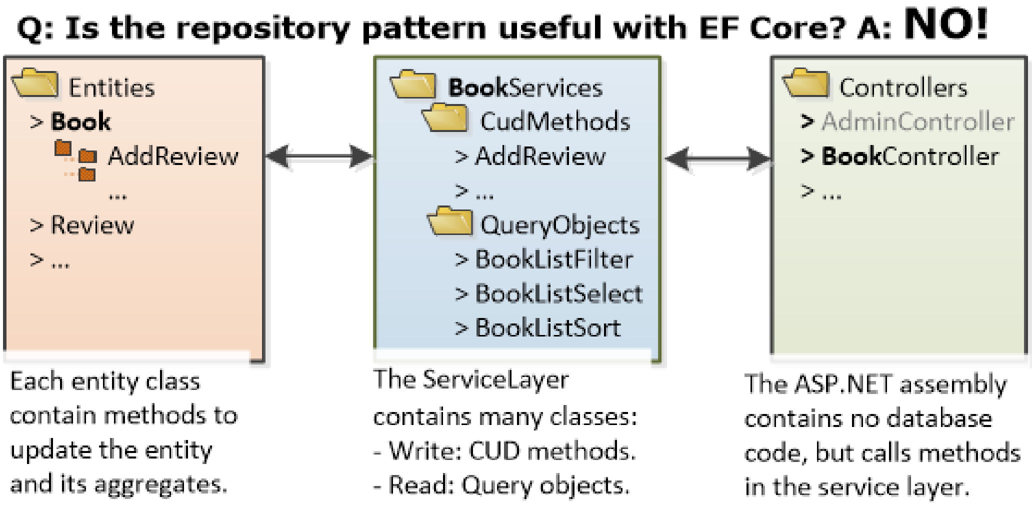

Q: Is the repository pattern useful with EF Core? A: NO!
Entities
> Book
AddReview
...
> Review
> ...
BookServices
CudMethods
> AddReview
> ...
QueryObjects
> BookListFilter
> BookListSelect
> BookListSort
Controllers
> AdminController
> BookController
> ...
Each entity class contain methods to update the entity and its aggregates.
The ServiceLayer contains many classes:
- Write: CUD methods.
- Read: Query objects.
The ASP.NET assembly contains no database code, but calls methods in the service layer.

Is the repository pattern useful with Entity Framework Core?

Last Updated: July 31, 2020 | Created: February 20, 2018

I wrote my first article about the repository pattern in 2014, and it is still a popular post. This is an updated article that takes account of a) the release of Entity Framework Core (EF Core) and b) further investigations of different EF Core database access patterns.

1. Original: Analysing whether Repository pattern useful with Entity Framework (May 2014).
2. First solution: Four months on – my solution to replacing the Repository pattern (Sept 2014).
3. **THIS ARTICLE: Is the repository pattern useful with Entity Framework Core?**
4. Architecture of Business Layer working with Entity Framework (Core and v6).
5. Creating Domain-Driven Design entity classes with Entity Framework Core.
6. GenericServices: A library to provide CRUD front-end services from a EF Core database.
7. Wrapping your business logic with anti-corruption layers – NET Core.

pattern on top of EF Core isn't helpful.

A better solution is to use EF Core directly, which allows you to use all of EF Core's feature to produce high-performing database accesses.

The aims of this article

This article looks at

- What people are saying about the Rep/UoW pattern with EF.
- The pro and cons of using a Rep/UoW pattern with EF.
- Three ways to replace the Rep/UoW pattern with EF Core code.
- How to make your EF Core database access code easy to find and refactor.
- A discussion on unit testing EF Core code.

I'm going assume you are familiar with C# code and either Entity Framework 6 (EF6.x) or Entity Framework Core library. I do talk specifically about EF Core, but most of the article is also relevant to EF6.x.

Setting the scene

In 2013 I started work on a large web application specifically for healthcare modelling. I used ASP.NET MVC4 and EF 5, which had just come out and supported SQL Spatial types which handles geographic data. At that time the prevalent database access pattern was a Rep/UoW pattern – see this article written by Microsoft in 2013 on database access using EF Core and the Rep/UoW pattern.

I built my application using Rep/UoW, but found it a real pain point during development. I was constantly having to 'tweak' the repository code to fix little problems, and each 'tweak' could break something else! It was this that made me research into how to better implement my database access code.

Coming more up to date, I was contracted by a start-up company at the end of 2017 to help with a performance issue with their EF6.x application. The main part of the performance issue turned out to be due to lazy loading, which was needed because the application used the Rep/UoW pattern.

What people are saying against the repository pattern

In researching as part of my review of the current Spatial Modeller™ design I found some blog posts that make a compelling case for ditching the repository. The most cogent and well thought-out post of this kind is 'Repositories On Top UnitOfWork Are Not a Good Idea'. Rob Conery's main point is that the Rep/UoW just duplicates what Entity Framework (EF) DbContext give you anyway, so why hide a perfectly good framework behind a façade that adds no value. What Rob calls 'this over-abstraction silliness'.

Another blog is 'Why Entity Framework renders the Repository pattern obsolete'. In this Isaac Abraham adds that repository doesn't make testing any easier, which is one thing it was supposed to do. This is even truer with EF Core, as you will see later.

So, are they right?

My views on the pros and cons of repository/unit-of-work pattern

Let me try and review the pros/cons of the Rep/UoW pattern in as even-handed way as I can. Here are my views.

The good parts of the Rep/UoW pattern (best first)

1. **Isolate your database code**: The big plus of a repository pattern is that you know where all your database access code is. Also, you normally split your repository into sections, like the Catalogue Repository, the Order Processing Repository, etc which makes it easy to find the code a specific query that has a bug or needs performance tuning. That is definitely a big plus.
2. **Aggregation**: Domain Driven-Design (DDD) is a way to design systems, and it suggests that you have a root entity, with other associated entities grouped to it. The example I use in my book "Entity Framework Core in Action" is a Book entity wit a collection of Review entities. The reviews only make sense when linked to a

cleverness and use T-SQL. This type of access should be hidden from higher layers, yet easy to find to help with maintenance/refactoring. I should point out that Rob Conery's post Command/Query Objects can also handle this.

