Using View-Models with Repository pattern

20

9

I'm using Domain driven N-layered application architecture with `EF code first` in my recent project, I defined my `Repository` contracts, In `Domain` layer. A basic contract to make other `Repositories` less verbose:

```
public interface IRepository<TEntity, in TKey> where TEntity : class
{
   TEntity GetById(TKey id);
   void Create(TEntity entity);
   void Update(TEntity entity);
   void Delete(TEntity entity);
}
```

And specialized `Repositories` per each `Aggregation root` , e.g:

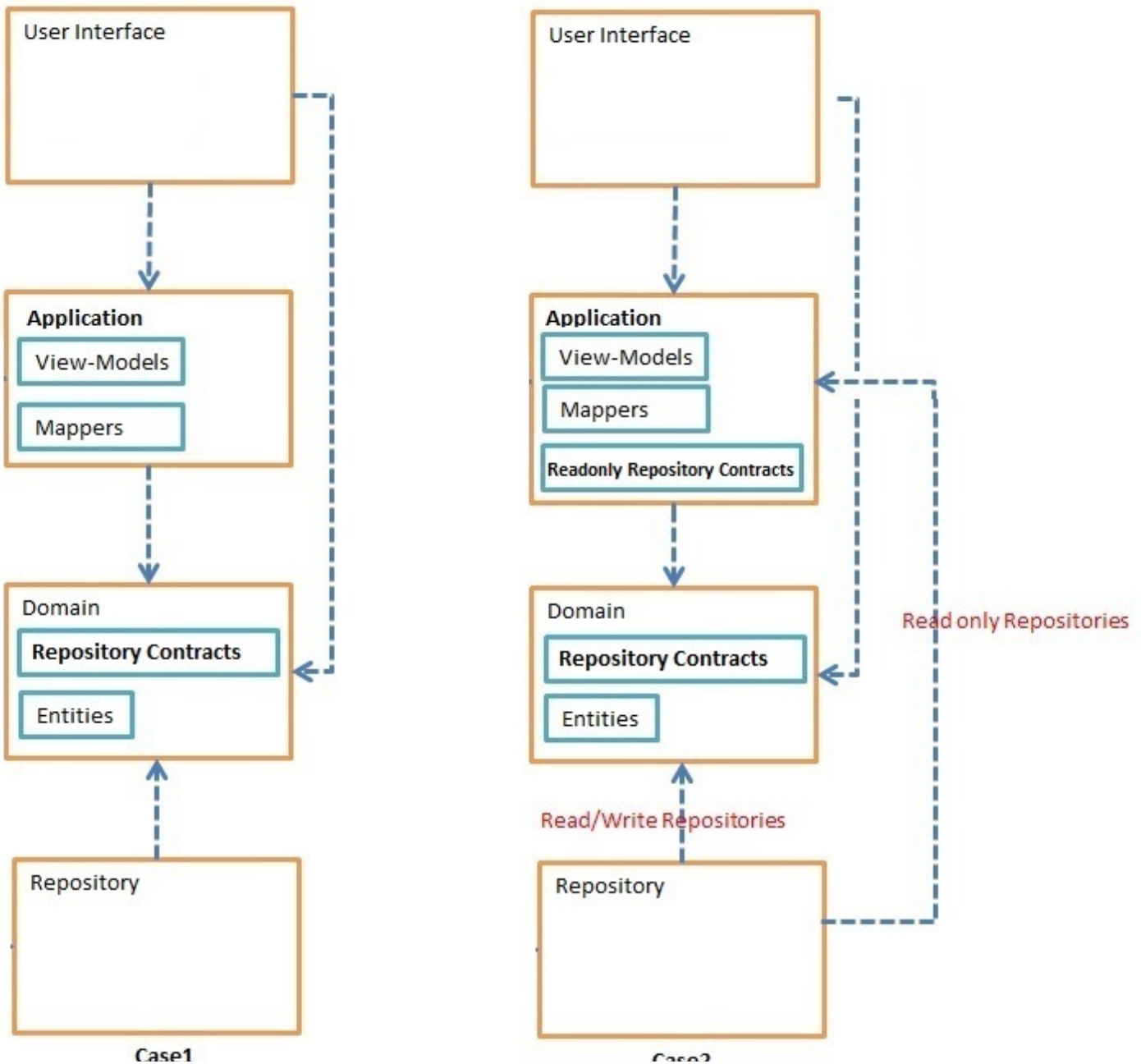
```
public interface IOrderRepository : IRepository<Order, int>
{
    IEnumerable<Order> FindAllOrders();
    IEnumerable<Order> Find(string text);
    //other methods that return Order aggregation root
}
```

