Упражнения: Да напишем ORM

Това **упражнение** предоставя инструкции стъпка-по-стъпка как да си направим наш **“ORM Framework”** на C#, както и примерно приложение, което използва рамката. Целта е да се постигне донякъде сходна функционалност с Entity Framework Core. **Ще получите частично-имплементиран скелет като C# проект**.

Проектна спецификация

Рамката трябва да поддържа следните **функционалности**:

- Свързване към БД чрез подаден низ за връзка
- **Откриване** на класове от данни **по време на изпълнението**
- **Извличане на данни** чрез **генериран от рамката SQL**
- **CRUD** операции (създаване, променяне, изтриване на данни) чрез **генериран от рамката SQL**

Преглед на рамката

Рамката се състои от следните **класове**:

- **`DbSet<T>`** – **Шаблонна колекция**, която съдържа самите **данни** в себе си. Класът **`DbContext`** съдържа няколко **`DbSet-a`**, като всеки от тях съответства на таблица от БД.
- **`DbContext`** – **Контекст на БД –** клас, който отговаря за **извличане на данните от БД** и **съпоставането на връзките** между тях (чрез т.нар. навигационни свойства).
- **`DatabaseConnection`** – Отговаря за **установяване на връзка с БД** и **изпращане на SQL заявки**. Ползва се от **`DbContext`**.
- **`ConnectionManager`** – Проста обвивка на **`DatabaseConnection`**, която ни позволява да ползваме **`using`** блок **за отваряне и затваряне на връзките** към БД
- **`ChangeTracker`** – Отговаря за проследяването на **доабвените, модифицираните** и **изтритите** данни от **`DbSets`**. Всеки **`DbSet`** има по един. Ползва се от **`DbContext`**, за да **съхрани промените** в БД.
- **`ReflectionHelper`** – Помощен клас, който съдържа някои методи свързани с рефлектирането на класовете.

Сега след като вече имате основно разбиране за това кой клас какво трябва да прави, нека да ги **имплементираме**.

Време е да **отворите** предоставения **скелете и да пишете код**.

1. Имплементация на ChangeTracker класа

В този клас ще имаме нужда от три **списъка**. Първият ще съдържа всички данни. Вторият ще пази информация за **добавените** записи, а третият – за премахнатите. Добавете шаблонно ограничение, за да ограничните шаблонните параметри, така щото да се приемат само такива типове, които имат конструктор без параметри.

```
internal class ChangeTracker<T>
    where T: class, new()
{
    private readonly List<T> allEntities;

    private readonly List<T> added;

    private readonly List<T> removed;
```

Конструкторът на **`ChangeTracker`** ще приема **колекция от данни** като параметър. В тялото му, ще трябва да инициализираме списъците за **добавени** и **премахнати** списъци. А **`allEntities`** полето ще съдържа **копия** на всички данни от родителския **`DbSet`**. Трябва да копираме данните, за да можем да разберем кои от тях са **променени,** когато дойде време да ги **запазим** в БД. За тази цел, извикваме **`CloneEntities()`** с **колекция от данни** като параметри.

```
public ChangeTracker(IEnumerable<T> entities)
{
    this.added = new List<T>();
    this.removed = new List<T>();

    this.allEntities = CloneEntities(entities);
}
```

Следващата стъпка е имплементацията на **`CloneEntities`** метода. Този метод ще върне **`List<T>`** с **копираните** данни. Ще ни е нужна още една променлива от тип **`PropertyInfo[]`**, за да запазим свойствата, които трябва да копираме. Интересуваме се само от свойства, които са част от БД, поради тази причина извличаме само свойствата с **валидни SQL типове**.

```
private static List<T> CloneEntities(IEnumerable<T> entities)
{
    var clonedEntities = new List<T>();

    var propertiesToClone = typeof(T).GetProperties()
        .Where(pi => DbContext.AllowedSqlTypes.Contains(pi.PropertyType))
        .ToArray();
```

Обхождаме всички **действителни** данни, създаваме нов празен запис данни от същия тип и **задаваме** всичките му свойства, подлежащи на копираме към стойностите от истинските данни. Накрая, добавяме **`clonedEntity`** към **`List<T>`**. След като сме готови с копирането на данните, връщаме тези данни.

```
    foreach (var entity in entities)
    {
        var clonedEntity = Activator.CreateInstance<T>();

        foreach (var property in propertiesToClone)
        {
            var value = property.GetValue(entity);
            property.SetValue(clonedEntity, value);
        }

        clonedEntities.Add(clonedEntity);
    }

    return clonedEntities;
}
```