4. **Easy to mock/test:** It is easy to mock an individual repository, which makes unit testing code that accesses the database easier. This was true some years ago, but nowadays this there are other ways around this problem, which I will describe later.

You will note that I haven't listed "replacement of EF Core with another database access library". This is one of the ideas behind the Rep/UoW, but my view it's a misconception, because a) it's very hard replace a database access library, and b) are you really going to swap such a key library in your application? You wouldn't put up a facade around ASP.NET or React.js, so why do that to your database access library?

The bad parts of the Rep/UoW pattern (worst first)

The first three items are all around performance. I'm not saying you can't write an efficient Rep/UoW's, but its hard work and I see many implementations that have built-in performance issues (including Microsoft's old Rep/UoW's implementation). Here is my list of the bad issues I find with the Rep/UoW pattern:

1. **Performance – handling relationships:** A repository normally returns a IEnumerable /IQueryable result of one type, for instance in the Microsoft example, a Student entity class. Say you want to show information from a relationship that the Student has, such as their address? In that case the easiest way in a repository is to use lazy loading to read the students' address entity in, and I see people doing this a lot. The problem is lazy loading causes a separate round-trip to the database for every relationship that it loads, which is slower than combining all your database accesses into one database round-trip. (The alternative is to have multiple query methods with different returns, but that makes your repository very large and cumbersome – see point 4).
2. **Data not in the required format:** Because the repository assembly is normally created near to the database assembly the data returned might not be in the exact format the service or user needs. You might be able to adapt the repository output, but its a second stage you have to write. I think it is much better to form your query closer to the front-end and include any adaption of the data you need (see more on this in the section "Service Layer" in one of my articles).

tures it will only update the properties that have changed.

4. **Too generic**: The allure of the Rep/UoW comes from the view that you can write one, generic repository then you use that to build all your sub-repositories, for instance Catalogue Repository, Order Processing Repository, etc. That should minimise the code you need to write, but my experience is that a generic repository works at the beginning, but as things get more complex you end up having to add more and more code to each individual repository.

“The more reusable the code is, the less usable it is.” Neil Ford, from the book Building evolutionary architectures.

To sum up the bad parts – a Rep/UoW hides EF Core, which means you can’t use EF Core’s features to produce simple, but efficient database access code.

How to use EF Core, but still benefit from the good parts of the Rep/UoW pattern

In the previous “good parts” section I listed **isolation**, **aggregation**, **hiding**, and **unit testing,** which a Rep/UoW did well. In this section I’m going to talk about a number different software patterns and practices which, when combined with a good architectural design, provides the same isolation, aggregation, etc. features when you are using EF Core directly.

I will explain each one and then pull them together in a layered software architecture.

1. Query objects: a way to isolate and hide database read code.

Database accessed can be broken down into four types: Create, Read, Update and Delete – known as CRUD. For me the read part, known as a query in EF Core, are often the hardest to build and performance tune. Many applications rely on good, fast queries such as, a list of products to buy, a list of things to do, and so on. The answer that people have come up with is query objects.

I first came across them in 2013 in Rob Conery’s article (mentioned earlier), where h refers to Command/Query Objects. Also, Jimmy Bogard produced post in 2012 called

The listing below gives a simple example of a query object that can select the order in which a list of integers is sorted.

```
public static class MyLinqExtension
{
    public static IQueryable<int> MyOrder
        (this IQueryable<int> queryable, bool ascending)
    {
        return ascending
            ? queryable.OrderBy(num => num)
            : queryable.OrderByDescending(num => num);
    }
}
```

And here is an example of how the MyOrder query object is called.

```
var numsQ = new[] { 1, 5, 4, 2, 3 }.AsQueryable();

var result = numsQ
    .MyOrder(true)
    .Where(x => x > 3)
    .ToArray();
```

The MyOrder query object works because the IQueryable type holds a list of commands, which are executed when I apply the ToArray method. In my simple example I'm not using a database, but if we replaced the numsQ variable with a DbSet<T> property from the application's DbContext, then the commands in the IQueryable<T> type would be converted to database commands.

Because the IQueryable<T> type isn't executed until the end, you can chain multiple query objects together. Let me give you a more complex example of a database query from my book "Entity Framework Core in Action". In the code below uses four query objects chained together to select, sort, filter and page the data on some books. You can see this in action on the live site http://efcoreinaction.com/.

```
public IQueryable<BookListDto> SortFilterPage
    (SortFilterPageOptions options)
{
    var booksQuery = _context.Books
        .AsNoTracking()
        .MapBookToDto()
        .OrderBooksBy(options.OrderByOptions)
        .FilterBooksBy(options.FilterBy,
                       options.FilterValue);

    options.SetupRestOfDto(booksQuery);

    return booksQuery.Page(options.PageNum-1,
                           options.PageSize);
}
```

turns IQueryable<T> too.

2. Approaches to handling Create, Update and Delete database accesses

The query objects handle the read part of the CRUD, but what about the Create, Update and Delete parts, where you write to the database? I'm going to show you two approaches to running a CUD action: direct use of EF Core commands, and using DDD methods in the entity class. Let's look at very simple example of an Update: adding a review in my book app (see http://efcoreinaction.com/).

