
  - *érték*: érték szerint kezelendő típusok, mindig másolódnak a memóriában, és a blokk végén törlődnek
  - *referencia*: biztonságos mutatókon keresztül kezelt típusok, a virtuális gép és a szemétgyűjtő felügyeli és törli őket
  - *mutató*: nem biztonságos mutatók, amelyek csak felügyeletmentes (`unsafe`) kódrészben használhatóak
- Az érték szerinti típusok nem vehetnek fel *null* értéket, míg a referencia szerinti típusok igen.

Objektumrelációs adatkezelés

Nullable érték szerinti típusok

- A nyelvbe korán (C# 2.0, 2005) bekerült a *nullable value types* fogalma, melyet a **?** operátor segítségével használhatunk
  - az **int** nullable típusa az **int?**, teljes nevén **Nullable<int>**
- Az ilyen típusok a megszokott értékeiken túl a null értéket is felvehetik
  - **bool?** esetén a változó értéke lehet **true**, **false** vagy **null**
  - a nullable értékek default értéke mindig a **null**.
- A nullable érték szerinti típusok tárolt értékét a **Value** tulajdonságuk adja meg, annak meglétét a **HasValue** jelzi

```
if (x.HasValue) // x típusa int?
    int y = x.Value; // x.Value típusa int
```

Objektumrelációs adatkezelés

Nullable referencia szerinti típusok

- Újabban (C# 8.0, 2019, .NET Core 3) a *nullable reference types* fogalma is elérhető lett a nyelvben, amely elnevezés kicsit félrevezető lehet elsőre
  - A referencia típusok eddig is felvehettek *null* értéket
  - Ha nem kezeltük, hogy egy referencia *null* értéket is felvehet és mégis dereferáltuk (kiértékeltük, metódust hívtunk rajta, stb.), az **`NullReferenceException`**
  - Ez az egyik legelterjedtebb futásidejű hiba típussá vált az elmúlt években.
  - Ebben hivatott segíteni a *nullable reference types*

Objektumrelációs adatkezelés

Nullable referencia szerinti típusok

- Projekt szinten engedélyezhető a *nullable reference types* használata, ekkor megkülönböztetjük a `T` és a `T?` típusokat
  - Engedélyezni a .csproj fájlban tudjuk, Visual Studio 2022 és .NET 6 használata esetén már ez az alapértelmezett:
    `<Nullable>enable</Nullable>`
  - Ha null értéket adunk egy T típusú változónak, akkor fordítási idejű figyelmeztetést kapunk
    ```
    string str = null; // fordítási idejű hiba
    ```
    - Kivétel a *null-forgiving* operátor használata:
      ```
      string str = null!;
      ```
  - Ha elmulasztjuk a null ellenőrzést egy `T?` típusú érték dereferálása előtt, az is fordítási idejű figyelmeztetés

Objektumrelációs adatkezelés

Nullable reference types használata az entitás adatmodellekben

- *Entity Framework Core* használata esetén az entitások tulajdonságait a keretrendszer tölti fel.
  - Elvárható, hogy a tulajdonságok ne null értéket kapjanak
  - Ugyanakkor a *null-forgiving* operátor használata nélkül fordítási hibát kapunk

```
class Order // entitástípus létrehozása
{
    [Key] // elsődleges kulcs
    public Int32 Id { get; set; }
    public String Content { get; set; } = null!;
    public Single Price { get; set; }
    public Customer Customer { get; set; } = null!;
}
```

Objektumrelációs adatkezelés

Nullable reference types használata LINQ-val

- Ha egy lekérdezés eredménye *null* is lehet, akkor a változó típusa *nullable* kell legyen:

```
Customer cust = db.Customers.FirstOrDefault(
  c => c.Email == "mcserep@inf.elte.hu"); // hiba
```

- *Nullable referencia típusú* értékek esetén a kiértékeléskor ellenőrzést kötelező végezni:

```
Customer? cust = db.Customers.FirstOrDefault(
    c => c.Email == "mcserep@inf.elte.hu");
Console.WriteLine(cust.Name); // hiba
```

  - Helyesen:

```
if (cust != null) Console.WriteLine(cust.Name);
```

  - Vagy *null-conditional* operátor használatával:

```
Console.WriteLine(cust?.Name);
```