How can I implement a transaction for my repositories with Entity Framework?

Asked 3 years, 5 months ago Active 1 month ago Viewed 11k times

6

5

I am trying to utilize the repository design pattern in my application for 2 reasons

1. I like to de-couple my application from Entity in case I decide to not use Entity Framework at some point
2. I want to be able reuse the logic that interacts with the model

I successfully setup and used the repository pattern. However, I have one complexity to deal with which is a transaction.

I want to be able to use transaction so that I can make multiple calls to the repository and then commit or rollback.

Here is my repository interface

```
using System;
using System.Collections.Generic;
using System.Linq.Expressions;

namespace Support.Repositories.Contracts
{
    public interface IRepository<TModel> where TModel : class
    {
        // Get records by it's primary key
        TModel Get(int id);

        // Get all records
        IEnumerable<TModel> GetAll();

        // Get all records matching a lambda expression
        IEnumerable<TModel> Find(Expression<Func<TModel, bool>> predicate);

        // Get the a single matching record or null
        TModel SingleOrDefault(Expression<Func<TModel, bool>> predicate);

        // Add single record
        void Add(TModel entity);

        // Add multiple records
        void AddRange(IEnumerable<TModel> entities);

        // Remove records
```

```
        void Remove(TModel entity);

        // remove multiple records
        void RemoveRange(IEnumerable<TModel> entities);
    }
}
```

Then I create an implementation for Entity Framework like so

```
using Support.Repositories.Contracts;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Linq.Expressions;

namespace Support.Repositories
{
    public class EntityRepository<TEntity> : IRepository<TEntity>
        where TEntity : class
    {
        protected readonly DbContext Context;
        protected readonly DbSet<TEntity> DbSet;

        public EntityRepository(DbContext context)
        {
            Context = context;
            DbSet = context.Set<TEntity>();
        }

        public TEntity Get(int id)
        {
            return DbSet.Find(id);
        }

        public IEnumerable<TEntity> GetAll()
        {
            return DbSet.ToList();
        }

        public IEnumerable<TEntity> Find(Expression<Func<TEntity, bool>> predicate)
        {
            return DbSet.Where(predicate);
        }

        public TEntity SingleOrDefault(Expression<Func<TEntity, bool>> predicate)
        {
```

```
            return DbSet.SingleOrDefault(predicate);
        }

        public void Add(TEntity entity)
        {
            DbSet.Add(entity);
        }

        public void AddRange(IEnumerable<TEntity> entities)
        {
            DbSet.AddRange(entities);
        }

        public void Remove(TEntity entity)
        {
            DbSet.Remove(entity);
        }

        public void RemoveRange(IEnumerable<TEntity> entities)
        {
            DbSet.RemoveRange(entities);
        }

    }
}
```

Now, I create a `IUnitOfWork` to interact with repository like so

```
using System;

namespace App.Repositories.Contracts
{
    public interface IUnitOfWork : IDisposable
    {
        IUserRepository Users { get; }
        IAddressRepository Addresses { get;  }
    }
}
```

Then I implemented this interface for Entity Framework like this:

```
using App.Contexts;
using App.Repositories.Contracts;
using App.Repositories.Entity;

namespace App.Repositories
```

```
{
    public class UnitOfWork : IUnitOfWork
    {
        private readonly AppContext _context;
        public IUserRepository  Users { get; private set; }
        public IAddressRepository Addresses { get; private set; }

        public UnitOfWork(AppContext context)
        {
            _context = context;

            Users = new UserRepository(_context);
            Addresses = new AddressRepository(_context);
        }

        public UnitOfWork() : this(new AppContext())
        {
        }

        public int Save()
        {
            return _context.SaveChanges();
        }

        public void Dispose()
        {
            _context.Dispose();
        }
    }
}
```

I am able to use the repository like this

```
using(var repository = new UnitOfWork())
{
     repository.Users.Add(new User(... User One ...))
     repository.Save();

     repository.Addresses(new Address(... Address For User One ...))
     repository.Save();

     repository.Users.Add(new User(... User Two...))
     repository.Save();

     repository.Addresses(new Address(... Address For User Two...))
     repository.Save();
}
```

Now, I want to be able to use database transaction so only when everything is good then commit otherwise rollback.

