AutoMapper Documentation

Release

Jimmy Bogard

Nov 09, 2017

User Documentation

1 Getting Started Guide **3**
- 1.1 What is AutoMapper? . . . 3
- 1.2 Why use AutoMapper? . . . 3
- 1.3 How do I use AutoMapper? . . . 3
- 1.4 Where do I configure AutoMapper? . . . 4
- 1.5 How do I test my mappings? . . . 4
- 1.6 Can I see a demo? . . . 4

2 Upgrade Guide **5**
- 2.1 Initialization . . . 5
- 2.2 Profiles . . . 5
- 2.3 IgnoreAllNonExisting extension . . . 5
- 2.4 Resolution Context things . . . 6
- 2.5 Value resolvers . . . 6
- 2.6 Type converters . . . 6
- 2.7 Circular references . . . 7
- 2.8 UseDestinationValue . . . 7

3 Conventions **9**
- 3.1 Conditional Object Mapper . . . 9
- 3.2 Member Configuration . . . 9
- 3.3 Expand-ability . . . 10
- 3.4 Multiple Configurations . . . 10
- 3.5 Profiles . . . 10

4 Flattening **13**

5 Reverse Mapping and Unflattening **17**
- 5.1 Customizing reverse mapping . . . 18

6 Projection **19**

7 Configuration Validation **21**
- 7.1 Overriding configuration errors . . . 22
- 7.2 Selecting members to validate . . . 22

8 Inline Mapping **23**

8.1 Inline configuration . . . 23
8.2 Inline validation . . . 24
8.3 Disabling inline maps . . . 24

9 Lists and Arrays **25**
9.1 Polymorphic element types in collections . . . 26

10 Nested Mappings **29**

11 Custom Type Converters **31**
11.1 System Type Converters . . . 33

12 Custom Value Resolvers **35**
12.1 Custom constructor methods . . . 36
12.2 Customizing the source value supplied to the resolver . . . 37
12.3 Passing in key-value to Mapper . . . 37
12.4 ForPath . . . 37

13 Value Transformers **39**

14 Null Substitution **41**

15 Before and After Map Action **43**

16 Dependency Injection **45**
16.1 Gotchas . . . 45
16.2 ASP.NET Core . . . 45
16.3 Ninject . . . 45
16.4 Simple Injector . . . 46

17 Mapping Inheritance **49**
17.1 Runtime polymorphism . . . 49
17.2 Specifying inheritance in derived classes . . . 50

18 Queryable Extensions **53**
18.1 Preventing lazy loading/SELECT N+1 problems . . . 54
18.2 Custom projection . . . 55
18.3 Custom Type Conversion . . . 55
18.4 Custom destination type constructors . . . 55
18.5 String conversion . . . 55
18.6 Explicit expansion . . . 56
18.7 Aggregations . . . 56
18.8 Parameterization . . . 56
18.9 Supported mapping options . . . 57

19 Configuration **59**
19.1 Profile Instances . . . 59
19.2 Naming Conventions . . . 60
19.3 Replacing characters . . . 61
19.4 Recognizing pre/postfixes . . . 61
19.5 Global property/field filtering . . . 62
19.6 Configuring visibility . . . 62
19.7 Configuration compilation . . . 63

20 Conditional Mapping **65**
20.1 Preconditions . . . 65

21 Open Generics 67

22 Understanding Your Mappings 69

23 Samples 71

24 Housekeeping 73

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 4.0
- .NET 4.5.2+
- .NET Platform Standard 1.1 & 1.3

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` or the static `Mapper` instance and CreateMap. You only need one `MapperConfiguration` instance typically per AppDomain and should be instantiated during startup. Alternatively, you can just use `Mapper.Initialize` (more examples of initial setup see in Static-and-Instance-API.

```
Mapper.Initialize(cfg => cfg.CreateMap<Order, OrderDto>());
//or
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, use the static or instance Mapper methods, depending on static or instance initialization:

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

If you're using the static Mapper method, 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 call Mapper.Initialize to configure the type maps.

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();
```

1.6 Can I see a demo?

Check out the dnrTV episode on AutoMapper.

CHAPTER 2

Upgrade Guide

2.1 Initialization

You now must use either `Mapper.Initialize` or `new MapperConfiguration()` to initialize AutoMapper. If you prefer to keep the static usage, use `Mapper.Initialize`.

If you have a lot of `Mapper.CreateMap` calls everywhere, move those to a Profile, or into `Mapper.Initialize`, called once at startup.

For examples see here.

