

 Filter topics

> [Getting Started](#)

> [Startup Templates](#)

> [Tutorials](#)

> [Fundamentals](#)

> [Infrastructure](#)

> [Architecture](#)

> [Modularity](#)

> [Domain Driven Design](#)

→ [Overall](#)

> [Domain Layer](#)

> [Application Layer](#)

→ [Application Services](#)

→ [Data Transfer Objects](#)

→ [Unit Of Work](#)

→ [Guide: Implementing DDD](#)

→ [Multi Tenancy](#)

→ [Microservices](#)

> [API](#)

> [User Interface](#)

> [Data Access](#)

> [Real Time](#)

→ [Testing](#)

> [Samples](#)

> [Application Modules](#)

> [Release Information](#)

> [Reference](#)

→ [Contribution Guide](#)

In this document

Unit of Work

ABP Framework's Unit Of Work (UOW) implementation provides an abstraction and control on a **database connection and transaction** scope in an application.

Once a new UOW started, it creates an **ambient scope** that is participated by **all the database operations** performed in the current scope and considered as a **single transaction boundary**. The operations are **committed** (on success) or **rolled back** (on exception) all together.

ABP's UOW system is;

- **Works conventional**, so most of the times you don't deal with UOW at all.
- **Database provider independent**.
- **Web independent**, that means you can create unit of work scopes in any type of applications beside web applications/services.

Conventions

The following method types are considered as a unit of work:

- ASP.NET Core MVC **Controller Actions**.
- ASP.NET Core Razor **Page Handlers**.
- **Application service** methods.
- **Repository methods**.

A UOW automatically begins for these methods **except** if there is already a **surrounding (ambient)** UOW in action. Examples;

- If you call a [repository](#) method and there is no UOW started yet, it automatically **begins a new transactional UOW** that involves all the operations done in the repository method and **commits the transaction** if the repository method **doesn't throw any exception**. The repository method doesn't know about UOW or transaction at all. It just works on a regular database objects (`DbContext` for [EF Core](#), for example) and the UOW is handled by the ABP Framework.
- If you call an [application service](#) method, the same UOW system works just as explained above. If the application service method uses some repositories, the repositories **don't begin a new UOW**, but **participates to the current unit of work** started by the ABP Framework for the application service method.
- The same is true for an ASP.NET Core controller action. If the operation has started with a controller action, then the **UOW scope is the controller action's method body**.

All of these are automatically handled by the ABP Framework.

Database Transaction Behavior

While the section above explains the UOW as it is database transaction, actually a UOW doesn't have to be transactional. By default;

- **HTTP GET** requests don't start a transactional UOW. They still starts a UOW, but **doesn't create a database transaction**.
- All other HTTP request types start a UOW with a database transaction, if database level transactions are supported by the

> [Getting Started](#)

> [Startup Templates](#)

> [Tutorials](#)

> [Fundamentals](#)

> [Infrastructure](#)

> [Architecture](#)

> [Modularity](#)

> [Domain Driven Design](#)

> [Overall](#)

> [Domain Layer](#)

> [Application Layer](#)

> [Application Services](#)

> [Data Transfer Objects](#)

> [Unit Of Work](#)

> [Guide: Implementing DDD](#)

> [Multi Tenancy](#)

> [Microservices](#)

> [API](#)

> [User Interface](#)

> [Data Access](#)

> [Real Time](#)

> [Testing](#)

> [Samples](#)

> [Application Modules](#)

> [Release Information](#)

> [Reference](#)

> [Contribution Guide](#)

underlying database provider.

This is because an HTTP GET request doesn't (and shouldn't) make any change in the database. You can change this behavior using the options explained below.

Default Options

`AbpUnitOfWorkDefaultOptions` is used to configure the default options for the unit of work system. Configure the options in the `ConfigureServices` method of your [module](#).

Example: Completely disable the database transactions

```
Configure<AbpUnitOfWorkDefaultOptions>(options =>
{
    options.TransactionBehavior = UnitOfWorkTransactionBehavior.NoTransaction;
});
```

Option Properties

- `TransactionBehavior` (`enum : UnitOfWorkTransactionBehavior`). A global point to configure the transaction behavior. Default value is `Auto` and work as explained in the "*Database Transaction Behavior*" section above. You can enable (even for HTTP GET requests) or disable transactions with this option.
- `Timeout` (`int?`): Used to set the timeout value for UOWs. **Default value is `null`** and uses to the default of the underlying database provider.
- `IsolationLevel` (`IsolationLevel?`): Used to set the [isolation level](#) of the database transaction, if the UOW is transactional.

