How to set up Automapper in ASP.NET Core

Asked 3 years, 1 month ago Active 2 months ago Viewed 131k times

219

100

I'm relatively new at .NET, and I decided to tackle .NET Core instead of learning the "old ways". I found a detailed article about setting up AutoMapper for .NET Core here, but is there a more simple walkthrough for a newbie?

c# asp.net-core automapper

edited Jun 23 at 14:28
Boann
41.1k 13 95 126

asked Oct 27 '16 at 2:34
theutz
7,559 4 13 18

4 See dotnetcoretutorials.com/2017/09/23/… – MiFreidgeim SO-stop being evil Dec 8 '17 at 6:15

For newer versions of core (>v1) check out @Saineshwar's answer stackoverflow.com/a/53455699/833878 – Robbie Feb 16 at 21:01

1 A complete answer with an example click this link – Iman Bahrampour May 25 at 6:08

13 Answers

496

I figured it out! Here's the details:

1. Add the main AutoMapper Package to your solution via NuGet.
2. Add the AutoMapper Dependency Injection Package to your solution via NuGet.
3. Create a new class for a mapping profile. (I made a class in the main solution directory called `MappingProfile.cs` and add the following code.) I'll use a `User` and `UserDto` object as an example.

```
public class MappingProfile : Profile {
    public MappingProfile() {
        // Add as many of these lines as you need to map your objects
        CreateMap<User, UserDto>();
```

```
            CreateMap<UserDto, User>();
        }
    }
```

4. Then add the AutoMapperConfiguration in the `Startup.cs` as shown below:

```
public void ConfigureServices(IServiceCollection services) {
    // .... Ignore code before this

   // Auto Mapper Configurations
    var mappingConfig = new MapperConfiguration(mc =>
    {
        mc.AddProfile(new MappingProfile());
    });

    IMapper mapper = mappingConfig.CreateMapper();
    services.AddSingleton(mapper);

    services.AddMvc();

}
```

5. To invoke the mapped object in code, do something like the following:

```
public class UserController : Controller {

    // Create a field to store the mapper object
    private readonly IMapper _mapper;

    // Assign the object in the constructor for dependency injection
    public UserController(IMapper mapper) {
        _mapper = mapper;
    }

    public async Task<IActionResult> Edit(string id) {

        // Instantiate source object
        // (Get it from the database or whatever your code calls for)
        var user = await _context.Users
            .SingleOrDefaultAsync(u => u.Id == id);

        // Instantiate the mapped data transfer object
        // using the mapper you stored in the private field.
        // The type of the source object is the first type argument
        // and the type of the destination is the second.
        // Pass the source object you just instantiated above
        // as the argument to the _mapper.Map<>() method.
```

```
            var model = _mapper.Map<UserDto>(user);

            // .... Do whatever you want after that!
        }
    }
```

I hope this helps someone starting fresh with ASP.NET Core! I welcome any feedback or criticisms as I'm still new to the .NET world!

edited Sep 24 '18 at 9:57

TanvirArjel

14.8k 7 30 63

answered Oct 27 '16 at 2:34

theutz

7,559 4 13 18

2 You should include the constructor for MappingProfile to contain the calls to CreateMap – Brian Seim at EvoDynamic LLC Nov 2 '16 at 1:37

3 The detailed article linked, lostechies.com/jimmybogard/2016/07/20/…, explains how `Profile` classes are located – Kieren Johnstone Dec 22 '16 at 8:25

18 @theutz You can merge those two CreateMap lines with a .ReverseMap() at the end of, well, either. Maybe comment it, but I find it more intuitive. – Astravagrant Mar 31 '17 at 12:40

