Eseményvezérelt alkalmazások

10. előadás

Objektumrelációs adatkezelés (Entity Framework)

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

Objektumrelációs adatkezelés

Relációs adatmodell

- A relációs adatbáziskezelő rendszerek (RDBMS) az adatokat táblázatokban (*táblákban*) tárolják
  - A tábláknak vannak oszlopai (*mezők*) és sorai (*rekordok*)
  - A tábla szerkezetének leírását *sémának* nevezzük
- A táblák egyes mezőire *kulcsokat* definiálhatunk, amelyekkel megadhatóak megszorítások (pl. egyedi előfordulás), valamint a lekérdező műveletek hatékonysága növelhető (*indexelés*)
  - A tábláknak lehet egy *elsődleges kulcsa*, amely az adatok elsődleges, fizikai rendezését megadja
- A táblák között kapcsolatok definiálhatóak *idegen kulcsok* segítségével, és elvárható a *hivatkozási épség* ellenőrzése

Objektumrelációs adatkezelés

Microsoft SQL Server

- A Microsoft rendelkezik saját SQL adatbázis-kezelő megoldással, a *Microsoft SQL Server*rel (*MSSQL*)
  - az *SQL Server Management Studio* az alapvető kliens eszköz, de használható Visual Studio is (*View/Server Explorer*, *View/SQL Server Object Explorer, Tools/Sql Server*)
  - saját adatkezelő nyelve van (*Transact-SQL*), amely kompatibilis az SQL szabvánnyal
    - tartalmaz pár speciális utasítást/típust is, pl. automatikus sorszámozást az `IDENTITY` utasítással
  - a felhasználó-kezelés támogatja az egyedi fiókokat és Windows authentikációt

Objektumrelációs adatkezelés

Az ADO.NET Core

- A .NET Core keretrendszerben az adatbázisokkal kapcsolatos adatelérésért az *ADO.NET Core* alrendszer biztosítja
  - elődje a .NET Framework-beli *ADO.NET*, amely pedig *ADO* (*ActiveX Data Objects*)-ből fejlődött ki
  - számos lehetőséget ad az adatok kezelésére, az egyszerű SQL utasítások végrehajtásától az összetett objektumrelációs adatmodellekig
  - az egyes adatbázis-kezelőket külön adapterek (*providere*k) támogatják, amelyek tetszőlegesen bővíthetőek
  - a közös komponensek a **`System.Data`** névtérben, az adatbázis-függő komponensek külön névterekben helyezkednek el (pl. **`System.Data.SqlClient`**, **`System.Data.OleDb`**)

Objektumrelációs adatkezelés

Adatbázis kapcsolat

- Az adatbázis-kapcsolatot egyben, szöveges formában adjuk meg (*connection string*)
  - általában tartalmazza a szerver helyét, az adatbázis nevét, a kapcsolódó adatait (felhasználónév/jelszó)
  - a pontos tartalom adatbázis-kezelőnként változik
  - pl.:

```
"Server=localhost;Database=myDataBase;
 User Id=myUser;Password=myPassword;"
   // SQL Server standard biztonsággal
"Server=127.0.0.1;Port=5432;Database=
 myDataBase;Integrated Security=true;"
   // PostgreSQL Windows authentikációval
```

  - Referencia: [*www.connectionstrings.com*](http://www.connectionstrings.com)