> *Note: If you want to try adding a review you can do that. There is a GitHub repo that goes with my book at https://github.com/JonPSmith/EfCoreInAction. To run the ASP.NET Core application then a) clone the repo, select branch Chapter05 (every chapter has a branch) and run the application locally. You will see an Admin button appear next to each book, with a few CUD commands.*

Option 1 – direct use of EF Core commands

The most obvious approach is to use EF Core methods to do the update of the database. Here is a method that would add a new review to a book, with the review information provided by the user. Note: the ReviewDto is a class that holds the information returned by the user after they have filled in the review information.

```
1  public Book AddReviewToBook(ReviewDto dto)
2  {
3      var book = _context.Books
4          .Include(r => r.Reviews)
5          .Single(k => k.BookId == dto.BookId);
6      var newReview = new Review(dto.numStars, dto.comment, dto
7      book.Reviews.Add(newReview);
8      _context.SaveChanges();
9      return book;
10 }
```

The steps are:

- **Lines 3 to 5**: load specific book, defined by the BookId in the review input, with i list of reviews
- **Line 6 to 7**: Create a new review and add it to the book's list of reviews

> *constructor that takes the application's DbContext, which is injected by dependecy injection (DI). The injected value is stored in the private field _context, which the AddReviewToBook method can use to access the database.*

This will add the new review to the database. It works, but there is another way to build this using a more DDD approach.

Option 2 – DDD-styled entity classes

EF Core offers us a new place to add your update code to – inside the entity class. EF Core has a feature called backing fields that makes building DDD entities possible. Backing fields allow you to control access to any relationship. This wasn't really possible in EF6.x.

DDD talks about **aggregation** (mentioned earlier), and that all aggregates should only be altered via a method in the root entity, which I refer to as *access methods*. In DDD terms the reviews are an aggregate of the book entity, so we should add a review via an access method called AddReview in the Book entity class. This changes the code above to a method in the Book entity, here

```
public Book AddReviewToBook(ReviewDto dto)
{
    var book = _context.Find<Book>(dto.BookId);
    book.AddReview(dto.numStars, dto.comment,
         dto.voterName, _context);
    _context.SaveChanges();
    return book;
}
```

The AddReview access method in the Book entity class would look like this:

```
public class Book
{
    private HashSet<Review> _reviews;
    public IEnumerable<Review> Reviews => _reviews?.ToList();
    //...other properties left out

    //...constructors left out

    public void AddReview(int numStars, string comment,
        string voterName, DbContext context = null)
    {
        if (_reviews != null)
        {
            _reviews.Add(new Review(numStars, comment, voterN
            {
                context.Add(new Review(numStars, comment, voterNa
            }
            else
            {
                throw new InvalidOperationException("Could not ad
            }
        }
        //... other access methods left out
```

This method is more sophisticated because it can handle two different cases: one where the Reviews have been loaded and one where it hasn't. But it is faster than the original case, as it uses a "create relationship via foreign keys" approach if the Reviews are not already loaded.

Because the access method code is inside the entity class it can be more complex if need be, because its going to be the ONLY version of that code you need to write (DRY). In option 1 you could have the same code repeated in different places wherever you need to update the Book's review collection.

> *NOTE: I have written an article called "Creating Domain-Driven Design entity classes with Entity Framework Core" all about DDD-styled entity classes. That has a much more detailed look at this topic. I have also updated my article on how to write business logic with EF Core to use the same DDD-styled entity classes.*

Why doesn't the method in the entity class called SaveChanges? In option 1 a single method contained all the parts: a) load entity, b) update entity, c) call SaveChanges to update the database. I could do that because I knew it was being called by a web action, and that was all I wanted to do.
With DDD entity methods you can't call SaveChanges in the entity method because you can't be sure the operation has finished. For instance, if you were loading a book from a backup you might want to create the book, add the authors, add any reviews, and then call SaveChanges so that everything is saved together.

Option 3: the GenericServices library

GenericServices).

These libraries don't really implement a repository pattern, but act as an adapter pattern between the entity classes and the actual data that the front-end needs. I have used the original, EF6.x, GenericServices and it has saved me months of writing boring front-end code. The new EfCore.GenericServices is even better, as it can work with both standard styled entity classes and DDD-styled entity classes.

Which option is best?

Option 1 (direct EF Core code) has the least code to write, but there is a possibility of duplication, because different parts of the application may want to apply CUD commands to an entity. For instance, you might have an update via the ServiceLayer when the user via changes things, but external API might not go through the ServiceLayer, so you have to repeat the CUD code.

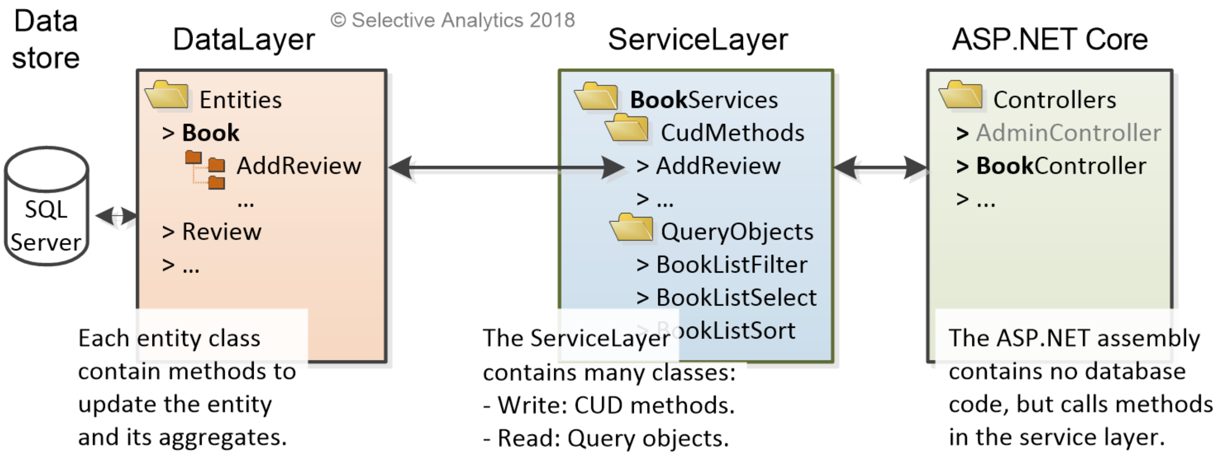
Option 2 (DDD-styled entity classes) places the crucial update part inside the entity class, so the code going to be available to anyone who can get an entity instance. In fact, because the DDD-styled entity class "locks down" access to properties and collections everybody HAS to use the Book entity's AddReview access method if they want to update the Reviews collection. For many reasons this is the approach I want to use in future applications (see my article for a discussion on the pros and cons). The (slight) down side is its it needs a separate load/Save part, which means more code.

