Repository Pattern in ASP.NET Core – Ultimate Guide

DESIGN PATTERNS

In this extensive guide, we will go through everything you will need to know about Repository Pattern in ASP.NET Core, Generic Repository Patterns, Unit of Work and related topics. We will build a project right from scratch where we implement a clean architecture to access data. The source code of this implemenation is over at my Github.

Table of Contents

1. What's a Repository Pattern?
2. Benefits of Repository Pattern
    2.1. Reduces Duplicate Queries
    2.2. De-couples the application from the Data Access Layer
3. Is Repository Pattern Dead?
4. Implementing Repository Pattern in ASP.NET Core 3.1
5. Setting up the Entities and EFCore
6. Let's Keep Repositories Away for a Moment.
7. Practical Use-Case of Repositories
8. Building a Generic Repository

11.1. What would happen if we didnt have an UnitOfWork Abstraction?
12. Testing with PostMan
13. Summary

What's a Repository Pattern?

classes that hide the logics required to store or retrieve data. That, our application will not care about what kind of ORM we are using, as everything related to the ORM is handled within a repository layer. This allows you to have a cleaner seperation of concerns. Repository pattern is one of the heavily used Design Patterns to build cleaner solutions.

Benefits of Repository Pattern

Reduces Duplicate Queries

Imagine having to write lines of code to just fetch some data from your datastore. Now what if this set of queries are going to be used in multiple places in the application. Not very ideal to write this code over and over again, right? Here is the added advantage of Repository Classes. You could write your data access code within the Repository and call it from multiple Controllers / Libraries. Get the point?

There are quite a lot of ORMs available for ASP.NET Core. Currently the most popular one is Entity Framework Core. But that change in the upcoming years. To keep in pace with the evolving technologies and to keep our Solutions upto date, it is highly crucial to build applications that can switch over to a new DataAccessTechnology with minimal impact on our application's code base.

There can be also cases where you need to use multiple ORMs in a single solution. Probably Dapper to fetch the data and EFCore to write the data. This is solely for performance optimizations.

for your application. EFCore becomes one of your options rather than your only option to access data.

> The Architecture should be independent of the Frameworks.
>
> – Uncle Bob (Robert Cecil Martin)

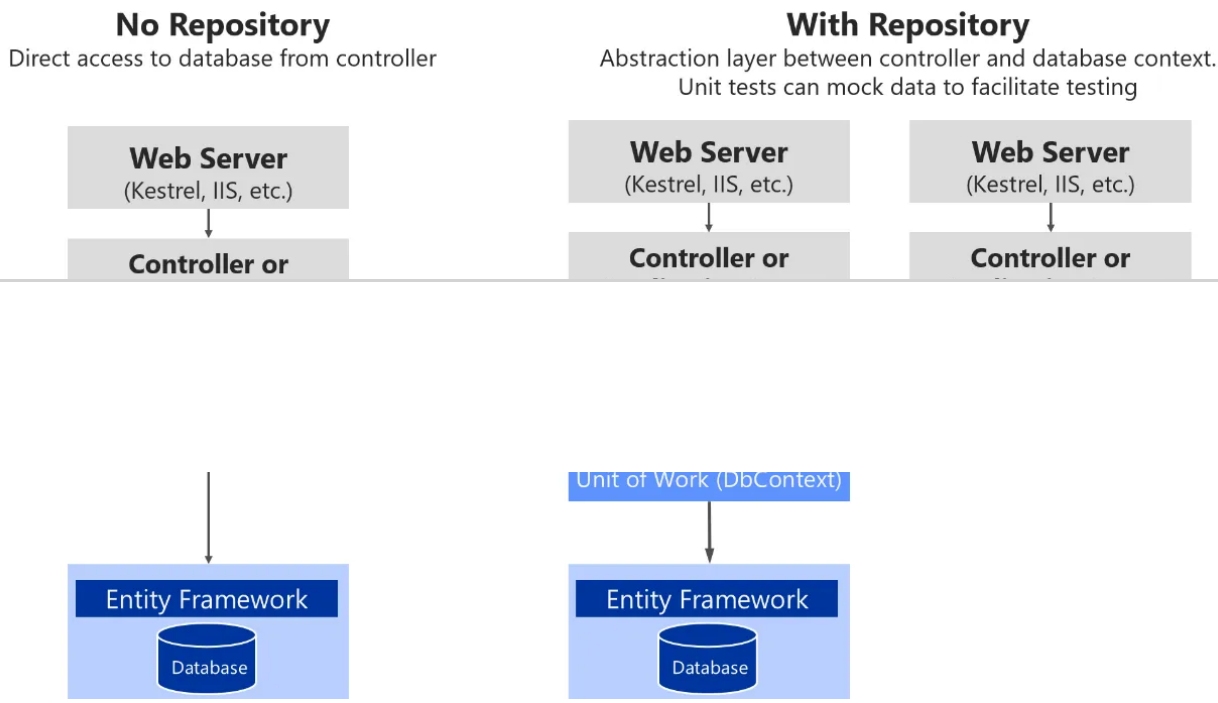
Building an Enterprise level ASP.NET Core Application would really need Repository Patterns to keep the codebase future proof for atleast the next 20-25 years (After which, probably the robots would take over 😛).

