Entity Framework Core 2

Albert István

ialbert@aut.bme.hu

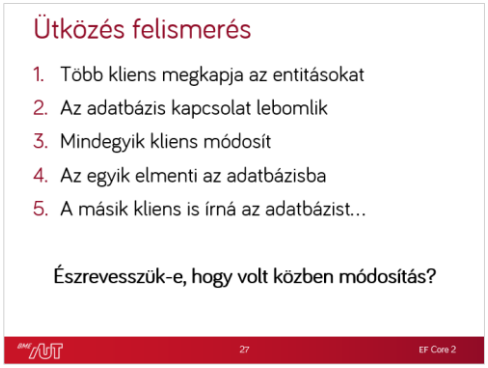
Q.B. 221, 1662

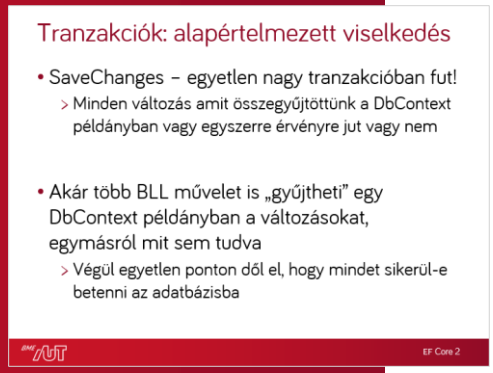
BME AUT

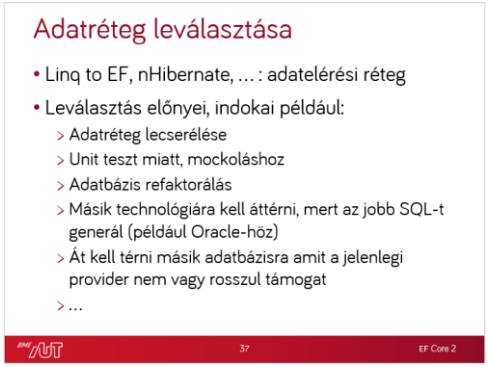
Automatizálási és
Alkalmazott
Informatikai Tanszék

Tartalom


Több rétegű alkalmazások
Böngésző
• HTML renderer
• REST+JSON kliens
Üzleti logika
• Üzleti objektumok
• WCF/REST kliens
Adatbázis
• Hatékony adattárolás
Vastag kliens
• WCF/REST kliens
• JSON / XML
Web UI
• HTML generálás
EF Core 2


Ütközés felismerés
1. Több kliens megkapja az entitásokat
2. Az adatbázis kapcsolat lebomlik
3. Mindegyik kliens módosít
4. Az egyik elmenti az adatbázisba
5. A másik kliens is írná az adatbázist...
Észrevesszük-e, hogy volt közben módosítás?
27
EF Core 2


Tranzakciók: alapértelmezett viselkedés
• SaveChanges – egyetlen nagy tranzakcióban fut!
> Minden változás amit összegyűjtöttünk a DbContext példányban vagy egyszerre érvényre jut vagy nem
• Akár több BLL művelet is „gyűjtheti" egy DbContext példányban a változásokat, egymásról mit sem tudva
> Végül egyetlen ponton dől el, hogy mindet sikerül-e betenni az adatbázisba
EF Core 2


Adatréteg leválasztása
• Linq to EF, nHibernate, ... : adatelérési réteg
• Leválasztás előnyei, indokai például:
> Adatréteg lecserélése
> Unit teszt miatt, mockoláshoz
> Adatbázis refaktorálás
> Másik technológiára kell áttérni, mert az jobb SQL-t generál (például Oracle-höz)
> Át kell térni másik adatbázisra amit a jelenlegi provider nem vagy rosszul támogat
> ...
37
EF Core 2

Kérdések?

