AutoMapper

> AutoMapper is a convention-based object-object mapper, that can be configured as an injectable service for the ASP.NET Core framework

- AutoMapper maps properties to properties where the name matches between the two involved entities (usually domain class and DTO)
- We need to provide a class that contains the mapping profiles (mapping instructions) between entities, this class is usually named "MappingProfiles", "AutoMapperProfiles" or some variation.
- When adding AutoMapper as a service, we need to provide the assembly that contains the configuration class ("MappingProfiles" or similar), that's why when registering the AutoMapper service, as an argument we pass the typeof for the configuration class.

Configure AutoMapper so it can be injected as a dependency in the classes of the application:

```
using EFCorePeliculas;
using EFCorePeliculas.CompiledModels;
using EFCorePeliculas.Services;
using Microsoft.EntityFrameworkCore;
using System.Text.Json.Serialization;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers().AddJsonOptions(options =>
options.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.IgnoreCycles);
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");


builder.Services.AddDbContextFactory<ApplicationDbContext>(opciones =>
{
   opciones.UseSqlServer(connectionString, sqlServer =>
sqlServer.UseNetTopologySuite());
   opciones.UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);
});

builder.Services.AddDbContext<ApplicationDbContext>();
builder.Services.AddAutoMapper(typeof(MappingProfiles));  // ---------------> THIS
IS THE LINE TO ADD AUTOMAPPER AS A SERVICE, WE ARE PASSING THE ASSEMBLY THAT
CONTAINS THE CONFIGURATION
```

AutoMapper configuration class ("MappingProfiles" or any similar name)

Example of AutoMapper configuration class, "MappingProfiles" class, this one has specifications on how to map certain properties of the target entity (dto) from the source entity (src), this in the first two "ForMember" methods. The last "ForMember" method is configuring a custom value resolver, which is similar but a little bit more convoluted:

```
using API.DTOs;
using AutoMapper;
using Core.Entities;

namespace API.Helpers
{
   public class MappingProfiles : Profile
   {
       public MappingProfiles()
       {
           CreateMap<Product, ProductToReturnDTO>()
               .ForMember(
                   dto => dto.ProductBrand,
                   opt => opt.MapFrom(src => src.ProductBrand.Name))
               .ForMember(
                   dto => dto.ProductType,
                   opt => opt.MapFrom(src => src.ProductType.Name))
               .ForMember(
                   dto => dto.PictureUrl,
                   opt => opt.MapFrom<ProductUrlResolver>());
       }
   }
}
```

Content of the custom value resolver class "ProductUrlResolver" of the previous example. This class implements the IValueResolver interface that comes from AutoMapper, so it implements the "Resolve" method, that appends a value to the property "PictureUrl" that comes from the source entity (Product):

```
using AutoMapper;

namespace API.Helpers
{
   public class ProductUrlResolver
       : IValueResolver<Product, ProductToReturnDTO, string>
   {
       private readonly IConfiguration _config;
       public ProductUrlResolver(IConfiguration config)
       {
           _config = config;
       }

       public string Resolve(
           Product source,
           ProductToReturnDTO destination,
           string destMember,
           ResolutionContext context)
       {
           if(!string.IsNullOrEmpty(source.PictureUrl))
           {
               return _config["ApiUrl"] + source.PictureUrl;
           }
           return null;
       }
   }
}
```

AutoMapper service injection and usage

Example of injecting AutoMapper service in an API controller, to map from our "Product" domain class into our "ProductToReturnDTO" DTO:

```
using Infrastructure.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace API.Controllers
{
    public class ProductsController : BaseApiController
    {
        private readonly IGenericRepository<Product> _productsRepo;
        private readonly IGenericRepository<ProductBrand> _productBrandsRepo;
        private readonly IGenericRepository<ProductType> _productTypesRepo;
        private readonly IMapper _mapper;

        public ProductsController(
            IGenericRepository<Product> productsRepo,
            IGenericRepository<ProductBrand> productBrandsRepo,
            IGenericRepository<ProductType> productTypesRepo,
            IMapper mapper)
        {
            _productsRepo = productsRepo;
            _productBrandsRepo = productBrandsRepo;
            _productTypesRepo = productTypesRepo;
            _mapper = mapper;
        }

        [HttpGet]
        public async Task<ActionResult<IReadOnlyList<ProductToReturnDTO>>>
            GetProducts()
        {
            var spec = new ProductsWithTypesAndBrandsSpecification();
            var products = await _productsRepo.ListAsync(spec);
            return Ok(
                _mapper.Map<IReadOnlyList<ProductToReturnDTO>>(products)
                    .ToList()); // ---- MAP
        }

        [HttpGet("{id:int}")]
        public async Task<ActionResult<ProductToReturnDTO>> GetProduct(int id)
        {
            var spec = new ProductsWithTypesAndBrandsSpecification(id);
            var product = await _productsRepo.GetEntityWithSpec(spec);
            return _mapper.Map<ProductToReturnDTO>(product); // --- MAP
        }
    }
}
```

As we could appreciate in the previous code example, as any other service from the dependency injection system in .NET, we add the corresponding interface ("IMapper" in this case) into the constructor of the class, and we initialize a readonly field from it, the dependency injection will then instantiate the service (AutoMapper) at the right time for our class to be able to access it when instantiated. When mapping using AutoMapper, we use the .Map<T> method, the type in the angular brackets is the target type, the type we want to map into (IReadOnlyList<ProductToReturnDTO> and ProductToReturnDTO in the example), and as an argument for the method, we provide the source entity we are mapping from (IReadOnlyList<Product> and Product in the example). Based on the matching names of the properties in the source and target entities, and on the additional configuration we provided in the AutoMapper configuration class ("MappingProfiles" or any similar name) then mapping will be performed.