Is Repository Pattern Dead?

This is one of the most debated topics within the .NET Core Community. Microsoft has built the Entity Framework Core using the Repository Pattern and Unit of Work Patterns. So, why do we need to add another layer of abstraction over the Entity Framework Core, which is yet another abstration of Data Access. The answer to this is also given by Microsoft.

Read more here – https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-

Microsoft themselves recommend using Repository Patterns in complex scenarios to reduce the coupling and provide better Testability of your solutions. In cases where you want the simplest possible code, you would want to avoid the Repository Pattern.



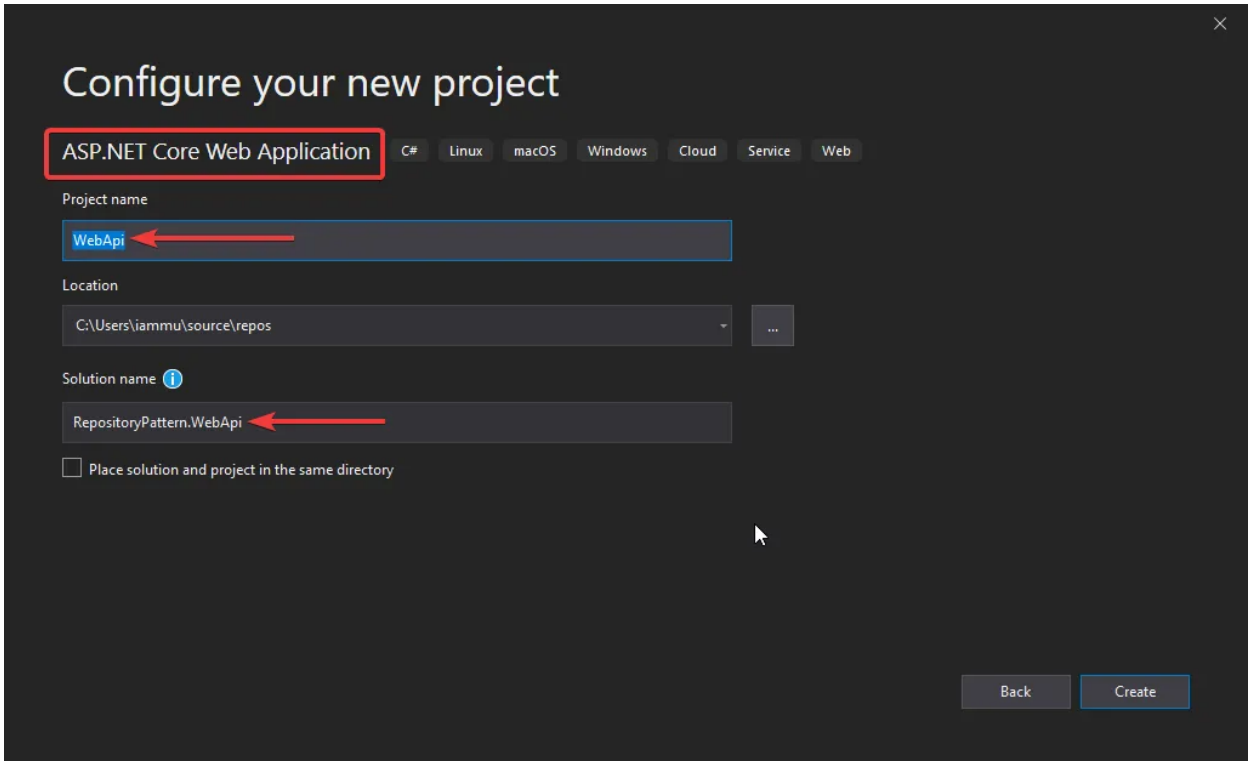
Custom Repository vs DBContext . Source : Microsoft

Adding the Repository has it's own benefits too. But i strongly advice to not use Design Patterns everywhere. Try to use it only whenever the scenario demands the usage of a Design Pattern. That being stated, Repository pattern is something that

Implementing Repository Pattern in ASP.NET Core 3.1

Let's implement Repository Pattern in an ASP.NET Core WebApi Project. What seperates this guide from the others is that we will also be working with a Clean Architecture in mind to demonstrate the real-life implementation. This means that we will be working with multiple Layers and Projects and also go through the basics of Dependency Inversion Principle.

Let's start by creating a new Solution. Here I am naming my Solution as **RepositoryPattern.WebApi** and the first project as **WebApi** (ASP.NET Core).