Objektumrelációs adatkezelés

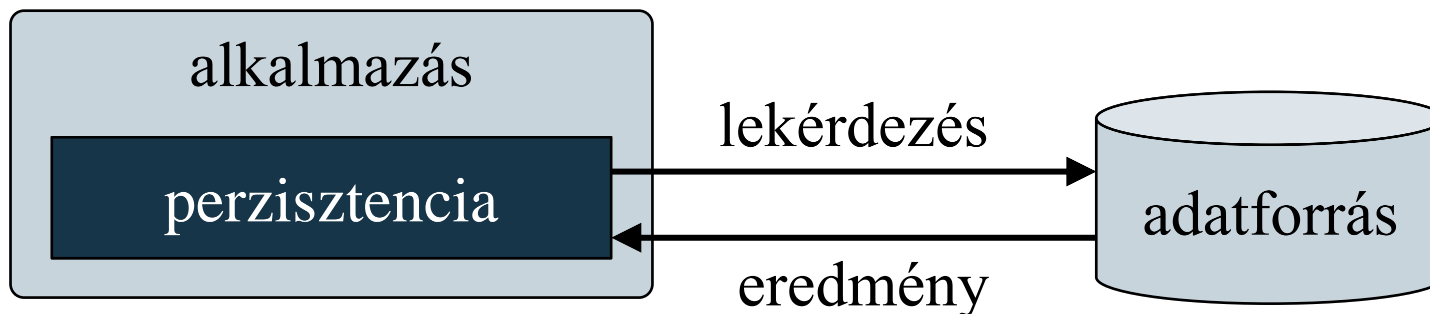
Adatkezelési megoldások

- Az adatbázisok kezelésének több módja adott a .NET Core keretrendszerben
  - *natív kapcsolat*: direkt SQL utasítások végrehajtása a fizikai adatbázison

Objektumrelációs adatkezelés

Natív kapcsolatok

- A *natív (direkt) kapcsolat* lehetővé teszi adatbázis lekérdezések (SQL) végrehajtását a fizikai adatbázison
  - *előnyei*: hatékony erőforrás-felhasználás, közvetlen kommunikáció
  - *hátrányai*: SQL ismerete szükséges, az utasítások a tényleges adatokon futnak (így állandó kapcsolat szükséges az adatbázissal), összetett tevékenységek leírása nehézkes



Objektumrelációs adatkezelés

Natív kapcsolatok

- A kapcsolódást az adatbázishoz az **`SqlConnection`** osztály biztosítja a megfelelő kapcsolati szöveg segítségével, pl.:
  `SqlConnection con = new SqlConnection("…");`
- Az adott kapcsolatban az **`SqlCommand`** osztály segítségével tudunk parancsokat létrehozni
  - a **`CommandText`** tulajdonság tárolja az utasítást
  - a végrehajtás a parancsokra különféleképpen történik
    - az **`ExecuteNonQuery()`** a nem lekérdezés jellegű utasításokat futtatja
    - az **`ExecuteScalar()`** az egy eredményt lekérdező utasításokat futtatja

Objektumrelációs adatkezelés

Natív kapcsolatok

- az **ExecuteReader()** az általános lekérdezéseket futtatja, az eredményt egy **SqlDataReader** olvasóobjektumba helyezi, amellyel soronként olvasunk

- Pl.:

```
SqlCommand command = con.CreateCommand();
command.CommandText = "select * from MyTable";
SqlDataReader reader = command.ExecuteReader();
while (reader.Read()){
   // amíg tudunk olvasni következő sort
   Console.WriteLine(reader.GetInt32(0) + ", "
      + reader.GetString(1));
   // megfelelően lekérjük az oszlopok tartalmát
};
```

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

Objektumrelációs adatkezelés

Logikai relációs modell

- Az adatbázis fizikai szerveződését a programkódban általános típusokra tükrözzük a `System.Data` névtérből:
  - az adatbázisoknak a `DataSet`, a tábláknak a `DataTable` kerül megfeleltetésre,
  - a sorokat a `DataRow`, a mezőket a `DataColumn` típus reprezentálja,
  - relációs kapcsolatok a `DataRelation`, egyéb megszorítások a `Constraint` objektumokkal írhatók le.
- A `DataSet`-be az adatok a `DataAdapter`-en keresztül kerülnek betöltésre és a módosítások szinkronizálásra az adatbázissal.
- Az értékek nem erősen típusozottak (`Object`).

Objektumrelációs adatkezelés

Logikai relációs modell létrehozása

- Pl. (adatbázis):

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

Objektumrelációs adatkezelés

Logikai relációs modell létrehozása

- Pl. (kód):

```
// a connection érvényes SqlConnection objektum
string queryString = "SELECT * FROM Customers";
SqlDataAdapter adapter = new
  SqlDataAdapter(queryString, connection);
DataSet dataSet = new DataSet();
adapter.Fill(dataSet, "Customers");

DataTable table = dataSet.Tables["Customers"];
DataRow newRow = table.NewRow();
newRow["Email"] = "mcserep@inf.elte.hu";
newRow["Name"] = "Máté";
// sor hozzáadása a kollekcióhoz
table.Rows.Add(newRow);
```

