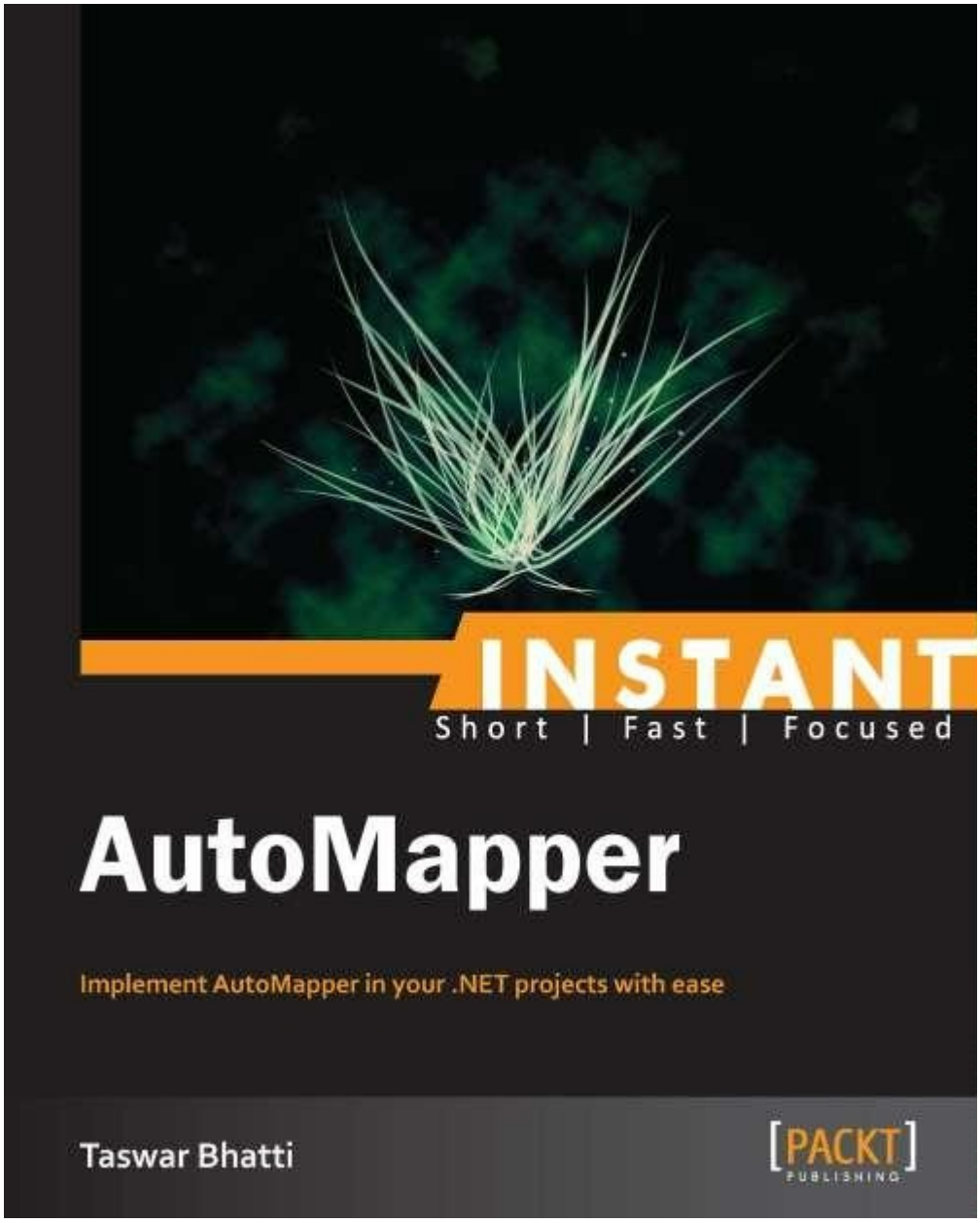

INSTANT
Short | Fast | Focused
AutoMapper
Implement AutoMapper in your .NET projects with ease
Taswar Bhatti
[PACKT]
PUBLISHING

Table of Contents

Instant AutoMapper

Instant AutoMapper

First published: July 2013

Production Reference: 1240713

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78328-205-0

www.packtpub.com

Credits

Author

Taswar Bhatti

Reviewer

Prashant Brall

Acquisition Editor

Usha Iyer

Content Commissioning Editor

Shaon Basu

Technical Editor

Dylan Fernandes

Dipika Gaonkar

Copy Editor

Gladson Monteiro

Project Coordinator

Sherin Padayatty

Proofreader

Stephen Copestake

Production Coordinator

Melwyn D'sa

Cover Work

Melwyn D'sa

About the Author

Taswar Bhatti was born and raised in Hong Kong. He is a web architect, team leader, and also a computer geek, with a degree in Mathematics and Computing Science from the University of Alberta. He has worked in a wide range of technologies from Unix/Linux to Microsoft stack. He is well versed in developing scalable Internet web architecture from design, implementation, measuring, tuning to fix performance-related issues in front- to back-end systems, and so on. His career includes working in academia, social media, and language translation companies. Although he enjoys many computer languages from statically to dynamically typed, he also enjoys functional languages such as, Scala, Java, C#, Ruby, F#, and JavaScript. He is also fluent in many spoken languages: Cantonese, English, Turkish, Urdu, and Hindi. In his spare time, he speaks in developer groups, code camps, tinkers around with new and shiny frameworks, and blogs. He currently resides in Ottawa, Canada, with his lovely wife and two children.

About the Reviewer

Prashant Brall is an independent consultant based in one of the most beautiful capitals of the world: Canberra. He has over 17 years of experience in application development, which includes 4 years working in the USA for Fortune 500 companies and major financial companies. As a software professional, Prashant loves crafting software and enjoys writing about his experiences on his blog at https://prashantbrall.wordpress.com.

In his leisure time, he enjoys watching movies and playing the piano and guitar.

A big thank you to my wife, Jhumur, for her love and support and for being my best friend. I would also like to thank my parents for encouraging me and showing me the difference between right and wrong throughout my childhood.

www.packtpub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt Publishing offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt Publishing books and eBooks.

Free Access for Packt Publishing account holders

If you have an account with Packt at www.packtpub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Chapter 1. Instant AutoMapper

Welcome to Instant AutoMapper. This book has been created to provide you with all the information that you need to get up-to-speed with AutoMapper. You will learn the basics of AutoMapper and all the core techniques are covered in detail; tricks of the trade and strategies are outlined when it comes to using AutoMapper.

This document contains the following sections:

So, what is AutoMapper? reveals what AutoMapper actually is, what you can do with it, and why it's so great.

Installation helps you learn how to download and install AutoMapper with minimal fuss and set it up in two easy steps.

Quick start – creating a sample project will show you how to perform one of the core tasks of AutoMapper: creating courses. The section also covers disallowing mapping on fields, wiring IoC containers, and unit-testing your mapping using AutoMapper.

