

Well designed query commands and/or specifications

Asked 11 years, 11 months ago Modified 7 years, 9 months ago

Viewed 28k times



96



I've been searching for quite some time for a good solution to the problems presented by the typical Repository pattern (growing list of methods for specialized queries, etc.. see:

<http://ayende.com/blog/3955/repository-is-the-new-singleton>).

I really like the idea of using Command queries, particularly through use of the Specification pattern. However, my problem with specification is that it only relates to the criteria of simple selections (basically, the where clause), and does not deal with the other issues of queries, such as joining, grouping, subset selection or projection, etc.. basically, all the extra hoops many queries must go through to get the correct set of data.

(note: I use the term "command" as in the Command pattern, also known as query objects. I'm not talking about command as in command/query separation where there is a distinction made between queries and commands (update, delete, insert))

So I'm looking for alternatives that encapsulate the entire query, but still flexible enough that you're not just swapping spaghetti Repositories for an explosion of command classes.

I've used, for instance Linqspecs, and while I find some value in being able to assign meaningful names to selection criteria, it's just not enough. Perhaps I'm seeking a blended solution that combines multiple approaches.

I am looking for solutions that others may have developed to either address this problem, or address a different problem but still satisfies these requirements. In the linked article, Ayende suggests using the nHibernate context directly, but I feel that largely complicates your business layer because it now also has to contain query information.

I'll be offering a bounty on this, as soon as the waiting period elapses. So please make your solutions bounty worthy, with good explanations and I will select the best solution, and upvote the runners up.

NOTE: I'm looking for something that is ORM based. Doesn't have to be EF or nHibernate explicitly, but those are the most common and would fit the best. If it can be easily adapted to other ORM's that would be a bonus. Linq compatible would also be nice.

UPDATE: I'm really surprised that there aren't many good suggestions here. It seems like people are either totally

CQRS, or they're completely in the Repository camp. Most of my apps are not complex enough to warrant CQRS (something with most CQRS advocates readily say that you should not use it for).

UPDATE: There seems to be a little confusion here. I'm not looking for a new data access technology, but rather a reasonably well designed interface between business and data.

Ideally, what i'm looking for is some kind of cross between Query objects, Specification pattern, and repository. As I said above, Specification pattern only deals with the where clause aspect, and not the other aspects of the query, such as joins, sub-selects, etc.. Repositories deal with the whole query, but get out of hand after a while. Query objects also deal with the whole query, but I don't want to simply replace repositories with explosions of query objects.

c#

repository-pattern

command-pattern

specification-pattern

Share

edited Jan 28, 2013 at 23:58

Improve this question

Follow

asked Jan 20, 2013 at 0:01



Erik Funkenbusch

93.4k ● 29 ● 200 ● 292

-
- 6 Fantastic question. I too would like to see what people with more experience than I suggest. I am working on a code base at the moment where the generic repository also contains overloads for Command objects or Query objects, who's structure is similar to what Ayende describes in his blog. PS: This might also attract some attention on programmers.SE. – [Simon Whitehead](#) Jan 20, 2013 at 0:55



Why not just use a repository that exposes IQueryable if you don't mind the dependency on LINQ? A common approach is a generic repository, and then when you need reusable logic above that you create a derived repository type with your additional methods. – [devdigital](#) Jan 20, 2013 at 12:34

@devdigital - Dependency on Linq is not the same as dependency upon data implementation. I would like to use Linq to objects, so I can sort or perform other business layer functions. But that doesn't mean I want dependencies on the data model implementation. What i'm really talking about here is layer/tier interface. As an example, I want to be able to change a query and not have to change it in 200 places, which is what happens if you push IQueryable directly into the business model. – [Erik Funkenbusch](#) Jan 20, 2013 at 23:20

-
- 1 @devdigital - which basically just moves the problems with a repository into your business layer. You're just shuffling the problem around. – [Erik Funkenbusch](#) Jan 21, 2013 at 0:10

-
- 1 @MystereMan Take a look at these 2 articles: blog.gauffin.org/2012/10/griffin-decoupled-the-queries and cuttingedge.it/blogs/steven/pivot/entry.php?id=92 – [david.s](#) Jan 21, 2013 at 7:55





103



+500



Disclaimer: Since there aren't any great answers yet, I decided to post a part from a great blog post I read a while ago, copied almost verbatim. You can find the full blog post [here](#). So here it is:

We can define the following two interfaces:

```
public interface IQuery<TResult>
{
}

public interface IQueryHandler<TQuery, TResult> where
{
    TResult Handle(TQuery query);
}
```

The `IQuery<TResult>` specifies a message that defines a specific query with the data it returns using the `TResult` generic type. With the previously defined interface we can define a query message like this:

```
public class FindUsersBySearchTextQuery : IQuery<User[]>
{
    public string SearchText { get; set; }
    public bool IncludeInactiveUsers { get; set; }
}
```

This class defines a query operation with two parameters, which will result in an array of `User` objects. The class that handles this message can be defined as follows:

```

public class FindUsersBySearchTextQueryHandler
    : IQueryHandler<FindUsersBySearchTextQuery, User[]>
{
    private readonly NorthwindUnitOfWork db;

    public FindUsersBySearchTextQueryHandler(Northwind
    {
        this.db = db;
    }

    public User[] Handle(FindUsersBySearchTextQuery qu
    {
        return db.Users.Where(x =>
x.Name.Contains(query.SearchText)).ToArray();
    }
}

```

We can now let consumers depend upon the generic `IQueryHandler` interface:

```

public class UserController : Controller
{
    IQueryHandler<FindUsersBySearchTextQuery, User[]>
    findUsersBySearchTextHandler;

    public UserController(
        IQueryHandler<FindUsersBySearchTextQuery, User
        findUsersBySearchTextHandler)
    {
        this.findUsersBySearchTextHandler = findUsersB
    }

    public View SearchUsers(string searchString)
    {
        var query = new FindUsersBySearchTextQuery
        {
            SearchText = searchString,
            IncludeInactiveUsers = false
        };

        User[] users = this.findUsersBySearchTextHandl
    }
}

```

```
        return View(users);  
    }  
}
```

Immediately this model gives us a lot of flexibility, because we can now decide what to inject into the `UserController`. We can inject a completely different implementation, or one that wraps the real implementation, without having to make changes to the `UserController` (and all other consumers of that interface).

The `IQuery<TResult>` interface gives us compile-time support when specifying or injecting `IQueryHandlers` in our code. When we change the `FindUsersBySearchTextQuery` to return `UserInfo[]` instead (by implementing `IQuery<UserInfo[]>`), the `UserController` will fail to compile, since the generic type constraint on `IQueryHandler<TQuery, TResult>` won't be able to map `FindUsersBySearchTextQuery` to `User[]`.

Injecting the `IQueryHandler` interface into a consumer however, has some less obvious problems that still need to be addressed. The number of dependencies of our consumers might get too big and can lead to constructor over-injection - when a constructor takes too many arguments. The number of queries a class executes can change frequently, which would require constant changes into the number of constructor arguments.

We can fix the problem of having to inject too many `IQueryHandlers` with an extra layer of abstraction. We

create a mediator that sits between the consumers and the query handlers:

```
public interface IQueryProcessor
{
    TResult Process<TResult>(IQuery<TResult> query);
}
```

The `IQueryProcessor` is a non-generic interface with one generic method. As you can see in the interface definition, the `IQueryProcessor` depends on the `IQuery<TResult>` interface. This allows us to have compile time support in our consumers that depend on the `IQueryProcessor`. Let's rewrite the `UserController` to use the new `IQueryProcessor`:

```
public class UserController : Controller
{
    private IQueryProcessor queryProcessor;

    public UserController(IQueryProcessor queryProcess
    {
        this.queryProcessor = queryProcessor;
    }

    public View SearchUsers(string searchString)
    {
        var query = new FindUsersBySearchTextQuery
        {
            SearchText = searchString,
            IncludeInactiveUsers = false
        };

        // Note how we omit the generic type argument,
        // but still have type safety.
        User[] users = this.queryProcessor.Process(que

        return this.View(users);
    }
}
```



```
}  
}
```

The `UserController` now depends on a `IQueryProcessor` that can handle all of our queries. The `UserController`'s `SearchUsers` method calls the `IQueryProcessor.Process` method passing in an initialized query object. Since the `FindUsersBySearchTextQuery` implements the `IQuery<User[]>` interface, we can pass it to the generic `Execute<TResult>(IQuery<TResult> query)` method. Thanks to C# type inference, the compiler is able to determine the generic type and this saves us having to explicitly state the type. The return type of the `Process` method is also known.

