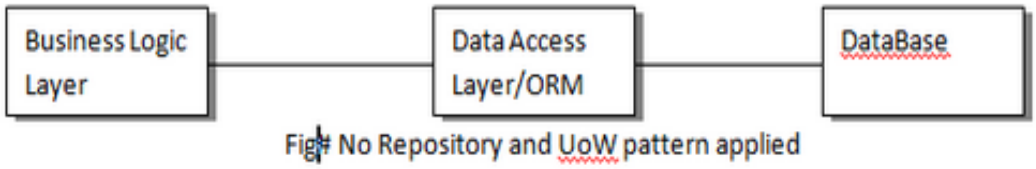
Repository Pattern and Unit of Work with Entity Framework

Repository pattern is used to create an abstraction layer between the data access layer and the business logic layer. This abstraction layer contains data manipulation methods which communicate with the data access layer to serve data as per the business requirements to the logical layer. The main purpose to create this layer is for isolating data access layer so that changes cannot affect the business logic layer directly.

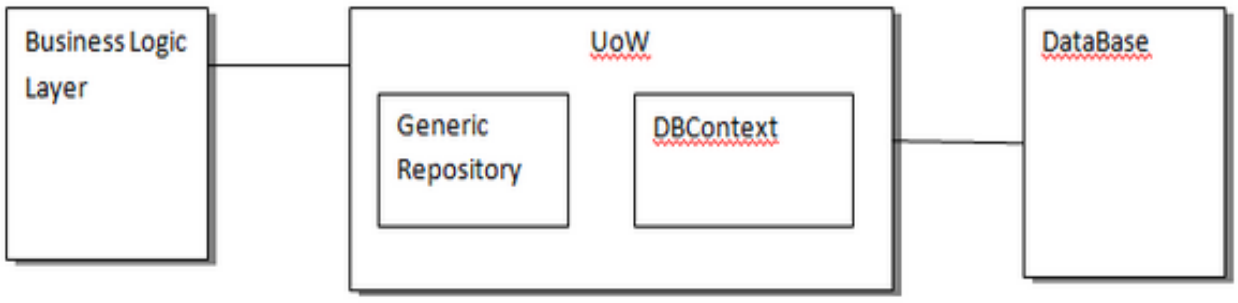
For example, a POS application is usually deployed to different stores. A few new clients wanted to use the Microsoft SQL Server database system and some others wanted Oracle and other databases. Also they had a few different relational database systems to store their data but business logic was almost the same. Repository pattern helps developers to detach the layer and add a new data access layer.

Unit of work is a pattern to handle transaction during data manipulation using the Repository pattern. So we can say, according to Martin Fowler, *Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problem.*

Implementing these patterns can help insulate the application from changes in the data store and also gives advantages to automate unit testing.

