With ASP.NET MVC and Data Transfer Objects, The Bigger the Better

Create the best object for moving data from your Controller to your View.

By Peter Vogel (https://visualstudiomagazine.com/forms/emailtoauthor.aspx?AuthorItem={DD0073EE-E182-4F32-B5FB-3F183FCFA220}&ArticleItem={2EA16CD8-A7AA-4C92-A83C-45BE52C4C463})

08/29/2016

In ASP.NET MVC, the Controller is responsible for manipulating your business objects to retrieve the data to display to the user in a View. Since the typical View requires data from several business entities, you need to combine data from several sources into a Data Transfer Object (DTO) to pass the data from the Controller to the View.

MOST POPULAR

One solution is to create a DTO with just enough properties to hold the data that the View needs. This class, for example, combines Customer information with SalesOrder information for a View that displays a SalesOrder-with-some-Customer-data:

```
Public Class CustOrderDTO
Public Property CustomerId As String
Public Property CustomerName As String
Public Property CustomerAddress As Address
Public Property SalesOrderId As String
Public Property DeliveryDate As DateTime
'...more SalesOrder properties...
End Class
```

Of course, filling all of those properties is going to require a lot of assignment statements (though using Automapper (https://visualstudiomagazine.com/articles/2012/02/01/simplify-your-projections-with-automapper.aspx) can help here). You're better off to follow the model of the CustomerAddress property in my sample DTO which holds a whole Address object rather than having individual street, city, etc. fields. Rather than having lots of individual fields, just put the whole Customer and SalesOrder object in your DTO:

```
Public Class CustOrderDTO
Public Property Cust As Customer
Public Property SOrder As SalesOrder
End Class
```

There's no real cost to this approach: It will take you just as long to retrieve the whole Customer object as to retrieve two or three properties (assuming you're not retrieving any BLOB fields or your Customer object doesn't have hundreds of properties). Furthermore, the code in your Controller to fill your DTO could shrink to as little as this:

```
Dim cDTO As New CustOrderDTO
cDTO.Customer = CustomerFactory.GetCustomerById(custId)
cDTO.SalesOrder = OrderFactory.GetOrderById(SoId)
```

And, as I discuss elsewhere (https://visualstudiomagazine.com/articles/2016/05/01/simplifying-data-retrieval.aspx), with the right LINQ statement and navigation properties in place, your code might get simpler yet.

There are other benefits. I bet, for example, the first request you get after rolling out this application is to add some Customer-related data to the View. Since you're already moving that data to the View all you have to do is add some more HTML to your View. This DTO is also more likely to be useful on other views (a View that displays Customer information with some SalesOrder data).

When I propose this to my clients I occasionally get some objections that "I'm moving these huge objects around." I'm not, of course: The .NET Framework just moves around references to my Customer, SalesOrder, and CustOrderDTO objects – there's only one copy of each object in memory at any time.

About the Author

Peter Vogel is a system architect and principal in PH&V Information Services. PH&V provides full-stack consulting from UX design through object modeling to database design. Peter tweets about his VSM columns with the hashtag #vogelarticles. His blog posts on user experience design can be found at http://blog.learningtree.com/tag/ui/ (http://blog.learningtree.com/tag/ui/).

PRINTABLE FORMAT (HTTPS://VISUALSTUDIOMAGAZINE.COM/ARTICLES/2016/08/29/ASP-NET-MVC-DATA-TRANSFER-OBJECTS.ASPX?P=1)

14 Comments Visual Studio Magazine 1 Login

Recommend 2 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Michael Malcharek • 2 years ago

This is so wrong in so many ways, i can't even list them all. Why do you call them still DTO? It's just a Container full of Database-Objects with (probably) Lazy-Loading and so the View will have database access all the time. Not to meantion the dependency you know have between your "layers". What should your overall architecture look like? Ball of Mud it is. You dont have any control over the data and when it get's loaded and when. Even the classnames in the short examples are showing some bad naming conventions, where every code-reviewer should look closer.

Of course there are cost to load everything, the database doesn't have magic powers, serializing is not zero-cost-magic, transfering the data over the wire from database to your programm is not magical. References are not magical as well, a keyword to read about for starters: Garbage Collection.

I can recommend reading some books about Class Design, Software Design and Architecture. Martin Fowler maybe?

3 • Reply • Share ›

MOST POPULAR

PeterHunterVogel → Michael Malcharek • 2 years ago

I think that Kevin has summed up some of my responses but there's some other issues you raise that I just don't get. It seems to me I'm just advocating for an implementation of the DTO design pattern: Objects with no behaviors and lots of getters and setters.

I think you may be crafting an argument that entity objects shouldn't be passed from the Model ("model" here meaning what we used to call the 'middle tier business objects that model the business) to the Controller. But if, as I assume here, you have entity objects being used in the Controller, I'm just not getting most of the concerns I think you're raising.