It is now the responsibility of the implementation of the `IQueryProcessor` to find the right `IQueryHandler`. This requires some dynamic typing, and optionally the use of a Dependency Injection framework, and can all be done with just a few lines of code:

```
sealed class QueryProcessor : IQueryProcessor  
{  
    private readonly Container container;  
  
    public QueryProcessor(Container container)  
    {  
        this.container = container;  
    }  
  
    [DebuggerStepThrough]  
    public TResult Process<TResult>(IQuery<TResult> qu  
    {  
        var handlerType = typeof(IQueryHandler<,>)
```

```
        .MakeGenericType(query.GetType(), typeof(T  
        dynamic handler = container.GetInstance(handle  
        return handler.Handle((dynamic)query);  
    }  
}
```

The `QueryProcessor` class constructs a specific `IQueryHandler<TQuery, TResult>` type based on the type of the supplied query instance. This type is used to ask the supplied container class to get an instance of that type. Unfortunately we need to call the `Handle` method using reflection (by using the C# 4.0 `dynamic` keyword in this case), because at this point it is impossible to cast the handler instance, since the generic `TQuery` argument is not available at compile time. However, unless the `Handle` method is renamed or gets other arguments, this call will never fail and if you want to, it is very easy to write a unit test for this class. Using reflection will give a slight drop, but is nothing to really worry about.

To answer one of your concerns:

So I'm looking for alternatives that encapsulate the entire query, but still flexible enough that you're not just swapping spaghetti Repositories for an explosion of command classes.

A consequence of using this design is that there will be a lot of small classes in the system, but having a lot of

small/focused classes (with clear names) is a good thing. This approach is clearly much better than having many overloads with different parameters for the same method in a repository, as you can group those in one query class. So you still get a lot less query classes than methods in a repository.

Share Improve this answer

edited Aug 11, 2014 at 10:32

Follow



Steven

172k ● 25 ● 346 ● 447

answered Jan 25, 2013 at 9:07



david.s

11.4k ● 6 ● 52 ● 83

2 Looks like you get the award. I do like the concepts, I was just hoping for someone to present something truly different. Congrats. – Erik Funkenbusch Jan 29, 2013 at 3:07

1 @FuriCuri, does a single class really need 5 queries? Perhaps you could look at that as being a class with too many responsibilities. Alternatively, if the queries are being aggregated then maybe they should actually be a single query. These are just suggestions, of course. – Sam Oct 1, 2013 at 3:03 ✎

1 @stakx You're absolutely right that in my initial example the generic `TResult` parameter of the `IQuery` interface is not useful. However, in my updated response, the `TResult` parameter is used by the `Process` method of the `IQueryProcessor` to resolve the `IQueryHandler` at runtime. – david.s Oct 14, 2013 at 8:47

1 I also have a blog with a very similar implementation which makes me think I'm on the right path, this is the link jupaol.blogspot.mx/2012/11/... and I have been using it for a

while in PROD applications, but I have had a problem with this approach. **Chaining and reusing queries** Let's say that I have a several small queries that need to be **combined** to create more complex queries, I ended up just duplicating the code but I'm looking for a much better and cleaner approach. Any ideas? – [Jupaol](#) Dec 9, 2013 at 16:12

- 4 @Cemre I ended up encapsulating my queries in Extension methods returning `IQueryable` and making sure to not enumerate the collection, then from the `QueryHandler` I just called/chain the queries. This gave me the flexibility to unit test my queries and chain them. I have an application service on top of my `QueryHandler`, and my controller is in charge to talk directly with the service instead of the handler – [Jupaol](#) Aug 13, 2015 at 0:59
-



4



My way of dealing with that is actually simplistic and ORM agnostic. My view for a repository is this: The repository's job is to provide the app with the model required for the context, so the app just asks the repo for **what** it wants but doesn't tell it **how** to get it.



I supply the repository method with a Criteria (yes, DDD style), which will be used by the repo to create the query (or whatever is required - it may be a webservice request). Joins and groups imho are details of how, not the what and a criteria should be only the base to build a where clause.

Model = the final object or data structure needed by the app.

```

public class MyCriteria
{
    public Guid Id {get;set;}
    public string Name {get;set;}
    //etc
}

public interface Repository
{
    MyModel GetModel(Expression<Func<MyCriteria, boo
    }

```

Probably you can use the ORM criteria (Nhibernate) directly if you want it. The repository implementation should know how to use the Criteria with the underlying storage or DAO.

