Advice on mapping of entities to domain objects [closed]

Asked 11 years, 1 month ago Modified 3 years, 7 months ago

Viewed 32k times

40

Closed. This question is opinion-based. It is not currently accepting answers.

Want to improve this question? Update the question so it can be answered with facts and citations by editing this post.

Closed 3 years ago.

Improve this question

I'm currently working in a project where we are starting to build an application using a DDD approach. We are now looking into using Entity Framework 6 code first to help us with data persistence. My question is how to best handle data mapping between our domain objects and EF entities?

c# entity-framework domain-driven-design

Share

edited Jan 10, 2014 at 10:11

Improve this question

Follow

Ilya Palkin

15.7k ● 2 ● 24 ● 38

asked Nov 15, 2013 at 21:37

mstrand

3,083 ● 5 ● 25 ● 27

4 Answers

Sorted by: Highest score (default)

56

To keep your app and yourself sane on the long term, NEVER EVER start your DDD app with persistence related issues (what db, what orm etc) and ALWAYS (yes, always) touch the db as the last stage of the development.

Model your Domain and in fact any other model except persistence. Use the Repository pattern to keep the app decoupled from the Persistence. Define the repo interface as needed by the app and not tied to the db access method (that's why you're implementing the persistence later so you won't get tempted to couple your app to persistence details).

Write in-memory implementations for the repo interfaces, this usually means a simple wrapper over a list or dictionary so it's VERY fast to write and more importantly trivial to change. Use those to actually test and develop the app.

After the interfaces are stable and the app works then it's time to write the persistence implementation where you

can use whatever you wish. In your case EF and there it comes the mapping.

Now, this is highly subjective there isn't a right or wrong way, there's the way YOU prefer doing things.

Personally, I have the habit of using mementos so I get the memento from the domain object and then manually mapping it to the (micro)ORM entities. The reason I'm doing it manually is because my mementos contain value objects. If I would be using AutoMapper I'd be needing to confingure it and in essence I'd be writing more code than doing it manually

Update (2015)

These days I just Json the object and either use a specific read model or store it directly in a read model with a `Data` column that contains the serialized object. I use Mementos only for very specific cases. < /update>

Depending on how your domain objects look and how the EF entities look you might get away with using automapper for the majority of the mapping. You will have a harder time testing your repositories though.

It's up to you **how** you do it, find the way it suits your style and it's easily maintainable but NEVER EVER design or modify your domain objects to be more compatible or to match the ORM entities. It's not about changing databases or ORMs, it's about having the Domain (and

the rest of the app) properly decoupled from the Persistence details (which the ORM is).

So resist the temptation to reuse things which are other layers' implementation details. The reason the application is structured in layers is because you want decoupling. Keep it that way.

Share Improve this answer Follow

edited Jul 16, 2015 at 17:34

answered Nov 18, 2013 at 9:11

MikeSW

16.3k ● 3 ● 41 ● 53

Thanks for the answer. What do you mean by 'Json the object' ? – Artyom Mar 14, 2017 at 14:15

@Artyom Serialize the object as json – MikeSW Mar 14, 2017 at 18:40

21

Why don't you simply use EF entities as Domain objects since you are looking into using **Entity Framework 6 code first**? So first you design Domain Model then Data Base structure.

I've been using NHibernate and believe that in EF you can also specify mapping rules from DB tables to your POCO objects, especially with EF6. It is an extra effort to develop another abstraction layer over EF entities. Let ORM be responsible for it.

I do not agree with this article that you might read "Entity Framework 5 with AutoMapper and Repository Pattern" and **there are many other articles where people simply use EF entities as Domain objects**:

- Creating an Entity Framework Data Model for an ASP.NET MVC Application (1 of 10)
- SO question "Domain Driven Design and Entity Framework 4.1 (code-first)"
- Generic information about DDD "Domain Driven Design and Development In Practice"
- Entity Framework 'Code First' Approach and Domain-Driven Design

AutoMapper will definitely help you when you start building presentation layer and face lots of UI specific view models. It is useful in building anemic models and a bit useless with real Domain objects when there are no public setters.