As you see, all of these methods depend on `Domain entities` . But in some cases, an application's `UI` , needs some data that isn't `Entity` , that data may made from two or more enteritis's data(`View-Model` s), in these cases, I define the `View-Model` s in `Application layer` , because they are closely depend on an `Application's` needs and not to the `Domain` .

So, I think I have 2 way's to show data as `View-Models` in the `UI` :

1. Leave the specialized `Repository` depends on `Entities` only, and map the results of `Repositories` 's method to `View-Models` when I want to show to user(in `Application Layer` usually).

`Application Layer` unlike the other `Repositories` 'e contract that put in `Domain`).



Suppose, my `UI` needs a `View-Model` with 3 or 4 properties(from 3 or 4 **big** `Entities`). It's data could be generate with simple projection, but in case 1, because my methods could not access to `View-Models`, I have to **fetch all the fields of all 3 or 4 tables** with sometimes, huge joins, and then map the results to `View-Models`. But, in case 2, I could simply use projection and fill the `View-Model`s directly.

So, I think in performance point of view, the case 2 is better than case 1. but I read that `Repository` should depend on `Entities` and not `View-Models` in design point of view.

Is there any better way that does not cause the `Domain` Layer depend on the `Application layer`, and also doesn't hit the performance? or is it acceptable that for reading queries, my `Repositories` depend on `View-Models` ?(case2)

c# entity-framework ef-code-first repository viewmodel

edited May 11 '14 at 7:45

asked Apr 25 '14 at 8:26

Masoud
4,156 7 44 98

Rename "specialized repository" into "service" and make a service use repositories to translate *specifications* (like `string text`) into LINQ statements that return domain objects or projections from entities. – Gert Arnold Apr 27 '14 at 13:25

@GertArnold: using your method sometimes cause, first, we fetch some unneeded fields to memory and then select few of them. – Masoud Apr 27 '14 at 14:26

3 Answers

21

+50

Perhaps using the **command-query separation** (at the application level) might help a bit.

You should make your repositories dependent on entities only, and keep only the *trivial* retrieve method - that is, *GetOrderById()* - on your repository (along with create / update / merge / delete, of course). Imagine that the entities, the repositories, the domain services, the user-interface commands, the application services that handles those commands (for example, a certain web controller that handles POST requests in a web application etc.) represents your **write model**, the **write-side** of your application.

Then build a separate **read model** that could be as dirty as you wish - put there the joins of 5 tables, the code that reads from a file the number of stars in the Universe, multiplies it with the number of books starting with A (after doing a query against Amazon) and builds

The **separation of reads and writes** should decrease the complexity of the program and make everything a bit more manageable. And you may also see that it won't break the design rules you have mentioned in your question (hopefully).

From a performance point of view, using a **read model**, that is, writing the code that **reads** data separately from the code that **writes / changes** data is as best as you can get :) This is because you can even mangle some SQL code there without sleeping bad at night - and SQL queries, if written well, will give your application a considerable speed boost.

