18

AutoMapper

This chapter covers

* Understanding AutoMapper
* Configuring AutoMapper
* Testing conventions
* Applying formatters to eliminate duplicative code
* Reducing markup to presentation only
* Ridding views of complexity

The open source library AutoMapper is a conventional object to object mapper. It takes source objects of one type and maps them to destination objects of another type. This is useful in many contexts, but we'll use it to map from a domain model to the model objects our views display - the presentation model. We call it conventional because the way it works does not depend on configuring each type's member's mapping, but relies on naming patterns and sensible defaults. You can check out the code and read more documentation at its website: [http://automapper.codeplex.com](http://automapper.codeplex.com/).

Given a source type and a destination type, AutoMapper will simply assign values from source members, properties and methods, to corresponding members on the destination. It does this automatically based on member names. Let's look at a couple quick examples, just to get started. In the first example, we want to map from an object named Source into an object named Destination. Listing 18.1 shows a source class and a destination class. The names match up, so AutoMapper will simply map the value (and call ToString() on the Source.Number property).

Listing 18.1 An introductory mapping

public class Source

{

public int Number { get; set; }

}

public class Destination

{

public string Number { get; set; }

}

[Test]

public void Demonstration1()

{

Mapper.CreateMap<Source, Destination>(); #1

var source = new Source {Number = 3}; #2

Destination destination =

Mapper.Map<Source, Destination>(source); #3

Console.WriteLine(destination.Number); #4

}

#1 Creating the mapping with AutoMapper

#2 Instantiating the Source object

#3 Performing the map with AutoMapper

#4 The output is "3"

The output of the test in listing 18.1 is the string "3". AutoMapper just looks at the names and when they match, it makes the assignment.

In reality our objects are rarely this simple. They are usually object hierarchies. AutoMapper can flatten graphs of objects, projecting the hierarchy to a new shape. In listing 18.2 we show AutoMapper flattening a very simple hierarchy.

Listing 18.2 Flattening a simple hierarchy

public class Source

{

public Child Child { get; set; }

}

public class Child

{

public int Number { get; set; }

}

public class Destination

{

public string ChildNumber { get; set; } #1

}

[Test]

public void Demonstration1()

{

Mapper.CreateMap<Source, Destination>();

var source = new Source

{

Child = new Child{ Number = 3}

};

Destination desintation =

Mapper.Map<Source, Destination>(source);

Console.WriteLine(desintation.ChildNumber); #2

}

#1 AutoMapper works with naming conventions

#2 The output is "3"

Again, AutoMapper relies on the name of the destination property to figure out where the source value will come from. Because our destination property is named ChildNumber, (3) AutoMapper will map from Child.Number.

AutoMapper can do much more than simple value assignment and flattening. Developers can configure special formatters and instruct AutoMapper to do other actions during the mapping process. Before we dive into AutoMapper, let's see what life was like before it and how we arrived at the decision to use object mapping.

*18.1 Life before AutoMapper*

Imagine a view that renders information about a customer. We discussed in chapter 2 that some trivial applications may choose to use persistent, domain model objects to as the data source for views. In the markup in listing 18.3, that's exactly the scenario in which we find ourselves.

Listing 18.3 A portion of Customer.aspx

<%@ Page Language="C#"

Inherits="System.Web.Mvc.ViewPage<Customer>" %> |#1

<%@ Import Namespace="Core.Model"%> |#1

<h2>Customer: <%= Html.Encode(Model.Name.First + " " + |#2

Model.Name.Middle + " " + Model.Name.Last) %></h2> |#2

<div class="customerdetails">

<p>Status: <%= Html.Encode(Model.Status) %></p> #3

<p>Total Amount Paid: $ |#4

<%= Html.Encode(Model.GetTotalAmountPaid()) %></p> |#4

<p>Address: <%= Html.Encode(Model.ShippingAddress.Line1) %>,

<%= Html.Encode(Model.ShippingAddress.Line2) %>,

<%= Html.Encode(Model.ShippingAddress.City) %>,

<%= Html.Encode(Model.ShippingAddress.State.DisplayName) %> #5

<%= Html.Encode(Model.ShippingAddress.Zip) %></p>

</div>

#1 Directly using the domain object

#2 Formatting complex components

#3 Everything must be encoded

#4 Apply standard formatting manually

#5 Domain objects interrogated deeply

This is complex markup - overly complex for the simple display it's rendering. We see common formatting rules, like applying the dollar sign to decimal values. There's that very suspicious name formatting (2) that will clearly look wrong if there's a missing middle name. We are manually and repeatedly applying encoding rules (3).

