Threenine.Map Documentation

Release u

Gary Woodfine

Aug 12, 2020

User Documentation

1 What is Threenine.Map **1**

2 Why use Threenine.Map **3**

3 How to install **5**

4 Source code **7**

5 Introduction **9**

5.1 Mapping Interfaces 9

5.2 Mapping Configuration Factory 11

CHAPTER 1

What is Threenine.Map

Threenine.Map is a code library to enable easier and leaner object-object mapper. Under the hood it makes use of the most popular C# object-object mapper Automapper.

Object-object mapping works by transforming an input object of one type into an output object of a different type.

What makes AutoMapper interesting is that it provides some interesting conventions to take the dirty work out of figuring out how to map type A to type B. As long as type B follows AutoMapper's established convention, almost zero configuration is needed to map two types.

CHAPTER 2

Why use Threenine.Map

Using AutoMapper, developers typically place mapping configuration in a completely separate profile class, which creates an artificial barrier between two things that are likely to change together: The Object Model and the mapping definition.

> Threenine.Map provides and alternative approach to help keep you Automapper Configuration logic clean and closer to the object model you're mapping whether that be Domain Objects, View Models and entities.
>
> Using the magic of reflection and assembly scanning Threenine.Map will also ensure your mapping logic is available when needed.

CHAPTER 3

How to install

Threenine.map is available via a nuget package Threenine.map. and can be easily installed to your project using either via package manager :

```
Install-Package Threenine.Map
```

Or dotnet core cli :

```
dotnet add package Threenine.Map
```

CHAPTER 4

Source code

All source code for this library is available for inspection at the Threenine.map Github

CHAPTER 5

Introduction

Threenine.Map helps you do this by just implementing one of the mapping interfaces in your view models and domain models, and you're set!

The library combines all the power of Automapper to assist in implementing mappings in projects by locating the mapping logic within the Domain Objects & View Models.

Threenine.Map supports the following platforms:

- .NET 4.0
- .NET 4.5.2+
- .NET Platform Standard 2.0

5.1 Mapping Interfaces

There are three interfaces that enable the definition of mappings making;

1. *IMapTo*
2. *IMapFrom*
3. *ICustomMap*

5.1.1 IMapTo & IMapFrom

The *IMapTo<T>* and *IMapFrom<T>* interfaces enable the definition of simple mappings i.e. If you have a database entity and domain entity that may have the same properies eg.

A database entity that has two properties

```
public class DatabaseEntity
{
    public string Name {get;set;}
```

(continues on next page)

(continued from previous page)

```
    public string Description {get;set;}
}
```

A domain object with the same 2 properties with the same names

```
public class DomainObject
{
    public string Name {get;set;}
    public string Description {get;set;}
}
```

You don't necessarily want to write mapping logic to map between the two objects. You have simply make use of the *IMapTo* and *IMapFrom* interfaces to define the mappings

```
public class DatabaseEntity : IMapTo<DomainObject>
{
    public string Name {get;set;}
    public string Description {get;set;}
}
```

Once the interface has been implemented and the designated class name supplied there is nothing else you need to do to implement the mapping. The automapper libraries take care of all the mapping for you.

5.1.2 ICustomMap

The ICustomMap interface is required if your objects require more complex mapping logic. The interface requires the implementation of a mapping method

```
 public interface ICustomMap
{
    void CustomMap(IMapperConfigurationExpression configuration);
}
```

It is in the CustomMap method that you can use all the power and functionality of Automapper to implement your required mapping logic

```
 public class ComplexDomainObject  : ICustomMap
{
    public string Firstname { get; set; }
    public string LastName { get; set; }
    public string Summary { get; set; }
    public string Title { get; set; }
    public int Age { get; set; }

    public void CustomMap(IMapperConfigurationExpression configuration)
    {
        configuration.CreateMap<ComplexDomainObject, SimpleEntity>()
            .ForMember(dest => dest.Name, opt => opt.MapFrom(src => string.Concat(␣
↪src.Firstname, " ", src.LastName )))
            .ForMember(dest => dest.Description, opt => opt.MapFrom(src => string.
↪Concat( src.Title , " ", src.Summary)))
            .ForMember(dest => dest.Age, opt => opt.MapFrom(src => src.Age));

    }
}
```

5.2 Mapping Configuration Factory

The MapConfigurationFactory is the mechanism you'll use to register your Mappings within your application. There are a number of methods available to help you to do so. However the most popular and easiest one to use is the Scan method.

5.2.1 Scan

The Scan method makes use of Reflection to query all referenced assemblies to all the Types that implement the IMapTo, IMapFrom and the ICustomMap interfaces and register them within your application domain.

To do so ise really easy, for instance if you are working on a Web Application (MVC or API), all you need to do is within you Configure method is

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        //Set up code for automapper configuration
        MapConfigurationFactory.Scan<Startup>();
    }
}
```

Your mappings will now be available throughout your application. So now making any mapping call such as;

```
public IEnumerable<Referrer> GetAllActive()
{
    var threats = _unitOfWork.GetRepository<Threat>()
        .Get(predicate: x => x.Status.Name == Enabled && x.ThreatType.Name == Referer␣
↪).AsEnumerable();
  return Mapper.Map<IEnumerable<Referrer>>(source: threats);

}
```

Will result in your Mapping being invoked

5.2.2 LoadMapsFromAssemblies

If speed and optimisation is a concern and you would rather explicitly pass in your assemblies containing your mapping logic you can do so, making use of the *LoadMapsFromAssemblies* method. It accepts a *params* array of assemblies, which you can supply an unlimited assemblies to it containing your mapping logic.

One way to do so would be to define a helper method to return an assembly by passing name to it, then retrieve the assmebly from those names

```
public class GetMappings
{
    public void Get()
    {
```

(continues on next page)

(continued from previous page)

```
        var domainObjects = GetAssemblyByName("DomainObjects");
        var entityObjects = GetAssemblyByName("EntityObjects");

        MapConfigurationFactory.LoadMapsFromAssemblies(domainObjects,  entityObjects);
    }

    private Assembly GetAssemblyByName(string name)
    {
        return AppDomain.CurrentDomain.GetAssemblies().SingleOrDefault(assembly =>␣
↪assembly.GetName().Name == name);
    }
}
```

5.2.3 LoadAllMappings

If you only want to load mappings from a particular assembly then you can make use of *LoadAllMappings* making use of a smilar stategy.

```
public class GetMappings
{
    public void Get()
    {
        var domainObjects = GetAssemblyByName("DomainObjects");
        MapConfigurationFactory.LoadAllMappings(domainObjects.GetTypes());
    }

    private Assembly GetAssemblyByName(string name)
    {
        return AppDomain.CurrentDomain.GetAssemblies().SingleOrDefault(assembly =>␣
↪assembly.GetName().Name == name);
    }
}
```