# Unit of Work using Repository Design Pattern

Back to: Dot Net Design Patterns With Real-Time Examples

Apron Patterns
Backyard Design Software
SQL Certification Training
Project Timeline Template
Network Management Software
Online Web Page Maker

## Unit of Work using Repository Design Pattern

The unit of work implementation manages in-memory database CRUD operations on entities as one transaction. So, if one of the operations is failed then the entire database operations will be rollback.

Unit of Work is the concept that is related to the effective implementation of the Repository design Pattern. So, to understand this concept it is important to understand the concept of the Repository Pattern.

Please read Part – 1, 2, 3 and 4 of this article series where we discussed Repository pattern in ASP.NET MVC application using Entity Framework.

In Part – 1, I discussed the basics of Repository Design pattern

In Part – 2, I discussed Non-Generic Repository Pattern

In Part – 3, I discussed generic Repository pattern.

### The Repository Pattern

As we already discussed a repository is nothing but a class defined for an entity, with all the possible database operations. For example, a repository for an Employee entity will have the basic CRUD operations and any other possible operations related to the Employee entity. A Repository Pattern can be implemented in two ways:

### One repository per entity (non-generic):

This type of implementation involves the use of one repository class for each entity. For example, if you have two entities, Employee, and Customer, each entity will have its own repository. We discussed this In Part – 2, of this article series with some example.

### Generic repository:

A generic repository is the one that can be used for all the entities, in other words, it can be either used for Employee or Customer or any other entity. We discussed this In Part – 3, of this article series with some example.

## Unit of Work in the Repository Pattern

The Unit of Work pattern is used to group one or more operations (usually database CRUD operations) into a single transaction or "unit of work" so that all operations either pass or fail as one. In simple word we can say that for a specific user action, say booking on a website, all the transactions like insert/update/delete and so on are done in one single transaction, rather than doing multiple database transactions. This means, one unit of work here involves insert/update/delete operations, all in one single transaction so that all operations either pass or fail as one

We are going to work with the same example that we used in our previous article.

Below is the code snippet for the Generic Repository class

```
namespace RepositoryUsingEFinMVC.GenericRepository
{
    public class GenericRepository<T> : IGenericRepository<T> where T : class
    {
```

```
public EmployeeDBContext _context = null;
public DbSet<T> table = null;

public GenericRepository()
{
    this._context = new EmployeeDBContext();
    table = _context.Set<T>();
}

public GenericRepository(EmployeeDBContext _context)
{
    this._context = _context;
    table = _context.Set<T>();
}

public IEnumerable<T> GetAll()
{
    return table.ToList();
}

public T GetById(object id)
{
    return table.Find(id);
}

public void Insert(T obj)
{
    table.Add(obj);
}

public void Update(T obj)
{
    table.Attach(obj);
    _context.Entry(obj).State = EntityState.Modified;
}
```

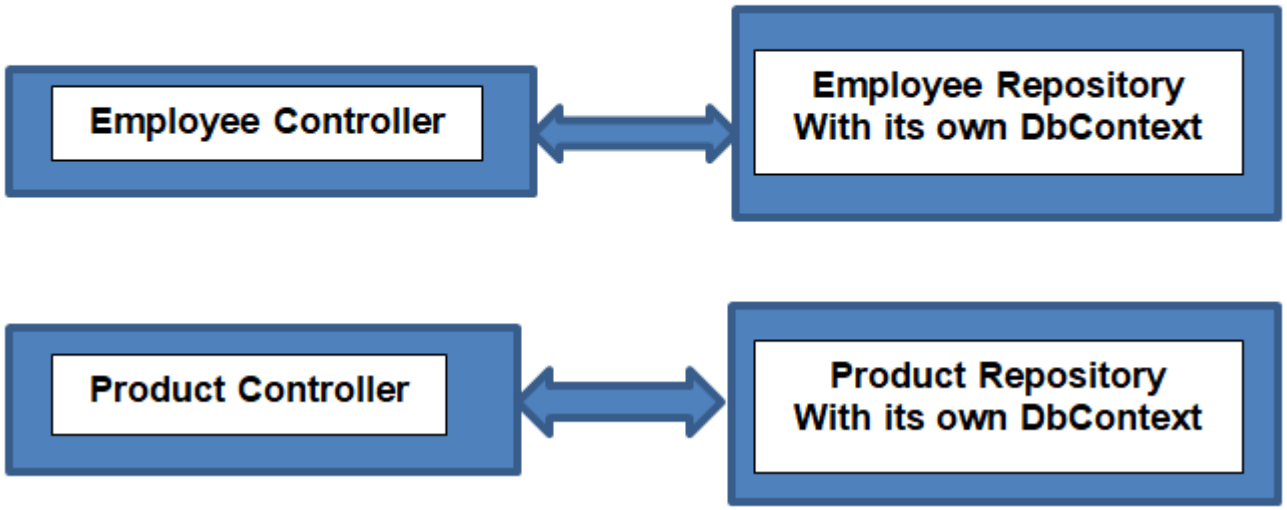
```
        public void Delete(object id)
        {
            T existing = table.Find(id);
            table.Remove(existing);
        }

        public void Save()
        {
            _context.SaveChanges();
        }
    }
}
```