След това, трябва да направим всички полета от тип **`IReadOnlyCollection<T>`**, защото не желаем някой да модифицира нашите списъци.

```
public IReadOnlyCollection<T> AllEntities => this.allEntities.AsReadOnly();

public IReadOnlyCollection<T> Added => this.added.AsReadOnly();

public IReadOnlyCollection<T> Removed => this.removed.AsReadOnly();
```

Имаме нужда от **`Add()`** и **`Remove()`** методи, които приемат параметър елемент **T**. Може да реализирате тези методи и сами.

```
public void Add(T item) => this.added.Add(item);

public void Remove(T item) => this.removed.Add(item);
```

Следващият метод е **`GetModifiedEntities()`**, който приема **`DbSet<T>`** променлива като параметър. Методът връща колекция от модифицирани данни. В този метод вземаме **първичните ключове** за текущият обект **T**.

```
public IEnumerable<T> GetModifiedEntities(DbSet<T> dbSet)
{
    var modifiedEntities = new List<T>();

    var primaryKeys = typeof(T).GetProperties()
        .Where(pi => pi.HasAttribute<KeyAttribute>())
        .ToArray();
```

След това, обхождаме **`IReadOnlyCollection allEntities`** и използваме **`GetPrimaryKeyValues()`** метода (ще го имплементираме след малко), който приема **`primaryKeys`** променливата като параметър и **текущия** елемент. Получаме данните от **`dbSet`**, които имат същия **`primaryKeyValues`** като **proxyEntity**.

```
foreach (var proxyEntity in this.AllEntities)
{
    var primaryKeyValues = GetPrimaryKeyValues(primaryKeys, proxyEntity).ToArray();

    var entity = dbSet.Entities
        .Single(e => GetPrimaryKeyValues(primaryKeys, e).SequenceEqual(primaryKeyValues));
```

Можем да проверим дали оригиналния обект е бил модифициран, чрез методът **`IsModified()`** (имплементира се по-късно). Ако засечем модификация, трябва да добавим действителните данни към **`modifiedEntities`**.

```
        var isModified = IsModified(proxyEntity, entity);
        if (isModified)
        {
            modifiedEntities.Add(entity);
        }
    }

    return modifiedEntities;
}
```

Следващият метод за имплементиране е **`IsModified()`**, който приема **оригинални данни** и **proxyEntity** като **параметри**. Те са гарантирано от един и същи тип, защото са от **същия шаблонен тип**.

Първо, ще извлечем всички свойства, които са валидни SQL типове и ще игнорираме останалите. Ще използваме тази променлива да проверим за променени данни. Това може да се случи чрез друга променлива от тип **`PropertyInfo[]`** и извикване на метод **`Equals`** за сравнение на стойностите на свойствата на **`originalEntity`** и **`proxyEntity`**. Накрая, проверяваме дали има **модифицирани** данни и връщаме резултата.

```
private static bool IsModified(T entity, T proxyEntity)
{
    var monitoredProperties = typeof(T).GetProperties()
        .Where(pi => DbContext.AllowedSqlTypes.Contains(pi.PropertyType));

    var modifiedProperties = monitoredProperties
        .Where(pi => !Equals(pi.GetValue(entity), pi.GetValue(proxyEntity)))
        .ToArray();

    var isModified = modifiedProperties.Any();

    return isModified;
}
```

Последният метод за този клас е статичен и ще връща **`IEnumerable`** колекция от **обекти**. Използвахме този метод, за да получим **стойностите на нашите първични ключове**. Методът ще приема **`IEnumerable<PropertyInfo>`** като параметър, който съдържа свойствата на първичния ключ и данните към които принадлежи първичния ключ. Този метод прави само едно нещо – извлича стойността на всеки **първичен ключ.**

```
private static IEnumerable<object> GetPrimaryKeyValues(IEnumerable<PropertyInfo> primaryKeys, T entity)
{
    return primaryKeys.Select(pk => pk.GetValue(entity));
}
```

2. Имплементиране на `DbSet` класа

Създайте шаблонен **`DbSet<TEntity>`** клас, който имплементира **`ICollection<TEntity>`**. Това трябва да изглежда така:

```
public class DbSet<TEntity> : ICollection<TEntity>
    where TEntity : class, new()
{
```

Нашият **`DbSet<T>`** клас представя колекцията от всички данни в контекста или всички данни, които могат да се извличат от БД от даден тип. Аргументът трябва да е от референтен тип, в т.ч. клас, интерфейс, делегат или масив и трябва да има публичен конструктор без параметри. В този клас, ще трябва да дефинираме две вътрешни /internal/ свойства с get iset. Първото е **`ChangeTracker<TEntity>`,** който предоставя достъп до елементи на контекста, които са свързани с проследяването на промените на данните. Вторият е **`IList<TEntity>`**, в който ще съхраняваме данните.