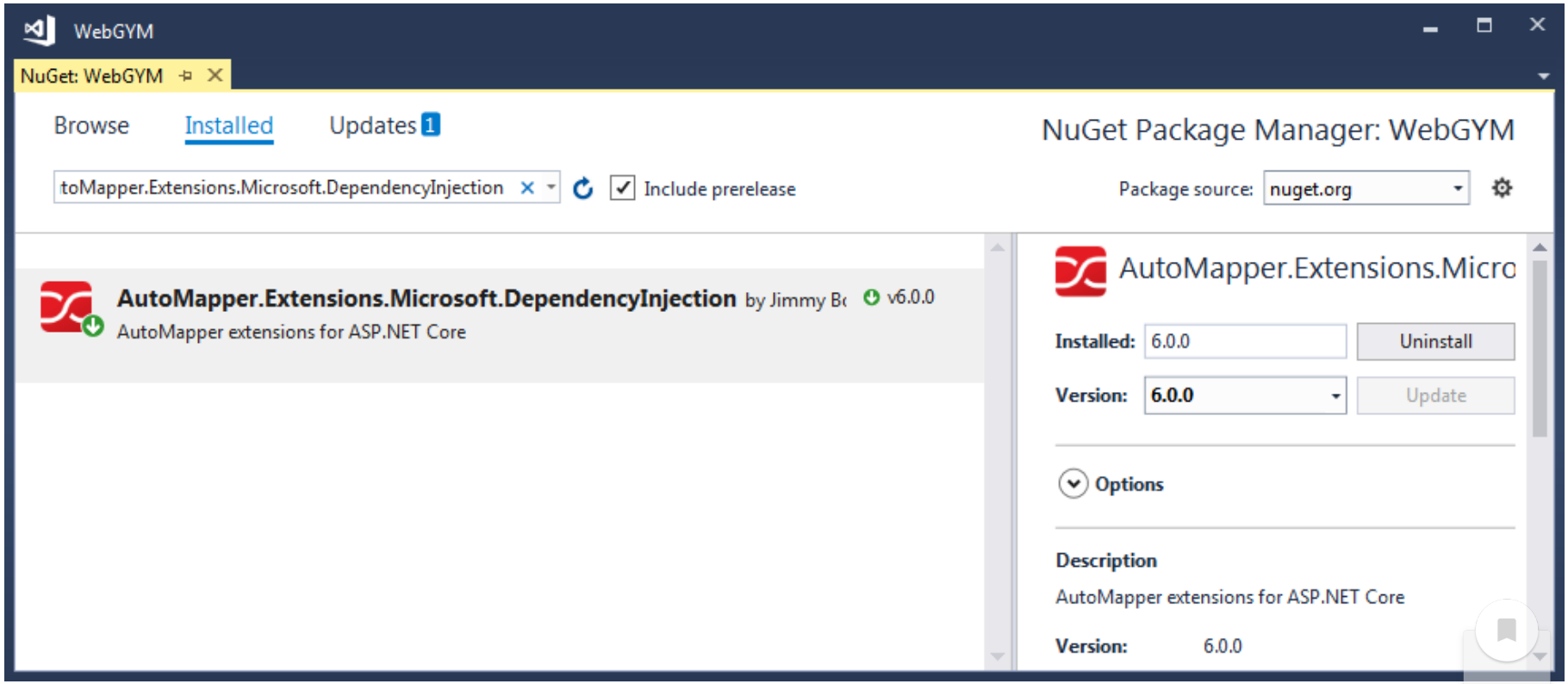
6 It might be helpful on Step 3 to mention adding a "using AutoMapper;" at the top so that the extension method is imported. – Rocklan May 30 '17 at 1:47

7 This worked fine with .net core 1.1, not anymore once I upgraded to .net core 2.0. I think, I need to explicitly specify the logic profile class assembly. Still researching how to accomplish that. Update: Ah the answer resides on your comment, I have to pass the typeof class which is my profile. // services.AddAutoMapper(typeof(Startup)); // <-- newer automapper version uses this signature – Esen Sep 23 '17 at 16:35