Albert István
ialbert@aut.bme.hu

Több rétegű alkalmazások

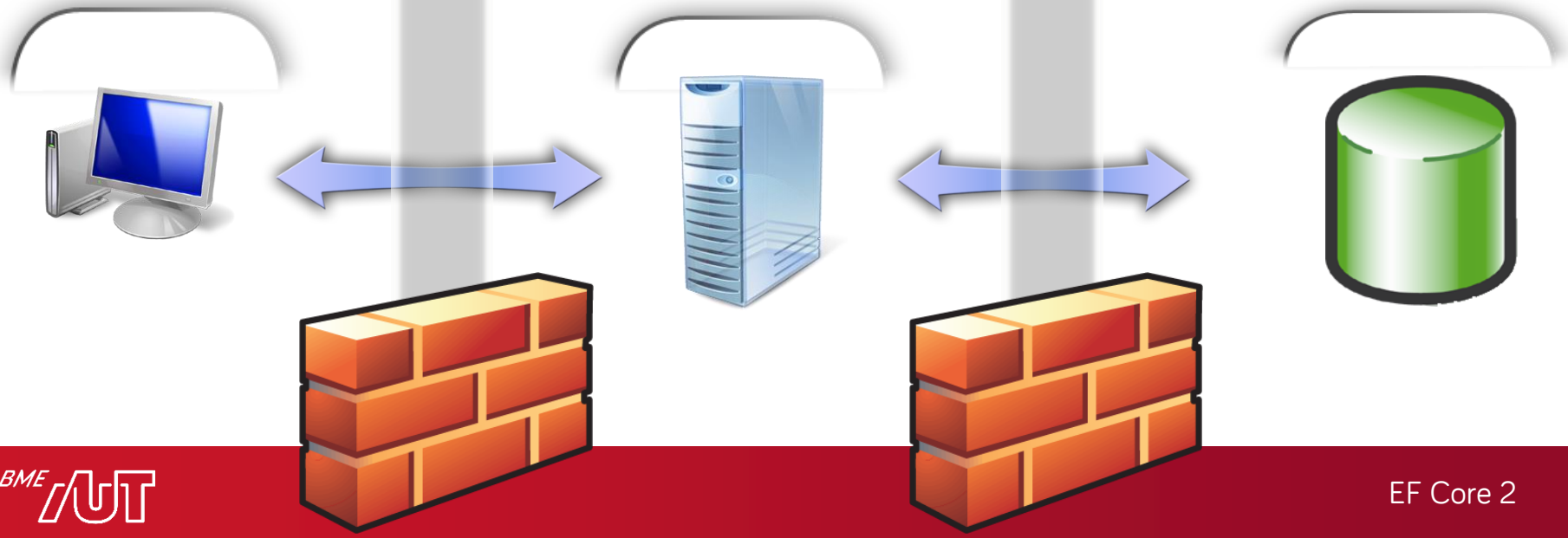
Böngésző
- HTML renderer
- REST+JSON kliens

Üzleti logika
- Üzleti objektumok
- WCF/REST kliens

Adatbázis
- Hatékony adattárolás

Vastag kliens
- WCF/REST kliens
- JSON / XML

Web UI
- HTML generálás



Mitől többrétegű a többrétegű….

- Többrétegű: angol terminológia - „n-tier”
- **Layer**: logikai réteg, a kód felépítését jellemzi, nem feltétlenül jelent fizikai gép határt
- **Tier**: a kód futtatásának fizikai helye, a logikai rétegek elhelyezése számítógépeken
- Számunkra most az N-rétegű:
  - Adatbázis
    + üzleti logikát szolgáltatásként publikáló réteg(ek)
    + kliens alkalmazás(ok)

‚Layers and tiers' – rétegek és szintek

- Layer (tipikusan)
  - Logikai struktúrák, tipikusan külön projektek
  - Absztrakciós szinteket vezetnek be
  - Általában nem mozgathatóak szabadon a tierek között
  - Egy processzben több layer is van
- Tier (tipikusan)
  - Fizikai géphatárokat jelölnek
  - Gyakran bizalmi határvonal is (trust boundary)
  - Meghatározza az oldalra skálázás egységeit
  - Hatással van a kommunikációs- és gyorsítótár architektúrára

Két- és többrétegű alkalmazások

- Kétrétegű alkalmazás (UI + DB)
  - Az entitások nem hagyják el a DbContext alkalmazástartományát (nincs sorosítás)
  - Az állapotkezelést a DbContext megoldja ha a funkciók kompaktak (például egy dialógus ablak)
- Többrétegű alkalmazás
  - Az entitás kikerül a DbContext alkalmazástartományából
  - Az állapotkövetés nem bízható a DbContext-re
  - **A változásokat a DbContext tudomására kell hozni**

A DbContext néhány tulajdonsága

- Az DbContext nem szálbiztos
- Példányosított DbContext tipikusan nyitott adatbázis kapcsolatot jelent
- A DbContext objektum tár nem arra lett tervezve, hogy nagy mennyiségű objektumot hatékonyan kezeljen (nem objektum adatbázis)
  - Az entitások nem törlődnek belőle automatikusan

EF használati minta

- Az DbContextet egyetlen funkció lefutásához példányosítjuk, majd engedjük el
  - Nincs konkurencia probléma
  - Nincs sok objektum
  - Az adatbázis kapcsolat minimális ideig van nyitva
- Használjunk ‘using’-ot!
- A kliens szolgáltasson adatot a módosításokhoz
  - Például az új objektummal együtt az eredeti objektumot is, de legalább az időbélyeget
- Ahol lehet, használjuk a NoTracking opciót