When the page is actually displayed, there's not only the danger of the screen not looking right, but it may not render at all. What if the ShippingAddress is null? We'll see a nasty null reference exception in the yellow screen of death that accompanies major ASP.NET errors. All these problems are caused by the view directly depending on the domain model, by the user interface knowing too much about the core logic of the software.

We know, from our examples in Chapter 2 and the previous section, that in most scenarios it's best to design a custom model for consumption by the view. Translating from the domain model - projecting it - to the presentation model is a straightforward programming task. Take the value from the source object and copy it to the right place on the destination object. Mix in some carefully applied formatting and flattening code, and our projection is complete. We can easily test this logic. An example of a hand-rolled mapper is shown in listing 18.4.

Listing 18.4 Mapping objects by hand

public class CustomerInfoMapper

{

public CustomerInfo MapFrom(Customer customer) #1

{

return new CustomerInfo #2

{

Id = customer.Id, |#3

Name = new NameFormatter() |#3

.Format(customer.Name), |#3

ShippingAddress = new AddressFormatter() |#3

.Format(customer.ShippingAddress), |#3

Status = customer.Status ?? string.Empty, |#3

TotalAmountPaid = customer.GetTotalAmountPaid() |#3

.ToString("c") |#3

};

}

}

#1 Accepts source type, returns destination

#2 Creating the destination presentation model

#3 Performing manual mapping

This class is testable and separates the view from the complexity of our domain model. It allows the view a change to work with the data as it's intended to be displayed. Listing 18.5 shows our view, updated to work with CustomerInfo instead of Customer.

Listing 18.5 Working with the manually mapped presentation model

<h2>Customer: <%= Html.Encode(Model.Name) %></h2> #1

<div class="customerdetails">

<p>Status: <%= Html.Encode(Model.Status) %></p>

<p>Total Amount Paid: <%= Html.Encode(Model.TotalAmountPaid) %></p> #2

<p>Address: <%= Model.ShippingAddress %></p>

</div>

#1 We're still encoding everywhere

#2 The shape fits the view however

This is much better. The markup addresses more of the what and where and less of the how. We're still encoding every property (1) - there are global rules that must be applied. While the manual mapping scenario we've shown in listing 18.2 is a marked improvement over rendering the domain model directly, it's still extremely tedious to write. It is tedious to write, expensive to maintain, error prone, and brittle. We can test it, but on a system featuring dozens of screens, this testing effort can bog down a project.

Now that you understand the problem AutoMapper solves, you will start to use it for some of these mapping tasks. AutoMapper allows us to forgo the manual mapping code, and gives us a hook to enable custom global or specific formatting rules. Instead of the imperative code we wrote in listing 18.4, we can declare the mapping and have AutoMapper perform the mapping behavior for us.

Declarative programming versus imperative programming

Imperative programming is traditional code we usually write. It expresses actions as a series of lines of code indicating logical flow and assignment. Imperative code consists of complex algorithms and logical statements that direct an exact sequence of operations. On the other hand, declarative programming just specifies what is to be done, not how to do it. Declarative code is simple - it's just a statement, not an instruction set. The canonical example in declarative programming is regular expressions. Imagine reproducing the text search represented by a complex regular expression with imperative if statements and loops. Avoiding that burden - and trusting good tools - is one path to rapid construction and hassle-free maintenance.

A sample AutoMapper configuration declaration is shown in listing 18.6.

Listing 18.6 A quick look at AutoMapper configuration code

CreateMap<Customer, CustomerInfo>()

.ForMember(x => x.ShippingAddress, opt =>

{

opt.AddFormatter<AddressFormatter>();

opt.SkipFormatter<HtmlEncoderFormatter>();

});

We'll return to listing 18.6 and cover AutoMapper basics in the next section.

