I am a freelance .NET Core back-end developer

Creating Domain-Driven Design entity classes with Entity Framework Core

Last Updated: December 8, 2018 | Created: March 6, 2018

This article is about how to apply a Domain-Driven Design (DDD) approach to the classes that the Entity Framework Core (EF Core) library maps to a database. This article is about why DDD is useful with a database, and how you can implement a DDD approach to data persistence classes using EF Core.

This article fits into the other articles on building business logic with EF Core. Here are some links to the other articles on this topic.

- Architecture of Business Layer working with Entity Framework (Core and v6) – revisited
- A library to run your business logic when using Entity Framework Core
- Creating Domain-Driven Design entity classes with Entity Framework Core (**this article**)
- GenericServices: A library to provide CRUD front-end services from a EF Core database
- Three approaches to Domain-Driven Design with Entity Framework Core – looks at different ways of implementing DDD in EF Core.

There are a number of benefits to using a DDD-styled entity classes, but the main one is that the DDD design moves the create/update code inside the entity class, which stops a developer from misinterpreting how to create or update that class.

The aims of this article

This article shows you how to build a DDD-styled entity class and then compares/contrasts the DDD version with the standard version. The parts are:

- Setting the scene – what DDD says about object design and persistence
- A look at what a DDD-styled entity class looks like
  - Comparing creating a new instance of a DDD-styled entity class
  - Compare altering the properties of a DDD-styled entity class
- Summarising the DDD-style I have developed in this article
- Looking at the pros and cons of this DDD-styled EF Core entity classes
- Other aspects of DDD not covered in this article
- Conclusion

This article is aimed at developers who use Microsoft's Entity Framework library, so I assume you are familiar with C# code and either Entity Framework 6 (EF6.x) or Entity Framework Core (EF Core) library. This article only applies to EF Core, as EF6.x doesn't have all the features needed to "lock down" the entity class.

Setting the scene – what DDD says about object design and persistence

Eric Evans talks about persisting objects (classes in C#) to a database, and here are a few of the points he made about what his DDD approach should do:

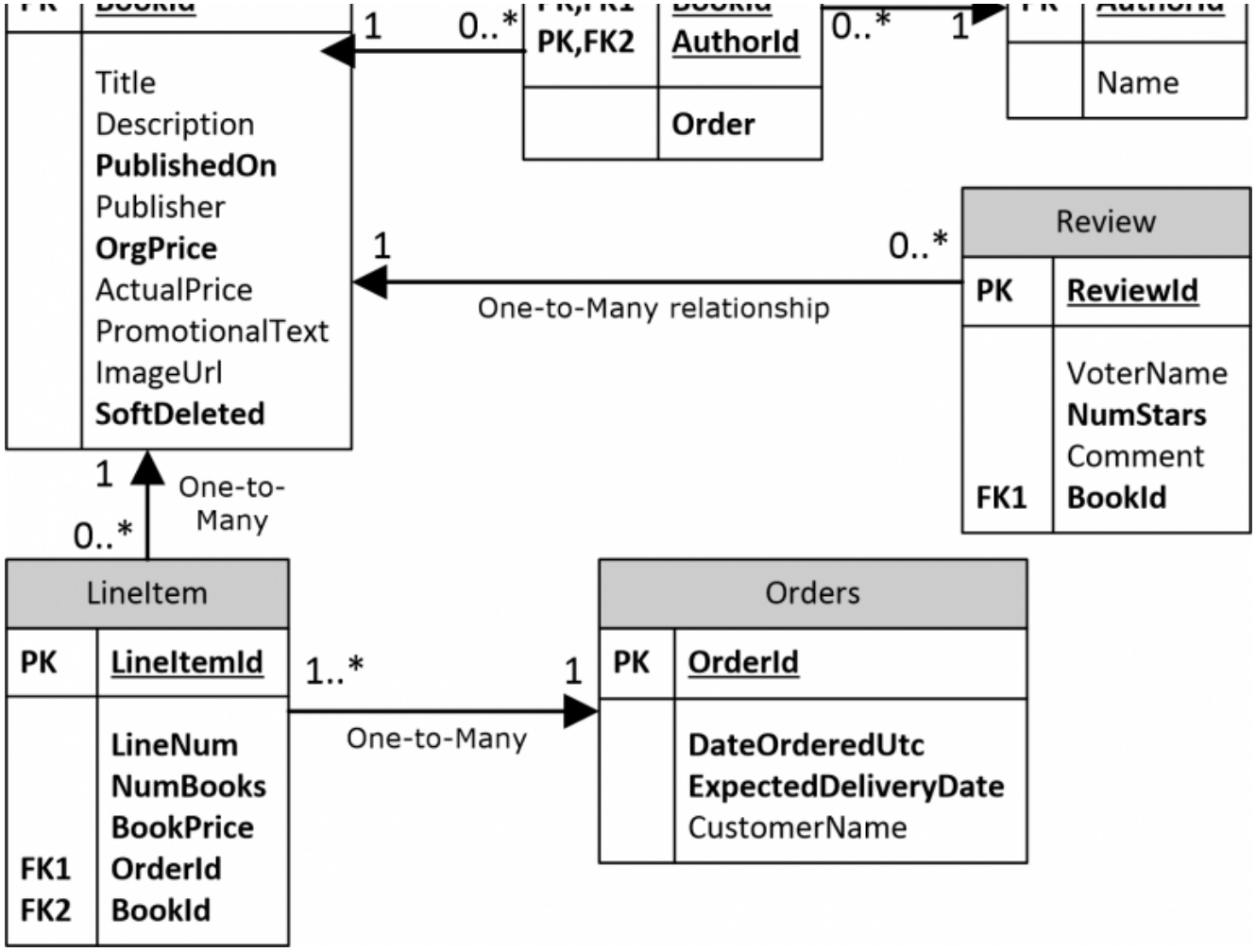
1. Present the client with a simple model for obtaining persistent objects (classes) and managing their life cycle.
2. Your entity class design should communicate design decisions about object access.
3. DDD has the concept of an *aggregate*, which is an entity that is connected to a *root* DDD says the aggregates should only by updated via the root entity.