Entitások felvétele a DbContextbe

- DbContext-en vagy DbSet-en hívott
  - Attach / Add / Update / Remove
    - És ezek …Range alternatívái
  - Lekérdezések NoTracking opció nélkül
- A kapcsolódó objektumok is csatolásra kerülnek
  - Egy entitás egy időben egyetlen DbContexthez lehet rendelve

Add metódus

- A gyökér entitás **Added** állapotba kerül
- Minden **eddig nem követett** entitás a gráfban Added állapotba kerül

```
using (var db = new BlogContext())
{
    var oldPost = db.Posts.First();
    var b = new Blog { Url = "https://blogs.msdn.microsoft.com/visualstudio/" };
    b.Posts.Add(new Post { Title = "Apply Now for Microsoft's Go Mobile Tech Workshops" })
    b.Posts.Add(oldPost);
    db.Blogs.Add(b);
    db.DumpTrackedEntities();
```

```
Post: Modified - PostId: 14, Blog: -2147482646, Title: Apply Now for Mic
Blog: Added - BlogId: -2147482646, Url: https://blogs.msdn.microsoft.com
Post: Added - PostId: -2147482645, Blog: -2147482646, Title: Apply Now f
```

Remove metódus

- A gyökér entitás **Deleted** állapotba kerül
- A többi entitás állapota nem változik egészen a SaveChanges hívásig…

```
using (var db = new BlogContext())
{
    var blog = db.Blogs.Include(b=>b.Posts).First();
    db.Blogs.Remove(blog);
    db.DumpTrackedEntities();
```

```
Blog: Deleted - BlogId: 17, Url: https://blogs.msdn.microsoft.com/develop
Post: Unchanged - PostId: 17, Blog: 17, Title: Apply Now for Microsoft's
Post: Unchanged - PostId: 18, Blog: 17, Title: New Year, New Dev - Windo
```

Update metódus

- A gyökér entitás **Modified** állapotba kerül
- Minden **eddig nem követett** entitás a gráfban:
  - **Modified** állapotba kerül ha ki van töltve az elsődleges kulcs
  - **Added** állapotba kerül, ha nincs kitöltve az elsődleges kulcs
- Az összes tulajdonság módosítottként jelenik meg – néha lehetünk ennél hatékonyabbak!

Update példa

- AsNoTracking: a DbContext nem követi az entitást
  - Mintha kódból hoztuk volna létre.
  - Egy interfészen kaptuk meg sorosítás után.

```
using (var db = new BlogContext())
{
    var blog = db.Blogs.AsNoTracking().Include(b => b.Posts).First();
    db.DumpTrackedEntities();
    db.Blogs.Update(blog);
    db.DumpTrackedEntities();
```

```
UpdateSample
DbContext has no tracked entities.
Blog: Modified - BlogId: 19, Url: https://blogs.msdn.microsoft.com/develo
Post: Modified - PostId: 21, Blog: 19, Title: Apply Now for Microsoft's G
Post: Modified - PostId: 22, Blog: 19, Title: New Year, New Dev - Windows
```

Attach metódus

- A gyökér entitás **Unchanged** állapotba kerül
- Minden **eddig nem követett** entitás a gráfban:
  - **Unchanged** állapotba kerül ha ki van töltve az elsődleges kulcs
  - **Added** állapotba kerül, ha nincs kitöltve az elsődleges kulcs
- Az összes tulajdonság módosítottként jelenik meg – néha lehetünk ennél hatékonyabbak!

Attach példa

- A már betöltött entitáson automatikusan felismeri a változtatást

```
using (var db = new BlogContext())
{
    var oldPost = db.Posts.First();
    oldPost.Title = "An other title";
    var blog = db.Blogs.First();
    blog.Posts.Add(oldPost);
    blog.Posts.Add(new Post { Title = "New Year, New Dev – Windows IoT Core" });
    db.Blogs.Attach(blog);
    db.DumpTrackedEntities();
```

```
AttachSample
Post: Modified - PostId: 25, Blog: 21, Title: An other title
Blog: Unchanged - BlogId: 21, Url: https://blogs.msdn.microsoft.com/develop
Post: Added - PostId: -2147482644, Blog: 21, Title: New Year, New Dev - Wir
```

