Lab: MiniORM Core

This **tutorial** provides step-by-step guidelines to build a **"ORM Framework"** in C#, as well as a sample app, which utilizes the framework. The app is designed to resemble the functionality of Entity Framework Core to an extent. **You will be provided with a partially-implemented framework C# project**.

Project Specification

The framework should support the following **functionality**:

- Connecting to a database via provided connection string
- **Discovering** entity classes at **runtime**
- **Retrieving entities** via **framework-generated SQL**
- **CRUD** operations (inserting, modifying, deleting entities) via **framework-generated SQL**

Framework Overview

The framework consists of the following **classes**:

- **`DbSet<T>`** – **Custom generic collection**, which holds the actual **entities** inside it. The **`DbContext`** class has several **`DbSets`**, which correspond to all the tables in the database.
- **`DbContext`** – **Database context** class, responsible for **retrieving entities from the database** and **mapping the relations** between them (through so-called navigation properties).
- **`DatabaseConnection`** – Responsible for **establishing database connections** and **sending SQL queries**. Usually used by the **`DbContext`**.
- **`ConnectionManager`** – Simple **`DatabaseConnection`** wrapper, which allows it to be wrapped in a **`using`** block for **opening and closing connections** to the database
- **`ChangeTracker`** – Responsible for tracking the **added**, **modified** and **deleted** entities from the **`DbSets`**. Every **`DbSet`** has one. Used by the **`DbContext`** to **persist changes** into the database.
- **`ReflectionHelper`** – Utility class, which contains some reflection-related methods.

Now that you have a very basic understanding of what each class should do, let's get to **implementing them**.

It's time to **open** the provided **skeleton** and **start writing code**.

1. Implement the ChangeTracker class

In this class we will need three **lists**. The first one will store all the entities. The second one will keep track of the **added** ones and the third – the removed ones. Add a generic constraint to **limit** the **generic type parameters** to only **reference types**, which have a **parameter-less constructor**.

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

    private readonly List<T> added;

    private readonly List<T> removed;
```

The **ChangeTracker** constructor will accept a **collection of entities** as a parameter. In its body, we have to initialize our **added** and **removed** lists. Then, the **allEntities** field will store **clones** of all the entities of the parent **DbSet**. We need to clone them, so we can find out which ones were **modified** when the time for **persisting them** into the database comes. To do that, we call **CloneEntities()** with the **collection of entities** as parameters.

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

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

The next step is to implement the **CloneEntities** method. This method will return a **List<T>** with the **cloned** entities. We need another variable of type **PropertyInfo[]** to collect the properties, we need to clone. We only care about properties, which are part of the database, so we only get the properties with **valid SQL types**.

```
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();
```

We iterate over each **real** entity, create a new blank entity of the same type and **set** all its cloneable properties to the real entity's property values. Lastly, we add the **clonedEntity** to our **List<T>**. After we're done cloning our entities, we return them to our caller.

```
    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;
}
```

Next, we need to make all the fields of type **`IReadOnlyCollection<T>`** because we don't want someone to modify our lists.

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

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

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

We need **`Add()`** and **`Remove()`** methods which will take as parameter **T** element. You can do this on your own.

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

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

The next method is **`GetModifiedEntities()`**, which takes a **`DbSet<T>`** variable as a parameter. The method returns a collection of modified entities. In this method we have to get the **primary keys** for the current **T** object, but you already know how to do that.

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

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

After that, we go through the **`IReadOnlyCollection`** of **`allEntities`** and use the **`GetPrimaryKeyValues()`** method (implement later), which takes our **`primaryKeys`** variable as a parameter and the **current** proxy entity from all the entities. Then, we get the entity from the **`dbSet`** which has the same **`primaryKeyValues`** as our our **proxy entity**.

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

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

We can check if the original object has been modified, using the **`IsModified()`** (implement later) method. If there is any modification, we have to add the real entity to our **`modifiedEntities`**.

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

    return modifiedEntities;
}
```

The next method to implement is **`IsModified()`**, which takes the **original** and **proxy entities** as **parameters**. They are guaranteed to be of the same type, because they are of the **same generic type**.

