How to map View Model back to Domain Model in a POST action?

Asked 10 years, 1 month ago Active 3 years, 10 months ago Viewed 24k times

86

58

Every article found in the Internet on using ViewModels and utilizing Automapper gives the guidelines of the "Controller -> View" direction mapping. You take a domain model along with all Select Lists into one specialized ViewModel and pass it to the view. That's clear and fine.
The view has a form, and eventually we are in the POST action. Here all the Model Binders come to the scene along with *[obviously] another* View Model which is *[obviously] related* to the original ViewModel at least in the part of naming conventions for the sake of binding and validation.

How do you map it to your Domain Model?

Let it be an insert action, we could use the same Automapper. But what if it was an update action? We have to retrieve our Domain Entity from the Repository, update it's properties according to the values in the ViewModel and save to the Repository.

ADDENDUM 1 (9th of February 2010): Sometimes, assigning Model's properties is not enough. There should be taken some action against Domain Model according to the values of View Model. I.e., some methods should be called on Domain Model. Probably, there should be a kind of an Application Service layer which stands between Controller and Domain in order to process View Models...

How to organize this code and where to place it to achieve the following goals?

- keep controllers thin
- honor SoC practice
- follow Domain-Driven Design principles
- be DRY
- to be continued ...

asp.net-mvc domain-driven-design viewmodel automapper separation-of-concerns

edited Feb 9 '10 at 2:36 asked Feb 5 '10 at 8:56

Anthony Serdyukov

4 Answers

37

I use an *IBuilder* interface and implement it using the ValueInjecter

```
public interface IBuilder<TEntity, TViewModel>
{
      TEntity BuildEntity(TViewModel viewModel);
      TViewModel BuildViewModel(TEntity entity);
      TViewModel RebuildViewModel(TViewModel viewModel);
}
```

... (implementation) *RebuildViewModel* just calls `BuildViewModel(BuilEntity(viewModel))`

```
[HttpPost]
public ActionResult Update(ViewModel model)
{
   if(!ModelState.IsValid)
    {
       return View(builder.RebuildViewModel(model);
    }

   service.SaveOrUpdate(builder.BuildEntity(model));
   return RedirectToAction("Index");
}
```

btw I don't write ViewModel I write Input cuz it's much shorter, but that just not really important
hope it helps

Update: I'm using this approach now in the ProDinner ASP.net MVC Demo App, it's called IMapper now, there's also a pdf provided where this approach is explained in detail

edited Apr 28 '16 at 9:15
ZenLulz
4,775 3 31 42

answered May 5 '10 at 18:06
Omu
60.6k 83 243 388

I like this approach. One thing I'm not clear on though is the implementation of IBuilder, especially in light of a tiered application. For example, my

@Matt Murrell look at prodinner.codeplex.com I do this in there, and I call it IMapper there instead of IBuilder – Omu Aug 18 '11 at 6:16

6 I like this approach, I implemented a sample of it here: gist.github.com/2379583 – Paul Stovell Apr 13 '12 at 20:33

To my mind it not compliant with Domain Model approach. It looks like some CRUD approach for unclear requirements. Should not we use Factories (DDD) and related methods in Domain Model to convey some reasonable action? In this way we'd better load an entity from DB and update it as required, right? So it looks like it's not fully correct. – Artyom Oct 27 '15 at 16:42

7

Tools like AutoMapper can be used to update existing object with data from source object. The controller action for updating might look like:

```
[HttpPost]
public ActionResult Update(MyViewModel viewModel)
{
    MyDataModel dataModel = this.DataRepository.GetMyData(viewModel.Id);
    Mapper<MyViewModel, MyDataModel>(viewModel, dataModel);
    this.Repostitory.SaveMyData(dataModel);
    return View(viewModel);
}
```

Apart from what is visible in the snippet above:

- POST data to view model + validation is done in ModelBinder (could be exended with custom bindings)
- Error handling (i.e. catching data access exception throws by Repository) can be done by [HandleError] filter

Controller action is pretty thin and concerns are separated: mapping issues are addressed in AutoMapper configuration, validation is done by ModelBinder and data access by Repository.

answered Feb 5 '10 at 13:06

PanJanek
6,015 1 29 39

might be not enough to assign some properties to it. Probably, some actions should be performed against Domain Model according to contents of View Model. However, +1 for sharing quite good approach. – Anthony Serdyukov Feb 9 '10 at 2:30

@Anton ValueInjecter can reverse flattening ;) – Omu Jun 14 '10 at 18:50

with this approach you do not keep the controller thin, you violate SoC and DRY ... as Omu mentioned you should have an seperated layer that care for the mapping stuff. – Rookian Jun 23 '10 at 8:02

5

I would like to say that you reuse the term ViewModel for both directions of the client interaction. If you have read enough ASP.NET MVC code in the wild you have probably seen the distinction between a ViewModel and an EditModel. I think that is important.