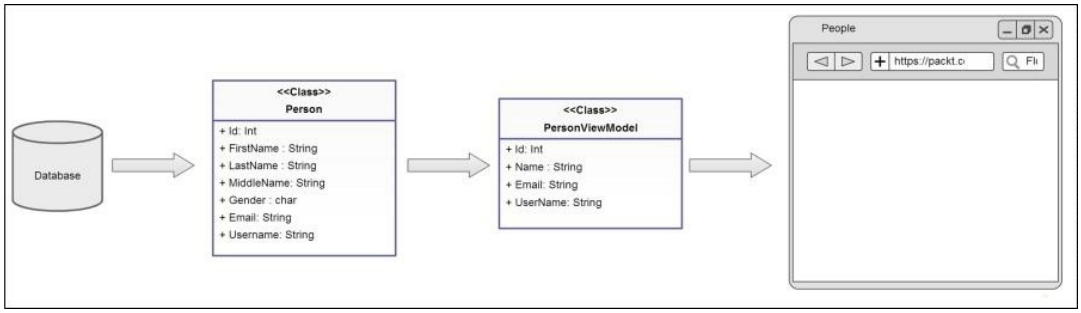
Top 12 features you need to know about covers fundamental features of AutoMapper such as flattening, null substitution, projection, nested mapping, lists and arrays, mapping inheritance, and so on.

People and places you should get to know provides you with many useful links to the project page and forums, as well as a number of helpful articles, tutorials, blogs, and the Twitter feeds of AutoMapper super-contributors.

So, what is AutoMapper?

AutoMapper is an open source and conventional object-to-object mapper. It eliminates the need of writing boring and tedious code for mapping one object type to another. AutoMapper conventions also alleviate the need for extra configurations to map between objects. Under the hood, AutoMapper uses reflection to build the object for us.

AutoMapper helps in transforming one object type into another while also providing mapping testing. In an application, AutoMapper usually fits in boundary layers, such as between database types, domain layers, and user interface/web UI layers. By segregating the models between layers, each model layer would only affect itself rather than propagating change throughout the layers, as shown in the following screenshot:


Database
<<Class>>
Person
+ Id: Int
+ FirstName : String
+ LastName : String
+ MiddleName: String
+ Gender : char
+ Email: String
+ Username: String
<<Class>>
PersonViewModel
+ Id: Int
+ Name : String
+ Email: String
+ UserName: String
People
https://packt.c
Fi

Installation

In two easy steps, you can be ready to start developing with AutoMapper.

Step 1 – what do I need?

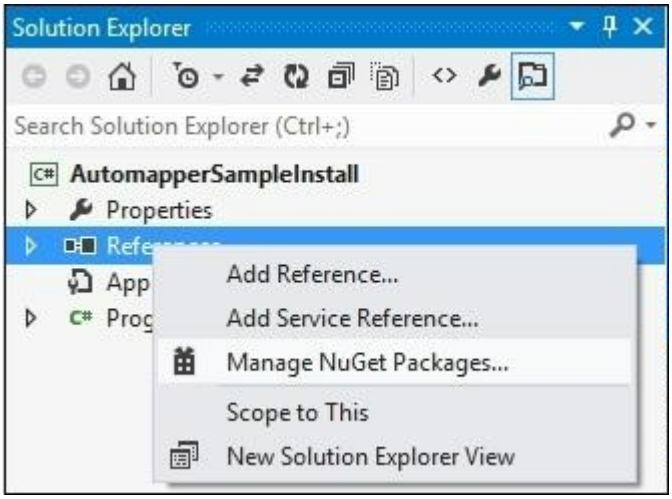
Before you install AutoMapper, you will need to check that you have all the required elements as follows:

- Visual Studio 2010 or 2012
- NuGet Package Management

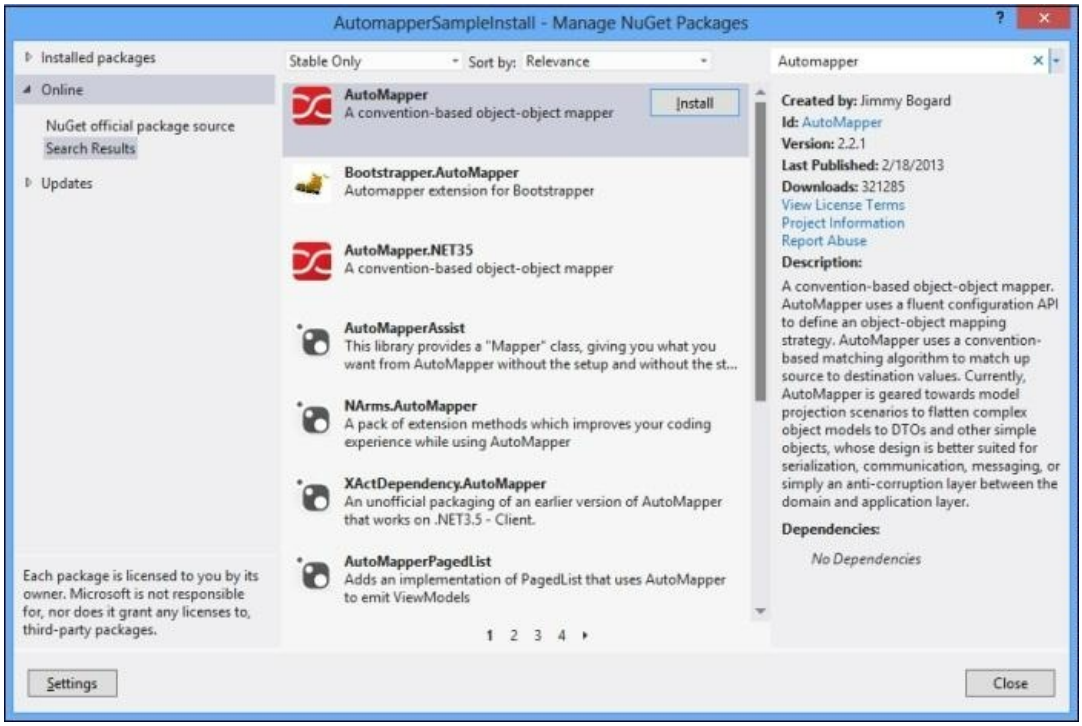
Step 2 – downloading and installing AutoMapper

The easiest way to download and install AutoMapper is to use the Package Management Console. Go to the Visual Studio menu, **View** | **Other Windows** | **Package Management Console**. Inside the console, type `PM > Install-Package AutoMapper`.

Alternatively, you can also use a graphical interface for installing AutoMapper. Right-click on **References** in your **Solution Explorer** window and click on **Manage NuGet Packages...** as shown in the following screenshot:



In the new NuGet Package Management window, select **Online** and search for **AutoMapper** as shown in the following screenshot:



Click on **Install** and AutoMapper will be installed in your current project.