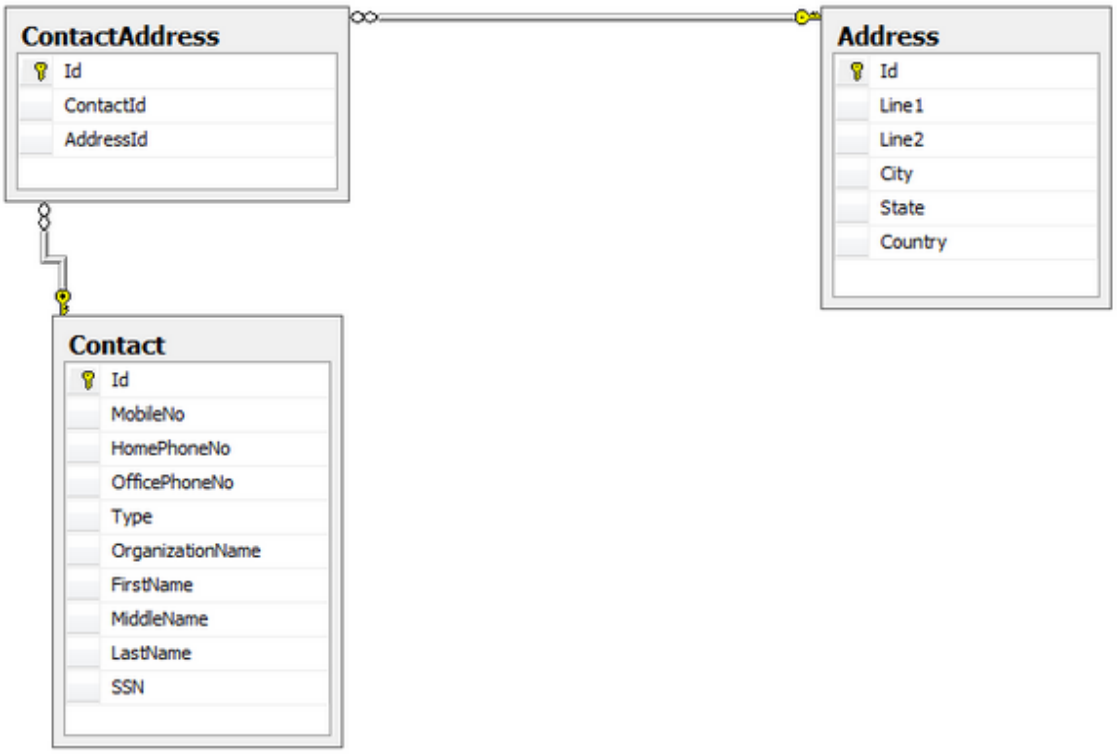


Fig# No Repository and UoW pattern applied



Fig# Repository and UoW pattern applied to access database

A database for the application with the following tables:



Contact.Repository class as well as its required interfaces.