First, we'll extract all properties, which are valid SQL types and ignore the rest. We will use this variable to check for any modified entities. This can be done by making another variable of type **`PropertyInfo[]`** and calling method **`Equals`** to compare our **`originalEntity`** and **`proxyEntity`**'s **property values**. Finally, we check if there are **any modified** properties and return the result.

```
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;
}
```

The last method for this class is static and will return an **`IEnumerable`** collection of **`objects`**. We used this method before to get our **primary key values**. The method will take an **`IEnumerable<PropertyInfo>`** as a parameter, which holds the primary key properties and the entity to which the primary keys belong to. This method only does one thing – gets each **primary key property's value**.

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

2. Implement the DbSet class

Create a generic **DbSet<TEntity>** class, which inherits from **ICollection<TEntity>**. It should look like this:

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

Our **DbSet<T>** class represents the collection of all entities in the context, or that can be queried from the database, of a given type. The type argument must be a reference type, including any class, interface, delegate, or array type and must have a public parameter-less constructor. In this class we have to define two internal properties with getters and setters. The first one is a which provides access to features of the context that deal with change tracking of entities. The second one is **IList<TEntity>**, where we are going to collect our entities.

The code should look like this:

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

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

Our **DbSet** constructor must be **internal** and should take parameters of type **IEnumerable<TEntity>** which will be our entities. The **constructor** sets the **entities property** and creates a **ChangeTracker**, so we can track changes in the entities.

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

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

Our **DbSet** class acts like an **ICollection<T>,** so we need to implement all of its methods.

First, we need to implement a method for **adding entities** in the database. If the parameter value is **null,** we throw an **ArgumentNullException** with the message "**Item cannot be null**". After this check, we **add** our item both in the **entities** property, and the **change tracker**.

```
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);
}
```

The **Clear** method removes all entities, by using the **Remove** method. We use it like this, so we can also let the **change tracker** know, that all the entities were removed.

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

The **Contains** method checks if our entities contain a particular entity.

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

The **CopyTo** method copies our **entities** to an array of type **T**, starting at a particular **array index**. We aren't going to use this anywhere, but it's a part of the **ICollection<T>** interface, so we need to implement it.

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

The **Count** property gets the count of our **entities**.

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

The **IsReadOnly** property checks, if our entities collection is of type **readonly**. Again, we need this because of the **ICollection<T>** interface.

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

The last method we have to implement from our **ICollection<T>** interface is the **Remove** method. We need to check for two problems. First, the **T** element must not be null. If it is, we throw an **ArgumentNullException** with a message "**Item cannot be null**". After that we need to create a **variable** where we check if we have **successfully removed** the item. If we have, we **remove it from our change tracker** as well.

```
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;
}
```

Our **DbSet** class has two more methods to implement. These methods are **IEnumerator<T> GetEnumerator()** and **IEnumerable.GetEnumerator().** We need them to **iterate** through our entities collection.

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

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

The last thing we need to do in this class is create a method which will remove a range of entities. To do that we need to iterate through our **`entities`** parameter and **remove each entity** inside of it.

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

3. Implement the DbContext

Create an **abstract `DbContext`** class. For starters, in this class, we need to have **two fields**. The first field is a **`DatabaseConnection`**. The second one is of type **`Dictionary<Type, PropertyInfo>`**, where we're going to store our **`DbSet<T>`** properties, once we discover them. Remember, since we're writing a **framework**, which other people are going to be using, we **don't know** what entities/**`DbSets`** they will define at **compile-time**, so we need to discover that **at runtime**.

When you're done, you should have something like this:

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

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

Now we need to create a **field**, where we store our **allowed SQL type**s. Think about what kinds of data you can store in **SQL Server**, and then list them in your field. When you're done, your code should look something like this:

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

We're going to use these later when we determine what entity properties we're going to involve in our database manipulations.

