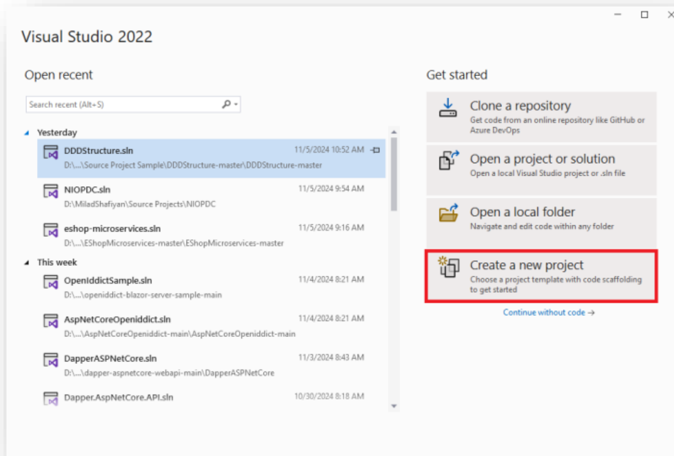
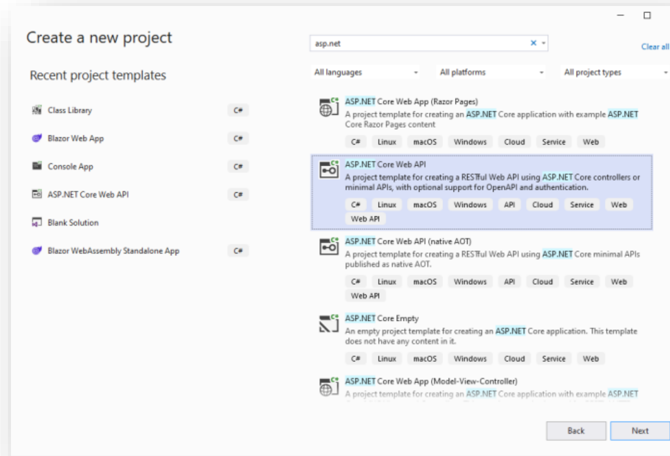
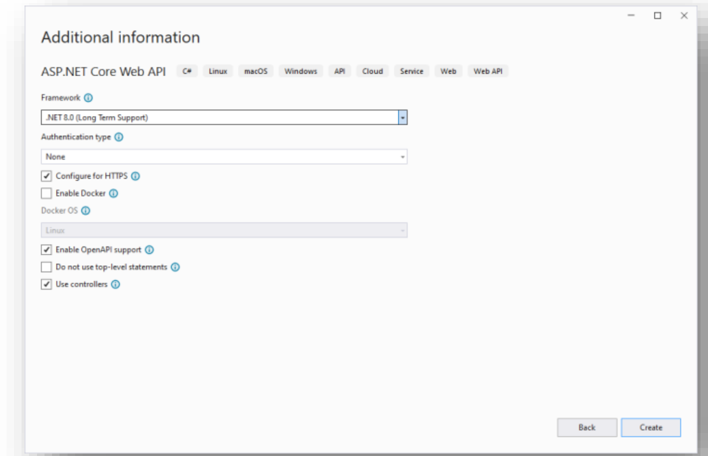
# آموزش ASP.NET CORE