Кодът трябва да изглежда по подобен начин:

```
internal ChangeTracker<TEntity> ChangeTracker { get; set; }

internal IList<TEntity> Entities { get; set; }
```

Нашият **`DbSet`** конструктор трябва да е **internal** и трябва да приема параметри от тип **`IEnumerable<TEntity>`**, които ще бъдат самите данни. **Конструкторът** задава **свойствата на данните** и създава **`ChangeTracker`**, за да може да следим промените в данните.

```
internal DbSet(IEnumerable<TEntity> entities)
{
    this.Entities = entities.ToList();

    this.ChangeTracker = new ChangeTracker<TEntity>(entities);
}
```

`DbSet` класа действа като **`ICollection<T>,`** затова трябва да имплементираме всички методи характерни за **`ICollection<T>`**.

Първо, трябва да имплементираме метод за **добавяне на данни** в БД. Ако стойността на параметъра е **null,** хвърляме изключение **`ArgumentNullException`** със съобщението "**`Item cannot be null`**". След тази проверка, **добавяме** елемента в **Entities**, а също така и в **ChangeTracker**.

```
public void Add(TEntity item)
{
    if (item == null)
    {
        throw new ArgumentNullException(nameof(item), "Item cannot be null!");
    }

    this.Entities.Add(item);

    this.ChangeTracker.Add(item);
}
```

`Clear` методът премахва всички данни, използвайки **`Remove`** метода. Използваме го както следва, така че да може и **регистърът на промени** да има информация, че данните са премахнати.

```
public void Clear()
{
    while (this.Entities.Any())
    {
        var entity = this.Entities.First();
        this.Remove(entity);
    }
}
```

`Contains` методът проверява дали в списъка от данни се съдържа конкретна данна.

```
public bool Contains(TEntity item) => this.Entities.Contains(item);
```

`CopyTo` методът копира нашите **данни** в масив от тип **T**, започвайки от определен **индекс в масива**. Няма да позлваме това където и да е, но е част от **`ICollection<T>`** интефейса, затова трябва да го имплементираме.

```
public void CopyTo(TEntity[] array, int arrayIndex) => this.Entities.CopyTo(array, arrayIndex);
```

`Count` свойството дава информация за броя на **данните**.

```
public int Count => this.Entities.Count;
```

`IsReadOnly` свойството проверява, ако нашата колекция от данни е от тип **readonly**. Също, това е нужно и заради интерфейса **`ICollection<T>`**.

```
public bool IsReadOnly => this.Entities.IsReadOnly;
```

Последният метод, който трябва да имплементираме от **`ICollection<T>`** интерфейса е **Remove** метода. Трябва да проверяваме за два проблема. Първо, **T** елементът не трябва да е **null**. Ако е, то хвърляме изключение **`ArgumentNullException`** със съобщение "**`Item cannot be null`**". След това трябва да създадем **променлива**, в която ще проверяваме дали сме премахнали успешно елемента. Ако сме, ще го премахнем и от регистъра на промени.

```
public bool Remove(TEntity item)
{
    if (item == null)
    {
        throw new ArgumentNullException(nameof(item), "item cannot be null!");
    }

    var removedSuccessfully = this.Entities.Remove(item);

    if (removedSuccessfully)
    {
        this.ChangeTracker.Remove(item);
    }

    return removedSuccessfully;
}
```

DbSet класа има още два метода за имплементираме. Тези методи са **IEnumerator<T> GetEnumerator()** и **IEnumerable.GetEnumerator().** Имаме нужда от тях, за да **обхождаме** колекцията от данни.

```
public IEnumerator<TEntity> GetEnumerator()
{
    return this.Entities.GetEnumerator();
}

IEnumerator IEnumerable.GetEnumerator()
{
    return this.GetEnumerator();
}
```