> *NOTE: Eric talks about a* DDD repository *– I don't recommend a repository pattern over EF because EF Core itself is already implements repository/UnitOfWork pattern (see my article "Is the repository pattern useful with Entity Framework Core?" for a fuller explanation on this point).*

I was already applying some of these rules in my EF6.x applications, but I couldn't fully implement what Eric was talking about, especially around the aggregates. But now with EF Core I can, and after some experimenting I have developed a pattern for building entity classes. The rest of this article describe my solution to using a DDD approach to building entity classes, and why DDD is better than the standard approach.

The design of a DDD-style entity class

I'm going to start by showing you DDD-styled entity class and then compare how they work compared to the standard way of building entity classes with EF Core. To help me I am going to use an example database I used with a web application that sells books (think super-simple Amazon). The figure below shows the database structure.



The top four tables are about a book, its authors and any reviews that book has, which is the part that this article looks at. I included the two tables at the bottom, as they are used in the business logic example, covered here.

I now create a Book entity class, which has relational links to the Review table and the Authors table, via the many-to-many BookAuthor linking table. The following code shows you the main structure of the DDD-styled Book entity class (you will see the actual code for the constructor an methods later in this article).

```
public class Book
{
    public const int PromotionalTextLength = 200;

    public int BookId { get; private set; }
    //… all other properties have a private set

    //These are the DDD aggregate propties: Reviews and AuthorLinks
    public IEnumerable<Review> Reviews => _reviews?.ToList();
    public IEnumerable<BookAuthor> AuthorsLink => _authorsLink?.ToList();

    //private, parameterless constructor used by EF Core
    private Book() { }

    //public constructor available to developer to create a new book
    public Book(string title, string description, DateTime publishedOn,
        string publisher, decimal price, string imageUrl, ICollection<Author> authors)
    {
        //code left out
    }

    //now the methods to update the book's properties
    public void UpdatePublishedOn(DateTime newDate)…
    public IGenericErrorHandler AddPromotion(decimal newPrice, string promotionalText)…
    public void RemovePromotion()…

    //now the methods to update the book's aggregates
    public void AddReview(int numStars, string comment, string voterName, DbContext context)…
    public void RemoveReview(Review review)…
}
```

- Line 9 and 10: Both of the one-to-many relationships (these are the aggregates that DDD talks about) are available only as a IEnumerable<T> property. This means you cannot add/remove items from the collection – you have use the access methods provided by the Book class.
- Line 13: EF Core needs a parameterless constructor, but it can have a private access modifier. That means no other code cannot create an instance via this parameterless constructor.
- Lines 16 to 20: The only way you can create an instance of the Book class is via its public constructor. This requires you to provide all the information needed to create a valid book.
- Lines 23 to 25: These three lines are the methods that allow a developer to alter the existing book's value properties.
- Lines 28 to 29: These methods allow a developer to alter the existing book's relational properties – known as aggregate by DDD.

The methods listed in lines 23 to 29 are referred to as *access methods* in this article. These access methods are the only way to change the properties and relationships inside the entity.

The net effect of all this is the class Book is "locked down", so the only way to create or alter the class is via specific constructor(s) and appropriately named access methods. This contrasts with the standard way of creating/updating an entity class in EF Core, that is, create an default entity with its public parameterless constructor, and set values/relationships via its public setter on all the class's properties. The next question is – why is this better?

Comparing creating a new instance of a DDD-styled entity class

Let's start by considering the code to create a new instance of Book entity using first the standard-style and then with the DDD-styled class. The code will take some data input from json file containing information on various

a. Standard entity class with public constructor and properties

```
var price = (decimal) (bookInfoJson.saleInfoListPriceAmount ?? DefaultBookPrice)
var book = new Book
{
    Title = bookInfoJson.title,
    Description = bookInfoJson.description,
    PublishedOn = DecodePubishDate(bookInfoJson.publishedDate),
    Publisher = bookInfoJson.publisher,
    OrgPrice = price,
    ActualPrice = price,
    ImageUrl = bookInfoJson.imageLinksThumbnail
};

byte i = 0;
book.AuthorsLink = new List<BookAuthor>();
foreach (var author in bookInfoJson.authors)
{
    book.AuthorsLink.Add(new BookAuthor
    {
        Book = book, Author = authorDict[author], Order = i++
    });
}
```