Controlling the Unit Of Work

In some cases, you may want to change the conventional transaction scope, create inner scopes or fine control the transaction behavior. The following sections cover these possibilities.

IUnitOfWorkEnabled Interface

This is an easy way to enable UOW for a class (or a hierarchy of classes) that is not unit of work by the conventions explained above.

Example: Implement `IUnitOfWorkEnabled` for an arbitrary service

In this document

 Filter topics

> **Getting Started**

> **Startup Templates**

> **Tutorials**

> **Fundamentals**

> **Infrastructure**

> **Architecture**

> [Modularity](#)

> Domain Driven Design

→ [Overall](#)

> [Domain Layer](#)

> Application Layer

→ [Application Services](#)

→ [Data Transfer Objects](#)

→ [Unit Of Work](#)

→ [Guide: Implementing DDD](#)

→ [Multi Tenancy](#)

→ [Microservices](#)

> **API**

> **User Interface**

> **Data Access**

> **Real Time**

→ **Testing**

> **Samples**

> **Application Modules**

> **Release Information**

> **Reference**

→ **Contribution Guide**

```
using System.Threading.Tasks;
using Volo.Abp.DependencyInjection;
using Volo.Abp.Uow;

namespace AbpDemo
{
    public class MyService : ITransientDependency, IUnitOfWorkScoped
    {
        public virtual async Task FooAsync()
        {
            //this is a method with a UOW scope
        }
    }
}
```

Then `MyService` (and any class derived from it) methods will be UOW.

However, there are **some rules should be followed** in order to make it working;

- If you are **not injecting** the service over an interface (like `IMyService`), then the methods of the service must be `virtual` (otherwise, [dynamic proxy / interception](#) system can not work).
- Only `async` methods (methods returning a `Task` or `Task<T>`) are intercepted. So, sync methods can not start a UOW.

Notice that if `FooAsync` is called inside a UOW scope, then it already participates to the UOW without needing to the `IUnitOfWorkEnabled` or any other configuration.

UnitOfWorkAttribute

`UnitOfWork` attribute provides much more possibility like enabling or disabling UOW and controlling the transaction behavior.

`UnitOfWork` attribute can be used for a **class** or a **method** level.

Example: Enable UOW for a specific method of a class

Share on : [Twitter](#) [LinkedIn](#) [Email](#)

In this document

https://docs.abp.io/en/abp/latest/Unit-Of-Work

3/10

 Filter topics

> [Getting Started](#)

> [Startup Templates](#)

> [Tutorials](#)

> [Fundamentals](#)

> [Infrastructure](#)

> [Architecture](#)

> [Modularity](#)

> [Domain Driven Design](#)

→ [Overall](#)

> [Domain Layer](#)

> [Application Layer](#)

→ [Application Services](#)

→ [Data Transfer Objects](#)

→ [Unit Of Work](#)

→ [Guide: Implementing DDD](#)

> [Multi Tenancy](#)

> [Microservices](#)

> [API](#)

> [User Interface](#)

> [Data Access](#)

> [Real Time](#)

→ [Testing](#)

> [Samples](#)

> [Application Modules](#)

> [Release Information](#)

> [Reference](#)

→ [Contribution Guide](#)

Share on :   

In this document

```
using System.Threading.Tasks;
using Volo.Abp.DependencyInjection;
using Volo.Abp.Uow;

namespace AbpDemo
{
    public class MyService : ITransientDependency
    {
        [UnitOfWork]
        public virtual async Task FooAsync()
        {
            //this is a method with a UOW scope
        }

        public virtual async Task BarAsync()
        {
            //this is a method without UOW
        }
    }
}
```

Example: Enable UOW for all the methods of a class

```
using System.Threading.Tasks;
using Volo.Abp.DependencyInjection;
using Volo.Abp.Uow;

namespace AbpDemo
{
    [UnitOfWork]
    public class MyService : ITransientDependency
    {
        public virtual async Task FooAsync()
        {
            //this is a method with a UOW scope
        }

        public virtual async Task BarAsync()
        {
            //this is a method with a UOW scope
        }
    }
}
```

Again, the **same rules** are valid here:

- If you are **not injecting** the service over an interface (like `IMyService`), then the methods of the service must be `virtual` (otherwise, [dynamic proxy / interception](#) system can not work).
- Only `async` methods (methods returning a `Task` or `Task<T>`) are intercepted. So, sync methods can not start a UOW.