Последното нещо, което трябва да направим за този клас е да направим метод, който ще премахва множество от елементи. За тази цел трябва да обходим **entities** параметъра и да **премахнем всеки елемент** в него.

```
public void RemoveRange(IEnumerable<TEntity> entities)
{
    foreach (var entity in entities.ToArray())
    {
        this.Remove(entity);
    }
}
```

3. Имплементиране на DbContext

Създайте **абстрактен DbContext** клас. За начало имаме нужда от **две полета.** Първото е **DatabaseConnection**. Второто е от тип **IEnumerable<PropertyInfo>**, където ще пазим свойствата на **DbSet<T>**, когато ги открием. Помнете, че понеже пишем **рамка**, която други хора биха ползвали, **не знаем** какви данни и **DbSet-ове** биха направили те по **време на компилиране.** Поради тази причина, ще трябва да разберем това по **време на изпълнението**.

Когато сте готови, то трябва да имате нещо такова:

```
public abstract class DbContext
{
    private readonly DatabaseConnection connection;

    private readonly Dictionary<Type, PropertyInfo> dbSetProperties;
```

Сега нека да създадем **поле**, където ще съхраняваме **позволените SQL типове**. Помислете или проверете какви типове данни може да съхраните в **SQL Server** и ги избройте в това поле. Накрая трябва да се получи това:

```
internal static readonly Type[] AllowedSqlTypes =
{
    typeof(string),
    typeof(int),
    typeof(uint),
    typeof(long),
    typeof(ulong),
    typeof(decimal),
    typeof(bool),
    typeof(DateTime)
};
```

Ще използваме тези по-късно, за да определяме свойствата на данните, които ще бъдат включени в манипулацията на базата данни.

Нашият **`DbContext`** конструктор трябва да е с достъп **protected** и да приема като параметър **`connectionString`**. В тялото на конструктора трябва да създадем инстанция на **`DatabaseConnection`** класа с **`connectionString`**. Трябва да инициализираме **`dbSetProperties`**, чрез метод **`DiscoverDbSets()`**, който ще имплементираме по-късно. След това трябва да отворим връзка към БД чрез метод **`InitializeDbSets().`** Извън using конструкцията, трябва да извикаме **`MapAllRelations`** метода (ще го имплемнтираме по-късно). Вашият конструктор трябва да изглежда така:

```
protected DbContext(string connectionString)
{
    this.connection = new DatabaseConnection(connectionString);

    this.dbSetProperties = this.DiscoverDbSets();

    using (new ConnectionManager(connection))
    {
        this.InitializeDbSets();
    }

    this.MapAllRelations();
}
```

Сега ще създадем единственият **public** метод – **`SaveChanges()`**. Всичко, което прави този метод е да **обхожда** всеки **`DbSet`** и да **изпълнява `Persist<TEntity>()` метода** за всеки **DbSet**. Понеже не знам какви са **шаблонните типове** на **DbSet**-овете, трябва да стартираме метода динамично, чрез отражение и да му даваме параметър за типа. След като направим метода за запазване, ще обградим неговото извикване в try/catch и ще му предоставим няколко различни вида изключения, които ще може да хване.

Първо, трябва да декларираме масив от **реални DbSet-ове като колекции**:

```
public void SaveChanges()
{
    var dbSets = this.dbSetProperties
        .Select(pi => pi.Value.GetValue(this))
        .ToArray();
```

Преди да направим каквото и да е запазване, трябва да се уверим, че всички данни в контекста са **валидни**. Ако има **невалидни** данни, то хвърляме **`InvalidOperationException`** със съобщение **`"{invalidEntities.Length} Invalid Entities found in {dbSet.Name}!"`**. Кодът трябва да изглежда по подобен начин:

```
    foreach (IEnumerable<object> dbSet in dbSets)
    {
        var invalidEntities = dbSet
            .Where(entity => !IsObjectValid(entity))
            .ToArray();

        if (invalidEntities.Any())
        {
            throw new InvalidOperationException(
                $"{invalidEntities.Length} Invalid Entities found in {dbSet.GetType().Name}!");
        }
    }
```

След това, трябва да ползваме **`using` конструкция**, която ще **отвори връзка** към нашата **БД.** Ние обграждаме в **using** всеки блок код, който **достъпва БД**, за да не се налага да затваряме връзката **ръчно**. Отваряне и затваряне на неща **ръчно**, без значение дали е връзка към БД, поток от данни или какъвто и да е неуправляван ресурс е добър начин да **забравим** да напишем **open/close** команди и да настъпим мотиката на мистериозните бъгове, които да ни загубят ценно време от живота. Така че **просто не го правете ръчно**.

