4.1 (latest) English

Search in documents

Contributors Edit Last edit: 11/20/2020 Share on :

Filter topics

- Getting Started
- Startup Templates
- Tutorials
- Fundamentals
- Infrastructure
- Architecture
  - Modularity
  - Domain Driven Design
    - → Overall
    - Domain Layer
      - → Entities & Aggregate Roots
      - → Value Objects
      - → Repositories
      - → Domain Services
      - → Specifications
    - Application Layer
    - → 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

Repositories

"*Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects*" (Martin Fowler).

Repositories, in practice, are used to perform database operations for domain objects (see Entities). Generally, a separated repository is used for each **aggregate root** or entity.

Generic Repositories

ABP can provide a **default generic repository** for each aggregate root or entity. You can inject `IRepository<TEntity, TKey>` into your service and perform standard **CRUD** operations.

> Database provider layer should be properly configured to be able to use the default generic repositories. It is **already done** if you've created your project using the startup templates. If not, refer to the database provider documents (EF Core / MongoDB) to configure it.

Example usage of a default generic repository:

```
public class PersonAppService : ApplicationService
{
    private readonly IRepository<Person, Guid> _personR

    public PersonAppService(IRepository<Person, Guid> p
    {
        _personRepository = personRepository;
    }

    public async Task Create(CreatePersonDto input)
    {
        var person = new Person { Name = input.Name, Ag

        await _personRepository.InsertAsync(person);
    }

    public List<PersonDto> GetList(string nameFilter)
    {
        var people = _personRepository
            .Where(p => p.Name.Contains(nameFilter))
            .ToList();

        return people
            .Select(p => new PersonDto {Id = p.Id, Name
            .ToList();
    }
}
```

4.1 (latest) English

Filter topics

- Getting Started
- Startup Templates
- Tutorials
- Fundamentals
- Infrastructure
- Architecture
  - Modularity
  - Domain Driven Design
    - Overall
    - Domain Layer
      - Entities & Aggregate Roots
      - Value Objects
      - Repositories
      - Domain Services
      - Specifications
    - Application Layer
    - 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

> See the "*IQueryable & Async Operations*" section below to understand how you can use **async extension methods**, like `ToListAsync()` (which is strongly suggested) instead of `ToList()` .

In this example;

- `PersonAppService` simply injects `IRepository<Person, Guid>` in it's constructor.
- `Create` method uses `InsertAsync` to save a newly created entity.
- `GetList` method uses the standard LINQ `Where` and `ToList` methods to filter and get a list of people from the data source.

> The example above uses hand-made mapping between entities and DTOs. See object to object mapping document for an automatic way of mapping.

Generic Repositories provides some standard CRUD features out of the box:

- Provides `Insert` method to save a new entity.
- Provides `Update` and `Delete` methods to update or delete an entity by entity object or it's id.
- Provides `Delete` method to delete multiple entities by a filter.
- Implements `IQueryable<TEntity>` , so you can use LINQ and extension methods like `FirstOrDefault` , `Where` , `OrderBy` , `ToList` and so on...

Basic Repositories

Standard `IRepository<TEntity, TKey>` interface extends standard `IQueryable<TEntity>` and you can freely query using standard LINQ methods. However, some ORM providers or database systems may not support standard `IQueryable` interface.

ABP provides `IBasicRepository<TEntity, TPrimaryKey>` and `IBasicRepository<TEntity>` interfaces to support such scenarios. You can extend these interfaces (and optionally derive from `BasicRepositoryBase`) to create custom repositories for your entities.

Depending on `IBasicRepository` but not depending on `IRepository` has an advantage to make possible to work with all data sources even if they don't support `IQueryable` . But major vendors, like Entity Framework, NHibernate or MongoDb already support `IQueryable` .

So, working with `IRepository` is the **suggested** way for typical applications. But reusable module developers may consider `IBasicRepository` to support a wider range of data sources.

Read Only Repositories

There are also `IReadOnlyRepository<TEntity, TKey>` and `IReadOnlyBasicRepository<Tentity, TKey>` interfaces for who only want to depend on querying capabilities of the repositories.

Generic Repository without a Primary Key

If your entity does not have an Id primary key (it may have a composite primary key for instance) then you cannot use the `IRepository<TEntity, TKey>` (or basic/readonly versions) defined above. In that case, you can inject and use `IRepository<TEntity>` for your entity.

4.1 (latest) English

Filter topics

- Getting Started
- Startup Templates
- Tutorials
- Fundamentals
- Infrastructure
- Architecture
  - Modularity
  - Domain Driven Design
    - Overall
    - Domain Layer
      - Entities & Aggregate Roots
      - Value Objects
      - Repositories
      - Domain Services
      - Specifications
    - Application Layer
    - 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

> `IRepository<TEntity>` has a few missing methods those normally works with the `Id` property of an entity. Because of the entity has no `Id` property in that case, these methods are not available. One example is the `Get` method that gets an id and returns the entity with given id. However, you can still use `IQueryable<TEntity>` features to query entities by standard LINQ methods.