Nota bene: I was joking a bit on what and how you can code your read side - the read-side code should be as **clean** and simple as the write-side code, of course :)

Furthermore, you may get rid of the *generic repository* interface if you want, as it just clutters the domain you are modeling and forces every concrete repository to expose methods that are not necessary :) See this. For example, it is highly probable that the Delete() method would never be used for the *OrderRepository* - as, perhaps, Orders should never be deleted (of course, as always, it depends). Of course you can keep the *database-row-managing* primitives in a single module and reuse those primitives in your concrete repositories, but to not expose those primitives to anyone else but the implementation of the repositories - simply because they are not needed anywhere else and may confuse a drunk programmer if publicly exposed.

Finally, perhaps it would be also beneficial to not think about *Domain Layer*, *Application Layer*, *Data Layer* or *View Models Layer* in a too strict manner. Please read this. Packaging your software modules by their real-world meaning / purpose (**or feature**) is a bit better than packaging them based on an *unnatural*, *hard-to-understand*, *hard-to-explain-to-a-5-old-kid* criterion, that is, packaging them **by layer**.

edited Apr 29 '14 at 21:13

answered Apr 29 '14 at 20:54

turdus-merula
4,738 5 27 41

Its great idea, but who define the structure of `DTO` s (or as I said `View-Models`)? the `Domain Layer` or `Application Layer` ? I think their depends on application requirements and so should be defined by `Application Layer` . what do you think? – Masoud May 3 '14 at 6:18

1 @Masoud, perhaps the *Domain Layer* is not a good place for them, as your intuition says. If you *really like* layering so much :), you may build a View-Models layer, that would be part of the **read-side** of your application. – turdus-merula May 3 '14 at 6:55

@Masoud, also read this and take a look on this. – turdus-merula May 3 '14 at 7:04

You said "it would be also beneficial to not think about[layering]", I don't get the idea yet, do you have any concrete sample?, in the other word how could I arrange my classes in your way, instead of layering? – Masoud May 3 '14 at 7:47

@Masoud, the other choice is to arrange your classes *by feature*. Here are some useful resources: 1, 2, 3, 4, 5. – turdus-merula May 3 '14 at 8:04

1

Well, to me I would map the ViewModel into Model objects and use those in my repositories for reading / writing purposes, as you may know there are several tools that you can do for this, in my personal case I use automapper which I found very easy to implement.

try to keep the dependency between the Web layer and Repository layers as separate as possible saying that repos should talk only to the model and your web layer should talk to your view models.

An option could be that you can use DTO's in a service and automap those objects in the web layer (it could be the case to be a one to one mapping) the disadvantage is that you may end up with a lot of boilerplate code and the dtos and view models could feel duplicated.

the other option is to return partial hydrated objects in your model and expose those objects as DTOs and map those objects to your view models this solution could be a little bit obscure but you can make the projections you want and return only the information that you need.

you can get rid of the of the view models and expose the dtos in your web layer and use them as view models, less code but more coupled approach.

edited Apr 30 '14 at 16:29

answered Apr 29 '14 at 22:49

jack.the.ripper
10.9k 8 55 104

Thanks Pedro, what do you mean with "partial hydrated objects"? – Masoud May 3 '14 at 2:58

Fill the properties that you need for any (not all of them) entity/class with a linq proyection – jack.the.ripper May 3 '14 at 4:14

1

I Kindof agree with Pedro here. using a application service layer could be benificial. if your aiming for an MVVM type of implementation i would advise to create a Model class that is responsible for holding the data that is retrieved using the service layer. Mapping the data with automapper is a really good idea if your entities, DTO's and models are named consistently (so you don't have to write a lot of manual mappings).

if your data is not changing that often you might want to consider introducing (sql/database) views that will transfer some of the heavy lifting to the database (where it is highly optimized). EF handles database views fairly well. Then retrieving the entity and mapping the data (from the views) to the model or DTO becomes fairly straightforward.

answered Apr 30 '14 at 18:53

Didier Caron
493 3 9