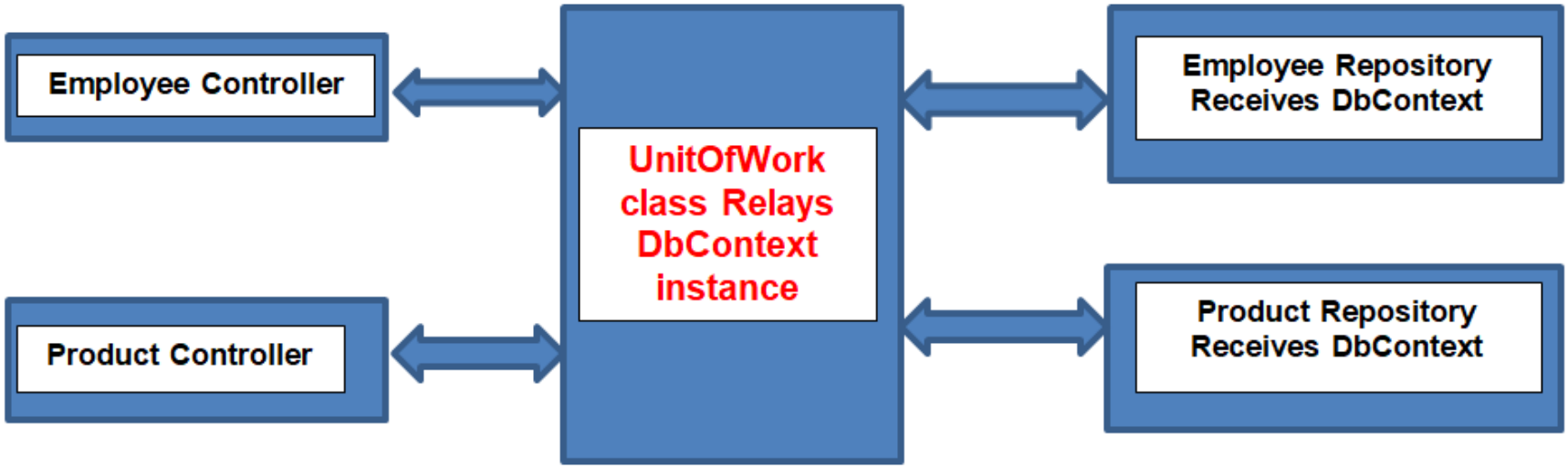
The code above seems to be fine. The issue arises when we add a repository to another entity, say Product. In that case, both repositories will generate and maintain their own instance of the DbContext class. This may lead to issues in the future, since each DbContext will have its own in-memory list of changes of the records, of the entities, that are being added/updated/modified, in a single transaction/operation. In such a case, if the SaveChanges of one of the repository fails and other one succeeds, it will result in database inconsistency. This is where the concept of UnitOfWork is playing an important role.