There is an old post by Jimmy Bogard "The case for two-way mapping in AutoMapper" where he tells that `"There is no two-way mapping because we never need two-way mapping."`

> So what are we using AutoMapper for? Our five profiles include:
>
> - From Domain to ViewModel (strongly-typed view models for MVC)

- From Domain to EditModel (strongly-typed view models for forms in MVC)
- From EditModel to CommandMessages – going from the loosely-typed EditModel to strongly-typed, broken out messages. A single EditModel might generate a half-dozen messages.
- From Domain to ReportModel – strongly-typed Telerik reports
- From Domain to EDI Model – flattened models used to generate EDI reports

Share Improve this answer

Follow

edited May 12, 2021 at 12:41

SH_SWAT

269 ● 4 ● 12

answered Nov 15, 2013 at 23:56

Ilya Palkin

15.7k ● 2 ● 24 ● 38

2 What about the situation where your database already exists, and the senior engineers want to keep tons of business logic locked in stored procedures for example. Any advice on how one might approach such a situation? (find a new job?)
– crush Nov 20, 2017 at 22:06

@crush If "your database already exists" so you cannot evolve database structure based on your model in code then you cannot use Code First approach and should use DB First. Although, I'd even challenge usage of Entity Framework (EF) or NHibernate. I don't need any complex ORM and can simply hide any data access behind your repositories. You

repositories internally can even call stored procedures or use lightweight ORM (e.g. Dapper) – Ilya Palkin Nov 21, 2017 at 0:09

@crush If "senior engineers want to keep tons of business logic locked in stored procedures" then I assume this business logic is covered by (integration) tests. There probably a reason for it. Usually it is performance. – Ilya Palkin Nov 21, 2017 at 0:14

I would ***only use EF or NH for write model***. In addition I'd ***keep aggregate boundaries relatively small*** and ***reference other aggregate roots by `Id` only***. There is a common pitfall in ORMs - they make referencing aggregate roots by reference very easy. – Ilya Palkin Nov 21, 2017 at 0:24

1 I've implemented all of repositories using Dapper and AutoMapper, but I was hoping to get my team used to EF, and start developing new features with Code First so they can see the advantages and hopefully move away from stored procedures (except where they are truly needed). The business logic has been locked away in stored procs traditionally because the senior developer seems like wasn't comfortable with languages outside of SQL to be quite honest, and didn't understand how to write reusable code components. I've also been enforcing that we use Id's instead of references for aggregate roots. – crush Nov 21, 2017 at 1:33

6

As I am researching this topic myself, I find that I am of the opinion that most developers using or recommending to use the EF POCO's as your domain objects simply don't have the necessity for DDD. DDD is a recommended approach for complex domain logic, and when you have complex domain logic, you will very

unlikely have a 1:1 mapping of your domain logic and data storage structures. It is true that in a lot of cases you will have domain objects with a similar structure to the database design, but as your domain becomes more complex you will find that a lot of your domain entities have a requirement to deviate from how the data is physically stored. I share the same opinion if you find yourself wanting to use an auto mapper - that you probably don't have the need for DDD in that case either. Your application is likely simple and would not benefit from the complexity of DDD. And mind you, DDD is not an architecture or design pattern, it is a method of building software to develop a rich domain model using ubiquitous language.

I work on an enterprise scale system and one example is our business logic for documents. For a document to exist, it must satisfy the requirements:

- Be assigned to a tab
- Be assigned default properties based on tab defaults
- The tab may or may not exist yet, and tabs also have default properties from a global definition
- Document might have an image (file), or it might not
- Our system has modules (tracking vs. imaging), so the customer might not even have access to store files
- Audit logging must occur
- Document history must be tracked

- Many more rules

So not only does the simple concept of a document involve numerous database tables of data, but there is a lot of business logic that occurs just to create it. There is no 1:1 mapping for this to occur, and because our system also has feature modules that must be separated, we have different types of Document business entities through inheritance and the decorator pattern.

We also have a database with around 200 database tables. We must maintain a very clean and optimized database. It is not possible to make business logic decisions based on the database, nor the other way around. They must be separate so we can maintain each in their own light, based on their own needs. The domain entities need to provide a rich domain model that makes sense, and the database must be optimized, correct, and fast.