Soft / Hard Delete

`DeleteAsync` method of the repository doesn't delete the entity if the entity is a **soft-delete** entity (that implements `ISoftDelete`). Soft-delete entities are marked as "deleted" in the database. Data Filter system ensures that the soft deleted entities are not retrieved from database normally.

If your entity is a soft-delete entity, you can use the `HardDeleteAsync` method to really delete the entity from database in case of you need it.

See the Data Filtering documentation for more about soft-delete.

Custom Repositories

Default generic repositories will be sufficient for most cases. However, you may need to create a custom repository class for your entity.

Custom Repository Example

ABP does not force you to implement any interface or inherit from any base class for a repository. It can be just a simple POCO class. However, it's suggested to inherit existing repository interface and classes to make your work easier and get the standard methods out of the box.

Custom Repository Interface

First, define an interface in your domain layer:

```
public interface IPersonRepository : IRepository<Person
{
    Task<Person> FindByNameAsync(string name);
}
```

This interface extends `IRepository<Person, Guid>` to take advantage of pre-built repository functionality.

Custom Repository Implementation

A custom repository is tightly coupled to the data access tool type you are using. In this example, we will use Entity Framework Core:

4.1 (latest) English

Filter topics

- Getting Started
- Startup Templates
- Tutorials
- Fundamentals
- Infrastructure
- Architecture
  - Modularity
  - Domain Driven Design
    - Overall
    - Domain Layer
      - Entities & Aggregate Roots
      - Value Objects
      - Repositories
      - Domain Services
      - Specifications
    - Application Layer
    - 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

```
public class PersonRepository : EfCoreRepository<MyDbCo
{
    public PersonRepository(IDbContextProvider<TestAppD
        : base(dbContextProvider)
    {

    }

    public async Task<Person> FindByNameAsync(string na
    {
        return await DbContext.Set<Person>()
            .Where(p => p.Name == name)
            .FirstOrDefaultAsync();
    }
}
```

You can directly access the data access provider (`DbContext` in this case) to perform operations.

> See EF Core or MongoDb document for more info about the custom repositories.

IQueryable & Async Operations

`IRepository` inherits from `IQueryable`, that means you can **directly use LINQ extension methods** on it, as shown in the example of the "*Generic Repositories*" section above.

Example: Using the `Where(...)` and the `ToList()` extension methods

```
var people = _personRepository
    .Where(p => p.Name.Contains(nameFilter))
    .ToList();
```

`.ToList`, `Count()` ... are standard extension methods defined in the `System.Linq` namespace (see all).

You normally want to use `.ToListAsync()`, `.CountAsync()` ... instead, to be able to write a **truly async code**.

However, you see that you can't use all the async extension methods in your application or domain layer when you create a new project using the standard application startup template, because;

- These async methods **are not standard LINQ methods** and they are defined in the Microsoft.EntityFrameworkCore NuGet package.
- The standard template **doesn't have a reference** to the EF Core package from the domain and application layers, to be independent from the database provider.

Based on your requirements and development model, you have the following options to be able to use the async methods.

> Using async methods is strongly suggested! Don't use sync LINQ methods while executing database queries to be able to develop a scalable application.

4.1 (latest) English

Filter topics

- Getting Started
- Startup Templates
- Tutorials
- Fundamentals
- Infrastructure
- Architecture
  - Modularity
  - Domain Driven Design
    - Overall
    - Domain Layer
      - Entities & Aggregate Roots
      - Value Objects
      - Repositories
      - Domain Services
      - Specifications
    - Application Layer
    - Guide: Implementing DDD
  - Multi Tenancy
  - Microservices
- API
- User Interface
- Data Access
- Real Time
- Testing
- Samples
- Application Modules
- Release Information
- Reference
- Contribution Guide

Option-1: Reference to the Database Provider Package

The easiest solution is to directly add the EF Core package from the project you want to use these async methods.

> Add the Volo.Abp.EntityFrameworkCore NuGet package to your project, which indirectly reference to the EF Core package. This ensures that you use the correct version of the EF Core compatible to the rest of your application.

When you add the NuGet package to your project, you can take full power of the EF Core extension methods.

Example: Directly using the `ToListAsync()` after adding the EF Core package

```
var people = _personRepository
    .Where(p => p.Name.Contains(nameFilter))
    .ToListAsync();
```

This method is suggested;

- If you are developing an application and you **don't plan to change** EF Core in the future, or you can **tolerate** it if you need to change later. We believe that's reasonable if you are developing a final application.