To avoid this, we will add another layer or intermediate between the controller and the Generic/Non-Generic repository. This layer will act as a centralized store for all the repositories to receive the instance of the DbContext. This will ensure that, for a unit of transaction that spans across multiple repositories, should either complete for all entities or should fail entirely, as all of them will share the same instance of the DbContext. In our above example, while adding data for the Employee and Product entities, in a single transaction, both will use the same DbContext instance. This situation, without and with Unit of work, can be represented as in the following diagram

## WITHOUT UNIT OF WORK




## WITH UNIT OF WORK



In the above representation, during a single operation, that involves Employee and Product entities, both of them use the same DbContext instance. This will ensure that even if one of them breaks, the other one is also not saved, thus maintaining the database consistency. So when SaveChanges is executed, it will be done for both of the repositories.

### Let us understand unit of work with one example.

Add a folder with the name UnitOfWork within the project.

Once the folder is added, add one interface with the name IUnitOfWork within UnitOfWork folder and copy and paste the below code.

```
using System.Data.Entity;
namespace RepositoryUsingEFinMVC.UnitOfWork
{
    public interface IUnitOfWork<out TContext>
        where TContext : DbContext, new()
    {
        TContext Context { get; }
        void CreateTransaction();
        void Commit();
        void Rollback();
        void Save();
    }
}
```

Next add one class file with the name UnitOfWork within UnitOfWork folder and then copy and paste the below code

```
using RepositoryUsingEFinMVC.GenericRepository;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.Entity.Validation;
namespace RepositoryUsingEFinMVC.UnitOfWork
{
```

```
public class UnitOfWork<TContext> : IUnitOfWork<TContext>, IDisposable
where TContext : DbContext, new()
{
//Here TContext is nothing but your DBContext class
//In our example it is EmployeeDBContext class
private readonly TContext _context;
private bool _disposed;
private string _errorMessage = string.Empty;
private DbContextTransaction _objTran;
private Dictionary<string, object> _repositories;
//Using the Constructor we are initializing the _context variable is nothing but
//we are storing the DBContext (EmployeeDBContext) object in _context variable
public UnitOfWork()
{
_context = new TContext();
}
//The Dispose() method is used to free unmanaged resources like files,
//database connections etc. at any time.
public void Dispose()
{
Dispose(true);
GC.SuppressFinalize(this);
}
//This Context property will return the DBContext object i.e. (EmployeeDBContext) object
public TContext Context
{
get { return _context; }
}
//This CreateTransaction() method will create a database Trnasaction so that we can do database
operations by
//applying do evrything and do nothing principle
public void CreateTransaction()
{
_objTran = _context.Database.BeginTransaction();
}
```

```
//If all the Transactions are completed successfuly then we need to call this Commit()
//method to Save the changes permanently in the database
public void Commit()
{
_objTran.Commit();
}
//If atleast one of the Transaction is Failed then we need to call this Rollback()
//method to Rollback the database changes to its previous state
public void Rollback()
{
_objTran.Rollback();
_objTran.Dispose();
}
//This Save() Method Implement DbContext Class SaveChanges method so whenever we do a transaction
we need to
//call this Save() method so that it will make the changes in the database
public void Save()
{
try
{
_context.SaveChanges();
}
catch (DbEntityValidationException dbEx)
{
foreach (var validationErrors in dbEx.EntityValidationErrors)
foreach (var validationError in validationErrors.ValidationErrors)
_errorMessage += string.Format("Property: {0} Error: {1}", validationError.PropertyName,
validationError.ErrorMessage) + Environment.NewLine;
throw new Exception(_errorMessage, dbEx);
}
}
protected virtual void Dispose(bool disposing)
{
if (!_disposed)
if (disposing)
```

```
_context.Dispose();
_disposed = true;
}
public GenericRepository<T> GenericRepository<T>() where T : class
{
if (_repositories == null)
_repositories = new Dictionary<string, object>();
var type = typeof(T).Name;
if (!_repositories.ContainsKey(type))
{
var repositoryType = typeof(GenericRepository<T>);
var repositoryInstance = Activator.CreateInstance(repositoryType.MakeGenericType(typeof(T)),
_context);
_repositories.Add(type, repositoryInstance);
}
return (GenericRepository<T>)_repositories[type];
}
}
}
```

