AutoMapper Documentation

Jimmy Bogard

Oct 16, 2020

Overview

A convention-based object-object mapper.

AutoMapper uses a fluent configuration API to define an object-object mapping strategy. AutoMapper uses a convention-based matching algorithm to match up source to destination values. AutoMapper is geared towards model projection scenarios to flatten complex object models to DTOs and other simple objects, whose design is better suited for serialization, communication, messaging, or simply an anti-corruption layer between the domain and application layer.

AutoMapper supports the following platforms:

- .NET 4.6.1+
- .NET Standard 2.0+

New to AutoMapper? Check out the *Getting Started Guide* page first.

CHAPTER 1

Getting Started Guide

1.1 What is AutoMapper?

AutoMapper is an object-object mapper. 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.

1.2 Why use AutoMapper?

Mapping code is boring. Testing mapping code is even more boring. AutoMapper provides simple configuration of types, as well as simple testing of mappings. The real question may be "why use object-object mapping?" Mapping can occur in many places in an application, but mostly in the boundaries between layers, such as between the UI/Domain layers, or Service/Domain layers. Concerns of one layer often conflict with concerns in another, so object-object mapping leads to segregated models, where concerns for each layer can affect only types in that layer.

1.3 How do I use AutoMapper?

First, you need both a source and destination type to work with. The destination type's design can be influenced by the layer in which it lives, but AutoMapper works best as long as the names of the members match up to the source type's members. If you have a source member called "FirstName", this will automatically be mapped to a destination member with the name "FirstName". AutoMapper also supports Flattening.

AutoMapper will ignore null reference exceptions when mapping your source to your target. This is by design. If you don't like this approach, you can combine AutoMapper's approach with custom value resolvers if needed.

Once you have your types you can create a map for the two types using a `MapperConfiguration` and CreateMap. You only need one `MapperConfiguration` instance typically per AppDomain and should be instantiated during startup. More examples of initial setup see in Setup.

```
var config = new MapperConfiguration(cfg => cfg.CreateMap<Order, OrderDto>());
```

The type on the left is the source type, and the type on the right is the destination type. To perform a mapping, call one of the `Map` overloads:

```
var mapper = config.CreateMapper();
// or
var mapper = new Mapper(config);
OrderDto dto = mapper.Map<OrderDto>(order);
```

Most applications can use dependency injection to inject the created `IMapper` instance.

AutoMapper also has non-generic versions of these methods, for those cases where you might not know the type at compile time.

1.4 Where do I configure AutoMapper?

Configuration should only happen once per AppDomain. That means the best place to put the configuration code is in application startup, such as the Global.asax file for ASP.NET applications. Typically, the configuration bootstrapper class is in its own class, and this bootstrapper class is called from the startup method. The bootstrapper class should construct a `MapperConfiguration` object to configure the type maps.

For ASP.NET Core the Dependency Injection article shows how to configure AutoMapper in your application.

1.5 How do I test my mappings?

To test your mappings, you need to create a test that does two things:

- Call your bootstrapper class to create all the mappings
- Call MapperConfiguration.AssertConfigurationIsValid

Here's an example:

```
var config = AutoMapperConfiguration.Configure();

config.AssertConfigurationIsValid();
```

CHAPTER 2

Understanding Your Mappings

AutoMapper creates an execution plan for your mapping. That execution plan can be viewed as an expression tree during debugging. You can get a better view of the resulting code by installing the ReadableExpressions VS extension. If you need to see the code outside VS, you can use the ReadableExpressions package directly. This DotNetFiddle has a live demo using the NuGet package, and this article describes using the VS extension.

```
var configuration = new MapperConfiguration(cfg => cfg.CreateMap<Foo, Bar>());
var executionPlan = configuration.BuildExecutionPlan(typeof(Foo), typeof(Bar));
```

Be sure to remove all such code before release.

For ProjectTo, you need to inspect `IQueryable.Expression`.

```
var expression = context.Entities.ProjectTo<Dto>().Expression;
```

CHAPTER 3

The MyGet Build

AutoMapper uses MyGet to publish development builds based on the master branch. This means that the MyGet build sometimes contains fixes that are not available in the current NuGet package. Please try the latest MyGet build before reporting issues, in case your issue has already been fixed but not released.

The AutoMapper MyGet gallery is available here. Be sure to include prereleases.

3.1 Installing the Package

If you want to install the latest MyGet package into a project, you can use the following command:

```
Install-Package AutoMapper -Source https://www.myget.org/F/automapperdev/api/v3/index.
↪json -IncludePrerelease
```

CHAPTER 4

Configuration

Create a `MapperConfiguration` instance and initialize configuration via the constructor:

```
var config = new MapperConfiguration(cfg => {
    cfg.CreateMap<Foo, Bar>();
    cfg.AddProfile<FooProfile>();
});
```

The `MapperConfiguration` instance can be stored statically, in a static field or in a dependency injection container. Once created it cannot be changed/modified.

```
var configuration = new MapperConfiguration(cfg => {
    cfg.CreateMap<Foo, Bar>();
    cfg.AddProfile<FooProfile>();
});
```

Starting with 9.0, the static API is no longer available.

4.1 Profile Instances

A good way to organize your mapping configurations is with profiles. Create classes that inherit from `Profile` and put the configuration in the constructor:

```
// This is the approach starting with version 5
public class OrganizationProfile : Profile
{
        public OrganizationProfile()
        {
                CreateMap<Foo, FooDto>();
                // Use CreateMap... Etc.. here (Profile methods are the same as␣
↪configuration methods)
        }
}
```

(continues on next page)

(continued from previous page)

```
// How it was done in 4.x - as of 5.0 this is obsolete:
// public class OrganizationProfile : Profile
// {
//     protected override void Configure()
//     {
//         CreateMap<Foo, FooDto>();
//     }
// }
```

In earlier versions the `Configure` method was used instead of a constructor. As of version 5, `Configure()` is obsolete. It will be removed in 6.0.

Configuration inside a profile only applies to maps inside the profile. Configuration applied to the root configuration applies to *all* maps created.

4.1.1 Assembly Scanning for auto configuration

Profiles can be added to the main mapper configuration in a number of ways, either directly:

```
cfg.AddProfile<OrganizationProfile>();
cfg.AddProfile(new OrganizationProfile());
```

or by automatically scanning for profiles:

```
// Scan for all profiles in an assembly
// ... using instance approach:
var config = new MapperConfiguration(cfg => {
    cfg.AddMaps(myAssembly);
});
var configuration = new MapperConfiguration(cfg => cfg.AddMaps(myAssembly));

// Can also use assembly names:
var configuration = new MapperConfiguration(cfg =>
    cfg.AddMaps(new [] {
        "Foo.UI",
        "Foo.Core"
    });
);

// Or marker types for assemblies:
var configuration = new MapperConfiguration(cfg =>
    cfg.AddMaps(new [] {
        typeof(HomeController),
        typeof(Entity)
    });
);
```

AutoMapper will scan the designated assemblies for classes inheriting from Profile and add them to the configuration.

4.2 Naming Conventions

You can set the source and destination naming conventions

```
var configuration = new MapperConfiguration(cfg => {
  cfg.SourceMemberNamingConvention = new LowerUnderscoreNamingConvention();
  cfg.DestinationMemberNamingConvention = new PascalCaseNamingConvention();
});
```

This will map the following properties to each other: `property_name -> PropertyName`

You can also set this at a per profile level

```
public class OrganizationProfile : Profile
{
  public OrganizationProfile()
  {
    SourceMemberNamingConvention = new LowerUnderscoreNamingConvention();
    DestinationMemberNamingConvention = new PascalCaseNamingConvention();
    //Put your CreateMap... Etc.. here
  }
}
```

If you don't need a naming convention, you can use the `ExactMatchNamingConvention`.

4.3 Replacing characters

You can also replace individual characters or entire words in source members during member name matching:

```
public class Source
{
    public int Value { get; set; }
    public int Ävíator { get; set; }
    public int SubAirlinaFlight { get; set; }
}
public class Destination
{
    public int Value { get; set; }
    public int Aviator { get; set; }
    public int SubAirlineFlight { get; set; }
}
```

We want to replace the individual characters, and perhaps translate a word:

```
var configuration = new MapperConfiguration(c =>
{
    c.ReplaceMemberName("Ä", "A");
    c.ReplaceMemberName("í", "i");
    c.ReplaceMemberName("Airlina", "Airline");
});
```

4.4 Recognizing pre/postfixes

Sometimes your source/destination properties will have common pre/postfixes that cause you to have to do a bunch of custom member mappings because the names don't match up. To address this, you can recognize pre/postfixes:

```
public class Source {
    public int frmValue { get; set; }
    public int frmValue2 { get; set; }
}
public class Dest {
    public int Value { get; set; }
    public int Value2 { get; set; }
}
var configuration = new MapperConfiguration(cfg => {
    cfg.RecognizePrefixes("frm");
    cfg.CreateMap<Source, Dest>();
});
configuration.AssertConfigurationIsValid();
```

By default AutoMapper recognizes the prefix “Get”, if you need to clear the prefix:

```
var configuration = new MapperConfiguration(cfg => {
    cfg.ClearPrefixes();
    cfg.RecognizePrefixes("tmp");
});
```

4.5 Global property/field filtering

By default, AutoMapper tries to map every public property/field. You can filter out properties/fields with the property/field filters:

```
var configuration = new MapperConfiguration(cfg =>
{
        // don't map any fields
        cfg.ShouldMapField = fi => false;

        // map properties with a public or private getter
        cfg.ShouldMapProperty = pi =>
                pi.GetMethod != null && (pi.GetMethod.IsPublic || pi.GetMethod.
↪IsPrivate);
});
```

4.6 Configuring visibility

By default, AutoMapper only recognizes public members. It can map to private setters, but will skip internal/private methods and properties if the entire property is private/internal. To instruct AutoMapper to recognize members with other visibilities, override the default filters ShouldMapField and/or ShouldMapProperty :

```
var configuration = new MapperConfiguration(cfg =>
{
    // map properties with public or internal getters
    cfg.ShouldMapProperty = p => p.GetMethod.IsPublic || p.GetMethod.IsAssembly;
    cfg.CreateMap<Source, Destination>();
});
```

Map configurations will now recognize internal/private members.

4.7 Configuration compilation

Because expression compilation can be a bit resource intensive, AutoMapper lazily compiles the type map plans on first map. However, this behavior is not always desirable, so you can tell AutoMapper to compile its mappings directly:

```
var configuration = new MapperConfiguration(cfg => {});
configuration.CompileMappings();
```

For a few hundred mappings, this may take a couple of seconds.

CHAPTER 5

Configuration Validation

Hand-rolled mapping code, though tedious, has the advantage of being testable. One of the inspirations behind AutoMapper was to eliminate not just the custom mapping code, but eliminate the need for manual testing. Because the mapping from source to destination is convention-based, you will still need to test your configuration.

AutoMapper provides configuration testing in the form of the AssertConfigurationIsValid method. Suppose we have slightly misconfigured our source and destination types:

```
public class Source
{
        public int SomeValue { get; set; }
}

public class Destination
{
        public int SomeValuefff { get; set; }
}
```

In the Destination type, we probably fat-fingered the destination property. Other typical issues are source member renames. To test our configuration, we simply create a unit test that sets up the configuration and executes the AssertConfigurationIsValid method:

```
var configuration = new MapperConfiguration(cfg =>
  cfg.CreateMap<Source, Destination>());

configuration.AssertConfigurationIsValid();
```

Executing this code produces an AutoMapperConfigurationException, with a descriptive message. AutoMapper checks to make sure that *every single* Destination type member has a corresponding type member on the source type.

5.1 Overriding configuration errors

To fix a configuration error (besides renaming the source/destination members), you have three choices for providing an alternate configuration:

- Custom Value Resolvers
- Projection
- Use the Ignore() option

With the third option, we have a member on the destination type that we will fill with alternative means, and not through the Map operation.

```
var configuration = new MapperConfiguration(cfg =>
  cfg.CreateMap<Source, Destination>()
        .ForMember(dest => dest.SomeValuefff, opt => opt.Ignore())
);
```

5.2 Selecting members to validate

By default, AutoMapper uses the destination type to validate members. It assumes that all destination members need to be mapped. To modify this behavior, use the `CreateMap` overload to specify which member list to validate against:

```
var configuration = new MapperConfiguration(cfg =>
  cfg.CreateMap<Source, Destination>(MemberList.Source);
  cfg.CreateMap<Source2, Destination2>(MemberList.None);
);
```

To skip validation altogether for this map, use `MemberList.None`.