The challenge you are facing is a complex one - which is the unfortunately reality that building software is not a trivial task, and unfortunately business logic and "data" are really one in the same - in that you cannot have one without the other. It just so happens that to actually build software, we must deal with this in a way that makes sense, scales both feature-wise, but performance-wise.

The challenge with all of this is that software has so many needs. So how do you have a rich domain model, use an ORM like EF, and also be able to solve problems like query for data in ways that don't fit into your domain

model? And how do you do this in a way where your codebase is not fucked up?

Largely for me with something like EF, it means finding a way to be able to easily create domain objects from your data (EF entities) while piggy backing on the unit of work pattern, context, change tracking, et cetera. To me this looks like a domain object that has access to the entity, tracked by a context, within a unit of work. This means that you have a method of loading a single entity into a domain object, or loading many entities in a single query expression for performance, and still using EF to do the change tracking for inserting and updating.

My first approach to this problem was actually to use Active Record, which is a natural fit with EF. Each domain entity has its own context, and manages all the business logic for creation, deletion, and updates. For 90% of the software this is great and a non-issue for performance. We use domain services with their own context for advanced query scenarios. I can't describe our entire approach here because it warrants a blog of its own, but...

I am also now researching a way to not use Active Record. I think repositories are key, being able to load many and handle query scenarios, and possibly specifications.

My point is though, I believe that if you truly need DDD, your data layer and domain layer should be completely

separate, and if you don't feel that way, you likely don't need DDD.

Share Improve this answer

Follow

answered Oct 27, 2019 at 2:37

David Anderson

13.7k ● 5 ● 52 ● 80

4

Oh no I wouldn't add that extra layer at all.

NHibernate and Entity Framework Code-First (I'd use EF) are designed to solve this exact problem - mapping your domain objects to your relational model (which don't have the same constraints on their design, so might, and probably will, be a different shape).

Just seems a shame to waste the excellent mapping abilities of EF and replace it with something else, even AutoMapper.

Share Improve this answer

Follow

answered Nov 16, 2013 at 21:38

Neil Barnwell

42k ● 31 ● 154 ● 226

21 Totally disagree with this. Any ORM is just a 'fake' OOP database on top of a RDBMS. It has nothing to do with domain objects mapping it has everything to do with data. Tying your domain objects to ORM entities is one of the BIGGEST anti-patterns right now which leads to complicated code and a lot of head aches as the ORM becomes the spine of the app. The repository pattern has this purpose: to decouple the Domain from the Persistence. Only if your app is CRUD you can use directly the ORM. ORM is for mapping

tables to POCOs and nothing more. – MikeSW Nov 18, 2013 at 8:51

6 I'm not sure we have the same definition of the repository pattern in mind. Sure, the repo pattern gives a layer of abstraction over data access, but the point here is *what are the objects that are returned?*. You're suggesting to query a database, using the ORM to map to objects, then use another mapping tool to map those objects to more objects? Maybe I should have specifically said EF **Code First**, where the *very idea* is that you write your domain objects first, then write one set of mapping to persist them to a DB.
– Neil Barnwell Nov 18, 2013 at 9:25

13 The Repo always returns application context (in this case Domain) objects. EF Code first is just marketing bs, no ORM can do magic and one almost ends up modifying the domain object to meet up orm needs (virtual properties, must-have setters etc). You either model a business concept or you're serving the ORM, you really can't do both properly. Even worse, a lot of devs still use the relational mindset (one to many, many to many) when modelling domain and the ORM supports (IMO requires) that mindset. Writing domain objects first isn't just a technical step, it means you don't care about db – MikeSW Nov 18, 2013 at 10:12

5 And with Code First there's no actual *coupling* to an ORM because the mapping is in a separate class, so the domain objects are still persistence-ignorant. What am I missing?
– Neil Barnwell Nov 18, 2013 at 17:59

7 See above: must-have setters for properties. It's not a big deal, but little things creep in and really, why should I care about how the ORM works when defining the domain model? Ditto for nosql db as well. When you'll have trouble mapping the domain to tables, the human nature kicks in and starts to adjust the domain model to become more compatible with the ORM. If the dev lacks the iron discipline to maintain the decoupling and caves in to use the silver bullet like

'everyone' does,everything can go wrong. And all EFCF exmaples are showing data structs, never a real domain object. – MikeSW Nov 19, 2013 at 16:35