*18.2 AutoMapper basics*

AutoMapper must be initialized and configured. And because AutoMapper relies on soft associations via naming conventions, it's important that developers have a way to test that the configuration is valid. We'll cover all these aspects and more in this section.

*18.2.1 - Initialization*

Listing 18.7 AutoMapper initialization

public class AutoMapperConfiguration

{

public static void Configure() #1

{

Mapper.Initialize(x => #2

x.AddProfile<ExampleProfile>()); #3

}

}

#1 A static entry point

#2 Initializing AutoMapper

#3 Adding a configuration profile

Listing 18.7 shows a sample class that initializes AutoMapper, which should be initialized before it's used, when the application starts. For ASP.NET MVC 2 applications, one place this could happen is Global.asax.cs. We'll cover profiles (3) next.

18.2.2 - Profiles

Profiles are the main vehicle for configuring AutoMapper. A profile is a collection of type mapping definitions. AutoMapper profiles are classes that derive from its Profile class. They also contain rules that apply to all maps defined in that profile. Profiles are effective for grouping mappings by context. An application may have one profile for mapping from the domain model to a presentation model and another profile for another purpose. Here, in listing 18.8, is a rich profile with several configuration directives.

Listing 18.8 Creating a sample profile

public class ExampleProfile : Profile #1

{

protected override string ProfileName

{

get { return "ViewModel"; }

}

protected override void Configure()

{

AddFormatter<HtmlEncoderFormatter>(); #2

ForSourceType<Name>().AddFormatter<NameFormatter>(); #3

ForSourceType<decimal>() |#4

.AddFormatExpression(context => |#4

((decimal)context.SourceValue).ToString("c")); |#4

CreateMap<Customer, CustomerInfo>() |#5

.ForMember(x => x.ShippingAddress, opt => |#5

{ |#5

opt.AddFormatter<AddressFormatter>(); |#5

opt.SkipFormatter<HtmlEncoderFormatter>(); |#5

}); |#5

}

}

#1 Derive from Profile

#2 Apply a global formatter

#3 Apply formatter for source type

#4 In-line formatting for source type

Let's investigate the profile piece by piece. First, each profile must derive from Profile and choose a unique ProfileName (1). The Configure method contains the configuration declarations. The first directive is AddFormatter<HtmlEncoderFormatter>(). This is a global instruction to AutoMapper, telling it to apply HTML encoding to every destination member. A second formatting directive tells AutoMapper to, whenever it's mapping from a Name object, use the NameFormatter (we will investigate NameFormatter in depth later in this chapter). There's another directive providing a special formatting expression AutoMapper should use when it's attempting to map from decimal objects. This expression will use the standard formatting string to display decimals as currency.

Finally, there is the CreateMap directive, which tells AutoMapper to plan to map from Customer to CustomerInfo. The ForMember call tells AutoMapper to apply the AddressFormatter but skip the HtmlEncoderFormatter when mapping to the ShippingAddress destination property. The rest of the CustomerInfo properties aren't specified, because they're mapped conventionally.

18.2.3 - Sanity checking

A reliance on convention is a double-edged sword. On one hand, it helpfully eliminates the developer's obligation to specify each member's mapping. But there is a danger if a property is renamed. If a source member is renamed it could no longer correspond to the appropriate destination member and the convention would be broken. Developers need fast feedback when changes like this happen. It's not acceptable to experience a run-time error. AutoMapper provides a method that will ensure its configuration is valid, checking that each destination member is mapped to a source member by convention or configuration. Listing 18.9 shows a profile that won't work: someone made a typographical error.

Listing 18.9 Examining a potentially dangerous typo

public class Destination

{

public string Name { get; set; }

public string Typo { get; set; } #1

}

public class Source

{

public string Name { get; set; }

public int Number { get; set; }

}

public class BrokenProfile : Profile

{

protected override void Configure()

{

CreateMap<Source, Destination>();

}

}

#1 This should be named "Number"

To protect against typos like this we can run, as part of our automated test suite, a special helper test, shown in listing 18.9.