2.2 Profiles

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

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

2.3 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);
```

2.4 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.

2.5 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)`

2.6 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);
}
```

2.7 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.

2.8 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 3

Conventions

3.1 Conditional Object Mapper

Conditional Object Mappers make new type maps based on conditional between the source and the destination type.

```
var config = new MapperConfiguration(cfg => {
    cfg.AddConditionalObjectMapper().Where((s, d) => s.Name == d.Name + "Dto");
});
```

3.2 Member Configuration

Member Configuration is like Configuration but you can have complete control on what is and isn't used.

```
var config = new MapperConfiguration(cfg => { cfg.AddMemberConfiguration(); });
```

AddMemberConfiguration() starts off with a blank slate. Everything that applies in Configuration by default is gone to start with.

3.2.1 Naming Conventions

`AddMemberConfiguration().AddMember<NameSplitMember>()` gets you default naming convention functionality.

Can overwrite the source and destination member naming conventions by passing a lambda through the parameter. SourceExtentionMethods can also be set here.

If you don't set anything AutoMapper will use DefaultMember, which will only check using the Name of the property.

PS: If you don't set this Flattening of objects will be disabled.

3.2.2 Replacing characters

`AddMemberConfiguration().AddName<ReplaceName>(_ => _.AddReplace("Ä", "A").AddReplace("í", "i"));`

3.2.3 Recognizing pre/postfixes

`AddMemberConfiguration().AddName<PrePostfixName>(_ => _.AddStrings(p => p.Prefixes, "Get", "get").AddStrings(p => p.DestinationPostfixes, "Set"));`

3.2.4 Attribute Support

`AddMemberConfiguration().AddName<SourceToDestinationNameMapperAttributesMember>();` * Currently is always on

Looks for instances of SourceToDestinationMapperAttribute for Properties/Fields and calls user defined isMatch function to find member matches.

MapToAttribute is one of them which will match the property based on name provided.

```
public class Foo
{
    [MapTo("SourceOfBar")]
    public int Bar { get; set; }
}
```

3.2.5 Getting AutoMapper Defaults

`AddMemberConfiguration().AddMember<NameSplitMember>().AddName<PrePostfixName>(_ => _.AddStrings(p => p.Prefixes, "Get"))`

Is the default values set by Configuration if you don't use AddMemberConfiguration().

3.3 Expand-ability

Each of the AddName and AddMember types are based on an interface ISourceToDestinationNameMapper and IChildMemberConfiguration. You can make your own classes off the interface and configure their properties through the lambda statement argument, so you can have fine tune control over how AutoMapper resolves property maps.

3.4 Multiple Configurations

Each configuration is its own set of rules that all must pass in order to say a property is mapped. If you make multiple configurations they are completely separate from one another.

3.5 Profiles

These can be added to Profile as well as the ConfigurationStore.

Each Profiles rules are separate from one another, and won't share any conditions. If a map is generated from AddConditionalObjectMapper of one profile, the AddMemberConfigurations of only that profile can be used to resolve the property maps.

3.5.1 Example

Shown below is two profiles for making conventions for transferring to and from a Data Transfer Object. Each one is isolated to one way of the mappings, and the rules are explicitly stated for each side.

```
// Flattens with NameSplitMember
// Only applies to types that have same name with destination ending with Dto
// Only applies Dto post fixes to the source properties
public class ToDTO : Profile
{
    protected override void Configure()
    {
        AddMemberConfiguration().AddMember<NameSplitMember>().AddName<PrePostfixName>(
                _ => _.AddStrings(p => p.Postfixes, "Dto"));
        AddConditionalObjectMapper().Where((s, d) => s.Name == d.Name + "Dto");
    }
}

// Doesn't Flatten Objects
// Only applies to types that have same name with source ending with Dto
// Only applies Dto post fixes to the destination properties
public class FromDTO : Profile
{
    protected override void Configure()
    {
        AddMemberConfiguration().AddName<PrePostfixName>(
                _ => _.AddStrings(p => p.DestinationPostfixes, "Dto"));
        AddConditionalObjectMapper().Where((s, d) => d.Name == s.Name + "Dto");
    }
}
```

CHAPTER 4

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
{
    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

Mapper.Initialize(cfg => cfg.CreateMap<Order, OrderDto>());

// Perform mapping

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.

CHAPTER 5

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:

```
Mapper.Initialize(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
};
```

```
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`.

5.1 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());
```

CHAPTER 6

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
Mapper.Initialize(cfg =>
  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 7

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:

```
Mapper.Initialize(cfg =>
  cfg.CreateMap<Source, Destination>());

Mapper.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.

7.1 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.