b. DDD-styled entity class, with specific public constructor with parameters

Now, here is the same code, but using the DDD-styled entity class with its public constructor.

```
var authors = bookInfoJson.authors.Select(x => authorDict[x]).ToList();
var book = new Book(bookInfoJson.title,
    bookInfoJson.description,
    DecodePubishDate(bookInfoJson.publishedDate),
    bookInfoJson.publisher,
    ((decimal?)bookInfoJson.saleInfoListPriceAmount) ?? DefaultBookPrice,
    bookInfoJson.imageLinksThumbnail,
```

```
2       string publisher, decimal price, string imageUrl,
3       ICollection<Author> authors)
4   {
5       if (string.IsNullOrWhiteSpace(title))
6           throw new ArgumentNullException(nameof(title));
7
8       Title = title;
9       Description = description;
10      PublishedOn = publishedOn;
11      Publisher = publisher;
12      ActualPrice = price;
13      OrgPrice = price;
14      ImageUrl = imageUrl;
15      _reviews = new HashSet<Review>();
16
17      if (authors == null || !authors.Any())
18          throw new ArgumentException(
19          "You must have at least one Author for a book", nameof(authors));
20
21      byte order = 0;
22      _authorsLink = new HashSet<BookAuthor>(
23          authors.Select(a => new BookAuthor(this, a, order++)));
24  }
```

Things to note from the DDD-styles constructor code are:

- Lines 1 to 2: The constructor requires you to provide all the data needed to create a property initialised instance of the Book class.
- Lines 5,6 and 17 to 19: The code has some business rule checks built in. In this case they are considered coding errors, so they throw an exception. If they were user-fixable errors I might use a static factory that returns Status<T>, where the Status type contains a list of errors.
- Lines 21 to 23: The creating of the BookAuthor linking table is done inside the constructor. The BookAuthor entity class can have a constructor with an internal access modifier, which stops the code outside the DataLayer from creating a BookAuthor instance.

- **The DDD-style controls access**. There is no possibility of changing a property by accident. Each change is done by a named method or constructor with defined parameters – it is very obvious what you are doing.
- **The DDD-style is DRY (don't repeat yourself)**. You might need to create a Book instance in a few places. By putting all the code in the Book's constructor then you don't have to repeat it in different places.
- **The DDD-style hides complex parts**. The Book has two properties, ActualPrice and OrgPrice, and both must be set to the same value when it is created. In the standard-style code it required the developer to know this fact, while the writer of DDD-style class knows how it should work and can put that code inside the constructor.
- **The DDD-style hides the setup of the aggregate, AuthorsLink**. In the stardard-style class the code had to create the BookAuthor entity class, including the ordering. With the DDD-style version that complication is hidden from the caller.
- **The DDD-style means the property setters can be private**. One reason for using a DDD-style is to "lock down" the entity, i.e. you cannot alter any of the properties or relationships directly.

This last point moves us onto the looking at the DDD-style for altering an entity's properties.

Compare altering the properties of a DDD-styled entity class

One of the advantages the Eric Evans says for a DDD-style entity is "They communicate design decisions about object access". I understand this to mean that the design of your entity class should a) make it obvious how to change data inside an entity, and b) make it obvious when you shouldn't change specific data in an entity. So let's compare two different updates that my business rules say are allowed – one is simple and the other is bit more complicated.

a. Standard entity class with public properties

The standard-style entity class woks by setting a property via is public setter, as shown in the following code.

```
var book = context.Find<Book>(dto.BookId);
book.PublishedOn = dto.PublishedOn;
context.SaveChanges();
```

b. The DDD-styled entity class with access methods

In the DDD-styled class the properties have private setters, so I have to set things via the access method.

```
var book = context.Find<Book>(dto.BookId);
book.UpdatePublishedOn( dto.PublishedOn);
context.SaveChanges();
```

There is nothing obviously different about these – in fact the DDD-style is a bit longer, as the UpdatePublishedOn method must be written (OK, it's only two lines of code, but its more work). But there is a difference – in the DDD-style entity class you know you can change the publication date because there is a method with an obvious name – UpdatePublishedOn . You also know you're not allowed to change the Publisher property, or any other property that doesn't have an access method. That is helpful to any developer that needs to interact with an entity class.

2. Adding/removing a promotion to a book

- The OrgPrice property holds its normal price – when there is no promotion the book should sell for the OrgPrice.
- The ActualPrice property holds the price that the book is currently selling for – if there is a promotion then the ActualPrice holds the new price. If there is not promotion it should be set to the OrgPrice.
- The PromotionText property must hold text to show they customer when there is a promotion, otherwise it should be null when there is no promotion.

The rules are fairly obvious to the person who implemented this business feature, but it may not be so obvious to another developer when he comes implement the front-end code to add a promotion. By creating the DDD-style AddPromotion and RemovePromotion methods in the Book entity class the implementer of the feature can hide the complexities of this implementation. The user of the Book entity has obviously names method to use.

