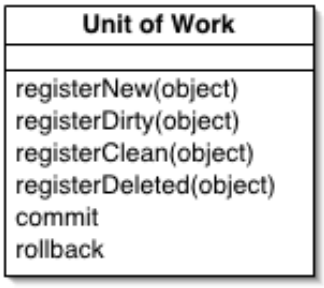
Unit of Work

Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.



When you're pulling data in and out of a database, it's important to keep track of what you've changed; otherwise, that data won't be written back into the database. Similarly you have to insert new objects you create and remove any objects you delete.

You can change the database with each change to your object model, but this can lead to lots of very small database calls, which ends up being very slow. Furthermore it requires you to have a transaction open for the whole interaction, which is impractical if you have a business transaction that spans multiple requests. The situation is even worse if you need to keep track of the objects you've read so you can avoid inconsistent reads.

A Unit of Work keeps track of everything you do during a business transaction that can affect the database. When you're done, it figures out everything that needs to be done to alter the database as a result of your work.

Unit of Work Design Pattern

By **Shivprasad koirala**, 6 May 2013

What is the use of Unit of Work design pattern?

Unit of Work design pattern does two important things: first it maintains in-memory updates and second it sends these in-memory updates as one transaction to the database.

So to achieve the above goals it goes through two steps:

- It maintains lists of business objects in-memory which have been changed (inserted, updated, or deleted) during a transaction.
- Once the transaction is completed, all these updates are sent as one **big unit of work** to be persisted physically in a database in one **go**.

What is "Work" and "Unit" in a software application?

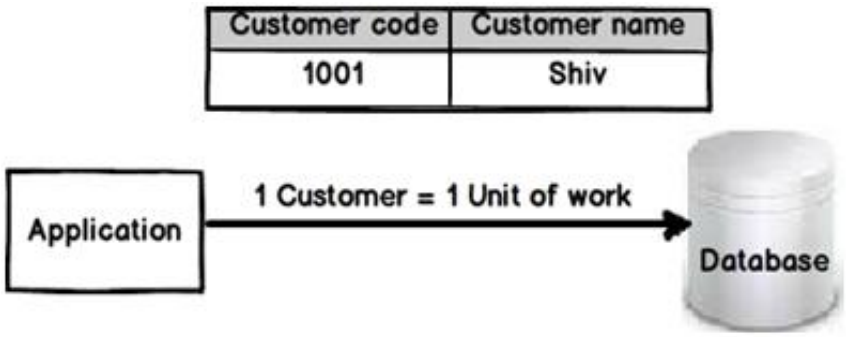
A simple definition of Work means ***performing some task***. From a software application perspective **Work** is nothing but inserting, updating, and deleting data. For instance let's say you have an application which maintains customer data into a database.

So when you add, update, or delete a customer record on the database it's one unit. In simple words the equation is.

Collapse | Copy Code

```
1 customer CRUD = 1 unit of work
```

Where CRUD stands for create, read, update, and delete operation on a single customer record.



Logical transaction! = Physical CRUD

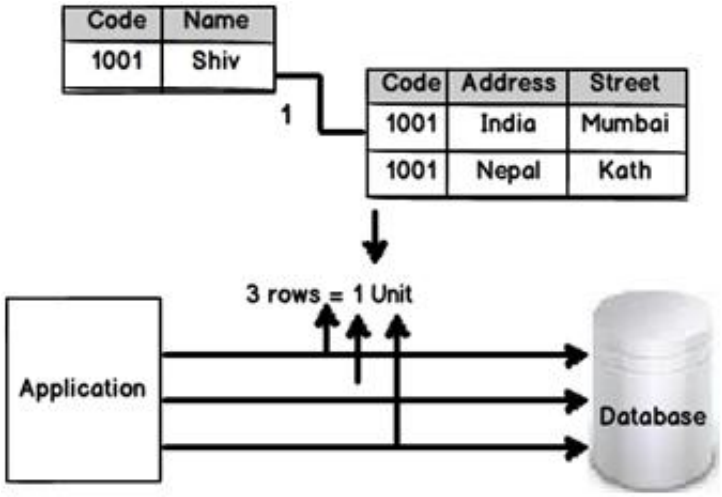
The equation which we discussed in the previous section changes a lot when it comes to real world

scenarios. Now consider the below scenario where every customer can have multiple addresses. Then many rows will become 1 unit of work.

