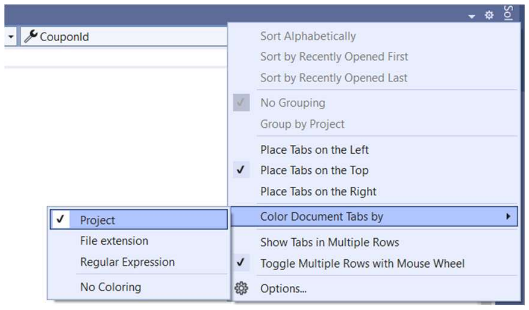
Working With Microservices

File → New → Project using Blank Solution template.
Add four folders to create structure of project
FrontEnd, Gateway, Integration and Services
Now add one new project in Services Folder
Select ASP.NET Core Web API project and name it Mango.Services.CouponAPI
Default Weather Forecast API will be created
Go to launchsettings.json and change https profile ApplicationUrl to 7001 (Just need to simply track and match with other APIs
Add models folder to project and add Coupon model

```
namespace Mango.Services.CouponAPI.Models
{
    public class Coupon
    {
        public int CouponId { get; set; }
        public string? CouponCode { get; set; }
        public double DiscountAmount { get; set; }
        public int MinAmount { get; set; }
    }
}
```

Tip: Go to gear icon at right top of tab, click it and select 'Color Document Tabs by' and – Project. It will show different project related tabs in different colors.



Data Transfer Object (DTO)

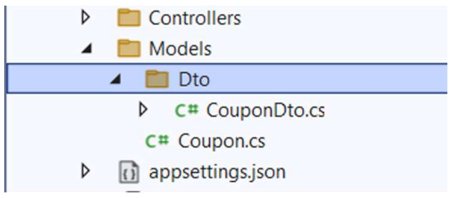
Till now our web API usually exposes the database entities to the client. The client receives data that maps directly to your database tables. However, that's not always a good idea. Sometimes you want to change the shape of the data that you send to client.

For example, you might want to:

- Remove circular references.
- Hide particular properties that clients are not supposed to view.
- Omit some properties in order to reduce payload size.
- Flatten object graphs that contain nested objects, to make them more convenient for clients.
- Avoid "over-posting" vulnerabilities. Over-Posting - A client can also send more data than you expected.

- Decouple your service layer from your database layer.

To accomplish this, you can define a data transfer object (DTO). A DTO is an object that defines how the data will be sent over the network. Here we have both classes same so you can manage with one class only or keep both as shown in figure below:



```
namespace
Mango.Services.CouponAPI.Models.Dto
{
    public class CouponDto
    {
        public int CouponId { get; set; }
        public string? CouponCode { get; set; }
        public double DiscountAmount { get; set; }
        public int MinAmount { get; set; }
    }
}
```

AutoMapper

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.

In our project we have added the coupon model and coupon DTO, but we need something to convert coupon DTO to coupon or vice versa. When we have that requirement, we can do that manual mapping, but it is tedious and rather there are many packages that we should use to enhance our speed and make code more elegant.

One of those packages is auto mapper and that, as the name suggests, is responsible for mapping one object to other. So, let's add auto mapper to our project and we need to store our coupon in a database.

Add required nuGet Package i.e. **AutoMapper** using nuGet Package Manager

As we are going to use SQL Server and Entity Framework Core so add those required packages also.

- **Microsoft.EntityFrameworkCore**
- **Microsoft.EntityFrameworkCore.SqlServer**
- **Microsoft.EntityFrameworkCore.Tools**
- **Microsoft.EntityFrameworkCore.Design**

We need to create a database using entity framework core. We have already added the NuGet packages for that. Add DB context. Right click on the project here, add a folder for Data and in that Data folder create a class named AppDbContext.

```
using Mango.Services.CouponAPI.Models;
using Microsoft.EntityFrameworkCore;
namespace Mango.Services.CouponAPI.Data
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions<AppDbContext> options)
: base(options)
        {
        }

        public DbSet<Coupon> Coupons { get; set; }
    }
}
```

Now add Data Annotations in Coupon class.

```
using System.ComponentModel.DataAnnotations;
namespace Mango.Services.CouponAPI.Models
{
    public class Coupon
    {
        [Key]
        public int CouponId { get; set; }
        [Required]
        public string? CouponCode { get; set; }
        [Required]
        public double DiscountAmount { get; set; }
        public int MinAmount { get; set; }
    }
}
```

Go to AppSettings.json and define connection string.

```
"ConnectionStrings": {
  "DefaultConnection":
"Server=(LocalDb)\\MSSQLLocalDB;Database=Mango_Coupon;Trusted_Connec
tion=True;TrustServerCertificate=True"
},
```