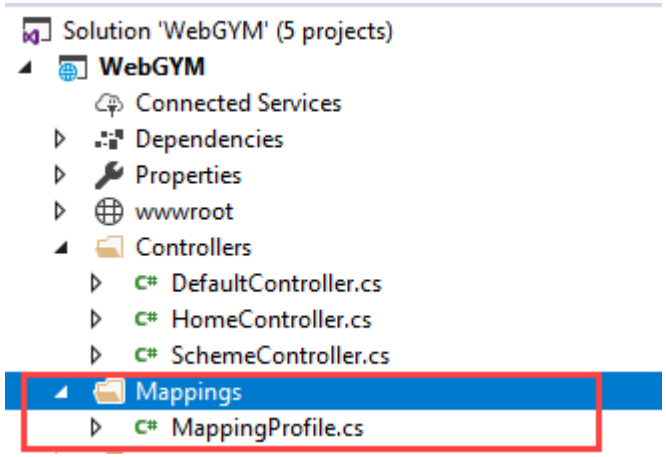
56

Step To Use AutoMapper with ASP.NET Core.

Step 1. Installing AutoMapper.Extensions.Microsoft.DependencyInjection from NuGet Package.

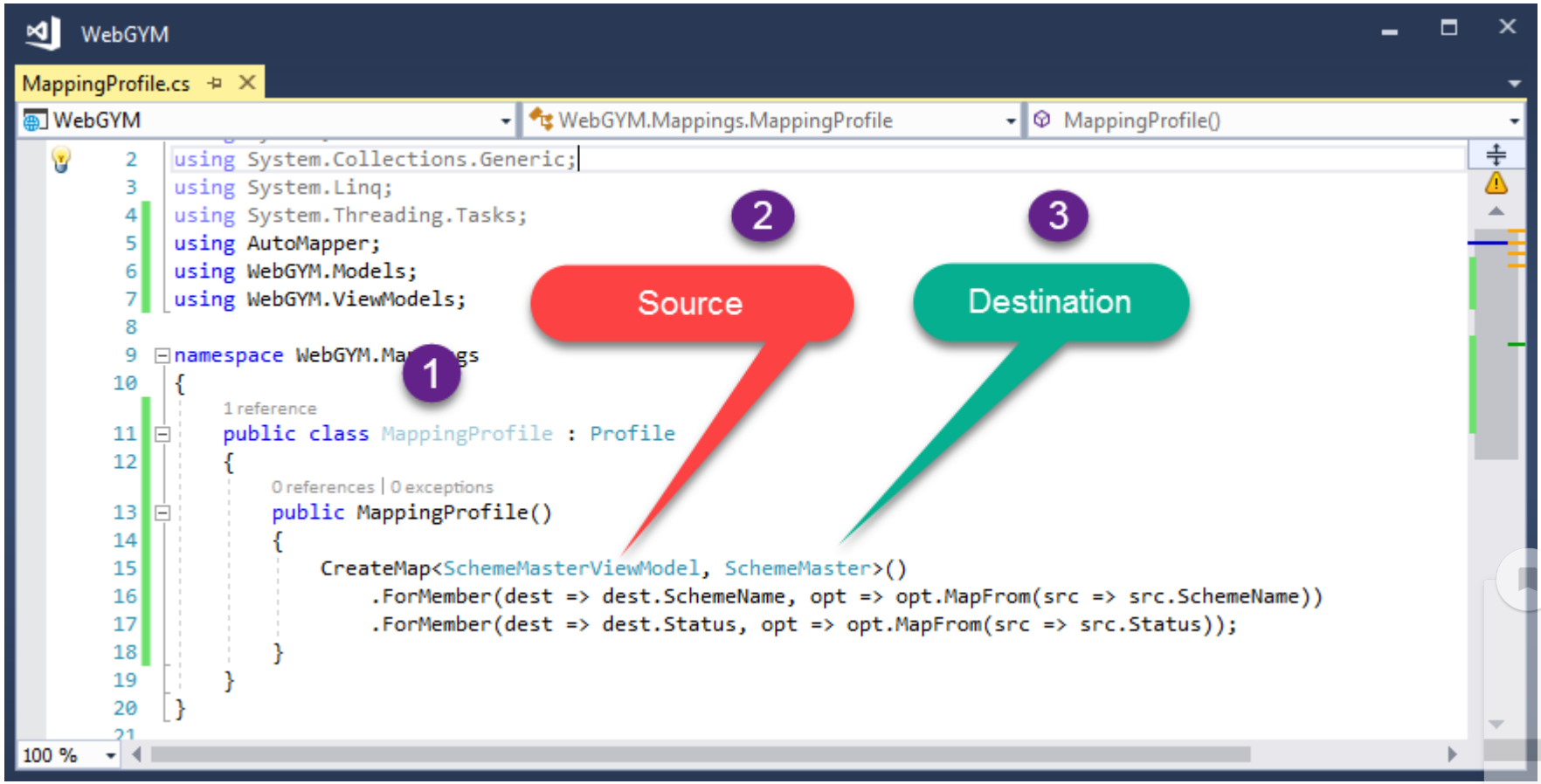


Step 2. Create a Folder in Solution to keep Mappings with Name "Mappings".



*Step 3. After adding Mapping folder we have added a class with Name "**MappingProfile**" this name can anything unique and good to understand.*

In this class, we are going to Maintain all Mappings.



Step 4. Initializing Mapper in Startup "ConfigureServices"

In Startup Class, we Need to Initialize Profile which we have created and also Register AutoMapper Service.

```
Mapper.Initialize(cfg => cfg.AddProfile<MappingProfile>());

services.AddAutoMapper();
```

Code Snippet to show ConfigureServices Method where we need to Initialize and Register AutoMapper.

```
public class Startup
{
    public Startup(IConfiguration configuration)
```

```
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }


    public void ConfigureServices(IServiceCollection services)
    {
        services.Configure<CookiePolicyOptions>(options =>
        {
            // This lambda determines whether user consent for non-essential cookies is
needed for a given request.
            options.CheckConsentNeeded = context => true;
            options.MinimumSameSitePolicy = SameSiteMode.None;
        });


        // Start Registering and Initializing AutoMapper

        Mapper.Initialize(cfg => cfg.AddProfile<MappingProfile>());
        services.AddAutoMapper();