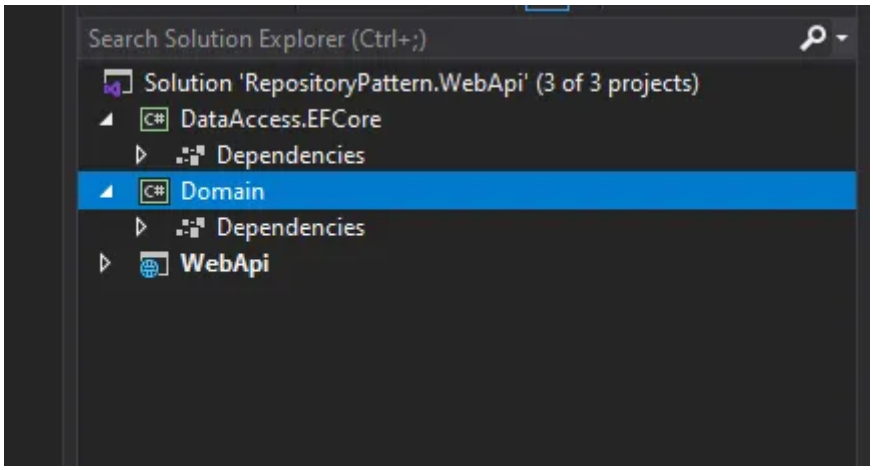
Simalary, let's add 2 more .NET Core Class Library Projects within the solution. We

- Domain – Holds the Entities and Interfaces. It does not depend on any other

represents everything related to EFCore. The aim is that, later down the road one can easily build another Data Access layer like DataAccess.Dapper or so. And our application would still support it. Here is where Dependency Inversion comes to play.

- WebApi – This is like the presentation layer of the entire solution. It depends on both the projects.

Here is how our Solution would look like now.



the Domain Project named Entities.

Create 2 very simple classes – Developer and Project under the Entities Folder.

```
public class Developer
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Followers { get; set; }
}
```

```
public class Project
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Next , we will setup and configure Entity Framework Core. Install these Required Packages in the DataAccess.EFCore Project. Here is where we would have our

```
Install-Package Microsoft.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

Add a reference to the **Domain Project** (where we have defined our entities) and create a new Class in the DataAccess.EFCore Project and Name it **ApplicationContext.cs**.

```
public class ApplicationContext : DbContext
{
    public ApplicationContext(DbContextOptions<ApplicationContext> options
    {
    }
    public DbSet<Developer> Developers { get; set; }
    public DbSet<Project> Projects { get; set; }
}
```

Once our Data Access Layer is done, let's move to the WebApi Project to register EFCore within the ASP.NET Core Application. We will also update the database in this step to accomadate the Developer and Project Table.

Firstly, Install this package on the WebApi Project. This allows you to run EF Core commands on the CLI.

Next, Navigate to Startup.cs and add this line to Register the ApplicationContext class that we created. Note that you will have to add a refernce of the DataAccess.EFCore Project to the WebApi Project.

```
services.AddDbContext<ApplicationContext>(options =>
options.UseSqlServer(
    Configuration.GetConnectionString("DefaultConnection"),
    b => b.MigrationsAssembly(typeof(ApplicationContext).Assembly.FullName
```

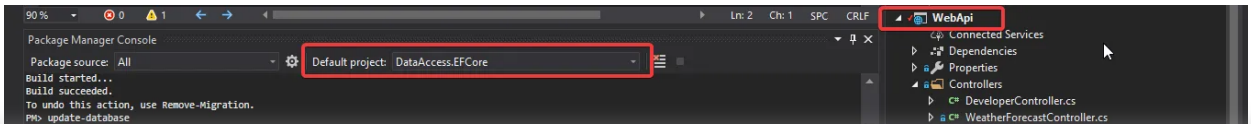
connection string.

```
"ConnectionStrings": {
  "DefaultConnection": "<your connection string>",
},
```

Finally, Let's update the database. Open your Package Manager Console on Visual Studio and run the following commands.

```
add-migration Initial
update-database
```

Make sure that you have set your Startup Project as WebApi and the Default Project as DataAccess.EFCore. Here is a screenshot.



Ps, This is a very basic setup of Entity Framework Core. I have written a detailed Guide on Entity Framework Core – Code First Apporach. Give it a look to learn more.

Let's Keep Repositories Away for a Moment.

Now that we have configured our EFCore Layer, let's talk a bit about the traditional Core is being tightly coupled within your application. So, Tommorow when there is something newer and better than EFCore, you would find it really annoying to implement the new tech and make the corresponding changes. Right?

One more disadvantage of directly using the dbContext directly is that you would be exposing the DbContext, which is totally insecure.

This is the reason to use Repository Pattern in ASP.NET Core Applications.

While Performing CRUD Operations with Entity Framework Core, you might have noticed that the basic essence of the code keeps the same. Yet we write it multiple times over and over. The CRUD Operations include Create, Read, Update, and Delete. So, why not have a class / interface setup where you can generalize each of these operations.

Building a Generic Repository

First up, let's add a new folder **Interfaces** in our Domain Project. Why? Because, we

domain layer will not depends on anything, rather, the other layers tend to depend on the Domain Layer's interface. This is a simple explanation of Dependency Inversion Principle. Pretty Cool, yeah?