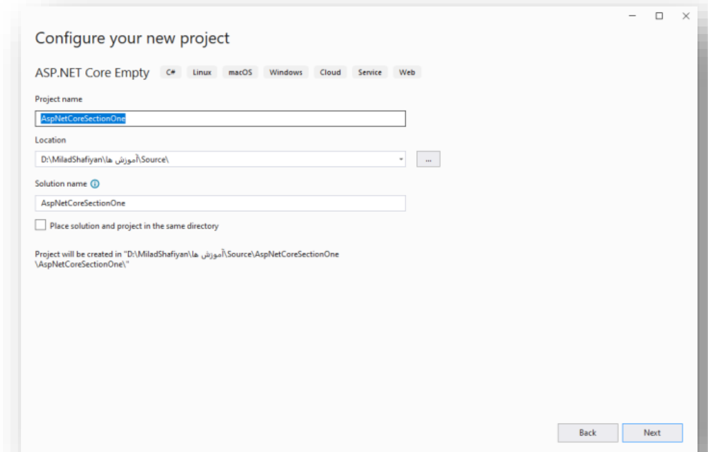
ارائه شده توسط  میلاد شفیعیان

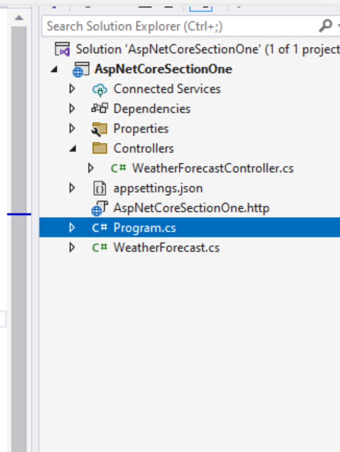
## ساخت پروژه در Visual Studio 2022 :












```
1   var builder = WebApplication.CreateBuilder(args);
2
3   // Add services to the container.
4
5   builder.Services.AddControllers();
6   // Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
7   builder.Services.AddEndpointsApiExplorer();
8   builder.Services.AddSwaggerGen();
9
10  var app = builder.Build();
11
12  // Configure the HTTP request pipeline.
13  if (app.Environment.IsDevelopment())
14  {
15      app.UseSwagger();
16      app.UseSwaggerUI();
17  }
18
19  app.UseHttpsRedirection();
20
21  app.UseAuthorization();
22
23  app.MapControllers();
24
25  app.Run();
```

میلاد شفیعیان

# ساختار Program.cs در Asp.Net Core



```csharp
var builder = WebApplication.CreateBuilder(args);

var configuration = builder.Configuration;
configuration.GetConnectionString("DatabaseName");
configuration.AddJsonFile("appsettings.json",
    optional: true,reloadOnChange: true);
configuration.AddEnvironmentVariables();

builder.Logging.AddConsole();
builder.Logging.AddEventSourceLogger();
builder.Logging.AddDebug();

// Add services to the container.
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.Run();
```

- الگوی سازنده (Builder Pattern)
- پیکربندی (Configuration)
- لاگینگ (Logging)
- مجموعه سرویس‌ها (Service Collection)
- خط لوله میدلور (Middleware Pipeline)
- Run Application

Solution Explorer

- Solution 'AspNetCoreSectionOne' (1 of 1 project)
  - AspNetCoreSectionOne
    - Connected Services
    - Dependencies
    - Properties
    - Controllers
      - WeatherForecastController.cs
    - appsettings.json
    - AspNetCoreSectionOne.http
    - Program.cs
    - WeatherForecast.cs

Dependency Injection الگوی طراحی برای مدیریت وابستگی‌ها در پروژه‌هاست. این الگو به شما اجازه می‌دهد تا وابستگی‌ها را (مثل سرویس‌ها، کلاس‌ها و اینترفیس‌ها) به طور جداگانه تعریف کنید و به راحتی آن‌ها را به اجزای دیگر تزریق کنید، بدون اینکه نیازی به ساخت مستقیم آن‌ها در هر بخش از کد باشد.

ASP.NET Core یک **کانتینر DI داخلی** دارد که برای **مدیریت و تزریق وابستگی‌ها** استفاده می‌شود. این کانتینر به طور پیش‌فرض همراه با پروژه ASP.NET Core آمده و به سادگی قابل استفاده است. در این الگو، سه مرحله اصلی وجود دارد:

۱- ثبت وابستگی‌ها (Registering Dependencies)

۲- تزریق وابستگی‌ها (Injecting Dependencies)

۳- مدیریت عمر وابستگی‌ها (Managing Dependency Lifetime)

Managing Dependency Lifetime → AddScoped

```
builder.Services.AddScoped<ITestService, TestService>();
```

→ Registering Dependencies

میلاد شفیعیان

# مراحل استفاده از Dependency Injection در ASP.NET Core



❖ تعریف اینترفیس و کلاس

```
namespace AspNetCoreSectionOne.Interfaces;

public interface ITestService
{
    void TestMethode();
}
```

```
using AspNetCoreSectionOne.Interfaces;

namespace AspNetCoreSectionOne.Impelements;

public class TestService : ITestService
{
    public void TestMethode()
    {
        throw new NotImplementedException();
    }
}
```

Solution Explorer

- Solution 'AspNetCoreSectionOne' (1 of 1 project)
  - External Sources
  - AspNetCoreSectionOne
    - Connected Services
    - Dependencies
    - Properties
    - Controllers
      - WeatherForecastController.cs
    - Impelements
      - TestService.cs
    - Interfaces
      - ITestService.cs
    - appsettings.json
    - AspNetCoreSectionOne.http
    - Program.cs
    - WeatherForecast.cs

❖ ثبت وابستگی‌ها در DI Container

```
builder.Services.AddScoped<ITestService, TestService>();
```