Egyetlen tulajdonság módosítása

- Ha az automatikus összehasonlítás nem jó, mert nincs meg az adatbázis állapot

```
using (var db = new BlogContext())
{
    var blog = db.Blogs.AsNoTracking().First();
    blog.Url = "https://google.com";
    var entry = db.Blogs.Attach(blog);
    db.DumpTrackedEntities();
    entry.Property(b => b.Url).IsModified = true;
    db.DumpTrackedEntities();
    foreach (var e in db.ChangeTracker.Entries())
        foreach (var p in e.Properties)
            if (p.IsModified)
                Console.WriteLine($"Modified property: {p.Metadata.Name}, " +
                    $"original value: {p.OriginalValue}, " +
                    $"new value: {p.CurrentValue}");
```

```
PropertyModificationSample
DbContext has no tracked entities.
Blog: Modified - BlogId: 22, Url: https://google.com, Posts:
Modified property: Url, original value: https://google.com, new value: https://goog
le.com
```

Tulajdonságok állítása egyszerre

- EntityEntry . CurrentValues
  . SetValue(currentEntity)
  - Beleírja a *currentEntity* tulajdonságainak értéket a csatolt entitásba és módosítottként megjelelöli a propertyket, amik különböztek az *eredeti* értékektől.

```
myDbContext.Entry(originalEntity).CurrentValues.SetValues(currentEntity);
```

Automatikus változás jelzés

- Összehasonlítja a DbContextben tárolt adatokat az objektum aktuális adataival
  - Az adat tárolás nem hivatkozással történik
  - Az adatai „lemásolódnak”

Többrétegű alkalmazások – állapotkövetés

- Többrétegű alkalmazás
  - Az entitás kikerül a DbContext alkalmazástartományából
  - Az állapotkövetés nem bízható a DbContextre
  - A változásokat a DbContext tudomására kell hozni
- Néhány megoldást mutatunk, de természetesen további lehetőségek is vannak

Egyetlen entitás van az interfészen

- Általában a metódus neve is jelzi, hogy mit kell tenni az entitással
  - Például: UpdateAddress, AddAddress stb.

- Módosításra példa:
  - Update(modifiedEntity)
  - SaveChanges

- Ütközés vizsgálat -> lásd később!

Objektum gráf jön az interfészen

- A fában minden entitás lehet új vagy módosított
- A törölt entitások nincsenek meg! ☹
- Naív Update hívás
  - Minden új entitás Added állapotú – OK
  - Minden entitás Modified lesz – az is, ami egyébként nem módosult! ☹
  - A törölt entitásokról nem tudunk egyáltalán ☹
- A kliensnek több információt kell átadnia, például egy állapotot is entitásonként!

Megoldási alternatívák

1. A kliens átadja az eredeti gráfot is, a szerver kiszámolja a különbséget.
2. A szerver az adatbázisból lekérdezi az aktuális gráfot és összehasonlítja a klienstől kapottal.
3. A kliens entitásonként átadja az állapotot is – töröltekkel együtt.

Ütközés felismerés

1. Több kliens megkapja az entitásokat
2. Az adatbázis kapcsolat lebomlik
3. Mindegyik kliens módosít
4. Az egyik elmenti az adatbázisba
5. A másik kliens is írná az adatbázist…

Észrevesszük-e, hogy volt közben módosítás?

Ütközés felismerés EF-fel

- Minden UPDATE és DELETE parancs tartalmaz egy WHERE feltételt, ami ellenőrzi, hogy a rekord nem változott meg a legutolsó lekérdezés óta
- Concurrency tokenek használatával
  - Bármelyik saját property lehet ilyen
- Timestamp segítségével
  - Az adatbázis automatikusan megnöveli minden módosítás során
  - Adatbázis motor / provider függő

Konfigurálás

- ConcurrencyCheck attribútum bármelyik propertyn

```
public class Person
{
  public int PersonId { get; set; }
  [ConcurrencyCheck]
  public string LastName { get; set; }
}
```

- Timestamp attribútum

```
public class Blog
{
  public int BlogId { get; set; }
  [Timestamp]
  public byte[] Timestamp { get; set; }
}
```

Ütközés felismerés és kezelés

1. DbUpdateConcurrencyException kivételt kapunk
   - Az Entries tartalmazza az ütköző entitások listáját
2. Lekérdezhetjük az adatbázis aktuális állapotot (például NoTracking opcióval)
3. Ismerni fogjuk
   - A menteni kívánt adatokat
   - Az adatbázis jelenlegi tartalmát
   - (Az általunk korábban lekérdezett adatokat)

Ütközés kezelés

- **Az első író nyer**
- Az utolsó író nyer
- A felhasználóra bízzuk (?)
- Összevetjük a módosított mezők halmazát
  - A konzisztenciát meg kell őrizni
- Az esetek túlnyomó részében nem egyetlen táblát érintenek a változások, hanem több rekordot!

