Unit of Work

A Unit of Work (UoW) keep tracking of all data change operations within one business transaction for database persistence. A business transaction could go through one or many requests, threads, functions and services. A completed business transaction must ensure all data change operations are either pass or fail as one. Any failed operation would cause all modified data unable to persist in the repository. The business transaction is an All-or-None operation.

A business transaction may contain non-business transaction operation (Monitoring, Security Auditing and Logging). The data change within those operations could be excluded from the UoW.

An UoW of the operations could join to another UoW to merge with other operations become a single UoW.

Sometimes, a parent UoW and child UoW may operate with in separated transaction or even without transaction boundary.

A Unit of Work implementation must provide mechanism to satisfy nested mergeable UoW, isolated UoW and non-transaction UoW,

The .NET framework [System.Transactions](https://docs.microsoft.com/en-us/dotnet/api/system.transactions?view=netframework-4.7.2) provide [TransactionScope](https://docs.microsoft.com/en-us/dotnet/api/system.transactions.transactionscope?view=netframework-4.7.2) to manage business transaction for database operation. It is an implementation of UoW. unfortunately, many database implementations do not support nested transactions rollback.

# UoW requirement:

1. Tracking data objects affected by a business transaction.

2. Coordinates data persistence component for writing out data change. Ensure UoW is All-or-None operation.

3. Persistence ignorance.

4.

# Type of Unit of Work

1. Nested Unit of work

A nested UoW is a new unit of work initiated inside another UoW. Changes made within a nested UoW could be merged with the parent UoW. Each nested UoW must perform commit or rollback. The data change operation performs in each UoW, but the transaction commit or rollback only occurred in topmost UoW when all nested UoWs completed. Any failed operation in nested UoW or parent UoW will cause entire UoW rollback.

**Nested Unit of Work**

B3

UoW Begin

UoW End

Repository 31

Repository 32

UoW provider 1 /DbContext1

UoW provider 1 /DbContext2

UoW provider 2 /DbContext3

UoW Begin

UoW End

Repository 11

Repository 12

Repository 21

Repository 22

B1

Repository 13

B2

Repository 41

Repository 42

A nested UoW always have a top-most UoW instance (B1) with one or many nested UoW instances (B3) and non-UoW operations (B2).

The operations within top-most UoW, nested UoW and non-UoW use same transaction as single unit of work.

A single UoW sample:

|  |
| --- |
| //Start UoW  using(var unitwork = new UnitOfWork())  {  //database operation  using(var db = new DbClient(\_testDb))  {  byte[] b = new byte[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };  db.Insert(table.TableName,  "v1\_int32,v2\_string, v3\_long, v4\_date, v5\_decimal,  v6\_Boolean, v7\_clob,v8\_guid",  99, "ABC", 123456789012345L,  new DateTime(2010, 1, 2, 3, 4, 5, 6),  12.2M, true, "11111", Guid.NewGuid());  }  //complete UoW  unitwork.Complete();  } |

Nested UoW sample:

|  |
| --- |
| //top-most UoW  using (var uow = new UnitOfWork())  {  //databse operation  using (var db = new DbClient(\_testDb))  {  db.Insert(table1.TableName,  "v1\_int32,v2\_string", 1, "A1");  }  //nested UoW  using (var uow2 = new UnitOfWork())  {  //database operation  using (var db = new DbClient(\_testDb))  {  db.Insert(table2.TableName,  "v1\_int32,v2\_string", 2, "A2");  }  //complete nested UoW  uow2.Complete();  }  //database operation  using (var db = new DbClient(\_testDb))  {  db.Insert(table3.TableName,  "v1\_int32,v2\_string", 3, "A3");  }  //complete UoW  uow.Complete();  } |

If a .NET TransactionScope UoW being used, make sure the code throw exception when operation failed. The reason to throw exception is to prevent other data persistent operation continue after a transaction rollback performed. This is a trick of TransactionScope in nested UoW.

|  |
| --- |
| public void UoWNestedUoWWithErrorTest()  {  using (var uow = new TransactionScope())  {  using (var db = new DbClient(myDB))  {  db.Sql("insert into T1 (v1) values (1)");  }  using (var uow2 = new TransactionScope())  {  if (!AddCustom(myDB))  {  //Comment out below exception raise statement will cause subsequence DB operation unexpected exception  //throw new System.Exception("Throw exception to avoid transaction continue.");  }  else  {  uow2.Complete();  }  }  //Below operation will cause unexpected error if above failed AddCustom() didn't throw an exception.  using (var db = new DbClient(myDB))  {  db.Sql("insert into T2 (v1) values (1)");  }  uow.Complete();  }  } |

2. Isolated Unit of work

An isolated UoW is a new unit of work initiated inside another UoW. Changes made within isolated UoW will be persisted independent from parent UoW. The transaction commit or rollback in isolated UoW will not affect parent UoW transaction.

