Repository pattern and mapping between domain models and Entity Framework

Asked 6 years, 2 months ago Active 1 year ago Viewed 42k times

42

30

My repositories deal with and provide persistence for a rich domain model. I do not want to expose the anemic, Entity Framework data entity to my business layers, so I need some way of mapping between them.

In most cases, constructing a domain model instance from a data entity requires the use of parameterised constructors and methods (since it is rich). It is not as simple as a property/field match. AutoMapper could be used for the opposite situation (mapping to data entities) but not when creating domain models.

Below is the core of my repository pattern.

The `EntityFrameworkRepository` class works with two generic types:

- `TDomainModel` : The rich domain model
- `TEntityModel` : The Entity Framework data entity

Two abstract methods are defined:

- `ToDataEntity(TDomainModel)` : To convert to data entities (for `Add()` and `Update()` methods)
- `ToDomainModel(TEntityModel)` : To construct domain models (for the `Find()` method).

Concrete implementations of these methods would define the mapping required for the repository in question.

```
public interface IRepository<T> where T : DomainModel
{
    T Find(int id);
    void Add(T item);
    void Update(T item);
}

public abstract class EntityFrameworkRepository<TDomainModel, TEntityModel> :
IRepository<TDomainModel>
    where TDomainModel : DomainModel
    where TEntityModel : EntityModel
{
    }

    public virtual TDomainModel Find(int id)
    {
        var entity = context.Set<TEntityModel>().Find(id);

        return ToDomainModel(entity);
    }

    public virtual void Add(TDomainModel item)
    {
        context.Set<TEntityModel>().Add(ToDataEntity(item));
    }

    public virtual void Update(TDomainModel item)
    {
        var entity = ToDataEntity(item);

        DbEntityEntry dbEntityEntry = context.Entry<TEntityModel>(entity);

        if (dbEntityEntry.State == EntityState.Detached)
        {
            context.Set<TEntityModel>().Attach(entity);

            dbEntityEntry.State = EntityState.Modified;
        }
    }

    protected abstract TEntityModel ToDataEntity(TDomainModel domainModel);
    protected abstract TDomainModel ToDomainModel(TEntityModel dataEntity);
}
```

Here is a basic example of a repository implementation:

```
public interface ICompanyRepository : IRepository<Company>
{
    // Any specific methods could be included here
}

public class CompanyRepository : EntityFrameworkRepository<Company, CompanyTableEntity>,
ICompanyRepository
{
    protected CompanyTableEntity ToDataEntity(Company domainModel)
    {
```

```
                City = domainModel.City
                IsActive = domainModel.IsActive
            };
        }

        protected Company ToDomainModel(CompanyTableEntity dataEntity)
        {
            return new Company(dataEntity.Name, dataEntity.IsActive)
            {
                City = dataEntity.City
            }
        }
    }
```

Problem:

A `Company` might be composed of many `Departments`. If I want to eagerly load these from the `CompanyRepository` when fetching a `Company` then where would I define the mapping between a `Department` and a `DepartmentDataEntity`?

I could provide more mapping methods in the `CompanyRepository`, but this will soon get messy. There would soon be duplicated mapping methods across the system.

What is a better approach to the above problem?

.net entity-framework domain-driven-design repository-pattern onion-architecture

edited Jan 8 '14 at 7:09

asked Jan 6 '14 at 7:13
Dave New
30.2k 37 151 342

1. define the Company stricly according to the business concept. 2. Model the entity use cases 3. Note that if done properly you won't be having the problem you have now. And ignore anything related to persistence – MikeSW Jan 7 '14 at 12:17

2 *AutoMapper could be used for the opposite situation [...] but not when creating domain models.* Why is it not possible *both* ways? – Thomas Weller Jan 8 '14 at 7:45

1 @ThomasWeller, there is an answer by creator of AutoMapper, Jimmmy Bogard - The case for two-way mapping in AutoMapper. – Ilya Palkin Jan 9 '14 at 22:27

@davenewza, did you decide to use the approach described in your question or did you map directly to the domain model? – w0051977 Feb 20 '18

6 Answers

30

✓

+100

> My repositories deal with and provide persistence for a rich domain model. I do not want to expose the anemic, Entity Framework data entity to my business layers, so I need some way of mapping between them.