❖ تزریق وابستگی در کلاس‌های دیگر

```
using AspNetCoreSectionOne.Interfaces;
using Microsoft.AspNetCore.Mvc;

namespace AspNetCoreSectionOne.Controllers
{
    [ApiController]
    [Route("[controller]")]
    1 reference
    public class WeatherForecastController : ControllerBase
    {
        private readonly ITestService _testService;
        0 references
        public WeatherForecastController(ITestService testService)
        {
            _testService = testService;
        }

        [HttpGet(Name = "GetWeatherForecast")]
        0 references
        public void Get()
        {
            _testService.TestMethode();
        }
    }
}
```

```csharp
using AspNetCoreSectionOne.Interfaces;
using Microsoft.AspNetCore.Mvc;

namespace AspNetCoreSectionOne.Controllers
{
    [ApiController]
    [Route("[controller]")]
    1 reference
    public class WeatherForecastController : ControllerBase
    {
        private readonly ITestService _testService;
        0 references
        public WeatherForecastController(ITestService testService)
        {
            _testService = testService;
        }

        [HttpGet(Name = "GetWeatherForecast")]
        0 references
        public void Get()
        {
            _testService.TestMethode();
        }
    }
}
```

## ❖ Constructor Injection

رایج‌ترین روش تزریق در ASP.NET Core

```csharp
public void SomeMethod([FromServices] ITestService testService)
{
    testService.TestMethode();
}
```

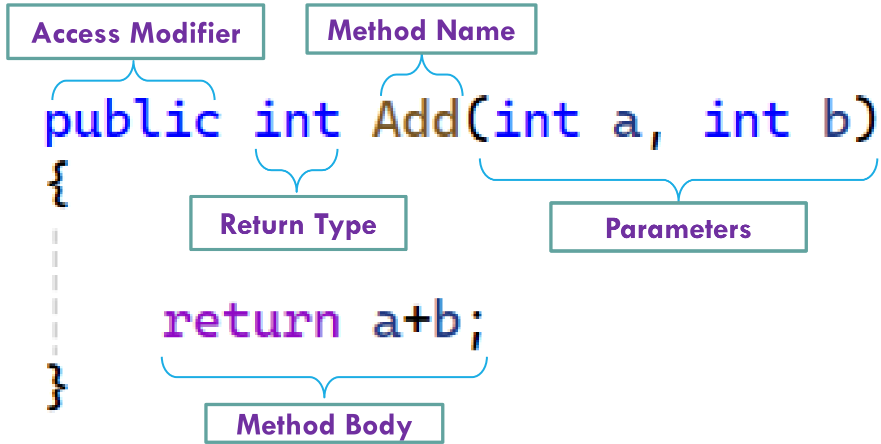
## ❖ Methode Injection

در صورتی که فقط در یک متد به وابستگی نیاز داشته باشید

```csharp
public class MyClass
{
    [FromServices]
    0 references
    public ITestService testService { get; set; }
}
```

## ❖ Property Injection

وابستگی را به عنوان یک پراپرتی تعریف میکند




❖ متد ها (Methods)

**۱- Access Modifier**

Public ، Private ، Protected و Internal که سطوح دسترسی متد را مشخص میکند

**۲- Return Type (نوع بازگشتی)**

نوع داده ای که متد باز میگرداند که میتواند بدون مقدار بازگشتی (void)، int، string ، decimal و ...

**۳- Method Name (نام متد)**

نامی معنا دارد و خوانا که برای متد اختصاص داده میشود

**۴- Parameters (پارامترها)**

ورودی‌هایی که به متد ارسال می‌شود. شامل نوع داده و نام پارامتر است. متد می‌تواند بدون پارامتر یا با پارامتر باشد

**۵- Method Body (بدنه متد)**

کدی که وظیفه متد را انجام می‌دهد و داخل آکولاد {} قرار می‌گیرد

❖ کلمه کلیدی return برای بازگرداندن مقدار (در صورت وجود نوع بازگشتی)

میلاد شفیعیان

# سطوح دسترسی (Access Modifier)

Asp.Net Core

```
2 references
public class method1
{
    1 reference
    public void DisplayMessage()
    {
        Console.WriteLine("This is a public method.");
    }
}
0 references
public class method2
{
    0 references
    public void CallMethod()
    {
        method1 example = new method1();
        example.DisplayMessage();
    }
}
```

❖ انواع Access Modifier ها

## ۱- Public

متد یا فیلد با این دسترس‌پذیری از هر کجا در برنامه (داخل یا خارج از کلاس) قابل دسترسی است

```
2 references
public class method1
{
    1 reference
    private void DisplayMessage()
    {
        Console.WriteLine("This is a public method.");
    }
}
0 references
public class method2
{
    0 references
    public void CallMethod()
    {
        method1 example = new method1();
        example.DisplayMessage();
    }
}
```