And that's it

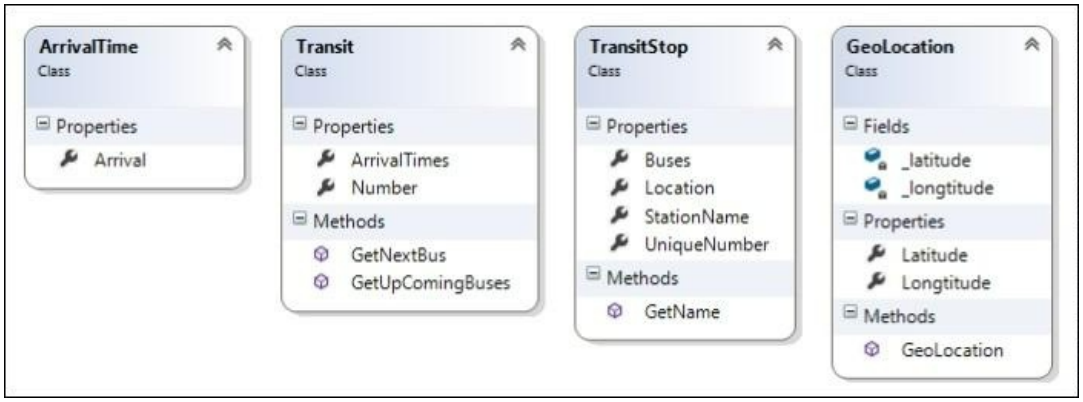
By this point, you will have a working installation of AutoMapper and are free to play around and discover more about it.

Quick start – creating a sample project

In this section, we will create a sample project based on a transit system. We will use AutoMapper to help us transform our domain objects to `ViewModel` objects that we will use in the presentation layer.

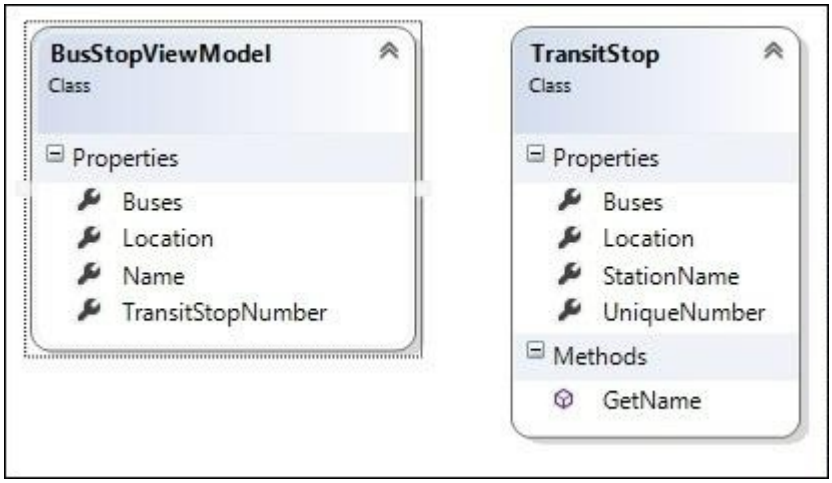
Before we begin, we need to explain the business overview of our domain objects used in the transit system. There are four main domain objects that we will work with. `TransitStop` will be the main entry point to our system; this represents a bus stop in the real world. The `TransitStop` domain object provides a commuter `GeoLocation` to give the latitude and longitude of the location, which would allow a commuter to easily locate nearby bus stops in a map if they desire to. A commuter can also use its `UniqueNumber` to find out when the next bus will be arriving at the bus stop. Each `TransitStop` has a `UniqueNumber` identifier so that a commuter can easily locate the stop without a GPS device.

From there, we have the `Transit` object, which represents the vehicle that will be arriving at the stop. There could be multiples vehicles arriving, and everyone has an arrival time object representing the arrival time of the bus/transit to the stop for the commuter to go from one destination to the other. The following screenshot shows us the overall model:



Step 1 – defining the ViewModel object

Our first view model that we will define and use in our presentation layer will be the `BusStopViewModel` object, which will be mapped from `TransitStop`. The following screenshot represents the same:



The preceding screenshot shows the class diagram of our `BusStopViewModel` and `TransitStop` domain objects. The code for the same is defined as follows:

```
public class BusStopViewModel
{
        public string Name { get; set; }
        public GeoLocationViewModel Location
{ get; set; }
        public int TransitStopNumber { get; set; }
}
```

Tip

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you have purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

Compare that to our domain object, as in the following code:

```
public class TransitStop
{
  public string StationName { get; set; }
  public int UniqueNumber { get; set; }
  public GeoLocation Location { get; set; }

  public IEnumerable<Transit> Buses { get; set; }

  public string GetName()
  {
```

```
    return string.Format("{0} - {1}",
UniqueNumber, StationName);
  }
}
```

Our `ViewModel` object will most likely have a different structure than our domain object, since there are properties or data that do not pertain to the presentation layer but are required for our domain object.

> **Note**
>
> Note that the naming convention in BusStopViewModel is mostly identical; in doing so, we will have AutoMapper automatically map the data for us.

Step 2 – creating the mapping

In our mapping code, we will use a repository to get the data. Our repository will be an in-memory data that we generate; however, in real-world situations, one would use a WCF web service, WebApi, SQL, or a NoSQL solution as in the following code:

```
public class TransitStopRepository
{
  private readonly IList<TransitStop> _data = new
List<TransitStop>();

  /// <summary>
  /// Initializes a new instance of the <see
cref="DataRepository"/> class.
  /// </summary>
  public TransitStopRepository()
  {
    Populate();
  }

  /// <summary>
  /// Populates this instance.
  /// </summary>
  private void Populate()
  {
    var transitStop = new TransitStop {
                    StationName = "Tsim Sha Tsui",
                    UniqueNumber = 123,
                    Location = new
GeoLocation(114.171575, 22.293314),
                    Buses = GenerateTSTBuses()
                };

    _data.Add(transitStop);
```

```
}
```

We are using an `IList` that contains all our transit data. The `Populate` method populates the list with all the information needed. It acts like a database containing all the `TransitStop` information. The following code interprets the same:

```
/// <summary>
/// Generates the TST KMB buses.
/// </summary>
/// <returns></returns>
private IEnumerable<Transit> GenerateTSTBuses()
{
  var buses = new List<Transit>
  {
    new Transit
    {
      Number = "23A",
      ArrivalTimes =
GenerateTimeTable(DateTime.Now),
      TransitColor = "Green"
    },
    new Transit
  {
      Number = "23B",
      ArrivalTimes =
GenerateTimeTable(DateTime.Now),
      TransitColor = "Yellow"
  }
  };
  return buses;
}
```