For example you can see in the below figure customer "Shiv" has two addresses, so for the below scenario the equation is:

Collapse | Copy Code

```
3 Customer CRUD = 1 Logical unit of work
```



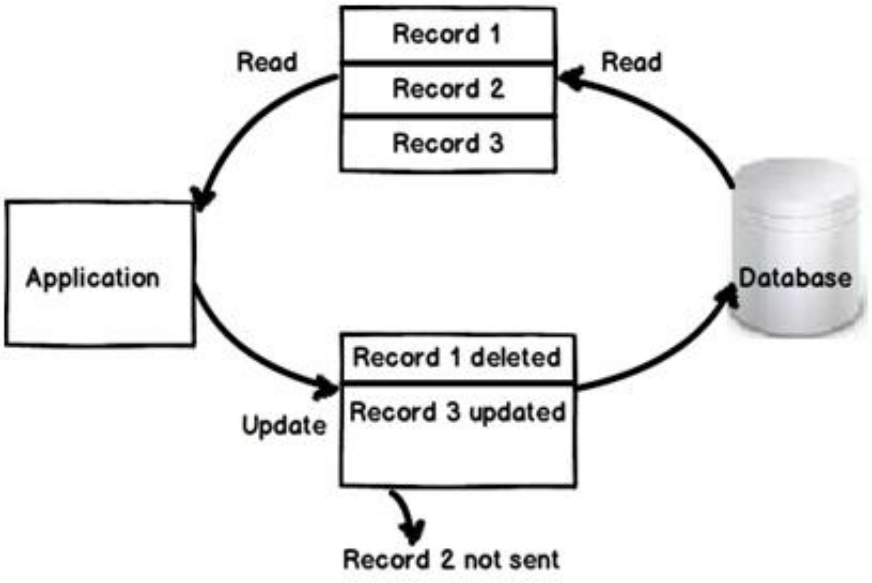
So in simple words, transactions for all of the three records should succeed or all of them should fail. It should be **ATOMIC**. In other words it's very much possible that many CRUD operations will be equal to 1 unit of work.

So this can be achieved by using simple transactions?

Many developers can conclude that the above requirement can be met by initiating all the CRUD operations in one transaction. Definitely under the cover it uses database transactions (i.e., `TransactionScope` object). But unit of work is much more than simple database transactions, it sends only changes and not all rows to the database.

Let me explain to you the same in more detail.

Let's say your application retrieves three records from the database. But it modifies only two records as shown in the below image. So only modified records are sent to the database and not all records. This optimizes the physical database trips and thus increases performance.



In simple words the final equation of unit of work is:

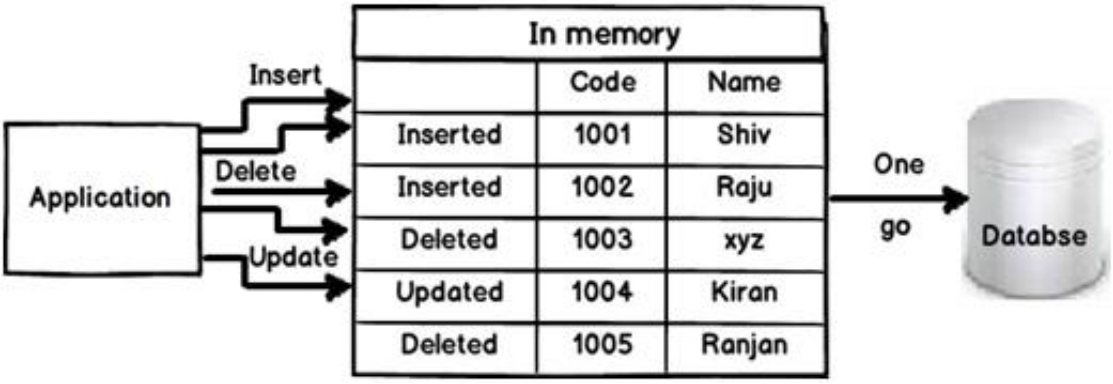
Collapse | Copy Code

```
1 Unit of work = Modified records in a transaction
```

So how can we achieve this?

To achieve the same the first thing we need is an in-memory collection. In this collection, we will add all the business objects. Now as the transaction happens in the application they will be tracked in this in-memory collection.

Once the application has completed everything it will send these changed business objects to the database in "one transaction". In other words either all of them will commit or all of them will fail.



Entity Framework Unit of Work and repositories

By Geert 27. February 2013 20:11

Catel is an application development platform for C# that was primarily focused on xaml languages (WPF, Silverlight, Windows Phone). The core however is usable by server implementations as well and more and more extensions are being developed. This blog post will give an introduction to the Unit of Work and the repositories that are available in Catel.