UnitOfWorkAttribute Properties

- `IsTransactional` (`bool?`): Used to set whether the UOW should be transactional or not. **Default value is `null`** . if you leave it `null` , it is determined automatically based on the conventions and the configuration.

 Filter topics

> [Getting Started](#)

> [Startup Templates](#)

> [Tutorials](#)

> [Fundamentals](#)

> [Infrastructure](#)

> [Architecture](#)

> [Modularity](#)

> [Domain Driven Design](#)

→ [Overall](#)

> [Domain Layer](#)

> [Application Layer](#)

→ [Application Services](#)

→ [Data Transfer Objects](#)

→ [Unit Of Work](#)

→ [Guide: Implementing DDD](#)

→ [Multi Tenancy](#)

→ [Microservices](#)

> [API](#)

> [User Interface](#)

> [Data Access](#)

> [Real Time](#)

→ [Testing](#)

> [Samples](#)

> [Application Modules](#)

> [Release Information](#)

> [Reference](#)

→ [Contribution Guide](#)

- `Timeout` (`int?`): Used to set the timeout value for this UOW. **Default value is `null`** and fallbacks to the default configured value.
- `IsolationLevel` (`IsolationLevel?`): Used to set the [isolation level](#) of the database transaction, if the UOW is transactional. If not set, uses the default configured value.
- `IsDisabled` (`bool`): Used to disable the UOW for the current method/class.

If a method is called in an ambient UOW scope, then the `UnitOfWork` attribute is ignored and the method participates to the surrounding transaction in any way.

Example: Disable UOW for a controller action

```
using System.Threading.Tasks;
using Volo.Abp.AspNetCore.Mvc;
using Volo.Abp.Uow;

namespace AbpDemo.Web
{
    public class MyController : AbpController
    {
        [UnitOfWork(IsDisabled = true)]
        public virtual async Task FooAsync()
        {
            //...
        }
    }
}
```

IUnitOfWorkManager

`IUnitOfWorkManager` is the main service that is used to control the unit of work system. The following sections explains how to directly work with this service (while most of the times you won't need).

Begin a New Unit Of Work

`IUnitOfWorkManager.Begin` method is used to create a new UOW scope.

Example: Create a new non-transactional UOW scope

Share on : [Twitter](#) [LinkedIn](#) [Email](#)

In this document

https://docs.abp.io/en/abp/latest/Unit-Of-Work

5/10

> [Getting Started](#)

> [Startup Templates](#)

> [Tutorials](#)

> [Fundamentals](#)

> [Infrastructure](#)

> [Architecture](#)

> [Modularity](#)

> [Domain Driven Design](#)

→ [Overall](#)

> [Domain Layer](#)

> [Application Layer](#)

→ [Application Services](#)

→ [Data Transfer Objects](#)

→ [Unit Of Work](#)

→ [Guide: Implementing DDD](#)

→ [Multi Tenancy](#)

→ [Microservices](#)

> [API](#)

> [User Interface](#)

> [Data Access](#)

> [Real Time](#)

→ [Testing](#)




> [Samples](#)

> [Application Modules](#)

> [Release Information](#)

> [Reference](#)

→ [Contribution Guide](#)

Share on :   

In this document

```
using System.Threading.Tasks;
using Volo.Abp.DependencyInjection;
using Volo.Abp.Uow;

namespace AbpDemo
{
    public class MyService : ITransientDependency
    {
        private readonly IUnitOfWorkManager _unitOfWorkManager;

        public MyService(IUnitOfWorkManager unitOfWorkManager)
        {
            _unitOfWorkManager = unitOfWorkManager;
        }

        public virtual async Task FooAsync()
        {
            using (var uow = _unitOfWorkManager.Begin(
                requiresNew: true, isTransactional: false
            ))
            {
                //...

                await uow.CompleteAsync();
            }
        }
    }
}
```

`Begin` method gets the following optional parameters:

- `requiresNew` (`bool`): Set `true` to ignore the surrounding unit of work and start a new UOW with the provided options. **Default value is `false` . If it is `false` and there is a surrounding UOW, `Begin` method doesn't actually begin a new UOW, but silently participates to the existing UOW.**
- `isTransactional` (`bool`). Default value is `false` .
- `isolationLevel` (`IsolationLevel?`): Used to set the [isolation level](#) of the database transaction, if the UOW is transactional. If not set, uses the default configured value.
- `Timeout` (`int?`): Used to set the timeout value for this UOW. **Default value is `null`** and fallbacks to the default configured value.