**Isolated Unit of Work**

UoW provider 1 /DbContext1

UoW provider 1 /DbContext2

UoW provider 2 /DbContext3

B3

UoW Begin

UoW End

Repository 31

Repository 32

UoW Begin

UoW End

Repository 11

Repository 12

Repository 21

Repository 22

B1

Repository 13

requireNew = true

Sample of Isolated Unit of Work.

|  |
| --- |
| //parent UoW contains isolated UoW  using (var parentUoW = new UnitOfWork())  {  //database operation  using (var db = new DbClient(\_testDb))  {  db.Insert(table1.TableName,  "v1\_int32,v2\_string", 1, "A1");  }  //isolated UoW  using (var uow2 = new UnitOfWork(UnitOfWorkOption.RequiresNew))  {  //isolated database operation.  //Failed operation will not affect parent transaction.  using (var db = new DbClient(\_testDb))  {  db.Insert(table2.TableName,  "v1\_int32,v2\_string", 2, "A2");  }  uow2.Complete();  }  //database operation  using (var db = new DbClient(\_testDb))  {  db.Insert(table3.TableName,  "v1\_int32,v2\_string", 3, "A3");  }  parentUoW.Complete();  } |

3. Parallel Unit of Work

Multiple UoWs executed in parallel (in different call context) and each UoW (B1 and B2) maintains its own state change. The repository operations without UoW (B3) will not join to any other UoW.

The parallel UoW is running in different threads without parent UoW.

UoW provider 1 /DbContext1

UoW provider 1 /DbContext2

UoW provider 2 /DbContext3

Main block

B3

Repository 41

Repository 42

UoW Begin

UoW End

Repository 11

Repository 12

Repository 21

Repository 22

B1

Repository 13

B2

UoW Begin

UoW End

Repository 31

Repository 32

A sample of parallel UoWs

|  |
| --- |
| public async Task UoW\_Parallel\_Async()  {  var task1 = AddDataAsyncUoW(table1.TableName, 11, "B1");  var task2 = AddDataAsyncUoW(table2.TableName, 11, "B2");  var task3 = AddDataAsync(table3.TableName, 11, "B3");  await Task.WhenAll(task1, task2, task3);  }  static async Task AddDataAsyncUoW(string tableName, int id, string value)  {  await Task.Run(() => AddDataUoW(tableName, id, value));  }  static void AddDataUoW(string tableName, int id, string value)  {  using (var uow = new UnitOfWork())  {  using (var db = new DbClient(\_testDb))  {  db.Insert(tableName,  "v1\_int32,v2\_string", id, value);  }  uow.Complete();  }  }  static async Task AddDataAsync(string tableName, int id, string value)  {  await Task.Run(() => AddData (tableName, id, value));  }  static void AddData(string tableName, int id, string value)  {  using (var db = new DbClient(\_testDb))  {  db.Insert(tableName, "v1\_int32,v2\_string", id, value);  }  } |

4. Nested non-transaction UoW

In most cases an UoW contains data persistence operations under a business transaction. But in case of certain operations which do not want to be a part of transaction it should use non-transaction UoW to isolate those operations.

|  |
| --- |
| using (var uow = new UnitOfWork())  {  //Transaction operations  using (var db = new DbClient(\_testDb))  {  db.Insert(table1.TableName,  "v1\_int32,v2\_string",  1, "A1");  isSqlite = db.Session.Database.ProviderName.Contains("SQLite");  }  //Non-transaction operations  using (var uow2 = new UnitOfWork(UnitOfWorkOption.Suppress))  {  using (var db = new DbClient(\_testDb))  {  db.Insert(table2.TableName, "v1\_int32,v2\_string", 2, "A2");  }  //Not necessary  uow2.Complete();  }  //Request to complete transaction operations  uow.Complete();  } |

5. UoW per request

A simple UoW created for each request (Web request, Service request or message request) and completed after request completed.

a. Create an UoW onRequestBegin and bind UoW to request context

b. Apply UoW to persistance component

c. Complete the UoW onRequestEnd and unbind UoW from request context

requireNew = false

UoW provider 1 /DbContext1

UoW provider 1 /DbContext2

UoW provider 2 /DbContext3

UoW End

Repository 11

Repository 12

Repository 21

Repository 22

Repository 13

Bx1

UoW

UoW End

Repository 31

Bx2

UoW

UoW End

Repository 31

UoW Begin

requireNew = true

# UoW, Interface, Provider and Persistence Ignorance:

Unit of Work is a Persistence Ignorance component, but underline UoW implementation always be persistence specific. To decouple application UoW with actual persistence implementation mechanism IoC is required to wire up the implementation to the UoW interface.

