**C#**

Last Update: Nov 2021.

Versions: C#10, .Net 6.0

1.1 C# runs on .NET which provides a virtual execution system called CLR (Common Language Runtime). CLR is MS’ implementation of Common Language Infrastructure (CLI). C# Source code compiles into an Intermediate Language (IL), i.e assemblies, the assembly can be .exe if it’s an application or .dll if the source code compiles into a library. C# doesn’t import or include anything since all the required libraries are loaded automatically at compile time. Still for semantic reasons we open the System namespace.  
  
Sample Program:  
  
using System;

Class Hello

{

Void main(string[] args)

{

Console.WriteLine(“Yo!”);

}

}

1.2 C# types:   
Integer: Int,long,short,sbyte,byte,ushort,uint,ulong

Decimal: Float, double, decimal (for decimal we append ‘m’ at the end of the number)

Bool: bool

Enum: enum

Characters: String, char

Misc: DateTime

1.2.1 Reference types ⬇

Base type for all reference types: object

User Defined: class, struct, interface, records

Array: […]

Tuple: (…)

Delegate, these are used to pass functions around: delegate <datatype> <name>(…)

1.2.2 Nullable types, can hold null as well: <T>?

1.2.3 Var: The type is resolved at compile time and we can use methods and attributes. Unlike dart, we can’t have multiple types for a single var object and they have to be initialized at declaration.

1.2.4 Dynamic: Just like in dart, this type can hold any type but the information is available only at runtime so methods and attributes aren’t available in compile time, hence they won’t be shown by static analysis tools.

1.3 DateTime: Plenty of methods to work with dates, DateTime.Now for current time.

1.4 Timespan: The difference between 2 DateTime objects.

1.5 [] array: Unlike C++, the size for these arrays can be taken at runtime. Multi-dimensional arrays ([,,])are different from jagged arrays ([][]).

int a = Convert.ToInt32(Console.ReadLine());

int[] b=new int[a]; //is hence valid.

[<one>,<two>,<three>] means array of size <one> which is an element of size <two> which is an element of size <three>.

int[,,] a = new int[4, 3, 2]

{

{

{ 1, 2 },

{ 3, 4 },

{ 5, 6 }

},

{

{ 7, 8 },

{ 9, 10 },

{ 11, 12 }

},

{

{ 13, 14 },

{ 15, 16 },

{ 17, 18 }

},

{

{ 19, 20 },

{ 21, 22 },

{ 23, 24 },

}

};

A[1,2,1] //to access value.

[<one>][<two>] is a jagged array, meaning <one> is an array with <two> arrays.

Int[][] b=new int[2][];

B[0]=new int[5];

B[1]=new int[2];

B[0][1] =2; //to access the position

To insert elements into a List<T>,

Use

Var list= List<int>(){1,2,3,4,5};

Or

Var list=List<int>();

List.AddRange(new []{1,2,3,4,5});

1.5.1 Params: Functions in c# can accept variable number of parameters using ‘params’ keyword. Just like variadic templates in c++.

For example:

public int add(int a, params string[] inps)

{

…

}

add(1,2,3,4,5,); //works

If we instead used normal array,

public int add(int a, params string[] inps)

{…}

add(1,new []{“s”,”b”,”z”}); //would be needed

At runtime, the comma separated parameters are put into a temporary array and passed into the array object. An array object is necessary to work with params.

Just like in c++, a ‘params’ parameter must be at the end of the parameter list.

1.6 Index and Range: From C#8 we get advanced indexes to be used with array types,

Array[2] to get 2nd element.

Array[^2] to get 2nd last element.

Array[^0] to get last element.

Array[2..5] to get an array from 2nd till 5th element (exclusive of 5th element).

Array[^2..^0] or Array[^2..] to get an array from 2nd last till last element (inclusive of last element).

Array[..3] to get get an array of the first 3 elements.

These work only on Arrays, to use with List, use List.ToArray().