'method1.DisplayMessage()' is inaccessible due to its protection level

## ۲- Private

متد یا فیلد فقط در داخل همان کلاس قابل دسترسی است

میلاد شفیعیان

# سطوح دسترسی (Access Modifier)

Asp.Net Core

```
1 reference
public class Parent
{
    1 reference
    protected void ProtectedMethod()
    {
        Console.WriteLine("This is a protected method.");
    }
}

0 references
public class Child : Parent
{
    0 references
    public void CallProtectedMethod()
    {
        ProtectedMethod();
    }
}

internal class InternalClass
{
    0 references
    internal void InternalMethod()
    {
        Console.WriteLine("This is an internal method.");
    }
}
```

## ۳- Protected

فقط در داخل کلاس اصلی یا کلاس‌های مشتق‌شده قابل دسترسی است

## ۴- Internal

فقط در همان اسمبلی (Assembly) قابل دسترسی است

## ۵- Protected Internal

ترکیبی از protected و internal از داخل اسمبلی یا از کلاس‌های مشتق‌شده قابل دسترسی است

## ۶- Private Internal

فقط از داخل کلاس اصلی یا کلاس‌های مشتق‌شده که در یک اسمبلی هستند قابل دسترسی است

## جدول مقایسه دسترس پذیری‌ها

| Access Modifier | داخل کلاس | کلاس مشتق‌شده | همان اسمبلی | خارج از اسمبلی |
|---|---|---|---|---|
| public | ✓ | ✓ | ✓ | ✓ |
| private | ✓ | ✗ | ✗ | ✗ |
| protected | ✓ | ✓ | ✗ | ✗ |
| internal | ✓ | ✓ | ✓ | ✗ |
| protected internal | ✓ | ✓ | ✓ | ✗ |
| private protected | ✓ | ✓ (همان اسمبلی) | ✓ | ✗ |

❖ فیلد ها (Properties)

### ۱- Access Modifier

Public ، Private ، Protected و Internal که سطوح دسترسی Property را مشخص میکند

### ۲- Type (نوع داده)

نوع داده‌ای که Property نگهداری یا برمی‌گرداند مانند int, string, bool و ...

### ۳- Property Name (نام فیلد)

نام Property که معمولاً به صورت PascalCase نوشته می‌شود

### ۴- Accessor (دسترسی)

دسترسی به فیلد مورد نظر را مشخص میکند

## ۱- Get Accessor

برای بازیابی مقدار Property استفاده می‌شود. معمولاً به متغیر خصوصی (private field) کلاس اشاره می‌کند

```
private int _age;
0 references
public int Age
{
    get { return _age; }
}
```

## ۲- Set Accessor

برای مقداردهی به Property استفاده می‌شود. معمولاً مقداری که به Property داده می‌شود را در یک متغیر ذخیره می‌کند

```
private int _age;
0 references
public int Age
{
    set { _age = value; }
}
```

## ۳- Auto-Implemented Property

زمانی که نیازی به فیلد خصوصی (private field) ندارید، از این ویژگی استفاده می‌شود. کامپایلر به صورت خودکار یک فیلد خصوصی ایجاد می‌کند

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

## ۴- Read Only Property

Property که فقط get دارد و امکان تغییر مقدار آن از خارج کلاس وجود ندارد

```
private string _name = "Milad Shafiyan";
0 references
public string Name
{
    get { return _name; }
}
```

## ۵- Write Only Property

Property که فقط set دارد و مقدار آن فقط قابل تنظیم است

```
private int _age;
0 references
public int Age
{
    set { _age = value; }
}
```

## ۱- کپسوله سازی فیلد ها (Encapsulation)

Property به شما این امکان را می‌دهد که متغیرهای خصوصی کلاس را از طریق متدهای کنترل‌شده بخوانید یا تغییر دهید