If you you use Entity Framework, it can map Rich Domain Model itself.

I've answered the similar question "Advice on mapping of entities to domain objects" recently.

I've been using NHibernate and know that in Entity Framework you can also specify mapping rules from DB tables to your POCO objects. It is an extra effort to develop another abstraction layer over Entity Framework entities. Let the ORM be responsible for all of the mappings, state tracking, unit of work and identity map implementation, etc. Modern ORMs know how to handle all these issues.

> AutoMapper could be used for the opposite situation (mapping to data entities) but not when creating domain models.

You are completely right.

Automapper is useful when one entity can be mapped into another without additional dependencies (e.g. Repositories, Services, ...).

> ... where would I define the mapping between a `Department` and a `DepartmentDataEntity` ?

I would put it into `DepartmentRepository` and add method `IList<Department> FindByCompany(int companyId)` in order to retreive company's departments.

> I could provide more mapping methods in the `CompanyRepository` , but this will soon get messy. There would soon be duplicated mapping methods across the system.
>
> What is a better approach to the above problem?

If it is needed to get list of `Department` s for another entity, a new method should be added to `DepartmentRepository` and simply used where it is needed.

36.5k 19 118 175 **9,313** 2 20 34

16 I hate to disagree, but I've found that EF does not properly map to a Rich Domain Model, but does best with an anemic POCO class. EF, as with other ORMs, carry many restrictions that conflict with business rules, such as visibility of setters, or requirements for constructors. Having gone down the road of creating a single entity class for both business rules and EF-purposes, I would advise against it. Use a data-layer model, and a business/domain-layer model, and map between them as needed. This will separate EF(ORM)-specific code from the domain logic, and will promote SRP. No minus though. – James Haug Dec 13 '16 at 4:16

2 I do agree that "ORMs, carry many restrictions that conflict with business rules". Although for some models those restrictions are not critical, especially when your team is disciplined. Every decision should be based on the context. – Ilya Palkin Dec 13 '16 at 9:51

5

Let's say you have the following data access object...

```
public class AssetDA
{
    public HistoryLogEntry GetHistoryRecord(int id)
    {
        HistoryLogEntry record = new HistoryLogEntry();

        using (IUnitOfWork uow = new NHUnitOfWork())
        {
            IReadOnlyRepository<HistoryLogEntry> repository = new
NHRepository<HistoryLogEntry>(uow);
            record = repository.Get(id);
        }

        return record;
    }
}
```

which returns a history log entry data entity. This data entity is defined as follows...

```
public class HistoryLogEntry : IEntity
{
    public virtual int Id
    { get; set; }

    public virtual int AssetID
    { get; set; }
    public virtual string Text
    { get; set; }

    public virtual Guid UserID
    { get; set; }

    public virtual IList<AssetHistoryDetail> Details { get; set; }
}
```

You can see that the property `Details` references another data entity `AssetHistoryDetail`. Now, in my project I need to map these data entities to Domain model objects which are used in my business logic. To do the mapping I have defined extension methods...I know it's an anti-pattern since it is language specific, but the good thing is that it isolate and breaks dependencies between each other...yeah, that's the beauty of it. So, the mapper is defined as follows...

```
internal static class AssetPOMapper
{
    internal static HistoryEntryPO FromDataObject(this HistoryLogEntry t)
    {
        return t == null ? null :
            new HistoryEntryPO()
            {
                Id = t.Id,
                AssetID = t.AssetID,
                Date = t.Date,
                Text = t.Text,
                UserID = t.UserID,
                Details = t.Details.Select(x=>x.FromDataObject()).ToList()
            };
    }

    internal static AssetHistoryDetailPO FromDataObject(this AssetHistoryDetail t)
    {
        return t == null ? null :
            new AssetHistoryDetailPO()
            {
                Id = t.Id,
                ChangedDetail = t.ChangedDetail,
                OldValue = t.OldValue,
                NewValue = t.NewValue
            };
    }
}
```

and that's pretty much it. All dependencies are in one place. Then, when calling a data object from the business logic layer I'd let `LINQ` do the rest...

```
var da = new AssetDA();
var entry =  da.GetHistoryRecord(1234);
var domainModelEntry = entry.FromDataObject();
```