Add a new interface, **Interfaces/IGenericRepository.cs**

```
1.  public interface IGenericRepository<T> where T : class
2.  {
3.      T GetById(int id);
4.      IEnumerable<T> GetAll();
5.      IEnumerable<T> Find(Expression<Func<T, bool>> expression);
10. }
```

This will be a Generic Interface, that can be used for both the Developer and Project Classes. Here T is the specific class.

The set of functions depends on your preference. Ideally, we require 7 functions that cover most of the data handling part.

1. Get's the entity By Id.

4. Adds a new record to the context
5. Add a list of records
6. Removes a record from the context
7. Removes a list of records.

Now, Let's implement this Interfaces. Create a new class in the DataAccess.EFCore Project and name it **Repositories/GenericRepository**

```
public class GenericRepository<T> : IGenericRepository<T> where T : class
```

```
    }

    public void Add(T entity)
    {
        _context.Set<T>().Add(entity);
    }

    public void AddRange(IEnumerable<T> entities)
    {
        _context.Set<T>().AddRange(entities);
    }

    public IEnumerable<T> Find(Expression<Func<T, bool>> expression)
    {
        return _context.Set<T>().Where(expression);
    }

    public IEnumerable<T> GetAll()
    {
        return _context.Set<T>().ToList();
    }

    public T GetById(int id)
    {
        return _context.Set<T>().Find(id);
    }

    public void Remove(T entity)
    {
```

```
41.             _context.Set<T>().RemoveRange(entities);
42.         }
43.     }
```

This class will implement the IGenericRepository Interface. We will also inject the ApplicationContext here. This way we are hiding all the actions related to the dbContext object within Repository Classes. Also note that, for the ADD and Remove Functions, we just do the operation on the dbContext object. But we are not yet commiting/updating/saving the changes to the database whatsover. This is not something to be done in a Repository Class. We would need Unit of Work Pattern

Understood why we used a Generic Repository instead of a IDevloperRepository?? When there are large number of entites in our application, we would need seperate repositories for each entities. But we do not want to implement all of the above 7 Functions in each and every Repository Class, right? Thus we made a generic repository that holds the most commonly used implementaions.

Now what happens if we need the records of Most Popular Developers from our Database? We do not have a function for it in our Generic Class, do we ? This is where we can see the advantage of building a Generic Repository.

Inheriting and Extending the Generic Repository

```
1.  public interface IDeveloperRepository : IGenericRepository<Developer>
2.  {
3.      IEnumerable<Developer> GetPopularDevelopers(int count);
4.  }
```

Here we are inheriting all the Functions of the Generic Repository, as well as adding a new Funciton 'GetPopularDevelopers'. Get it?

Let's implement the IDeveloperRepostory. Go to the DataAccess Project and under Repositories folder add a new class DeveloperRepository

```
2.  {
3.      public DeveloperRepository(ApplicationContext context):base(context)
4.      {
5.      }
6.      public IEnumerable<Developer> GetPopularDevelopers(int count)
7.      {
8.          return _context.Developers.OrderByDescending(d => d.Followers).Tak
9.      }
10. }
```

You can notice that we have not implemented all the 7 functions here, as it is is already implemented in our Generic Repository. Saves a lot of lines, yeah?

Similary, let's create interface and implementation for ProjectRepository.

```
1.  public interface IProjectRepository : IGenericRepository<Project>
2.  {
3.  }
```

```
1.  public class ProjectRepository : GenericRepository<Project>, IProjectRepos
2.  {
3.      public ProjectRepository(ApplicationContext context): base(context)
4.      {
5.      }
6.  }
```

You can see that the interface and implementations are quite blank. So why create new class and interface for Project? This can also attribute to a good practice while developing applications. We also anticipate that in future, there can be functions

Finally, let's register these Interfaces to the respective implementaions in the Startup.cs of the WebApi Project. Navigate to Startup.cs/ConfigureServices Method and add these lines.

```
1.  #region Repositories
2.  services.AddTransient(typeof(IGenericRepository<>), typeof(GenericReposito
3.  services.AddTransient<IDeveloperRepository, DeveloperRepository>();
4.  services.AddTransient<IProjectRepository, ProjectRepository>();
5.  #endregion
```

Unit Of Work Pattern

Unit of Work Pattern is a design pattern with which you can expose various respostiories in our application. It has very similar properties of dbContext, just that Unit of Work is not coupled to any framework like dbContext to Entity Framework Core.

Till now, we have built a couple of repositories. We can easily inject these repositories to the constructor of the Services classes and access data. This is quite

Object, we use Unit Of Work.

Unit of Work is responsible for exposing the available Repositories and to Commit Changes to the DataSource ensuring a full transaction, without loss of data.

The other major advantage is that, multiple repository objects will have different instances of dbcontext within them. This can lead to data leaks in complex cases.