The IUnitOfWork "Complete()" indicates a business transaction need be committed. The transaction will be rollback if "Complete()" is not presented in the scope of UoW.

|  |
| --- |
| /// <summary>  /// Interface of Unit of work.  /// Unit of work maintains a list of objects affected by a business transaction.  /// The unit of work may contains many other unit of works, all child unit of work will be merged into one transaction.  /// Any uncompleted unit of work will cause whole transaction rollback before persistent the data.  /// </summary>  public interface IUnitOfWork:IDisposable  {  /// <summary>  /// Complete the unit of work.  /// Completed unit of work must be done before end of the scope.  /// </summary>  void Complete();  } |

The IUnitOfWork is a disposable interface. Dispose() may cause the transaction commit or rollback.

An application may have more than one IUnitOfWork implementation components for different data persistence mechanism. A type of IUnitOfWork instance is created by unit of work provider which must implement IUnitOfWorkProvider interface.

|  |
| --- |
| public interface IUnitOfWorkProvider:IProvider  {  /// <summary>  /// Create a given type scope of unit of work.  /// </summary>  /// <returns>Create a new unit of work instance</returns>  /// <param name="option">A transaction scope option for the unit of work.  /// The concept of transaction scope option is similar as TransactionScopeOption.  /// See reference https://github.com/dotnet/docs/blob/master/docs/framework/data/transactions/implementing-an-implicit-transaction-using-transaction-scope.md  /// Required: Join the ambient transaction, or create a new one if one does not exist.  /// RequiresNew: Be a new root scope, that is, start a new unit of work transaction and have that transaction be the new ambient transaction inside its own scope.  /// Suppress: New connection but not take part in a transaction at all. There is no ambient transaction as a result.  IUnitOfWork Create(UnitOfWorkOption option);  } |

Some sample UoW implementation:

DBClient Unit of Work implementation

|  |
| --- |
| /// <summary>  /// Manages each database independantly. The trasactions under same database can be merged into one trasaction as one unit of work.  /// For multiple databases operations, a unit of work maintains all database transactions in same UoW, one transaction per database.  /// Any uncompleted operation could cause all transactions rollback in a unit of work.  /// Note: multi-databases operations within single UoW may not work properly if an exception throw in UoW Dispose() moment.  ///  /// The Unit of Work may not work for cross processor transaction and DTC (distributed transaction coordinator).  /// Use TrasactionScopeUnitOfWork() for DTC if it is required.  ///  /// The UoW transaction is logical context based ambiente translation. It works in multi-threads environment.  ///  /// </summary>  public class DbUnitOfWork:IUnitOfWork  {  /// <summary>  /// Initializes a new instance of the <see cref="T:qshine.DbUnitOfWork"/> class.  /// Join the ambient transaction, or create a new one if one does not exist.  /// </summary>  public DbUnitOfWork();  /// <summary>  /// Initializes a new instance of the <see cref="T:qshine.DbUnitOfWork"/> class .  /// Create a new scope of unit of work based on selected scope option.  /// </summary>  /// <param name="scopeOption">Scope of the unit of work</param>  public DbUnitOfWork(UnitOfWorkOption scopeOption);  /// <summary>  /// Complete the unit of work and commit the transaction.  /// </summary>  public void Complete();  public void Dispose();  protected virtual void Dispose(bool disposing);  } |

NHibernate UoW implementation

public class NHibernateUnitOfWork:IUnitOfWork, IUnitOfWorkProvider

{

public NHibernateUnitOfWork();

public NHibernateUnitOfWork(UnitOfWorkOption scopeOption);

public void Complete();

public void Dispose();

public IUnitOfWork Create(UnitOfWorkOption scopeOption)

{

return new NHibernateUnitOfWork(scopeOption);