```
Mapper.Initialize(cfg =>
  cfg.CreateMap<Source, Destination>()
    .ForMember(dest => dest.SomeValuefff, opt => opt.Ignore())
);
```

7.2 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:

```
Mapper.Initialize(cfg =>
  cfg.CreateMap<Source, Destination>(MemberList.Source);
  cfg.CreateMap<Source2, Destination2>(MemberList.None);
);
```

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

CHAPTER 8

Inline Mapping

AutoMapper creates type maps on the fly (new in 6.2.0). When you call `Mapper.Map` for the first time, AutoMapper will create the type map configuration and compile the mapping plan. Subsequent mapping calls will use the compiled map.

8.1 Inline configuration

To configure an inline map, use the mapping options:

```
var source = new Source();

var dest = Mapper.Map<Source, Dest>(source, opt => opt.ConfigureMap().ForMember(dest␣
↪=> dest.Value, m => m.MapFrom(src => src.Value + 10)));
```

You can use local functions to make the configuration a little easier to read:

```
var source = new Source();

void ConfigureMap(IMappingOperationOptions<Source, Dest> opt) {
    opt.ConfigureMap()
       .ForMember(dest => dest.Value, m => m.MapFrom(src => src.Value + 10))
};

var dest = Mapper.Map<Source, Dest>(source, ConfigureMap);
```

You can use closures in this inline map as well to capture and use runtime values in your configuration:

```
int valueToAdd = 10;
var source = new Source();

void ConfigureMap(IMappingOperationOptions<Source, Dest> opt) {
    opt.ConfigureMap()
       .ForMember(dest => dest.Value, m => m.MapFrom(src => src.Value + valueToAdd))
};

var dest = Mapper.Map<Source, Dest>(source, ConfigureMap);
```

8.2 Inline validation

The first time the map is used, AutoMapper validates the map using the default validation configuration (destination members must all be mapped). Subsequent map calls skip mapping validation. This ensures you can safely map your objects.

You can configure the member list used to validate, to validate the source, destination, or no members to validate per map:

```
var source = new Source();

var dest = Mapper.Map<Source, Dest>(source, opt => opt.ConfigureMap(MemberList.None);
```

You can also turn off inline map validation altogether (not recommended unless you're explicitly testing all of your maps):

```
Mapper.Initialize(cfg => cfg.ValidateInlineMaps = false);
```

8.3 Disabling inline maps

To turn off inline mapping:

```
Mapper.Initialize(cfg => cfg.CreateMissingTypeMaps = false);
```

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:

```
Mapper.Initialize(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 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; }
}

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:

```
Mapper.Initialize(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

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>();
    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 11

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()
{
    Mapper.Initialize(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>();
    });
    Mapper.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.

11.1 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 12

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)
```

```
    {
        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:

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

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

```
Mapper.Initialize(cfg =>
   cfg.CreateMap<Source, Destination>()
     .ForMember(dest => dest.Total, opt => opt.ResolveUsing<CustomResolver>());
Mapper.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;
    }
}
```

12.1 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:

```
Mapper.Initialize(cfg => cfg.CreateMap<Source, Destination>()
    .ForMember(dest => dest.Total,
        opt => opt.ResolveUsing(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.

12.2 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:

```
Mapper.Initialize(cfg => {
cfg.CreateMap<Source, Destination>()
    .ForMember(dest => dest.Total,
        opt => opt.ResolveUsing<CustomResolver, decimal>(src => src.SubTotal));
cfg.CreateMap<OtherSource, OtherDest>()
    .ForMember(dest => dest.OtherTotal,
        opt => opt.ResolveUsing<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
    }
}
```

12.3 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

```
Mapper.CreateMap<Source, Dest>()
    .ForMember(d => d.Foo, opt => opt.ResolveUsing((src, dest, destMember, res) =>␣
↪res.Context.Options.Items["Foo"]));
```

12.4 ForPath

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

CHAPTER 13

Value Transformers

Value transformers apply an additional transormation 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

```
Mapper.Initialize(cfg => {
    cfg.ValueTransformers.Add<string>(val + "!!!");
});

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

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

CHAPTER 14

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 15

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:

```
Mapper.Initialize(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.