Tranzakciók: alapértelmezett viselkedés

- SaveChanges – egyetlen nagy tranzakcióban fut!
  - Minden változás amit összegyűjtöttünk a DbContext példányban vagy egyszerre érvényre jut vagy nem
- Akár több BLL művelet is „gyűjtheti” egy DbContext példányban a változásokat, egymásról mit sem tudva
  - Végül egyetlen ponton dől el, hogy mindet sikerül-e betenni az adatbázisba

Példa

```
Entity count in db before Tx: 1
Entity count in db before first SaveChanges: 1
Entity count in db before second SaveChanges: 2
Entity count in db during Tx in active DbContext instance: 3
Entity count in db during Tx in the other DbContext instance: 1
Entity count in db after Tx in the other DbContext instance: 3
```

```
using (var db = new BlogContext())
{
    Console.WriteLine($"Count before Tx: {db.Blogs.Count()}");
    using (var transaction = db.Database.BeginTransaction())
    {
        try
        {
            db.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
            Console.WriteLine($"Count before first SaveChanges: {db.Blogs.Count()}");
            db.SaveChanges();

            db.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/visualstudio" });
            Console.WriteLine($"Count before second SaveChanges: {db.Blogs.Count()}");
            db.SaveChanges();

            Console.WriteLine($"Count during Tx in active DbContext instance: {db.Blogs.Co
            using (var db2 = new BlogContext())
            {
                Console.WriteLine($"Count during Tx in the other DbContext instance: {db2.
                transaction.Commit();
                Console.WriteLine($"Count after Tx in the other DbContext instance: {db2.E
            }
        }
```

Adatbázis kapcsolat megosztása

- Relációs adatbázisok esetén!
- A tranzakció DbConnection-höz tartozik
- Mindegyik DbContextnek ugyanazt a kapcsolatot kell használnia
- OnConfiguring metódusban…

```
private DbConnection _connection = null;

0 references
public BlogContext(DbConnection connection)
{
    _connection = connection;
}


0 references
protected override void OnConfiguring(DbContextOpt
{
    if (_connection == null)
        optionsBuilder.UseSqlServer(@"Server=(loca
    else
        optionsBuilder.UseSqlServer(_connection);
}
```

Példa

```
Count before Tx: 3
Count before SaveChanges: 3
Count during Tx in DbContext1 instance: 5
Count during Tx in DbContext2 instance: 5
Count after Tx DbContext2 instance: 5
```

```
using (var db = new BlogContext())
using (var db2 = new BlogContext(db.Database.GetDbConnection()))
using (var transaction = db.Database.BeginTransaction())
    try
    {
        Console.WriteLine($"Count before Tx: {db.Blogs.Count()}");

        db2.Database.UseTransaction(transaction.GetDbTransaction());

        db.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
        db2.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/visualstudio" });
        Console.WriteLine($"Count before SaveChanges: {db.Blogs.Count()}");
        db.SaveChanges();
        db2.SaveChanges();
        Console.WriteLine($"Count during Tx in DbContext1 instance: {db.Blogs.Coun
        Console.WriteLine($"Count during Tx in DbContext2 instance: {db2.Blogs.Cou
        transaction.Commit();
        Console.WriteLine($"Count after Tx DbContext2 instance: {db2.Blogs.Count()
    }
```

Adatréteg leválasztása

- Linq to EF, nHibernate, … : adatelérési réteg
- Leválasztás előnyei, indokai például:
  - Adatréteg lecserélése
  - Unit teszt miatt, mockoláshoz
  - Adatbázis refaktorálás
  - Másik technológiára kell áttérni, mert az jobb SQL-t generál (például Oracle-höz)
  - Át kell térni másik adatbázisra amit a jelenlegi provider nem vagy rosszul támogat
  - …

Szükség van-e a lecserélésre?

- YAGNI: You aren't gonna need it
  - "do the simplest thing that could possibly work"
- KISS: Keep it simple and stupid

- **Agile**: ne a jövőbeli, ***lehetséges*** követelmények alapján tervezd és implementáld a rendszert!
  - Az a Megrendelő költsége amikor kéri…
- **Waterfall**: nagyon gondolt át, hogy az alkalmazás életciklusa alatt erre szükség lesz-e ilyen rugalmasságra
  - Ha igen, áthárítható-e a költségek?

Alternatívák

- Mi a pontos ok, amiért szükség lehet erre?