Our **DbContext** constructor must be protected and should take as parameter **connectionString**. In the body of the constructor we have to create an instance of the **DatabaseConnection** class with the **connectionString**. We should initialize our **dbSetProperties** by using a method called **DiscoverDbSets()** which we will implement later. After that we need a using statement like before, where to open a connection to our database and in this using we should call a method **InitializeDbSets().** Out of the using statement body we have to call **MapAllRelations** method (implement later). Your constructor should look like this:

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

    this.dbSetProperties = this.DiscoverDbSets();

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

    this.MapAllRelations();
}
```

Now we're going to create the only **public** method – **SaveChanges()**. All this method does is **iterate** over each **DbSet** and **execute the Persist<TEntity>() method** for each of those DB sets. Since we don't know what the **generic types** of our DB sets are, we need to dynamically invoke the method, using reflection and provide it with a type parameter. After we make the persist method, we'll wrap its invocation in a try/catch block and provide a few different types of exceptions it's going to catch.

First, we need to declare an array of our **actual DB sets as collections**:

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

Before we do any persisting, we need to ensure each entity in the context is **valid**. If any **invalid entities** exist in our DB set, we **throw** an **InvalidOperationException** with message **"{invalidEntities.Length} Invalid Entities found in {dbSet.Name}!"**. The code should look like this:

```
    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}!");
        }
    }
```

After that, we need a **using block**, which will **open a connection** to our **database.** We wrap each code block, which **accesses** the **database** in a using block, so we don't have to **close** our connection **manually**. Opening and closing

stuff **manually**, whether it be a database connection, or a stream, or any unmanaged resource is a great recipe for **forgetting** to write **open**/**close** statements and encountering mysterious bugs, so **don't ever do it**.

Anyway, in this using block, we need to create **another `using` block** – this time for **starting a database transaction**. That way, if anything goes wrong, **no entities** will be inserted/modified/deleted. The code looks something like this:

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

Now we need to find the entity type of each **`DbSet`**. We need another variable, which will hold the **Persist** method (which we are going to implement later) and make a generic version of that method, using the DB set type. The code looks like this:

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

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

Last, but not least, we need to invoke this method inside a **`try`** block with a couple of **`catch`** blocks for the different types of exceptions. In the **`try`** block, we will invoke the **Persist** method for the **`dbSet`**. The code looks like this:

```
                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();
        }
    }
}
```

The first catch block will handle **`TargetInvocationException`**. If the invoked method **throws an exception**, this is the exception we need to **catch**. Consequently, this block **throws** the inner exception, because this is the actual exception, which occurred within the method invocation, which the **second and third `catch`** blocks will handle.

The second and third **`catch`** blocks will handle **`InvalidOperationException`** and **`SqlException`** respectively. In both cases we, need to **`rollback`** the transaction. If no exceptions are thrown, we're going to **commit** the transaction and **save our changes** to the database.

Now it's time to implement our **`Persist<TEntity>` method**. It accepts a **`DbSet`** as the **generic type** parameter and a **transaction**.

First, we need to create a variable where we can save our **current table name** (string) using the **`GetTableName()`** method (which we'll implement later). Then, we need an array, where we will collect our columns by calling the **`FetchColumnNames()`** method (also implemented later). Then, we check the **dbSet's `ChangeTracker`** for **any** added entities, and if there are any, we use the **`InsertEntities()`** method, which we already have in our **`DbConnection`** class.

```
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);
    }
```

Now we need our **modified** entities. We can get them by using **`GetModifiedEntities()`**, which we will have in our **`ChangeTracker`** class. If there are any modified entities, we **update** our database using **`UpdateEntities()`**, which accepts our **entities**, the **table name** and **table columns** as parameters.

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

Lastly, we check if there are any removed entities using the **`ChangeTracker`**'s **`Removed`** collection. If we have any, we **delete** them from our database too.

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

The next step is to create a method for initializing dbSets called **`InitializeDbSets()`**. For each DB set, we will invoke the **`PopulateDbSet(dbSetProperty)`** method **dynamically**, because we are providing the **generic type parameter** at **runtime**, since we don't know what the framework user's **DB** sets are.

```
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});
    }
}
```

Our next method to implement is **PopulateDbSet<TEntity>()**. We retrieve the entities from the database, using the **LoadTableEntities<TEntity>()** method. Then, we create a new **DbSet<TEntity>** instance, passing the entities to its constructor.

Finally, we need to replace the actual **DbSet** property in the current **DbContext instance** with the one we just created. Since the **Db sets** don't have a setter, we need to replace its backing field, by using the **ReflectionHelper.ReplaceBackingField()** method. This works, because every auto-property has a **private**, **autogenerated backing field**.