CHAPTER 16

Dependency Injection

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

```
Mapper.Initialize(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(Mapper.Configuration, childContainer.GetInstance);

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

16.1 Gotchas

Using DI is effectively mutually exclusive with using the IQueryable.ProjectTo extension method. Use `IEnumerable.Select(_mapper.Map<DestinationType>).ToList()` instead.

16.2 ASP.NET Core

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

16.3 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.AddProfiles(GetType().Assembly);
        });

        return config;
    }
}
```

16.4 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);

        var profiles = typeof(SomeProfile).Assembly.GetTypes()
            .Where(t => typeof(Profile).IsAssignableFrom(t))
            .ToList();

        mce.AddProfiles(profiles);

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

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

        return m;
    }
}

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

```
}
```

CHAPTER 17

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.

17.1 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 { }

Mapper.Initialize(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.

17.2 Specifying inheritance in derived classes

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

```
Mapper.Initialize(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>();
});
```

17.2.1 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
Mapper.Initialize(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.Equals("google", 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.

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

CHAPTER 18

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

You can use the Queryable Extensions like so:

```
Mapper.Initialize(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>().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.

18.1 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,
        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.

18.2 Custom projection

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

```
Mapper.Initialize(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.

18.3 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 ProjectUsing method:

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

`ProjectUsing` is slightly more limited than `ConvertUsing` as only what is allowed in an Expression and the underlying LINQ provider will work.

18.4 Custom destination type constructors

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

```
cfg.CreateMap<Source, Dest>()
    .ConstructProjectionUsing(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.

18.5 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>().ToList();
orders[0].OrderType.ShouldEqual("Online");
```

18.6 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>(
    dest => dest.Customer,
    dest => dest.LineItems);
// or string-based
dbContext.Orders.ProjectTo<OrderDto>(
    null,
    "Customer",
    "LineItems");
```

For more information, see the tests.

18.7 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'.

18.8 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.

18.9 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
- Ignore
- UseValue
- NullSubstitute

Not supported:

- Condition
- DoNotUseDestinationValue
- SetMappingOrder
- UseDestinationValue
- ResolveUsing
- Before/AfterMap
- Custom resolvers
- Custom type 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 19

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.

Alternatively, you can use the static Mapper instance to initialize AutoMapper:

```
Mapper.Initialize(cfg => {
    cfg.CreateMap<Foo, Bar>();
    cfg.AddProfile<FooProfile>();
});
```

19.1 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)
    }
}
```

```
// 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.

19.1.1 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.AddProfiles(myAssembly);
});
// ... or static approach:
Mapper.Initialize(cfg => cfg.AddProfiles(myAssembly));

// Can also use assembly names:
Mapper.Initialize(cfg =>
    cfg.AddProfiles(new [] {
        "Foo.UI",
        "Foo.Core"
    });
);

// Or marker types for assemblies:
Mapper.Initialize(cfg =>
    cfg.AddProfiles(new [] {
        typeof(HomeController),
        typeof(Entity)
    });
);
```

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

19.2 Naming Conventions

You can set the source and destination naming conventions

```
Mapper.Initialize(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
  }
}
```

19.3 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:

```
Mapper.Initialize(c =>
{
    c.ReplaceMemberName("Ä", "A");
    c.ReplaceMemberName("í", "i");
    c.ReplaceMemberName("Airlina", "Airline");
});
```

19.4 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; }
}
Mapper.Initialize(cfg => {
    cfg.RecognizePrefixes("frm");
    cfg.CreateMap<Source, Dest>();
});
Mapper.AssertConfigurationIsValid();
```

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

```
Mapper.Initialize(cfg => {
    cfg.ClearPrefixes();
    cfg.RecognizePrefixes("tmp");
});
```

19.5 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:

```
Mapper.Initialize(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);
});
```

19.6 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 :

```
Mapper.Initialize(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.

19.7 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:

```
Mapper.Initialize(cfg => {});
Mapper.Configuration.CompileMappings();
```

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

CHAPTER 20

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.

```
Mapper.Initialize(cfg => {
  cfg.CreateMap<Foo,Bar>()
    .ForMember(dest => dest.baz, opt => opt.Condition(src => (src.baz >= 0)));
});
```

20.1 Preconditions

Similarly, there is a precondition. The difference is that it runs sooner in the mapping process, before the source value is resolved (think MapFrom or ResolveUsing). 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. You can see the steps yourself.

CHAPTER 21

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
Mapper.Initialize(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:

```
Mapper.Initialize(cfg =>
   cfg.CreateMap(typeof(Source<>), typeof(Destination<>)).
↪ConvertUsing(typeof(Converter<>)));
```

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

```
Mapper.Initialize(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<,>`.

CHAPTER 22

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.

```
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 23

Samples

The source code contains unit tests and samples for all of the features listed above. To view the samples, browse the source code.

CHAPTER 24

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