How to map Data Access to Business Logic objects in Entity Framework

Asked 5 years, 4 months ago Active 4 years, 5 months ago Viewed 11k times

2

4

I am using Entity Framework in an ASP.NET C# MVC application.

I have objects that are generated by EF in the Data Access Layer:

```
namespace Project1.DataAccess
{
    using System;
    using System.Collections.Generic;

    public partial class User
    {
        public User()
        {
            this.Files = new HashSet<File>();
            this.Folders = new HashSet<Folder>();
        }
        //...

    }
}
```

Now, I want to create Business Logic objects, and then map them with the Data Access ones:

```
namespace Project1.Logic
{
    public class User
```

I have a very small number of tables in the database. Do I need to use `Automapper`? If no, how can I achieve the mapping?

c# asp.net entity-framework

asked Nov 24 '14 at 15:55

enb081

3,089 ● 11 ● 37 ● 64

Why do you have an `Id` property in your BLL layer `User` class - out of curiosity? – wahwahwah Dec 4 '14 at 0:17

Yes, all the properties of the DA User are in the BLL User too. – enb081 Dec 4 '14 at 7:50

4 Answers

Active | Oldest | Votes

11

+50

I use EF6 all the time to generate my Data Access Layer from MSSQL tables, then I create a set of objects that represent how I want interact with my code (or display it) which are POCO. The "mapping" is taken care of through the implementation of a Repository pattern. Below is a generic interface that helps me ensure all my repo classes follow the same shape.

```
public interface IDataRepository<T>
{
    IQueryable<T> Get();
    T Get(int id);
    T Add(T obj);
    T Update(T obj);
    void Delete(T obj);
}
```

Then, I create repo classes like this. (using your UserBusiness and UserDAL classes)

```
public class NewRepo : IDataRepository<UserBusiness>
{
    YourContext db = new YourContext();
```

```
        });
    }

    public UserBusiness Get(int id)
    {
        return (from u in db.UserDAL where u.Id == id select new UserBusiness()
        {
           Id = u.Id,
           Name = u.Name
        }).FirstOrDefault();
    }

    public Order Add(UserBusiness obj)
    {
        UserDAL u= new UserDAL();
        u.Name = obj.Name;

        db.UserDAL.Add(u);
        db.SaveChanges();

        //Assuming the database is generating your Id's for you
        obj.Id = u.Id;

        return obj;

    }

    public Order Update(UserBusiness obj)
    {
        UserDAL u= new UserDAL();
        u.Id = obj.Id;
        u.Name = obj.Name;

        db.Entry(u).State = EntityState.Modified;
        db.SaveChanges();

        return obj;
    }

    public void Delete(UserBusiness obj)
    {
        UserDAL u = db.UserDAL
            .Where(o=>o.Id == obj.Id)
```

```
    }
}
```

From within your application, you'd now use the methods of your repo class instead of your DBContext.

Lastly, I often end up adding another layer of 'Service Classes' that interact with my repos which manage the internal data of the Business classes...or you could make your Business classes 'smarter' by adding the repo methods to them. My preference is to keep POCO's dumb and build service classes to get, set and edit properties.

Yes, there is a bunch of left-right mapping to "convert" one class to another, but it's a clean separation of internal business logic classes for later on. Straight table to POCO conversions seems silly at first, but just wait until your DBA want's to normalize a few fields or you decide to add a collection to those simple objects. Being able to manage your business objects without breaking the rest of you app is priceless.

Edit: Below there is generic version of the Repository, which makes creating new repositories a lot easier.

This is base class for all Business Logic Layer classes:

```
public class BaseEntity
{
    public int Id { get; set; }
}
```

This is base class for all Data Access Layer classes:

```
public class BaseEntityDAL
{
    [Key]
    [Column("Id")]
    public int Id { get; set; }
}
```

This is generic base class for repository (note, that we use AutoMapper here):

```
protected virtual TDAL Map(TBLL obj)
{
    Mapper.CreateMap<TBLL, TDAL>();
    return Mapper.Map<TDAL>(obj);
}

protected virtual TBLL Map(TDAL obj)
{
    Mapper.CreateMap<TDAL, TBLL>();
    return Mapper.Map<TBLL>(obj);
}

protected abstract IQueryable<TBLL> GetIQueryable();

public BaseRepository(MyDbContext context, DbSet<TDAL> dbSet)
{
    if (context == null)
        throw new ArgumentNullException(nameof(context));
    if (dbSet == null)
        throw new ArgumentNullException(nameof(dbSet));

    this.context = context;
    this.dbSet = dbSet;
}

public TBLL Get(int id)
{
    var entity = dbSet
        .Where(i => i.Id == id)
        .FirstOrDefault();

    var result = Map(entity);

    return result;
}

public IQueryable<TBLL> Get()
{
    return GetIQueryable();
}

public TBLL Add(TBLL obj)
```

```
        obj.Id = entity.Id;

        return obj;
    }

    public TBLL Update(TBLL obj)
    {
        var entity = Map(obj);

        context.Entry(entity).State = EntityState.Modified;
        context.SaveChanges();

        return obj;
    }

    public void Delete(TBLL obj)
    {
        TDAL entity = dbSet
            .Where(e => e.Id == obj.Id)
            .FirstOrDefault();

        if (entity != null)
        {
            context.Entry(entity).State = EntityState.Deleted;
            context.SaveChanges();
        }
    }
}
```