Listing 18.9 AutoMapperConfigurationTester

[TestFixture]

public class AutoMapperConfigurationTester

{

[Test]

public void Should\_map\_everything()

{

AutoMapperConfiguration.Configure(); #1

Mapper.AssertConfigurationIsValid(); #2

}

}

#1 Configure AutoMapper normally

#2 Testing the mapping configuration

When this test is run against our broken profile from listing 18.9 we'll get a helpful message indicating that the Typo property is not mapped.

18.2.4 - Reducing repetitive formatting code

Previously in this chapter we've mentioned applying special formatters to member mappings. Each of these formatters are implementations of IValueFormatter, an AutoMapper interface that defines the contract between AutoMapper and our custom formatting code. Listing 18.10 shows this interface.

Listing 18.10 Examining the IValueFormatter interface

public interface IValueFormatter

{

string FormatValue(ResolutionContext context);

}

Our customer formatting implementation will accept a ResolutionContext, which supplies the original value and other metadata and outputs a string result. To make it easier on client developers, a simple base class can be implemented. Listing 18.11 shows BaseFormatter, which pulls the source value out of the context and checks for null values.

Listing 18.11 Implementing IValueFormatter on the BaseFormatter class

public abstract class BaseFormatter<T> : IValueFormatter

{

public string FormatValue(ResolutionContext context)

{

if (context.SourceValue == null)

return null;

if (!(context.SourceValue is T)) #1

{

object value = context.SourceValue;

return value == null ?

string.Empty : value.ToString();

}

return FormatValueCore((T) context.SourceValue); #2

}

protected abstract string FormatValueCore(T value); #3

}

#1 Try ToString if wrong type

#2 Return abstract method

#3 Derivations implement strongly-typed method

Deriving from BaseFormatter makes writing a custom formatter straightforward. All we need to do is implement its abstract FormatValueCore method, which receives the strongly-typed source value. AutoMapper will catch any null reference exceptions in formatters or in regular mapping. It just returns an empty string or the default value. Listing 18.12 shows the NameFormatter discussed earlier in section 18.1 of this chapter.

Listing 18.12 Deriving NameFormatter to handle combining properties

public class NameFormatter : BaseFormatter<Name>

{

protected override string FormatValueCore(Name value)

{

var sb = new StringBuilder(); #1

if (!string.IsNullOrEmpty(value.First)) |#2

{ |#2

sb.Append(value.First); |#2

} |#2

if (!string.IsNullOrEmpty(value.Middle))

{

sb.Append(" " + value.Middle);

}

if (!string.IsNullOrEmpty(value.Last))

{

sb.Append(" " + value.Last);

}

if (value.Suffix != null)

{

sb.Append(", " + value.Suffix.DisplayName);

}

return sb.ToString();

}

}

#1 Using StringBuilder to craft the output

#2 Basic formatting logic

Harnessing AutoMapper allows the developer to write this code once and apply it in many places with just a declaration. When configured like the profile in listing 18.8, this formatter will be applied to all source members of type Name.

18.2.5 - Another look at our views

With our configuration complete, our markup is focused only on layout and tedious logic there is replaced. Listing 18.13 shows the resulting view.

Listing 18.13 The final view markup

<h2>Customer: <%= Model.Name %></h2>

<div class="customerdetails">

<p>Status: <%= Model.Status %></p>

<p>Total Amount Paid: <%= Model.TotalAmountPaid %></p>

<p>Address: <%= Model.ShippingAddress %></p>

</div>

18.3 Summary

In this chapter we understood how views can quickly become unmanageable with logical checks and formatting that is best handled elsewhere. When we first encountered the friction of these views, we manually mapped custom presentation models. This worked well, but is tedious and error prone. AutoMapper is an object-to-object mapper. It maps values from one object to another according to its configuration. We saw how to initialize and configure AutoMapper, how to follow the conventions, and learned the basics of leveraging AutoMapper hooks to globally apply formatting.

In the next chapter, you will learn how to keep your controllers lightweight and under control. By striving to reduce duplication and eliminate developer friction, we'll craft small and targeted controller actions.