We use the `GenerateTSTBuses` method to generate two transit routes that will be arriving at the transit stop. It will also generate a timetable for the transits arriving at the transit stop for our in-memory database so that our domain objects will be fully populated when we send a query to them, as in the following code:

```
/// <summary>
/// Generates a timetable with 2 minute apart
arrival time.
/// </summary>
/// <param name="from">From.</param>
/// <param name="incrementMinutes">The
incrementMinutes.</param>
/// <returns></returns>
private IEnumerable<ArrivalTime>
GenerateTimeTable(DateTime from)
{
 //create a list
  var list = new List<ArrivalTime>();
```

```
  var everyXminute = 2;

  //we will always generate 10 transits arriving
to the stop
  for(var i = 0; i < 10; i++)
  {
    var arrivalTime = new ArrivalTime
    {
      Arrival = from.AddMinutes(everyXminutes)
    };

    //double it for next
  everyXminutes *= 2;

  list.Add(arrivalTime);
  }

  return list;
}
```

The `GenerateTimeTable` method generates the `ArrivalTime` object of the transit acting as a timetable for buses; we have hardcoded it to ten transits for our in-memory database, each arriving at an increment of 2 minutes apart.

Now that our repository has data fully populated, we can now use our repository to get fully populated domain objects. The following example uses the `GetByUniqueNumber` method to get a `TransitStop` object from the repository:

```
public static BusStopViewModel GetTransitStop(int
uniqueNumber)
{
  var repo = new TransitStopRepository();
  var tStop =
repo.GetByUniqueNumber(uniqueNumber);
  //create the map from -> to object
  AutoMapper.Mapper.CreateMap<TransitStop,
BusStopViewModel>();
```

We then create the declaration of our mapping of the domain object to `ViewModel` by calling `CreateMap`, where AutoMapper will automatically map the objects that have the same naming convention. The step of creating a map is essential to AutoMapper your doing so tells AutoMapper what to map and what not to map.

Step 3 – mapping the object

The mapping is done by calling the `Map` method in `AutoMapper` and passing the data to map from. In the preceding example, the `tStop` variable

contains data that we will be passing into for AutoMapper. By calling the `Map` method, we are essentially requesting AutoMapper to create another object based on the data we have provided, as illustrated in the following code:

```
  //map the object
  var result = AutoMapper.Mapper.Map<TransitStop,
BusStopViewModel>(tStop);
  return result;
        }
```

In the preceding code, we are requesting `AutoMapper` to map the `TransitStop` object into `BusStopViewModel` object.

Step 4 – not mapping certain data

Let's assume that the `Location` object is a very computing-intense object that requires lots of memory and CPU, or that we wish to use some other means of finding geolocation from a service, such as Google or Bing Maps. We can request AutoMapper not to map the object for us in our `CreateMap` method by using its mapping expression method `ForMember` as follows:

```
AutoMapper.Mapper.CreateMap<TransitStop,
BusStopViewModel>()
.ForMember(dest => dest.Location, opt =>
opt.Ignore());
```

By providing the `Ignore` option in the `ForMember` method, `AutoMapper` will not map the `Location` object.

> **Note**
>
> Note that AutoMapper will use a default value for `Location`; in the preceding case, it will substitute `Location` with a null value.

Step 5 – testing the mapping

In order to test/verify that our mapping is correct, AutoMapper provides us with a method `Assertconfigurationisvalid,` as follows:

```
AutoMapper.Mapper.AssertConfigurationIsValid();
```

By calling the method `AssertConfigurationIsValid` after the `CreateMap` method, it will automatically validate our mapping for us and throw an `AutoMapperConfigurationException,` if invalid. Normally, the place to put

the assertion code would be in your unit test. The following code shows a unit test testing the mapping:

```
[Test]
public void TestBusViewModelMapping()
{
  AutoMapper.Mapper.CreateMap<TransitStop,
BusStopViewModel>()
  .ForMember(dest => dest.Location, opt =>
opt.Ignore());

  AutoMapper.Mapper.AssertConfigurationIsValid();
}
```

Step 6 – using a profile to group mapping

AutoMapper allows you to use configuration profiles to group mapping. The benefit of using a profile is that it removes the individual mapper class and code duplication and provides a separation of concerns; thus, it reduces complexity in your code. A profile class is implemented by inheriting from `AutoMapper.Profile` and overriding its `Configure` method. We can also provide a Name for the profile by overriding the `ProfileName` property. The following code performs the stated task:

```
public class TransitProfile: AutoMapper.Profile
{
  public override string ProfileName
  {
    get {return "TransitViewModelMappings"; }
  }


  protected override void Configure()
  {
    AutoMapper.Mapper.CreateMap<TransitStop,
BusStopViewModel>()
    .ForMember(dest => dest.TransitStopNumber,
opt => opt.MapFrom(src => src.UniqueNumber));

    //create the map for transit to bus view model
    //use the BusUpComing resolver to resolve up
coming bus
    AutoMapper.Mapper.CreateMap<Transit,
BusViewModel>()
    .ForMember(dest => dest.UpComingBus, opt =>
opt.ResolveUsing<BusUpComingResolver>());
   }
}
```

Step 7 – bootstrapping the startup of a profile

The best place to use a profile is during the startup of an application. By moving the `CreateMap` configurations into a startup/bootstrap method, it will significantly improve the performance of mapping.

We will use the following example of a web application, where we can use the `Application_Start` method to wire up our AutoMapper configuration:

```
protected void Application_Start()
{
  AreaRegistration.RegisterAllAreas();
  RegisterGlobalFilters(GlobalFilters.Filters);
  RegisterRoutes(RouteTable.Routes);
  //run all the startup Task
   StartUp.Run();
}

public static class StartUp {
  public static void Run()
  {


//get all the starup task from the IoC container
//using the service locator
    var startupTask =
ServiceLocator.Current.GetAllInstances<IStartupTask>();

    //call the execute method on the startup task
    foreach (var task in startupTask)
    {
      task.Execute();
    }
  }
}

public interface IStartupTask
{
        void Execute();
}

 public class AutoMapperStartupTask: IStartupTask
 {
   public void Execute()
   {
     AutoMapper.Mapper.Initialize(config =>{
 config.AddProfile<TransitProfile>();
   }
}
}
```

In the preceding code we are using a bootstrapping class `StartUp` that has only one method called `Run`. The `Run` method uses an `IoC` container to get all the classes in our assembly that inherits `IStartupTask` and calls the `Execute` method on them. In the real world, one will have multiple classes implementing `IStartupTask` (for example, caching resources, loading data, and so on). In our example, we have created a class called `AutoMapperStartupTask` that implements the interface `IStartupTask` and in its `Execute` method, we are calling the `Initialize` method passing in a lamda function for AutoMapper to load the configuration of `TransitProfile`. This will eliminate the need to call `CreateMap` in our mapping calls.