Option 3 (the EF6.x or EF Core GenericServices library) is my preferred approach, especially now I have build the EfCore.GenericServices version that handles DDD-styled entity classes. As you will see in the article about EfCore.GenericServices, this library drastically reduces the code you need to write in your web/mobile/desktop application. Of course, you still need to access the database in your business logic, but that is another story.

Organising your CRUD code

One good thing about the Rep/UoW pattern is it keeps all your data access code in one place. When swapping to using EF Core directly, then you could put your data a cess code anywhere, but that makes it hard for you or other team members to find i semblies shown (I have left out the business logic, and in a hexagonal architecture you will have more assemblies). The three assemblies shown are:

- **ASP.NET Core**: This is the presentation layer, either providing HTML pages and/or a web API. This no database access code but relies on the various methods in the ServiceLayer and BusinessLayers.
- **ServiceLayer**: This contains the database access code, both the query objects and the Create, Update and Delete methods. The service layer uses an adapter pattern and command pattern to link the data layer and the ASP.NET Core (presentation) layer. (see this section from one of my articles about the service layer).
- **DataLayer**: This contains the application's DbContext and the entity classes. The DDD-styled entity classes then contain access methods to allow the root entity and its aggregates to be changed.



> *NOTE: The library GenericServices (EF6.x) and EfCore.GenericServices (EF Core) mentioned earlier are, in effect, a library that provides ServiceLayer features, i.e. that act as an adapter pattern and command pattern between the DataLayer and your web/mobile/desktop application.*

The point I want make from this figure is, by using different assemblies, a simple naming standard (see the word **Book** in bold in the figure) and folders, you can build an application where your database code is isolated and it's easy to find. As your application grows this can be critical.

Unit testing methods that use EF Core

Thankfully things have moved on with EF Core and you can simulate the database with an in-memory database. In-memory databases are quicker to create and have a default start point (i.e. empty), so it's much easier to write tests against. See my article, Using in-memory databases for unit testing EF Core applications, for a detailed look at how you can do that, plus an NuGet package called EfCore.TestSupport that provide methods to make writing EF Core unit tests quicker to write.

Conclusions

My last project that used the Rep/UoW pattern was back in 2013, and I have never used a Rep/UoW pattern again since then. I have tried a few approaches, a custom library called GenericServices with EF6.x, and now a more standard query object and DDD entity methods with EF Core. They are easy to write and normally perform well, but if they are slow it's easy to find and performance tune individual database accesses.

In the book I wrote for Manning Publications I have a chapter where I performance tune a ASP.NET Core application that "sells" books. That process used query objects and DDD entity methods and shows that it can produce great performing database accesses (see my article Entity Framework Core performance tuning – a worked example for a summary).

My own work follows the query object for reads and DDD-styled entity classes with their access methods for CUD and business logic. I do need to use these in a proper application to really know if they work, but its promising. Wtach this space for more in DDD-styled entity classes, architecture that benefit from that, and maybe a new library :).

Happy coding!

Jon P Smith .NET Core, Entity Framework

✉ Subscribe ▾ Login

Join the discussion

B *I* U̲ ~~S~~ {} [+]

39 COMMENTS Newest ▾

Mike 4 months ago

Regarding the anti-repo argument “Many Rep/UoW implementations try to hide EF Core, and in doing so don’t make use of all its features…”, isn’t it actually a strength of the repo pattern to be able to do performance tweaks such as update/remove logic changes in the repo, without changing the rest of the codes?

Let’s say a previous .Update() or .Remove() method in the repo was using the less efficient EF methods that result in unnecessary queries/operations. Later with an EF update, the programmer decides to make it more efficient by manipulating EntityState etc. He only has to update this in the repos.

If those operations were performed directly against the DbContext, then all the codes that performed these operations have to be changed to use the new logic.

Am I missing something here?

+ 1 — Reply

Jon P Smith 4 months ago

Author

Reply to Mike

Hi Mike,

detect changes system. I'm not saying all repositories are like that, but some are.

In fact I'm not against the repository pattern – I use it in some of my business logic and I have used it for Azure blob access etc. But for EF Core queries I find a generic Repository<T> breaks the separation of concerns pattern. By that I mean that I can't easily performance tune a query for say a Book class in a generic repository. I found that exact issue when I used a Repository<T> many years ago.

Writing straightforward EF Core queries is really easy so a repository doesn't add much. Instead I have a library called EfCore.GenericService which handles simple queries using selects (using a Select query is often a bit more efficient). But for EF Core queries that need performance tuned I hand-code each query.

If you have found a good way to use the repository pattern with EF Core then that's great. I have found another way that works for me and I thought others mike like to read about it.

+ 0 − Reply

David Ramirez 5 months ago

Hi Jon!

I'm working on a desktop application that uses WinForms and Entity Framework 6 and I'm wondering if it would be possible to use the same layered approach you've taken?.

As you know WinForms doesn't implement a pattern such as MVC so everything is in the same place (UI controls, database access, validation, business rules, etc.). So let's say that I split my application the way you did, should I use one DbContext per service?. Should I call the service layer directly from the WinForms layer?. Can I share Entities between layers or should I use different entities per layer?

+ 0 − Reply

Jon P Smith 5 months ago

MVC or Core is the same – a HTTP request calls a method, so the layered approach should work, but don't think you have dependency injection which makes it a bit harder, especially about getting a "DbContext per service" your talked about.

Have a look my article called Six ways to build better Entity Framework (Core and EF6) applications which might help.

Last edited 5 months ago by Jon P Smith

+ 0 − Reply

jnrohde 5 months ago

Great post. I am confused about some of the organisation though.

In the "Option 2 – DDD methods in entity classes"-section I take it the AddReviewToBook(ReviewDto dto) method is from the service layer, right? But it uses the context, which should live in the Data Layer. What am I missing?

+ 0 − Reply

Jon P Smith 5 months ago

Reply to jnrohde

Updated answer.