Objektumrelációs adatkezelés

Logikai relációs modell lekérdezése

- Pl. (kód):

```
// az összes tábla összes rekordjának kiírása
foreach(DataTable table in dataSet.Tables)
{
   foreach(DataRow row in table.Rows)
   {
      foreach(DataColumn column in table.Columns)
      {
         Console.WriteLine(row[column]);
      }
   }
}
```

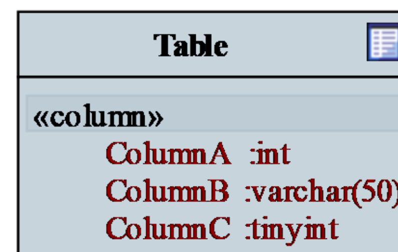
Objektumrelációs adatkezelés

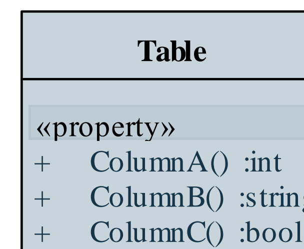
Adatkezelési megoldások

- 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

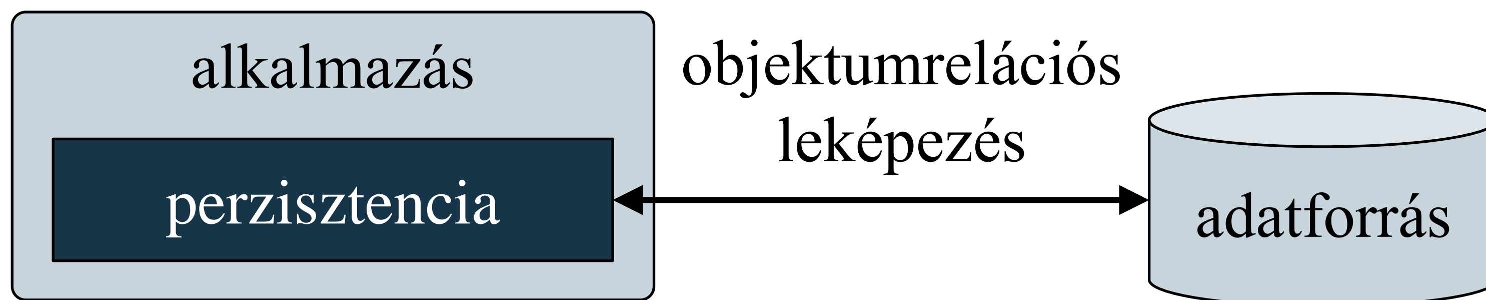
- Az adatkezelő programokat általában objektumorientáltan építjük fel, így célszerű, hogy az adatkezelés is így történjen
- A relációs adatbázisokban
  - az adatokat táblákba csoportosítjuk, amely meghatározza az adatok sémáját, felépítésének módját, azaz *típusát*
  - egy sor tárolja egy adott elem 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

- 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)
);
```

- 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; }
}
```

- 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
  - egy tábla nem tárolja a csatolt adatokat, azok betöltése explicit kérhető (**`Include`**)

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

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);
```

- 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

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

Példa

Feladat: Készítsünk egyszerű grafikus felületű alkalmazást, amellyel megjeleníthetjük, valamint szerkeszthetjük hallgatók adatait. Az adatokat adatbázisban tároljuk.

- a programot *code-first* módon valósítjuk meg, a hallgatókat a **`Student`** entitás típus írja le
- hozzuk létre a **`StudentsContext`** osztályt a **`Microsoft.EntityFrameworkCore.Entity.DbContext`** osztályból származtatva, az adatbázis kontextus leírására
- egészítsük ki a nézetmodellt és a nézetet egy új paranccsal (**`StudentSaveCommand`**), amelyet egy gombhoz kötünk
- az adatbázis kapcsolat *connection string*-jét az **`App.config`** konfigurációs állományban helyezzük el

Objektumrelációs adatkezelés

Példa

Megvalósítás (`Student.cs`):

```
…
public class Student : INotifyPropertyChanged
{
   [Key]
   public Int32 Id { … }
   public String FirstName { … }
   public String LastName { … }
   public String StudentCode { … }
}
…
```

Objektumrelációs adatkezelés
Példa

Megvalósítás (`StudentsContext.cs`*):*