- Unit tesztelés?
  - Keress in-memory adatbázist, SQLite, …
- Adatbázis csere
  - Keress fizetős komponenseket, hátha van provider
- **Érdemes-e elrejteni az adatréteget az üzleti logikától?**

Adat(elérési) réteg leválasztása

- Leválasztás: tipikusan interfész kialakítása
- A interfészen explicit jelennek meg az alsó réteg szolgáltatásai, funkciói

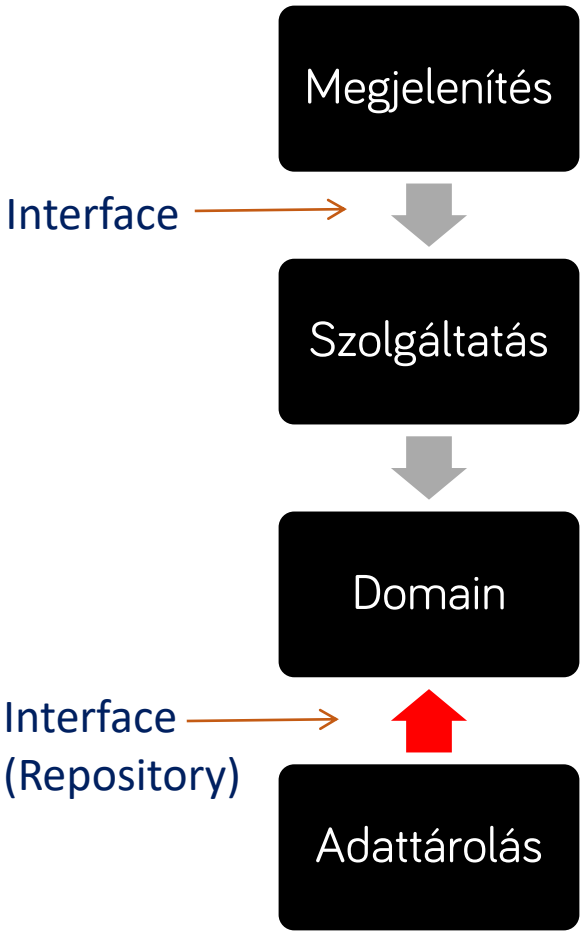
- Széles körben használt, gyakran javasolt, de erősen vitatott megoldás:

Repository pattern

Repository Pattern

- Absztrakció az adatréteg fölött
- Specifikus hely, ahol az adatok elérésének módját módosítani kell
- Az adattáblák helye
- Könnyen cserélhető másik megvalósításra
- Elrejti a részleteket
- Több alternatív megvalósítás

Helye a többrétegű alkalmazásokban



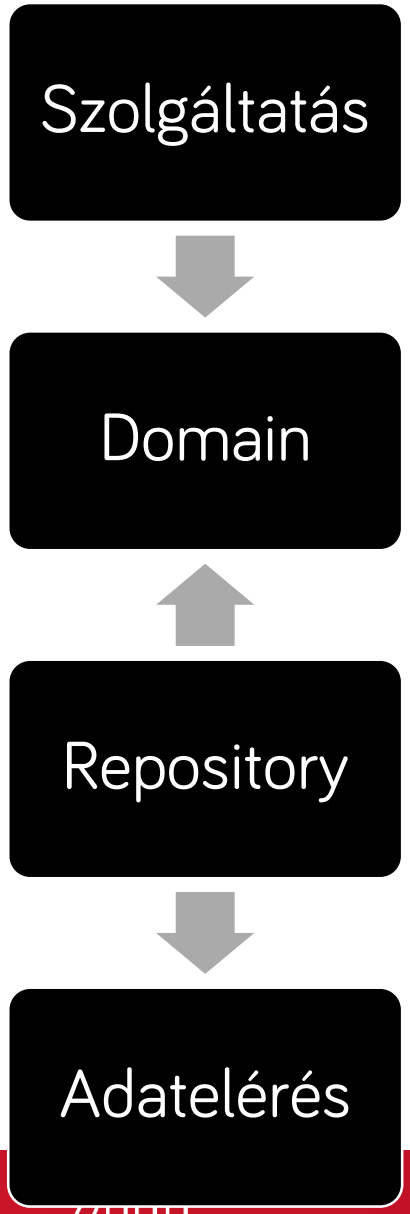
ASP.NET MVC, WPF, XAML, …

WCF, WebAPI vagy POCO-k

DDD POCO-val

ADO.NET, nHibernate, EF POCOs, Micro ORM

Repository példa




```
public User GetUserById(int userId)
```

```
public class User
{
    public int UserId;
    public string FirstName;
    public string LastName;
}
```

```
public class UserRepository : IUserRepository
{
    public User GetUserById(int userId)
    {
        //DB Call here…
    }
}
```