Let's say that you have a requirement to insert a new Developer and a new Project

any change to the database. This is exactly why we decided to not include SaveChanges in any of the repostories. Clear?

Rather, the SaveChanges will be available in the UnitOfWork Class. You will get a better idea once you see the impelemntation.

Let's get started with the IUnitOfWork. Create a new Interface in the domain Project, Interfaces/IUnitOfWork

```
5.          int Complete();
6.      }
```

You can see that we are listing the interfaces of the required Repositories within the UOW Interface. It's also a Disposable Element. Finally we have a 'Complete' Function which will save the changes to the database.

Let's implement this interface. Create the implementation at the DataAccess Project. Add a new class in the **UnitOfWork/UnitOfWork.cs**

```
        private readonly ApplicationContext _context;
        public UnitOfWork(ApplicationContext context)
        {
            _context = context;
            Developers = new DeveloperRepository(_context);
            Projects = new ProjectRepository(_context);
        }
        public IDeveloperRepository Developers { get; private set; }
        public IProjectRepository Projects { get; private set; }
        public int Complete()
        {
            return _context.SaveChanges();
        }
        public void Dispose()
        {
            _context.Dispose();
```

Note that, Here we are injecting a private AppplicationContext. Let's wire up or controllers with these Repositories. Ideally you would want to have a service layer between the Repository and Controllers. But, to keep things fairly simple, we will avoid the service layer now.

Before that, let's not forget to register the IUnitofWork Interface in our Application. Navigate to Startup.cs/ConfigureServices Method and add this line.

```
services.AddTransient<IUnitOfWork, UnitOfWork>();
```

Wiring up with an API Controller

Add a new Empty API Controller in the WebAPI Project under the Controllers folder.

```
        private readonly IUnitOfWork _unitOfWork;
        public DeveloperController(IUnitOfWork unitOfWork)
        {
            _unitOfWork = unitOfWork;
        }
    }
```

Here are injecting only the IUnitOfWork object. This way, you can completely avoid writing lines and lines of injections to your controllers.

Now, let's say we need two endpoints for this controller. A POST and a GET Method.

- Get all the Popular Developers.
- Insert a new Develoeper an a new Project.

We'll start working on the methods now.

```
    public IActionResult GetPopularDevelopers([FromQuery]int count)
    {
        var popularDevelopers = _unitOfWork.Developers.GetPopularDevelopers(co
        return Ok(popularDevelopers);
    }
```

descending order of the follower count.

Line #4 Returns a 200 Status Ok with the developers.

```
1.  [HttpPost]
2.  public IActionResult AddDeveloperAndProject()
3.  {
4.      var developer = new Developer
5.      {
6.          Followers = 35,
7.          Name = "Mukesh Murugan"
8.      };
9.      var project = new Project
10.     {
15.         _unitOfWork.Complete();
16.     return Ok();
17. 
18. }
```

Line 4-12 is where I build sample entity objects just for demonstration purpose.

Line 13 – Add the developer object to the uow context.

Line 14 – Add the project object to the uow context.

Line 15 – Finally commits the changes to the database.

What would happen if we didnt have an UnitOfWork Abstraction?

Let's say Line 13 saves the developer to the database, but for some reason, Line 14 fails to store the project. This can be quite fatal for applications due to data inconsistency. By introducing a unit of work, we are able to store both the developer and project in one go, in a single transaction.

Adding a new Developer and Project. Open up Postman and send a POST request to https://localhost:xxxx/api/developer

As expected, we are able to retreive the inserted data. That's it for this extensive guide on Repository Pattern in ASP.NET Core Applications or C# in general.

Liked this article? I have added another article that discusses around the Generic Repository Pattern, it's limitation and how we can use Specification Pattern to overcome it. Specification Pattern in ASP.NET Core is one of the coolest design patterns to have in your application. Read the entire article here – Specification Pattern in ASP.NET Core – Enhancing Generic Repository Pattern

Summary

We learnt all about Repository Pattern in ASP.NET Core Application, Generic Repositories, Unit Of Work , a cleaner way to access data with layered projects, and other Use Case Scenarios. This covers nearly everything that you would need to know to become a Pro at Repository Pattern in ASP.NET Core. Have any suggestions or question? Feel free to leave them in the comments section below. Here is the source code of the implementation for your reference. Thanks and Happy Coding! 😀

← PREVIOUS

Healthchecks in ASP.NET Core –

NEXT →

Custom User Management in ASP.NET

Similar Posts

Implementing CQRS with MediatR in ASP.NET Core – Ultimate Guide

By Mukesh Murugan May 16, 2020

Specification Pattern in ASP.NET Core – Enhancing Generic Repository Pattern

By Mukesh Murugan April 24, 2021

Comment *

Name *

Email *

Website

POST COMMENT

55 Comments

Arjunan says:

June 28, 2020 at 12:53 pm

Reply

Mukesh Murugan says:

June 28, 2020 at 3:45 pm

Thanks! Hope you are liking the blog posts 😀

Regards

Reply

Victor says:

June 28, 2020 at 2:41 pm

Great Article. Quick question, can the repository pattern above be implemented with the CQRS pattern?

