# Entity Framework 6 Database-First and Onion Architecture

Asked 5 years, 8 months ago Active 5 years, 6 months ago Viewed 5k times

12

6

I am using Entity Framework 6 database-first. I am converting the project to implement the onion architecture to move towards better separation of concerns. I have read many articles and watched many videos but having some issues deciding on my solution structure.

I have 4 projects: Core, Infrastructure, Web & Tests.

From what I've learned, the .edmx file should be placed under my "Infrastructure" folder. However, I have also read about using the Repository and Unit of Work patterns to assist with EF decoupling and using Dependency Injection.

With this being said:

- Will I have to create Repository Interfaces under CORE for ALL entities in my model? If so, how would one maintain this on a huge database? I have looked into automapper but found issues with it presenting IEnumererables vs. IQueryables but there is an extension available it has to hlep with this. I can try this route deeper but want to hear back first.
- As an alternative, should I leave my edmx in Infrastructure and move the .tt T4 files for my entities to CORE? Does this present any tight coupling or a good solution?
- Would a generic Repository interface work well with the suggestion you provide? Or maybe EF6 already resolves the Repository and UoW patterns issue?

Thank you for looking at my question and please present any alternative responses as well.

I found a similar post here that was not answered: EF6 and Onion architecture - database first and without Repository pattern

entity-framework automapper n-tier-architecture separation-of-concerns onion-architecture

edited May 23 '17 at 12:17
Community ♦
1 1

asked Jun 24 '14 at 16:50
Sean Merron
342 3 14

2 Interfaces and entities should be in Core. For a large database, look into bounded contexts and domain driven design. The goal of Onion Architecture is for your Core project(s) to have no references to external frameworks such as EF, AutoMapper, ASP.NET, WCF, etc. For EF specifically, it's a bit harder to separate your entities and EF itself if you're using EDMX. – Anthony Chu Jun 24 '14 at 17:02

Agree with @AnthonyChu on EDMX. You should look into Reverse Engineered Code First with EF Power Tools. – EfrainReyes Jun 24 '14 at 17:25

Thanks guys. @AnthonyChu will bounded contexts and DDD provide a solution that works with database-first or an alternative? – Sean Merron Jun 24 '14 at 17:58

@EfrainReyes This appears to be an alternative ruling out database-first in Onion. I want to verify that it is NOT possible to use database-first while still maintaining the principles of Onion. – Sean Merron Jun 24 '14 at 17:59

1 Why are you even considering onion architecture? What problems is it solving for you? It sounds like you are creating a lot of layers and abstractions under the guise of encapsulation. – Jimmy Bogard Jun 25 '14 at 13:07

## 2 Answers

9

Database first doesn't completely rule out Onion architecture (aka Ports and Adapters or Hexagonal Architecture, so you if you see references to those they're the same thing), but it's certainly more difficult. Onion Architecture and the separation of concerns it allows fit very nicely with a domain-driven design (I think you mentioned on twitter you'd already seen some of my videos on this subject on Pluralsight).

You should definitely avoid putting the EDMX in the Core or Web projects - Infrastructure is the right location for that. At that point, with database-first, you're going to have EF entities in Infrastructure. You want your business objects/domain entities to live in Core, though. At that point you basically have two options if you want to continue down this path:

1) Switch from database first to code first (perhaps using a tool) so that you can have POCO entities in Core.

2) Map back and forth between your Infrastructure entities and your Core objects, perhaps using something like AutoMapper. Before EF supported POCO entities this was the approach I followed when using it, and I would write repositories that only dealt with Core objects but internally would map to EF-specific entities.

As to your questions about Repositories and Units of Work, there's been a lot written about this already, on SO and elsewhere. You can certainly use a generic repository implementation to allow for easy CRUD access to a large set of entities, and it sounds like that may be a quick way for you to move forward in your scenario. However, my general recommendation is to avoid generic repositories as your go-to means of accessing your business objects, and instead use Aggregates (see DDD or my DDD course w/Julie Lerman on Pluralsight) with one concrete repository per Aggregate Root. You can separate out complex business entities from CRUD operations, too, and only follow the Aggregate approach where it is warranted. The benefit you get from this approach is that you're constraining how the objects are accessed, and getting similar benefits to a Facade over your (large) set of database entities.

Don't feel like you can only have one dbcontext per application. It sounds like you are evolving this design over time, not starting with a green field application. To that end, you could keep your .edmx file and perhaps a generic repository for CRUD purposes, but then create a new code first dbcontext for a specific set of operations that warrant POCO entities, separation of concerns, increased testability, etc. Over time, you can shift the bulk of the essential code to use this, while still keeping the existing dbcontext so you don't lose and current functionality.

answered Jun 24 '14 at 20:39

ssmith
**5,670** 5 35 62

Thanks @ssmith it looks like I may need to dive into DDD and CF further vs. continuing down the db first path. I'm green to it and sure I'll have a ton of questions I can find answers to online but if I move this direction how do you personally deal with integrating db changes into your POCO classes? Do you strictly force migrations or still run into scenarios where you have to propagate these db changes into your code like a db first model update would do? – Sean Merron Jun 24 '14 at 21:51

If you are following a DDD approach, the database responds to support changes to your Model, not the other way around. There shouldn't be any database changes that require you to change your model. Ideally you have a database that you have complete control over just for your application, and it integrates with other systems or databases in your organization via services or some form of anti-corruption layer, not direct database interaction. Hopefully that helps. – ssmith Jun 25 '14 at 15:57

1 Exactly what I was looking for @ssmith thanks again! I just opened up your DDD course with julie to start learning more about this concept. – Sean Merron Jun 26 '14 at 1:02

the EDMX file is a resource; However, if you build it with entity deploy, you're embedding metadata into the assembly it resides in, so it has a function a runtime. You can migrate the T4 TextTemplate files that generate the entities and dbcontext into other assemblies/projects. You do this by modifying the TextTemplate, updating the edmx filepath in the text. Note: The relative pathing between your projects now becomes a factor. The EDMX file can also be used by WebAPI 2 OData DataServiceContext for LINQ to Service operations. so it's really more of a shared resource in this sense.. – Brett Caswell Sep 26 '14 at 15:20

whatever assembly/project the edmx ends up in, it has to be added as a dependency to whatever project initializes the dbcontext instance.. the connectionstring meta res declarations will attempt to resolve/use the embedded entity meta data at the point the dbcontext is used, I think..which I suspect will be initialized in your infrastructure service layer with the generic repository... – Brett Caswell Sep 26 '14 at 15:27

1

I am using entity framework 6.1 in my DDD project. Code first works out very well if you want to do Onion Architecture.

In my project we have completely isolated Repository from the Domain Model. Application Service is what uses repository to load aggregates from and persist aggregates to the database. Hence, there is no repository interfaces in the domain (core).

Second option of using T4 to generate POCO in a separate assembly is a good idea. Please remember that your domain model (core) should be persistence-ignorant.

While generic repository are good for enforcing aggregate-level operations, I prefer using specific repository more, simply because not every Aggregate is going to need all of those generic repository operations.

http://codingcraft.wordpress.com/

answered Aug 24 '14 at 4:38

Anshuman

**31** 2

Thank you, I have done a lot of research on DDD since this post and will be following a similar approach. It's great to hear about other's implementation of DDD. – Sean Merron Aug 25 '14 at 7:44