Let's look at the AddPromotion and RemovePromotion access methods, as there is some further learning to take from these.

```
1  public IGenericErrorHandler AddPromotion(decimal newPrice, string promotionalText)
2  {
3      var status = new GenericErrorHandler();
4      if (string.IsNullOrWhiteSpace(promotionalText))
5      {
6          status.AddError(
7              "You must provide some text to go with the promotion.",
8               nameof(PromotionalText));
9          return status;
10     }
11
12     ActualPrice = newPrice;
13     PromotionalText = promotionalText;
14     return status;
15 }
```

Things to note from this code are:

To add a new promotion you would write:

```
var book = context.Find<Book>(dto.BookId);
var status = book.AddPromotion(newPrice, promotionText);
if (!status.HasErrors)
   context.SaveChanges();
return status;
```

The RemovePromotion is much simpler, and there is not possibility of a user-fixable error, so the method's return type is void

```
public void RemovePromotion()
{
    ActualPrice = OrgPrice;
    PromotionalText = null;
}
```

These two examples are quite different. The first example, setting the PublishOn property was so simple that the standard-styled class was fine. But the second example contained implementation details on a promotion is added/removed which wouldn't be obvious to anyone who hadn't worked on the Book entity. In that case the DDD-style access method hides that implementation detail, which makes it easier for the developer.

Also, the second version shows that there are some value updates can have a piece of business logic in them. For small pieces of business logic like this we can still use an access method by having the method a status result, which contains any errors.

3. Handling aggregates – the Reviews collection property

The backing field feature allows a developer to hide the actual collection in a private field and expose the collection as a IEnumerable<T>, which doesn't allow the adding, removing or clearing of the collection. The code below shows the part of the Book entity class that defines the Reviews collection as a backing field.

```
public class Book
{
    private HashSet<Review> _reviews;
    public IEnumerable<Review> Reviews => _reviews?.ToList();
     //… rest of code not shown
}
```

The other thing you need to do is to tell EF Core that on a read in of the Reviews relationship it should write to backing field, not the property. The configuration code to do that is shown below.

```
protected override void OnModelCreating
    (ModelBuilder modelBuilder)
{
     modelBuilder.Entity<Book>()
        .FindNavigation(nameof(Book.Reviews))
        .SetPropertyAccessMode(PropertyAccessMode.Field);
    //… other non-review configurations left out
}
```

To access the Reviews collection I added two methods, AddReview and RemoveReview, to the Book class. The AddReview methods is the most interesting one, and is shown below

```
public void AddReview(int numStars, string comment, string voterName,
    DbContext context = null)
{
    if (_reviews != null)
    {
        _reviews.Add(new Review(numStars, comment, voterName));
    }
```

```
14          {
15              context.Add(new Review(numStars, comment, voterName, BookId));
16          }
17          else
18          {
19              throw new InvalidOperationException("Could not add a new review.");
20          }
21      }
```

Things to note from this code are:

- Lines 4 to 7: I purposely don't initialise the _reviews field in the private parameterless constructor that EF Core uses when loading the entity. This allows my code to detect if the _reviews collection was loading, say via the .Include(p => p.Reviews) method. (For the create case I initialise the _reviews collection in the public constructor, so that Reviews can be added on creation).
- Lines 8 to 12: If the Reviews collection has not been loaded then the code needs to use the DbContext, so this just checks its set.
- Lines 13 to 16: If the Book has already been written to the database, then it will have a valid key. In that case I use a quicker technique to add a Review, by setting the foreign key in the new Review instance and write it out to the database (see section 3.4.5 in my book for this technique).
- Line 19: If none of these work then there is a code problem, so I throw an exception.

> *Note: I have designed all of my access methods to handle the case where only the root entity is loaded. It is up to the access method on how it achieves the update of the aggregates – this could include loading additional relationships.*

Summarising the DDD-style I have developed in this article

which will return Status<T>, where T is the class it is creating.
2. All properties should have a private setter, i.e. they are read-only from outside the class
3. For navigational properties that are collections you need to use backing fields and a IEnumerable<T> for the public property. That stops developers from changing the collection.
4. You need to provide methods for all the different ways you want to change the data in the entity class, both property updates and aggregate updates. These access methods either return void if the process has no user-fixable errors in it, or a status value containing success or a list of errors if the code can detect user-fixable errors.
5. The 'scope' of what the code in the entity class should take on is important. I have found limiting the methods to only changing the root entity and its aggregates is a good idea. I also limit my validation checks to making sure the entities I am updating are correctly formed, i.e. I don't check outside business rules like stock availability etc. – that is the role of proper business logic code.
6. The access methods must assume only the root entity has been loaded. If it needs relationship that isn't loaded, then it must load the extra relational data itself. This simplifies the calling pattern to all access methods.

This last item means there is a standard way of calling access methods – here are the two forms, starting with the access method in which no user-fixable errors could occur:

```
var book = context.Find<Book>(bookId);
book.UpdateSomething( someData);
context.SaveChanges();
```

If there could be an error then the code would be

```
var book = context.Find<Book>(bookId);
var status = book.UpdateThatCouldHaveErrors( someData);
if (!status.HasErrors)
      context.SaveChanges();
return status;
```

The good parts of using a DDD approach with EF Core

- **Using access methods to change value property is clearer.** The act of making all the properties have a private setter and then having to use meaningfully-named methods to change the properties is positive. The named access methods make it obvious what you can change, and going via a method means it can return an error status if it detects a problem.
- **Altering the aggregates via business-oriented methods works well too.** Hiding the relationships, such as the Book's one-to-many relationship to the Review class means that the details of how that update is done is kept inside the root entity class.
- **Using specific constructors to create entity instances ensures an entity is created properly**. Moving the code for building a valid entity class into the class's constructor reduces the likelihood of a developer incorrectly interpreting how a class should be initialized.