Mukesh Murugan says:

June 28, 2020 at 3:13 pm

It really “depends”. If you are building a small application, NO. If you are a building an Enterprise level application with like 10K Lines of code, you would benefit by deceoupling the dbContext away from the Application layer. The only problem that i see is that you would end up with on additional layer of arbitration. But if

Reply

Mukesh Murugan says:

June 28, 2020 at 3:44 pm

Hi again, Have a look at https://github.com/dotnet-architecture/eShopOnWeb . This is one of repositories maintained by Microsoft themselves. You can see at https://github.com/dotnet-architecture/eShopOnWeb/blob/master/src/Web/Features/MyOrders/GetMyOrdersHandler.cs that they themselves are using Repository Pattern along with CQRS. Summary is that, if you think it is maintainable and easy to understand for you, then Repository + CQRS is also a way to go about it.

Thanks and Regards

Reply

Gaurav Vashisth says:

June 29, 2020 at 3:27 am

Hey , I am just a beginner in .net and c# . And I want to know about the job opportunities in c# .net in India .
I am in final year of my graduation which is done now and I want to get a job and basically I am from a bit of java background like core Java and few days back a got an internship where i work on Microsoft dynamics 365 technology and where i have to work on JavaScript and C sharp . So please tell me what I do further

Rasmus Schultz says:

June 29, 2020 at 5:13 am

Please read this article:

https://altkomsoftware.pl/en/blog/create-better-code-using-domain-driven-design/

“In domain driven design a repository is not just a data access object, which implements all CRUD and database queries needed for given entity type. In domain driven design repository should be part of the ubiquitous language and should reflect business concepts. This means that methods on repository should follow our domain language not the database concepts.
We often see generic repositories with methods for CRUD and methods allowing execution of ad-hoc queries. This is not a good way if we want to follow DDD principles.
Instead we should only expose operations required and create methods for queries related to business concepts.”

Don’t feel bad though. Everyone gets this wrong at first, and all the popular articles on the subject teach it this way. Once you’ve tried repositories the DDD way, you’ll see the benefits. 🙂

Reply

macoratti says:
July 3, 2020 at 11:59 am

In small and medium applications can still I use instead repository pattern as “document database” ?

Reply

Mukesh Murugan says:
July 3, 2020 at 4:26 pm

Usage of design patterns solely depends on the developer. If you are comfortable and think that Repository pattern makes sense, go ahead with it. This patterns helps make your application more decoupled.

Thanks and Regards

Reply

Thank you for this article.

I've been using repository pattern since MVC4 but haven't used GenericRepository and UnitOfWork. I just started a new project using .NETCore Api and MongoClient for MongoDb. I will try using this approach and if it works then this will become my new design pattern.

Thanks Mukesh.

Reply

Mukesh Murugan says:
June 30, 2020 at 2:34 pm

Hi, It's quite a brilliant pattern if used wisely. Keep me posted about your project. Thanks and Regards.

Reply

John McHale says:
July 3, 2020 at 2:24 pm

What about using a generic UnitOfWork Pattern. Otherwise, any time you want to add a new repository (e.g. ManagersRepository, TasksRepository) you will have to change your IUnitOFWork interface as well as the concrete UnitOfWork class

Reply

July 6, 2020 at 2:04 pm

How would be the idea of implementing repository patter with onion architecture? Can you please direct me with some articles?

Reply

Mukesh Murugan says:

Open Source Project, Clean Architecture for WebApi.

Please check the repository.

https://github.com/iammukeshm/CleanArchitecture.WebApi

Reply

Jugan says:

July 7, 2020 at 1:01 am

Thanks, Mukesh. I am working on learning new architectures and patterns (I have developed all my projects with only N-layer). I view your blog will be a lead to it. I will go through the repository for better knowledge.

Reply

Hi Mukesh, I was going through your GitHub repo on CleanArchitecture. It has served me to acquire knowledge about architecture in detail. I would wish to know if the UOW layer is not required/possible or have you proposed to integrate in the future.

Thanks in advance

Reply

Hi Jugan,
Yes, since the Repository pattern implementation depends on the develops choice, it can still be clean with a UOW Layer. But to have a much better separation, UOW is a must. I am still in process of building the API. (Currently working on the Auth Services). I will be adding the UOW as well in a few days. However I guess you can get a good overview of clean architecture via the github repo already. Do share it within your community.

Thanks and Regards.

Vinod Kumar Gupta says:
July 8, 2020 at 1:15 am

Nice article Mukesh, appreciated

Reply

Mukesh Murugan says:
July 8, 2020 at 3:24 am

Hi, Thanks 🙂

Reply

Dota says:
July 11, 2020 at 2:52 pm

implemented the unit of work?