Top 12 features you need to know about

As you start to use AutoMapper, you will realize that there is a wide variety of things that you can do with it. This section will teach you all about the most commonly performed tasks and most commonly used features as follows:

- Flattening
- Null substitution
- Projection
- Nested mapping
- Lists, arrays, and dictionaries
- Mapping inheritance
- Custom resolvers
- Custom type converters
- Dynamic mapping
- Interface mapping
- Existing object mapping
- Inversion of control

Flattening

AutoMapper will only map object pairs that are explicitly registered by the `CreateMap` method, and to perform the mapping, we need to call the `Map` method.

One of the main reasons for mapping is often to convert a complex object into a simple one. Consider the mapping of a transit domain entity to `BusStopViewModel`, as illustrated in the following code:

```
public class FlatteningMapping
{
public static BusStopViewModel
GetTransitStopWithoutLocation(int uniqueNumber)
   {
     var repo = new TransitStopRepository();
  var transitStop =
repo.GetByUniqueNumber(uniqueNumber);

  //create the map from -> to object, but don't
map the location
  //and map transit stop number from unique number
  AutoMapper.Mapper.CreateMap<TransitStop,
BusStopViewModel>()
  .ForMember(dest => dest.TransitStopNumber, opt
=> opt.MapFrom(src => src.UniqueNumber))
```

```
  .ForMember(dest => dest.Location, opt =>
opt.Ignore());

//map the object
var result = AutoMapper.Mapper.Map<TransitStop,
BusStopViewModel>(transitStop);
   return result;
                }
            }
```

We have created a map by calling the `CreateMap` method and have provided AutoMapper with details of how to map from `UniqueNumber` to `TransitStopNumber` for our `ViewModel`. At the same time, we have requested AutoMapper not to map the location entity. The `Name` property in `BusViewModel` would also match the `GetName` method inside the `Transit` entity as long as we use the naming convention, for example using the same name or using the `Get` method's naming convention for property names. AutoMapper will automatically map without any additional configuration.

Null substitution

AutoMapper allows us to use the extension member of `NullSubsitute` to supply an alternative value for the destination value when the source value is null. Most of the time, we will see null cases handled inside the presentation layer something along the lines of:

```
@if (ViewModel.Name == null){
  <p>No Name</p>
} else {
    @ViewModel.Name
}
```

We can avoid the `if` statement in our presentation layer by using the `NullSubstitution` method in AutoMapper, as illustrated in the following code:

```
public static IEnumerable<BusStopViewModel>
GetAllTransitStopWithNames()
{
  var repo = new TransitStopRepository();
         var transitStop = repo.GetAll();

  //create the map from -> to object, but don't
map the location
  //and map transit stop number from unique number
  //substitute null values with Unknown
  AutoMapper.Mapper.CreateMap<TransitStop,
BusStopViewModel>()
```

```
  .ForMember(dest => dest.TransitStopNumber, opt
=> opt.MapFrom(src => src.UniqueNumber))
  .ForMember(dest => dest.Name, opt =>
opt.NullSubstitute("Unknown"));

  //map the collection
  var result =
AutoMapper.Mapper.Map<IEnumerable<TransitStop>,
IEnumerable<BusStopViewModel>>(transitStop);

  return result;
}
```

From the preceding code snippet, we can see that we have substituted null `Name` of a `TransitStop` with `Unknown`; by doing so, we avoid having conditional statements in our display view code, thus removing complexity.

Projection

Projection helps to transform our objects. For example, we may want to project into `ArrivalTime` of the `Transit` entity and convert it to `ArrivalMinutes` by projecting into the `DateTime.Minute` property. We will need to provide AutoMapper with a custom member configuration by using the `MapFrom` expression as follows:

```
public static BusStopViewModel
GetTransitStopWithArrivalTime(int uniqueNumber)
{
  var repo = new TransitStopRepository();

  var transitStop =
repo.GetByUniqueNumber(uniqueNumber);

  //create the map from -> to object
  //and map transit stop number from unique number
  AutoMapper.Mapper.CreateMap<TransitStop,
BusStopViewModel>()
  .ForMember(dest => dest.TransitStopNumber, opt
=> opt.MapFrom(src => src.UniqueNumber));

  //create the map for transit to bus view model
  //This is an inner object of TransitStop, and
we are defining it
  //for automapper to know how to map the inner
object
  //We will project the next arrival bus into the
bus view model
  //with the minutes of the next arrival
  AutoMapper.Mapper.CreateMap<Transit,
BusViewModel>()
```

```
  .ForMember(dest => dest.NextArrivalInMinutes,
opt => opt.MapFrom(src =>
src.NextBusArrival.Minute));

  //map the object
  var result = AutoMapper.Mapper.Map<TransitStop,
BusStopViewModel>(transitStop);

  return result;
    }
```

In the preceding example, we have provided two mapping definitions using `CreateMap`, one for `TransitStop` to `BusStopViewModel` and another for the inner entity of `Transit` to `BusViewModel`. We then are calling the `MapFrom` function in AutoMapper, which will be evaluated during mapping from our source `Transit` entity. We then invoke the `NextBusArrival` property, and we call its inner property of `DateTime` to request the `Minutes` for our `BusStopViewModel`. By doing so we are requesting AutoMapper to project the mapping to `NextArrivalInMinutes` property.

Nested mapping

In a real-world application, one will have a very complex domain and will have aggregated root objects with hierarchical child entities that would also require mapping. AutoMapper allows us to either flatten the object or to map it altogether to another type of object.

The following example uses the `TransitStop` domain object that has a complex entity nested inside consisting of a `GeoLocation` entity, which has the longitude and latitude information of the bus stop. We will first create a map for `TransitStop` to `BusStopViewModel` and also create a map for its inner child entity `GeoLocation` to `GeoLocationViewModel`, thus allowing and teaching AutoMapper to map the child complex entity to our `GeoLocationViewModel`.

```
public static BusStopViewModel
GetTransitStopWithLocation(int uniqueNumber)
{
var repo = new TransitStopRepository();
       var transitStop =
repo.GetByUniqueNumber(uniqueNumber);

//create the map from -> to object
//and map transit stop number from unique number
AutoMapper.Mapper.CreateMap<TransitStop,
BusStopViewModel>()
.ForMember(dest => dest.TransitStopNumber, opt =>
opt.MapFrom(src => src.UniqueNumber));

//create the map for geolocation
```

```
AutoMapper.Mapper.CreateMap<GeoLocation,
GeoLocationViewModel>();

//map the object
var result = AutoMapper.Mapper.Map<TransitStop,
BusStopViewModel>(transitStop);

return result;
}
```

By using the `CreateMap` method for `GeoLocation` after the `TransitStop` domain entity's `CreateMap` method, we have let AutoMapper know how it will map the nested complex entity. The final result of `BusViewModel` will also contain a child `GeoLocationViewModel` fully populated and mapped. Using nested mapping allows us to have complex entities mapped to a similar hierarchy structure in our `ViewModels`.