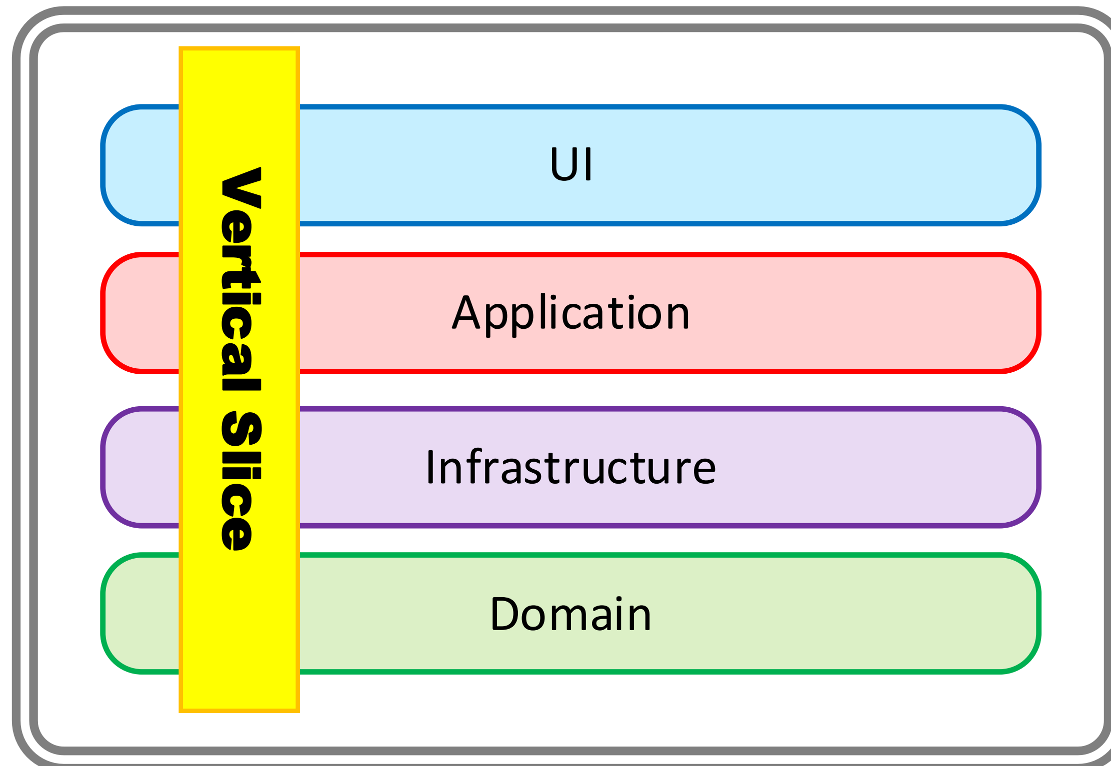
می‌توانید در set یا get شرطهایی اضافه کنید تا مقادیر ورودی یا خروجی کنترل شوند

```
private int _age;
0 references
public int Age
{
    get { return _age; }
    set
    {
        if (value > 0)
            _age = value;
        else
            throw new ArgumentException("Age must be positive.");
    }
}
```

❖ تاریخچه

Vertical Slice Architecture در اوایل دهه ۲۰۱۰ با ظهور معماری میکروسرویس‌ها (Microservices) و توسعه چابک (Agile Development) در مباحث معماری ماژولار و طراحی مبتنی بر ویژگی‌ها (Feature-Based Design) مطرح شد

این ایده توسط افرادی نظیر Jimmy Bogard سازنده کتابخانه MediatR و Udi Dahan (پیشگام در معماری‌های مدرن) مورد حمایت و توسعه قرار گرفت.
یکی از کتاب‌های تأثیرگذار در این زمینه، Clean Architecture نوشته (Robert C. Martin) بود که رویکردی مشابه این معماری را با تمرکز بر جداسازی منطق کسب‌وکار ارائه داد



# معماری Vertical Slice

میلاد شفیعیان

❖ ویژگی های اصلی

**۱- تقسیم‌بندی بر اساس ویژگی‌ها (Feature-based Division)**

✓ هر برش یا Slice شامل کدهای مربوط به یک قابلیت مشخص است.
✓ این برش‌ها به صورت مستقل از دیگر بخش‌های نرم‌افزار توسعه، تست، و مستقر می‌شوند.

**۲- استقلال و خودکفایی هر Slice (Self-contained Vertical Slice)**

✓ هر برش شامل تمام اجزای لازم است، از جمله:
✓ کنترلر یا API Endpoint در صورت وجود
✓ منطق کسب‌وکار (Business Logic)
✓ دسترسی به داده (Data Access)
✓ این استقلال باعث کاهش وابستگی بین ماژول‌ها می‌شود.

**۳- تمرکز بر قابلیت‌ها به‌جای لایه‌ها (Capability-driven Design)**

✓ به‌جای جداسازی کد بر اساس لایه‌های تکنیکی، مانند UI, BLL و DAL، کد بر اساس ویژگی‌های کسب‌وکار (Business Features) سازمان‌دهی می‌شود.

**۴- توسعه موازی آسان (Ease of Parallel Development)**

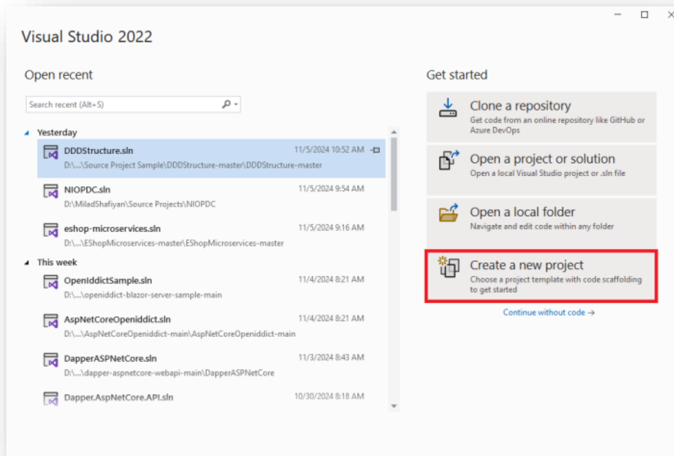
✓ تیم‌های مختلف می‌توانند همزمان بر روی ویژگی‌های مختلف کار کنند بدون اینکه اختلالی در دیگر بخش‌ها ایجاد شود.