The bad parts of using a DDD approach with EF Core

- **My design includes some EF Core code inside the access methods,** and this is considered an anti-pattern by some people. The problem is that your domain entities are now linked to the database access code, which in DDD terms isn't a good thing. I found if I didn't do this then I was relying on the caller to know what needed to be loaded, which breaks the separation of concerns rule.
- **The DDD CRUD code requires more code to be written.** For simple updates, like the change to the publication date of a book, the DDD-styled access methods seem a little bit of an overkill. Is it worth it?

My decision to allow EF Core code inside the access methods was a tough call. I tried to keep EF Core commands out of the access methods, but this led to the caller of the method to have to load the various navigational properties need by the method, and if the caller didn't do that correctly the update could fail silently (especially on one-to-one relationships). That to me was a not acceptable, so I allowed EF Core code inside some methods (by not constructors).

The other bad point was that DDD CRUD access method requires more code, and I'm still thinking about that one. Should I bite the bullet and make all updates go through a DDD access method, or is there a case for allowing some properties to have a public setter and allow direct change of the value? The purest in me would say always use access methods, but I know that there is a ton of boring updates to be done in a typical application, and it's easier to do directly. Only a real project will tell me what works best.

Other aspects of DDD not covered in this article

This article is already very long so I'm going to finish here, but that means I will leave some big areas out. Some I have already written about and some I will be writing about in the future. Here are the things I have left out.

- **Business logic and DDD**. I have used DDD concepts with business logic for some years, and with EF Core's new features I expect to move some of the business logic into the DDD-styles entity classes. Read this article "Architecture of Business Layer working with Entity Framework (Core and v6) – revisited".
- **A DDD repository pattern**. Eric Evans talks about building a repository pattern to hide your database access library. I have concluded that using a repository pattern with EF Core is a bad idea – you can read why in this article "Architecture of Business Layer working with Entity Framework (Core and v6) – revisited".
- **Multiple DBContexts/Bounded Contexts**: I have been thinking a lot about spitting a database across multiple DbContext's, for instance having a BookContext that only knows about the Book entity and its aggregates, and a

Conclusion

EF Core, with its backing field feature, allows the building a DDD-styled entity classes, which is "locked down", i.e. the public constructor(s) define how to create an instance and properties can only be updated via access methods. The benefits are that makes it more obvious how the entity class should be used, and knowledge about the inner workings of the entity class are held inside the entity class itself.

It has taken me some time to get a DDD style that provided a consistent interface to the entity class access methods. Getting the access method pattern right is important because it makes the calls quicker and simpler to code. In any application that uses a database you will get a lot of CRUD (Create, Read, Update and Delete) database accesses and you need produce these quickly and easily – so a simple and consistent pattern will make you quicker at writing CRUD code.

> *NOTE: You can find the code in this article in GenericBizRunner GitHub repo. That repo also contains an example ASP.NET Core application where you can try changing a book using these access methods. Please do clone it and run it locally – it uses an in-memory Sqlite database so it should run anywhere.*

Happy coding.

Jon P Smith .NET Core, Domain-Driven Design, Entity Framework

Montgomery Beltz • 2 months ago

Hey **@Jon P Smith**
I'm starting a project with high hopes of utilizing the DDD structure you show here but I'm having trouble getting the entity configured correctly in the OnModelCreating(). We have created the DB in sql first and then generated and EF Core model from the database. Is it still possible to go through each entity and modify them to support DDD pattern? Thanks for any input!

Jon P Smith Mod → Montgomery Beltz • 2 months ago

Hi,

If you create entity classes by reverse engineering an existing database they are 'normal' classes, e.g. the properties have public setters. You could hand-end these entity classes to turn then into DDD-styled classes. But if you change your database and need to reverse engineer your classes again you will lose all your edits to make the classes DDD, which isn't what you want.

EfSchemaCompare feature in my EfCore.TestSupport library - see https://github.com/JonPSmit...

Reply • Share ›

Montgomery Beltz ➚ Jon P Smith • 2 months ago

Thanks so much for the quick reply! I will certainly check out your EfSchemaCompare, I made a goofy mistake that was causing the problems I was having.

At this point, we've generated a fairly large db context from an existing db, and then I chose one entity to start applying DDD changes to and then created migrations, so far so good but I haven't gotten that far yet.

My hopes were to not have as much to hand code starting out, modify the entities to support DDD and then generate migrations from here on.

Reply • Share ›

Jon P Smith Mod ➚ Montgomery Beltz • 2 months ago

That sounds much more doable - create your initial entity classes from a DB and then swap over to using migrations to handle the update of the database as your entity classes change.

All the best with your project!

Reply • Share ›

Kimberly Abacco • 5 months ago

Awesome explanation on aggregates!

Reply • Share ›

and would like to check with you on the Boundary context of DDD. If I would like to use DbContext as a boundary for each domain. How could we share an entity between the boundary and how to manage the migration properly?

Thanks
Duy

• Reply • Share ›

Jon P Smith Mod ➚ baoduy2412 • 6 months ago • edited

Hi Duy,

Firstly, having multiple DbContexts without any shared entities will cause a problem if you try to add migrations to each DbContext. See this StackOverFlow question on what the problem is, and how to get around it.

But, you are right that sharing entity classes across multiple DbContexts can cause extra problems for EF Core's migrations because both migrations will try to create/update the table(s) that are shared between DbContexts.