The Current Unit Of Work

UOW is ambient, as explained before. If you need to access to the current unit of work, you can use the `IUnitOfWorkManager.Current` property.

Example: Get the current UOW

https://docs.abp.io/en/abp/latest/Unit-Of-Work

6/10

 Filter topics

- > [Getting Started](#)
- > [Startup Templates](#)
- > [Tutorials](#)
- > [Fundamentals](#)
- > [Infrastructure](#)
- ✓ [Architecture](#)

> [Modularity](#)

✓ Domain Driven Design

→ [Overall](#)

> [Domain Layer](#)

✓ Application Layer

→ [Application Services](#)

→ [Data Transfer Objects](#)

→ [Unit Of Work](#)

→ [Guide: Implementing DDD](#)

→ [Multi Tenancy](#)

→ [Microservices](#)

> [API](#)

> [User Interface](#)

> [Data Access](#)

> [Real Time](#)

→ [Testing](#)

> [Samples](#)

> [Application Modules](#)

> [Release Information](#)

> [Reference](#)

→ [Contribution Guide](#)

```
using System.Threading.Tasks;
using Volo.Abp.DependencyInjection;
using Volo.Abp.Uow;

namespace AbpDemo
{
    public class MyProductService : ITransientDependency
    {
        private readonly IUnitOfWorkManager _unitOfWorkManager;

        public MyProductService(IUnitOfWorkManager unitOfWorkManager)
        {
            _unitOfWorkManager = unitOfWorkManager;
        }

        public async Task FooAsync()
        {
            var uow = _unitOfWorkManager.Current;
            //...
        }
    }
}
```

`Current` property returns a `IUnitOfWork` object.

Current Unit Of Work can be `null` if there is no surrounding unit of work. It won't be `null` if your class is a conventional UOW class, you manually made it UOW or it was called inside a UOW scope, as explained before.

SaveChangesAsync

`IUnitOfWork.SaveChangesAsync()` method can be needed to save all the changes until now to the database. If you are using EF Core, it behaves exactly same. If the current UOW is transactional, even saved changes can be rolled back on an error (for the supporting database providers).

Example: Save changes after inserting an entity to get its auto-increment id

Share on : [Twitter](#) [LinkedIn](#) [Email](#)

In this document

https://docs.abp.io/en/abp/latest/Unit-Of-Work

7/10

> **Getting Started**

> **Startup Templates**

> **Tutorials**

> **Fundamentals**

> **Infrastructure**

> **Architecture**

> [Modularity](#)

> Domain Driven Design

→ [Overall](#)

> [Domain Layer](#)

> Application Layer

→ [Application Services](#)

→ [Data Transfer Objects](#)

→ [Unit Of Work](#)

→ [Guide: Implementing DDD](#)

> [Multi Tenancy](#)

> [Microservices](#)

> **API**

> **User Interface**

> **Data Access**

> **Real Time**

> **Testing**

> **Samples**

> **Application Modules**

> **Release Information**

> **Reference**

> **Contribution Guide**

In this document

```
using System.Threading.Tasks;
using Volo.Abp.Application.Services;
using Volo.Abp.Domain.Repositories;

namespace AbpDemo
{
    public class CategoryAppService : ApplicationService
    {
        private readonly IRepository<Category, int> _categoryRepository;

        public CategoryAppService(IRepository<Category, int> categoryRepository)
        {
            _categoryRepository = categoryRepository;
        }

        public async Task<int> CreateAsync(string name)
        {
            var category = new Category {Name = name};
            await _categoryRepository.InsertAsync(category);

            //Saving changes to be able to get the auto-incremented id
            await UnitOfWorkManager.Current.SaveChangesAsync();

            return category.Id;
        }
    }
}
```

This example uses auto-increment `int` primary key for the `Category` entity. Auto-increment PKs require to save the entity to the database to get the id of the new entity.