Adatelérés absztrahálása

Repository Pattern

- Adatrétegtől független üzleti logika
- A domain logika és az adatelérés szétválasztása
- Adatelérés/adatbázis lecserélése
- Több adatforrás egységes kezelése
- Adattárolási paradigma cseréje (pl nosql, lucene stb)
- Unit tesztek támogatása
- Fejlesztés jobb párhuzamosítása
- Duplikált lekérdezések csökkentése
- Jobb control a lekérdezések felett
- Objektum orientált API
- Egységes szabályok
- Cache-elés

- Még egy újabb réteg – több munka
- Lecsökkenti az ORM technológiák erejét
- Kockázat: adattárolási API mégis beszivárog az üzleti logikába

Az ORM és a repository

- Mások a célok
- Repository
  - Tárolással kapcsolatos összes funkció absztrakciója és egységbe zárása
  - Architekturális minta
- ORM
  - Támogatott relációs adatbázis elérésének absztrakciója
  - Repository mögött használva „implementációs részlet”

Repository funkciók a gyakorlatban

- Entitások hozzáadása és elvétele
- Az interfész táblákat, gyűjteményeket használ
- A tranzakció kezelés/lezárás nem itt van
- **Lekérdezések a kívánt kritériumoknak megfelelően**

Lekérdező metódusok

Customer[] WithSurname(string surname)

…

Lekérdezés általánosabban

Customer[] Find(ICustomerSpecification spec)

bool ICustomerSpecification.IsSatisfiedBy(Customer c)

- A domain modell része
- Komponálható (összetett kifejezések)
- Memória alapon és SQL-t generálva is működnie kell!