5.3 Custom validations

You can add custom validations through an extension point. See here.

CHAPTER 6

Dependency Injection

AutoMapper supports the ability to construct Custom Value Resolvers, Custom Type Converters, and Value Converters using static service location:

```
var configuration = new MapperConfiguration(cfg =>
{
    cfg.ConstructServicesUsing(ObjectFactory.GetInstance);

    cfg.CreateMap<Source, Destination>();
});
```

Or dynamic service location, to be used in the case of instance-based containers (including child/nested containers):

```
var mapper = new Mapper(configuration, childContainer.GetInstance);

var dest = mapper.Map<Source, Destination>(new Source { Value = 15 });
```

6.1 Queryable Extensions

Starting with 8.0 you can use IMapper.ProjectTo. For older versions you need to pass the configuration to the extension method `IQueryable.ProjectTo<T>(IConfigurationProvider)`.

Note that IQueryable.ProjectTo is more limited than IMapper.Map, as only what is allowed by the underlying LINQ provider is supported. That means you cannot use DI with value resolvers and converters as you can with Map.

6.2 Examples

6.2.1 ASP.NET Core

There is a NuGet package to be used with the default injection mechanism described here and used in this project.

You define the configuration using profiles. And then you let AutoMapper know in what assemblies are those profiles defined by calling the `IServiceCollection` extension method `AddAutoMapper` at startup:

```
services.AddAutoMapper(profileAssembly1, profileAssembly2 /*, ...*/);
```

or marker types:

```
services.AddAutoMapper(typeof(ProfileTypeFromAssembly1),␣
↪typeof(ProfileTypeFromAssembly2) /*, ...*/);
```

Now you can inject AutoMapper at runtime into your services/controllers:

```
public class EmployeesController {
        private readonly IMapper _mapper;

        public EmployeesController(IMapper mapper) => _mapper = mapper;

        // use _mapper.Map or _mapper.ProjectTo
}
```

6.2.2 AutoFac

There is a third-party NuGet package you might want to try.

Also, check this blog.

6.2.3 Ninject

For those using Ninject here is an example of a Ninject module for AutoMapper

```
public class AutoMapperModule : NinjectModule
{
    public override void Load()
    {
        Bind<IValueResolver<SourceEntity, DestModel, bool>>().To<MyResolver>();

        var mapperConfiguration = CreateConfiguration();
        Bind<MapperConfiguration>().ToConstant(mapperConfiguration).
↪InSingletonScope();

        // This teaches Ninject how to create automapper instances say if for instance
        // MyResolver has a constructor with a parameter that needs to be injected
        Bind<IMapper>().ToMethod(ctx =>
             new Mapper(mapperConfiguration, type => ctx.Kernel.Get(type)));
    }

    private MapperConfiguration CreateConfiguration()
    {
        var config = new MapperConfiguration(cfg =>
        {
            // Add all profiles in current assembly
            cfg.AddMaps(GetType().Assembly);
        });

        return config;
```

(continues on next page)

(continued from previous page)

```
    }
}
```

6.2.4 Simple Injector

The workflow is as follows:

1. Register your types via MyRegistrar.Register
2. The MapperProvider allows you to directly inject an instance of IMapper into your other classes
3. SomeProfile resolves a value using PropertyThatDependsOnIocValueResolver
4. PropertyThatDependsOnIocValueResolver has IService injected into it, which is then able to be used

The ValueResolver has access to IService because we register our container via MapperConfigurationExpression.ConstructServicesUsing

```
public class MyRegistrar
{
    public void Register(Container container)
    {
        // Injectable service
        container.RegisterSingleton<IService, SomeService>();

        // Automapper
        container.RegisterSingleton(() => GetMapper(container));
    }

    private AutoMapper.IMapper GetMapper(Container container)
    {
        var mp = container.GetInstance<MapperProvider>();
        return mp.GetMapper();
    }
}

public class MapperProvider
{
    private readonly Container _container;

    public MapperProvider(Container container)
    {
        _container = container;
    }

    public IMapper GetMapper()
    {
        var mce = new MapperConfigurationExpression();
        mce.ConstructServicesUsing(_container.GetInstance);

        mce.AddMaps(typeof(SomeProfile).Assembly);

        var mc = new MapperConfiguration(mce);
        mc.AssertConfigurationIsValid();

        IMapper m = new Mapper(mc, t => _container.GetInstance(t));
```

(continues on next page)

(continued from previous page)

```
        return m;
    }
}

public class SomeProfile : Profile
{
    public SomeProfile()
    {
        var map = CreateMap<MySourceType, MyDestinationType>();
        map.ForMember(d => d.PropertyThatDependsOnIoc, opt => opt.MapFrom
↪<PropertyThatDependsOnIocValueResolver>());
    }
}

public class PropertyThatDependsOnIocValueResolver : IValueResolver<MySourceType,␣
↪object, int>
{
    private readonly IService _service;

    public PropertyThatDependsOnIocValueResolver(IService service)
    {
        _service = service;
    }

    int IValueResolver<MySourceType, object, int>.Resolve(MySourceType source, object␣
↪destination, int destMember, ResolutionContext context)
    {
        return _service.MyMethod(source);
    }
}
```

6.2.5 Castle Windsor

For those using Castle Windsor here is an example of an installer for AutoMapper

```
public class AutoMapperInstaller : IWindsorInstaller
{
    public void Install(IWindsorContainer container, IConfigurationStore store)
        {
            // Register all mapper profiles
            container.Register(
                Classes.FromAssemblyInThisApplication(GetType().Assembly)
                .BasedOn<Profile>().WithServiceBase());

            // Register IConfigurationProvider with all registered profiles
            container.Register(Component.For<IConfigurationProvider>().
↪UsingFactoryMethod(kernel =>
            {
                return new MapperConfiguration(configuration =>
                {
                    kernel.ResolveAll<Profile>().ToList().ForEach(configuration.
↪AddProfile);
                });
            }).LifestyleSingleton());
```

(continues on next page)

(continued from previous page)

```
            // Register IMapper with registered IConfigurationProvider
            container.Register(
                Component.For<IMapper>().UsingFactoryMethod(kernel =>
                    new Mapper(kernel.Resolve<IConfigurationProvider>(), kernel.
↪Resolve)));
        }
}
```

6.2.6 Catel.IoC

For those using Catel.IoC here is how you register AutoMapper. First define the configuration using profiles. And then you let AutoMapper know in what assemblies those profiles are defined by registering AutoMapper in the ServiceLocator at startup:

```
ServiceLocator.Default.RegisterInstance(typeof(IMapper), new␣
↪Mapper(CreateConfiguration()));
```

Configuration Creation Method:

```
public static MapperConfiguration CreateConfiguration()
{
    var config = new MapperConfiguration(cfg =>
    {
        // Add all profiles in current assembly
        cfg.AddMaps(GetType().Assembly);
    });

    return config;
}
```

Now you can inject AutoMapper at runtime into your services/controllers:

```
public class EmployeesController {
        private readonly IMapper _mapper;

        public EmployeesController(IMapper mapper) => _mapper = mapper;

        // use _mapper.Map or _mapper.ProjectTo
}
```

CHAPTER 7

Projection

Projection transforms a source to a destination beyond flattening the object model. Without extra configuration, AutoMapper requires a flattened destination to match the source type's naming structure. When you want to project source values into a destination that does not exactly match the source structure, you must specify custom member mapping definitions. For example, we might want to turn this source structure:

```
public class CalendarEvent
{
        public DateTime Date { get; set; }
        public string Title { get; set; }
}
```

Into something that works better for an input form on a web page:

```
public class CalendarEventForm
{
        public DateTime EventDate { get; set; }
        public int EventHour { get; set; }
        public int EventMinute { get; set; }
        public string Title { get; set; }
}
```

Because the names of the destination properties do not exactly match the source property (`CalendarEvent.Date` would need to be `CalendarEventForm.EventDate`), we need to specify custom member mappings in our type map configuration:

```
// Model
var calendarEvent = new CalendarEvent
{
        Date = new DateTime(2008, 12, 15, 20, 30, 0),
        Title = "Company Holiday Party"
};

// Configure AutoMapper
var configuration = new MapperConfiguration(cfg =>
```

(continues on next page)

(continued from previous page)

```
  cfg.CreateMap<CalendarEvent, CalendarEventForm>()
        .ForMember(dest => dest.EventDate, opt => opt.MapFrom(src => src.Date.Date))
        .ForMember(dest => dest.EventHour, opt => opt.MapFrom(src => src.Date.Hour))
        .ForMember(dest => dest.EventMinute, opt => opt.MapFrom(src => src.Date.
↪Minute)));

// Perform mapping
CalendarEventForm form = mapper.Map<CalendarEvent, CalendarEventForm>(calendarEvent);

form.EventDate.ShouldEqual(new DateTime(2008, 12, 15));
form.EventHour.ShouldEqual(20);
form.EventMinute.ShouldEqual(30);
form.Title.ShouldEqual("Company Holiday Party");
```

Each custom member configuration uses an action delegate to configure each individual member. In the above example, we used the `MapFrom` option to perform custom source-to-destination member mappings. The `MapFrom` method takes a lambda expression as a parameter, which is then evaluated later during mapping. The `MapFrom` expression can be any `Func<TSource, object>` lambda expression.

CHAPTER 8

Nested Mappings

As the mapping engine executes the mapping, it can use one of a variety of methods to resolve a destination member value. One of these methods is to use another type map, where the source member type and destination member type are also configured in the mapping configuration. This allows us to not only flatten our source types, but create complex destination types as well. For example, our source type might contain another complex type:

```
public class OuterSource
{
        public int Value { get; set; }
        public InnerSource Inner { get; set; }
}

public class InnerSource
{
        public int OtherValue { get; set; }
}
```

We *could* simply flatten the OuterSource.Inner.OtherValue to one InnerOtherValue property, but we might also want to create a corresponding complex type for the Inner property:

```
public class OuterDest
{
        public int Value { get; set; }
        public InnerDest Inner { get; set; }
}

public class InnerDest
{
        public int OtherValue { get; set; }
}
```

In that case, we would need to configure the additional source/destination type mappings:

```
var config = new MapperConfiguration(cfg => {
    cfg.CreateMap<OuterSource, OuterDest>();
```

(continues on next page)

(continued from previous page)

```
    cfg.CreateMap<InnerSource, InnerDest>();
});
config.AssertConfigurationIsValid();

var source = new OuterSource
        {
                Value = 5,
                Inner = new InnerSource {OtherValue = 15}
        };
var mapper = config.CreateMapper();
var dest = mapper.Map<OuterSource, OuterDest>(source);

dest.Value.ShouldEqual(5);
dest.Inner.ShouldNotBeNull();
dest.Inner.OtherValue.ShouldEqual(15);
```

A few things to note here:

- Order of configuring types does not matter
- Call to Map does not need to specify any inner type mappings, only the type map to use for the source value passed in

With both flattening and nested mappings, we can create a variety of destination shapes to suit whatever our needs may be.

CHAPTER 9

Lists and Arrays

AutoMapper only requires configuration of element types, not of any array or list type that might be used. For example, we might have a simple source and destination type:

```
public class Source
{
        public int Value { get; set; }
}

public class Destination
{
        public int Value { get; set; }
}
```

All the basic generic collection types are supported:

```
var configuration = new MapperConfiguration(cfg => cfg.CreateMap<Source, Destination>
↪());

var sources = new[]
        {
                new Source { Value = 5 },
                new Source { Value = 6 },
                new Source { Value = 7 }
        };

IEnumerable<Destination> ienumerableDest = mapper.Map<Source[], IEnumerable
↪<Destination>>(sources);
ICollection<Destination> icollectionDest = mapper.Map<Source[], ICollection
↪<Destination>>(sources);
IList<Destination> ilistDest = mapper.Map<Source[], IList<Destination>>(sources);
List<Destination> listDest = mapper.Map<Source[], List<Destination>>(sources);
Destination[] arrayDest = mapper.Map<Source[], Destination[]>(sources);
```

To be specific, the source collection types supported include:

- IEnumerable
- IEnumerable<T>
- ICollection
- ICollection<T>
- IList
- IList<T>
- List<T>
- Arrays

For the non-generic enumerable types, only unmapped, assignable types are supported, as AutoMapper will be unable to "guess" what types you're trying to map. As shown in the example above, it's not necessary to explicitly configure list types, only their member types.