> **Note**
>
> Note that the order of the `CreateMap` is not critical; `GeoLocation` could have come before the `TransitStop CreateMap` method. Also, it is not necessary to provide additional information for the `Map` method when we map `TransitStop`.

Lists, arrays, and dictionaries

There is no additional configuration required to map arrays, lists, IEnumerables, or ICollections while using AutoMapper. AutoMapper only requires configuration for element types. In our example of array mapping, we will define two simple classes, namely `StopName` and `StopNameViewModel`, that would only contain a single property, `string Name`:

```
public class StopName
{
/// <summary>
/// Gets or sets the name.
/// </summary>
/// <value>
/// The name.
/// </value>
public string Name { get; set; }
}

public class StopNameViewModel
{
/// <summary>
/// Gets or sets the name.
```

```
    /// </summary>
    /// <value>
    /// The name.
    /// </value>
    public string Name { get; set; }
    }
```

We are now going to populate the `StopName` class in an array and request AutoMapper to map the entities to `StopViewModel`.

```
    public class ArrayDictMapping
{
    /// <summary>
    /// Gets the transit stop.
    /// </summary>
    /// <returns></returns>
    public static StopNameViewModel[]
GetTransitStopName()
    {
          var transitStops = new [] {
        new StopName { Name = "Taksim/Chapulcu" },
        new StopName { Name = "Tsim Sha Shui" },
        new StopName { Name = "Rideau" }
      };

  AutoMapper.Mapper.CreateMap<StopName,
StopNameViewModel>();
  var result = AutoMapper.Mapper.Map<StopName[],
StopNameViewModel[]>(transitStops);

    return result;
    }
```

From the preceding example, we have defined an array to contain all the `StopName` entities, and AutoMapper will automatically map into an array of `StopNameViewModel`. Another example will be the case of a dictionary collection type as follows:

```
/// <summary>
/// Gets the transit stop dict.
/// </summary>
/// <returns></returns>
public static
IDictionary<string,StopNameViewModel>
GetTransitStopDict()
{
    var transitStops = new
Dictionary<string,StopName>  {
          {"Istanbul", new StopName { Name =
"Taksim/Chapulcu" }},
      {"Hong Kong", new StopName { Name = "Tsim
Sha Shui" }},
```

```
        {"Ottawa", new StopName { Name = "Rideau" }}
        };

  AutoMapper.Mapper.CreateMap<StopName,
StopNameViewModel>();
  var result =
AutoMapper.Mapper.Map<Dictionary<string,
StopName>, IDictionary<string,
StopNameViewModel>>(transitStops);

    return result;
  }
```

We have defined a dictionary that contains city names and the `StopName` entity, a string key, and a `StopName` value collection. Again, AutoMapper will automatically map the objects with the matching keys and map the `StopName` entities to `StopNameViewModel` for us.

Mapping inheritance

AutoMapper allows polymorphic mapping and automatically selects the appropriate derived mapping for the class. For our example, we will create a `ParaTransit` class that inherits from the `Transit` entity. The `ParaTransit` entity represents a transportation service well equipped for persons with disabilities, who are unable to use conventional transit services. The `ParaTransit` entity will contain an extra property called `DriverName` with a string data type. This will allow a commuter with disabilities to have a better service by knowing the name of the bus driver who would be arriving to pick them. We will also extend our `ViewModel ParaTransitViewModel` by adding the `DriverName` property inheriting from `BusViewModel`:

```
public class ParaTransit : Transit{
    /// <summary>
    /// Gets or sets the Driver Name.
    /// </summary>
    /// <value>
    /// The Driver Name.
    /// </value>
    public string DriverName { get; set; }
}

public class ParaTransitViewModel : BusViewModel
{
    /// <summary>
    /// Gets or sets the Driver Name.
    /// </summary>
    /// <value>
    /// The Driver Name.
    /// </value>
    public string DriverName { get; set; }
```

```
}
```

We will now create a mapping with an array of `Transit` and `ParaTransit`, and use the `Include` method to let AutoMapper know that, whenever there is a mapping of `Transit`, there may also be a derived type of `ParaTransit` in it:

```
public class InheritMapping
{
    /// <summary>
    /// Gets the transit stop.
    /// </summary>
    /// <returns></returns>
    public static BusViewModel[] GetParaTransit()
    {
        var transitStops = new [] {
        new Transit { Number = "23A" },
        new ParaTransit { DriverName = "
Chapulcu", Number = "34A"},
        new ParaTransit{ DriverName = "Chapulay",
Number="34B" }
      };

    AutoMapper.Mapper.CreateMap<Transit,
BusViewModel>()
       .Include<ParaTransit,
ParaTransitViewModel>();

        AutoMapper.Mapper.CreateMap<ParaTransit,
ParaTransitViewModel>();

    var result =
AutoMapper.Mapper.Map<Transit[],BusViewModel[]>(transitStops);

        return result;
    }
}
```

In doing so, we have now let AutoMapper map the entities to `BusViewModel`, and we can cast back the object to `ParaTransitViewModel` and also have its `DriverName` populated.

Note

Note that `CreateMap` still needs to be called for the `ParaTransit` entity, since one still needs to teach AutoMapper how to map from `ParaTransit` to `ParaTransitViewModel`.

Custom resolvers

There are some cases where we need additional help in resolving data, for example, calculating some value for the data, or formatting it in some form for the presentation layer, or even vice versa from the presentation layer back into our domain. AutoMapper allows us to use `CustomValueResolver` to resolve such situations. By doing so, it eliminates any code clutter we may have in our presentation or domain layer. For example, using the Razor view engine in our web layer, we may have an input form with two input fields `Hour` and `Minutes` along the lines of:

```
Hour: @Html.TextBox("Hour") <br />
Minute: @Html.TextBox("Minutes") <br />
<input type="submit" />
```

> **Note**
>
> The validation of the input would be outside the scope of this book but please do check out JQuery Validation Plug at http://jqueryvalidation.org/.

We can use a `CustomValueResolver` entity to convert the `Hour` and `Minutes` properties into our `ArrivalTime` domain entity.