Note that you can define the same to map Domain Model objects to Data Entities.

answered Jan 6 '14 at 7:43
Leo
12.9k 2 31 52

2 I am very tempted to do something like this, but then my domain models are coupled with my infrastructure layer. – Dave New Jan 6 '14 at 9:13

3 I think you're slightly overemphasizing the idea of *coupling* here - your infrastructure and your domain are 'coupled' anyway (also in your example code). As long as the mapping code resides in one location which is NOT the domain model and NOT the entity, everything is fine. – Thomas Weller Jan 8 '14 at 8:46

1 @davenewza how exactly would your domain models be coupled to your infrastructure layer? Are the domain models calling methods or instantiating concrete objects of your infrastructure layer? I havent made such a suggestion...I was talking about mappings between domain models and data entities – Leo Jan 8 '14 at 9:04

1 +1, extension methods don't make your domain layer dependent on the infrastructure. They can be declared in the infrastructure itself, or another separate module. This is equivalent to using Mapper objects residing in your infrastructure layer, only a bit more concise and technology-specific. – guillaume31 Jan 8 '14 at 14:39

4

With entity framework, across all layers IMHO, it is generally a bad idea to convert from entity models to other form of models (domain models, value objects, view models, etc.) and vice versa except only in the application layer because you will lose a lot of EF capabilities that you can only achieve through the entity objects, like loss of change tracking and loss of LINQ queryable.

It's better to do the mapping between your repository layer and the application layer. Keep the entity models in the repository layer.

answered Jan 10 '14 at 12:20
Ronald
1,490 4 18 32

3

- You can easily call extension methods for other entities when they are within a containing entity.
- You can easily deal with collections by creating IEnumerable<> extensions.

A simple example:

```
public static class LevelTypeItemMapping
{
    public static LevelTypeModel ToModel(this LevelTypeItem entity)
    {
        if (entity == null) return new LevelTypeModel();

        return new LevelTypeModel
        {
            Id = entity.Id;
            IsDataCaptureRequired = entity.IsDataCaptureRequired;
            IsSetupRequired = entity.IsSetupRequired;
            IsApprover = entity.IsApprover;
            Name = entity.Name;
        }
    }
    public static IEnumerable<LevelTypeModel> ToModel(this IEnumerable<LevelTypeItem>
entities)
    {
        if (entities== null) return new List<LevelTypeModel>();

        return (from e in entities
                select e.ToModel());
    }

}
```

...and you use them like this.....

```
 using (IUnitOfWork uow = new NHUnitOfWork())
        {
        IReadOnlyRepository<LevelTypeItem> repository = new NHRepository<LevelTypeItem>
(uow);
        record = repository.Get(id);

        return record.ToModel();

        records = repository.GetAll(); // Return a collection from the DB
```

Not perfect, but very easy to follow and very easy to reuse.

answered Jan 9 '14 at 16:10
Dave R
1,386 15 22

2

Like previous posts have said. It's probably best to wait untill after the repositories to do actual mapping.. BUT I like working with auto-mapper. It provides a very easy way to map objects to other objects. For some separation of concerns, you can also define the mappings in a separate project. These mappings are also generic / type-based:

1. You specify the base type in the mapping and define how it fills the destination type
2. Where you need the mapping, you just call Mapper.Map(baseTypeObject, DestinationTypeObject)
3. Automapper should handle the rest

This could do the trick, if I understood your question correctly.

answered Jan 10 '14 at 12:36
Tsasken
661 5 16

0

I dont like to map manually so for mapping i am using http://valueinjecter.codeplex.com/

answered Jan 8 '14 at 15:38
Volodymyr Bilyachat
13k 4 38 58