+ 0 − Reply

Jon P Smith 5 months ago

Reply to jnrohde

Hi jnrohde,

Sorry, to save space I only showed the AddReviewToBook method, but it's in a class called AddReviewService which has the application's DbContext injected via dependency injection. The application's DbContext in a ASP.NET Core applictaion is always created by DI, so that you get the correctly scoped version.

https://github.com/JonPSmith/EfCoreInAction/ and select the Chapter05 branch.

I hope that helps, and I am glad you found the article helpful.

+ 0 − Reply

Even Schjølberg 5 months ago

Hi Jon!
I'm currently using a repository/uow pattern with EF Core and find your work very interesting.
I'm looking to implement something like option 2 in this article.
I think it would help with my scenarios were I want to access business parameters from db from inside entity class methods.

I'm wondering how you would approach my problem:
I want to save a new order and need to get the next available order number from a counter in another entity.
Problem is, after getting the next available order number the counter will have to be incremented and persisted to db.
Now I don't want to call SaveChanges from within the entity class method and might want to wrap it all in a transaction so that there won't be any concurrency issues.

Any advice and thoughts as to how to go about this would be highly appreciated.

+ 0 − Reply

Jon P Smith 5 months ago

Reply to Even Schjølberg

Hi Even,

OK, I think there two parts to what you are talking about: a) how to organise the code inside the DDD-styled entity and b) how to access a database-generated value before the whole saga has finished.

Anything else, like dealing with things related to steps or stages in a database update I consider business logic. I may not have explained that well as it is a bit subtle, but in your “order” entity and “another” entity your wording makes me think that there needs to be some business logic outside the two entities to handle that, maybe with transactions. That, for me, is definitely outside the code inside the DDD-styled entities.

The other part, accessing a database-generated value inside a saga, is a more known issue. The typical way to do that is with a transaction where you call SaveChanges in the middle of the transaction, which gives you access to the database-generated value. The other, not so obvious approach is using EF Core’s clever add/update code. For instance, if your “another” entity had a navigational link to your “Order” entity (e.g. a property of type “Order”) when you save an “Order” with an “another” entity, then EF Core will know they should be linked and will copy the “Order’s” primary key into the “another” entity’s foreign key for you.

My comments feel a bit too complicated – sorry about that :(. The problem is what you are asking about is complicated! I hope my words help.

PS. Make sure to look at https://www.thereformedprogrammer.net/creating-domain-driven-design-entity-classes-with-entity-framework-core/ article, and maybe my https://www.thereformedprogrammer.net/architecture-of-business-layer-working-with-entity-framework-core-and-v6-revisited/ article.

+ 0 − Reply

Even Schjølberg 5 months ago

Reply to Jon P Smith

Thanks for your reply. You clarifying what the code inside the entities are supposed to do is very helpful to make sure I understand you correctly. I

I have a service with a method to place an order.
The caller, let's say a controller, will have an instance of this service with a scoped context. And will call SaveChanges to persist.
Problem is, inside the service method I want to get, increment and persist the order counter.
Naturally I could wrap everything in a transaction from the controller, but I don't think the controller should have this responsibility?

Simplified: Can I (and should I) create a transaction inside the service method, save changes there, and commit it only if SaveChanges is called successfully from the calling controller?

I'm sorry If I fail to explain my problem good enough.

+ 0 − Reply

Jon P Smith 2 years ago

Reply to Even Schjølberg

I'm glad some of what I wrote was useful 🙂
Here is how I handle issues that need transactions in my code.

If you have read my article on business logic (https://www.thereformedprogrammer.net/architecture-of-business-layer-working-with-entity-framework-core-and-v6-revisited/) then you will see that I create a method to handle my business logic in a separate layer. If I have two methods where the second method need database values I would use a transaction. Typically have some code running the ServiceLayer that handles the calling of two business logic method within a transaction (see this article https://www.thereformedprogrammer.net/architecture-of-business-layer-calling-multiple-business-methods-in-one-http-request/).

This approach might seem like a lot of code in extra layers, but for anything other than a very business logic, in chapter 4 of my book (http://bit.ly/2m8KRAZ). I also have a Git repo with the code in the book, and you might like to look at the example that needs a transaction.

– https://github.com/JonPSmith/EfCoreInAction/blob/Chapter04/BizLogic/Orders/Concrete/PlaceOrderPart1.cs
– https://github.com/JonPSmith/EfCoreInAction/blob/Chapter04/BizLogic/Orders/Concrete/PlaceOrderPart2.cs

I call these two business methods from something I call a 'BizRunner' that runs in the ServiceLayer. Here is a link to the BizRunner that handles two methods in a transaction. https://github.com/JonPSmith/EfCoreInAction/blob/Chapter04/ServiceLayer/BizRunners/RunnerTransact2WriteDb.cs.

+ 0 − Reply

Ethyl Seria 5 months ago

Good day sir,

Thank you for this helpful information. Now I want to move on using Layered Architecture just like you've mentioned. I'm a bit confuse now because this blog (https://programmingwithmosh.com/asp-net/layered-architecture/) offers other way of separating things using Layered Architecture.

So, what I did was like this:

-BusinessLogicLayer
-DataAccessLayer
-ServiceLayer (which is the ASP.NET Core API project)

From your sample above, ServiceLayer == BusinessLogicLayer and from his blog, ServiceLayer == API Service. I'm a bit confuse now sir what exactly should I need follow.