When mapping to an existing collection, the destination collection is cleared first. If this is not what you want, take a look at AutoMapper.Collection.

9.1 Handling null collections

When mapping a collection property, if the source value is null AutoMapper will map the destination field to an empty collection rather than setting the destination value to null. This aligns with the behavior of Entity Framework and Framework Design Guidelines that believe C# references, arrays, lists, collections, dictionaries and IEnumerables should NEVER be null, ever.

This behavior can be changed by setting the `AllowNullCollections` property to true when configuring the mapper.

```
var configuration = new MapperConfiguration(cfg => {
    cfg.AllowNullCollections = true;
    cfg.CreateMap<Source, Destination>();
});
```

The setting can be applied globally and can be overridden per profile and per member with `AllowNull` and `DoNotAllowNull`.

9.2 Polymorphic element types in collections

Many times, we might have a hierarchy of types in both our source and destination types. AutoMapper supports polymorphic arrays and collections, such that derived source/destination types are used if found.

```
public class ParentSource
{
        public int Value1 { get; set; }
}

public class ChildSource : ParentSource
{
        public int Value2 { get; set; }
}
```

(continues on next page)

(continued from previous page)

```
public class ParentDestination
{
        public int Value1 { get; set; }
}

public class ChildDestination : ParentDestination
{
        public int Value2 { get; set; }
}
```

AutoMapper still requires explicit configuration for child mappings, as AutoMapper cannot "guess" which specific child destination mapping to use. Here is an example of the above types:

```
var configuration = new MapperConfiguration(c=> {
    c.CreateMap<ParentSource, ParentDestination>()
              .Include<ChildSource, ChildDestination>();
    c.CreateMap<ChildSource, ChildDestination>();
});

var sources = new[]
        {
                new ParentSource(),
                new ChildSource(),
                new ParentSource()
        };

var destinations = mapper.Map<ParentSource[], ParentDestination[]>(sources);

destinations[0].ShouldBeInstanceOf<ParentDestination>();
destinations[1].ShouldBeInstanceOf<ChildDestination>();
destinations[2].ShouldBeInstanceOf<ParentDestination>();
```

CHAPTER 10

Construction

AutoMapper can map to destination constructors based on source members:

```
public class Source {
    public int Value { get; set; }
}
public class SourceDto {
    public SourceDto(int value) {
        _value = value;
    }
    private int _value;
    public int Value {
        get { return _value; }
    }
}
var configuration = new MapperConfiguration(cfg => cfg.CreateMap<Source, SourceDto>
↪());
```

If the destination constructor parameter names don't match, you can modify them at config time:

```
public class Source {
    public int Value { get; set; }
}
public class SourceDto {
    public SourceDto(int valueParamSomeOtherName) {
        _value = valueParamSomeOtherName;
    }
    private int _value;
    public int Value {
        get { return _value; }
    }
}
var configuration = new MapperConfiguration(cfg =>
  cfg.CreateMap<Source, SourceDto>()
    .ForCtorParam("valueParamSomeOtherName", opt => opt.MapFrom(src => src.Value))
);
```

This works for both LINQ projections and in-memory mapping.

You can also disable constructor mapping:

```
var configuration = new MapperConfiguration(cfg => cfg.DisableConstructorMapping());
```

You can configure which constructors are considered for the destination object:

```
// don't map private constructors
var configuration = new MapperConfiguration(cfg => cfg.ShouldUseConstructor = ci => !
↪ci.IsPrivate);
```

CHAPTER 11

Flattening

One of the common usages of object-object mapping is to take a complex object model and flatten it to a simpler model. You can take a complex model such as:

```
public class Order
{
        private readonly IList<OrderLineItem> _orderLineItems = new List
↪<OrderLineItem>();

        public Customer Customer { get; set; }

        public OrderLineItem[] GetOrderLineItems()
        {
                return _orderLineItems.ToArray();
        }

        public void AddOrderLineItem(Product product, int quantity)
        {
                _orderLineItems.Add(new OrderLineItem(product, quantity));
        }

        public decimal GetTotal()
        {
                return _orderLineItems.Sum(li => li.GetTotal());
        }
}

public class Product
{
        public decimal Price { get; set; }
        public string Name { get; set; }
}

public class OrderLineItem
{
```

(continues on next page)

(continued from previous page)

```
        public OrderLineItem(Product product, int quantity)
        {
                Product = product;
                Quantity = quantity;
        }

        public Product Product { get; private set; }
        public int Quantity { get; private set;}

        public decimal GetTotal()
        {
                return Quantity*Product.Price;
        }
}

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

We want to flatten this complex Order object into a simpler OrderDto that contains only the data needed for a certain scenario:

```
public class OrderDto
{
        public string CustomerName { get; set; }
        public decimal Total { get; set; }
}
```

When you configure a source/destination type pair in AutoMapper, the configurator attempts to match properties and methods on the source type to properties on the destination type. If for any property on the destination type a property, method, or a method prefixed with "Get" does not exist on the source type, AutoMapper splits the destination member name into individual words (by PascalCase conventions).

```
// Complex model

var customer = new Customer
        {
                Name = "George Costanza"
        };
var order = new Order
        {
                Customer = customer
        };
var bosco = new Product
        {
                Name = "Bosco",
                Price = 4.99m
        };
order.AddOrderLineItem(bosco, 15);

// Configure AutoMapper

var configuration = new MapperConfiguration(cfg => cfg.CreateMap<Order, OrderDto>());

// Perform mapping
```

(continues on next page)

(continued from previous page)

```
OrderDto dto = mapper.Map<Order, OrderDto>(order);

dto.CustomerName.ShouldEqual("George Costanza");
dto.Total.ShouldEqual(74.85m);
```

We configured the type map in AutoMapper with the CreateMap method. AutoMapper can only map type pairs it knows about, so we have explicitly register the source/destination type pair with CreateMap. To perform the mapping, we use the Map method.

On the OrderDto type, the Total property matched to the GetTotal() method on Order. The CustomerName property matched to the Customer.Name property on Order. As long as we name our destination properties appropriately, we do not need to configure individual property matching.

If you want to disable this behavior, you can use the `ExactMatchNamingConvention`:

```
cfg.DestinationMemberNamingConvention = new ExactMatchNamingConvention();
```

11.1 IncludeMembers

If you need more control when flattening, you can use IncludeMembers. You can map members of a child object to the destination object when you already have a map from the child type to the destination type (unlike the classic flattening that doesn't require a map for the child type).

```
class Source
{
    public string Name { get; set; }
    public InnerSource InnerSource { get; set; }
    public OtherInnerSource OtherInnerSource { get; set; }
}
class InnerSource
{
    public string Name { get; set; }
    public string Description { get; set; }
}
class OtherInnerSource
{
    public string Name { get; set; }
    public string Description { get; set; }
    public string Title { get; set; }
}
class Destination
{
    public string Name { get; set; }
    public string Description { get; set; }
    public string Title { get; set; }
}

cfg.CreateMap<Source, Destination>().IncludeMembers(s=>s.InnerSource, s=>s.
↪OtherInnerSource);
cfg.CreateMap<InnerSource, Destination>(MemberList.None);
cfg.CreateMap<OtherInnerSource, Destination>();

var source = new Source { Name = "name", InnerSource = new InnerSource{ Description =
↪"description" },
```

(continues on next page)

(continued from previous page)

```
                            OtherInnerSource = new OtherInnerSource{ Title = "title" } }
↪;
var destination = mapper.Map<Destination>(source);
destination.Name.ShouldBe("name");
destination.Description.ShouldBe("description");
destination.Title.ShouldBe("title");
```

So this allows you to reuse the configuration in the existing maps for the child types `InnerSource` and `OtherInnerSource` when mapping the parent types `Source` and `Destination`. It works in a similar way to mapping inheritance, but it uses composition, not inheritance.

The order of the parameters in the `IncludeMembers` call is relevant. When mapping a destination member, the first match wins, starting with the source object itself and then with the included child objects in the order you specified. So in the example above, `Name` is mapped from the source object itself and `Description` from `InnerSource` because it's the first match.

IncludeMembers integrates with `ReverseMap`. An included member will be reversed to

```
ForPath(destination => destination.IncludedMember, member => member.MapFrom(source =>␣
↪source))
```

and the other way around. If that's not what you want, you can avoid `ReverseMap` (explicitly create the reverse map) or you can override the default settings (using `Ignore` or `IncludeMembers` without parameters respectively).

For details, check the tests.

CHAPTER 12

Reverse Mapping and Unflattening

Starting with 6.1.0, AutoMapper now supports richer reverse mapping support. Given our entities:

```
public class Order {
  public decimal Total { get; set; }
  public Customer Customer { get; set; }
}

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

We can flatten this into a DTO:

```
public class OrderDto {
  public decimal Total { get; set; }
  public string CustomerName { get; set; }
}
```

We can map both directions, including unflattening:

```
var configuration = new MapperConfiguration(cfg => {
  cfg.CreateMap<Order, OrderDto>()
     .ReverseMap();
});
```

By calling `ReverseMap`, AutoMapper creates a reverse mapping configuration that includes unflattening:

```
var customer = new Customer {
  Name = "Bob"
};

var order = new Order {
  Customer = customer,
  Total = 15.8m
```

(continues on next page)

(continued from previous page)

```
};

var orderDto = mapper.Map<Order, OrderDto>(order);

orderDto.CustomerName = "Joe";

mapper.Map(orderDto, order);

order.Customer.Name.ShouldEqual("Joe");
```

Unflattening is only configured for `ReverseMap`. If you want unflattening, you must configure `Entity` -> `Dto` then call `ReverseMap` to create an unflattening type map configuration from the `Dto` -> `Entity`.

12.1 Customizing reverse mapping

AutoMapper will automatically reverse map "Customer.Name" from "CustomerName" based on the original flattening. If you use MapFrom, AutoMapper will attempt to reverse the map:

```
cfg.CreateMap<Order, OrderDto>()
  .ForMember(d => d.CustomerName, opt => opt.MapFrom(src => src.Customer.Name))
  .ReverseMap();
```

As long as the `MapFrom` path are member accessors, AutoMapper will unflatten from the same path (`CustomerName` => `Customer.Name`).

If you need to customize this, for a reverse map you can use `ForPath`:

```
cfg.CreateMap<Order, OrderDto>()
  .ForMember(d => d.CustomerName, opt => opt.MapFrom(src => src.Customer.Name))
  .ReverseMap()
  .ForPath(s => s.Customer.Name, opt => opt.MapFrom(src => src.CustomerName));
```

For most cases you shouldn't need this, as the original MapFrom will be reversed for you. Use ForPath when the path to get and set the values are different.

If you do not want unflattening behavior, you can remove the call to `ReverseMap` and create two separate maps. Or, you can use Ignore:

```
cfg.CreateMap<Order, OrderDto>()
  .ForMember(d => d.CustomerName, opt => opt.MapFrom(src => src.Customer.Name))
  .ReverseMap()
  .ForPath(s => s.Customer.Name, opt => opt.Ignore());
```

12.2 IncludeMembers

`ReverseMap` also integrates with `IncludeMembers` and configuration like

```
ForMember(destination => destination.IncludedMember, member => member.MapFrom(source␣
↪=> source))
```

CHAPTER 13

Mapping Inheritance

Mapping inheritance serves two functions:

- Inheriting mapping configuration from a base class or interface configuration
- Runtime polymorphic mapping

Inheriting base class configuration is opt-in, and you can either explicitly specify the mapping to inherit from the base type configuration with `Include` or in the derived type configuration with `IncludeBase`:

```
CreateMap<BaseEntity, BaseDto>()
   .Include<DerivedEntity, DerivedDto>()
   .ForMember(dest => dest.SomeMember, opt => opt.MapFrom(src => src.OtherMember));

CreateMap<DerivedEntity, DerivedDto>();
```

or

```
CreateMap<BaseEntity, BaseDto>()
   .ForMember(dest => dest.SomeMember, opt => opt.MapFrom(src => src.OtherMember));

CreateMap<DerivedEntity, DerivedDto>()
    .IncludeBase<BaseEntity, BaseDto>();
```

In each case above, the derived mapping inherits the custom mapping configuration from the base mapping configuration.