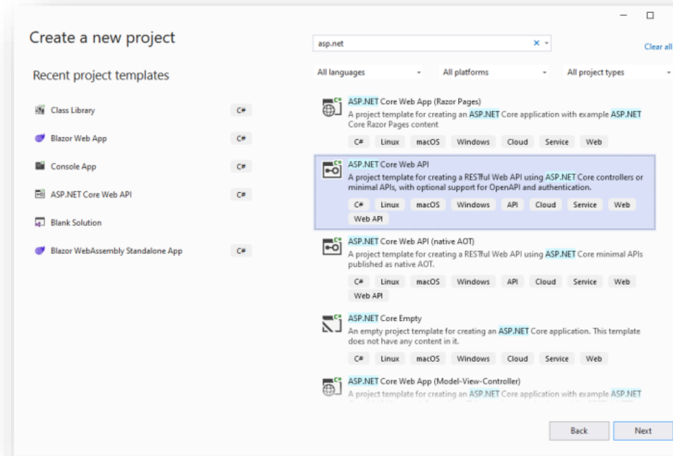
میلاد شفیعیان

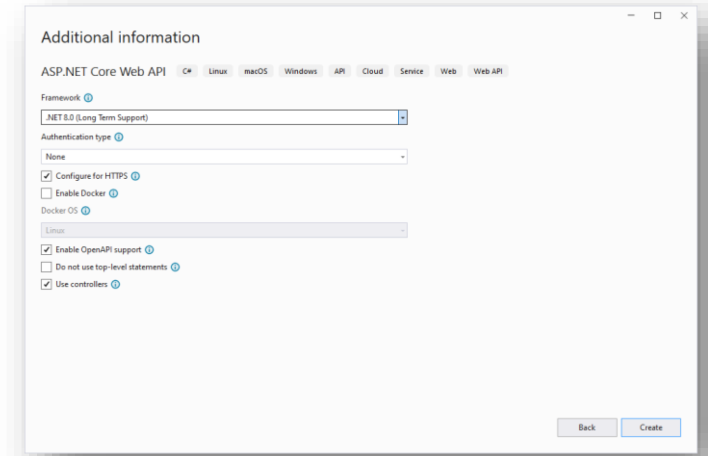
# ساخت اولین پروژه با معماری Vertical Slice

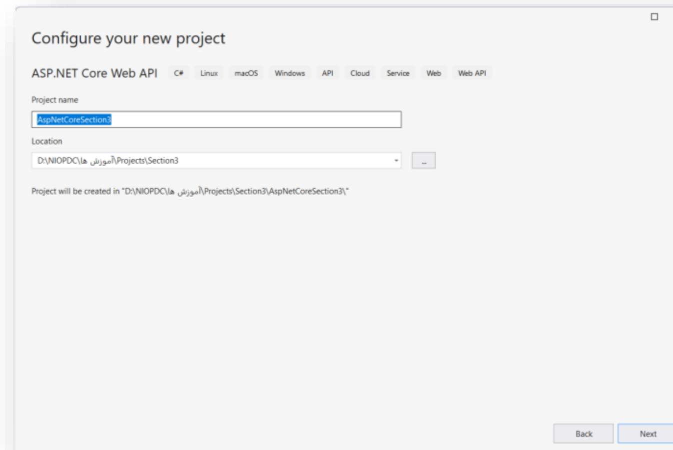


## ساخت پروژه در Visual Studio 2022 :



```
◢ Domain
   ▷ Entities
   ▷ Interfaces
```

## ❖ ایجاد لایه Domain

### ۱- Entities

- **ایجاد کلاس BaseEntity :**

یک کلاس پایه (Base Class) برای موجودیت‌های (Entities) دیگر در پروژه استفاده می‌شود. زمانی که ویژگی ها (Property) مشترک در بسیاری از موجودیت‌های دیگر داریم استفاده می‌شود.