1.7 Boxing: Since object is a base class for every type, we can hold values in it then cast them.  
For ex.  
int i=2;

Object o=i; //Boxing, a box with the type info is allocated to hold the value.

i=(int)o; //unboxing. The box is checked and if it is of the casting type, the value is returned.

1.8 out: out passes a variable by reference. Out initialized variables can’t be used without assignment however. Useful to pass variables to functions with multiple value returns. The functions have to assign value to all out initialized variables. Another use for out is declaring variables and passing them as reference using out in the argument.

For example:  
void pp(out int a)

{a=2}

Int a;

Pp(out a); works

And,

class Program

{

static int pp(out int b)

{

b = 2;

return 2;

}

static void Main(string[] args)

{

int a=2 ;

if(pp(out int b)!=0)

{

a = b;

}

Console.WriteLine(b);

}

}

1.9 ref: same as out except, they have to initialized before being passed, which means they can be used without assignment in the called function.

void pp(ref int a)

{}

Int a;

Pp(ref a); works

1.10 Access Modifiers:

1.10.1 Public: Access to all.  
1.10.2 Private: Access to only class.  
1.10.3 Protected: Access to class and children.  
1.10.4 internal: Access to the file.  
1.10.5 protected internal: Protected+internal  
1.10.6 private protected: Protected+private

1.10.7 Const: For constants.

1.10.8 Event: For event types we use this keyword.

1.10.9 Abstract: For abstract methods.

1.10.10 Extern: Declare a method that is implemented externally, i.e another file. Just give it’s signature.

1.10.11 Readonly: Immutable value of the variable, struct and ref. For readonly fields, it means they can only be written at time of object declaration. Same for struct, for ref it means the value will be passed but won’t be mutable.

1.10.12 Virtual: Alllows a method to be overridden by children.

1.10.13 Volatile: Same as in c++, means no optimization on the variable and the value will be read from the memory each time it is needed.

1.10.14 Static: It means that the method or attribute is a part of the class/reference type itself and not of instance.

1.10.15 Unsafe: Requires -unsafe flag to be used, this keyword is used to be able to use pointers.

1.10.16 Sealed: Like final modifier in c++, these types cannot be inherited. For sealed class, class can’t be inherited and for sealed public virtual void pp(){} , pp can’t be inherited.

1.10.17 Override: This keyword is required when we are giving an implementation to a virtual, abstract or implemented method.

1.10.18 In: For generic types <in T>, this is used to implement contravariance.

1.10.19 New: Used to get fresh instance of any object, we can use it to declare methods in child classes with the same name as in base classes.

1.10.20 Out: For generic types <out T>, this is used to implement covariance.

1.10.21 Async: Used to specify that a method is asynchronous.

1.10.22 Init-Only: Init is introduced in c#9 and allows a value to be initialized only in the constructor or the object initializer. The difference between readonly and initonly is that readonly <primitive type> (not fields) doesn’t allow value to be initialized in the constructor, init-only allows that. To use init only,

public myClass{

public string pp {init;}

public string \_pd;

public string pd{

int{ pd= value;}

}

}

init is used in place of set. But instead of allowing property to be changed any no. of times, it limits access to constructor or object initializer only. It also allows ctor to be removed completely.

public myClass {

public string pp{get;init;}

}

var obj= new myClass(){pp=”yo”}; //works and now pp can’t be changed.

1.11 Anonymous types: In C# we can create a type that holds read only values but without all the extra code of classes,structs and records. This is also called an anonymous class, and we don’t need to give type to the variables whilst declaring.

Usage:

Var <name>= new {<var1>=stuff,<var2>=stuff};

<name>.<var1> ; //to get the value

The classes and other types are, in contrast, called ‘real types’.

1.12 ConcurrentBag<T>: In the System.Collections.Concurrent namespace, this collection class is like a Set but is thread-safe and allows multiple threads to use it at once.