To include all derived maps, from the base type map configuration:

```
CreateMap<BaseEntity, BaseDto>()
    .IncludeAllDerived();

CreateMap<DerivedEntity, DerivedDto>();
```

13.1 Runtime polymorphism

Take:

```
public class Order { }
public class OnlineOrder : Order { }
public class MailOrder : Order { }

public class OrderDto { }
public class OnlineOrderDto : OrderDto { }
public class MailOrderDto : OrderDto { }

var configuration = new MapperConfiguration(cfg => {
    cfg.CreateMap<Order, OrderDto>()
        .Include<OnlineOrder, OnlineOrderDto>()
        .Include<MailOrder, MailOrderDto>();
    cfg.CreateMap<OnlineOrder, OnlineOrderDto>();
    cfg.CreateMap<MailOrder, MailOrderDto>();
});

// Perform Mapping
var order = new OnlineOrder();
var mapped = mapper.Map(order, order.GetType(), typeof(OrderDto));
Assert.IsType<OnlineOrderDto>(mapped);
```

You will notice that because the mapped object is a OnlineOrder, AutoMapper has seen you have a more specific mapping for OnlineOrder than OrderDto, and automatically chosen that.

13.2 Specifying inheritance in derived classes

Instead of configuring inheritance from the base class, you can specify inheritance from the derived classes:

```
var configuration = new MapperConfiguration(cfg => {
  cfg.CreateMap<Order, OrderDto>()
    .ForMember(o => o.Id, m => m.MapFrom(s => s.OrderId));
  cfg.CreateMap<OnlineOrder, OnlineOrderDto>()
    .IncludeBase<Order, OrderDto>();
  cfg.CreateMap<MailOrder, MailOrderDto>()
    .IncludeBase<Order, OrderDto>();
});
```

13.3 As

For simple cases, you can use `As` to redirect a base map to an existing derived map:

```
    cfg.CreateMap<Order, OnlineOrderDto>();
    cfg.CreateMap<Order, OrderDto>().As<OnlineOrderDto>();

    mapper.Map<OrderDto>(new Order()).ShouldBeOfType<OnlineOrderDto>();
```

13.4 Inheritance Mapping Priorities

This introduces additional complexity because there are multiple ways a property can be mapped. The priority of these sources are as follows

- Explicit Mapping (using .MapFrom())
- Inherited Explicit Mapping
- Ignore Property Mapping
- Convention Mapping (Properties that are matched via convention)

To demonstrate this, lets modify our classes shown above

```
//Domain Objects
public class Order { }
public class OnlineOrder : Order
{
    public string Referrer { get; set; }
}
public class MailOrder : Order { }

//Dtos
public class OrderDto
{
    public string Referrer { get; set; }
}

//Mappings
var configuration = new MapperConfiguration(cfg => {
    cfg.CreateMap<Order, OrderDto>()
        .Include<OnlineOrder, OrderDto>()
        .Include<MailOrder, OrderDto>()
        .ForMember(o=>o.Referrer, m=>m.Ignore());
    cfg.CreateMap<OnlineOrder, OrderDto>();
    cfg.CreateMap<MailOrder, OrderDto>();
});

// Perform Mapping
var order = new OnlineOrder { Referrer = "google" };
var mapped = mapper.Map(order, order.GetType(), typeof(OrderDto));
Assert.IsNull(mapped.Referrer);
```

Notice that in our mapping configuration, we have ignored `Referrer` (because it doesn't exist in the order base class) and that has a higher priority than convention mapping, so the property doesn't get mapped.

If you do want the `Referrer` property to be mapped in the mapping from `OnlineOrder` to `OrderDto` you should include an explicit mapping in the mapping like this:

```
    cfg.CreateMap<OnlineOrder, OrderDto>()
        .ForMember(o=>o.Referrer, m=>m.MapFrom(x=>x.Referrer));
```

Overall this feature should make using AutoMapper with classes that leverage inheritance feel more natural.

CHAPTER 14

Attribute Mapping

In addition to fluent configuration is the ability to declare and configure maps via attributes. Attribute maps can supplement or replace fluent mapping configuration.

14.1 Type Map configuration

In order to search for maps to configure, use the `AddMaps` method:

```
var configuration = new MapperConfiguration(cfg => cfg.AddMaps("MyAssembly"));

var mapper = new Mapper(configuration);
```

`AddMaps` looks for fluent map configuration (`Profile` classes) and attribute-based mappings.

To declare an attribute map, decorate your destination type with the `AutoMapAttribute`:

```
[AutoMap(typeof(Order))]
public class OrderDto {
    // destination members
```

This is equivalent to a `CreateMap<Order, OrderDto>()` configuration.

14.1.1 Customizing type map configuration

To customize the overall type map configuration, you can set the following properties on the `AutoMapAttribute`:

- ReverseMap (bool)
- ConstructUsingServiceLocator (bool)
- MaxDepth (int)
- PreserveReferences (bool)
- DisableCtorValidation (bool)

- IncludeAllDerived (bool)
- TypeConverter (Type)

These all correspond to the similar fluent mapping configuration options. Only the `sourceType` value is required to map.

14.2 Member configuration

For attribute-based maps, you can decorate individual members with additional configuration. Because attributes have limitations in C# (no expressions, for example), the configuration options available are a bit limited.

Member-based attributes are declared in the `AutoMapper.Configuration.Annotations` namespace.

If the attribute-based configuration is not available or will not work, you can combine both attribute and profile-based maps (though this may be confusing).

14.2.1 Ignoring members

Use the `IgnoreAttribute` to ignore an individual destination member from mapping and/or validation:

```
using AutoMapper.Configuration.Annotations;

[AutoMap(typeof(Order))]
public class OrderDto {
    [Ignore]
    public decimal Total { get; set; }
```

14.2.2 Redirecting to a different source member

It is not possible to use `MapFrom` with an expression in an attribute, but `SourceMemberAttribute` can redirect to a separate named member:

```
using AutoMapper.Configuration.Annotations;

[AutoMap(typeof(Order))]
public class OrderDto {
   [SourceMember("OrderTotal")]
   public decimal Total { get; set; }
```

Or use the `nameof` operator:

```
using AutoMapper.Configuration.Annotations;

[AutoMap(typeof(Order))]
public class OrderDto {
   [SourceMember(nameof(Order.OrderTotal))]
   public decimal Total { get; set; }
```

You cannot flatten with this attribute, only redirect source type members (i.e. no "Order.Customer.Office.Name" in the name). Configuring flattening is only available with the fluent configuration.

14.2.3 Additional configuration options

Additional attribute-based configuration options include:

- `MapAtRuntimeAttribute`
- `MappingOrderAttribute`
- `NullSubstituteAttribute`
- `UseExistingValueAttribute`
- `ValueConverterAttribute`
- `ValueResolverAttribute`

Each corresponds to the same fluent configuration mapping option.

CHAPTER 15

Dynamic and ExpandoObject Mapping

AutoMapper can map to/from dynamic objects without any explicit configuration:

```
public class Foo {
    public int Bar { get; set; }
    public int Baz { get; set; }
    public Foo InnerFoo { get; set; }
}
dynamic foo = new MyDynamicObject();
foo.Bar = 5;
foo.Baz = 6;

var configuration = new MapperConfiguration(cfg => {});

var result = mapper.Map<Foo>(foo);
result.Bar.ShouldEqual(5);
result.Baz.ShouldEqual(6);

dynamic foo2 = mapper.Map<MyDynamicObject>(result);
foo2.Bar.ShouldEqual(5);
foo2.Baz.ShouldEqual(6);
```

Similarly you can map straight from `Dictionary<string, object>` to objects, AutoMapper will line up the keys with property names. For mapping to destination child objects, you can use the dot notation.

```
var result = mapper.Map<Foo>(new Dictionary<string, object> { ["InnerFoo.Bar"] = 42 }
↪);
result.InnerFoo.Bar.ShouldEqual(42);
```

CHAPTER 16

Open Generics

AutoMapper can support an open generic type map. Create a map for the open generic types:

```
public class Source<T> {
    public T Value { get; set; }
}

public class Destination<T> {
    public T Value { get; set; }
}

// Create the mapping
var configuration = new MapperConfiguration(cfg => cfg.CreateMap(typeof(Source<>),␣
↪typeof(Destination<>)));
```

You don't need to create maps for closed generic types. AutoMapper will apply any configuration from the open generic mapping to the closed mapping at runtime:

```
var source = new Source<int> { Value = 10 };

var dest = mapper.Map<Source<int>, Destination<int>>(source);

dest.Value.ShouldEqual(10);
```

Because C# only allows closed generic type parameters, you have to use the System.Type version of CreateMap to create your open generic type maps. From there, you can use all of the mapping configuration available and the open generic configuration will be applied to the closed type map at runtime. AutoMapper will skip open generic type maps during configuration validation, since you can still create closed types that don't convert, such as `Source<Foo> -> Destination<Bar>` where there is no conversion from Foo to Bar.

You can also create an open generic type converter:

```
var configuration = new MapperConfiguration(cfg =>
   cfg.CreateMap(typeof(Source<>), typeof(Destination<>)).
↪ConvertUsing(typeof(Converter<>)));
```

AutoMapper also supports open generic type converters with any number of generic arguments:

```
var configuration = new MapperConfiguration(cfg =>
   cfg.CreateMap(typeof(Source<>), typeof(Destination<>)).
↪ConvertUsing(typeof(Converter<,>)));
```

The closed type from `Source` will be the first generic argument, and the closed type of `Destination` will be the second argument to close `Converter<,>`.

The same idea applies to value resolvers. Check the tests.

CHAPTER 17

Queryable Extensions

When using an ORM such as NHibernate or Entity Framework with AutoMapper's standard `mapper.Map` functions, you may notice that the ORM will query all the fields of all the objects within a graph when AutoMapper is attempting to map the results to a destination type.

If your ORM exposes `IQueryables`, you can use AutoMapper's QueryableExtensions helper methods to address this key pain.

Using Entity Framework for an example, say that you have an entity `OrderLine` with a relationship with an entity `Item`. If you want to map this to an `OrderLineDTO` with the `Item`'s `Name` property, the standard `mapper.Map` call will result in Entity Framework querying the entire `OrderLine` and `Item` table.

Use this approach instead.

Given the following entities:

```
public class OrderLine
{
  public int Id { get; set; }
  public int OrderId { get; set; }
  public Item Item { get; set; }
  public decimal Quantity { get; set; }
}

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

And the following DTO:

```
public class OrderLineDTO
{
  public int Id { get; set; }
  public int OrderId { get; set; }
```

(continues on next page)

(continued from previous page)

```
  public string Item { get; set; }
  public decimal Quantity { get; set; }
}
```

You can use the Queryable Extensions like so:

```
var configuration = new MapperConfiguration(cfg =>
    cfg.CreateMap<OrderLine, OrderLineDTO>()
    .ForMember(dto => dto.Item, conf => conf.MapFrom(ol => ol.Item.Name)));

public List<OrderLineDTO> GetLinesForOrder(int orderId)
{
  using (var context = new orderEntities())
  {
    return context.OrderLines.Where(ol => ol.OrderId == orderId)
              .ProjectTo<OrderLineDTO>(configuration).ToList();
  }
}
```

The `.ProjectTo<OrderLineDTO>()` will tell AutoMapper's mapping engine to emit a `select` clause to the IQueryable that will inform entity framework that it only needs to query the Name column of the Item table, same as if you manually projected your `IQueryable` to an `OrderLineDTO` with a `Select` clause.

Note that for this feature to work, all type conversions must be explicitly handled in your Mapping. For example, you can not rely on the `ToString()` override of the `Item` class to inform entity framework to only select from the `Name` column, and any data type changes, such as `Double` to `Decimal` must be explicitly handled as well.

17.1 The instance API

Starting with 8.0 there are similar ProjectTo methods on IMapper that feel more natural when you use IMapper with DI.

17.2 Preventing lazy loading/SELECT N+1 problems

Because the LINQ projection built by AutoMapper is translated directly to a SQL query by the query provider, the mapping occurs at the SQL/ADO.NET level, and not touching your entities. All data is eagerly fetched and loaded into your DTOs.

Nested collections use a Select to project child DTOs:

```
from i in db.Instructors
orderby i.LastName
select new InstructorIndexData.InstructorModel
{
    ID = i.ID,
    FirstMidName = i.FirstMidName,
    LastName = i.LastName,
    HireDate = i.HireDate,
    OfficeAssignmentLocation = i.OfficeAssignment.Location,
    Courses = i.Courses.Select(c => new InstructorIndexData.InstructorCourseModel
    {
        CourseID = c.CourseID,
```

(continues on next page)

(continued from previous page)