MongoDB Case

If you are using MongoDB, you need to add the Volo.Abp.MongoDB NuGet package to your project. Even in this case, you can't directly use async LINQ extensions (like `ToListAsync`) because MongoDB doesn't provide async extension methods for `IQueryable<T>`, but provides for `IMongoQueryable<T>`. You need to cast the query to `IMongoQueryable<T>` first to be able to use the async extension methods.

Example: Cast `IQueryable<T>` to `IMongoQueryable<T>` and use `ToListAsync()`

```
var people = ((IMongoQueryable<Person>)_personRepositor
    .Where(p => p.Name.Contains(nameFilter)))
    .ToListAsync();
```

Option-2: Use the IRepository Async Extension Methods

ABP Framework provides async extension methods for the repositories, just similar to async LINQ extension methods.

Example: Use `CountAsync` and `FirstOrDefaultAsync` methods on the repositories

Share on :

In this document

4.1 (latest) English

Filter topics

- Getting Started
- Startup Templates
- Tutorials
- Fundamentals
- Infrastructure
- Architecture
  - Modularity
  - Domain Driven Design
    - Overall
    - Domain Layer
      - Entities & Aggregate Roots
      - Value Objects
      - Repositories
      - Domain Services
      - Specifications
    - Application Layer
    - Guide: Implementing DDD
  - Multi Tenancy
  - Microservices
- API
- User Interface
- Data Access
- Real Time
- Testing
- Samples
- Application Modules
- Release Information
- Reference
- Contribution Guide

```
var countAll = await _personRepository
    .CountAsync();

var count = await _personRepository
    .CountAsync(x => x.Name.StartsWith("A"));

var book1984 = await _bookRepository
    .FirstOrDefaultAsync(x => x.Name == "John");
```

The standard LINQ extension methods are supported: *AllAsync, AnyAsync, AverageAsync, ContainsAsync, CountAsync, FirstAsync, FirstOrDefaultAsync, LastAsync, LastOrDefaultAsync, LongCountAsync, MaxAsync, MinAsync, SingleAsync, SingleOrDefaultAsync, SumAsync, ToArrayAsync, ToListAsync*.

This approach still **has a limitation**. You need to call the extension method directly on the repository object. For example, the below usage is **not supported**:

```
var count = await _bookRepository.Where(x => x.Name.Con
```

This is because the object returned from the `Where` method is not a repository object, it is a standard `IQueryable` interface. See the other options for such cases.

This method is suggested **wherever possible**.

Option-3: IAsyncQueryableExecuter

`IAsyncQueryableExecuter` is a service that is used to execute an `IQueryable<T>` object asynchronously **without depending on the actual database provider**.

Example: Inject & use the `IAsyncQueryableExecuter.ToListAsync()` method

Share on :

In this document

4.1 (latest) English

Filter topics

Share on :

In this document

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Volo.Abp.Application.Dtos;
using Volo.Abp.Application.Services;
using Volo.Abp.Domain.Repositories;
using Volo.Abp.Linq;

namespace AbpDemo
{
    public class ProductAppService : ApplicationService
    {
        private readonly IRepository<Product, Guid> _pr
        private readonly IAsyncQueryableExecuter _async

        public ProductAppService(
            IRepository<Product, Guid> productRepositor
            IAsyncQueryableExecuter asyncExecuter)
        {
            _productRepository = productRepository;
            _asyncExecuter = asyncExecuter;
        }

        public async Task<ListResultDto<ProductDto>> Ge
        {
            //Create the query
            var query = _productRepository
                .Where(p => p.Name.Contains(name))
                .OrderBy(p => p.Name);

            //Run the query asynchronously
            List<Product> products = await _asyncExecut

            //...
        }
    }
}
```

> `ApplicationService` and `DomainService` base classes already have `AsyncExecuter` properties pre-injected and usable without needing an explicit constructor injection.

ABP Framework executes the query asynchronously using the actual database provider's API. While that is not a usual way to execute a query, it is the best way to use the async API without depending on the database provider.

This method is suggested;

- If you want to develop your application code **without depending** on the database provider.
- If you are building a **reusable library** that doesn't have a database provider integration package, but needs to execute an `IQueryable<T>` object in some case.

For example, ABP Framework uses the `IAsyncQueryableExecuter` in the `CrudAppService` base class (see the application services document).

Share on :

Option-4: Custom Repository Methods

You can always create custom repository methods and use the database provider specific APIs, like async extension methods here. See EF Core or MongoDb document for more info about the custom repositories.

This method is suggested;

- If you want to **completely isolate** your domain & application layers from the database provider.
- If you develop a **reusable application module** and don't want to force to a specific database provider, which should be done as a best practice.

In this document