We will create an `ArrivalTimeViewModel` class that contains only three properties, namely, hours, minutes (using an integer data type), and the arrival time that is a string data type we will use later in our customer type converter example.

```
public class ArrivalTimeViewModel
{
  ///<summary>
  /// Gets or sets the hour
  ///</summary>
  ///<value>
  /// Hour
  ///</value>
  public int Hour { get; set; }

  ///<summary>
  /// Gets or sets the minutes
  ///</summary>
  ///<value>
  /// Minutes
  ///</value>
  public int Minutes { get; set;}

  ///<summary>
```

```
  /// Gets or sets the Arrival
  ///</summary>
  ///<value>
  /// Arrival
  ///</value>
  public string Arrival { get; set; }
}
```

Our `CustomValueResolver` object will need to inherit from `ValueResolver` and pass in the generic types of `ArrivalTimeViewModel` and `DateTime`. In our resolver, we will need to override the method `ResolveCore` to return a `DateTime` structure, which will convert the data from the `ArrivalTimeViewModel` object into a `DateTime` structure as follows:

```
public class ArrivalTimeResolver :
ValueResolver<ArrivalTimeViewModel, DateTime>
{
  protected override string
ResolveCore(ArrivalTimeViewModel source)
  {
  return DateTime.Now.Date.Add(new
TimeSpan(source.Hour, source.Minute, 0));
  }
}
```

From here, we will let AutoMapper know that it should use the resolver when it tries to map the `Hour` and `Minute` properties into the `ArrivalTime` entity, again by using the `CreateMap` method. The resolver acts as a plugin for AutoMapper, allowing one to chain many more resolvers if one wishes to. This task can be performed by the following code:

```
public static void
UpdateArrivalWithResolver(ArrivalTimeViewModel
model)
    {
//create the map for ArrivalTimeViewModel to
ArrivalTime
//use the ArrivalTime resolver to resolve time

 AutoMapper.Mapper.CreateMap<ArrivalTimeViewModel,
ArrivalTime>()
      .ForMember(dest => dest.Arrival, opt =>
opt.ResolveUsing<ArrivalTimeResolver>());


//map the object
var result =
AutoMapper.Mapper.Map<ArrivalTimeViewModel,
ArrivalTime>(model);

var repo = new TransitStopRepository();
repo.Update(result);
```

```
}
```

By using the `ResolveUsing<ArrivalTimeResolver>()` method to chain into `ForMember`, AutoMapper will use reflection to build the object for us.

> **Note**
>
> Note that one can also provide the object if they do not wish to use reflection by creating the instance for AutoMapper as follows:
>
> ```
> opt =>
> opt.ResolveUsing<ArrivalTimeResolver>().ConstructedBy(()
> => new ArrivalTimeResolver());
> ```

Custom type converters

There are situations where one has to convert a certain type to another type, for example, a string into a `DateTime` object. AutoMapper allows us to provide global customer type converters when we wish to convert certain objects to strongly typed objects. The main difference between custom type converters and custom value resolvers is that custom type converters are used globally, whereas custom value resolvers are used only for specific mappings.

For our example, we will be modifying our `BusViewModel` domain object to contain three properties, namely, color, which is an `enum` type that contains only four colors, and `NextArrivalInMinutes` as an integer representing minutes. The code is as follows:

```
public class BusViewModel
{
        /// <summary>
        /// Gets or sets the next arrival in
mintues.
        /// </summary>
        /// <value>
        /// The next arrival in mintues.
        /// </value>
        public int NextArrivalInMintues { get;
set; }


        /// <summary>
        /// Gets or sets the color.
        /// </summary>
        /// <value>
        /// The color.
        /// </value>
```

```
    public BusColor Color { get; set; }
}
///Bus color enumeration
public enum BusColor
{
  Red, Green, Blue, Yellow
}
```

In the following example, we will be converting the `TransitViewModel` color type, which is a string type, to an enum type for our `BusViewModel` domain object. To do so, we will need to create an object that implements the `ITypeConverter` interface and provides the convert implementation as follows:

```
internal class BusColorTypeConverter :
ITypeConverter<string, BusColor>
    {
/// <summary>
/// Converts the specified context.
/// </summary>
/// <param name="context">The context.</param>
/// <returns></returns>
public BusColor Convert(ResolutionContext context)
{
  const bool ignoreCase = true;
  string value = context.SourceValue.ToString();
  return (BusColor)Enum.Parse(typeof(BusColor),
value, ignoreCase);
    }
}
```

In the preceding class, we can see that we are providing a string that would be converted to a `BusColor` enum type. We resolve the value by using the `ResolutionContext`, which provides us with a property, `SourceValue`. We then use the `Enum.Parse` method to convert the object to an enum.

Now we need to let AutoMapper know about the converter. By now, you will probably know that we need to use the `CreateMap` method:

```
    public class ConvertTypeMapping
   {
        /// <summary>
    /// Gets the transit stop with resolver.
/// </summary>
/// <param name="uniqueNumber">The unique
number.</param>
/// <returns></returns>
public static BusStopViewModel
GetTransitStopWithConverters(int uniqueNumber)
{
  var repo = new TransitStopRepository();
```

```
  var transitStop =
repo.GetByUniqueNumber(uniqueNumber);

//create the map from -> to object
    //and map transit stop number from unique
number
    AutoMapper.Mapper.CreateMap<TransitStop,
BusStopViewModel>()
    .ForMember(dest => dest.TransitStopNumber,
opt => opt.MapFrom(src => src.UniqueNumber));


  //convert string to buscolor type using
converter
  AutoMapper.Mapper.CreateMap<string, BusColor>()
  .ConvertUsing<BusColorTypeConverter>();



  //convert DateTime to string type using
converter
  AutoMapper.Mapper.CreateMap<DateTime, string>()
  .ConvertUsing<DateTimeToStringTypeConverter>();

  AutoMapper.Mapper.CreateMap<ArrivalTime,
ArrivalTimeViewModel>();
    //map the object
    var result =
AutoMapper.Mapper.Map<TransitStop,
BusStopViewModel>(transitStop);

    return result;
   }
}
```

Another helpful example may be if we choose not to use `DateTime` but instead elect to send a default format that we require for the presentation layer. We will then use a `CustomTypeConverter` such as in the following code:

```
public class DateTimeToStringTypeConverter :
ITypeConverter<DateTime, string>
{
/// <summary>
/// Converts the specified context.
/// </summary>
/// <param name="context">The context.</param>
/// <returns></returns>
public string Convert(ResolutionContext context)
{
    var date = (DateTime) context.SourceValue;
    return date.ToString("dd/M/yyyy - H:mm:ss",
CultureInfo.InvariantCulture);
  }
```

```
}
```

In the preceding example, we have requested AutoMapper to always convert a `DateTime` entity to a string conversion, whenever it occurs, in order to use the format of `"dd/M/yyyy - H:mm:ss"`.

> **Note**
>
> Note that, when you apply `CustomTypeConverter`, it becomes global; thus, you may wish to provide it in an AutoMapper Profile.

Dynamic mapping

In most scenarios when we use AutoMapper, we tend to know what type of object we will map into; however, there may be cases where the object is not known till runtime. In such situations, AutoMapper provides a method called `DynamicMap`, where one can map an object to a known interface.