My first take is to add a new method called `BeginTransaction()` to my `UnitOfWork` class. But will couple my code to Entity Framework only.

Now, I am thinking to create a new interface that provides `BeginTransaction()`, `Commit()` and `Rollback()` method which will allow me to write an implementation for any ORM.

i.e.

```
namespace Support.Contracts
{
    public IRepositoryDatabase
    {
        SomethingToReturn BeginTransaction();

        void Commit();
        void Rollback();
    }
}
```

The question is how would I tie `IRepositoryDatabase` back to my UnitOfWork so I can implement correctly? And what would `BeginTransaction()` needs to return?

c# entity-framework transactions repository repository-pattern

edited Oct 7 '16 at 5:19
marc_s
627k 142 1192 1323

asked Oct 6 '16 at 22:18
Junior
9,052 12 54 127

EF will store by default all changed entities or none on Save() call. – Sir Rufo Oct 6 '16 at 22:29

but how would I begin transaction? – Junior Oct 6 '16 at 22:30

2 Change 10 entities and after that call Save(). It will be written in a single transaction by EF (or nothing will be written, if any error occurs). – Sir Rufo Oct 6 '16 at 22:32

1 var user = new User(); var address = new Address { User = user }; – Sir Rufo Oct 6 '16 at 22:40

1 What if I want to change to a different framework and not entity. May be the other framework does not handle everything like Entity. Wouln't I need some

kind of a wrapper to ensure this will work the same regardless of the implementation? – Junior Oct 6 '16 at 22:44

2 Answers

13

I think I figured out the way to do it. (I hope I did it the right way)

Here is what I have done, I hope this helps someone looking to do the same thing.

I created a new Interface like so

```
using System;

    namespace Support.Repositories.Contracts
    {
        public interface IDatabaseTransaction : IDisposable
        {
            void Commit();

            void Rollback();
        }
    }
```

Then I implemented `IDatabaseTransaction` for Entity framework like so

```
using Support.Repositories.Contracts;
using System.Data.Entity;

namespace Support.Entity.Repositories
{
    public class EntityDatabaseTransaction : IDatabaseTransaction
    {
        private DbContextTransaction _transaction;

        public EntityDatabaseTransaction(DbContext context)
        {
            _transaction = context.Database.BeginTransaction();
        }

        public void Commit()
        {
            _transaction.Commit();
```

```
        }

        public void Rollback()
        {
            _transaction.Rollback();
        }

        public void Dispose()
        {
            _transaction.Dispose();
        }
    }
}
```

Then, I added a new method called `BeginTransaction()` to my `IUnitOfWork` contract like so

```
using System;

namespace App.Repositories.Contracts
{
    public interface IUnitOfWork : IDisposable
    {
        IDatabaseTransaction BeginTransaction();
        IUserRepository Users { get; }
        IAddressRepository Addresses { get;  }
    }
}
```

Finally, following is my `UnitOfwork` implementation for Entity

```
using App.Contexts;
using App.Repositories.Contracts;
using App.Repositories.Entity;
using Support.Repositories;


namespace App.Repositories
{
    public class UnitOfWork : IUnitOfWork
    {
        private readonly AppContext _context;
        public IUserRepository  Users { get; private set; }
        public IAddressRepository Addresses { get; private set; }

        public UnitOfWork(AppContext context)
        {
```

```
            _context = context;

            Users = new UserRepository(_context);
            Addresses = new AddressRepository(_context);
        }

        public UnitOfWork() : this(new AppContext())
        {
        }

        public int Save()
        {
            return _context.SaveChanges();
        }

        public void Dispose()
        {
            _context.Dispose();
        }

        public IDatabaseTransaction BeginTransaction()
        {
            return new EntityDatabaseTransaction(_context);
        }
    }
}
```