Overview of Unit of Work and repositories

The Repository and Unit of Work (UoW) pattern are very useful patterns to create an abstraction level over the DbContext that is provided by Entity Framework. A much heard excuse not to use repositories is that EF itself already works with repositories (the DbContext) and a UoW (in the SaveChanges method). Below are a few examples why it is a good thing to create repositories:

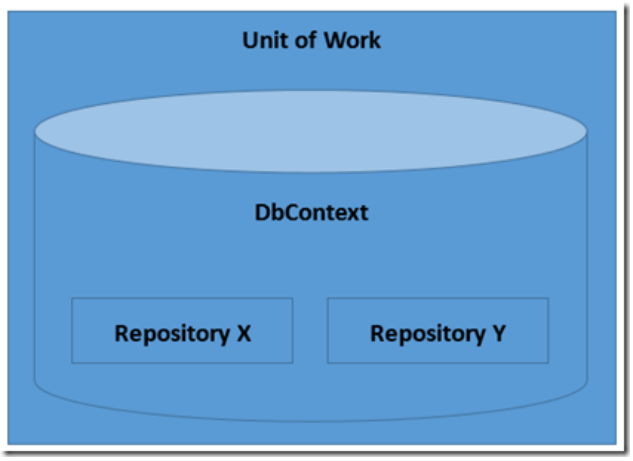
- Abstract away some of the more complex features of Entity Framework that the end-developer should not be bothered with
- Hide the actual DbContext (make it internal) to prevent misuse
- Keep security checks and saving and rollback in a single location
- Force the use of the Specification pattern on queries

A Unit of Work (UoW) is a a combination of several actions that will be grouped into a transaction. This means that either all actions inside a UoW are committed or rolled back. The advantage of using a UoW is that multiple save actions to multiple repositories can be grouped as a unit.

A repository is a class or service responsible for providing objects and allowing end-developers to query data. Instead of querying the DbContext directly, the DbContext can be abstracted away to provide default queries and force required functionality to all end-developers of the DbContext.

A Unit of Work (UoW) is a a combination of several actions that will be grouped into a transaction. This means that either all actions inside a UoW are committed or rolled back. The advantage of using a UoW is that multiple save actions to multiple repositories can be grouped as a unit.

A repository is a class or service responsible for providing objects and allowing end-developers to query data. Instead of querying the DbContext directly, the DbContext can be abstracted away to provide default queries and force required functionality to all end-developers of the DbContext.



The image above shows that the Unit of Work is the top-level component to be used. Each UoW contains its own DbContext instance. The DbContext can either be injected or will be created on the fly. Then the UoW also contains repositories which always get the DbContext injected. This way, all repositories inside a UoW share the same DbContext.

The DbContextManager

The DbContextManager class allows the sharing of DbContext (with underlying ObjectContext) classes in Entity Framework 5. The good thing about this is that the same context can be used in the same scope without having to recreate the same type of the same context over and over again.

```
1: using (var dbContextManager = DbContextManager<MyEntities>.GetManager())
2: {
3:     var dbContext = dbContextManager.DbContext;
4:
5:     // TODO: handle logic with dbContext here
6: }
```

Creating a Unit of Work

A UoW can be created by simply instantiating it. The end-developer has the option to either inject the DbContext or let the DbContextManager take care of it automatically.

```
1: using (var uow = new UnitOfWork<MyDbContext>())
2: {
```

```
    // get repositories and query away
}
```

Creating a repository

A repository can be created very easily by deriving from the EntityRepositoryBase class. Below is an example of a customer repository:

```
public class CustomerRepository : EntityRepositoryBase<Customer, int>, ICustomerRepository
{
    public CustomerRepository(DbContext dbContext)
        : base(dbContext)
    {
    }
}

public interface ICustomerRepository : IEntityRepository<Customer, int>
{
}
```

Retrieving repositories from a Unit of Work

Once a UoW is created, it can be used to resolve repositories. To retrieve a repository from the UoW, the following conditions must be met:

1. The container must be registered in the *ServiceLocator* as *Transient* type. If the repository is declared as non-transient, it will be instantiated as new instance anyway.
2. The repository must have a constructor accepting a *DbContext* instance

To retrieve a new repository from the UoW, use the following code:

```
using (var uow = new UnitOfWork<MyDbContext>())
{
    var customerRepository = uow.GetRepository<ICustomerRepository>();

    // all interaction with the customer repository is applied to the unit of work
}
```

Saving a Unit of Work

It is very important to save a Unit of Work. Once the Unit of Work gets out of scope (outside the using), all changes will be discarded if not explicitly saved.

```
using (var uow = new UnitOfWork<MyDbContext>())
{
    var customerRepository = uow.GetRepository<ICustomerRepository>();

    // all interaction with the customer repository is applied to the unit of work

    uow.SaveChanges();
}
```