**Next Modify the IGenericRepository.cs file as shown below**

```
using System.Collections.Generic;
namespace RepositoryUsingEFinMVC.GenericRepository
{
public interface IGenericRepository<T> where T : class
{
IEnumerable<T> GetAll();
T GetById(object id);
void Insert(T obj);
void Update(T obj);
void Delete(T obj);
}
}
```

**Next modify the GenericRepository.cs class file as shown below**

```
using RepositoryUsingEFinMVC.DAL;
using RepositoryUsingEFinMVC.UnitOfWork;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.Entity.Validation;
using System.Linq;
namespace RepositoryUsingEFinMVC.GenericRepository
{
public class GenericRepository<T> : IGenericRepository<T>, IDisposable where T : class
{
private IDbSet<T> _entities;
private string _errorMessage = string.Empty;
private bool _isDisposed;
public GenericRepository(IUnitOfWork<EmployeeDBContext> unitOfWork)
: this(unitOfWork.Context)
{
}
public GenericRepository(EmployeeDBContext context)
{
_isDisposed = false;
Context = context;
}
public EmployeeDBContext Context { get; set; }
public virtual IQueryable<T> Table
{
get { return Entities; }
}
protected virtual IDbSet<T> Entities
{
get { return _entities ?? (_entities = Context.Set<T>()); }
}
```

```
public void Dispose()
{
if (Context != null)
Context.Dispose();
_isDisposed = true;
}
public virtual IEnumerable<T> GetAll()
{
return Entities.ToList();
}
public virtual T GetById(object id)
{
return Entities.Find(id);
}
public virtual void Insert(T entity)
{
try
{
if (entity == null)
throw new ArgumentNullException("entity");
Entities.Add(entity);
if (Context == null || _isDisposed)
Context = new EmployeeDBContext();
//Context.SaveChanges(); commented out call to SaveChanges as Context save changes will be
//called with Unit of work
}
catch (DbEntityValidationException dbEx)
{
foreach (var validationErrors in dbEx.EntityValidationErrors)
foreach (var validationError in validationErrors.ValidationErrors)
_errorMessage += string.Format("Property: {0} Error: {1}", validationError.PropertyName,
validationError.ErrorMessage) + Environment.NewLine;
throw new Exception(_errorMessage, dbEx);
}
}
```

```
public void BulkInsert(IEnumerable<T> entities)
{
try
{
if (entities == null)
{
throw new ArgumentNullException("entities");
}
Context.Configuration.AutoDetectChangesEnabled = false;
Context.Set<T>().AddRange(entities);
Context.SaveChanges();
}
catch (DbEntityValidationException dbEx)
{
foreach (var validationErrors in dbEx.EntityValidationErrors)
{
foreach (var validationError in validationErrors.ValidationErrors)
{
_errorMessage += string.Format("Property: {0} Error: {1}", validationError.PropertyName,
validationError.ErrorMessage) + Environment.NewLine;
}
}
throw new Exception(_errorMessage, dbEx);
}
}
public virtual void Update(T entity)
{
try
{
if (entity == null)
throw new ArgumentNullException("entity");
if (Context == null || _isDisposed)
Context = new EmployeeDBContext();
SetEntryModified(entity);
```

```
//Context.SaveChanges(); commented out call to SaveChanges as Context save changes will be called
with Unit of work
}
catch (DbEntityValidationException dbEx)
{
foreach (var validationErrors in dbEx.EntityValidationErrors)
foreach (var validationError in validationErrors.ValidationErrors)
_errorMessage += Environment.NewLine + string.Format("Property: {0} Error: {1}",
validationError.PropertyName, validationError.ErrorMessage);
throw new Exception(_errorMessage, dbEx);
}
}
public virtual void Delete(T entity)
{
try
{
if (entity == null)
throw new ArgumentNullException("entity");
if (Context == null || _isDisposed)
Context = new EmployeeDBContext();
Entities.Remove(entity);
//Context.SaveChanges(); commented out call to SaveChanges as Context save changes will be called
with Unit of work
}
catch (DbEntityValidationException dbEx)
{
foreach (var validationErrors in dbEx.EntityValidationErrors)
foreach (var validationError in validationErrors.ValidationErrors)
_errorMessage += Environment.NewLine + string.Format("Property: {0} Error: {1}",
validationError.PropertyName, validationError.ErrorMessage);
throw new Exception(_errorMessage, dbEx);
}
}
public virtual void SetEntryModified(T entity)
{
```

```
Context.Entry(entity).State = EntityState.Modified;
}
}
}
```