        // End Registering and Initializing AutoMapper

        services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    }}
```

Step 5. Get Output.

To Get Mapped result we need to call AutoMapper.Mapper.Map and pass Proper Destination and Source.

```
AutoMapper.Mapper.Map<Destination>(source);
```

CodeSnippet

```
    [HttpPost]
    public void Post([FromBody] SchemeMasterViewModel schemeMaster)
    {
        if (ModelState.IsValid)
        {
            var mappedresult = AutoMapper.Mapper.Map<SchemeMaster>(schemeMaster);
        }
    }
```

edited Nov 24 '18 at 7:06

answered Nov 24 '18 at 6:18

Saineshwar

2,475 3 32 38

4 I get the following error: `'Mapper' does not contain a definition for 'initialize'` . I'm using `AutoMapper.Extensions.Microsoft.DependencyInjection` version 7.0.0 – kimbaudi Oct 14 at 2:06

Super detailed answer. Thank you sir. – Rod Hartzell Oct 29 at 22:26

37

I want to extend @theutz's answers - namely this line :

```
// services.AddAutoMapper(typeof(Startup));  // <-- newer automapper version uses this
signature.
```

There is a bug (*probably*) in AutoMapper.Extensions.Microsoft.DependencyInjection version 3.2.0. (I'm using .NET Core 2.0)

This is tackled in this GitHub issue. If your classes inheriting AutoMapper's Profile class exist outside of assembly where you Startup class is they will probably not be registered if your AutoMapper injection looks like this:

```
services.AddAutoMapper();
```

unless you explicitly specify which assemblies to search AutoMapper profiles for.

