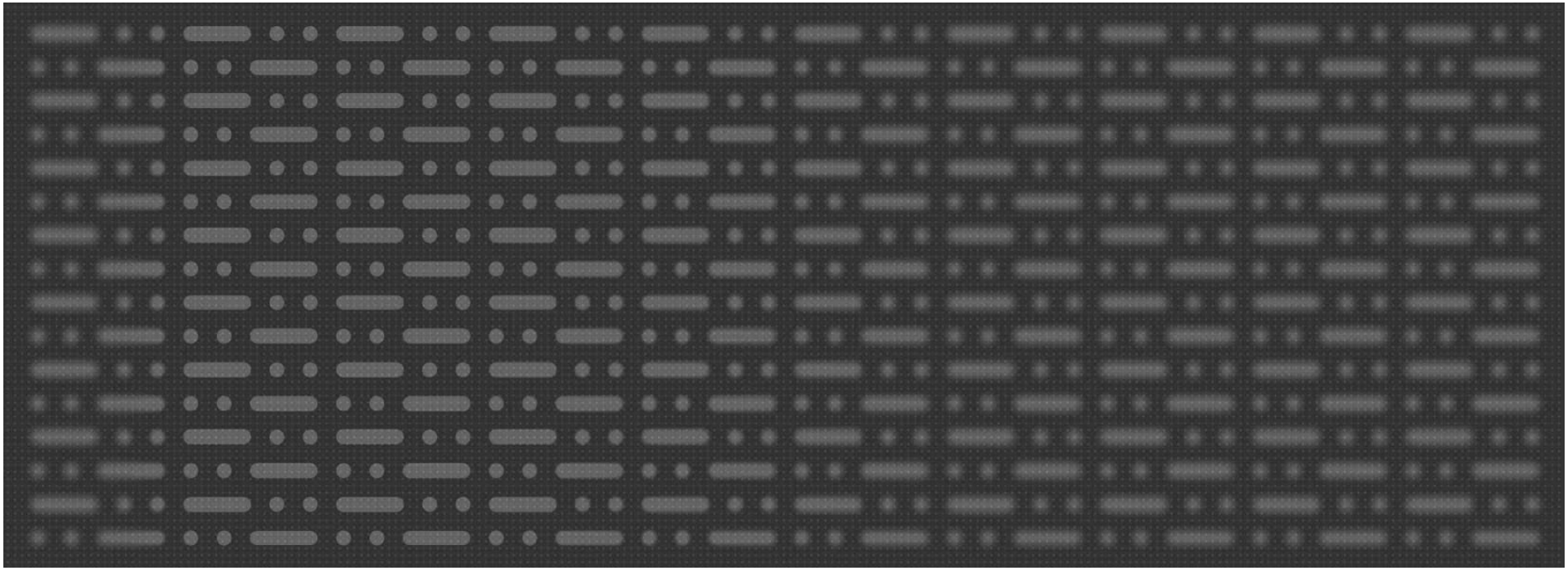
mono

Unit of Work - a Design Pattern

Author
Ante Ljubic

Date
Friday, Jan 13, 2017



mono

different situations. Unit of work is one of them - and in this post, we'll explain how to use it and how to implement it in ASP.NET using Entity Framework.

When to use „Unit of Work“?

Imagine you have users and roles, and it is important to ensure that all users have roles assigned to them. This is where Unit of Work comes in. It helps us save everything, user and his roles - or nothing, if something fails. Unit of Work is mostly used when we want to get a set of actions happen, and if one action fails, to cancel all of them.

In general, Unit of Work has two important tasks:

- to keep a list of requests in one place (the list can contain multiple insert, update, and delete requests), and
- to send that requests as one transaction to the database.

An example implementation based on Entity Framework

For the `DbContext` variable, a new context is instantiated in the following way:

```
protected IMyDbContext DbContext { get; private set; }
```

and this is a constructor that uses the `DbContext` variable:

```
        {
            throw new ArgumentNullException("DbContext");
        }
        DbContext = dbContext;
    }
```

Items will be added using this method:

```
    public Task<int> AddAsync<T>(T entity) where T : class
    {
        DbEntityEntry dbEntityEntry = DbContext.Entry(entity);
        if (dbEntityEntry.State != EntityState.Detached)
        {
            dbEntityEntry.State = EntityState.Added;
        }
        else
        {
            DbContext.Set<T>().Add(entity);
        }
        return Task.FromResult(1);
    }
```