В тази using конструкция, трябва да създадем **още една `using` конструкция** – този път за **стартиране на транзакция в БД.** По този начин, ако нещо се обърка, **данните няма да бъдат засегнати.** Кодът е както следва:

```
    using (new ConnectionManager(connection))
    {
        using (var transaction = this.connection.StartTransaction())
        {
```

Сега трябва да разберем типа на всеки **`DbSet`**. Имаме нужда от друга променлива, която ще пази **Persist** метода (ще бъде имплементирам по-късно) и ще прави шаблонна версия на този метод, чрез типа на DbSet. Кодът е както следва:

```
            foreach (IEnumerable dbSet in dbSets)
            {
                var dbSetType = dbSet.GetType().GetGenericArguments().First();

                var persistMethod = typeof(DbContext)
                    .GetMethod("Persist", BindingFlags.Instance | BindingFlags.NonPublic)
                    .MakeGenericMethod(dbSetType);
```

Накрая, трябва да извикаме този метод в **`try`** блок с няколко **`catch-a`**. В **`try`** блокът, ще извикаме **Persist** методът за **`dbSet`**. Кодът е както следва:

```
                try
                {
                    persistMethod.Invoke(this, new object[] {dbSet});
                }
                catch (TargetInvocationException tie)
                {
                    throw tie.InnerException;
                }
                catch (InvalidOperationException)
                {
                    transaction.Rollback();
                    throw;
                }
                catch (SqlException)
                {
                    transaction.Rollback();
                    throw;
                }
            }

            transaction.Commit();
        }
    }
}
```

Първият catch блок ще обработва **`TargetInvocationException`**. Ако извиканият метод **хвърли изключение**, това е изключението, което ще трябва да **хванем**. Съответно, този блок **хвърля** вътрешното изключение, защото това е реалното изключение,което е възникнало в извикването на метода, като с това ще се занимават **втори и трети `catch`** блокове.

Втори и трети **`catch`** блокове ще обработват съответно **`InvalidOperationException`** и **`SqlException`**. И в двата случая, трябва да изпълним **`rollback`** на транзакцията. Ако не бъдат хвърлени изключения, ще изпълним **commit** на транзакцията и ще **запазим нашите промени** в БД.

Сега е време да имплементираме **`Persist<TEntity>` метода**. Той приема **`DbSet`** като **шаблонен тип** и **транзакция**.

Първо, трябва да създадем променлива, където да запазим **името на текущата таблица** (като низ) използвайки **`GetTableName()`** метода (ще бъде имплементиран по-късно). После ще ползваме масив, в който ще пазим колоните, извиквайки **`FetchColumnNames()`** метода (също ще бъде имплементиран по-късно). Тогава проверяваме **регистъра на промените за съответния dbSet** за **каквито и да е** добавени данни, съответно ако има такива, ползваме **`InsertEntities()`** метода, който вече имаме в **`DbConnection`** класа.

```
private void Persist<TEntity>(DbSet<TEntity> dbSet)
    where TEntity : class, new()
{
    var tableName = GetTableName(typeof(TEntity));

    var columns = this.connection.FetchColumnNames(tableName).ToArray();

    if (dbSet.ChangeTracker.Added.Any())
    {
        this.connection.InsertEntities(dbSet.ChangeTracker.Added, tableName, columns);
    }
```

Сега ще имаме нужда от **променените** данни. Можем да ги извлечем чрез **`GetModifiedEntities()`**, който е част от **`ChangeTracker`** класа. Ако има модифицирани данни, то ги **обновяваме**, използвайки **`UpdateEntities()`**, който приема **данните**, **името на таблицата** и **колоните на таблицата** като параметри.

```
var modifiedEntities = dbSet.ChangeTracker.GetModifiedEntities(dbSet).ToArray();
if (modifiedEntities.Any())
{
    this.connection.UpdateEntities(modifiedEntities, tableName, columns);
}
```

Накрая, проверяваме дали има премахнати записи чрез колекцията **`Removed`** в регистъра на промените. Ако има такива, то ги **изтриваме** и от базата данни.

```
    if (dbSet.ChangeTracker.Removed.Any())
    {
        this.connection.DeleteEntities(dbSet.ChangeTracker.Removed, tableName, columns);
    }
}
```