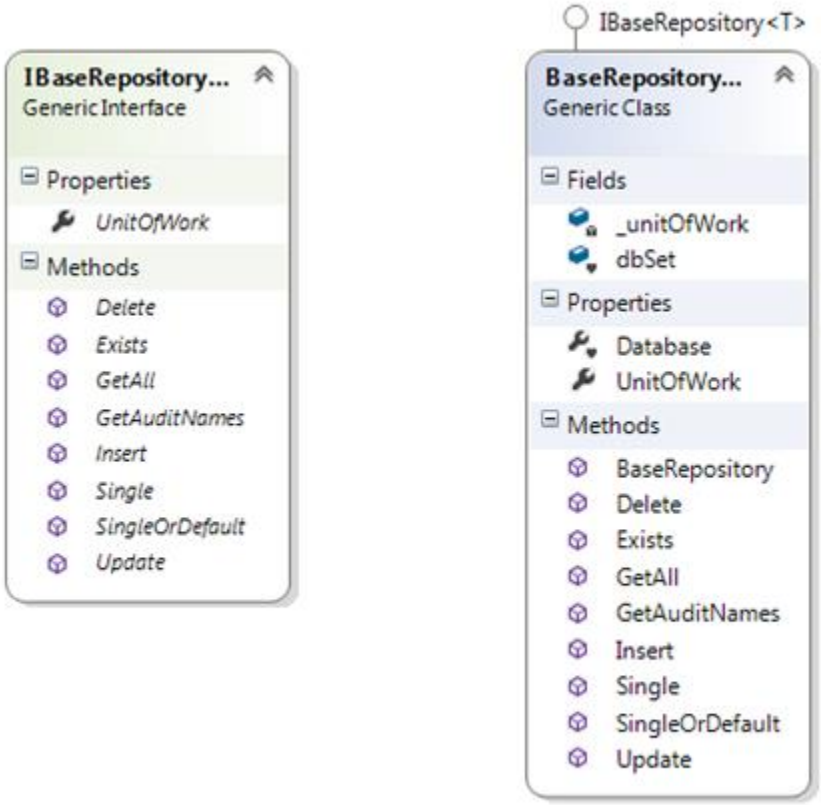


Fig: Interface and class diagram given for EntityFramework Repository

```
public interface IBaseRepository<T>
   {

      T Single(object primaryKey);

      T SingleOrDefault(object primaryKey);

      IEnumerable<T> GetAll();

      bool Exists(object primaryKey);

      int Insert(T entity);
```

```
        void Update(T entity);

public interface IUnitOfWork : IDisposable
    {

        /// <summary>
        /// Call this to commit the unit of work
        /// </summary>
        void Commit();

        /// <summary>
        /// Return the database reference for this UOW
        /// </summary>
        DbContext Db { get; }

        /// <summary>
        /// Starts a transaction on this unit of work
        /// </summary>
        void StartTransaction();


    }

public class UnitOfWork : IUnitOfWork
    {
        private TransactionScope _transaction;
        private readonly ContactDBEntities _db;


        public UnitOfWork()
        {
            _db = new ContactDBEntities();

        }

        public void Dispose()
        {

        }
```

```
        public void StartTransaction()
        {

            _transaction = new Transaction Scope();




        }

        public void Commit()
        {
            _db.SaveChanges();
            _transaction.Complete();
        }

        public DbContext Db
        {
            get { return _db; }
        }



    }

public class BaseRepository<T> : IBaseRepository<T>
        where T : class
{
    private readonly IUnitOfWork _unitOfWork;
    internal DbSet<T> dbSet;
    public BaseRepository(IUnitOfWork unitOfWork)
    {
        if (unitOfWork == null) throw new ArgumentNullException("unitOfWork");
        _unitOfWork = unitOfWork;
        this.dbSet = _unitOfWork.Db.Set<T>();
    }
```

```
/// <summary>
/// Returns the object with the primary key specifies or throws
/// </summary>
/// <typeparam name="TU">The type to map the result to</typeparam>
/// <param name="primaryKey">The primary key</param>
/// <returns>The result mapped to the specified type</returns>
public T Single(object primaryKey)
{
   var dbResult = dbSet.Find(primaryKey);
   return dbResult;
}

public T SingleOrDefault(object primaryKey)
{
   var dbResult = dbSet.Find(primaryKey);
   return dbResult;
}

public bool Exists(object primaryKey)
{
   return dbSet.Find(primaryKey) == null ? false : true;
}

public virtual int Insert(T entity)
{
  dynamic obj= dbSet.Add(entity);
  this._unitOfWork.Db.SaveChanges();
  return obj.Id;
}

public virtual void Update(T entity)
{
   dbSet.Attach(entity);
   _unitOfWork.Db.Entry(entity).State = EntityState.Modified;
   this._unitOfWork.Db.SaveChanges();
}

public int Delete(T entity)
{
   if (_unitOfWork.Db.Entry(entity).State == EntityState.Detached)
```

```
        {
            dbSet.Attach(entity);
        }
        dynamic obj=dbSet.Remove(entity);
        this._unitOfWork.Db.SaveChanges();
        return obj.Id;
    }

    public IUnitOfWork UnitOfWork { get { return _unitOfWork; } }
    internal DbContext Database { get { return _unitOfWork.Db; } }

        public IEnumerable<T> GetAll()
    {
        return dbSet.AsEnumerable().ToList();
    }
    }
}
```

```
public class ContactRepository: BaseRepository<Contact>
    {

        public ContactRepository(IUnitOfWork unit):base(unit)
        {

        }

    }
public class AddressRepository : BaseRepository<Address>
{
  public AddressRepository(IUnitOfWork unit)
   : base(unit)
   {
   }
}
```

Testing driver

```
    private IUnitOfWork uow = null;
    private ContactRepository repo = null;
     uow = new UnitOfWork();
     repo = new ContactRepository(uow);

        var contacts = repo.GetAll();
        return View(contacts.ToList());

       Contact contact = repo.SingleOrDefault(id);
        repo.Insert(contact);
       repo.Update(contact);
        Contact contact = repo.SingleOrDefault(id);
        repo.Delete(contact);

    protected override void Dispose(bool disposing)
    {
        uow.Dispose();
        base.Dispose(disposing);
    }
}
```

You have to build the UnitOfWork class by implementing IUnitOfWork and this class will be responsible for creating the EF. Then the repository class (ContactRepository) will be initialized with the uow object.