I can give you three possible approaches:

1. You apply a migration for each DbContext, but you need to hand-edit a migration(s) so that there is only one create/update to the shared class/table.
2. You create a combined DbContext that covers all the table and is only used for migrating the database. This means you need to keep all the DbContexts in step (which can be hard), but it allows you to have one migration (I did this in my PermissionAccessControl2 - see https://github.com/JonPSmit... , with unit tests to check they don't get out of step.

 • Reply • Share ›

baoduy2412 ➚ Jon P Smith • 6 months ago

Hi @**@Jon P Smith**

Thank you so much for your advices. Seems the second approach is suitable to my application. Let me try out your suggestions.

 • Reply • Share ›

Jon P Smith Mod ➚ baoduy2412 • 6 months ago

You might like to look at a simple test that I added that checks that the expected tables are created by my combinedDb - see https://github.com/JonPSmit... It's pretty simple, but it at least checks if you left out any tables.

You can do more comprehensive tests with my EfSchemaCompare if you are building a real system - see https://github.com/JonPSmit...

 • Reply • Share ›

Karim Behboodi • 6 months ago

Don't want to be rude, but your design is a disaster and one of the most misleading tutorials about DDD that I've seen.
First of all, you have misunderstood the DDD itself, which is NOT about technical aspect of the design.
Second, You failed to design a Domain Model -which is about technical aspect of the design-, because one of its basic rules, Persistence Ignorance, has been broken, with present of DbContext.
Finally, the nearest thing to what you've proposed is Active Record, By definition, unlike the Domain Model approach, Active Record is aware of its

implement DDD and this article is about DDD with EF Core, so if course its going to focus on persistence. So it shows how to implement CRUD (Create, Read, Update Delete) features rather than business logic.

In fact my DDD business logic approach very much persistent ignorant - see this section in my DDD Biz Logic article https://www.thereformedprog...

Other people have told me I have gone too far with DDD and also that I haven't gone far enough, to the point that I wrote an article comparing different approaches - see https://www.thereformedprog...

If you have a way to implement DDD that works for you then that's great. You might like to write your own article about it to help others.

Gapwe • a year ago

Hi ! First of all, thanks for those articles, I'm trying to apply those patterns to a personnal project.
One thing remains unclear for now and I can't find the answer in any of the articles serie nor in the repo.
In the example here with Book, Author and Review :
I get that Book is the "Root" entity of the domain and sorry if my question is dumb but who should be in charge of adding a new Book?

From what I understand here are the steps I follow (please correct me if I'm wrong) :
Book entity has a method "AddReview(..,...,..,_context)".
The service layer has a AddReviewService with a AddReviewToBook method that uses the AddReview method and calls the SaveChanges();
A Controller injects the IAddReviewService and calls the AddReviewToBook

ServiceLayer so that I can call it from a Controller.
That would mean having a AddBook() method (in the book entity? sounds weird no?),
an AddBookService in the service layer to call the AddBook() method and call SaveChanges();

I am definetly missing something here and i would appreciate some help.
I'll keep looking :p
Thanks for reading !

^ | v • Reply • Share ›

Jon P Smith Mod ➚ Gapwe • a year ago

Hi Gapwe,

Your question is good - DDD is really quite complex and it took me a lot of time to get a pattern that worked for me. Let me tell you what I do.

When it comes to a root entity, like the Book, I either create it via a constructor or if it could return errors then I use a static factory that returns a status. You can see both options in the Book class in the example code in my EfCore.GenericServices. You might like to clone that repo - you can run the RazorPageApp and it creates and seeds an in-memory database so you can play with it.

Now, I have explained my apporach but there are people who do this differently - there's no right or wrong, just what works for you and the business needs. You might like to look at my article Three approaches to Domain-Driven Design with Entity Framework Core which looks at other approaches. That might help you see the overall approach.

All the best on your journey of DDD!

^ | v • Reply • Share ›

didn't think about the factory. I'll look into that too.

Another question already ! I'm thinking about the pros and cons of different options when it comes to multiple levels of aggregates.
In the example you use with Book and Reviews, if each Review contained an aggregate of, let's say, CustomerComment, I see two options here :
1) The Book entity contains the methods to add a CustomerComment with the correct ReviewId
2) It is the Review entity that contains the method to add a CustomerComment

I'm currently working on building the two options to discover the pros/cons but I would appreciate some returns from your experience dealing with this on a bigger scale.

Also, just a matter of style (I guess?), but I was thinking about grouping the CUD services in one Service (ex: "IReviewService" instead of "IAddReviewService", "IUpdateReviewService" and "IDeleteReviewService").
Kind of what you did with the "IAddRemovePromotionService".
Do you think of any bad implication of doing so?
In a big scale project, having three times more Interfaces and Concrete class can become confusing when looking for a specific service in the solution.

Again, thanks for those articles, I've never heard of DDD before last week and as for now, I really enjoy the cleanness of the code it provides.

On handling an "aggregate of an aggregate" my answer is it depends. I would tend to keep going through the root, but it could get very heavy. I had one hierarchical set of aggregates and that didn't work well so I added a method to a higher aggregate. Not sure whether that was best though. "aggregate of an aggregate" isn't something I have used enough to come to a definitive pattern yet. If you get something useful then you should add it your own blog.