Register dbContext in program.cs

```
builder.Services.AddDbContext<AppDbContext>(option =>
{
    option.UseSqlServer(builder.Configuration.GetConnectionString
("DefaultConnection"));
});
```

Now add-migrations and update database using package manager console.

```
PM> add-migration AddCouponToDB
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
```

```
PM> update-database
```

We have created our coupons table in the database, but we need to populate some entries in there. We can seed some database directly but that way we do not start with an empty table. Seeding database is very simple with entity framework core right here we have to override the OnModelCreating

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.Entity<Coupon>().HasData(new Coupon
    {
        CouponId = 1,
        CouponCode = "10OFF",
        DiscountAmount = 10,
        MinAmount = 20
    });
    modelBuilder.Entity<Coupon>().HasData(new Coupon
    {
        CouponId = 2,
        CouponCode = "20OFF",
        DiscountAmount = 20,
        MinAmount = 40
    });
}
```

Now add-migrations and update database using package manager console.

```
PM> add-migration SeedCouponTables
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
```

Now, rather than updating database, what I want to do is if there are any pending migrations I want when I run the application, it should automatically apply those migrations. That way we do not have to write the command update database.

Go to Program.cs. Create a method as:

```
static void ApplyMigration(WebApplication? app)
{
    using (var scope = app.Services.CreateScope())
    {
        var _db =
scope.ServiceProvider.GetRequiredService<AppDbContext>();

        if (_db.Database.GetPendingMigrations().Count() > 0)
        {
            _db.Database.Migrate();
        }
    }
}
```
Also call it in main method in pipeline.
```
app.MapControllers();
ApplyMigration(app);
app.Run();
```
Try executing and check database seeding is done in table in SQL Server.

Go Controller folder and add an Empty controller named CouponApiController and add:
```
private readonly AppDbContext _db;
public CouponAPIController(AppDbContext db)
{
    _db = db;
}
```
Add an endpoint :
```
[HttpGet]
public object Get()
{
    try
    {
        IEnumerable<Coupon> objlist = _db.Coupons.ToList();
        return objlist;
    }
    catch (Exception ex)
    {
        throw;
    }
    return null;
}
```
Now get individual Coupon and for that add another endpoint
```
[HttpGet]
[Route("{id:int}")]
public object Get(int id)
{
    try
    {
        Coupon objlist = _db.Coupons.First(u => u.CouponId == id);
        return objlist;
    }
    catch (Exception ex)
    {

        throw;
    }
    return null;
}
```
Test API after rebuild.

We have two end points one get and another one is get with Id. In both of the end points we are returning different things. But what if we want to have a common response for all the end points in our website?

Like right now, if there is any error, we cannot return that back. We are returning the coupon data. If we want something to be common for all the API endpoints in our microservices, that way, whenever an API or rather multiple APIs are being consumed by an application, it can be sure that okay, the response that we will get from API will always be in one object format. And that will be a great architecture for our API response.

Add one class to project named ResponseDto

Also add private ResponseDto _response;

declaration in controller.

```
namespace Mango.Services.CouponAPI.Models.Dto
{
    public class ResponseDto
    {
        public object? Result { get; set; }
        public bool IsSuccess { get; set; } = true;
        public string Message { get; set; } = "";
    }
}
```
Add a field in controller:

```
public class CouponAPIController : ControllerBase
{
        ....
        private ResponseDto _response;
        ....
}
```
Add following in CTOR

```
public CouponAPIController(AppDbContext db)
{
        _db = db;
        _response = new ResponseDto();
}
```
Now modify both endpoints

```
[HttpGet]
```

```
public ResponseDto Get()
{
    try
    {
        IEnumerable<Coupon> objlist = _db.Coupons.ToList();
        _response.Result = objlist;
    }
    catch (Exception ex)
    {
        _response.IsSuccess = false;
        _response.Message = ex.Message;
    }
    return _response;
}
[HttpGet]
[Route("{id:int}")]
public ResponseDto Get(int id)
{
    try
    {
        Coupon obj = _db.Coupons.First(u => u.CouponId == id);
        _response.Result = obj;
    }
    catch (Exception ex)
    {
        _response.IsSuccess = false;
        _response.Message = ex.Message;
    }
    return _response;
}
```
Run and Test API.

Adding AutoMapper to project

Next thing that we want to work on here is when we are returning something here, you can see we are returning the coupon object. Here we have a list or IEnumerable of coupon and in another endpoint, we are returning the coupon itself.

Already we have added DTOs in our project. We should not return the coupon or the data object itself. We should return the DTO. What you might have to do here is you might have to do a manual conversion. Like here you can say coupon DTO is equal to new and then you can map the individual like coupon ID is equal to object coupon ID. Next one coupon code. Then we have discount amount and we have minimum amount. Once you do the conversion, you can return back the coupon data. That's not great. Think about if you had to do that for all the endpoints in all the microservices, that will be a nightmare.

So, here is a need of automapper. And for that inside the project add a new class file named MappingConfig. You can do all of this mapping configuration directly in Program.cs, but I like to separate it out otherwise the Program.cs will be clumsy.