If we want to edit (update) items, we need the update method:

```
    public Task<int> UpdateAsync<T>(T entity) where T : class
    {
        DbEntityEntry dbEntityEntry = DbContext.Entry(entity);
```

```
    dbEntityEntry.State = EntityState.Modified;

    return Task.FromResult(1);
}
```

To delete item(s) we will use following:

```
public Task<int> DeleteAsync<T>(T entity) where T : class
{
    DbEntityEntry dbEntityEntry = DbContext.Entry(entity);
    if (dbEntityEntry.State != EntityState.Deleted)
    {
        dbEntityEntry.State = EntityState.Deleted;
    }
    else
    {
        DbContext.Set<T>().Attach(entity);
        DbContext.Set<T>().Remove(entity);
    }
    return Task.FromResult(1);
}

public Task<int> DeleteAsync<T>(string ID) where T : class
{
    var entity = DbContext.Set<T>().Find(ID);
    if (entity == null)
    {
        return Task.FromResult(0);
    }
    return DeleteAsync<T>(entity);
}
```

mono

Besides the basic create, update, and delete methods described here, to commit the transaction, we will need a `CommitAsync` method:

```
public async Task<int> CommitAsync()
{
    int result = 0;
    using (TransactionScope scope = new TransactionScope(TransactionScopeAsyncFlowOption.Enabled))
    {
        result = await DbContext.SaveChangesAsync();
        scope.Complete();
    }
    return result;
}
```

Finally, the method to dispose the whole transaction if any of the actions fail:

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

For all that methods we usually create the interface like this:

```
    Task<int> CommitAsync();

    Task<int> DeleteAsync<T>(T entity) where T : class;

    Task<int> DeleteAsync<T>(string ID) where T : class;

    Task<int> UpdateAsync<T>(T entity) where T : class;
}
```

And here is the Unit of Work factory that creates it:

```
public interface IUnitOfWorkFactory
{
    IUnitOfWork CreateUnitOfWork();
}
```

Note: I used Ninject Factory extension here, so I didn't have to implement it.

How to use it

All you need to do now is to create the Unit of Work in the method where you need it:

```
public class MyClass
{
```

```
        this.uowFactory = uowFactory;
    }

    public async Task<int> MyMethod()
    {
        // Creates the Unit of Work
        var unitOfWork = uowFactory.CreateUnitOfWork();
    }
}
```

Note: `MyMethod` should be async because of `CommitAsync`, `AddAsync` and other async methods.

Next, add requests to `unitOfWork` in `MyMethod` in order in which you want them to perform. For example, to add user and then add his roles:

```
// Insert user into user table
var userId = await unitOfWork.AddAsync(user);

// Multiple inserts for user roles
foreach (var userRole in userRoles)
{
    // Assign UserId before the insert
    userRole.UserId = userId.Value;

    // Insert into user role table
    await unitOfWork.AddAsync(userRole);
}
```

```
// Commit
await unitOfWork.CommitAsync();
```

All requests added to the Unit of Work will be sent to the database when you call `CommitAsync`. Since the Unit of Work is handling them all in the `TransactionScope`, it will assure that no unfinished work will make changes to our data - either all actions will be successful, or no changes will be done at all.

MORE ARTICLES

ARTICLE PUBLISHED IN

unit-of-work design-pattern entity-framework

YOUR OPINION MATTERS!

mono

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

ALSO ON MONO SOFTWARE

Model binding in ASP.NET MVC

1 comment • 2 years ago

Diwas Poudel — Thank you very much

Multiple BMS Monitor

1 comment • 6 months ago

Yuriy Martini — Very useful. Thanks a lot!

Aerial image stitching

3 comments • a year ago

OmgMan — Good day Dino, I am working on a very similar project for my undergraduate degree. I am in the early stages of my project and do not have …

Creating NodeJS modules with both promise and callback API support using Q

1 comment • 2 years ago

Nasser — Well, sounds good

Subscribe Add Disqus to your siteAdd DisqusAdd Disqus' Privacy PolicyPrivacy PolicyPrivacy

Blog

Showcase

PRODUCTS

Baasic

eCTD Office

Clokke

TECHNOLOGIES

.NET

Python

JavaScript

Mobile

Cloud

CAREERS

Software developer

UI and UX designer

WORK WITH US

GET A QUOTE