```
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);
}
```

Now, we implement a new method, called **MapAllRelations()**. All this method will do is call **MapRelations()** dynamically for each **DB** set **property**. This method looks very similar to the **InitializeDbSets()** method.

```
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});
    }
}
```

Now it's time to implement our **MapRelations<TEntity>()** method, we talked about before. This method accepts a **DbSet<TEntity>** variable as its only parameter.

This method maps all of the relations of the DB set. There are two types of relations: **Foreign key properties**, which map **many-to-one** relations, and **collections**, which map **one-to-many** and **many-to-many** relations. First, we **map** the **navigation properties** and then we **map the collections**. In order to discover what **collections** our **`TEntity`** has, we need to reflect the class and find **every property**, which is of type **`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();
```

After we find our collections, we iterate through them and call the **MapCollection** method **dynamically** for each of them, much like the previo us 2 methods that did something similar.

```
    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});
    }
}
```

Now it's time for the **`MapCollection<TDbSet, TCollection>()`** method's implementation, which accepts a **`DbSet<TDbSet>`** and **`PropertyInfo`** variables as parameters. Now, we need to get the primary and the foreign keys. The primary ones we find by getting all the **properties** which have a **`[Key]` attribute** in the **`collectionType`** variable we declared before. We can get the **foreign key** in the same way but using the **`entityType`** variable.

```
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>());
```

We check if we are dealing with a **many-to-many** relation, which is only true if we have **2 or more** primary keys. If we have a many-to-many relation, we can get the foreign key by finding the first type property, whose name is equal to the **foreign key attribute's name** and has the **same property type** as the entity type.

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

Now we get the collection's DB set, which we will filter with a **`where`** clause and extract all of the entities whose foreign keys are equal to the primary key of the current entity.

Finally, call the **`ReflectionHelper.ReplaceBackingField()`** method and we replace the **`null`** collection with a **populated** collection.

At the end your code should look like this:

```
        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);
        }
    }
```

The next method to implement is **`MapNavigationProperties<TEntity>()`** which accepts a **DB set** as a parameter. This method finds the **entity's foreign keys** (there could be **multiple**) and iterates over them. For each of those foreign keys, we discover what **navigation property** they point to and **its type**. Then, we use this type to get the other side of the relation's **DB set**. Then, **for each entity** in that **DB set**, we find the **first entity**, whose **primary key value is equal** to the **foreign key value** in our **`TEntity`**. Finally, we replace the navigation property's value (which is currently **`null`**) with the entity we found.

```
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);
        }
    }
}
```

We mentioned a method, called **`IsObjectValid()`** which accepts an **`object`** as a parameter and returns a **`bool`** as result. Since the **`Validator`** class is part of **`System.Data.Annotations`**, which is very old, we need to write a bunch of boilerplate code to use it. So instead of writing this everywhere we need to validate an object, we can just extract it into a method. The boilerplate looks like this:

```
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;
}
```

The next method to implement is **`LoadTableEntities<TEntity>()`** method. We have to declare a few variables in it. The first one will keep the type of the **`TEntity`** and it will be our **table**. The next one will be for the **columns** and it will be an **array of strings**. There, we will keep the **column names** for the current table by calling **`GetEntityColumnNames`** (implement this **last**). The third variable will be for the **table name** and we will get it by calling **`GetTableName()`** (implement **second**). The last one and the one our method will return is the **`fetchedRows`** variable. We can get the fetched rows by calling the **`DbConnection`**'s **`FetchResultSet<TEntity>()`** method with the expected parameters.

```
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;
}
```

Let's implement **`GetTableName()`**, which returns a string and gets the **`tableType`** as a parameter. You can implement it yourself 😊

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

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

    return tableName;
}
```

We are almost done with this class. But before that, we need to implement a simple **`DiscoverDbSets()`** method. We used that method in our constructor to populate our **`dbSetProperties`** field, which is a Dictionary with a Type as a key and a **`PropertyInfo`** as a value. Its code is only 2 lines long. Two annoyingly long lines…

```
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;
}
```

The last method is **`GetEntityColumnNames()`**, which returns an **array of strings** with the **column names** and accepts the **table type** as a parameter. Finally, we need to get the **table properties**, which are of **valid SQL types** and are contained in the **column names**. After that, we get the property names (using the **`Select`** **LINQ** extension method) and **return** them.