Syntax:

ConcurrentBag<T> <varname>= new ConcurrentBag<T>(//can take a List<T> as an argument);

We add items to the CB just like a List and can use other methods like TryPeek (return) and TryTake (return and remove) to return items.

1.13 ArrayPool<T>: In the System.Buffers namespace, this collection is used when arrays need to be allocated and de-allocated very frequently as they would cause Garbage Collector to run more frequently and hence reduce performance. This collection allocates memory and gives parts of that space to arrays to rent and return. It is thread-safe.

Syntax:

ArrayPool can be used in 3 ways,

Shared Pool: Max array length is 2^20 and it is ready to use.

var <varname>= ArrayPool<T>.Shared;

var <varname2>=<varname>.Rent(<int size>);

//returns an Array<T> of <int size> to varname2.

//When <varname2> is no longer needed.

<varname>.Return(<varname2>); //this clears the data put in by <varname2> and //returns the space to the Shared Pool

//or get the Array using var obj=ArrayPool<T>.Shared.Rent(<int>) and return using //ArrayPool<T>.Shared.Return(obj).

1.13.1 Create: This method allows us to create an ArrayPool with custom max array length and max arrays per bucket.

var <varname>=ArrayPool<T>.Create(<int max array length>, <int max array per bucket>); //Here bucket is a pool of arrays that is created automatically for similar //length arrays and helps with faster access. Recommended value is 10.

var <varname2>=…. //same way rent and return.

1.13.2 Custom: Extend the ArrayPool<T> and create a custom impl.

1.13.3 It is imp to note, an ArrayPool instanced Array must be returned and the array object it used (<varname2> here) must not be accessed any more.

1.13.4 Rent method guarantees that an array of given size or greater will be returned but not less, however renting a buffer with size greater than the max array length hammers performance beyond recognition.

1.14 ExpandoObject: In the System.Dynamic namespace, this dynamic type can be used to hold values just like anonymous types but allows any type of Object to be added at runtime (much like a dictionary).

Syntax:

Dynamic <varName>= new ExpandoObject();

<varName>.anything=anyvalue;

1.15 Pre-processor directives:These are used to selectively compile and run different sections of code. Or just for semantic reasons.

#define:

#if-#else/#elif-#endif:

#region <regionName> and #endregion: Any code inside this region is inserted into the documentation system, for semantic reasons the region name starts with ‘snippet\_’.

There are other directives too, check docs.

1.16 C# supports nested classes and functions.

1.17 Class: class <classname> <base class> <interfaces >{}  
Classes and members are private by default, append public to each member and action to make them public. We use ‘this’ to access the instance’s attributes. We can use ‘this’ to access a class’ methods with the object as context.

1.18 Constructors: C# supports constructor overloading. It supports instance and static constructors, instance constructors aren’t inherited.

Static <Classname> {} for static constructor, this can access static fields.

1.18.1 Inheritance: We inherit just like in c++, by using <classname>:<base class>{}. C# doesn’t support multiple inheritance. Classes can implement multiple interfaces though. Interfaces are extended just like other classes. We call base class stuff with base.

1.18.1.1 For interfaces,

Interface PP

{public string oo();

}

Class GG: PP

{

String PP.oo()

{…}

}  
1.18.1.2 For Generic Types, we use  
class <classname> <types> : <baseclass> <types>

And passing values to base class is necessary by using base(<args>);  
public class PP <T,K>

{

public T one;

public K two;

public PP(T first, K second) => (one, two) = (first, second);

}

public class MM<T,K> : PP<T,K>

{

T f;

K m;

MM(T first, K second) : base(first, second) => (f, m) = (first, second);

}

1.19 Indexer: Also called smart arrays, allow classes to overload the [] operator and hence return value from an array container. Can be overloaded as well. The index type is overloaded, can have multiple types per overload as well.

using System;

public class PP<T>

{

private T[] one = new T[5];

public T this [int index]

{

get

{

return one[index];

}

set

{

one[index] = value;

}

}

public (T,string) this[int index, string index2]

{

get

{

return (one[index], "s");

}

set

{

(one[index],\_) = value;

}

}

}

class Program

{

static void Main()

{

PP<int> pp = new PP<int>();

pp[0] = 1;

Console.WriteLine(pp[0,"k"]);

}

}

I have no use of the multi-type overload, but it’s there and this is one way we can use it.