This example is an [application service](#) derived from the base `ApplicationService` class, which already has the `IUnitOfWorkManager` service injected as the `UnitOfWorkManager` property. So, no need to inject it manually.

Since getting the current UOW is pretty common, there is also a `CurrentUnitOfWork` property as a shortcut to the `UnitOfWorkManager.Current` . So, the example above can be changed to use it:

```
await CurrentUnitOfWork.SaveChangesAsync();
```

Alternative to the SaveChanges()

Since saving changes after inserting, updating or deleting an entity can be frequently needed, corresponding [repository](#) methods has an optional `autoSave` parameter. So, the `CreateAsync` method above could be re-written as shown below:

> [Getting Started](#)

> [Startup Templates](#)

> [Tutorials](#)

> [Fundamentals](#)

> [Infrastructure](#)

> [Architecture](#)

> [Modularity](#)

> [Domain Driven Design](#)

→ [Overall](#)

> [Domain Layer](#)

> [Application Layer](#)

→ [Application Services](#)

→ [Data Transfer Objects](#)

→ [Unit Of Work](#)

→ [Guide: Implementing DDD](#)

→ [Multi Tenancy](#)

→ [Microservices](#)

> [API](#)

> [User Interface](#)

> [Data Access](#)

> [Real Time](#)

→ [Testing](#)

> [Samples](#)

> [Application Modules](#)

> [Release Information](#)

> [Reference](#)

→ [Contribution Guide](#)

```
public async Task<int> CreateAsync(string name)
{
    var category = new Category {Name = name};
    await _categoryRepository.InsertAsync(category, autoSave: true);
    return category.Id;
}
```

If your intent is just to save the changes after creating/updating/deleting an entity, it is suggested to use the `autoSave` option instead of manually using the `CurrentUnitOfWork.SaveChangesAsync()` .

Note-1: All changes are automatically saved when a unit of work ends without any error. So, don't call `SaveChangesAsync()` and don't set `autoSave` to `true` unless you really need it.

Note-2: If you use `Guid` as the primary key, you never need to save changes on insert to just get the generated id, because `Guid` keys are set in the application and are immediately available once you create a new entity.

Other IUnitOfWork Properties/Methods

- `OnCompleted` method gets a callback action which is called when the unit of work successfully completed (where you can be sure that all changes are saved).
- `Failed` and `Disposed` events can be used to be notified if the UOW fails or when it is disposed.
- `Complete` and `Rollback` methods are used to complete (commit) or roll backs the current UOW, which are normally used internally by the ABP Framework but can be used if you manually start a transaction using the `IUnitOfWorkManager.Begin` method.
- `Options` can be used to get options that was used while starting the UOW.
- `Items` dictionary can be used to store and get arbitrary objects inside the same unit of work, which can be a point to implement custom logics.

ASP.NET Core Integration

Unit of work system is fully integrated to the ASP.NET Core. It properly works when you use ASP.NET Core MVC Controllers or Razor Pages. It defines action filters and page filters for the UOW system.

You typically do nothing to configure the UOW when you use ASP.NET Core.

Unit Of Work Middleware

`AbpUnitOfWorkMiddleware` is a middleware that can enable UOW in the ASP.NET Core request pipeline. This might be needed if you need to enlarge the UOW scope to cover some other middleware(s).

Example:

Share on :

[Twitter](#) [LinkedIn](#) [Email](#)

In this document

https://docs.abp.io/en/abp/latest/Unit-Of-Work

9/10

```
app.UseUnitOfWork();
app.UseConfiguredEndpoints();
```

In this document

- > **Getting Started**
- > **Startup Templates**
- > **Tutorials**
- > **Fundamentals**
- > **Infrastructure**
- ✓ **Architecture**
 - > [Modularity](#)
 - ✓ Domain Driven Design
 - [Overall](#)
 - > [Domain Layer](#)
 - ✓ Application Layer
 - [Application Services](#)
 - [Data Transfer Objects](#)
 - Unit Of Work
 - [Guide: Implementing DDD](#)
 - [Multi Tenancy](#)
 - [Microservices](#)
- > **API**
- > **User Interface**
- > **Data Access**
- > **Real Time**
- **Testing**
- > **Samples**
- > **Application Modules**
- > **Release Information**
- > **Reference**
- **Contribution Guide**