And here is how I consume the UnitOfWork implementation from my controller

```
using(var unitOfWork = new UnitOfWork())
using(var transaction = new unitOfWork.BeginTransaction())
{
     try
     {
         repository.Users.Add(new User(... User One ...))
         repository.Save();

         repository.Addresses(new Address(... Address For User One ...))
         repository.Save();

         repository.Users.Add(new User(... User Two...))
         repository.Save();

         repository.Addresses(new Address(... Address For User Two...))
         repository.Save();
         transaction.Commit();
     }
```

```
        catch(Exception)
        {
            transaction.Rollback();
        }

    }
```

edited Jan 15 at 8:13

bereket gebredingle
2,295 1 11 27

answered Oct 7 '16 at 16:03

Junior
9,052 12 54 127

2 Shouldn't be `repository` changed with `unitOfWork` variable name in working example at the bottom? – Xawery Wiśniowiecki Dec 6 '16 at 10:24

0

While comments by Sir Rufo are correct, you did said that wanted an EF independent solution and although usually abstracting away from the ORM is an overkill, if you are still set on handling the transaction yourself you can use `TransactionScope` (which was apparently the way to achieve control over the transaction before having `BeginTransaction` in the `context.Database`).

Please see the following article for details: https://msdn.microsoft.com/en-us/data/dn456843.aspx

Relevant bits are that you can enclose all the calls in a `TransactionScope` (this will actually work out of the box in other ORMs as well):

```
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.SqlClient;
using System.Linq;
using System.Transactions;

namespace TransactionsExamples
{
    class TransactionsExample
    {
        static void UsingTransactionScope()
        {
            using (var scope = new TransactionScope(TransactionScopeOption.Required))
            {
                using (var conn = new SqlConnection("..."))
                {
                    conn.Open();

                    var sqlCommand = new SqlCommand();
                    sqlCommand.Connection = conn;
```

```
                    sqlCommand.CommandText =
                        @"UPDATE Blogs SET Rating = 5" +
                            " WHERE Name LIKE '%Entity Framework%'";
                    sqlCommand.ExecuteNonQuery();

                    using (var context =
                        new BloggingContext(conn, contextOwnsConnection: false))
                    {
                        var query = context.Posts.Where(p => p.Blog.Rating > 5);
                        foreach (var post in query)
                        {
                            post.Title += "[Cool Blog]";
                        }
                        context.SaveChanges();
                    }
                }

                scope.Complete();
            }
        }
    }
}
```

But do need to mind the following caveats:

There are still some limitations to the `TransactionScope` approach:

- Requires .NET 4.5.1 or greater to work with asynchronous methods
- It cannot be used in cloud scenarios unless you are sure you have one and only one connection (cloud scenarios do not support distributed transactions)
- It cannot be combined with the `Database.UseTransaction()` approach of the previous sections
- It will throw exceptions if you issue any DDL (e.g. because of a Database Initializer) and have not enabled distributed transactions through the MSDTC Service

edited Oct 7 '16 at 5:20

marc_s
627k 142 1192 1323

answered Oct 6 '16 at 22:45

Fredy Treboux
2,457 1 21 29

Can I wrap my commands into a method that accepts action? something like this? `public void ExecuteWithInTransaction(Action actions) { var trans = _context.Database.BeginTransaction(); try { actions(); trans.Commit(); } catch (Exception) { trans.Rollback(); } }` – Junior Oct 6 '16 at 23:25

I suppose you can, but it seems very unorthodox and it still ties your ExecuteWithInTransaction to EF I think unless you make it an interface and provide the implementation somehow. – Fredy Treboux Oct 7 '16 at 0:08