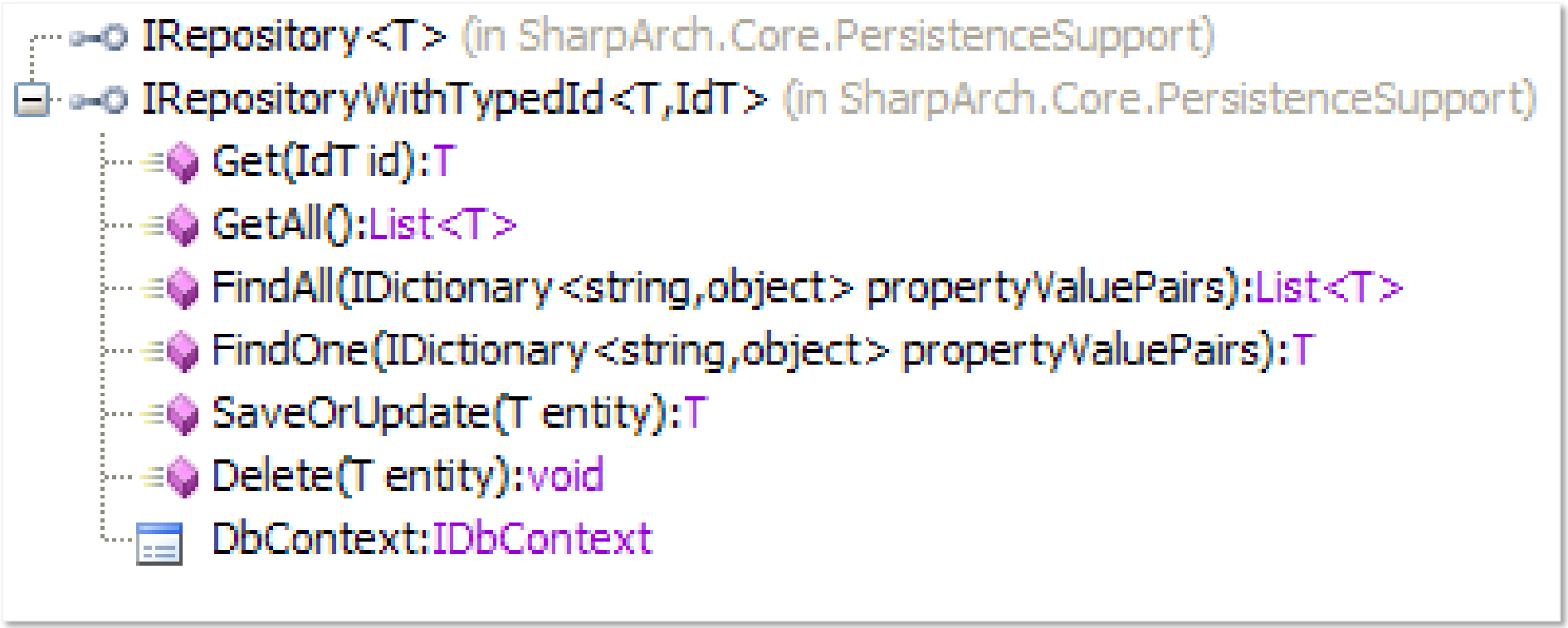
Általános Repository minta

IRepository<T>

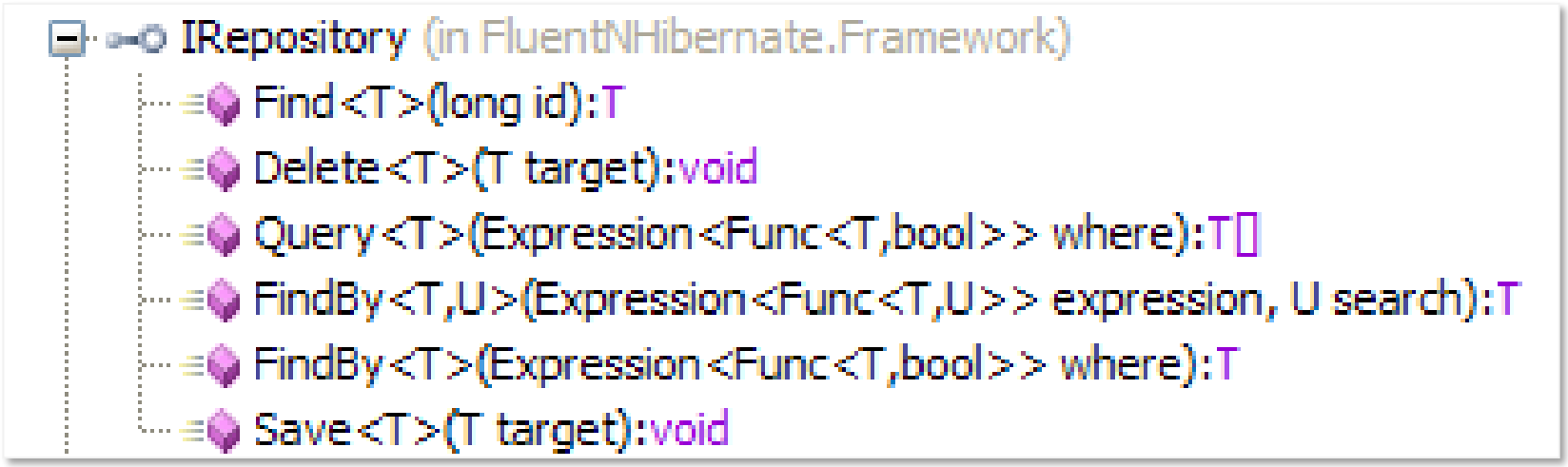
T[] Find<T>(ISpecification<T> specification)

Rhino Commons

Sharp Architecture



Fluent NHibernate



Használható az IQueryable?

IQueryable<T> GetAll<T>() metódussal a BLL-ben:

```
var customers =
  repository.GetAll().ThatMatch(criteria)
  .AsPagedList(pageNumber, pageSize);
```

- Az SQL a repositoryn kívül fut le
- A lekérdezési logika az alkalmazás nem-domain területeire is beszivároghat
- „Leaky abstraction”

A tranzakciókról

- Nem a repository feladata
- Unit of Work minta (NHibernate session, L2EF DataContext)
- A tranzakció határok az üzleti logika felelőssége

A repository minta kritikája

- „**Felesleges**”: nincs szükség újabb absztrakciós rétegre
- **Bonyolítja** a kódot, hiszen „minden” lekérdezést explicit meg kell írni, nem használható a linq
- Ha kiengedjünk az IQueryable-t, akkor elveszítjük a minta előnyeit, mert lekérdezést végül a BLL állítja elő
- „Visszatérés” a tárolt eljárások világába
- **Tesztelés**: nem könnyebb mockolni

A repository minta előnye

- **Domain-driven**: domain specifikus nézetet ad az adatbázisról – szemben a linq-ben megfogalmazott lekérdezésekkel amik általánosak
- **Elrejtés/aggregáció**: elrejti az összefüggő entitások (/táblák) komplexitását és tisztább nézetet ad

EF és a Repository

- Finomítsuk a repository mintát
  - Nem csak ID alapú lekérdezés…
  - Lehessen tetszőleges feltételeket megadni
  - Csoportosítani, joinolni, előkérdezni, …
  - Tegyük lehetővé a tranzakciókat az interfészen
- De hisz ez pont az EF!
  - Nagyon általános lekérdező interfész
  - Az implementáció az adatbázis providerben van
  - A DbContext pont egy UoW megvalósítás!
- Az EF Core célja, hogy javítsa az EF …6 hibáit
  - Például legyen memória adatbázis támogatás stb.