Can we have
services.AddTransient();

Instead of
services.AddTransient(typeof(IGenericRepository), typeof(GenericRepository));
services.AddTransient();
services.AddTransient();

If not, what is the reason to keep the repositories? Thanks in advance and keep up the great work!

Reply

Mukesh Murugan says:
July 11, 2020 at 4:10 pm

Yes, Thanks for pointing out that to me. You really don't need to Register the IRepos and UOW takes care of all that. You will just need a services.AddTransient(); in your startup. That's all you would need. I guess I have left out some residue code.

Thanks and Regards

Reply

Very much appreciated…

Reply

Mukesh Murugan says:
July 11, 2020 at 3:55 pm

Thanks for the feedback! 🙂

Reply

John Doe says:
July 16, 2020 at 12:12 pm

Good article. However, may I suggest a few improvements:

only slow but may also crash your server if you have large tables with limited RAM. The IQueryable would allow you to improve the performance.
2. You do not need the unit of work pattern. EF Core already implements it. See this post (https://www.programmingwithwolfgang.com/repository-pattern-net-core/) for better generic repository implementations.

Either way, thanks for your detailed explanations.

Reply

Hey, nice article only question is how you will implement an identity framework in this pattern I mean in which layer? I have read many articles in which the user manager and sign in manager are used within the controller itself which means giving the database calls from the controller. Any idea or article on this will help thanks.

Reply

Mukesh Murugan says:
July 25, 2020 at 8:09 pm

Hi, I would not put them within the controller. If I was using an Onion Architecture, I would add an IAccountService, that has AuthenticateAsync, RegisterAsync and so on. Then I would create a new Infrastructure Layer, Infrastructure.Identity and implement IAccountService in this layer. I dont have an

Repository – https://github.com/iammukeshm/CleanArchitecture.WebApi . The same concept is implemented here.

Thanks and Regards

Reply

Rahul says:

July 26, 2020 at 4:50 am

Reply

idchlife says:

August 8, 2020 at 5:53 pm

Hi! Thanks for the article! It's very nice!

Several years ago I was developing with Symfony 2 framework. Doctrine is used as ORM with unit of work and repositories there.

The thing is, one of the practices there was: create service, which inherits from default entity repository (with all common methods, findById, find, findAll etc). So instead of getting generic repository we were able to get generic repository as service + our additional methods in this service. Basically a win-win situation. We had full first-level access to all the default methods of repositories with our additional methods like "findByEmail" or "findByRelationshipId" etc.

When I started using AspNet with Entity Framework Core, I was surprised that I did not witness word “Repository” anywhere.
Some time passed and I discovered that actually EF Core introduced unit of work and repositores as db context, and apparently Entity Framework did not have those inside.

Is it possible in AspNet + Ef Core to code like it is possible in Symfony + Doctrine? To create separate repositories of entities which wll inherit all default DbSet methods etc? Maybe it is the way, to create custom class which would extend DbSet and register it inside DbContext?

Lokendra Chand says:
August 24, 2020 at 5:30 am

Hi, Mukesh sir,

I understood this . But could you please let me know how to use Unit of Work in Two tables when we need to join table. Furthermore, how to setup generic repository for multiple table query

Reply

Sreenath Ganga says:
September 1, 2020 at 4:07 am

Hi Mukesh
Well explained and simply explained. Very good for those who likes to use this

blogs are super

Hope we can meet some time in TVM

Reply

Mukesh Murugan says:

September 1, 2020 at 10:55 am

Hi Sreenath! Thanks for the feedback 😀 Sure, we should! Can you please connect to my LinkedIn – https://codewithmukesh.com/linkedin.

Thanks 🙂

Reply

Saif says:

September 22, 2020 at 8:07 am

Hi Mukesh,

Great article. But what will be the problem if we refactor the GenericRepository as like

public interface IGenericRepository where T : class

{

Task GetByIdAsync (int id);

Task<IEnumerable> GetAllAsync ();

Task<IEnumerable> FindAsync (Expression<Func> expression);

Task AddAsync (T entity);

Task AddRangeAsync (IEnumerable entities);

Task RemoveRangeAsync (IEnumerable entities);

}

Reply

Niraj Dahal says:

October 4, 2020 at 3:45 am

Very helpful !! Implemented these patterns in my own API as well.

Reply

Mahmoud Soliman says:

October 28, 2020 at 6:06 pm

Hello

It is a wonderful article.

I have a question “sorry if It seems stupid”

In the developer controller we have injected IUnitOfWork in the controller’s constructor, so the .Net core framework will take care of providing a UnitOfWork instance to the controller, but UnitOfWork needs ApplicationContext instance to be constructed.

The question is Who will provide this ApplicationContext instance to the constructor of UnitOfWorkClass?

Reply

Hi, Thanks to your article !

Question : Is the instantiation of repositories in the UnitOfWork class is respectfull of dependency injection principe ?

If not, how can i fix it ?

