

# Articles Tutorials

## In this document

[Edit on GitHub](#)

### Introduction

### Connection & Transaction Management in ASP.NET Boilerplate

Conventional Unit Of Work Methods

Controlling the Unit Of Work

### Unit Of Work in Detail

Disabling Unit Of Work

Non-Transactional Unit Of Work

A Unit Of Work Method Calls Another

Unit Of Work Scope

Automatically Saving Changes

IRepository.GetAll() Method

UnitOfWork Attribute Restrictions

### Options

### Methods

SaveChanges

### Events

## Introduction

Connection and transaction management is one of the most important concepts in an application that uses a database. You need to know when to open a connection, when to start a transaction, and how to dispose the connection, and so on... ASP.NET Boilerplate manages database connections and transactions by using its **unit of work** system.

## Connection & Transaction Management in ASP.NET Boilerplate

ASP.NET Boilerplate **opens** a database connection (it may not be opened immediately, but opened during the first database usage, based on the ORM provider implementation) and begins a **transaction** when **entering** a **unit of work method**. You can use the connection safely in this method. At the end of the method, the transaction is **committed** and the connection is **disposed**. If the method throws an **exception**, the transaction is **rolled back**, and the connection is disposed. In this way, a unit of work method is **atomic** (a **unit of work**). ASP.NET Boilerplate does all of this automatically.

If a unit of work method calls another unit of work method, both use the same connection & transaction. The first entered method manages the connection & transaction and then the others reuse it.

The default **IsolationLevel** for a unit of work is **ReadUncommitted** if it is not configured. It can be easily configured using unit of work [options](#).

## Conventional Unit Of Work Methods

Some methods are unit of work methods by default:

- All [MVC](#), [Web API](#) and [ASP.NET Core MVC](#) Controller actions.

- All [Application Service](#) methods.
- All [Repository](#) methods.

Assume that we have an [application service](#) method like the one below:

	Copy
<pre>public class PersonAppService : IPersonAppService {     private readonly IPersonRepository _personRepository;     private readonly IStatisticsRepository _statisticsRepository;      public PersonAppService(IPersonRepository personRepository, IStatisticsRepository statisticsRepository)     {         _personRepository = personRepository;         _statisticsRepository = statisticsRepository;     }      public void CreatePerson(CreatePersonInput input)     {         var person = new Person { Name = input.Name, EmailAddress = input.EmailAddress };         _personRepository.Insert(person);         _statisticsRepository.IncrementPeopleCount();     } }</pre>	

In the CreatePerson method, we're inserting a person using the person repository and incrementing the total people count using the statistics repository. Both of these repositories **share** the same connection and transaction, since the application service method is a unit of work by default. ASP.NET Boilerplate opens a database connection and starts a transaction when entering the CreatePerson method, and if no exception is thrown, it commits the transaction at the end of it. If an exception is thrown, it rolls everything back. In this way, all the database operations in the CreatePerson method become **atomic** (a **unit of work**).

In addition to the default, conventional unit of work classes, you can add your own convention in the PreInitialize method of your [module](#) like below:

	Copy
<pre>Configuration.UnitOfWork.ConventionalUowSelectors.Add(type =&gt; ...);</pre>	

You should check the type and return true if the type must be a conventional unit of work class.

## Controlling the Unit Of Work

The unit of work **implicitly** works for the methods defined above. In most cases you don't have to control the unit of work manually for web applications. You can **explicitly** use it if you want to control the unit of work somewhere else. There are two approaches for it.

### UnitOfWork Attribute

The first and preferred approach is to use the **UnitOfWork** attribute. Example:

	Copy
<pre>[UnitOfWork] public void CreatePerson(CreatePersonInput input) {     var person = new Person { Name = input.Name, EmailAddress = input.EmailAddress };     _personRepository.Insert(person);     _statisticsRepository.IncrementPeopleCount(); }</pre>	

This way, the CreatePerson method becomes a unit of work and manages the database connection and transaction. Both repositories use the same unit of work. Note that you do not need the UnitOfWork attribute if this is an application service method. See the '[unit of work method restrictions](#)' section.

Filter...

latest (v7.1) ▾

ABP Framework

Overall

- [Introduction](#)
- [Tutorials & Articles](#)
- [NLayer Architecture](#)
- [Module System](#)
- [Startup Configuration](#)
- [Multi-Tenancy](#)
- [OWIN Integration](#)
- [Debugging](#)
- [API Reference](#)

Common Structures

- [Dependency Injection](#)
- [Session](#)
- [Caching](#)
- [Logging](#)
- [Setting Management](#)
- [Timing](#)
- [Object To Object Mapping \(and AutoMapper Integration\)](#)
- [Email Sending \(and MailKit Integration\)](#)

Domain Layer

- [Entities](#)

<a href="#">Multi-Lingual Entities</a>
<a href="#">Value Objects</a>
<a href="#">Repositories</a>
<a href="#">Domain Services</a>
<a href="#">Specifications</a>
<a href="#">Unit Of Work</a>
<a href="#">Domain Events (EventBus)</a>
<a href="#">Data Filters</a>
<a href="#">Dynamic Parameter System</a>
<a href="#">Object Comparators</a>

Application Layer

<a href="#">Application Services</a>
<a href="#">Data Transfer Objects</a>
<a href="#">Validating Data Transfer Objects</a>
<a href="#">Authorization</a>
<a href="#">Feature Management</a>
<a href="#">Audit Logging</a>
<a href="#">Entity History</a>

Distributed Service Layer

ASP.NET Web API

<a href="#">Web API Controllers</a>
<a href="#">Dynamic Web API Layer</a>
<a href="#">OData Integration</a>

There are options for the `UnitOfWork` attribute. See the 'unit of work in detail' section. The `UnitOfWork` attribute can also be used on classes to configure all the methods of them. The method attribute overrides the class attribute if it exists.