- **BaseEntity<T> :**

به صورت جنریک (**Generic**) تعریف شده است، به این معنا که می‌تواند با هر نوع داده‌ای برای Id استفاده شود

- **[Key] :**

از این **Data Annotation** برای مشخص کردن کلید اصلی (Primary Key) استفاده می‌شود این ویژگی به **Entity Framework** می‌گوید که این فیلد باید به عنوان کلید اصلی در پایگاه داده ذخیره شود

```
using System.ComponentModel.DataAnnotations;

namespace Section3.WebApi.Domain.Entities;

3 references
public class BaseEntity<T>
{
    [Key]
    0 references
    public T Id { get; set; }
    0 references
    public DateTime CreateDate { get; set; }
}
```

```
[Table("User")]
11 references
public class User : BaseEntity<long>
{
    0 references
    public string FirstName { get; set; }
    0 references
    public string LastName { get; set; }
    0 references
    public string UserName { get; set; }
    0 references
    public string NationalCode { get; set; }
    0 references
    public string Password { get; set; }
}
```

## ❖ ایجاد کلاس User

### ۱- مدل User

- [Table("User")]

این ویژگی از نوع Data Annotation است و یک Attribute است که برای مشخص کردن نام جدول در پایگاه داده استفاده می‌شود
این خط کد به Entity Framework Core می‌گوید که کلاس User باید با جدولی به نام User در پایگاه داده مطابقت داشته باشد

- Inheritance (وراثت)

این الگو یکی از اصول **برنامه‌نویسی شی‌گرا** (OOP) است که در آن کلاس‌ها می‌توانند از کلاس‌های دیگر ارث‌بری کنند و ویژگی‌ها و متدهای آن‌ها را دوباره استفاده یا گسترش دهند

- public class User : BaseEntity<long>

کلاس User از کلاس BaseEntity<long> ارث بری می‌کند.
این ارث بری باعث می‌شود که User تمام ویژگی‌ها و رفتارهای BaseEntity<long> را به ارث ببرد.
در اینجا long نوع داده‌ای است که برای شناسه (Id) کلاس User استفاده می‌شود

```csharp
[Table("Role")]
2 references
public class Role : BaseEntity<long>
{
    0 references
    public string Name { get; set; }
}

[Table("UserRole")]
1 reference
public class UserRole : BaseEntity<long>
{
    [ForeignKey(nameof(User))]
    0 references
    public long User_Ref { get; set; }
    1 reference
    public User User { get; set; }

    [ForeignKey(nameof(Role))]
    0 references
    public long Role_Ref {  get; set; }
    1 reference
    public Role Role { get; set; }
}
```

## ❖ ایجاد کلاس Role

## ❖ ایجاد کلاس UserRole

کلاس UserRole که در اینجا تعریف شده است، به عنوان یک کلاس موجودیت (Entity) برای نگاشت اطلاعات بین دو موجودیت دیگر، یعنی User و Role استفاده می‌شود.

این کلاس در مفاهیم برنامه‌نویسی شی‌گرا و ORM (Object-Relational Mapping) نقش کلیدی دارد

- **[ForeignKey(nameof(User))]**

این خط کد مشخص می‌کند که ویژگی User_Ref به عنوان کلید خارجی مرتبط با کلاس User استفاده می‌شود. این ویژگی به عنوان Foreign Key Attribute شناخته می‌شود و در ORM برای ایجاد روابط بین موجودیت‌ها کاربرد دارد

- **public long User_Ref { get; set; }**

این یک پروپرتی عمومی (public property) از نوع long است که به عنوان کلید خارجی برای اشاره به شناسه کاربر استفاده می‌شود. این نوع از پروپرتی‌ها در طراحی بانک‌های اطلاعاتی برای ایجاد ارتباط بین جداول به کار می‌روند و نمایانگر Association یا Relationship در مدل‌های داده هستند

- **public User User { get; set; }**

این پروپرتی نشان‌دهنده Navigation Property است که در ORM برای پیمایش و دسترسی به داده‌های مرتبط استفاده می‌شود. این نوع پروپرتی‌ها روابط بین موجودیت‌ها را در مدل‌های داده نشان می‌دهند

## ❖ Interfaces

### تعریف Interface

در برنامه‌نویسی شی‌گرا، اینترفیس (Interface) یک نوع خاص از کلاس است که شامل مجموعه‌ای از امضاهای متدها (method signatures) و خواص (properties) است، اما پیاده‌سازی این متدها را ندارد. به عبارت دیگر، اینترفیس تنها مشخص می‌کند که یک کلاس باید چه متدهایی را پیاده‌سازی کند، ولی خود متدها را پیاده‌سازی نمی‌کند. این مفهوم در طراحی نرم‌افزار برای پیاده‌سازی چندریختی (Polymorphism) و توسعه‌ی کد با انعطاف بیشتر استفاده می‌شود

**نکته :** از C# 8.0 به بعد، می‌توان متدهایی را با پیاده‌سازی در اینترفیس‌ها تعریف کرد، تا کلاس‌هایی که این اینترفیس را پیاده‌سازی می‌کنند نیازی به پیاده‌سازی مجدد آن متدها نداشته باشند