Reply

---

nirupama says:

November 23, 2020 at 8:14 pm

Hi .. Thanks Much for the article … If you could please suggest a way to avoid the database concurrency issues using this pattern as db context has to be disposed off immeadiately .. Appreciate it

Reply

---

bommi says:

March 22, 2021 at 8:53 pm

Hi, nice article!

One question though, did you forget to add an Update methode ? Seems like all the CRUD's are there but and update is missing ?

Reply

Brian says:

I was thinking the same thing. Will this work:

public T Update(T entity)

{

var entityEntry = _context.Update(entity);

return entityEntry.Entity;

}

Reply

Chris says:

March 25, 2021 at 4:55 pm

Fantastic article. Very clear and well structured. Has helped me a lot. Thank you

Reply

---

prats says:

March 26, 2021 at 3:39 am

Great explanation. So easy to understand

Reply

Rashik Hasnat says:

May 10, 2021 at 5:07 pm

Reply

Sumesh Sukumaran says:

May 26, 2021 at 12:41 pm

Line number 7 and 8 of unitofwork. Why you are newing developerrepository since you have the IDeveloperRepository interface?

Reply

Ricardo says:

June 1, 2021 at 12:21 am

You help me a lot with this guide. Thankyou

Reply

Khoi Nguyen says:

July 7, 2021 at 8:59 pm

Awesome article!

Reply

---

zach says:

Excellently done

I have a question about the method GetAll(). How would one write this to allow Eager Loading of Related Data?

https://docs.microsoft.com/en-us/ef/core/querying/related-data/eager

Here is an example:

```
using (var context = new BloggingContext())
{
var blogs = context.Blogs
.Include(blog => blog.Posts)
.ToList();
}
```

Reply

Shakir says:

January 28, 2022 at 8:50 am

It's a nice article but I am getting an error while adding migration. Please have a look at the error below. I also selected DataAccess.EFCore from the dropdown. Any help will be highly appreciated.

Your target project 'DataAccess.EFCore' doesn't match your migrations assembly 'Domain'. Either change your target project or change your migrations assembly. Change your migrations assembly by using DbContextOptionsBuilder. E.g. options.UseSqlServer(connection, b => b.MigrationsAssembly("DataAccess.EFCore")). By default, the migrations assembly is the assembly containing the DbContext.

Reply

Maroun says:

February 7, 2022 at 9:00 am

Hi,

Thanks for the great article!

Why are you injecting transient and not scoped?

Reply

Mahfoud Bouabdallah says:

March 9, 2022 at 3:09 pm

Thank you very much for the great article .
If you can update this article with implantation AutoMapper I will be very grateful.
And if you can make example of loading related data will be nice
Best Regards

Reply

Jay says:

March 23, 2022 at 12:39 pm

Hi Mukesh,

without Unit of Work, and I am getting below exception when I can try to do multiple database operations in the same service.

A second operation was started on this context before a previous operation completed. This is usually caused by different threads concurrently using the same instance of DbContext. For more information on how to avoid threading issues with DbContext, see https://go.microsoft.com/fwlink/?linkid=2097913.

Reply

Sumit G says:

April 6, 2022 at 12:18 pm

Hello Mukesh,
Really appreciated!
you have simplified the complex process into a simple line of code which will help the community to clear the fundamental of code and functionality.

Thanks for sharing the detailed information

Reply

Shiva Sharan says:

May 9, 2022 at 7:36 am

can you explain hoe to testthe lunit of work implemention for all the CURD operations

Reply

Ryan says:

May 17, 2022 at 8:07 pm

This article is clear and concise. I've been struggling to understand the repository pattern for some time now, and now I understand it. Keep up the good work!

Reply

HaizzTen says:

June 7, 2022 at 1:50 am

I'm quite lucky to find out this post! Thank you and google SE.

Reply

1SamOnline says:

June 10, 2022 at 6:27 pm

My project
WebAssembly…

I was using simple pattern.
(client side) blazor page httpclient service ==> (server side) api controller return answer

I heard that AddScoped, AddTransient, AddSingleton has some reason to use.
So… I changed to use interface.
(client side) blazor page call => interface => httpclient service ====> (server side) api controller => interface => dbcontext manager ===> api controller return answer

I made copy of all files in this post.. Anyway and now I will run this solution using generic interface.

There are too many different kind of Dbcontext in my project. I want to have a simple and combined single interface ^___^;;;

Reply

Abdelouab says:

June 21, 2022 at 4:02 pm

Great work, thank you 🙂

Reply

Search for articles..

Upcoming eBook!

Currently writing an eBook about ASP.NET Core Web API 6.0. Will be sharing my eBook writing journey on a daily basis via my Twitter Handle!

Follow me on Twitter

Detailed articles and guides around .NET, Golang, AWS and other technologies that I come across or

CATEGORIES

AWS

.NET

Golang

Design Patterns

Microservices

NAVIGATE

Home

Blog

About

Contact

Subscribe

CONTACT ME

You can mail me or reach me out at LinkedIn!

Do not forget to Endorse me on LinkedIn if you like my content!

high quality and well detailed!

You can read the Privacy Policy here.