## IUnitOfWorkManager [↗](#)

The second approach is to use the `IUnitOfWorkManager.Begin(...)` method as shown below:

	Copy
<pre>public class MyService {     private readonly IUnitOfWorkManager _unitOfWorkManager;     private readonly IPersonRepository _personRepository;     private readonly IStatisticsRepository _statisticsRepository;      public MyService(IUnitOfWorkManager unitOfWorkManager, IPersonRepository personRepository, IStatisticsRepository statisticsRepository)     {         _unitOfWorkManager = unitOfWorkManager;         _personRepository = personRepository;         _statisticsRepository = statisticsRepository;     }      public void CreatePerson(CreatePersonInput input)     {         var person = new Person { Name = input.Name, EmailAddress = input.EmailAddress };          using (var unitOfWork = _unitOfWorkManager.Begin())         {             _personRepository.Insert(person);             _statisticsRepository.IncrementPeopleCount();              unitOfWork.Complete();         }     } }</pre>	

You can inject and use `IUnitOfWorkManager` as shown here (Some base classes already have `UnitOfWorkManager` injected by default: MVC Controllers, [application services](#), [domain services](#)...). This way, you can create a **limited scope** unit of work. Using this approach, you must call the **Complete** method manually. If you don't call it, the transaction is rolled back and the changes are not saved.

The **Begin** method has overloads that set the **unit of work options**. It's simpler (and recommended) to use the `UnitOfWork` attribute if you don't otherwise have a good reason.

## Unit Of Work in Detail

### Disabling Unit Of Work

You may want to disable the unit of work **for the conventional unit of work methods** . To do that, use the `UnitOfWorkAttribute`'s **IsDisabled** property. Example usage:

	Copy
<pre>[UnitOfWork(IsDisabled = true)] public virtual void RemoveFriendship(RemoveFriendshipInput input) {     _friendshipRepository.Delete(input.Id); }</pre>	

Normally, you don't want to do this, but in some situations you may want to disable the unit of work:

- You may want to use the unit of work in a limited scope with the `UnitOfWorkScope` class as described above.

Note that if a unit of work method calls this RemoveFriendship method, disabling this method is ignored, and it will use the same unit of work with the caller method. So, disable carefully! The code above works well since the repository methods are a unit of work by default.

## Non-Transactional Unit Of Work

By its nature, a unit of work is transactional. ASPNET Boilerplate starts, commits or rolls back an explicit database-level transaction. In some special cases, the transaction may cause problems since it may lock some rows or tables in the database. In these situations, you may want to disable the database-level transaction. The UnitOfWork attribute can get a boolean value in its constructor to work as non-transactional. Example usage:

	Copy
<pre>[UnitOfWork(isTransactional: false)] public GetTasksOutput GetTasks(GetTasksInput input) {     var tasks = _taskRepository.GetAllWithPeople(input.AssignedPersonId, input.State);     return new GetTasksOutput     {         Tasks = Mapper.Map&lt;List&lt;TaskDto&gt;&gt;(tasks)     }; }</pre>	

We recommend you use this attribute as **[UnitOfWork(isTransactional: false)]**. It's more readable and explicit, but you could also use [UnitOfWork(false)].

Note that ORM frameworks like NHibernate and EntityFramework internally save changes in a single command. Assume that you updated a few Entities in a non-transactional UOW. Even in this situation all the updates are performed at end of the unit of work with a single database command. If you execute an SQL query directly, it's performed immediately and not rolled back if your UOW is not transactional.

There is a restriction for non-transactional UOWs. If you're already in a transactional unit of work scope, setting isTransactional to false is ignored (use the Transaction Scope Option to create a non-transactional unit of work in a transactional unit of work).

Use a non-transactional unit of work carefully since most of the time things should be transactional to ensure data integrity. If your method just reads data and does not change it, it can be safely non-transactional.

## A Unit Of Work Method Calls Another

The unit of work is ambient. If a unit of work method calls another unit of work method, they share the same connection and transaction. The first method manages the connection and then the other methods reuse it.

## Unit Of Work Scope

You can create a different and isolated transaction in another transaction or you can create a non-transactional scope in a transaction. .NET defines [TransactionScopeOption](#) for that. You can set the Scope option of the unit of work to control it.