```
using AutoMapper;
using Mango.Services.CouponAPI.Models;
using Mango.Services.CouponAPI.Models.Dto;
namespace Mango.Services.CouponAPI
{
    public class MappingConfig
    {
        public static MapperConfiguration RegisterMaps()
        {
            var mappingConfig = new MapperConfiguration(config =>
            {
                config.CreateMap<CouponDto, Coupon>();
                config.CreateMap<Coupon, CouponDto>();
            });
            return mappingConfig;
        }
    }
}
```

Now modify your controller

```
using AutoMapper;
using Mango.Services.CouponAPI.Data;
using Mango.Services.CouponAPI.Models;
using Mango.Services.CouponAPI.Models.Dto;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace Mango.Services.CouponAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CouponAPIController : ControllerBase
    {
        private readonly AppDbContext _db;
        private ResponseDto _response;
        private IMapper _mapper;

        public CouponAPIController(AppDbContext db, IMapper mapper)
        {
            _db = db;
            _mapper = mapper;
            _response = new ResponseDto();
        }

        [HttpGet]
        public ResponseDto Get()
        {
            try
            {
                IEnumerable<Coupon> objList = _db.Coupons.ToList();
                _response.Result =
_mapper.Map<IEnumerable<CouponDto>>(objList);
            }
            catch (Exception ex)
            {
                _response.IsSuccess = false;
                _response.Message = ex.Message;
            }
            return _response;
        }
        [HttpGet]
        [Route("{id:int}")]
        public ResponseDto Get(int id)
        {
            try
            {
                Coupon obj = _db.Coupons.First(u => u.CouponId ==
id);
                _response.Result = _mapper.Map<CouponDto>(obj);
            }
            catch (Exception ex)
            {
                _response.IsSuccess = false;
                _response.Message = ex.Message;
            }
            return _response;
        }
    }
}
```

Let’s add one more end point to get coupon details by coupon code.

```
[HttpGet]
[Route("GetByCode/{code}")]
```

```
public ResponseDto GetByCode(string code)
{
    try
    {
        Coupon obj = _db.Coupons.FirstOrDefault(u =>
u.CouponCode.ToLower() == code.ToLower());
        _response.Result = _mapper.Map<CouponDto>(obj);
    }
    catch (Exception ex)
    {
        _response.IsSuccess = false;
        _response.Message = ex.Message;
    }
    return _response;
}
```

Run and Test API

You might face errors as we have not registered mapper in Program.cs. Let's do it.

Go to Program.cs and add following lines before building app.

```
var builder = WebApplication.CreateBuilder(args);
…
…
…
IMapper mapper = MappingConfig.RegisterMaps().CreateMapper();
builder.Services.AddSingleton(mapper);
builder.Services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());

…
…
…

var app = builder.Build();
```

Test API.

Now try to add HttpPost method for adding Coupon Data.

```
[HttpPost]
public ResponseDto Post([FromBody] CouponDto couponDto)
{
    try
    {
        Coupon obj = _mapper.Map<Coupon>(couponDto);
        _db.Coupons.Add(obj);
        _db.SaveChanges();
        _response.Result = _mapper.Map<CouponDto>(obj);
    }
    catch (Exception ex)
    {
        _response.IsSuccess = false;
        _response.Message = ex.Message;
    }
    return _response;
}
```
Add HttpPut method

```
[HttpPut]
public ResponseDto Put([FromBody] CouponDto couponDto)
{
    try
    {
        Coupon obj = _mapper.Map<Coupon>(couponDto);
        _db.Coupons.Update(obj);
        _db.SaveChanges();

        _response.Result = _mapper.Map<CouponDto>(obj);
    }
    catch (Exception ex)
    {
        _response.IsSuccess = false;
        _response.Message = ex.Message;
    }
    return _response;
}
```
Add HttpDelete method
```
[HttpDelete]
[Route("{id:int}")]
public ResponseDto Delete(int id)
{
    try
    {
        Coupon obj = _db.Coupons.First(u => u.CouponId == id);
        _db.Coupons.Remove(obj);
        _db.SaveChanges();
    }
    catch (Exception ex)
    {
        _response.IsSuccess = false;
        _response.Message = ex.Message;
    }
    return _response;
}
```