```
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;
}
```

And with this final method, we are done with the framework! Let's go ahead and test it out by writing a simple application, which utilizes it and defines its own data model.

Create a Simple Client App

Now that the framework is ready, let's see how it discovers our database types, tables, relationships and much more, all using the power of reflection.

4. Create the Database

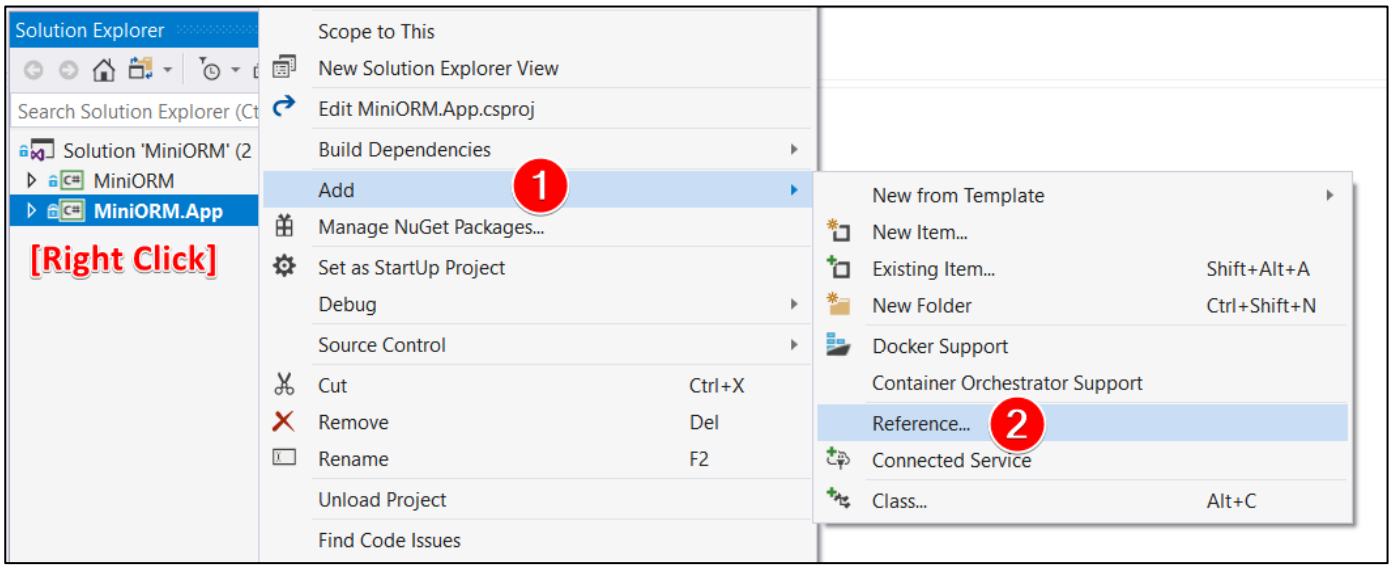
Import this SQL script into SSMS:

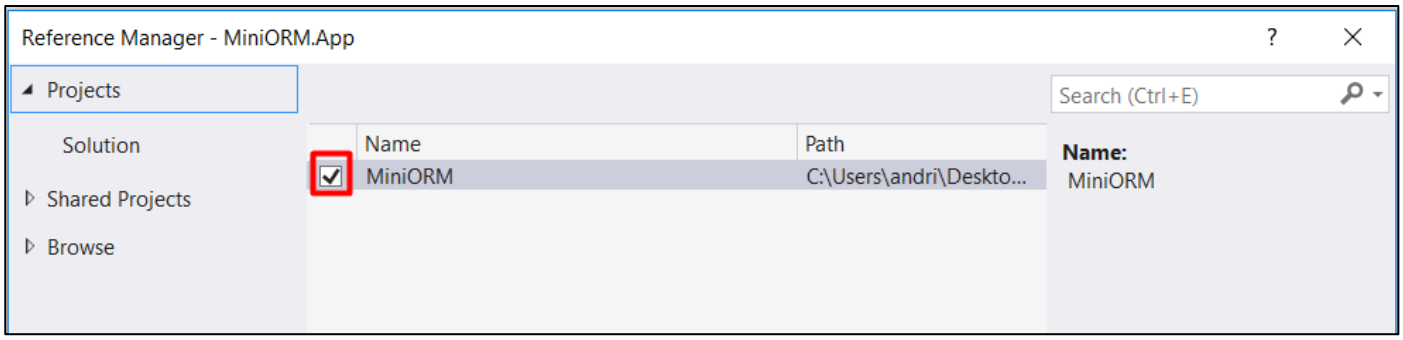
```
CREATE DATABASE MiniORM
GO
USE MiniORM
GO
CREATE TABLE Projects
(
   Id INT IDENTITY PRIMARY KEY,
   Name VARCHAR(50) NOT NULL
)