```
        CourseTitle = c.Title
    }).ToList()
};
```

This map through AutoMapper will result in a SELECT N+1 problem, as each child `Course` will be queried one at a time, unless specified through your ORM to eagerly fetch. With LINQ projection, no special configuration or specification is needed with your ORM. The ORM uses the LINQ projection to build the exact SQL query needed.

17.3 Custom projection

In the case where members names don't line up, or you want to create calculated property, you can use MapFrom (the expression-based overload) to supply a custom expression for a destination member:

```
var configuration = new MapperConfiguration(cfg => cfg.CreateMap<Customer,␣
↪CustomerDto>()
    .ForMember(d => d.FullName, opt => opt.MapFrom(c => c.FirstName + " " + c.
↪LastName))
    .ForMember(d => d.TotalContacts, opt => opt.MapFrom(c => c.Contacts.Count()));
```

AutoMapper passes the supplied expression with the built projection. As long as your query provider can interpret the supplied expression, everything will be passed down all the way to the database.

If the expression is rejected from your query provider (Entity Framework, NHibernate, etc.), you might need to tweak your expression until you find one that is accepted.

17.4 Custom Type Conversion

Occasionally, you need to completely replace a type conversion from a source to a destination type. In normal runtime mapping, this is accomplished via the ConvertUsing method. To perform the analog in LINQ projection, use the ConvertUsing method:

```
cfg.CreateMap<Source, Dest>().ConvertUsing(src => new Dest { Value = 10 });
```

The expression-based `ConvertUsing` is slightly more limited than Func-based `ConvertUsing` overloads as only what is allowed in an Expression and the underlying LINQ provider will work.

17.5 Custom destination type constructors

If your destination type has a custom constructor but you don't want to override the entire mapping, use the ConstructUsing expression-based method overload:

```
cfg.CreateMap<Source, Dest>()
    .ConstructUsing(src => new Dest(src.Value + 10));
```

AutoMapper will automatically match up destination constructor parameters to source members based on matching names, so only use this method if AutoMapper can't match up the destination constructor properly, or if you need extra customization during construction.

17.6 String conversion

AutoMapper will automatically add `ToString()` when the destination member type is a string and the source member type is not.

```
public class Order {
    public OrderTypeEnum OrderType { get; set; }
}
public class OrderDto {
    public string OrderType { get; set; }
}
var orders = dbContext.Orders.ProjectTo<OrderDto>(configuration).ToList();
orders[0].OrderType.ShouldEqual("Online");
```

17.7 Explicit expansion

In some scenarios, such as OData, a generic DTO is returned through an IQueryable controller action. Without explicit instructions, AutoMapper will expand all members in the result. To control which members are expanded during projection, set ExplicitExpansion in the configuration and then pass in the members you want to explicitly expand:

```
dbContext.Orders.ProjectTo<OrderDto>(configuration,
    dest => dest.Customer,
    dest => dest.LineItems);
// or string-based
dbContext.Orders.ProjectTo<OrderDto>(configuration,
    null,
    "Customer",
    "LineItems");
// for collections
dbContext.Orders.ProjectTo<OrderDto>(configuration,
    null,
    dest => dest.LineItems.Select(item => item.Product));
```

For more information, see the tests.

17.8 Aggregations

LINQ can support aggregate queries, and AutoMapper supports LINQ extension methods. In the custom projection example, if we renamed the `TotalContacts` property to `ContactsCount`, AutoMapper would match to the `Count()` extension method and the LINQ provider would translate the count into a correlated subquery to aggregate child records.

AutoMapper can also support complex aggregations and nested restrictions, if the LINQ provider supports it:

```
cfg.CreateMap<Course, CourseModel>()
    .ForMember(m => m.EnrollmentsStartingWithA,
          opt => opt.MapFrom(c => c.Enrollments.Where(e => e.Student.LastName.
↪StartsWith("A")).Count()));
```

This query returns the total number of students, for each course, whose last name starts with the letter 'A'.

17.9 Parameterization

Occasionally, projections need runtime parameters for their values. Consider a projection that needs to pull in the current username as part of its data. Instead of using post-mapping code, we can parameterize our MapFrom configuration:

```
string currentUserName = null;
cfg.CreateMap<Course, CourseModel>()
    .ForMember(m => m.CurrentUserName, opt => opt.MapFrom(src => currentUserName));
```

When we project, we'll substitute our parameter at runtime:

```
dbContext.Courses.ProjectTo<CourseModel>(Config, new { currentUserName = Request.User.
↪Name });
```

This works by capturing the name of the closure's field name in the original expression, then using an anonymous object/dictionary to apply the value to the parameter value before the query is sent to the query provider.

You may also use a dictionary to build the projection values:

```
dbContext.Courses.ProjectTo<CourseModel>(Config, new Dictionary<string, object> { {
↪"currentUserName", Request.User.Name} });
```

However, using a dictionary will result in hard-coded values in the query instead of a parameterized query, so use with caution.

17.10 Supported mapping options

Not all mapping options can be supported, as the expression generated must be interpreted by a LINQ provider. Only what is supported by LINQ providers is supported by AutoMapper:

- MapFrom (Expression-based)
- ConvertUsing (Expression-based)
- Ignore
- NullSubstitute
- Value transformers

Not supported:

- Condition
- SetMappingOrder
- UseDestinationValue
- MapFrom (Func-based)
- Before/AfterMap
- Custom resolvers
- Custom type converters
- ForPath
- Value converters

- **Any calculated property on your domain object**

Additionally, recursive or self-referencing destination types are not supported as LINQ providers do not support this. Typically hierarchical relational data models require common table expressions (CTEs) to correctly resolve a recursive join.

CHAPTER 18

Expression Translation (UseAsDataSource)

Automapper supports translating Expressions from one object to another in a separate package. This is done by substituting the properties from the source class to what they map to in the destination class.

Given the example classes:

```
public class OrderLine
{
  public int Id { get; set; }
  public int OrderId { get; set; }
  public Item Item { get; set; }
  public decimal Quantity { get; set; }
}

public class Item
{
  public int Id { get; set; }
  public string Name { get; set; }
}

public class OrderLineDTO
{
  public int Id { get; set; }
  public int OrderId { get; set; }
  public string Item { get; set; }
  public decimal Quantity { get; set; }
}

var configuration = new MapperConfiguration(cfg =>
{
  cfg.AddExpressionMapping();

  cfg.CreateMap<OrderLine, OrderLineDTO>()
    .ForMember(dto => dto.Item, conf => conf.MapFrom(ol => ol.Item.Name));
  cfg.CreateMap<OrderLineDTO, OrderLine>()
    .ForMember(ol => ol.Item, conf => conf.MapFrom(dto => dto));
```

(continues on next page)

(continued from previous page)

```
  cfg.CreateMap<OrderLineDTO, Item>()
    .ForMember(i => i.Name, conf => conf.MapFrom(dto => dto.Item));
});
```

When mapping from DTO Expression

```
Expression<Func<OrderLineDTO, bool>> dtoExpression = dto=> dto.Item.StartsWith("A");
var expression = mapper.Map<Expression<Func<OrderLine, bool>>>(dtoExpression);
```

Expression will be translated to `ol => ol.Item.Name.StartsWith("A")`

Automapper knows `dto.Item` is mapped to `ol.Item.Name` so it substituted it for the expression.

Expression translation can work on expressions of collections as well.

```
Expression<Func<IQueryable<OrderLineDTO>,IQueryable<OrderLineDTO>>> dtoExpression =␣
↪dtos => dtos.Where(dto => dto.Quantity > 5).OrderBy(dto => dto.Quantity);
var expression = mapper.Map<Expression<Func<IQueryable<OrderLine>,IQueryable
↪<OrderLine>>>(dtoExpression);
```

Resulting in `ols => ols.Where(ol => ol.Quantity > 5).OrderBy(ol => ol.Quantity)`

18.1 Mapping Flattened Properties to Navigation Properties

AutoMapper also supports mapping flattened (TModel or DTO) properties in expressions to their corresponding (TData) navigation properties (when the navigation property has been removed from the view model or DTO) e.g. CourseModel.DepartmentName from the model expression becomes Course.Department in the data expression.

Take the following set of classes:

```
public class CourseModel
{
    public int CourseID { get; set; }

    public int DepartmentID { get; set; }
    public string DepartmentName { get; set; }
}
public class Course
{
    public int CourseID { get; set; }

    public int DepartmentID { get; set; }
    public Department Department { get; set; }
}

public class Department
{
    public int DepartmentID { get; set; }
    public string Name { get; set; }
}
```

Then map exp below to expMapped.

```
Expression<Func<IQueryable<CourseModel>, IIncludableQueryable<CourseModel, object>>>␣
↪exp = i => i.Include(s => s.DepartmentName);
Expression<Func<IQueryable<Course>, IIncludableQueryable<Course, object>>> expMapped␣
↪= mapper.MapExpressionAsInclude<Expression<Func<IQueryable<Course>,␣
↪IIncludableQueryable<Course, object>>>>(exp);
```

The resulting mapped expression (expMapped.ToString()) is then `i => i.Include(s => s.Department);` . This feature allows navigation properties for the query to be defined based on the view model alone.

18.2 Supported Mapping options

Much like how Queryable Extensions can only support certain things that the LINQ providers support, expression translation follows the same rules as what it can and can't support.

18.3 UseAsDataSource

Mapping expressions to one another is a tedious and produces long ugly code.

`UseAsDataSource().For<DTO>()` makes this translation clean by not having to explicitly map expressions. It also calls `ProjectTo<TDO>()` for you as well, where applicable.

Using EntityFramework as an example

```
dataContext.OrderLines.UseAsDataSource().For<OrderLineDTO>().Where(dto => dto.
Name.StartsWith("A"))
```

Does the equivalent of

```
dataContext.OrderLines.Where(ol => ol.Item.Name.StartsWith("A")).
ProjectTo<OrderLineDTO>()
```

18.3.1 When ProjectTo() is not called

Expression Translation works for all kinds of functions, including `Select` calls. If `Select` is used after `UseAsDataSource()` and changes the return type, then `ProjectTo<>()` won't be called and `mapper.Map` will be used instead.

Example:

```
dataContext.OrderLines.UseAsDataSource().For<OrderLineDTO>().Select(dto =>
dto.Name)
```

Does the equivalent of

```
dataContext.OrderLines.Select(ol => ol.Item.Name)
```

18.3.2 Register a callback, for when an UseAsDataSource() query is enumerated

Sometimes, you may want to edit the collection, that is returned from a mapped query before forwarding it to the next application layer. With `.ProjectTo<TDto>` this is quite simple, as there is no sense in directly returning the resulting `IQueryable<TDto>` because you cannot edit it anymore anyways. So you will most likely do this:

```
var configuration = new MapperConfiguration(cfg =>
    cfg.CreateMap<OrderLine, OrderLineDTO>()
    .ForMember(dto => dto.Item, conf => conf.MapFrom(ol => ol.Item.Name)));

public List<OrderLineDTO> GetLinesForOrder(int orderId)
{
  using (var context = new orderEntities())
  {
    var dtos = context.OrderLines.Where(ol => ol.OrderId == orderId)
              .ProjectTo<OrderLineDTO>().ToList();
    foreach(var dto in dtos)
    {
        // edit some property, or load additional data from the database and augment␣
↪the dtos
    }
    return dtos;
  }
}
```

However, if you did this with the `.UseAsDataSource()` approach, you would lose all of its power - namely its ability to modify the internal expression until it is enumerated. To solve that problem, we introduced the `.OnEnumerated` callback. Using it, you can do the following:

```
var configuration = new MapperConfiguration(cfg =>
    cfg.CreateMap<OrderLine, OrderLineDTO>()
    .ForMember(dto => dto.Item, conf => conf.MapFrom(ol => ol.Item.Name)));

public IQueryable<OrderLineDTO> GetLinesForOrder(int orderId)
{
  using (var context = new orderEntities())
  {
    return context.OrderLines.Where(ol => ol.OrderId == orderId)
              .UseAsDataSource()
              .For<OrderLineDTO>()
              .OnEnumerated((dtos) =>
              {
                 foreach(var dto in dtosCast<OrderLineDTO>())
                 {
                      // edit some property, or load additional data from the database␣
↪and augment the dtos
                 }
              }
   }
}
```

this `OnEnumerated(IEnumerable)` callback is executed, when the `IQueryable<OrderLineDTO>` itself is enumerated. So this also works with the OData samples mentioned above: The OData $filter and $orderby expressions are still converted into SQL, and the `OnEnumerated()` callback is provided with the filtered, ordered resultset from the database.

CHAPTER 19

AutoMapper.Extensions.EnumMapping

The built-in enum mapper is not configurable, it can only be replaced. Alternatively, AutoMapper supports convention based mapping of enum values in a separate package AutoMapper.Extensions.EnumMapping.

19.1 Usage

For method `CreateMap` this library provide a `ConvertUsingEnumMapping` method. This method add all default mappings from source to destination enum values.

If you want to change some mappings, then you can use `MapValue` method. This is a chainable method.

Default the enum values are mapped by value (explicitly: `MapByValue()`), but it is possible to map by name calling `MapByName()`.