I don't know your domain and the model requirements but it would be strange if the best way is that the app to build the query itself. The model changes so much that you can't define something stable?

This solution clearly requires some additional code but it doesn't couple the rest of the to an ORM or whatever you're using to access the storage. The repository does its job to act as a facade and IMO it's clean and the 'criteria translation' code is reusable

Share Improve this answer

answered Jan 20, 2013 at 8:40

Follow




MikeSW

16.3k ● 3 ● 41 ● 53

- 1 This does not address the problems of repository growth, and having an ever expanding list of methods to return various

kinds of data. I understand you may not see a problem with this (many people don't), but others see it differently (I suggest reading the article I linked to, there are lots of other people with similar opinions). – [Erik Funkenbusch](#) Jan 21, 2013 at 8:39

- 1 I does address it, because the criteria makes lots of methods unnecessary. Of course, not of all of them I can't say much without knowing anything about the thing you need. I'm under the impression though that you want to query directly the db so, probably a repository is just in the way. If you need to work directly with the relational storage, go for it directly, no need for a repository. And as a note, it's annoying how many people quote Ayende with that post. I don't agree with it and I think that many devs are just using the pattern the wrong way. – [MikeSW](#) Jan 21, 2013 at 9:16 
-

- 1 It may reduce the problem somewhat, but given a large enough application it will still create monster repositories. I don't agree with Ayende's solution of using nHibernate directly in the main logic, but I do agree with him about the absurdity of out of control repository growth. I'm not wanting to directly query the database, but I don't just want to move the problem from a repository to an explosion of query objects either. – [Erik Funkenbusch](#) Jan 21, 2013 at 9:46
-



I've done this, supported this and undone this.

2



The major problem is this: no matter how you do it, the added abstraction does not gain you independence. It will leak by definition. In essence, you're inventing an entire layer just to make your code look cute... but it does not reduce maintenance, improve readability or gain you any type of model agnosticism.



The fun part is that you answered your own question in response to Olivier's response: "this is essentially duplicating the functionality of Linq without all the benefits you get from Linq".

Ask yourself: how could it not be?

Share Improve this answer

answered Jan 28, 2013 at 19:43

Follow



Stu

15.8k ● 4 ● 45 ● 74

Well, I've definitely experienced the problems of integrating Linq into your business layer. It is very powerful, but when we make data model changes it's a nightmare. Things are improved with repositories, because I can make the changes in a localized place without affecting the business layer much (other than when you have to also change the business layer to support the changes). But, repositories become these bloated layers that violate SRP massively. I understand your point, but it doesn't really solve any problems either.

– Erik Funkenbusch Jan 28, 2013 at 20:06

If your data layer uses LINQ, and data model changes require changes to your business layer... you're not layering properly. – Stu Jan 28, 2013 at 22:11

I thought you were saying you no longer added that layer. When you say the added abstraction doesn't gain you anything, that implies you are in agreement with Ayende about passing the nHibernate session (or EF context) directly into your business layer. – Erik Funkenbusch Jan 28, 2013 at 22:30



1



You can use a fluent interface. The basic idea is that methods of a class return the current instance this very class after having performed some action. This allows you to chain method calls.

By creating an appropriate class hierarchy, you can create a logical flow of accessible methods.

```
public class FinalQuery
{
    protected string _table;
    protected string[] _selectFields;
    protected string _where;
    protected string[] _groupBy;
    protected string _having;
    protected string[] _orderByDescending;
    protected string[] _orderBy;

    protected FinalQuery()
    {
    }

    public override string ToString()
    {
        var sb = new StringBuilder("SELECT ");
        AppendFields(sb, _selectFields);
        sb.AppendLine();

        sb.Append("FROM ");
        sb.Append("[").Append(_table).AppendLine("]");

        if (_where != null) {
            sb.Append("WHERE").AppendLine(_where);
        }

        if (_groupBy != null) {
            sb.Append("GROUP BY ");
            AppendFields(sb, _groupBy);
            sb.AppendLine();
        }
    }
}
```



```

        if (_having != null) {
            sb.Append("HAVING").AppendLine(_having);
        }

        if (_orderBy != null) {
            sb.Append("ORDER BY ");
            AppendFields(sb, _orderBy);
            sb.AppendLine();
        } else if (_orderByDescending != null) {
            sb.Append("ORDER BY ");
            AppendFields(sb, _orderByDescending);
            sb.Append(" DESC").AppendLine();
        }

        return sb.ToString();
    }

    private static void AppendFields(StringBuilder sb,
    {
        foreach (string field in fields) {
            sb.Append(field).Append(", ");
        }
        sb.Length -= 2;
    }
}

public class GroupedQuery : FinalQuery
{
    protected GroupedQuery()
    {
    }

    public GroupedQuery Having(string condition)
    {
        if (_groupBy == null) {
            throw new InvalidOperationException("HAVING
clause");
        }
        if (_having == null) {
            _having = " (" + condition + ")";
        } else {
            _having += " AND (" + condition + ")";
        }
    }
}