Reply to Ethyl Seria

Hi Ethyl,

I think Mosh Hamedani's article is saying that same things as I am suggesting, but from a different persective. One of my detailed article on a layered architecture, with the reasons why it is useful, then have a look at point 1 in https://www.thereformedprogrammer.net/six-ways-to-build-better-entity-framework-core-and-ef6-applications/

I should also say there are lots of ways to organise your software. Here is one of many articles on different architectural approaches https://techbeacon.com/top-5-software-architecture-patterns-how-make-right-choice

+ 0 − Reply

goldman1337 5 months ago

good day jon, i'm reasking what i've asked on twitter (should've realized it was easier asking here)
while looking at how the service layer looks, i noticed you have CUD methods and query objects. While i understand the need for both to keep from repeating code, i keep wondering if having both isn't basically just a different way to implement a repository (instead of one class with querys+CUD, a bunch of different classes separated).

also, wanted to take the opportunity to ask if the CUD methods are saving all CUD (even simple add/delete) or just specific, more complex methods (like, for example, a method that gets all users and updates a field, that might be repeated in 2 different services).

i'm sorry for the possible crappy english, it's not my main laguage and thank you

+ 0 − Reply

Jon P Smith 5 months ago

Reply to goldman1337

people trying to shortcut things, like losing lazy loading to handle loading relationships. So, my response is NOT to create a big repository, but to create a series of focused methods in the entity class for Create/Update/Delete and separate query objects for read. It must easier to build the exact thing you need than try to make a repository that does everything. Did you see the quote from Neil Ford – "The more reusable the code is, the less usable it is."

The only problem I have then is calling these methods, but I built a library EfCore.GenericServices that does that.

On your question about the CUD methods. I use ctors/static factories for create and method calls for updates. But I do have a hybrid form where I can directly update 'simple' properties that have no business logic (see https://www.thereformedprogrammer.net/pragmatic-domain-driven-design-supporting-json-patch-in-entity-framework-core/ where I talk about support JSON Patch).

You might also like to see an article I released yesterday which compare/contrasts three differe ways of applying DDD to EF Core (see https://www.thereformedprogrammer.net/three-approaches-to-domain-driven-design-with-entity-framework-core/)

+ 0 − Reply

Pedro Faustino 5 months ago

Hello Jon,

Following your idea of having query objects, I wonder how do you compose an OR query for filtering? Imagine you have 2 query object methods (methodA and methodB) that are reused across your application and now you have a new query where you need to have a filter composed by methodA OR methodB.

One idea that I've for tackling this problem is to apply the specification pattern, which would allow us to create query

Jon P Smith 5 months ago

Reply to Pedro Faustino

Hi Pedro,

You can add LINQ methods, but either the LINQ is pre-compiled or you have to build the LINQ manually (hard work!). When I have needed to dynamically build AND/OR queries and I use the PredicateBuilder – see http://www.albahari.com/nutshell/predicatebuilder.aspx

+ 0 − Reply

Pedro Faustino 2 years ago

Reply to Jon P Smith

Seems like a good idea. On my project I'm using this library https://github.com/gnaeus/EntityFramework.CommonTools#ef-specification, which abstracts the implementation of the PredicateBuilder with the implementation of the specification pattern.

+ 0 − Reply

Pedro Faustino 2 years ago

Reply to Jon P Smith

Could you also provide an example of how you would encapsulate a query that needs raw SQL?

+ 0 − Reply

michael lang 5 months ago

This is not a new argument. It's actually a very old argument.

From an architectural perspective Entity Framework is essentially the database, its NOT a business layer, its a very direct representation of the database schema. Sure you can use EF directly from the top tier of your application, whether that be an API controller or a GUI. But essentially a short road to disaster, and a lot of code duplication, extra work and bugs.

Repositories are simply a place to define and encapsulate business rules, in classic application architecture, they are the middle tier or business layer, sure by all means do away with them and replace them with something not called a repository, but don't fool yourself into thinking there's no use for them.

+ 1 − Reply

Jon P Smith 5 months ago

Reply to michael lang

Hi Michael,

This article is dealing with CRUD. I have a different approach for business logic, which is what you are taking about – see Architecture of Business Layer working with Entity Framework (Core and v6). In that I use per-business logic repository.

I have added links to other related articles like this to the start of this article. I hope that helps.

+ 0 − Reply

toadicus 5 months ago

I'll be honest. While there are some interesting points made in this article, I'm rather inclined to disagree with the result. I'm a software architect and have been for a few years. And I use a form of the Repository/UoW pattern you describe quite heavily; it is an extremely useful tool.

Let me explain; first, I have multiple repository types. Yes, there is a repository type that knows how to store a DDD aggregate into a database, but the concept of "repository" doesn't just cover databases. A repository is something that knows how to write to and from a data store, regardless of what that data store is. For example, I have a repository that knows how to write to a restful API for a partner company. I have many repositories that know how to store and read from Redis cache implementations. I don't want

Chain-Of-Responsibility pattern with a single interface for all repositories within a stack. For example, if I have a ThingDomainModel and I want to store it in a repository that is designed for that purpose, I would call an IThingRepository that had a save method that would accept the ThingDomainModel and would know what to do with it. My service layer just has to call the repository to save, or to get, or whatever. I use an IOC to wire up the Chain-Of-Responsibility, so any object that gets instantiated with an IThingRepository gets the top repository of the CoR, usually the Cache repository. That cache repository has a "Next" property (which is an IThingRepository) and it knows that it needs to delegate to the "Next" repository if it doesn't have that object in its cache storage. The database repository is usually but not always the last repository, and it knows how to write its data to its data store and return objects it contains.

This might seem like a lot of overengineering, but it's not. The beauty of this is the easy transition to microservices where each DDD stack might have its own datastore. Your article doesn't actually deal with this significant issue; eventually you'll hit a point (if you're successful) where your database must be specialized and broken up. Some data is better stored in a NoSQL environment, some in a relational database, some maybe can't be stored onsite (such as the API-based repo I mentioned).

One major red flag for me is the description you gave of a DDD Aggregate from Martin Fowler. A DataModel is different than a DomainModel, sure, but both should essentially be data objects; Martin Fowler didn't say anywhere in his article that the base entity should have a "AddReview" method or any other methods, he merely said that they should be handled as a single unit.

That's where Repository shines. Yes, there are some complexities, but in the end, I don't want my service to have to worry about how to store its DomainModel. I shouldn't have to be concerned with whether or not it's from companies that have realized that they’ve been coded into a corner and don’t know how to escape. If your company is really successful and you need to scale, the model you’re suggesting will not. It leads to monolithic code.