```
public class StudentsContext : DbContext
{
    public StudentsContext(
      DbContextOptions<StudentsContext> options)
      : base(options)
    { }

    public DbSet<Student> Students { get; set; }
}
```

Objektumrelációs adatkezelés

Példa

Megvalósítás (**`App.xaml.cs`**)*:*

```
…
async void App_Startup(object sender, StartupEventArgs e)
{
  var contextOptions = new
      DbContextOptionsBuilder<StudentsContext>()
      .UseSqlServer("...").Options;
  var dbContext = new StudentsContext(contextOptions);
  // adatbázis kontextus létrehozása

  await dbContext.Database.EnsureCreatedAsync();
  var viewModel = new StudentsViewModel(dbContext);
  // nézetmodell létrehozása
}
…
```

Objektumrelációs adatkezelés

Példa

Megvalósítás (`StudentViewModel.cs`*):*

```
public class StudentsViewModel
      : INotifyPropertyChanged
{
   private StudentsContext _context;
   public StudentsViewModel(StudentsContext context)
   {
      _context = context;
      foreach (Student student in _context.Students)
      {
         Students.Add(student);
      }
      // ...
   }
…
```

Objektumrelációs adatkezelés

Példa

Megvalósítás (`StudentViewModel.cs`*):*

```
    public void AddNewStudent()
    {
      Students.Add(NewStudent);
      _context.Students.Add(NewStudent);
      // új hallgató hozzáadása a kontextushoz
    }

    public async Task PersistStudents()
    {
      await _context.SaveChangesAsync();
      // változtatások perzisztálása
    }
  }
```

Objektumrelációs adatkezelés

Példa

Feladat: Készítsünk egy Tic-Tac-Toe programot, amelyben két játékos küzdhet egymás ellen.

- valósítsuk meg a játékállás perzisztálását relációs adatbázissal, *EF Core* felhasználásával, *code-first* módon
- hozzuk létre a **`TicTacToeGame.Persistence.Db`** *.NET Standard* projektet, benne a **`TicTacToeContext`** osztályt az adatbázis kontextus leírására, a tábla mezőit a **`Field`**, a táblát a **`Table`** entitás típus írja le
- egészítsük ki a **`ITicTacToeDataAccess`** interfészt egy **`ListAsnyc()`** eljárással, visszaadja az elérhető mentéseket
- definiáljunk egy új nézetet (**`GameSelectorWindow`**), amelyen keresztül kiválaszthatjuk a betöltendő játékállást

Objektumrelációs adatkezelés

Példa

Megvalósítás (`TicTacToeContext.cs`):

```
public class TicTacToeContext : DbContext
{
   public TicTacToeContext(
     DbContextOptions<TicTacToeContext> options)
     : base(options)
   { }

   public DbSet<Table> Tables { get; set; }
   public DbSet<Field> Fields { get; set; }
}
```

Objektumrelációs adatkezelés

Példa

Megvalósítás (`Table.cs`*):*

```
public class Table
{
  [Key]
  public Int32 Id { get; set; }
  [MaxLength(32)]
  public String Name { get; set; }
  public DateTime Time { get; set; }
  public ICollection<Field> Fields { get; set; }}

  public Table() {
    Fields = new List<Field>();
    Time = DateTime.Now;
  }
}
```

Objektumrelációs adatkezelés

Példa

Megvalósítás (`Field.cs`):

```
public class Field
{
   [Key]
   public Int32 Id { get; set; }

   public Player Player { get; set; }

   public Table Table { get; set; }
}
```

Objektumrelációs adatkezelés

Példa

Megvalósítás (`TicTacToeDbDataAccess.cs`*):*

```
public class TicTacToeDbDataAccess
      : ITicTacToeDataAccess
{
  private TicTacToeContext _context;

  public TicTacToeDbDataAccess(
        DbContextOptions<TicTacToeContext> options)
  {
    _context = new TicTacToeContext(options);
    _context.Database.EnsureCreated();
  }
…
```

Objektumrelációs adatkezelés

Példa

Megvalósítás (`TicTacToeDbDataAccess.cs`*):*

```
public async Task<Player[]> LoadAsync(String name){
    try  {
        Table table = _context.Tables
                .Include(t => t.Fields)
                .Single(t => t.Name == name);
        return table.Fields
                .OrderBy(f => f.Id)
                .Select(f => f.Player)
                .ToArray();
    }
    catch {
        throw new TicTacToeDataException("...");
}
…
```