**Dependency injection**: Entity Framework contexts should be added to the services container using the Scoped lifetime (once per request).

services.AddDbContext

ASP.NET services can be configured with the following lifetimes:

**Transient**: Transient lifetime services are created each time they're requested. This lifetime works best for lightweight, stateless services.

**Scoped**: Scoped lifetime services are created once per request.

If you're using a scoped service in a middleware, inject the service into the Invoke or InvokeAsync method. Don't inject via constructor injection because it forces the service to behave like a singleton.

**Singleton**: Singleton lifetime services are created the first time they're requested (or when ConfigureServices is run if you specify an instance there) and then every subsequent request will use the same instance. If your application requires singleton behaviour, allowing the services container to manage the service's lifetime is recommended instead of implementing the singleton design pattern and managing your object's lifetime in the class yourself.

**Startup.cs file**:

Startup class configures services and the app's request pipeline.

Can optionally include a ConfigureServices method to configure the app's services.

Must include a Configure method to create the app's request processing pipeline.

ASP.NET Core no longer uses Global.asax, web.config, or App\_Start folders. Instead, all startup tasks are done in Startup.cs

Startup.cs (like old Global.asax) is entry point for application. The Startup class configures the request pipeline that handles all requests made to the application.

Static content in:

wwwroot directory in project is for static content like css, Js, images. And it’s the default root of your server.

**Kestrel** is a cross-platform web server for ASP.NET Core. Kestrel is the web server that is included by default in ASP.NET Core new project templates. If you expose your application to the Internet, you must use IIS, or Apache as a reverse proxy server.

WebListener is also a web server for ASP.NET Core that runs only on Windows.

**Host**: ASP.NET Core apps require a host in which to execute. The host is responsible for application startup and lifetime management. Other responsibility of host’s includes ensuring the application’s services and the server are available and properly configured. Don’t confuse yourself with a Server. The host is responsible for starting the app and its management, where the server is responsible for accepting HTTP requests.

**Razor Pages** is a new feature of ASP.NET Core and it was released with ASP.NET Core 2.0 release. Razor Pages are simple pages or views without controllers and introduced with the intent of creating page focused scenarios where there is no real logic is involved. You will find razor pages similar to ASP.NET Web Forms. They work on the convention and need to be placed in Pages folder and the extension is .cshtml. Razor pages uses handler methods to deal with incoming HTTP request

ASP.NET Core 2.1 introduces supports for SignalR, HTTPS by default

ASP.NET Core MVC 1.0 is Cloud Ready.

MVC + Web API + Web Pages = ASP.NET Core MVC 1.0

ASP.NET Core MVC 1.0 has inbuilt support for Dependency Injection.

There is no App\_Start, App\_Data, Global.asax and root web.config file. App\_Start is replaced by Startup.cs and web.config is replaced by appsetting.json. There is also a new configuration system, where JSON is preferred over XML for configuration settings.

Everything is Nuget package in Core MVC 1.0. There is no dll by default.

**\_ViewImports.cshtml** in Core MVC 1.0 provides namespaces which can be used by all other views. In previous MVC projects, this was provided by the web.config file in the Views folder; since the web.config no longer exists, global namespaces are now provided by this file.

**Tag Helpers** allows to enable server-side code to participate in creating and rendering HTML elements in Razor files.

It also support RESTful style routes with attribute routing. That means you can even declare RESTful like routes like

[HttpGet("Our Route")] and [HttpPost("Our Route")]

a model binder converts posted form values to a model:

public async Task<IActionResult> Create( [**Bind** ("EnrollmentDate,FirstMidName,LastName") ] Student student)

**() => Operator**: (anonymous delegate) with no parameters

() => action.GenerateDescription()

Log.Info("I did something: {0}", () => action.GenerateDescription());

// short-hand for bellow

delegate void () { return action.GenerateDescription(); }

Another Example:

**(a, b) => a + b**

This is roughly equivalent to:

**delegate int (int a, int b) { return a + b; }**

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

**Generic**:

Use generic types to maximize code reuse, type safety, and performance.

The most common use of generics is to create collection classes.

List<T> (In most cases), Dictionary<TKey,TValue>, Queue<T>, ICollection<T>,

Information on the types that are used in a generic data type may be obtained at run-time by using reflection.

**Generic Type Parameters**:

A type parameter is a placeholder for a specific type (constructed type) that a client specifies when they instantiate a variable of the generic type.

**Type Parameter Naming Guidelines**:

Consider using **T** as the type parameter name for types with one single letter type parameter.

For more than one generic type parameter A widely adopted standard **is T1, T2, T3**, etc.