Grouping services can work, but they have to be very linked. I an a fan of the single responsibility principle, which gives me a set of classes/code that are closely related (see the diagram at the top of my article about the repository pattern). I believe this approach is know as the "vertical architecture" (I think).

Have fun.

 • Reply • Share ›

Piotr Kołodziej • a year ago

Let's assume a set of entities (Registration, Room, Invoice, Order etc..) all have a Status column. Where would you handle updating statuses? I am asking because it seems very natural to me to change status to Registration.Paid when i call registration.Pay but I struggle with registration.AddInvoice which in my current project does this:

```
public void AddInvoice(Invoice invoice)
{
_invoices.Add(invoice);

if(Status == RegistrationStatus.Open && invoice.GrandTotal > 0)
Status = RegistrationStatus.Overdue;
```

Jon P Smith Mod ➚ Piotr Kołodziej • a year ago • edited

Hi Piotr,

I'm a great fan of the "Single Responsibility Pattern", and that would say that your approach to AddInvoice is correct. By keeping everything about creating and changing an entity inside that entity then if you either want to refactor or move it, then its all in one place.

The dependency on RegistrationStatus.Open looks fine to me (I do assume the RegistrationStatus is an Enum in the entity your AddInvoice is in). This is a small bit of business logic and think having that in the method is good. If the business logic got more complex, say with the status relying on other root entities, then I might elevate the AddInvoice to a full-blown piece of business logic - see my article Architecture of Business Layer working with Entity Framework (Core and v6) – revisited, but that looks like overkill for what you are doing.

PS. One thing that does worry me is your `_invoices.Add(invoice);`. It looks like your using a backing field for the invoices, and the way you have written it relies on you having loaded the Invoices relationship before you call AddInvoice. If you didn't reload that relationship it would either a) fail because the collection was null or worse b) if you gave it an empty collection it would delete all the existing invoices and add just the latest one. See how I handle this problem in the section on handling the reviews collection in this article.

Piotr Kołodziej ➚ Jon P Smith • a year ago

Thanks for the reply. I did remove null checking on _invoices to

your article:

"The access methods must assume only the root entity has been loaded. If it needs relationship that isn't loaded, then it must load the extra relational data itself. This simplifies the calling pattern to all access methods."

To be honest I still prefer the solution that you took here:

https://github.com/JonPSmit...

if (_lineItems == null)
throw new NullReferenceException("You must use .Include(p => p.LineItems) before calling this method.");

Reply • Share ›

Jon P Smith Mod ➚ Piotr Kołodziej • a year ago

That's fine. I'm really glad you are aware of it as its a potential "fail silently" error and I hate those. Just wanted to check.

PS. Thanks for buying my book :)

Reply • Share ›

ardalis • 2 years ago

The result of this approach is that your entities all are tightly coupled to EF Core. That's not good, since your domain model should be made up of persistence-ignorant POCOs. You fail to note this when you write "The only bad point was that DDD CRUD access method requires more code, and I'm still thinking about that one."

I saw references to hexagonal/onion architecture in another article of yours. If

this architecture, it's basically impossible to have references from your entities to your DbContext types. This helps avoid the tight coupling that you're setting up with your approach.

 • Reply • Share ›

Jon P Smith Mod ➚ ardalis • 2 years ago

Hi ardalis,

Yes, thanks for pointing that out. I have added a section on the "bad parts" of my design to bring that out in a better way. I used to have the entity classes in a separate assembly so that they were definitely not linked to EF Core. The problem I had with going DDD for entity classes is if I don't put EF Core code inside some of the access methods, then I am relying on the caller to load certain navigational properties for the access method to work correctly. That causes two problems:
1. The called needs to know what navigational properties to load to allow the access method to work properly. In that can I might as well not use access methods, as I now have spread the database code across two parts of the code, which isn't a good idea.
2. If I did split the code into two parts then there are cases, mainly around one-to-one relationships, where if the caller didn't load the correct relationships then the update would fail silently, i.e. it wouldn't correctly do the update.

So, you are right that I have mixed database access into my domain entities, but the alternative is not to go to a DDD-styled entity class. However the important point is that the business logic, which I consider to be outside the entity, (see my reply to ES later in these comments) doesn't need to knew about the database. I consider the access methods a kind of Create/Update repository pattern.

I hope this comment, plus the updated article, helps. I understand your

ardalis ➔ Jon P Smith • 2 years ago

Also two reference applications you or your readers may find useful:

https://github.com/dotnet-a... which includes 120+ page eBook from Microsoft (free)

https://github.com/ardalis/... (solution starter for Clean Architecture implementation with DDD concepts and ASP.NET Core 2.1 / EF Core 2.1)

1 ▲ | ▼ • Reply • Share ›

ardalis ➔ Jon P Smith • 2 years ago

When you need an entity with its relationships (probably an Aggregate), you should fetch it from the repository and it should have all of its state loaded by EF (or at least sufficient state to be able to perform its logic correctly). If, probably for performance reasons, you don't always want to load all of the relationships, you can use a separate read model for that purpose. This addresses both of your two problems above. The caller doesn't need to know anything - it just gets the aggregate from the repository. And if the Aggregate is always fully loaded, you avoid the problems you describe in your #2 problem. Probably the reason you don't always eager load navigation properties is performance. In that case, determine where and if you have actual performance problems and address them with read models and/or caching. HTH.