1.20 Properties: Fields are variables in a class, properties are the variables that have getters and/or setters set for them.

1.21 Getter and Setter: By using a getter only or setter only we can set a read only or write only property to a variable.

The basic syntax is,  
class PP

{

Int a{public get; private set;} //the compiler fills in the rest. They are public by default. Called auto-implemented properties.

}

For custom properties we declare separate variables where ‘value’ is given by compiler:

public class PP<T, K>

{

T \_one;

K \_two;

public T one

{

get

{

return \_one;

}

set

{

one = value;

}

}

public K two

{

get

{

return \_two;

}

set

{

two=value;

}

}

}

* + 1. We can also apply default values to properties,

Example:

public int Value {get;} = 2;

1.22 New: In C# new invokes the constructor of the object and returns a new instance. Value types don’t need new but reference types do.  
<classname> O=new <classname>();

1.22.1 We can also use <classname> something=new(<optional args>) to get instance of a class, can be called a shorthand new. [Unknown name for this concept].

<Classname> o=new();

1.23 Default: default(<type>); returns the default value of the type.

1.24 Generic types/ Type Parameters: Classes, structs, interfaces and delegates support generic types.  
We use them like  
  
class Hello <T,K>

{}

Unlike C++, the type isn’t inferred automatically at the call site, we have to explicitly mention it.  
Hello hi=new Hello<int,string>();

using System;

public class PP <T,K>

{

public T one;

public K two;

public PP(T first, K second) => (one, two) = (first, second);

}

class Program

{

static void Main(string[] args)

{

var p = new PP<int, string>(1, "one");

Console.WriteLine(p.one);

Console.WriteLine(p.two);

}

}

* 1. Generic Constraint: We can constraint the type of generic using where.

For example:

Class ABC<T> where T: class,new(){}

And Class ABC<T>: <extending classes/interfaces>: where T: class,new(){}

if we are extending a class.

Here, T must be a class and that class must have a default constructor with no parameters. We can replace type constraints for T with other types as well and add more using ‘,’.

For example:

public abstract class IdentityDbContext<

TUser, TRole, TKey, TUserClaim, TUserRole, TUserLogin, TRoleClaim, TUserToken>

: IdentityUserContext<TUser, TKey, TUserClaim, TUserLogin, TUserToken>

where TUser : IdentityUser<TKey>

where TRole : IdentityRole<TKey>

where TKey : IEquatable<TKey>

where TUserClaim : IdentityUserClaim<TKey>

where TUserRole : IdentityUserRole<TKey>

where TUserLogin : IdentityUserLogin<TKey>

where TRoleClaim : IdentityRoleClaim<TKey>

where TUserToken : IdentityUserToken<TKey>

{…}

* + 1. For methods, we can use generics like so:

static void Swap<T>(ref T lhs, ref T rhs)

{

T temp;

temp = lhs;

lhs = rhs;

rhs = temp;

}

* 1. Deconstructing: Tuples, as created above, (one,two) = (first,second), are deconstructed and their values copied to the same position elements on the left. Just like structured bindings in c++.

We can use it at declaration using, (var a, var b, var c)=(1,2,3) or any function that returns a tuple etc.

We can discard a value using \_, (var a,\_,var c)=(1,2,3) will discard 2.

* We don’t need to declare variables when deconstructing and can also mix the two.

int x = 0;

(x, int y) = 2;

//works

* + 1. For user defined types we overload Deconstruct,  
       so   
       public void Deconstruct(out T first, out K second){}

Now to call deconstruct,

Var p=new PP<int,string>(1,”one”);

(int a, string b)=p; and it will work.