Следващата стъпка е да създадем метод за инициализиране на dbSet-ове наречен **`InitializeDbSets()`**. За всеки DbSet, ще извикваме **`PopulateDbSet(dbSetProperty)`** метода **динамично**, защото ще предоставяме **параметър от шаблонен тип по време на изпълнението**, понеже не знаем какви ще са **DbSet**-овете.

```
private void InitializeDbSets()
{
    foreach (var dbSet in this.dbSetProperties)
    {
        var dbSetType = dbSet.Key;
        var dbSetProperty = dbSet.Value;

        var populateDbSetGeneric = typeof(DbContext)
            .GetMethod("PopulateDbSet", BindingFlags.Instance | BindingFlags.NonPublic)
            .MakeGenericMethod(dbSetType);

        populateDbSetGeneric.Invoke(this, new object[] {dbSetProperty});
    }
}
```

Следващият метод за имплементиране е **`PopulateDbSet<TEntity>()`**. Ще извличаме данните от БД, чрез **`LoadTableEntities<TEntity>()`** метода. След това, ще създаваме нова **`DbSet<TEntity>`** инстанция, като подаваме данните към конструктора.

Накрая, трябва да заменим реалнитото свойство на **`DbSet`** в текущата **инстанция на `DbContext`** с тази, която създадохме. Понеже **DbSet**-овете нямат setter, трябва да заменим полето, чрез **`ReflectionHelper.ReplaceBackingField()`** метода. Това работи, защото всяко **авто-свойство** има **private**, **автоматично генерирано поле**.

```
private void PopulateDbSet<TEntity>(PropertyInfo dbSet)
    where TEntity : class, new()
{
    var entities = LoadTableEntities<TEntity>();

    var dbSetInstance = new DbSet<TEntity>(entities);
    ReflectionHelper.ReplaceBackingField(this, dbSet.Name, dbSetInstance);
}
```

Сега трябва да имплементираме нов метод **MapAllRelations()**. Всичко, което прави този метод е да извиква **MapRelations()** динамично за всяко свойстов на **DbSet**. Този метод изглежда много подобно на **InitializeDbSets()** метода.

```
private void MapAllRelations()
{
    foreach (var dbSetProperty in this.dbSetProperties)
    {
        var dbSetType = dbSetProperty.Key;

        var mapRelationsGeneric = typeof(DbContext)
            .GetMethod("MapRelations", BindingFlags.Instance | BindingFlags.NonPublic)
            .MakeGenericMethod(dbSetType);

        var dbSet = dbSetProperty.Value.GetValue(this);

        mapRelationsGeneric.Invoke(this, new[] {dbSet});
    }
}
```

Сега е време да имплемнтираме **MapRelations<TEntity>()** метода, за който говорихме по-рано. Този метод приема **DbSet<TEntity>** променлива като **единствен** параметър.

Този метод **съпоставя** всички релации в DbSet-а. Има **два** типа релации: Свойства базирани на **външни ключове**, които съпоставят **много-към-един** отношения и **колекции**, които съпостват **един-към-много** и **много-към-много** отношения. Първо, съпоставяме **навигационните свойства**, а след това съпоставяме **колекциите**. С цел да открием какви колекции има **TEntity**, трябва да **отразим** класа и да открием всички **свойства**, които са от тип **ICollection<>**.

```
private void MapRelations<TEntity>(DbSet<TEntity> dbSet)
    where TEntity : class, new()
{
    var entityType = typeof(TEntity);

    MapNavigationProperties(dbSet);

    var collections = entityType
        .GetProperties()
        .Where(pi =>
            pi.PropertyType.IsGenericType &&
            pi.PropertyType.GetGenericTypeDefinition() == typeof(ICollection<>))
        .ToArray();
```

След като открием колекциите, трябв ада ги обходим и да извикаме **MapCollection** метода динамично за всяка от тях, подобно на предните два метода.

```
    foreach (var collection in collections)
    {
        var collectionType = collection.PropertyType.GenericTypeArguments.First();

        var mapCollectionMethod = typeof(DbContext)
            .GetMethod("MapCollection", BindingFlags.Instance | BindingFlags.NonPublic)
            .MakeGenericMethod(entityType, collectionType);

        mapCollectionMethod.Invoke(this, new object[] {dbSet, collection});
    }
}
```