**Modify the IEmployeeRepository as shown below**

```
using RepositoryUsingEFinMVC.DAL;
using RepositoryUsingEFinMVC.GenericRepository;
using System.Collections.Generic;
namespace RepositoryUsingEFinMVC.Repository
{
public interface IEmployeeRepository : IGenericRepository<Employee>
{
IEnumerable<Employee> GetEmployeesByGender(string Gender);
IEnumerable<Employee> GetEmployeesByDepartment(string Dept);
}
}
```

**Modify the EmployeeRepository as shown below**

```
using System.Linq;
using System.Collections.Generic;
using RepositoryUsingEFinMVC.DAL;
using RepositoryUsingEFinMVC.GenericRepository;
using RepositoryUsingEFinMVC.UnitOfWork;
namespace RepositoryUsingEFinMVC.Repository
{
public class EmployeeRepository : GenericRepository<Employee>, IEmployeeRepository
{
public EmployeeRepository(IUnitOfWork<EmployeeDBContext> unitOfWork)
: base(unitOfWork)
{
}
```

```
public EmployeeRepository(EmployeeDBContext context)
: base(context)
{
}
public IEnumerable<Employee> GetEmployeesByGender(string Gender)
{
return Context.Employees.Where(emp => emp.Gender == Gender).ToList();
}
public IEnumerable<Employee> GetEmployeesByDepartment(string Dept)
{
return Context.Employees.Where(emp => emp.Dept == Dept).ToList();
}
}
}
```

**Next Modify the Employee Controller as shown below.**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using RepositoryUsingEFinMVC.Repository;
using System.Web.Mvc;
using RepositoryUsingEFinMVC.DAL;
using RepositoryUsingEFinMVC.GenericRepository;
using RepositoryUsingEFinMVC.UnitOfWork;
namespace RepositoryUsingEFinMVC.Controllers
{
public class EmployeeController : Controller
{
private UnitOfWork<EmployeeDBContext> unitOfWork = new UnitOfWork<EmployeeDBContext>();
private GenericRepository<Employee> repository;
private IEmployeeRepository employeeRepository;
public EmployeeController()
```

```
{
//If you want to use Generic Repository with Unit of work
repository = new GenericRepository<Employee>(unitOfWork);
//If you want to use Specific Repository with Unit of work
employeeRepository = new EmployeeRepository(unitOfWork);
}
[HttpGet]
public ActionResult Index()
{
var model = repository.GetAll();
//Using Specific Repository
//var model = employeeRepository.GetEmployeesByDepartment(1);
return View(model);
}
[HttpGet]
public ActionResult AddEmployee()
{
return View();
}
[HttpPost]
public ActionResult AddEmployee(Employee model)
{
try
{
unitOfWork.CreateTransaction();
if (ModelState.IsValid)
{
repository.Insert(model);
unitOfWork.Save();
//Do Some Other Task with the Database
//If everything is working then commit the transaction else rollback the transaction
unitOfWork.Commit();
return RedirectToAction("Index", "Employee");
}
}
```

```
catch (Exception ex)
{
//Log the exception and rollback the transaction
unitOfWork.Rollback();
}
return View();
}
[HttpGet]
public ActionResult EditEmployee(int EmployeeId)
{
Employee model = repository.GetById(EmployeeId);
return View(model);
}
[HttpPost]
public ActionResult EditEmployee(Employee model)
{
if (ModelState.IsValid)
{
repository.Update(model);
unitOfWork.Save();
return RedirectToAction("Index", "Employee");
}
else
{
return View(model);
}
}
[HttpGet]
public ActionResult DeleteEmployee(int EmployeeId)
{
Employee model = repository.GetById(EmployeeId);
return View(model);
}
[HttpPost]
public ActionResult Delete(int EmployeeID)
```