**۱- ساخت IUserRepository:**

```
public interface IUserRepository
{
    2 references
    void CreateUser(User User);
    1 reference
    void UpdateUser(User User);
    2 references
    List<User> GetAllUser();
    1 reference
    User GetUserById();
}
```

## ❖ ساخت کلاس ApplicationDbContext

### ۱- نصب پکیج های مورد نیاز برای Entity Framework

- ✓ Microsoft.EntityFrameworkCore
- ✓ Microsoft.EntityFrameworkCore.SqlServer
- ✓ Microsoft.EntityFrameworkCore.Design
- ✓ Microsoft.EntityFrameworkCore.Tools

| Package | Description | Versions |
| --- | --- | --- |
| **Microsoft.EntityFrameworkCore** by Microsoft | Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works with SQL Server, Azure SQL Database, SQLite, Azure Cosmos D... | 8.0.10 / 9.0.0 |
| **Microsoft.EntityFrameworkCore.Design** by Microsoft | Shared design-time components for Entity Framework Core tools. | 8.0.10 / 9.0.0 |
| **Microsoft.EntityFrameworkCore.SqlServer** by Microsoft | Microsoft SQL Server database provider for Entity Framework Core. | 8.0.10 / 9.0.0 |
| **Microsoft.EntityFrameworkCore.Tools** by Microsoft | Entity Framework Core Tools for the NuGet Package Manager Console in Visual Studio. | 8.0.10 / 9.0.0 |

### ۲- ساخت ApplicationDbContext :

کلاس ApplicationDbContext که از کلاس DbContext ارث بری می‌کند، یک پیاده‌سازی رایج از الگوی Context در معماری Entity Framework Core است که وظیفه مدیریت ارتباطات با پایگاه داده و ارائه یک سطح انتزاعی برای کار با داده‌ها را بر عهده دارد
کلاس DbContext در Entity Framework Core به عنوان Unit of Work و Repository عمل می‌کند

- **DbSet<TEntity>**

DbSet نمایانگر یک مجموعه از موجودیت‌ها (Entities) در پایگاه داده است که عملیات CRUD(ایجاد، خواندن، به‌روزرسانی، حذف) را برای یک جدول خاص مدیریت می‌کند

- **DbContextOptions**

این شیء تنظیمات مورد نیاز برای پیکربندی DbContext، مانند رشته اتصال Connection String و نوع پایگاه داده (SQL Server، SQLite) را نگه می‌دارد

- **base(options)**

کلمه کلیدی base سازنده پایه (یعنی سازنده کلاس DbContext) را فراخوانی می‌کند و تنظیمات مربوطه را به آن ارسال می‌کند تا توسط DI در زمان اجرا به کلاس تزریق گردد.

```
public class ApplicationDbContext : DbContext
{
    0 references
    public ApplicationDbContext(DbContextOptions options) : base(options
    {
    }

    2 references
    public DbSet<User> Users { get; set; }
    0 references
    public DbSet<Role> Roles { get; set; }
    0 references
    public DbSet<UserRole> UserRoles { get; set; }
}
```

## ❖ تنظیمات ConnectionString :

### ۱– افزودن تنظیمات ConnectionString در appsettings.json

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=.\\MSSQLSERVER2022; Initial Catalog=VSDB; TrustServerCertificate=True; User Id=sa; Password=          "
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

### ۲– افزودن تنظیمات در appsettings.json

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

- ✓ **AddDbContext** : این متد مسئول اضافه کردن DbContext به Dependency Injection (DI) Container است
- ✓ **ApplicationDbContext** : این کلاس که از DbContext به ارث می‌برد، نماینده پایگاه داده و شامل مجموعه‌های داده (DbSet) و پیکربندی‌های مدل پایگاه داده می‌شود
- ✓ **UseSqlServer** : به EF میگوید که به Sql Server متصل شود

❖ Migration :

```
Package Manager Console
Package source: All    Default project: Section3.WebApi
Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages. Some packages may include dependencies which
are governed by additional licenses. Follow the package source (feed) URL to determine any dependencies.

Package Manager Console Host Version 6.11.0

Type 'get-help NuGet' to see all available NuGet commands.

PM> Add-Migration Initial -c ApplicationDbContext -o Data/Migrations
```

DbContext مورد نظر

محل ذخیره فایل Migration

نام Migration

121 %

❖ به روز رسانی Database :

```
Package Manager Console
Package source: All    Default project: Section3.WebApi
Reverting the model snapshot.
Done.
PM> update-database
```

- VSDB
  - Database Diagrams
  - Tables
    - System Tables
    - FileTables
    - External Tables
    - Graph Tables
    - dbo.__EFMigrationsHistory
    - dbo.Role
    - dbo.User
    - dbo.UserRole
    - Dropped Ledger Tables