AutoMapper in ASP.NET Core Web API

Mohamed ELshafei

What is AutoMapper

AutoMapper is an object-to-object mapper which helps you to avoid manual mapping of each required property while transferring data from one object to another.

Manual mapping is tedious and time-consuming whereas Automapper does mapping in a very clean and readable way.

In an MVC application you mostly map model objects to View Models similarly in Web API you map domain objects to DTOs.

Challenge:

```
var dbContext = new MyDBDataContext();
var userDetails = dbContext.Users.FirstOrDefault(userId => userId.UserId == id);
var user = newLearningMVC.Models.User();
            if (userDetails != null)
            {
                user.UserId = userDetails.UserId;
                user.FirstName = userDetails.FirstName;
                user.LastName = userDetails.LastName;
                user.Address = userDetails.Address;
                user.PhoneNo = userDetails.PhoneNo;
                user.EMail = userDetails.EMail;
                user.Company = userDetails.Company;
                user.Designation = userDetails.Designation;
            }
 return View(user);
```

Install AutoMapper NuGet Package

- AutoMapper NuGet package developed by **Jimmy Bogard**, gives you all the required functionality to map object-to-object. Install the following required NuGet packages.

```
Install-package AutoMapper
```

- Inject AutoMapper Dependency

```
builder.Services.AddAutoMapper(typeof(MapperConfig));
```

Setup AutoMapper configuration

- Create a folder with name Configurations and add a class to it with name MapperConfig.cs. This class needs to be inherited from Profile class of AutoMapper.
- create a mapping between domain objects to different DTOs.

```
using AutoMapper;

namespace GeekStore.API.Core.Configurations
{
    public class MapperConfig : Profile
    {
        public MapperConfig()
        {
            CreateMap<Category,
                GetCategoryDetailsDto>().ReverseMap();
            CreateMap<Category,
                UpdateCategoryDiscountDto>().ReverseMap();
            CreateMap<Category,
                GetCategoryProductsDto>();
            CreateMap<Product,
                ProductDto>().ReverseMap();
        }
    }
}
```

Inject AutoMapper Dependency

- The MapperConfig file having AutoMapper configuration needs to be injected in the .NET pipeline. Add this code to **Program.cs** file

```
builder.Services.AddAutoMapper(typeof(MapperConfig));
```

- You may have **multiple configurations files** like one for the category and another for product objects, you can inject like this.

```
builder.Services.AddAutoMapper(typeof(CategoryMapperConfig),typeof(ProductMapperConfig) );
```

Invoke Mapped object in API Controller

```
namespace GeekStore.API.Core.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CategoryController : ControllerBase
    {
        private readonly IMapper _mapper;
        private readonly GeeksStoreContext _context;

        public CategoryController(IMapper mapper,
            GeeksStoreContext context)
        {
            this._mapper = mapper;
            this._context = context;
        }
    }
}
```

Map List

- Here Database Context gets a list of Category domain object, AutoMapper converts it to a list of GetCategoryDetailsDto object, and return it to the client.

```
[HttpGet("GetCategories")]
public async Task<ActionResult<List<GetCategoryDetailsDto>>> GetCategories()
{
    var categories = await this._context.Categories.ToListAsync();
    var records = _mapper.Map<List<GetCategoryDetailsDto>>(categories);
    return Ok(records);
}
```

Include child

- This code gets a **List of Categories** including a **List of Product** domain models and AutoMapper maps it to a List of GetCategoryDetailsDto.

```
[HttpGet("GetCategoryProducts/{categoryID}")]
public async Task<ActionResult<GetCategoryDetailsDto>> GetCategoryProducts(int categoryID)
{

    var products = await this._context.Categories.Where(c => c.Id == categoryID)
        .Include(_ => _.Products).ToListAsync();

    var result = _mapper.Map<List<Category>, List<GetCategoryProductsDto>>(products);
    return Ok(result);
}
```

One to One Mapping

- Get one Category domain object and map it to GetCategoryDetailsDto

```
[HttpGet("GetCategoryDetails/{categoryID}")]
public async Task<ActionResult<GetCategoryDetailsDto>> GetCategoryDetails(int categoryID)
{
    var category = await this._context.Categories.FindAsync(categoryID);
    if (category == null)
    {
        throw new Exception($"CategoryID {categoryID} is not found.");
    }

    var categoryDto = this._mapper.Map<GetCategoryDetailsDto>(category);

    return Ok(categoryDto);
}
```

Custom property Mapping

- Sometimes property mapping is not going to be mapped directly one to one like View Model has property the ***Name*** however Data Model has two properties for name as ***First Name and Last Name*** so while mapping data models to view model two properties need to concatenate.
- Custom Mapping can be done in multiple ways:
  - AfterMap OR BeforeMap
  - ForMember, MapFrom
  - IValueResolver

Custom property Mapping” AfterMap OR BeforeMap”

- You can implement custom logic using AfterMap and BeforeMap. This will apply Category Discount to the List of Products.

- Add this mapping to the MapperConfig file.

```
CreateMap<Category, GetCategoryProductsDto>()
    .AfterMap((src, dest) =>
    {
        dest.Products.ForEach(x => x.Price -= src.CategoryDiscount);
    });
```

Custom property Mapping” ForMember, MapFrom”

- ForMember configures the value of the destination object from custom logic. This logic can be built by source properties or anything else.

- Map Price of each Product from List of Product collection using ForMember and MapFrom..

```
CreateMap<Product, ProductDto>()
   .ForMember(
      dest => dest.Price,
      opt => opt.MapFrom
      ((src, dest, member, context)
         => src.Price - src.Category.CategoryDiscount));
```

Custom property Mapping” IValueResolver”

- AutoMapper allows for configuring custom value resolvers for destination members using **IValueResolver** interface. You can build custom logic for destination properties by implementing this interface.
- To set the Product price after calculating the category discount, add a new class under the folder **Configurations**. The name of the class will be **GetProductDiscount**.
- This code returns a decimal value by deducting the CategoryDiscount of Product object.

Custom property Mapping” IValueResolver”

```
public class GetProductDiscount : IValueResolver<Product, ProductDto, decimal>
  {
     public decimal Resolve(Product source,
        ProductDto destination, decimal destMember,
        ResolutionContext context)
     {
        return source.Price - source.Category.CategoryDiscount;
     }
  }
```

- Use this while configuring mapping between Product and ProductDTO.

```
CreateMap>Product, ProductDto>()
   .ForMember(dest => dest.Price,
      source => source.MapFrom<GetProductDiscount>())
   .ForMember(dest => dest.CategoryName,
      source => source.MapFrom(s => s.Category.CategoryName));
```