For return values it's also very common to use **TResult** to distinguish it from any other type arguments.

Dictionary uses <TKey, TValue>

Other could be TEntity, TElement, TItem,

public delegate **TOutput** Converter<**TInput**, **TOutput**>(TInput from);

some (old!) naming guidelines do prefix descriptive type parameter names with "T". For example, a parameter constrained to ISession may be called TSession.

public interface ISessionChannel<**TSession**>

{

**TSession** Session { get; }

}

**Constraints on Type Parameters**:

When you define a generic class, you can apply restrictions to the kinds of types that client code can use for type arguments when it instantiates your class.

**where T : class** The type argument must be a reference type; this applies also to any class, interface, delegate, or array type.

**where T :<base class name>** The type argument must be or derive from the specified base class.

Example: public class GenericList<T> **where T : Employee** , IEmployee

C# 7:

#region **Read-only auto-properties**

//compiler can only enforces that the FirstNameOlderVeris not modified from any code outside the class.

public string FirstNameOlderVer { get; private set; } //older version

// in C# 6 You declare the auto-property with only a get accessor

// properties can be set only in the body of a constructor

**public string FirstNameNewVer { get; }**

#endregion

#region **Expression - bodied function members (Imp)**

//expression bodied functions use a lambda expression to define the body of a method or property

public string FullName => $"FirstNameNewVer: {FirstNameNewVer} Jadoon ";

public string GetNameN(string fullname) => $"you sent: {fullname}";

**public void CalculateN(int x, int y) => Console.WriteLine("Total: " + (x + y));**

#endregion

#region **Dictionary Initializer**

//C# 5.0 Dictionary Initializer

public Dictionary<string, Customer> DictionaryOld = new Dictionary<string, Customer>()

{

{"a", new Customer() { CustomerName = "Ron" }},

{"b", new Customer() { CustomerName = "Robert" }}

};

//C# 6 manages the curly braces

public Dictionary<string, Customer> Dictionarynew = new Dictionary<string, Customer>()

{

["a"] = new Customer() { CustomerName = "Ron" },

["b"] = new Customer() { CustomerName = "Robert" }

};

#endregion

#region **Auto-Property Initializers**

//Auto-Property Initializers let you declare the initial value for an auto-property as part of the property declaration.

**public ICollection<double> Grades { get; } = new List<double>();**

**public Guid Id { get; } = Guid.NewGuid();**

#endregion

public Form1()

{

InitializeComponent();

//You can create read-only auto-properties that can be set only in constructors.

**FirstNameNewVer = "Read-only auto-properties Ramis";**

}

private void btnVer6\_Click(object sender, EventArgs e)

{

#region Null-conditional operators/Null Check Operator

//Simply replace the member access . with ?.

string strName = null;

string[] splitNameArray = strName?.Split();

Console.WriteLine("Splited Data FrstName :{0} LastName :{1}", **splitNameArray?[0],** splitNameArray?[1]);

//**var first = person?.FirstName ?? "Unspecified";**

#endregion

#region **String Interpolation / Expression-bodied function members (Imp)**

//simply **use $ "{}"**

Console.WriteLine($"Expression-bodied - Splited Data FrstName :{splitNameArray?[0]} LastName :{splitNameArray?[1]}");

//There aren't any limitations on the expressions you can place between the braces. You can execute a complex LINQ query inside

//Carefull: : is always interpreted as the separator between the expression being formatted and the format string.

#endregion

Console.WriteLine(GetNameN(" Expression - bodied function"));

CalculateN(10, 55);

}

try

{

int index = 10;

int j = (x \* index) / y;

}

**catch (InvalidOperationException exc) when (exc.Message.Contains("unknown error"))**

{

Console.WriteLine("Mail error 1:" + exc.Message);

//return "Site Moved";

}

private void btnVer7\_Click(object sender, EventArgs e)

{

string input = "10";

//declare out values inline as arguments to the method

#region older version

int numericResult;

if (int.TryParse(input, out numericResult))

Console.WriteLine(numericResult);

else

Console.WriteLine("Could not parse input");

#endregion

// C# 7 version

if (int.TryParse(input, out int result))

{

Console.WriteLine(result);

//return result;

} else {

Console.WriteLine("Could not parse input"); //return null; }

**Unit of Work**:

Unit of Work is a design pattern which maintains a list of transactions ,updates the data source with the changes and also provides solution to the concurrency problems. So if the appliation is performing multiple concurrent operations then Unit of Work is the solution. suppose you have to update two different entity types as part of the same transaction, If each uses a separate database context instance, one might succeed and the other might fail.