CREATE TABLE Departments
(
   Id INT IDENTITY PRIMARY KEY,
   Name VARCHAR(50) NOT NULL
)

CREATE TABLE Employees
(
   Id INT IDENTITY PRIMARY KEY,
   FirstName VARCHAR(50) NOT NULL,
   MiddleName VARCHAR(50),
   LastName VARCHAR(50) NOT NULL,
   IsEmployed BIT NOT NULL,
   DepartmentId INT
   CONSTRAINT FK_Employees_Departments FOREIGN KEY
   REFERENCES Departments(Id)
)
CREATE TABLE EmployeesProjects
(
   ProjectId INT NOT NULL
   CONSTRAINT FK_Employees_Projects REFERENCES Projects(Id),
   EmployeeId INT NOT NULL
   CONSTRAINT FK_Employees_Employee REFERENCES Employees(Id),
   CONSTRAINT PK_Projects_Employees
   PRIMARY KEY (ProjectId, EmployeeId)
)
GO
INSERT INTO MiniORM.dbo.Departments (Name) VALUES ('Research');
```

```
INSERT INTO MiniORM.dbo.Employees (FirstName, MiddleName, LastName, IsEmployed, DepartmentId) VALUES
('Stamat', NULL, 'Ivanov', 1, 1),
('Petar', 'Ivanov', 'Petrov', 0, 1),
('Ivan', 'Petrov', 'Georgiev', 1, 1),
('Gosho', NULL, 'Ivanov', 1, 1);
INSERT INTO MiniORM.dbo.Projects (Name)
VALUES ('C# Project'), ('Java Project');
INSERT INTO MiniORM.dbo.EmployeesProjects (ProjectId, EmployeeId) VALUES
(1, 1),
(1, 3),
(2, 2),
(2, 3)
```

5. Create the Project

Create a new C# **Console Application**, called "**MiniORM.App**" and **add a reference** to the **MiniORM** project:

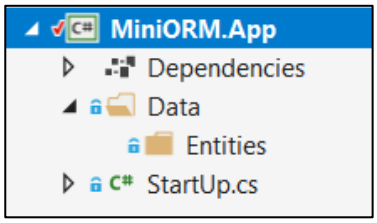




6. Define the Data Model

Now it's time to create our data models using all the information we have in our database.

Create a **Data** directory, and inside it, create an **Entities** directory. When you're done, you should have the following folder structure:



Now, let's get to creating the data model. First, create a **Department** class inside of the **Entities** folder. Entity classes have **one property for each column** of the table.

Create two properties – **Id** and **Name**. For the **Id**, use the **[Key]** annotation (**using System.ComponentModel.DataAnnotations**) to let our framework know that this is the **primary key** of the entity. We can forbid the **Name property** from having a **null value** upon calling **SaveChanges()** by adding the **[Required]** annotation to it. Our **framework** takes care of **validating every object** before persisting any changes. Finally, add an **ICollection** of employees as a **navigation property** for all of the **employees**, who belong to a particular **department**. When you're done, the class should look like this:

```
namespace MiniORM.App.Data.Entities
{
    using System.Collections.Generic;
    using System.ComponentModel.DataAnnotations;

    public class Department
    {
        [Key]
        public int Id { get; set; }

        [Required]
        public string Name { get; set; }

        public ICollection<Employee> Employees { get; }
    }
}
```

Next, create a **Project** class with an **Id** and a **Name**. The **Id** is our **primary key**, while the **Name** should be **required** (not null). Additionally, create a **navigation property**, called **EmployeesProjects**, which is a mapping entity between our **Employee** and **Project** entities. We'll create this class later.

It's generally a good idea to use a **get-only** property of type **ICollection<T>** for our navigation properties to prevent them **from being redeclared** outside of our framework. When you're done, your code should look like this:

```
namespace MiniORM.App.Data.Entities
{
    using System.Collections.Generic;
    using System.ComponentModel.DataAnnotations;

    public class Project
    {
        [Key]
        public int Id { get; set; }

        [Required]
        public string Name { get; set; }

        public ICollection<EmployeeProject> EmployeeProjects { get; }
    }
}
```

After that, create an **Employee** class and use the same logic. The only difference between the other two models we've created is that in the **Employee** class, we need a **foreign key** to our **Department** model. We can do that by using **[ForeignKey(nameof(Department))]** annotation above the **DepartmentId** property.

```
namespace MiniORM.App.Data.Entities
{
    using System.Collections.Generic;
    using System.ComponentModel.DataAnnotations;
    using System.ComponentModel.DataAnnotations.Schema;

    public class Employee
    {
        [Key]
        public int Id { get; set; }

        [Required]
        public string FirstName { get; set; }

        public string MiddleName { get; set; }

        [Required]
        public string LastName { get; set; }

        public bool IsEmployed { get; set; }

        [ForeignKey(nameof(Department))]
        public int DepartmentId { get; set; }

        public Department Department { get; set; }

        public ICollection<EmployeeProject> EmployeeProjects { get; }
    }
}
```

The last class to create is **`EmployeesProject`**, where we will have a **composite** key for the **`Projects`** and **`EmployeesId`** property. Then make the two composite keys **foreign** keys too.

```
namespace MiniORM.App.Data.Entities
{
    using System.ComponentModel.DataAnnotations;
    using System.ComponentModel.DataAnnotations.Schema;

    public class EmployeeProject
    {
        [Key]
        [ForeignKey(nameof(Employee))]
        public int EmployeeId { get; set; }

        [Key]
        [ForeignKey(nameof(Project))]
        public int ProjectId { get; set; }

        public Employee Employee { get; set; }

        public Project Project { get; set; }
    }
}
```

Now, let's create our **`DbContext`** class. Create a **`SoftUniDbContextClass`** in the **`Data`** folder, which **inherits** from our base **`DbContext`** class and has **`DbSets`** for all the **models** we've created. Make sure to inherit the constructor too.

```
namespace MiniORM.App.Data
{
    using Entities;

    public class SoftUniDbContext : DbContext
    {
        public SoftUniDbContext(string connectionString)
            : base(connectionString)
        {
        }

        public DbSet<Employee> Employees { get; }

        public DbSet<Department> Departments { get; }

        public DbSet<Project> Projects { get; }

        public DbSet<EmployeeProject> EmployeesProjects { get; }
    }
}
```

That's it! Our data model is done. Now it's time to test out the framework.

7. Test the Framework

Let's test our **`MiniORM`** Framework by inserting some sample data in our database. Go to your main method and declare your **connection string**. After that create an instance of the **`SoftUniDbContext`** class with your connection string and insert a new **`Employee`**. Then, find the **first employee** and modify their **name**. Finally, call the context's **`SaveChanges()`** method.

```
namespace MiniORM.App
{
    using System.Linq;
    using Data;
    using Data.Entities;

    public class StartUp
    {
        public static void Main(string[] args)
        {
            var connectionString = "Server=.;Database=MiniORM;Integrated Security=True";

            var context = new SoftUniDbContext(connectionString);

            context.Employees.Add(new Employee
            {
                FirstName = "Gosho",
                LastName = "Inserted",
                DepartmentId = context.Departments.First().Id,
                IsEmployed = true,
            });

            var employee = context.Employees.Last();
            employee.FirstName = "Modified";

            context.SaveChanges();
        }
    }
}
```

If everything works without any exceptions, we should be done! You've just gained some valuable insight into how an ORM Framework like **Entity Framework Core** is written. In fact, the **`MiniORM.App`** code can be transferred to a project, using **Entity Framework Core** and it will work **identically** without requiring any syntax changes!

You can try extending the framework by implementing extra stuff like **concurrency control** by yourself. Happy coding 😊