```
using AutoMapper.Extensions.EnumMapping;

public enum Source
{
    Default = 0,
    First = 1,
    Second = 2
}

public enum Destination
{
    Default = 0,
    Second = 2
}

internal class YourProfile : Profile
{
    public YourProfile()
    {
        CreateMap<Source, Destination>()
```

(continues on next page)

(continued from previous page)

```
            .ConvertUsingEnumMapping(opt => opt
                        // optional: .MapByValue() or MapByName(), without␣
↪configuration MapByValue is used
                        .MapValue(Source.First, Destination.Default)
            )
            .ReverseMap(); // to support Destination to Source mapping, including␣
↪custom mappings of ConvertUsingEnumMapping
    }
}
    ...
```

19.2 Default Convention

The package AutoMapper.Extensions.EnumMapping will map all values from Source type to Destination type if both enum types have the same value (or by name or by value). All Source enum values which have no Source equivalent, will throw an exception if EnumMappingValidation is enabled.

19.3 ReverseMap Convention

For method `ReverseMap` the same convention is used as for default mappings, but it also respects override enum value mappings if possible.

The following steps determines the reversed overrides:

1. Create mappings for `Source` to `Destination` (default convention), including custom overrides.
2. Create mappings for `Destination` to `Source` (default convention), without custom overrides (must be determined)
3. The mappings from step 1 will be used to determine the overrides for the `ReverseMap`. Therefore the mappings are grouped by `Destination` value.

   ```
   3a) if there is a matching `Source` value for the `Destination` value, then that␣
↪mapping is preferred and no override is needed
   ```

   It is possible that a `Destination` value has multiple `Source` values specified by override mappings.

   We have to determine which `Source` value will be the new `Destination` for the current `Destination` value (which is the new `Source` value)

   For every `Source` value per grouped `Destination` value:

   ```
   3b) if the `Source` enum value does not exists in the `Destination` enum type,␣
↪then that mapping cannot reversed

   3c) if there is a `Source` value which is not a `Destination` part of the␣
↪mappings from step 1, then that mapping cannot reversed

   3d) if the `Source` value is not excluded by option b and c, the that `Source`␣
↪value is the new `Destination` value.
   ```

4. All overrides which are determined in step 3 will be applied to mappings from step 2.
5. Finally, the custom mappings provided to method `ReverseMap` will be applied.

19.4 Testing

AutoMapper provides a nice tooling for validating typemaps. This package adds an extra `EnumMapperConfigurationExpressionExtensions.EnableEnumMappingValidation` extension method to extend the existing `AssertConfigurationIsValid()` method to validate also the enum mappings.

To enable testing the enum mapping configuration:

```
public class MappingConfigurationsTests
{
    [Fact]
    public void WhenProfilesAreConfigured_ItShouldNotThrowException()
    {
        // Arrange
        var config = new MapperConfiguration(configuration =>
        {
            configuration.EnableEnumMappingValidation();

            configuration.AddMaps(typeof(AssemblyInfo).GetTypeInfo().Assembly);
        });

        // Assert
        config.AssertConfigurationIsValid();
    }
}
```

CHAPTER 20

Custom Type Converters

Sometimes, you need to take complete control over the conversion of one type to another. This is typically when one type looks nothing like the other, a conversion function already exists, and you would like to go from a "looser" type to a stronger type, such as a source type of string to a destination type of Int32.

For example, suppose we have a source type of:

```
public class Source
{
        public string Value1 { get; set; }
        public string Value2 { get; set; }
        public string Value3 { get; set; }
}
```

But you would like to map it to:

```
public class Destination
{
        public int Value1 { get; set; }
        public DateTime Value2 { get; set; }
        public Type Value3 { get; set; }
}
```

If we were to try and map these two types as-is, AutoMapper would throw an exception (at map time and configuration-checking time), as AutoMapper does not know about any mapping from string to int, DateTime or Type. To create maps for these types, we must supply a custom type converter, and we have three ways of doing so:

```
void ConvertUsing(Func<TSource, TDestination> mappingFunction);
void ConvertUsing(ITypeConverter<TSource, TDestination> converter);
void ConvertUsing<TTypeConverter>() where TTypeConverter : ITypeConverter<TSource,␣
↪TDestination>;
```

The first option is simply any function that takes a source and returns a destination (there are several overloads too). This works for simple cases, but becomes unwieldy for larger ones. In more difficult cases, we can create a custom ITypeConverter<TSource, TDestination>:

```
public interface ITypeConverter<in TSource, TDestination>
{
        TDestination Convert(TSource source, TDestination destination,␣
↪ResolutionContext context);
}
```

And supply AutoMapper with either an instance of a custom type converter, or simply the type, which AutoMapper will instantiate at run time. The mapping configuration for our above source/destination types then becomes:

```
[Test]
public void Example()
{
    var configuration = new MapperConfiguration(cfg => {
      cfg.CreateMap<string, int>().ConvertUsing(s => Convert.ToInt32(s));
      cfg.CreateMap<string, DateTime>().ConvertUsing(new DateTimeTypeConverter());
      cfg.CreateMap<string, Type>().ConvertUsing<TypeTypeConverter>();
      cfg.CreateMap<Source, Destination>();
    });
    configuration.AssertConfigurationIsValid();

    var source = new Source
    {
        Value1 = "5",
        Value2 = "01/01/2000",
        Value3 = "AutoMapperSamples.GlobalTypeConverters.
↪GlobalTypeConverters+Destination"
    };

    Destination result = mapper.Map<Source, Destination>(source);
    result.Value3.ShouldEqual(typeof(Destination));
}

public class DateTimeTypeConverter : ITypeConverter<string, DateTime>
{
    public DateTime Convert(string source, DateTime destination, ResolutionContext␣
↪context)
    {
        return System.Convert.ToDateTime(source);
    }
}

public class TypeTypeConverter : ITypeConverter<string, Type>
{
    public Type Convert(string source, Type destination, ResolutionContext context)
    {
          return Assembly.GetExecutingAssembly().GetType(source);
    }
}
```

In the first mapping, from string to Int32, we simply use the built-in Convert.ToInt32 function (supplied as a method group). The next two use custom ITypeConverter implementations.

The real power of custom type converters is that they are used any time AutoMapper finds the source/destination pairs on any mapped types. We can build a set of custom type converters, on top of which other mapping configurations use, without needing any extra configuration. In the above example, we never have to specify the string/int conversion again. Where as Custom Value Resolvers have to be configured at a type member level, custom type converters are global in scope.

20.1 System Type Converters

The .NET Framework also supports the concepts of type converters, through the TypeConverter class. AutoMapper supports these types of type converters, in configuration checking and mapping, without the need for any manual configuration. AutoMapper uses the TypeDescriptor.GetConverter method for determining if the source/destination type pair can be mapped.

CHAPTER 21

Custom Value Resolvers

Although AutoMapper covers quite a few destination member mapping scenarios, there are the 1 to 5% of destination values that need a little help in resolving. Many times, this custom value resolution logic is domain logic that can go straight on our domain. However, if this logic pertains only to the mapping operation, it would clutter our source types with unnecessary behavior. In these cases, AutoMapper allows for configuring custom value resolvers for destination members. For example, we might want to have a calculated value just during mapping:

```
public class Source
{
        public int Value1 { get; set; }
        public int Value2 { get; set; }
}

public class Destination
{
        public int Total { get; set; }
}
```

For whatever reason, we want Total to be the sum of the source Value properties. For some other reason, we can't or shouldn't put this logic on our Source type. To supply a custom value resolver, we'll need to first create a type that implements IValueResolver:

```
public interface IValueResolver<in TSource, in TDestination, TDestMember>
{
        TDestMember Resolve(TSource source, TDestination destination, TDestMember␣
↪destMember, ResolutionContext context);
}
```

The ResolutionContext contains all of the contextual information for the current resolution operation, such as source type, destination type, source value and so on. An example implementation:

```
public class CustomResolver : IValueResolver<Source, Destination, int>
{
        public int Resolve(Source source, Destination destination, int member,␣
↪ResolutionContext context)
```

(continues on next page)

(continued from previous page)

```
        {
        return source.Value1 + source.Value2;
        }
}
```

Once we have our IValueResolver implementation, we'll need to tell AutoMapper to use this custom value resolver when resolving a specific destination member. We have several options in telling AutoMapper a custom value resolver to use, including:

- MapFrom<TValueResolver>
- MapFrom(typeof(CustomValueResolver))
- MapFrom(aValueResolverInstance)

In the below example, we'll use the first option, telling AutoMapper the custom resolver type through generics:

```
var configuration = new MapperConfiguration(cfg =>
   cfg.CreateMap<Source, Destination>()
         .ForMember(dest => dest.Total, opt => opt.MapFrom<CustomResolver>()));
configuration.AssertConfigurationIsValid();

var source = new Source
        {
                Value1 = 5,
                Value2 = 7
        };

var result = mapper.Map<Source, Destination>(source);

result.Total.ShouldEqual(12);
```

Although the destination member (Total) did not have any matching source member, specifying a custom resolver made the configuration valid, as the resolver is now responsible for supplying a value for the destination member.

If we don't care about the source/destination types in our value resolver, or want to reuse them across maps, we can just use "object" as the source/destination types:

```
public class MultBy2Resolver : IValueResolver<object, object, int> {
    public int Resolve(object source, object dest, int destMember, ResolutionContext␣
↪context) {
        return destMember * 2;
    }
}
```

21.1 Custom constructor methods

Because we only supplied the type of the custom resolver to AutoMapper, the mapping engine will use reflection to create an instance of the value resolver.

If we don't want AutoMapper to use reflection to create the instance, we can supply it directly:

```
var configuration = new MapperConfiguration(cfg => cfg.CreateMap<Source, Destination>
↪()
        .ForMember(dest => dest.Total,
```

(continues on next page)

(continued from previous page)

```
                opt => opt.MapFrom(new CustomResolver())
        ));
```

AutoMapper will use that specific object, helpful in scenarios where the resolver might have constructor arguments or need to be constructed by an IoC container.

21.2 The resolved value is mapped to the destination property

Note that the value you return from your resolver is not simply assigned to the destination property. Any map that applies will be used and the result of that mapping will be the final destination property value. Check the execution plan.

21.3 Customizing the source value supplied to the resolver

By default, AutoMapper passes the source object to the resolver. This limits the reusability of resolvers, since the resolver is coupled to the source type. If, however, we supply a common resolver across multiple types, we configure AutoMapper to redirect the source value supplied to the resolver, and also use a different resolver interface so that our resolver can get use of the source/destination members:

```
var configuration = new MapperConfiguration(cfg => {
cfg.CreateMap<Source, Destination>()
    .ForMember(dest => dest.Total,
        opt => opt.MapFrom<CustomResolver, decimal>(src => src.SubTotal));
cfg.CreateMap<OtherSource, OtherDest>()
    .ForMember(dest => dest.OtherTotal,
        opt => opt.MapFrom<CustomResolver, decimal>(src => src.OtherSubTotal));
});

public class CustomResolver : IMemberValueResolver<object, object, decimal, decimal> {
    public decimal Resolve(object source, object destination, decimal sourceMember,␣
↪decimal destinationMember, ResolutionContext context) {
// logic here
    }
}
```

21.4 Passing in key-value to Mapper

When calling map you can pass in extra objects by using key-value and using a custom resolver to get the object from context.