Finally, when we got all above, this is sample implementation of repository, nice and clean:

```
public class ContractRepository : BaseRepository<ContractDAL, Contract>
{
    protected override IQueryable<Contract> GetIQueryable()
    {
        return dbSet
            .Select(entity => new Contract()
            {
                // We cannot use AutoMapper here, because Entity Framework
                // won't be able to process the expression. Hence manual
                // mapping.
                Id = entity.Id
```

```
                Notes = entity.Notes
            });
    }

    public ContractRepository(MyDbContext context)
        : base(context, context.Contracts)
    {

    }
}
```

edited Oct 9 '15 at 19:45

Spook
21k ● 13 ● 71 ● 131

answered Dec 7 '14 at 4:14

Phil Cazella
679 ● 4 ● 11

1 I modified your class and created a generic version, which simplifies creating new repositories. I also edited your post to include these classes, such that they may be easily reused by everyone else. Hope you don't mind. – Spook Oct 9 '15 at 19:40

@Spook Your generic example is very appreciated, thank you very much. But I got a question: I was under the impression that your presentation layer (controller) should have no knowledge of your DAL, and only work with the BLL. But after implementing it, I've noticed that in order to instantiate ContractRepository in my controller and make use of it, I need to pass it an instance of my DbContext (which is my DAL), which I think I shouldn't have to do. Could you add a quick example of how you'd call your ContractRepository from your controller? – Antrim Feb 24 '16 at 16:00

1 @Antrim Use Dependency Injection, Luke :) If you teach DI container how to instantiate DB Context, it will automatically provide it to ContractRepository constructor without any action required from the presentation layer. Also, if you teach the container how to instantiate ContractRepository, you may simply pass it through an interface to controller's constructor and DI will do the rest. Take a look here: asp.net/mvc/overview/older-versions/hands-on-labs/… – Spook Feb 25 '16 at 8:36

@Spook, question if these are intended to be EF classes how do you add the BaseEntityDAL to them, where it doesn't get lost every time you rebuild the data layer? – dblwizard Jul 3 '18 at 6:04

@dblwizard From the code I shared, it seems like I've bee using Code First approach - in such case there is no rebuilding data layer (in terms, rebuilding data layer is equal to entering changes immediately in classes by myself). – Spook Jul 3 '18 at 7:54

1

If your project is relatively small, I would recommend not to use DTOs at all — instead, you can use Entity Framework Code First and reuse your business entities across multiple layers (just make sure to place Code First entities to some common library).

1

If you want to Use Plain Old Clr Objects in your Business Model Design and map them to your database tables, you may use the Code First Approach for Entity Framework. In Code first, nothing will be generated for you. However, you will have the responsability to map your Business Objects to your Database Tables and fields. You can basiscally do it with two ways:

- **Fluent API** Using this approach, you will define the relationship between your Object and relational entities in your datacontext object: Here is an example from msdn http://msdn.microsoft.com/en-us/data/jj591620.aspx
- **Data Annotations** Now Using this approach, you will map your classes to your database entities using Data Annotations that will be part of the Business Objects themselves: An example: http://www.entityframeworktutorial.net/code-first/dataannotation-in-code-first.aspx

The two methods will generate the same mapping for you, but I prefer Fluent Api method since it provides a stronger mapping API and keeps your BOs independant of any mapping logic that will be centralized in your datacontext.

But.. Once you generate classes, these will be binded and mapped for you, which is the Database first Approach. Hence, you can extend these classes since they are partial. You can find in this blog post details on the different workflows made on EF that will help you use the right one for your needs: http://blog.smartbear.com/development/choosing-the-right-entity-framework-workflow/

answered Nov 24 '14 at 16:16

Hamdi Baligh
778 ● 9 ● 27

0

If your DAL and BLL `User` objects are *exactly* the same, you can use a function like this to do the mapping:

```
public void SetProperties(object source, object target)
{
    var type = target.GetType();
    foreach (var prop in source.GetType().GetProperties())
    {
        var propGetter = prop.GetGetMethod();
        var propSetter = type.GetProperty(prop.Name).GetSetMethod();
        var valueToSet = propGetter.Invoke(source, null);
        propSetter.Invoke(target, new[] { valueToSet });
    }
```

defeats the purpose of having a DAL and a BLL in the first place.

This is an instance where using generics and interfaces can be very helpful. So, given the object `User` example:

```
public class User
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

... You could create a generic "repository" class or some other data access methodology for your objects that inherits from a data access interface **(DAL)**:

```
public class DALMethods<T> : IDALMethods<T> where T : class
    {
        private UserContext _db;
        private DbSet<T> _set;

        public DALMethods(UserContext db)
        {
            _db = db;
            _set = _db.Set<T>();
        }

        public void Create(T entity)
        {
            _set.Add(entity);
            _db.SaveChanges();
        }

        //... Expressly dispose context method needed.
    }
```

.. And then your **BLL** would just be concerned with the `User` business logic:

```
public class UserBLL : IBLLMethods<User>
{
    private DALMethods<User> _repository;
```

```
    }

    public bool CreateUserIfNameIsBob(User user)
    {
        // Create bob if bob
        if (user.Name == "Bob")
        {
            _repository.Create(user);
            return true;
        }

        // Not bob
        return false;
    }
}
```

The examples above are left purposefully generic, but I think they illustrate the point. If your `User` object changes, it won't prevent your BLL and DAL layers from working. You can use interfaces like `IDALMethods<T>` to enforce constraints on implementation, or to further decouple your code using an IoC container.

HTH

edited Dec 5 '14 at 16:42

answered Dec 4 '14 at 18:03

wahwahwah

2,701 ● 1 ● 16 ● 34