```
{
Employee model = repository.GetById(EmployeeID);
repository.Delete(model);
unitOfWork.Save();
return RedirectToAction("Index", "Employee");
}
}
}
```

Now run the application and see everything is working as expected.

**SUMMARY**

In this article, I try to explain **Unit of Work using Repository Design Pattern** in ASP.NET MVC application using Entity Framework step by step with a simple example. I hope this article will help you with your need. I would like to have your feedback. Please post your feedback, question, or comments about this article

Factory Diagram Mp4 Download Operations Management Dictionary 3 and 4 Lesson Plan infolinks

1. APRON PATTERNS
2. BACKYARD DESIGN SOFTWARE
3. PROJECT TIMELINE TEMPLATE
4. NETWORK MANAGEMENT
5. FREE PROGRAMMING TUTORIALS
6. FREE GRAPHICS SOFTWARE
7. SQL CERTIFICATION TRAINING
8. DOWNLOAD ANDROID APPS
9. CLOUD COMPUTING SOLUTIONS
10. FREE PERSONALIZED APPS

← Previous Lesson
**Repository Pattern Implementation Guidelines in c#**

Next Lesson →
**Factory Design Pattern in C#**

## 2 thoughts on “Unit of Work using Repository Design Pattern”

**JAMSHED**
MARCH 31, 2019 AT 6:20 PM

Very Simple and Effective

Reply

**TEMIDAYO**
MAY 31, 2019 AT 9:01 AM

The Interface IDbSet is missing i guess. It cannot be built because of this

Reply

## Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name*

Email*

Website

Post Comment

**SINGLETON DESIGN PATTERN**

- Singleton Design Pattern in C#
- Why Singleton Class sealed in C#
- Thread-safe Singleton Design Pattern in C#
- Lazy Loading and Eager loading in Singleton Design Pattern
- Singleton VS Static class in C#
- Singleton Design Pattern Real Time Example in C#

**DEPENDENCY INJECTION DESIGN PATTERN**

- Dependency Injection in C#
- Property and Method Dependency Injection in C#
- Dependency Injection using Unity Container in MVC

---

**REPOSITORY DESIGN PATTERN**

- Repository Design Pattern in C#
- How to implement Repository Design Pattern in C#
- Generic Repository Pattern in C#
- Repository Pattern Implementation Guidelines in c#
- **Unit of Work using Repository Design Pattern**

---

**Factory Design Pattern**

- Factory Design Pattern in C#
- Factory Method Design Pattern in C#
- Abstract Factory Design Pattern in C#

---

**INVERSION OF CONTROL**

- Introduction to Inversion of Control
- Inversion of Control Using Factory Pattern in C#
- Inversion of Control Using Dependency Inversion Principle
- Inversion of Control Using Dependency Injection Design pattern
- Inversion of Control Containers in C#

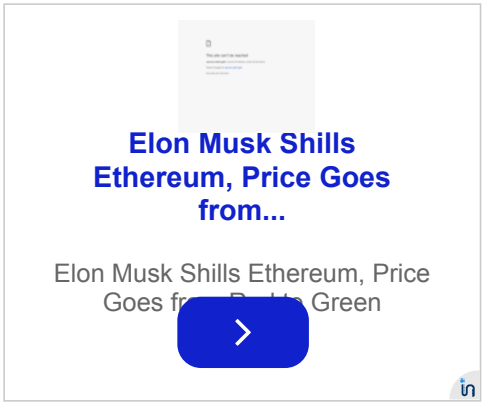
1 APRON PATTERNS

2 BACKYARD DESIGN SOFTWARE

3 PROJECT TIMELINE TEMPLATE

4 NETWORK MANAGEMENT

5 SQL CERTIFICATION TRAINING






Copyright © 2019 Dot Net Tutorials | Design by Sunrise Pixel