```
mapper.Map<Source, Dest>(src, opt => opt.Items["Foo"] = "Bar");
```

This is how to setup the mapping for this custom resolver

```
cfg.CreateMap<Source, Dest>()
    .ForMember(dest => dest.Foo, opt => opt.MapFrom((src, dest, destMember, context)␣
↪=> context.Items["Foo"]));
```

21.5 ForPath

Similar to ForMember, from 6.1.0 there is ForPath. Check out the tests for examples.

21.6 Resolvers and conditions

For each property mapping, AutoMapper attempts to resolve the destination value **before** evaluating the condition. So it needs to be able to do that without throwing an exception even if the condition will prevent the resulting value from being used.

As an example, here's sample output from BuildExecutionPlan (displayed using ReadableExpressions) for a single property:

```
try
{
        var resolvedValue =
        {
                try
                {
                        return // ... tries to resolve the destination value here
                }
                catch (NullReferenceException)
                {
                        return null;
                }
                catch (ArgumentNullException)
                {
                        return null;
                }
        };

        if (condition.Invoke(src, typeMapDestination, resolvedValue))
        {
                typeMapDestination.WorkStatus = resolvedValue;
        }
}
catch (Exception ex)
{
        throw new AutoMapperMappingException(
                "Error mapping types.",
                ex,
                AutoMapper.TypePair,
                AutoMapper.TypeMap,
                AutoMapper.PropertyMap);
};
```

The default generated code for resolving a property, if you haven't customized the mapping for that member, generally doesn't have any problems. But if you're using custom code to map the property that will crash if the condition isn't met, the mapping will fail despite the condition.

This example code would fail:

```
public class SourceClass
{
        public string Value { get; set; }
```

(continues on next page)

(continued from previous page)

```
}

public class TargetClass
{
        public int ValueLength { get; set; }
}

// ...

var source = new SourceClass { Value = null };
var target = new TargetClass;

CreateMap<SourceClass, TargetClass>()
        .ForMember(d => d.ValueLength, o => o.MapFrom(s => s.Value.Length))
        .ForAllMembers(o => o.Condition((src, dest, value) => value != null));
```

The condition prevents the Value property from being mapped onto the target, but the custom member mapping would fail before that point because it calls Value.Length, and Value is null.

Prevent this by using a PreCondition instead or by ensuring the custom member mapping code can complete safely regardless of conditions:

```
        .ForMember(d => d.ValueLength, o => o.MapFrom(s => s != null ? s.Value.Length␣
↪: 0))
```

CHAPTER 22

Conditional Mapping

AutoMapper allows you to add conditions to properties that must be met before that property will be mapped.

This can be used in situations like the following where we are trying to map from an int to an unsigned int.

```
class Foo{
  public int baz;
}

class Bar {
  public uint baz;
}
```

In the following mapping the property baz will only be mapped if it is greater than or equal to 0 in the source object.

```
var configuration = new MapperConfiguration(cfg => {
  cfg.CreateMap<Foo,Bar>()
    .ForMember(dest => dest.baz, opt => opt.Condition(src => (src.baz >= 0)));
});
```

If you have a resolver, see here for a concrete example.

22.1 Preconditions

Similarly, there is a PreCondition method. The difference is that it runs sooner in the mapping process, before the source value is resolved (think MapFrom). So the precondition is called, then we decide which will be the source of the mapping (resolving), then the condition is called and finally the destination value is assigned.

```
var configuration = new MapperConfiguration(cfg => {
  cfg.CreateMap<Foo,Bar>()
    .ForMember(dest => dest.baz, opt => {
        opt.PreCondition(src => (src.baz >= 0));
        opt.MapFrom(src => {
```

(continues on next page)

(continued from previous page)

```
            // Expensive resolution process that can be avoided with a PreCondition
        });
    });
});
```

You can see the steps yourself.

See here for a concrete example.

CHAPTER 23

Null Substitution

Null substitution allows you to supply an alternate value for a destination member if the source value is null anywhere along the member chain. This means that instead of mapping from null, it will map from the value you supply.

```
var config = new MapperConfiguration(cfg => cfg.CreateMap<Source, Dest>()
    .ForMember(destination => destination.Value, opt => opt.NullSubstitute("Other␣
↪Value")));

var source = new Source { Value = null };
var mapper = config.CreateMapper();
var dest = mapper.Map<Source, Dest>(source);

dest.Value.ShouldEqual("Other Value");

source.Value = "Not null";

dest = mapper.Map<Source, Dest>(source);

dest.Value.ShouldEqual("Not null");
```

The substitute is assumed to be of the source member type, and will go through any mapping/conversion after to the destination type.

CHAPTER 24

Value Converters

Value converters are a cross between Type Converters and Value Resolvers. Type converters are globally scoped, so that any time you map from type `Foo` to type `Bar` in any mapping, the type converter will be used. Value converters are scoped to a single map, and receive the source and destination objects to resolve to a value to map to the destination member. Optionally value converters can receive the source member as well.

In simplified syntax:

- Type converter = `Func<TSource, TDestination, TDestination>`
- Value resolver = `Func<TSource, TDestination, TDestinationMember>`
- Member value resolver = `Func<TSource, TDestination, TSourceMember, TDestinationMember>`
- Value converter = `Func<TSourceMember, TDestinationMember>`

To configure a value converter, use at the member level:

```
public class CurrencyFormatter : IValueConverter<decimal, string> {
    public string Convert(decimal source, ResolutionContext context)
        => source.ToString("c");
}

var configuration = new MapperConfiguration(cfg => {
   cfg.CreateMap<Order, OrderDto>()
       .ForMember(d => d.Amount, opt => opt.ConvertUsing(new CurrencyFormatter()));
   cfg.CreateMap<OrderLineItem, OrderLineItemDto>()
       .ForMember(d => d.Total, opt => opt.ConvertUsing(new CurrencyFormatter()));
});
```

You can customize the source member when the source member name does not match:

```
public class CurrencyFormatter : IValueConverter<decimal, string> {
    public string Convert(decimal source, ResolutionContext context)
        => source.ToString("c");
}
```

(continues on next page)

(continued from previous page)

```
var configuration = new MapperConfiguration(cfg => {
   cfg.CreateMap<Order, OrderDto>()
       .ForMember(d => d.Amount, opt => opt.ConvertUsing(new CurrencyFormatter(), src␣
↪=> src.OrderAmount));
   cfg.CreateMap<OrderLineItem, OrderLineItemDto>()
       .ForMember(d => d.Total, opt => opt.ConvertUsing(new CurrencyFormatter(), src␣
↪=> src.LITotal));
});
```

If you need the value converters instantiated by the service locator, you can specify the type instead:

```
public class CurrencyFormatter : IValueConverter<decimal, string> {
    public string Convert(decimal source, ResolutionContext context)
        => source.ToString("c");
}

var configuration = new MapperConfiguration(cfg => {
   cfg.CreateMap<Order, OrderDto>()
       .ForMember(d => d.Amount, opt => opt.ConvertUsing<CurrencyFormatter, decimal>
↪());
   cfg.CreateMap<OrderLineItem, OrderLineItemDto>()
       .ForMember(d => d.Total, opt => opt.ConvertUsing<CurrencyFormatter, decimal>
↪());
});
```

If you do not know the types or member names at runtime, use the various overloads that accept `System.Type` and `string`-based members:

```
public class CurrencyFormatter : IValueConverter<decimal, string> {
    public string Convert(decimal source, ResolutionContext context)
        => source.ToString("c");
}

var configuration = new MapperConfiguration(cfg => {
   cfg.CreateMap(typeof(Order), typeof(OrderDto))
       .ForMember("Amount", opt => opt.ConvertUsing(new CurrencyFormatter(),
↪"OrderAmount"));
   cfg.CreateMap(typeof(OrderLineItem), typeof(OrderLineItemDto))
       .ForMember("Total", opt => opt.ConvertUsing(new CurrencyFormatter(), "LITotal
↪"));
});
```

Value converters are only used for in-memory mapping execution. They will not work for `ProjectTo`.

CHAPTER 25

Value Transformers

Value transformers apply an additional transformation to a single type. Before assigning the value, AutoMapper will check to see if the value to be set has any value transformations associated, and will apply them before setting.

You can create value transformers at several different levels:

- Globally
- Profile
- Map
- Member

```
var configuration = new MapperConfiguration(cfg => {
    cfg.ValueTransformers.Add<string>(val => val + "!!!");
});

var source = new Source { Value = "Hello" };
var dest = mapper.Map<Dest>(source);

dest.Value.ShouldBe("Hello!!!");
```

CHAPTER 26

Before and After Map Action

Occasionally, you might need to perform custom logic before or after a map occurs. These should be a rarity, as it's more obvious to do this work outside of AutoMapper. You can create global before/after map actions:

```
var configuration = new MapperConfiguration(cfg => {
  cfg.CreateMap<Source, Dest>()
    .BeforeMap((src, dest) => src.Value = src.Value + 10)
    .AfterMap((src, dest) => dest.Name = "John");
});
```

Or you can create before/after map callbacks during mapping:

```
int i = 10;
mapper.Map<Source, Dest>(src, opt => {
    opt.BeforeMap((src, dest) => src.Value = src.Value + i);
    opt.AfterMap((src, dest) => dest.Name = HttpContext.Current.Identity.Name);
});
```

The latter configuration is helpful when you need contextual information fed into before/after map actions.

26.1 Using `IMappingAction`

You can encapsulate Before and After Map Actions into small reusable classes. Those classes need to implement the `IMappingAction<in TSource, in TDestination>` interface.

Using the previous example, here is an encapsulation of naming some objects "John":

```
public class NameMeJohnAction : IMappingAction<SomePersonObject,␣
↪SomeOtherPersonObject>
{
    public void Process(SomePersonObject source, SomeOtherPersonObject destination,␣
↪ResolutionContext context)
    {
```

(continues on next page)

(continued from previous page)

```
        destination.Name = "John";
    }
}

var configuration = new MapperConfiguration(cfg => {
  cfg.CreateMap<SomePersonObject, SomeOtherPersonObject>()
    .AfterMap<NameMeJohnAction>();
});
```

26.1.1 Asp.Net Core and `AutoMapper.Extensions.Microsoft.DependencyInjection`

If you are using Asp.Net Core and the `AutoMapper.Extensions.Microsoft.DependencyInjection` package, this is also a good way of using Dependency Injection. You can't inject dependencies into `Profile` classes, but you can do it in `IMappingAction` implementations.

The following example shows how to connect an `IMappingAction` accessing the current `HttpContext` to a `Profile` after map action, leveraging Dependency Injection:

```
public class SetTraceIdentifierAction : IMappingAction<SomeModel, SomeOtherModel>
{
    private readonly IHttpContextAccessor _httpContextAccessor;

    public SetTraceIdentifierAction(IHttpContextAccessor httpContextAccessor)
    {
        _httpContextAccessor = httpContextAccessor ?? throw new␣
↪ArgumentNullException(nameof(httpContextAccessor));
    }

    public void Process(SomeModel source, SomeOtherModel destination,␣
↪ResolutionContext context)
    {
        destination.TraceIdentifier = _httpContextAccessor.HttpContext.
↪TraceIdentifier;
    }
}

public class SomeProfile : Profile
{
    public SomeProfile()
    {
        CreateMap<SomeModel, SomeOtherModel>()
            .AfterMap<SetTraceIdentifierAction>();
    }
}
```

Everything is connected together by:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddAutoMapper(typeof(Startup).Assembly);
    }
```

(continues on next page)

(continued from previous page)

```
    //..
}
```

See `AutoMapper.Extensions.Microsoft.DependencyInjection` *for more info.*

CHAPTER 27

API Changes

Starting with version 9.0, you can find out what changed in the public API from the last major version release. From the releases page you can reach the source code for that release and the version of ApiCompatBaseline.txt in that tree will tell you what changed. A major version release is compared with the previous major version release (so 9.0.0 with 8.0.0) and a minor version release with the current major version release (so 9.1.1 with 9.0.0).

CHAPTER 28

10.0 Upgrade Guide

Release notes.

28.1 All collections are mapped by default, even if they have no setter

You'll have to explicitly ignore those you don't want mapped. See also this.

28.2 Matching constructor parameters will be mapped from the source, even if they are optional

You can always rename things or use an explicit `MapFrom`.

28.3 `Context.Mapper.Map` overloads that receive a context were removed

Not needed, because the context is passed by default, so you can change the context instance you already have.

28.4 `UseDestinationValue` is now inherited by default

You can override that with `DoNotUseDestinationValue`.