Never have one repository call another. It is forbidden. Yes, that introduces “querying” issues, but there are ways around that as well. Your DDD domain service also can’t know about another stack’s repositories. I’m sure there are some rolling of eyes at this, but wait until you’re trying to pull your monolith apart so that you can scale one particular service in a microservice, and you’ll understand quite a bit better. A composite service on top of your domain stacks can have access to multiple domain services, and NEVER a repository.

Keep it simple, keep your code doing what it’s supposed to do (A POCO is NOT a Data Layer in DDD, Single Responsibility screams at you). That’s the core of a principled software architecture approach, and DDD is simply a higher order of that same concept. A domain of code deals with a particular domain of concern ONLY; an Order domain knows about Orders. It doesn’t know about Products, for example. You can reference a product in the Order domain by Id if you want, but cut the foreign keys between domains! They are a pure killer when you try to scale. Just rehydrate them at an aggregate service that knows how to reassemble your super-composite (if you really have to. I find it to be almost useless to return the kitchen sink.)

I’m dealing with this structure in my current job as I try to dismantle the monolith they built, which is extremely similar to what you’ve described (only it uses nHibernate instead of EF), and I recently left a company that was a multi-million dollar company, at one time making over 20 million per year, but it slowly killed itself by using the Entity-IsA-Repository-IsA-DataService-IsA-BIService mentality… customers that had had been happy noticed article to fixate on, but I did. That said, if your projects remain small you might not ever notice… but when you have to scale, there will be significant problems.

Oh my gosh, not intending to offend either. lol

Oh, am I talkin' too loud?
Sometimes I get over excited, shoot off at the mouth
I never had a group of friends before
I promise that I'll make y'all proud

+ 5 − Reply

Jon P Smith 5 months ago

Reply to toadicus

Hi toadicus,

Wow, thanks for sharing that – I really found it very useful. I did have to print it out and read it to think it through. I really appreciate your insights and they help me on my journey and thank you for taking the time to do that.

The first thing to say is my opinion is "there are no silver bullets in software". There are pros/cons in every approach and it's the skill of the developer to pick the correct approach that fits the business need. I have been contracted by several startups/small companies that need a quick development that can evolve over time. I believe my approach and libraries work well there.

I appreciate your warnings about evolving as the application grows as that is one of the "sweet spots" I work in. I believe my architecture/libraries work well in that space, but I'm still thinking and learning – in fact I wrote a long article looking at different DDD approaches to make me think it through – see Three approaches to Domain-Driven Design with Entity Framework Core. Your comments are a useful review of what I am doing.

Having set down with a cup of tea (I am British) and thought about what you said here are some comments below:

footprint – if I want to move the root+aggregates it's pretty easy.
2. My "repository replacement" is my EfCore.GenericServices library for CRUD and my EfCore.GeneriBizRunner for handling business logic. I don't focus on these libraries in this article because I wanted to look at the overall "repository in EF Core" issue, but they are core to my "quick to build, easy to refactor" approach.
3. I'm a fan of DDD bounded contexts in the database, and all my libraries support multiple DbContexts mapped to the same, or different databases. Like your repositories my bounded contexts don't talk to each. It sounds like your repositories are smaller than a typical DDD bounded contexts, but maybe not.

To end I was commissioned to design and build a large ASP.NET Core system with multiple databases and multiple ASP.NET applications last year (this project created two articles – see https://www.thereformedprogrammer.net/a-better-way-to-handle-authorization-in-asp-net-core/). I used my approach and it has gone well so far (two ASP.NET apps so far). Working across bound contexts is the difficult bit – we used calls between business logic in one bounded context to business logic in the other bounded context to handle that.

+ 3 — Reply

Fred . 5 months ago

This code:

```
private HashSet<Review> _reviews;
public IEnumerable<Review> Reviews =>
_reviews?.ToList()
```

I write as:

This always exposes a list even if empty, so I don't have to worry about it being null.
`AsReadOnly()` is a light-weight operation which creates a list by reference, while `ToList()` is an expensive operation which copies the list into an new list which results in higher memory usage.

+ 0 − Reply

Jon P Smith 5 months ago

Reply to Fred .

OK. Two different things here.

Firstly the initialization of the _review (or any collection navigation). Many people do that but I don't. It all about writing code that is fail-safe. The problem with initializing the review is if you forget to load that review entries, e.g. by using .Include(p => p.Reviews) and I then added a review the result would be that all existing reviews would be silently deleted and you would only have the one review you added. By making it null this can't happen, and your code would fail with a null reference.

The second one is another trick I learnt from Arthur Vickers on the EF Core team. You uses .ToList() to creat a new list. That way if someone casts the property back to a List and adds something it won't affect the HashSet. See this old, but useful artcile by Arthur Vickers – https://blog.oneunicorn.com/2016/10/28/collection-navigation-properties-and-fields-in-ef-core-1-1/

+ 0 − Reply

Fred . 1 year ago

Reply to Jon P Smith

Thank you for your insightful comment. I've now reconsidered initializing my navigation properties that are collections.
The `.ToList()` method creates an entirely new list

0 — Reply

Fred . 5 months ago

Thank you for a great article. I felt trapped by the repository pattern because I wanted to do things such as `AsNoTracking()` and `Include()` and felt limited when using the repository pattern. The idea of a query class resonated well with me, I really liked that.

When reading articles about the repository pattern and EF Core, it often makes you seem like your choice is EF with a repository or EF without a repository while listing the advantages and disadvantages. When ditching the repository and using a query class against a a `DbContext` I once again felt limited, because DbContext does not have any interface, so it makes it difficult to exchange to something else to feed data from (yes, I know I can use SQLite in-memory).

So now I got the idea of calling the DbContext from the query class but with something in between. That thing between is not a repository (the DbContext is the repository), and since it is not a repository it can break the big rule of repositories not to expose a `IQueryable`. So I created a class (I am not sure what to call it, but I call it an adapter), which has an interface, and exposes my domain entities (I do not consider them database entities) from the DbSet.