}

}

Entity Framework UoW implementation

public class EFUnitOfWork:IUnitOfWork, IUnitOfWorkProvider

{

public EFUnitOfWork();

public EFUnitOfWork(UnitOfWorkOption scopeOption);

public void Complete();

public void Dispose();

public IUnitOfWork Create(UnitOfWorkOption scopeOption)

{

return new EFUnitOfWork(scopeOption);

}

}

TransactionScope UoW implementation

public class TransactionScopeUnitOfWork:IUnitOfWork, IUnitOfWorkProvider

{

public TransactionScopeUnitOfWork ();

public TransactionScopeUnitOfWork (UnitOfWorkOption scopeOption);

public void Complete();

public void Dispose();

public IUnitOfWork Create(UnitOfWorkOption scopeOption)

{

return new TransactionScopeUnitOfWork (scopeOption);

}

}

To manage all different UoW aware operations in one UoW an UnitOfWork manager class implemented to coordinate and unify all type of UoW instances created from UoW providers.

|  |
| --- |
| /// <summary>  /// Manages Unit of Work for all UnitOfWork providers.  /// Each unit of work provider can create UnitOfWork instance for transaction management.  /// </summary>  public class UnitOfWork:IDisposable  {  /// <summary>  /// Gets or sets a list of UoW providers.  /// Single UoW may contain many UoW transaction implementations for differnt database, transaction process.  /// Each UoW transaction implementation different UoW provider.  /// The UoW can be injected by DI or configuration.  /// </summary>  /// <value>The provider.</value>  public static IList<IUnitOfWorkProvider> Providers { get;set;}  /// <summary>  /// Create a UoW instance  /// </summary>  /// <param name="option"></param>  public UnitOfWork(UnitOfWorkOption option = UnitOfWorkOption.Required);  /// <summary>  /// Dispose UoW  /// </summary>  public void Dispose();  /// <summary>  /// Indicates all operations within scope are completed sucessfully.  /// </summary>  public void Complete();  } |

The UnitOfWork instance will wire up all UoW implementation components through IoC in scope of UoW.

(Note: If the application UoW contains more than one UoW instances with different UoW provider. The UoW single business transaction may not be guaranteed if the failure occurred in one of UoW Dispose() time.

)

# Multiple UoW Providers and Persistence transactions

UoW provider is a unit of work implementation for a specific type of persistence store (such as ADO.NET database, EF DbContext, NHibernate session or .NET TransactionScope). An UoW could contain many UoW instances created through different UoW provider. Same provider could mange multiple persistence stores (databases) in different transaction.

The UoW guarantee all persistence operations (could be in different persistence implementation) under single business transaction (not persistence transaction). Any persistence operation failure cause entire UoW rollback.

There is one exception that UoW could not guarantee all persistence transactions are committed in single UoW if one of the transaction failures occurred in UoW instance Dispose() moment.

**Unit of Work with multiple providers**

B1

UoW Begin

Repository 11

B2

Repository 41

Repository 42

Repository 12

X

Complete()

Committed

Committed(?)

Committed

UoW End

Repository 13

Repository 22

Repository 21

UoW provider 2 /DbContext3

UoW provider 1 /DbContext2

UoW provider 1 /DbContext1

B3

UoW Begin

UoW End

Repository 31

Repository 32

# UoW and Multi-Threads and Call Context

UoW perform a single persistence transaction in the call context (ambient transactions in logic call context). The call context is the path of the code calling from begin of the unit of work to end of the unit of work. If the UoW call context go through multi-threads the repository and uow under child thread will be automatically merged into parent uow. It is very useful when the main operation contains many async await task operations. Those async await operations will be merged into parent UoW.

But in other handle, some operations may not want to be part of business transaction. The operation could be a separated transaction or not be part of any transaction.

To differentiate each UoW block transaction participation a UnitOfWorkOption parameter is added in UnitOfWork constructor.

|  |
| --- |
| /// <summary>  /// Unit of Work options,  /// The concept of the option is similar as System.Transactions.TransactionScopeOption.  /// </summary>  public enum UnitOfWorkOption  {  //  // Summary:  // A transaction is required by the UoW. It uses an ambient transaction if one  // already exists. Otherwise, it creates a new transaction before entering the scope.  // This is the default value.  Required = 0,  //  // Summary:  // A new transaction is always created for the UoW.  RequiresNew = 1,  //  // Summary:  // The ambient transaction context is suppressed when creating the UoW. All operations  // within the UoW are done without an ambient transaction context.  Suppress = 2  } |