28.5 `AllowNull` allows you to override per member `AllowNullDestinationValues` and `AllowNullCollections`

This used to be ignored for `Map`. Now it's consistent with `ProjectTo`.

28.6 The `ResolutionContext` no longer has a public constructor

You can test the entire `Map` operation.

28.7 Mapping from `dynamic` in .NET 4.6.1

Due to a refactoring of `IMapper`, you might need to add a cast to `object` when mapping from `dynamic`.

28.8 Source validation

Only simple source members expressions are considered, `MapFrom(d => d.Member, s => s.SourceMember)`.

28.9 `MaxDepth`

When reaching `MaxDepth`, destination collections are null/empty, they used to contain `null` values.

28.10 String based `MapFrom`-s are reversed now, also applies to attribute mapping

You can always not use it and explicitly create the reverse map. Or ignore the reversed member.

28.11 `ReverseMap` will also reverse the naming conventions

You can always not use it and explicitly create the reverse map.

CHAPTER 29

9.0 Upgrade Guide

29.1 The static API was removed

Switch to the instance based API, preferably using dependency injection. See here and here.

29.2 AutoMapper no longer creates maps automatically (CreateMissingTypeMaps and conventions)

You will need to explicitly configure maps, manually or using reflection. Also consider attribute mapping.

CHAPTER 30

8.1.1 Upgrade Guide

The purpose of this release is to allow you to upgrade to 9.0 gradually.

30.1 AutoMapper no longer creates maps automatically by default

`CreateMissingTypeMaps` was deprecated and its default value changed to `false`. If you were relying on this, your app will no longer work by default.

If you're not interested in upgrading to 9.0, where dynamic mapping was removed, you should stick with 8.1.

Otherwise you can port your app gradually to 9.0 by creating the needed maps. Setting `CreateMissingTypeMaps` to `false` will get you the 9.0 behavior and setting it to `true` will revert to the 8.1 behavior.

CHAPTER 31

8.0 Upgrade Guide

31.1 ProjectUsing

The `ProjectUsing` method consolidated with `ConvertUsing`:

```
// IMappingExpression

// Old
void ConvertUsing(Func<TSource, TDestination> mappingFunction);
void ProjectUsing(Expression<Func<TSource, TDestination>> mappingExpression);

// New
void ConvertUsing(Expression<Func<TSource, TDestination>> mappingExpression);
```

To migrate, replace all usages of `ProjectUsing` with `ConvertUsing`.

The `ConvertUsing` expression-based method will be used for both in-memory mapping and LINQ projections. You cannot have separate configuration for in-memory vs. LINQ projections.

31.1.1 Existing `ConvertUsing` usages

The change from `Func` to `Expression` may break some existing usages. Namely:

- `ConvertUsing` using lambda statements, method groups, or delegates
- Dual configuration of `ProjectUsing` and `ConvertUsing`

For the first case, you may either:

- Convert to a lambda expression
- Move to the `Func`-based overloads

The `Func`-based overloads accept more parameters, so you may have to add the parameters to your delegates.

31.1.2 Motivation

Simplify overloads, and to make it clear that you cannot have separate configuration for LINQ projections vs. in-memory mapping.

31.2 ConstructProjectionUsing

The `ConstructProjectionUsing` method consolidated with `ConstructUsing`:

```
// IMappingExpression<TSource, TDestination>

// Old
IMappingExpression<TSource, TDestination> ConstructUsing(Func<TSource, TDestination>␣
↪ctor);
IMappingExpression<TSource, TDestination> ConstructUsing(Func<TSource,␣
↪ResolutionContext, TDestination> ctor);
IMappingExpression<TSource, TDestination> ConstructProjectionUsing(Expression<Func
↪<TSource, TDestination>> ctorExpression);

// New
IMappingExpression<TSource, TDestination> ConstructUsing(Expression<Func<TSource,␣
↪TDestination>> ctor);
IMappingExpression<TSource, TDestination> ConstructUsing(Func<TSource,␣
↪ResolutionContext, TDestination> ctor);

// IMappingExpression

// Old
IMappingExpression ConstructUsing(Func<object, object> ctor);
IMappingExpression ConstructUsing(Func<object, ResolutionContext, object> ctor);
IMappingExpression ConstructProjectionUsing(LambdaExpression ctorExpression);

// New
IMappingExpression ConstructUsing(Expression<Func<object, object>> ctor);
IMappingExpression ConstructUsing(Func<object, ResolutionContext, object> ctor);
```

To migrate, replace all usages of `ConstructProjectionUsing` with `ConstructUsing`.

The `ConstructUsing` expression-based method will be used for both in-memory mapping and LINQ projections. You cannot have separate configuration for in-memory vs. LINQ projections.

31.2.1 Existing `ConstructUsing` usages

The change from `Func` to `Expression` may break some existing usages. Namely:

- `ConstructUsing` using lambda statements, method groups, or delegates
- Dual configuration of `ConstructProjectionUsing` and `ConstructUsing`

For the first case, you may either:

- Convert to a lambda expression
- Move to the `Func`-based overload

The `Func`-based overload accepts more parameters, so you may have to add the parameters to your delegates.

31.2.2 Motivation

Simplify overloads, and to make it clear that you cannot have separate configuration for LINQ projections vs. in-memory mapping.

31.3 ResolveUsing

The `ResolveUsing` method consolidated with `MapFrom`:

```
// IMappingExpression

// Old
void ResolveUsing(Func<TSource, TDestination> mappingFunction);
void ResolveUsing(Func<TSource, TDestination, TDestination> mappingFunction);
void ResolveUsing<TResult>(Func<TSource, TDestination, TMember, TResult>␣
↪mappingFunction);
// Many, many overloads
void MapFrom(Expression<Func<TSource, TDestination>> mapExpression);

// New
void MapFrom(Expression<Func<TSource, TDestination>> mappingExpression);
void MapFrom<TResult>(Func<TSource, TDestination, TResult> mappingFunction);
void MapFrom<TResult>(Func<TSource, TDestination, TMember, TResult> mappingFunction);
```

To migrate, replace all usages of `ResolveUsing` with `MapFrom`.

The `MapFrom` expression-based method will be used for both in-memory mapping and LINQ projections. You cannot have separate configuration for in-memory vs. LINQ projections.

31.3.1 Existing `ResolveUsing` usages

The change from `Func` to `Expression` may break some existing usages. Namely:

- `ResolveUsing` using lambda statements, method groups, or delegates
- Dual configuration of `ResolveUsing` and `MapFrom`

For the first case, you may either:

- Convert to a lambda expression
- Move to the `Func`-based overloads

The `Func`-based overloads accept more parameters, so you may have to add the parameters to your delegates.

31.3.2 Motivation

Simplify overloads, and to make it clear that you cannot have separate configuration for LINQ projections vs. in-memory mapping.

31.4 UseValue

Underneath the covers, `UseValue` called `MapFrom`. `UseValue` consolidated with `MapFrom`.

To migrate, replace all usages of `UseValue` with `MapFrom`:

```
// Old
cfg.CreateMap<Source, Dest>()
    .ForMember(dest => dest.Date, opt => opt.UseValue(DateTime.Now));

// New
cfg.CreateMap<Source, Dest>()
    .ForMember(dest => dest.Date, opt => opt.MapFrom(src => DateTime.Now));
```

This can be simplified to a global find and replace of `UseValue(` with `MapFrom(src =>`.

31.4.1 Motivation

To make the underlying configuration more explicit. Historically, `MapFrom` only allowed mapping from an individual source member. This restriction went away with 5.0, so there is no longer a need for additional redundant configuration options originally meant to work around this restriction.

31.5 ForSourceMember Ignore

`ISourceMemberConfigurationExpression.Ignore` was renamed to `DoNotValidate` to avoid confusion. It only applies when validating source members, with `MemberList.Source`. It does not affect the mapping itself or validation for the default case, `MemberList.Destination`. To migrate, replace all usages of `Ignore` with `DoNotValidate`:

```
// Old
cfg.CreateMap<Source, Dest>()
    .ForSourceMember(source => source.Date, opt => opt.Ignore());

// New
cfg.CreateMap<Source, Dest>()
    .ForSourceMember(source => source.Date, opt => opt.DoNotValidate());
```

31.6 Generic maps validation

Generic maps are now validated. The generic map itself is validated at configuration time for the non generic members, so AssertConfigurationIsValid should catch errors for those. And the closed generic map will be validated when it's used, possibly at runtime. If you don't care about those errors, you need to override them.

CHAPTER 32

5.0 Upgrade Guide

32.1 Initialization

You now must use `MapperConfiguration` to initialize AutoMapper.

If you have a lot of `Mapper.CreateMap` calls everywhere, move those to a Profile.

For examples see here.

32.2 Profiles

Instead of overriding a Configure method, you configure directly via the constructor:

```
public class MappingProfile : Profile {
    public MappingProfile() {
        CreateMap<Foo, Bar>();
        RecognizePrefix("m_");
    }
}
```

32.3 IgnoreAllNonExisting extension

A popular Stack Overflow post introduced the idea of ignoring all non-existing members on the destination type. It used things that don't exist anymore in the configuration API. This functionality is really only intended for configuration validation.

In 5.0, you can use ReverseMap or CreateMap passing in the MemberList enum to validate against the source members (or no members). Any place you have this IgnoreAllNonExisting extension, use the CreateMap overload that validates against the source or no members:

```
cfg.CreateMap<ProductDto, Product>(MemberList.None);
```

32.4 Resolution Context things

ResolutionContext used to capture a lot of information, source and destination values, along with a hierarchical parent model. For source/destination values, all of the interfaces (value resolvers and type converters) along with config options now include the source/destination values, and if applicable, source/destination members.

If you're trying to access some parent object in your model, you will need to add those relationships to your models and access them through those relationships, and not through AutoMapper's hierarchy. The ResolutionContext was pared down for both performance and sanity reasons.

32.5 Value resolvers

The signature of a value resolver has changed to allow access to the source/destination models. Additionally, the base class is gone in favor of interfaces. For value resolvers that do not have a member redirection, the interface is now:

```
public interface IValueResolver<in TSource, in TDestination, TDestMember>
{
    TDestMember Resolve(TSource source, TDestination destination, TDestMember␣
↪destMember, ResolutionContext context);
}
```

You have access now to the source model, destination model, and destination member this resolver is configured against.

If you are using a ResolveUsing and passing in the `FromMember` configuration, this is now a new resolver interface:

```
public interface IMemberValueResolver<in TSource, in TDestination, in TSourceMember,␣
↪TDestMember>
{
    TDestMember Resolve(TSource source, TDestination destination, TSourceMember␣
↪sourceMember, TDestMember destMember, ResolutionContext context);
}
```

This is now configured directly as `ForMember(dest => dest.Foo, opt => opt.ResolveUsing<MyCustomResolver, string>(src => src.Bar)`

32.6 Type converters

The base class for a type converter is now gone in favor of a single interface that accepts the source and destination objects and returns the destination object:

```
public interface ITypeConverter<in TSource, TDestination>
{
    TDestination Convert(TSource source, TDestination destination, ResolutionContext␣
↪context);
}
```

32.7 Circular references

Previously, AutoMapper could handle circular references by keeping track of what was mapped, and on every mapping, check a local hashtable of source/destination objects to see if the item was already mapped. It turns out this tracking is very expensive, and you need to opt-in using PreserveReferences for circular maps to work. Alternatively, you can configure MaxDepth:

```
// Self-referential mapping
cfg.CreateMap<Category, CategoryDto>().MaxDepth(3);

// Circular references between users and groups
cfg.CreateMap<User, UserDto>().PreserveReferences();
```

Starting from 6.1.0 PreserveReferences is set automatically at config time whenever the recursion can be detected statically. If that doesn't happen in your case, open an issue with a full repro and we'll look into it.

32.8 UseDestinationValue

UseDestinationValue tells AutoMapper not to create a new object for some member, but to use the existing property of the destination object. It used to be true by default. Consider whether this applies to your case. Check recent issues.

```
cfg.CreateMap<Source, Destination>()
   .ForMember(d => d.Child, opt => opt.UseDestinationValue());
```

CHAPTER 33

Examples

The source code contains unit tests for all of the features listed above. Use the GitHub search to find relevant examples.

CHAPTER 34

Housekeeping

The latest builds can be found at NuGet

The dev builds can be found at MyGet

The discussion group is hosted on Google Groups