It looks like this:

```
public interface IBookAdapter
{
    IQueryable Books { get; }

}
public class BookAdapter : IBookAdapter
{
    private readonly BookDbContext _context;
    public BookAdapter(BookDbContext context)
    {
}
```

My query class gets dependency injected with a `IBookAdapter` then queries the IQueryable books property. This way I can switch out the book adapter for another adapter which can internally work with a list that it exposes though the property as `AsQueryable()`. This way I find myself getting all the flexibility of not having a repository with the benefits of having a repository.

+ 0 − Reply

Jon P Smith 5 months ago

Reply to Fred .

Hi Fred,

I'm glad my ramblings help you along the way. I tend to unit test with real databases, but your IBookAdapter makes sense if you want to test with a dummy database. If you have seen my article on business logic (see https://www.thereformedprogrammer.net/architecture-of-business-layer-working-with-entity-framework-core-and-v6-revisited/) you will see I create min-repositories, one per business logic.

It's a pity we can't mock a DbSet, then you could mock the whole DbContext. I did try but DbSet is SO complex.

PS. I think your IBookAdapter isn't really an adapter pattern. Not sure there is a correct pattern name for that, but maybe a Facade? See https://sourcemaking.com/design_patterns/facade.

Keep on leaning and trying things. All the best!

+ 0 − Reply

Fred . 1 year ago

Reply to Jon P Smith

since DbSet implements IQueryable, so I just expose it as an IQueryable which is much easier to work with. Then I can switch out the shim for something else that implements the same interface but internally creates a list and exposes it through the IQuerable property by calling `.AsQueryable()` on the list.

+ 0 − Reply

Андей 5 months ago

are you realy? The last example about DDD principles. you have dependency on context, which is the implementation of persistant layer and should be placed in the infrastructure layer, so as I see, there is violation DDD principles, isn't?

+ 0 − Reply

Jon P Smith 5 months ago

Reply to Андей

Some think that way, but some don't. You might like to read my article "Three approaches to Domain-Driven Design with Entity Framework Core" (link here) where I compare/contrast three different approaches to CRUD and Business Logic when using DDD.

+ 0 − Reply

Андей 1 year ago

Reply to Jon P Smith

Okey, 'Clean Architecture' Robert C. Martin. He told, that domain layer with your main logic should not be dependent on any other layer, so that is even not the Ddd concept, but clean architecture concept, because you couple your domain with persistantce

+ 0 − Reply

Eric Eskildsen 5 months ago

For example, if you take a dependency on DbContext, it's unclear to the caller or implementer which of its DbSets your service actually depends on.

On the other hand, if you depend on specific repositories, the caller/implementer only has to care about the repositories' methods. If your repository only has Get methods, they don't need to be concerned with writing. So if they're writing unit tests, they only need to mock up read scenarios. If they're reimplementing with an alternative data store (say JSON or SQLite), they only need to code for reading data.

+ 0 − Reply

Jon P Smith 5 months ago

Reply to Eric Eskildsen

I kind of agree with you about the problems of bigger projects. I designed a SasS application for a client and we were into hundreds of DTOs within 3 or 4 person months (typically each CRUD command has one DTO).

You say

> *if you take a dependency on DbContext, it's unclear to the caller or implementer which of its DbSets your service actually depends on.*

That would be true, but in my case I use my EfCore.GenericServices (see https://github.com/JonPSmith/EfCore.GenericServices) library to replace repositories. This library handles 80% of my CRUD calls and because of the way it works, with different methods for Create, Read, Update or Delete and using named DTOs, it is fairly easy to see what is is going on in a Controller or elsewhere. (NOTE: I use hand-coded services and/or Query Objects for CRUD commands that my library can't do).

I do end up with a lot of DTOs and keeping them tidy is a challenge. In the SaaS application we ended up with having a systems turning into a ball of mud. I'm not sure that's a repository issue but a design/architecture issue and there are many ways to solve that. Thanks for your comments.

+ 0 − Reply

dasjestyr 5 months ago

I keep hearing this statement saying that implementing repo or UoW is redundant because EF core already does it so you shouldn't. But… what happens when I need to strip EF core out? Now that data abstraction becomes useless and has to be guttet and rewritten. With the repo pattern, I just need to re-implement it in one place and all is good. I've gotten counter arguments like "well how often are you changing out DB implementations?" The answer is "more often than you think" — I an Architect at a fortune 400 and this is normal for us. As the system evolves and we decompose large services into small services, databases often get replaced with new database or even service calls to the new services that now serve that data. Also, we sometimes find opportunities to replace SQL with NoSQL because it fits the application better. The rule of thumb is that if you down "own" the interface, then you shouldn't build abstractions on it. For example, we don't "own" DbContext because Microsoft does. MS may change it, you may need to get rid of it, it may have multiple reasons to change.

The other issue I take with it is now you have to construct those models in a way that makes sense to EF — or spend the time to do all the mapping to make it make sense to EF. So either you're letting the backing service dictate your domain model design or you're spending a bunch of extra time adapting it. Also, that means you're exposing your persistence model (the entities) to your application layer at the very least. So if you do need to change it, as mentioned earlier, those tendrils are going to be everywhere.

What I would like to see is a clean solution for preserving the UoW functionality but also keeping the layers perfectly

A lot of talk about unit testing, only to conclude that it is done using in-memory databases. In other words, if you want to unit test your business logic, bring EF Core along for the ride. That's not unit testing, that's integration testing.

+ 0 − Reply

Midix 5 months ago

Reply to robotron

I guess, in the context of the article "unit testing" was used as an umbrella term that includes different kinds of units, where, for example, entire back-end can be considered a standalone module.

+ 0 − Reply

Alex D 3 months ago

Reply to robotron

I agree that using the in-memory database provider is more integrative than unit, but unit testing can easily be done in EF.Core by mocking the DbSets. This allows to easily define unit tests without generic repositories.

+ 0 − Reply

Jon P Smith 3 months ago

Author

Reply to Alex D

Hi Alex D,

Thanks for the link to the MockQueryable library. That does look useful.

+ 1 − Reply