Our example shows an interface called `ITransitStopName`, where there is only one property string type called `Name`. The code for it is as follows:

```
public interface ITransitStopName
{
  string Name { get; set; }
}
```

In order to do the mapping, we will call the `DynamicMap` method and pass in our dynamic object for AutoMapper to map into the interface `ITransitStopName` as follows:

```
public class DynamicMapping
{
    /// <summary>
    /// Gets the transit stop.
    /// </summary>
    /// <returns></returns>
    public static ITransitStopName
GetTransitStopName()
    {
        var transitStop = new {Name =
"Taksim/Chapulcu"};

    //dynamic map
    var result =
AutoMapper.Mapper.DynamicMap<ITransitStopName>(transitStop);

        return result;
    }
```

```
}
```

By doing so, AutoMapper automatically creates the configuration for us, unless a default configuration is provided before the `DynamicMap` call. AutoMapper also calls the validation method for us in case there are invalid mappings.

Interface mapping

In some scenarios, one may come across a web service that has a contract data type of an interface rather than a concrete type. It becomes tedious to generate concrete types just for the sake of the web service. AutoMapper obviates that by allowing one to map an object to an interface type.

In the following example, we will create an interface type called `IVehicleName` and a concrete type called `PoliceVehicle`, both just containing one property of string. The code for it is as follows:

```
public interface IVehicleName
{
  string VehicleName { get; set; }
}

public class PoliceVehicle
{
  public string TypeName { get; set; }
}
```

We will then use the `Initialize` method to create the map and also provide the `ForMember` method to let AutoMapper know what to map the data to. The code for performing the stated task is as follows:

```
public class InterfaceMapping
{
    /// <summary>
    /// Gets the transit stop.
    /// </summary>
    /// <returns></returns>
    public static IVehicleName GetVehicleName()
    {
      AutoMapper.Mapper.Initialize(config =>
    config.CreateMap<PoliceVehicle,
IVehicleName>()
    .ForMember(dest => dest.VehicleName, opt =>
opt.MapFrom(src => src.TypeName))
    );

       var vehicle = new PoliceVehicle {TypeName
= "TOMA"};
```

```
    //map to interface
      var result =
AutoMapper.Mapper.Map<PoliceVehicle,
IVehicleName>(vehicle);

        return result;
    }
}
```

From the preceding code, we can now retrieve an `IVehicle` type, and pass it or use it in our messaging/webservice layer.

Existing object mapping

AutoMapper allows you to map objects to existing types. For example, if you have an object with certain existing values and you do not wish to have them overwritten by AutoMapper, nor do you wish to have AutoMapper generate a new object, you can pass in the existing object while mapping as the second parameter.

In our example, we have a `BusStopViewModel` domain object that already has a `GeoLocationViewModel` entity populated, and we wish to reuse the same object but not to overwrite the `GeoLocationViewModel` entity. The code for performing the stated task is as follows:

```
public class ExistingObjectMapping
{
    /// <summary>
    /// Gets the transit stop without location.
    /// </summary>
    /// <param name="uniqueNumber">The unique
number.</param>
    /// <returns></returns>
    public static BusStopViewModel
GetTransitStopWithExistingObject(int uniqueNumber)
    {
        var repo = new TransitStopRepository();
               var transitStop =
repo.GetByUniqueNumber(uniqueNumber);
               var busStopViewModel = new
BusStopViewModel
        {
            Location = new GeoLocationViewModel
{Longtitude = 1, Latitude = 1}
         };

        //create the map from->to object, but
don't map the location
         AutoMapper.Mapper.CreateMap<TransitStop,
BusStopViewModel>()
         .ForMember(dest => dest.Location, opt =>
opt.Ignore());
```

```
            //map to the existing object
            var result =
AutoMapper.Mapper.Map(transitStop,
busStopViewModel);

            return result;
        }
}
```

In the preceding example, we have used the `CreateMap` method for AutoMapper to ignore the mapping of the `Location` property, and we have also passed the `BusStopViewModel` class as the second parameter to AutoMapper so that it does not generate a new object for us and reuses the object and values.

Inversion of control

For the construction of `CustomValueResovler`, `CustomTypeFormatter`, and `CustomTypeConvertors`, AutoMapper allows us to use an inversion of control container, such as StructureMap, Ninject, Unity, and so on.

In order to use `StructureMap`, we will use the `Initialize` method to pass in the `ObjectFactory` entity as follows:

```
AutoMapper.Mapper.Initialize(config =>
{

 config.ConstructServicesUsing(ObjectFactory.GetInstance);
    AutoMapper.Mapper.CreateMap<TransitStop,
BusStopViewModel>();
});

and for Ninject we would use the Kernel to get
the Instance.
AutoMapper.Mapper.Initialize(config =>
{
    config.ConstructServicesUsing(x =>
Kernel.Get(x));
    AutoMapper.Mapper.CreateMap<TransitStop,
BusStopViewModel>();
});
```

By using an `IoC` container, we will have all our `CustomValueResolver`, `CustomTypeFormatter`, and `CustomTypeConvetors` entities automatically loaded for us whenever AutoMapper requests are made.

People and places you should get to know

If you need help with AutoMapper, here are some people and places that will prove invaluable:

Official sites

- Homepage: http://automapper.org/
- Manual and documentation: https://github.com/AutoMapper/AutoMapper/wiki/Getting-started
- Wiki: https://github.com/AutoMapper/AutoMapper/wiki/Getting-started
- Source code: https://github.com/AutoMapper/AutoMapper

Articles and tutorials

The top five AutoMapper resources are as follows:

- A 47-minute video from the author of AutoMapper, *Jimmy Bogard*, on DNRTV featuring AutoMapper at http://www.dnrtv.com/?showNum=155
- A 7-step guide to AutoMapper at http://taswar.zeytinsoft.com/2011/03/07/automapper-mapping-objects-part-1-of-7-nullsubsitution/
- Visual studio article on using AutoMapper at http://visualstudiomagazine.com/articles/2012/02/01/simplify-your-projections-with-automapper.aspx
- Microsoft Channel9's 16-minute video on using AutoMapper by Brandon Satrom at http://channel9.msdn.com/posts/ASPNET-MVC-With-Community-Tools-Part-10-AutoMapper
- Article on CodeProject by Andriy Buday on using AutoMapper at http://www.codeproject.com/Articles/61629/AutoMapper

Community

- Official mailing list at https://groups.google.com/forum/?fromgroups#!forum/automapper-users
- Stackoverflow tag at http://stackoverflow.com/questions/tagged/automapper
- Github at https://github.com/AutoMapper/

Blogs

- Author of AutoMapper at http://lostechies.com/jimmybogard/author/jimmybogard/

Twitter

- Jimmy Bogard: `@jbogard`
- Me: `@taswarbhatti`
- Patrick Desjardins: `@mrdesjardins`
- [Keep this one in!] For more open source information, follow Packt Publishing at http://twitter.com/#!/packtopensource