Сега е ред на **MapCollection<TDbSet, TCollection>()** да бъде имплементиран. Този метод приема **DbSet<TDbSet>** и **PropertyInfo** променливи като параметри. Сега, трябва да вземем първичните и външните ключове. Първичните ключове се намират чрез извличане на всички свойства с **[Key]** атрибут в **collectionType**, а външните ключове – по същи начин но в **entityType.**

```
private void MapCollection<TDbSet, TCollection>(DbSet<TDbSet> dbSet, PropertyInfo collectionProperty)
    where TDbSet : class, new() where TCollection : class, new()
{
    var entityType = typeof(TDbSet);
    var collectionType = typeof(TCollection);

    var primaryKeys = collectionType.GetProperties()
        .Where(pi => pi.HasAttribute<KeyAttribute>())
        .ToArray();

    var primaryKey = primaryKeys.First();
    var foreignKey = entityType.GetProperties()
        .First(pi => pi.HasAttribute<KeyAttribute>());
```

Трябва да проверим дали си имаме работа с много-към-много релация, което е вярно само ако имаме 2 или повече първични ключа. Ако имаме много-към-много релация, можем да извлечем външния ключ, намирайки типа на първото свойство, чието име е равно на името на атрибута на външния ключ и има същия тип свойство като данните.

```
var isManyToMany = primaryKeys.Length >= 2;
if (isManyToMany)
{
    primaryKey = collectionType.GetProperties()
        .First(pi => collectionType
                        .GetProperty(pi.GetCustomAttribute<ForeignKeyAttribute>().Name)
                        .PropertyType == entityType);
}
```

Сега ще вземем DbSet-а на колекцията, който ще филтрираме чрез **where-клауза** и ще извлечем всички данни, чиито външни ключове са равни на първичния ключ на текущата данна.

Накрая, извикваме **ReflectionHelper.ReplaceBackingField()** метода и заменяме **null** колекцията със запълнената колекция

Накрая вашият код трябва да изглежда така:

```
        var navigationDbSet = (DbSet<TCollection>) this.dbSetProperties[collectionType].GetValue(this);

        foreach (var entity in dbSet)
        {
            var primaryKeyValue = foreignKey.GetValue(entity);

            var navigationEntities = navigationDbSet
                .Where(navigationEntity => primaryKey.GetValue(navigationEntity).Equals(primaryKeyValue))
                .ToArray();

            ReflectionHelper.ReplaceBackingField(entity, collectionProperty.Name, navigationEntities);
        }
    }
```

Следващият метод за имплементиране е **`MapNavigationProperties<TEntity>()`**, който приема **DB set** като параметър. Този метод намира **външните ключове на данните** (те могат да са **няколко**) и ги обхожда. За всеки от тези външни ключове, намираме неговите **навигационно свойство** и **тип.** След това, използваме този тип, за да извлечем другата страна на релацията на този **DB set**. Тогава, **за всяка данна** в този **DB set**, намираме **първият запис от данни**, чиито **първичен ключ е равен** на **външния ключ** в **`TEntity`**. Накрая, заменяме навигационното свойство (което текущо е **`null`**) със записа от данни, който сме намерили.

```
private void MapNavigationProperties<TEntity>(DbSet<TEntity> dbSet)
    where TEntity : class, new()
{
    var entityType = typeof(TEntity);

    var foreignKeys = entityType.GetProperties()
        .Where(pi => pi.HasAttribute<ForeignKeyAttribute>())
        .ToArray();

    foreach (var foreignKey in foreignKeys)
    {
        var navigationPropertyName =
            foreignKey.GetCustomAttribute<ForeignKeyAttribute>().Name;
        var navigationProperty = entityType.GetProperty(navigationPropertyName);

        var navigationDbSet = this.dbSetProperties[navigationProperty.PropertyType]
            .GetValue(this);

        var navigationPrimaryKey = navigationProperty.PropertyType.GetProperties()
            .First(pi => pi.HasAttribute<KeyAttribute>());

        foreach (var entity in dbSet)
        {
            var foreignKeyValue = foreignKey.GetValue(entity);

            var navigationPropertyValue = ((IEnumerable<object>) navigationDbSet)
                .First(currentNavigationProperty =>
                    navigationPrimaryKey.GetValue(currentNavigationProperty).Equals(foreignKeyValue));

            navigationProperty.SetValue(entity, navigationPropertyValue);
        }
    }
}
```