## Automatically Saving Changes

If a method is a unit of work, ASPNET Boilerplate automatically saves all the changes at the end of the method. Assume that we need a method to update the name of a person:

	Copy
<pre>[UnitOfWork] public void UpdateName(UpdateNameInput input) {     var person = _personRepository.Get(input.PersonId);     person.Name = input.NewName; }</pre>	

That's all you have to do! The name was updated. We didn't even have to call the `_personRepository.Update` method. The ORM framework keeps track of all the changes of entities in a unit of work and reflects these changes to the database.

Note that we do not need to declare the `UnitOfWork` attribute for conventional unit of work methods.

## IRepository.GetAll() Method

When you call the `GetAll()` method of a repository, there must be an open database connection since it returns `IQueryable`. This is needed because of the deferred execution of `IQueryable`. It does not perform the database query unless you call the `ToList()` method or use `IQueryable` in a `foreach` loop, or somehow access the queried items. So when you call the `ToList()` method, the database connection has to be alive.

Consider the example below:

	Copy
<pre>[UnitOfWork] public SearchPeopleOutput SearchPeople(SearchPeopleInput input) {     // get IQueryable&lt;Person&gt;     var query = _personRepository.GetAll();      // add some filters if selected     if (!string.IsNullOrEmpty(input.SearchedName))     {         query = query.Where(person =&gt; person.Name.StartsWith(input.SearchedName));     }      if (input.IsActive.HasValue)     {         query = query.Where(person =&gt; person.IsActive == input.IsActive.Value);     }      // get paged result list     var people = query.Skip(input.SkipCount).Take(input.MaxResultCount).ToList();      return new SearchPeopleOutput { People = Mapper.Map&lt;List&lt;PersonDto&gt;&gt;(people) }; }</pre>	

Here the `SearchPeople` method is a unit of work since the `ToList()` method of `IQueryable` is called in the method body. The database connection must also be open when `IQueryable.ToList()` is executed.

In most cases you will use the `GetAll` method safely in a web application, since all the controller actions are a unit of work by default. This way, the database connection is available during the entire request.

## UnitOfWork Attribute Restrictions

You can use the `UnitOfWork` attribute for:

- All **public** or **public virtual** methods for classes that are used over an interface (Like an application service used over a service interface).
- All **public virtual** methods for self-injected classes (Like **MVC Controllers** and **Web API Controllers**).
- All **protected virtual** methods.

We recommended you always make the methods **virtual**. You can **not use the attribute for private methods** because ASP.NET Boilerplate uses dynamic proxying for that, and because private methods can not be seen from derived classes. The `UnitOfWork` attribute (and any proxying) does not work if you don't use [dependency injection](#) and instantiate the class yourself.

## Options

There are some options that can be used to change the behavior of a unit of work.

First, we can change the default values of all the unit of works in the [startup configuration](#). This is generally done in the PreInitialize method of our [module](#).

	Copy
<pre>public class SimpleTaskSystemCoreModule : AbpModule {     public override void PreInitialize()     {         Configuration.UnitOfWork.IsolationLevel = IsolationLevel.ReadCommitted;         Configuration.UnitOfWork.Timeout = TimeSpan.FromMinutes(30);     }      //...other module methods }</pre>	

As a second option, we can override the defaults for a particular unit of work. The **UnitOfWork** attribute constructor and IUnitOfWorkManager.**Begin** method have overloads to set these options.

As a final option, you can use the startup configuration to configure the default unit of work attributes for the ASP.NET MVC, Web API and ASP.NET Core MVC Controllers (see their documentation for more info).

## Methods

The UnitOfWork system works seamlessly and invisibly, but in some special cases you may need to call its methods.

You can access the current unit of work in one of two ways:

- You can directly use the **CurrentUnitOfWork** property if your class is derived from some specific base classes (ApplicationService, DomainService, AbpController, AbpApiController... etc.)
- You can inject IUnitOfWorkManager into any class and use the **IUnitOfWorkManager.Current** property.

## SaveChanges

ASP.NET Boilerplate saves all changes at the end of a unit of work. You don't have to do anything, but sometimes you may want to save changes to the database in the middle of a unit of work operation. For example, after saving some changes, we may want to get the Id of a newly inserted [Entity](#) using [EntityFramework](#).

You can use the **SaveChanges** or **SaveChangesAsync** method of the current unit of work.

Note that if the current unit of work is transactional, all changes in the transaction are rolled back if an exception occurs. Even the saved changes!

## Events

A unit of work has the **Completed**, **Failed** and **Disposed** events. You can register to these events and perform any operations you need. For example, you may want to run some code when the current unit of work successfully completes. Example:

	Copy
--	------

```
public void CreateTask(CreateTaskInput input)
{
    var task = new Task { Description = input.Description };


    if (input.AssignedPersonId.HasValue)
    {
        task.AssignedPersonId = input.AssignedPersonId.Value;
        _unitOfWorkManager.Current.Completed += (sender, args) => { /* TODO: Send email to assigned person */ };
    }


    _taskRepository.Insert(task);
}
```



 Stars  10k

Follow

 3,750 followers

 [Stack Overflow](#)

2013 - 2022 © [Volosoft](#)



Copyright © 2021 .NET Foundation