It can be done like this in your Startup.ConfigureServices:

```
services.AddAutoMapper(<assembies> or <type_in_assemblies>);
```

where *"assemblies"* and *"type_in_assemblies"* point to the assembly where Profile classes in your application are specified. E.g:

```
services.AddAutoMapper(typeof(ProfileInOtherAssembly),
typeof(ProfileInYetAnotherAssembly));
```

I **suppose** (and I put emphasis on this word) that due to following implementation of parameterless overload (source code from GitHub) :

```
public static IServiceCollection AddAutoMapper(this IServiceCollection services)
{
     return services.AddAutoMapper(null, AppDomain.CurrentDomain.GetAssemblies());
}
```

we rely on CLR having already JITed assembly containing AutoMapper profiles which might be or might not be true as they are only jitted when needed (more details in this StackOverflow question).

edited Jun 26 at 13:26
RJFalconer
7,047 3 37 57

answered May 4 '18 at 15:30
GrayCat
866 8 21

4 Thats the correct answer for latest version of AutoMapper and AspNetCore – Joshit May 7 at 9:31

1 this was the answer I was looking for for AutoMapper 8.1 (latest version) – Tinaira Jun 16 at 16:08

29

theutz' answer here is very good, I just want to add this:

If you let your mapping profile inherit from `MapperConfigurationExpression` instead of `Profile`, you can very simply add a test to verify your mapping setup, which is always handy:

```
[Fact]
public void MappingProfile_VerifyMappings()
{
    var mappingProfile = new MappingProfile();

    var config = new MapperConfiguration(mappingProfile);
    var mapper = new Mapper(config);

    (mapper as IMapper).ConfigurationProvider.AssertConfigurationIsValid();
}
```

answered Mar 30 '17 at 12:39
Arve Systad
5,150 1 29 57

I am getting one error : "AutoMapper Extension Dependency injection is incompatible with asp.net core 1.1 ". Please help! – Rohit Arora Aug 31 '17 at 4:38

It seems the definition of "verify" is up for debate. This blows up when certain properties are ommitted by design to prevent mapping. – Jeremy Holovacs Mar 21 '18 at 19:36

2 If you don't want a property mapped, set it up with .Ignore(). That way, it forces you to actively think about handling each case - making sure you don't miss out on stuff when changes are being made. Super practical, actually. So yes, the verify-test is a larger safety net than many people realise. It's not foolproof, but it takes care of the first 90%. – Arve Systad Mar 23 '18 at 5:13

14

I solved it this way (similar to above but I feel like it's a cleaner solution) for .NET Core 2.2/Automapper 8.1.1 w/ Extensions.DI 6.1.1.

Create MappingProfile.cs class and populate constructor with Maps (I plan on using a single class to hold all my mappings)

```
public class MappingProfile : Profile
{
    public MappingProfile()
    {
        CreateMap<Source, Dest>().ReverseMap();
    }
}
```

In Startup.cs, add below to add to DI (the assembly arg is for the class that holds your mapping configs, in my case, it's the MappingProfile class).

```
//add automapper DI
services.AddAutoMapper(typeof(MappingProfile));
```

In Controller, use it like you would any other DI object

```
[Route("api/[controller]")]
[ApiController]
public class AnyController : ControllerBase
{
    private readonly IMapper _mapper;

    public AnyController(IMapper mapper)
    {
        _mapper = mapper;
    }

    public IActionResult Get(int id)
    {
```

```
            var entity = repository.Get(id);
            var dto = _mapper.Map<Dest>(entity);

            return Ok(dto);
        }
    }
```

answered Aug 1 at 20:20

Coy Meeks
215 2 6

1 I like your answer. I think wrapping `MappingProfiles` with `new Type[]{}` as shown in this answer is unnecessary. – Money Oriented Programmer Aug 29 at 12:18

10

In my Startup.cs (Core 2.2, Automapper 8.1.1)

```
services.AddAutoMapper(new Type[] { typeof(DAL.MapperProfile) });
```

In my data access project

```
namespace DAL
{
    public class MapperProfile : Profile
    {
        // place holder for AddAutoMapper (to bring in the DAL assembly)
    }
}
```

In my model definition

```
namespace DAL.Models
{
    public class PositionProfile : Profile
    {
        public PositionProfile()
        {
            CreateMap<Position, PositionDto_v1>();
        }
```

```
    }

    public class Position
    {
        ...
    }
```

answered Jun 9 at 22:25

Brian Rice
2,065 1 21 38

Why don't you just use `services.AddAutoMapper( typeof(DAL.MapperProfile) );` instead of `services.AddAutoMapper(new Type[] { typeof(DAL.MapperProfile) });` ? – Money Oriented Programmer Aug 29 at 12:13

6

I am using AutoMapper 6.1.1 and asp.net Core 1.1.2.