A ViewModel represents all the information required to render a view. This could include data that is rendered in static non-interactive places and also data purely to perform a check to decide on what exactly to render. A Controller GET action is generally responsible for packaging up the ViewModel for its View.

An EditModel (or perhaps an ActionModel) represents the data required to perform the action the user wanted to do for that POST. So an EditModel is really trying to describe an action. This will probably exclude some data from the ViewModel and although related I think it's important to realize they are indeed different.

One Idea

That said you could very easily have an AutoMapper configuration for going from Model -> ViewModel and a different one to go from EditModel -> Model. Then the different Controller actions just need to use AutoMapper. Hell the EditModel could have a functions on it to validate it's properties against the model and to apply those values to the Model itself. It's not doing anything else and you have ModelBinders in MVC to map the Request to the EditModel anyway.

Another Idea

Beyond that something I have been thinking about recently that sort of works off the idea of an ActionModel is that what the client is posting back to you is actually the description of several actions the user performed and not just one big glob of data. This would certainly require some Javascript on the client side to manage but the idea is intriguing I think.

Essentially as the user performs actions on the screen you have presented them, Javascript would start create a list of action objects. An example is possibly the user is at an employee information screen. They update the last name and add a new address because the employee has recently been married. Under the covers this produces a `ChangeEmployeeName` and an `AddEmployeeMailingAddress` objects to a list. The user clicks 'Save' to commit the changes and you submit the list of two objects, each containing just the information needed to perform each action.

have methods that completed the action on the Model they work with. So the Controller action ends up just getting an Id for the Model instance to pull and a list of actions to perform on it. Or the actions have the id in them to keep them very separate.

So maybe something like this gets realized on the server side:

```
public interface IUserAction<TModel>
{
     long ModelId { get; set; }
     IEnumerable<string> Validate(TModel model);
     void Complete(TModel model);
}

[Transaction] //just assuming some sort of 2-tier with transactions handled by filter
public ActionResult Save(IEnumerable<IUserAction<Employee>> actions)
{
     var errors = new List<string>();
     foreach( var action in actions )
     {
         // relying on ORM's identity map to prevent multiple database hits
         var employee = _employeeRepository.Get(action.ModelId);
         errors.AddRange(action.Validate(employee));
     }

     // handle error cases possibly rendering view with them

     foreach( var action in editModel.UserActions )
     {
         var employee = _employeeRepository.Get(action.ModelId);
         action.Complete(employee);
         // against relying on ORMs ability to properly generate SQL and batch changes
         _employeeRepository.Update(employee);
     }

     // render the success view
}
```

That really makes the posting back action fairly generic since you are relying on your ModelBinder to get you the correct IUserAction instance and your IUserAction instance to either perform the correct logic itself or (more likely) call into the Model with the info.

If you were in a 3 tier environment the IUserAction could just be made simple DTOs to be shot across the boundary and performed in a similar method on the app layer. Depending on how you do that layer it could be split up very easily and still remain in a transaction (what comes to mind is Agatha's request/response and taking advantage of DI and NHibernate's identity map).

hope it helps and I would love to hear of other ways to manage the interactions.

answered Oct 8 '10 at 13:42
Sean Copenhaver
8,451 1 14 15

Interesting. Regarding the distinction between ViewModel and EditModel... are you sugesting that for an edit function you would use a ViewModel to create the form, and then bind to an EditModel when the user posted it? If so, how would you deal with situations where you would need to repost the form due to validation errors (for example when the ViewModel contained elements to populate a drop down) - would you just include the drop down elements in the EditModel also? In which case, what would be the difference between the two? – UpTheCreek Oct 8 '10 at 14:30

I'm guessing your concern is that if I use an EditModel and there is an error then I have to rebuild my ViewModel which could be very expensive. I would say just rebuild the ViewModel and make sure it has a place to put user notification messages (probably both positive and negative ones such as validation errors). If it turns out to be a performance problem you could always cache the ViewModel until that session's next request ended (probably being the post of the EditModel). – Sean Copenhaver Oct 8 '10 at 15:18

0

You don't need mapping viewmodel to domain because your viewmodel may be created more than domain model. Viewmodels optimized for screen (ui) and different from domain model.

http://lostechies.com/jimmybogard/2009/06/30/how-we-do-mvc-view-models/

answered May 29 '14 at 15:02
oguzh4n
673 9 27