▲ | ▼ • Reply • Share ›

Niklas Wulff ➔ ardalis • 2 years ago

I am very curious about this suggestion, do you have an example on how this could be implemented?

involving multiple bounded contexts and microservices here: https://github.com/dotnet-a...

From the former sample, you can see how a Specification can be used to determine how much state should be included when fetching an entity (e.g. eager loading): https://github.com/dotnet-a...

You should of course avoid lazy loading in web applications for performance reasons: https://ardalis.com/avoid-l...

Hope this helps.

^ | v • Reply • Share ›

Niklas Wulff ➚ ardalis • a year ago

Thanks!

^ | v • Reply • Share ›

Jon P Smith Mod ➚ ardalis • 2 years ago • edited

HI Ardalis,

Sorry for the delay in replying - I was working on a client project.

You may have seen I'm not a great fan of some repository patterns (see https://www.thereformedprog...). A repository uses a Facade design pattern, which if build in the database layer tends to think in database terms, while many CRUD accesses need to think in UI/API terms. I have seen many repositories that bring performance issues,

Neil Ford's comment “The more reusable the code is, the less usable it is.” - I tend to find repositories stop me from tweaking things, as I'm not sure what else it will effect, while Query Objects and DDD-styled access methods, which are specific solutions to a problem, work better for me.

Reply • Share ›

M F • 2 years ago

How would you handle 0..1 to many self referencing relationships in the constructor? For instance, I have ParentCategory and ChildCategories below

```
public class Category
{
public int Id { get; private set; }

public int? ParentCategoryId { get; private set; }
public Category ParentCategory { get; private set; }

private HashSet<category> _childCategories;
public IEnumerable<category> ChildCategories => _childCategories?.ToList();

private Category() { }

public Category(
int id,
int? parentCategoryId = null
)
```

see more

Reply • Share ›

On your JsonConvert.DeserializeObject problem let me explain what was happening in your non-DDD version.
- For each Category the Deserialize would set the ChildCategories list
- The AddRange will treat all the deserialized Categories as in the "Added" state, i.e. it will create a new row. EF will use the navigational properties list to define the relationships (EF Core will ignore primary/foreign keys in instances in the "Added" state).

So, when you changed over to DDD-styled class the Deserialize can't set the _childCategories field, so EF can't find the relationship. Therefore you end up with separate Categories with no links between them. One way around this is to add the setting of the ChildCategories via a constructor and use the https://www.newtonsoft.com/... feature. (but see my comments about ChildCategories below).

On your ParentCategoryId problem I'm not so sure what is going on. I do notice you have defined a one-to-one relationship (ParentCategory) and an one-to-many relationship (ChildCategories) to the same class, but only one foreign key of ParentCategoryId - I don't know what EF Core does with that (maybe defines a shadow foreign key??). I think having a one-to-many relationship in a hierarchical class is a bit problematic, because what group are you pointing to? - maybe a one-to-one ChildCategory, with a ChildCategoryId foreign key would work, but I don't know your application enough to say that for sure.

I hope that helps.

 • Reply • Share ›

Robert McRackan ➚ Jon P Smith • a year ago

Maybe I can shed some light on the 1..0 question because I'm also

```
public int BookId { get; set; }
public virtual AmazonPropertySet AznProps { get; private set; }
}
public class AmazonPropertySet
{
public int BookId { get; set; }
public Book Book { get; set; }
public string Link { get; private set; }
public decimal Price { get; private set; }
}
modelBuilder.Entity<amazonpropertyset>()
.HasKey(t => t.BookId);
modelBuilder.Entity<amazonpropertyset>()
.HasRequired(t => t.Book)
.WithOptional(x => x.AznProps);
```

EF Gives us a table for AmazonPropertySet with fields for BookId, Link, and Price. In trying to follow your pattern, there's no difference between AznProps is null because "it wasn't loaded from the db" vs it's null because "it's supposed to be null."

▲ | ▼ • Reply • Share ›

Jon P Smith Mod ➚ Robert McRackan • a year ago

Hi Robert,

I'm not sure what the problem that Makalii was having, but I can confirm that with an optional one-to-one the the relationship can be null a) because there isn't anything there, or b) you didn't load the relationship. That's why not need to be so careful with the "one" side of an optional relationships.

it will remain as its default value. That's why, by default, I leave "Many" relationships at null, as it helps me catch if I forget to load it when I needed it.

 • Reply • Share ›

Robert McRackan ➚ Jon P Smith • a year ago

Thank you for you quick reply!

Agreed. Sorry for being unclear. My question is: given the ambiguity of null-ness, how can your pattern (ie: the one you used in your AddReview example) be adapted to the case of optional one-to-one? You use the collection as a shorthand to determine new-ed up by code (empty collection) vs loaded from context (is null). Without that shorthand, I'm left with this mess. Do you have a better way?

```
private bool isFromCode { get; }
private Book() { }
public Book(my book params) { isFromCode = true; ... }
public void AddAznProps(string link, decimal price,
DbContext context = null)
{
if (AznProps != null) populateExisting(link, price);
else if (context == null) throw new Exception("need
context");
else if (!context.ContainsAznProps(this.BookId))
this.AznProps = new AmazonPropertySet(link, price);
// we now know the AznProps is in the db
else if (isFromCode) throw new Exception("shouldn't be
possible");
```