First of all, define Profile classes inherited by Profile Class of Automapper. I Created IProfile interface which is empty, the purpose is only to find the classes of this type.

```
public class UserProfile : Profile, IProfile
    {
        public UserProfile()
        {
            CreateMap<User, UserModel>();
            CreateMap<UserModel, User>();
        }
    }
```

Now create a separate class e.g Mappings

```
public class Mappings
    {
     public static void RegisterMappings()
     {
       var all =
       Assembly
          .GetEntryAssembly()
          .GetReferencedAssemblies()
          .Select(Assembly.Load)
          .SelectMany(x => x.DefinedTypes)
```

```
            .Where(type =>
typeof(IProfile).GetTypeInfo().IsAssignableFrom(type.AsType()));

            foreach (var ti in all)
            {
                var t = ti.AsType();
                if (t.Equals(typeof(IProfile)))
                {
                    Mapper.Initialize(cfg =>
                    {
                        cfg.AddProfiles(t); // Initialise each Profile classe
                    });
                }
            }
        }

    }
```

Now in MVC Core web Project in Startup.cs file, in the constructor, call Mapping class which will initialize all mappings at the time of application loading.

```
Mappings.RegisterMappings();
```

edited Oct 21 '18 at 13:02
Roy Scheffers
2,478 10 21 29

answered Jul 25 '17 at 1:19
Aamir
477 7 10

You can just create a subclass from profile class, and when program is running services.AddAutoMapper(); line of codes The automapper automatically knows them. – isaeid Oct 1 '17 at 13:49

I don't think this is necessary if you use AutoMapper.Extensions.Microsoft.DependancyInjection which is available in nuget. – Greg Gum Aug 18 '18 at 10:43

5

For ASP.NET Core (tested using 2.0+ and 3.0), if you prefer to read the source documentation:
https://github.com/AutoMapper/AutoMapper.Extensions.Microsoft.DependencyInjection/blob/master/README.md

Otherwise following these 4 steps works:

1. Install AutoMapper.Extensions.Microsoft.DependancyInjection from nuget.

2. Simply add some profile classes.
3. Then add below to your startup.cs class. `services.AddAutoMapper(OneOfYourProfileClassNamesHere)`
4. Then simply Inject IMapper in your controllers or wherever you need it:

```
public class EmployeesController {

    private readonly IMapper _mapper;

    public EmployeesController(IMapper mapper){

        _mapper = mapper;
    }
```

And if you want to use ProjectTo its now simply:

```
var customers = await dbContext.Customers.ProjectTo<CustomerDto>
(_mapper.ConfigurationProvider).ToListAsync()
```

edited Sep 29 at 20:35

answered Feb 19 at 6:26

dalcam

697 8 19

3

I like a lot of answers, particularly @saineshwar 's one. I'm using .net Core 3.0 with AutoMapper 9.0, so I feel it's time to update its answer.