Alternatively, I'm wondering if you're considering a scenario where one person is developing the Controller while another is developing the View and it's the responsibility of the "controller developer" to limit the "view developer's" access to data. Either way, you're going to need to make a case that passing an object from the Controller to the View is significantly different from passing an object from an application to a method in, for example, some other part of the same application or even some middle tier business object in the same domain to

see more

^ | ˅ • Reply • Share ›

Michael Malcharek ➜ PeterHunterVogel • 2 years ago

Hey Peter,

thanks for the response. I like that you are advocating the DTO-Pattern, i think it's important to have clear different between the layers and that is why i don't like the idea of bringing the model-object into it, because it soften the boundaries they should have. Missing boundaries and no clear purposes are muddy for me and i would like to avoid in software. Probably there are a lot of definitions for mud. :)

Indeed i thought that the classes that are getting passed into the DTO will be entities from database. I'm now confused what exactly this object are look like and what they intended to offer in functionality and which role they have in the complete design. I guess that is the bottom-line of my comment and reason for so much open interpretation. There are to many open ends and questions after reading this article without given a scenario or talking about pros and cons.

This article shows a way to do something and praising it with: "There's no real cost to this approach: [...]" and eliminating annoying mapping

see more

^ | ˅ • Reply • Share ›

PeterHunterVogel ➜ Michael Malcharek • 2 years ago

You know, I think if I'd returned a collection of objects instead of single objects, I wouldn't have misled so many readers. With a collection I bet more people would have assumed that I'd have realized the collection with a ToList or something.

^ | ˅ • Reply • Share ›

Kevin Dahl ➜ Michael Malcharek • 2 years ago

Do you really know and understand how things are actually working under the hood! You're statements make it appear that you don't really understand what's going on in the system regardless of the architecture and how it's layered.

When objects (not primitive types) are being used in a system they are being passed by reference so passing an object around is more efficient than passing around several primitive types not to mention easier to maintain, extend and read.

As far as passing model objects to a view, it's makes a lot of sense to do it the way Peter is advocating. If a model object does support lazy loading it means a couple of things.
1. If the data is needed, the round trip to where ever (usually db) is required regardless of when it happens, so your statement about lazy loading is a non issue.
2. IMO, the model object should not know or care how it is populated. It doesn't

understand the concept of a database or lazy loading. If lazy loading from the database were to actually occur when an object is in the view, it should be done in a way that is unknown to the model object. Conceptually, the view is never accessing the database.

Remember the HTML created and sent to the browser is rendered server side. Only the data needed from a model object is sent over the wire. In other words, if the controller returns an object with 100 properties to a view and the view only uses 2 of those properties then only the values of those 2 properties are actually sent to the browser, the other 98 values are not sent to the browser.

• Reply • Share ›

Michal Nebes Kevin Dahl • 2 years ago

Hi Kevin, I don't think you are entirely correct in some of your statements. You are right that a round trip cost is required to get the data from a database regardless of where exactly it happens, but if you rely on lazy loads the amount of such round trips may get very high.

Consider the following situation: in the DTO example above you have an object representing a sales order. Now let's say it has 100 positions and we want to display those. If we rely on lazy-loading it means a 100 additional round trips to the database. If you do not rely on lazy loads and take care to load the data yourself in a repository of sorts you can easily see that you can load everything with one query thus having 1 vs 101 queries. If one round trip takes, say, 2 ms we just saved 200 ms.

Now let's say the "Position" object is connected to a, say, "Delivery" object which we also lazy load to display some single value. Not only we have another 100 round trips but we also have all those objects in memory even though we actually only want to display some values from there.

see more

• Reply • Share ›

PeterHunterVogel Michal Nebes • 2 years ago

I think that the lazy loading issue has some legs on it. But the argument works only if you assume that "encapsulating data access in the repository" means that you'd prefer to build and return entities from the repository that are disconnected from the DbContext. That's a perfectly reasonable thing to do, of course. But if you're not, then you the lazy loading problem is as possible in the Controller as it is in the View.

The reverse is also true: if the Controller is getting its Customer and SalesOrder entities as disconnected objects then lazy loading is no more a problem in the View than it is in the Controller: You can't trigger a load from either place. I'd still claim that you might as well pass the whole SalesOrder object to the

MOST POPULAR

View as pass individual parts of it.

And, with a disconnected object, it's not clear to me that the SalesOrder object would get anywhere near 85000 bytes. It's also not clear to me that simply referencing those objects from the DTO would move them to Gen2 (if they weren't already there, of course). I'm willing to be wrong about the Gen2 issue but the mechanism isn't obvious to me.

Reply • Share ›

Michal Nebes PeterHunterVogel • 2 years ago

Hi Peter, thanks for your answer! yes, you are of course right that if the DTO only contains simple "POCO" entities that make it impossible for the View to load connected data on demand, there is actually no lazy loading problem there. I was more trying to show that IF such an object would support lazy loading it is definitely not the same performance-wise as preloading all (and only) the needed data.

