How should I implement the repository pattern for complex object models?

Asked 7 years, 4 months ago Active 2 years, 7 months ago Viewed 11k times

21

6

Our data model has almost 200 classes that can be separated out into about a dozen functional areas. It would have been nice to use domains, but the separation isn't that clean and we can't change it.

We're redesigning our DAL to use Entity Framework and most of the recommendations that I've seen suggest using a Repository pattern. However, none of the samples really deal with complex object models. Some implementations that I've found suggest the use of a repository-per-entity. This seems ridiculous and un-maintainable for large, complex models.

Is it really necessary to create a UnitOfWork for each operation, and a Repository for each entity? I could end up with thousands of

asp.net-mvc entity-framework design-patterns

edited Oct 16 '12 at 17:45
Robert Harvey
181k 48 418 625

asked Oct 16 '12 at 17:43

Eric Falsken
313 1 2 5

3 Answers

6

First, you will run through every entity that you have and ask yourself :

> Does it make sense to persist this entity alone?

What I mean here, is that in your object model, some entities are likely contained in others in a master-detail relationship. If your entity is highly dependant on another, do not create a repository for this entity alone because it won't likely be persisted by itself. Instead, you will create a repository for each parent class, and you will make sure that the child entities and other relations are persisted at the same time.

I don't like how they implemented the UnitOfWork pattern in the link you provided. I suggest you to do something more along these lines. You don't have to define one class per repository. The same class can be used for every repository, each having their own instance of the UnitOfWork class for the lifetime of the operation. You can also reuse the same UnitOfWork in different repositories for changes that you want persisted at the same time.

edited Oct 16 '12 at 19:09

answered Oct 16 '12 at 19:03

marco-fiset
7,793 8 30 46

Nice link! I'm checking it out. – Eric Falsken Oct 17 '12 at 0:47

Your link (and answer) were the most helpful although I ended up not using the patterns provided. I ended up using a wrapper class that proxied the Add/Attach/Remove/SaveChanges methods to work as a generic repository, and then added a Query/QuerySingle/QueryAll method that accepts custom query classes to hold my query implementation. This is working out nicely so that I can do both LINQ and T-SQL queries without getting EF directly involved. – Eric Falsken Oct 23 '12 at 21:21

Entity Framework, with its use of transactions, is *already* an implementation of the Unit of Work pattern. – Greg Burghardt Oct 26 '15 at 13:44

1 link is dead Newtonian Jul 2 '17 at 16:42

4

Maybe you should have a look at Domain-Driven Design. It recommends grouping your objects in Aggregates and creating repositories only for the Root entity of each Aggregate. It's a nice way to bring order to a big complex object model.

Your different functional areas could be Bounded Contexts. There are various techniques to deal with overlapping Bounded Contexts if the separation between them isn't clear.

answered Oct 18 '12 at 16:11

guillaume31
7,407 15 28

0

What theoretical foundation have you found for "Repository pattern" database design? I'm guessing none. It's a layer of complexity resting on a bogus premise: that the "persistence layer" is something you need to be isolated from. From what I can tell, it plays to widespread programming ignorance of relational design principles, and panders to a presupposed centrality of the application's OO design. But it doesn't justify its existence on rational, let alone theoretical, grounds.

The One True Path to database design hasn't changed in 30 years: analyze the data. Find the keys and constraints and many-to-one relationships. Design your tables according to Boyce-Codd normal form. Use views and stored procedures to facilitate data access.

Unloved though it may be, a SQL DBMS provides a better isolation layer than anything the programmer can provide. It's true that as the database changes the SQL used to access it might have to change. But note that that's true *regardless of whether or not there's a mediation layer*. As long as the application uses views and stored procedures to access the DBMS, ordinary SQL engineering tools will indicate when changes to the underlying database invalidate those views and stored procedures.

Therein lies the challenge, of course. A mediation layer provides the illusion of isolation and the comfort to the programmer of writing code, which he knows how to do. Learning database design and SQL requires learning something new. Most people would rather die than think, and many succeed.

Someone on your team will doubtless object that writing SQL to support your 200 classes is a lot of work. No doubt. But at least it's *useful* work. If you design a database by mimicking your OO design, you will also do a lot of work, much of it useless, and defeat most of what the DBMS offers you.

For example, you contemplate reflecting Unit of Work in the database (as table or tables, I presume). *Unit of work* is a built-in service of the DBMS: `begin transaction` ... update database ... `commit transaction` . Nothing to model, other than the mapping of your classes to the database tables in SQL.

answered Jul 2 '17 at 21:14

James K. Lowden
117 1