Споменахме метод **`IsObjectValid()`**, който приема **`object`** параметър и връща **`bool`**. Понеже **`Validator`** класа е част от **`System.Data.Annotations`**, който е доста стар, трябва да напишем доста

стереотипен код / boilerplate code/, за да го ползваме. Затова вместо да пишем това навсякъде, където трябва да валидираме обект, тази функционалност се поставя в собствен метод. Кодът е както следва:

```
private static bool IsObjectValid(object e)
{
    var validationContext = new ValidationContext(e);
    var validationErrors = new List<ValidationResult>();

    var validationResult =
        Validator.TryValidateObject(e, validationContext, validationErrors, validateAllProperties: true);
    return validationResult;
}
```

Следващият метод за имплементиране е **`LoadTableEntities<TEntity>()`** метода. В него трябва да декларираме няколко променливи. Първата ще съхранява типа на **`TEntity`** и ще бъде нашата **таблица**. Следващият ще бъде за **колоните** и ще бъде **масив от низове**. Там, ще пазим **имената на колоните** за текущата таблица, извиквайки **`GetEntityColumnNames`** (имплементирайте това **накрая**). Третата променлива ще бъде за **името на таблицата** и ще я получаваме чрез **`GetTableName()`** (имплементирайте това **второ по ред**). Последната променлива, която ще връща и метода е **`fetchedRows`** променливата. Можем да получим извлечените редове, извиквайки **`FetchResultSet<TEntity>()`** метода от **`DbConnection`** със съответните параметри.

```
private IEnumerable<TEntity> LoadTableEntities<TEntity>()
    where TEntity : class
{
    var table = typeof(TEntity);

    var columns = GetEntityColumnNames(table);

    var tableName = GetTableName(table);

    var fetchedRows = this.connection.FetchResultSet<TEntity>(tableName, columns).ToArray();

    return fetchedRows;
}
```

Сега нека да имплементираме **`GetTableName()`**, който връща низ и получава **`tableType`** параметър. Можете да се справите и сами! 😊

```
private string GetTableName(Type tableType)
{
    var tableName = ((TableAttribute) [illegible].Name;

    if (tableName == null)
    {
        tableName = this.dbSetProperties[tableType].Name;
    }

    return tableName;
}
```

Почти сме готови с този клас, но трябва да имплементираме **`DiscoverDbSets()`** метод. Ще ползваме този метод в нашия конструктор, за да попълним **`dbSetProperties`** полето, което е Dictionary, където ключът е Type, а **`PropertyInfo`** е стойност. Кодът е както следва:

```
private Dictionary<Type, PropertyInfo> DiscoverDbSets()
{
    var dbSets = this.GetType().GetProperties()
        .Where(pi => pi.PropertyType.GetGenericTypeDefinition() == typeof(DbSet<>))
        .ToDictionary(pi => pi.PropertyType.GetGenericArguments().First(), pi => pi);

    return dbSets;
}
```

Последният метод е **`GetEntityColumnNames()`**, който връща **масив от низове** с **имената на колоните** и приема **table type** като параметър. Накрая, трябва да извлечем **свойствата на таблицата**, които са от **валиден SQL тип** и се съдържат в **имената на колоните**. След това, извличаме имената на свойствата и ги **връщаме**.

```
private string[] GetEntityColumnNames(Type table)
{
    var tableName = this.GetTableName(table);
    var dbColumns =
        this.connection.FetchColumnNames(tableName);

    var columns = table.GetProperties()
        .Where(pi => dbColumns.Contains(pi.Name) &&
                     !pi.HasAttribute<NotMappedAttribute>() &&
                     AllowedSqlTypes.Contains(pi.PropertyType))
        .Select(pi => pi.Name)
        .ToArray();

    return columns;
}
```

С това финализирахме нашата рамка!

Министерство на образованието и науката (МОН)

- Настоящият курс (презентации, примери, задачи, упражнения и др.) е разработен за нуждите на Национална програма "**Обучение за ИТ кариера**" на МОН за подготовка по професия "Приложен програмист".

- Курсът е базиран на учебно съдържание и методика, предоставени от **фондация "Софтуерен университет"** и се разпространява под **свободен лиценз CC-BY-NC-SA** (Creative Commons Attribution-Non-Commercial-Share-Alike 4.0 International).