```

```

        return this;
    }

    public FinalQuery OrderBy(params string[] fields)
    {
        _orderBy = fields;
        return this;
    }

    public FinalQuery OrderByDescending(params string[] fields)
    {
        _orderByDescending = fields;
        return this;
    }
}

public class Query : GroupedQuery
{
    public Query(string table, params string[] selectFields)
    {
        _table = table;
        _selectFields = selectFields;
    }

    public Query Where(string condition)
    {
        if (_where == null) {
            _where = " (" + condition + ")";
        } else {
            _where += " AND (" + condition + ")";
        }
        return this;
    }

    public GroupedQuery GroupBy(params string[] fields)
    {
        _groupBy = fields;
        return this;
    }
}

```

You would call it like this

```
string query = new Query("myTable", "name", "SUM(amazon  
    .Where("name LIKE 'A%'")  
    .GroupBy("name")  
    .Having("COUNT(*) > 2")  
    .OrderBy("name")  
    .ToString());
```

You can only create a new instance of `Query`. The other classes have a protected constructor. The point of the hierarchy is to "disable" methods. For instance, the `GroupBy` method returns a `GroupedQuery` which is the base class of `Query` and does not have a `Where` method (the `Where` method is declared in `Query`). Therefore it is not possible to call `Where` after `GroupBy`.

It is however not perfect. With this class hierarchy you can successively hide members, but not show new ones. Therefore `Having` throws an exception when it is called before `GroupBy`.

Note that it is possible to call `Where` several times. This adds new conditions with an `AND` to the existing conditions. This makes it easier to construct filters programmatically from single conditions. The same is possible with `Having`.

The methods accepting field lists have a parameter `params string[] fields`. It allows you to either pass single field names or a string array.

Fluent interfaces are very flexible and do not require you to create a lot of overloads of methods with different

combinations of parameters. My example works with strings, however the approach can be extended to other types. You could also declare predefined methods for special cases or methods accepting custom types. You could also add methods like `ExecuteReader` or `ExecuteScalar<T>`. This would allow you to define queries like this

```
var reader = new Query<Employee>(new MonthlyReportField  
    })  
    .Where(new CurrentMonthCondition())  
    .Where(new DivisionCondition{ DivisionType = DivisionType.DivisionTypeA })  
    .OrderBy(new StandardMonthlyReportSorting())  
    .ExecuteReader();
```

Even SQL commands constructed this way can have command parameters and thus avoid SQL injection problems and at the same time allow commands to be cached by the database server. This is not a replacement for an O/R-mapper but can help in situations where you would create the commands using simple string concatenation otherwise.

[Share](#) [Improve this answer](#)

[edited Mar 4, 2017 at 0:26](#)


[Follow](#)

answered Jan 20, 2013 at 2:12



[Olivier Jacot-Descombes](#)

112k ● 14 ● 145 ● 196

3 Hmm.. Interesting, but your solution appears to have problems with SQL Injection possibilities, and doesn't really create prepared statements for pre-compiled execution (thus performing more slowly). It could probably be adapted to fix those problems, but then we're stuck with the non-type safe dataset results and what not. I would prefer an ORM based solution, and perhaps I should specify that explicitly. This is essentially duplicating the functionality of Linq without all the benefits you get from Linq. – [Erik Funkenbusch](#) Jan 20, 2013 at 2:34 

I'm aware of these problems. This is just a quick and dirty solution, showing how a fluent interface can be constructed. In a real world solution you would probably “bake” your existing approach into a fluent interface adapted to your needs. – [Olivier Jacot-Descombes](#) Jan 20, 2013 at 16:33