My argument about the garbage collection is also strongly based on the assumption that the DTO can trigger lazy loads if a view requests it. In that case it is hard to foresee the final size of the object and chances are it will end up if not on LHO then at least in Gen2. I am not claiming it will automatically always happen, I'm just saying it is a consideration and the larger your DTO grows the bigger are the chances it will be treated as a long lived object (especially since it's passed between controller and view). Also, objects referenced in a DTO surely cannot be garbage collected as long as the DTO references them. So assuming the DTO is a long lived object, all objects it references will eventually be long lived as well.

Reply • Share ›

PeterHunterVogel Michal Nebes • 2 years ago

OK -- I misled you in the article, then! Sorry about that.

Reply • Share ›

Michal Nebes Kevin Dahl • 2 years ago

Hi Kevin, I don't think you are entirely correct in some of your statements. You are right that a round trip cost is required to get the data from a database regardless of where exactly it happens, but if you rely on lazy loads the amount of such round trips may get very high.

Consider the following situation: in the DTO example above you have an object representing a sales order. Now let's say it has 100 positions and we want to display those. If we rely on lazy-loading it means a 100 additional round trips to the database. If you do not rely on lazy loads and take care to load the data yourself in a repository of sorts you can easily see that you can load everything with one query thus having 1 vs

101 queries. If one round trip takes, say, 2 ms we just saved 200 ms.

Now let's say the "Position" object is connected to a, say, "Delivery" object which we also lazy load to display some single value. Not only we have another 100 round trips but we also have all those objects in memory even though we actually only want to display some values from there.

see more

Reply • Share ›

Joel Gallagher • 2 years ago

Another problem I didn't see mentioned in other comments here is that of the Overposting / Mass Assignment security vulnerability. (http://odetocode.com/blogs/...

What's to stop the malicious user adding fields to the return data when they submit? Your controller expects the full Customer & SalesOrder classes, so if it finds the added field that the malicious user has added to the query / cookie / url / POST data it will update accordingly

Reply • Share ›

PeterHunterVogel → Joel Gallagher • a year ago

Well, except that the problem exists if you include *any* property in the model that isn't displayed in the View (e.g. some property that you use to control the way parts of the View are displayed). Paring the model down to just the properties going into the HTML and putting anything else in the ViewBag doesn't strike me as a better solution.

Isn't the right answer to only use the data that you expect to get back and to validate that data before using it? I'm not saying sending back malicious data isn't a problem worth worrying about. I just think the appropriate solution is to validate all your input data before using it and ignore the rest. Having said that, I could see an argument around using more precisely defined DTOs as a way of limiting the "surface available to be hacked," though.

As an alternative, if you wanted to limit the data coming back you could create a separate class to use as a parameter in whatever Post method accepts data back from the browser. That 'input' class would only have properties for the data you want to use. However, you'd still need to validate the returned data you use. Keeping that input class in sync with the original model would probably be a pain, though -- I'd live in fear that I'd add some data annotation to the original DTO and forget to copy it to the input class.

(and sorry to take so long to respond)

Reply • Share ›

Tamayi Mlanda • 2 years ago

If you are exposing your DTO through API as well as views though, this approach might leave you exposing data that you should not be publicly exposing, or just plain sending too much (unnecessary) data down the wire. Many times, you will use the same DTO in both; your views as well as any API that you expose - such as for building a mobile version of your application. In these cases, the good old AutoMapper and creating a DTO with just enough properties to hold the data that the View/API needs might not be a bad idea at all.

Reply • Share ›

PeterHunterVogel → Tamayi Mlanda • 2 years ago

You're right in the sense that I'm leaving it to the View to decide which data to expose. As a result, I wouldn't recommend that this strategy be used for Web Services quite so blindly as I'm suggesting here (though, when it comes to designing messages in an SOA, I do prefer "chunky" messages). But in the scenario I'm using here -- moving data between a Controller and a View -- I still think it's the right way to go. If, as a result, I need a finer grained DTO for some other purpose, I'm willing to design that DTO separately.

Reply • Share ›

ALSO ON **VISUAL STUDIO MAGAZINE**

MOST POPULAR

Windows 10 Mobile Revival Effort Grows -- Visual Studio Magazine

2 comments • 14 days ago

Reader Man — I Advice to open source Windows 10 mobile, and give the community a hand in reviving.

Programming Will Always Be Hard -- Visual Studio Magazine

16 comments • 2 months ago

PeterHunterVogel — After years of writing instructions for students (and arguing with family members), I have lost all faith in …

C# Makes GitHub's Top 5 Machine Learning Languages List -- Visual …

1 comment • 13 days ago

Kent Brook — Thank you David! That was an awesome, balanced and informative post!

21 New Tools and Extensions for Visual Studio 2017 -- Visual Studio Magazine

1 comment • 15 days ago

Admirador de Damas — Visual Studio 2017 Community ?

Subscribe **Add Disqus to your siteAdd DisqusAdd**

comments powered by Disqus (http://disqus.com)