What worked for me was in Startup.ConfigureServices(...) register the service in this way:

```
    services.AddAutoMapper(cfg => cfg.AddProfile<MappingProfile>(),
                           AppDomain.CurrentDomain.GetAssemblies());
```

I think that rest of @saineshwar answer keeps perfect. But if anyone is interested my controller code is:

```
[HttpGet("{id}")]
public async Task<ActionResult> GetIic(int id)
{
    // _context is a DB provider
    var Iic = await _context.Find(id).ConfigureAwait(false);
```

```
    if (Iic == null)
    {
        return NotFound();
    }

    var map = _mapper.Map<IicVM>(Iic);

    return Ok(map);
}
```

And my mapping class:

```
public class MappingProfile : Profile
{
    public MappingProfile()
    {
        CreateMap<Iic, IicVM>()
            .ForMember(dest => dest.DepartmentName, o => o.MapFrom(src =>
src.Department.Name))
            .ForMember(dest => dest.PortfolioTypeName, o => o.MapFrom(src =>
src.PortfolioType.Name));
            //.ReverseMap();
    }
}
```

----- EDIT -----

After reading the docs linked in the comments by Lucian Bargaoanu, I think it's better to change this answer a bit.

The parameterless `services.AddAutoMapper()` (that had the @saineshwar answer) doesn't work anymore (at least for me). But if you use the NuGet assembly AutoMapper.Extensions.Microsoft.DependencyInjection, the framework is able to inspect all the classes that extend AutoMapper.Profile (like mine, MappingProfile).

So, in my case, where the class belong to the same executing assembly, the service registration can be shortened to
`services.AddAutoMapper(System.Reflection.Assembly.GetExecutingAssembly());`
(A more elegant approach could be a parameterless extension with this coding).

Thanks, Lucian!

edited Oct 17 at 11:00

answered Oct 17 at 9:08

Vic

111 1 6

docs.automapper.org/en/latest/… – Lucian Bargaoanu Oct 17 at 9:17

1

services.AddAutoMapper(); didn't work for me. (I am using Asp.Net Core 2.0)

After configuring as below

```
var config = new AutoMapper.MapperConfiguration(cfg =>
{
    cfg.CreateMap<ClientCustomer, Models.Customer>();
});
```

initialize the mapper IMapper mapper = config.CreateMapper();

and add the mapper object to services as a singleton services.AddSingleton(mapper);

this way I am able to add a DI to controller

```
private IMapper autoMapper = null;

public VerifyController(IMapper mapper)
{
 autoMapper = mapper;
}
```

and I have used as below in my action methods

```
ClientCustomer customerObj = autoMapper.Map<ClientCustomer>(customer);
```

answered Mar 24 '18 at 15:27

Venkat pv

43 4

Hi @venkat you probably just needed to add the AutoMapper.Extensions.Microsoft.DependancyInjection package to your project – dalcam Sep 25 at 18:25

0

about theutz answer , there is no need to specify the **IMapper mapper** parrameter at the controllers constructor.

you can use the Mapper as it is a static member at any place of the code.

```
public class UserController : Controller {
   public someMethod()
   {
      Mapper.Map<User, UserDto>(user);
   }
}
```

answered May 10 '17 at 5:38

yaronmil
43 1 4

10 But statics are a bit anti-testable, no? – Scott Fraley May 17 '17 at 17:40

3 Yep. This will work in many cases, but if you have no configured mapping when invoking this method in a test, It'll throw an exception (and thus failing the test for the wrong reason). With an injected `IMapper` you can mock that and, for example, just make it return null if it's irrelevant for the given test. – Arve Systad May 29 '17 at 20:48

0

Asp.Net Core 2.2 with AutoMapper.Extensions.Microsoft.DependencyInjection.

```
public class MappingProfile : Profile
{
  public MappingProfile()
  {
      CreateMap<Domain, DomainDto>();
  }
}
```

In Startup.cs

```
services.AddAutoMapper(typeof(List.Handler));
```

edited Oct 12 at 7:42

answered Oct 12 at 7:19

user3036876


177 1 2 11

0

To add onto what Arve Systad mentioned for testing. If for whatever reason you're like me and want to maintain the inheritance structure provided in theutz solution, you can set up the MapperConfiguration like so:

```
var mappingProfile = new MappingProfile();
var config = new MapperConfiguration(cfg =>
{
    cfg.AddProfile(mappingProfile);
});
var mapper = new Mapper(config);
```

I did this in NUnit.

answered Mar 9 '18 at 20:18


LandSharks
169 3 10