* + 1. We can have multiple deconstructs per user defined type, however they must have different parameter list.
  1. Interface: Just like in dart, interface are used to implement concepts or rather abstract classes. Interfaces can create definitions but not body (until c# 8, after c# 8 they can even give body). Interfaces also support inheritance, even multiple inheritance. Interfaces can only have public members. But unlike abstract classes, classes which inherit interfaces cannot be directly used to call the implemented methods. Instead we upcast the class object into an interface object and then we can call the methods implemented in the class through the interface object, aka Dependency Injection.

For example:

Interface A{

String pp();

}

Class A2: A

{

String A.pp(){…}

}

//now to call pp();

A aObject=new A2();

aObject.pp();

Interface <interfacename>: <base interface>{} for an interface to inherit an interface.

* 1. Async: We can make a function async and then await it’s result. Like dart has Future<T>, we have Task<T> here.

Syntax:

<access modifier> <static/instance> async Task<T> <name>(<params>){…}

Any T it returns will be a Task<T>.

* + 1. We can make Main async by having async and returning just Task.
    2. For an async method stuff(), we can store type of either the Task<T> of it or the T,

Var a=stuff(); stores type Task<T>

Var b=await stuff(); stores type T.’

Await a; //consumes a and converts it into T.

a.Result; to get a’s value.

* + 1. If an exception is thrown in stuff() it is added to AggregateException.InnerExceptions collection of the Task object. That is, any number of exceptions thrown in a task are collected and passed at once outside, which is when the Task is consumed at await. If we process an exception inside then it isn’t added to this collection.
    2. Await Task.Run(()=><method>()); runs a task parallely on a separate thread.
    3. Task Parallel Library is used for CPU bound async tasks, whereas raw await async which execute on the same thread are used for I/O bound async tasks.
    4. Just like dart there is, await Task.WhenAll(many tasks), await Task.WhenAny(…) etc.
    5. Async void should be rarely used (except for event handlers) since exceptions thrown inside cannot be caught outside.
    6. 
    7. ValueTask<T> is another type which can be used instead of Task<T>, however it can be faster if the function can return a synchronous code (i.e. doesn’t lag the performance in allocating Task<T> for a synchronous function), but the function can be asynchronous, in which case it will return Task<T> and then ValueTask<T> will be converted to Task<T> which is more costlier than just simply using Task<T>.
    8. Task Completion Source: It is the source for creating a Task and also the source for that Task’s completion. We use this class to let some instance either set result in the underlying task, or set an exception. It’s like a manager of a Task<T> but more flexible.

To use it,

TaskCompletionSource<int> t1=new TaskCompletionSource<int>();

Task<int> task1= t1.Task;

Task.Factory.StartNew(()=>{ //Start the task in some other thread.

//wait or whatever;

t1.SetResult(10);

//or

//t1.SetException(new SomeException());

});

Stopwatch sw=Stopwatch.StartNew();

int result = t1.Result; //block the execution till it has a result.

sw.Stop()

//print sw.EllapsedMilliseconds.

If SetException is called it will throw it when result tries to obtain the value.

* + 1. SynchronizationContext: It’s a type that provides a virtual Post method which takes a delegate to be executed asynchronously (among many other methods). The base Post method simply calls ThreadPool.QueueUserWorkItem which asynchronously executes the delegate.
    2. await Task.Yield(): Normally, when we have an async method, there’s no guarantee if it’s contents will be executed right at call time (synchronously) or after the other stuff. But if we use this method at the start of an async method, it makes sure that the method will be run after the rest of the process, i.e. asynchronously. It’s not a magic sync-to-async converter, it only delays the execution but the execution occurs on the same thread and when the processing begins the control may not return to other stuff if the code is synchronous hence it may ‘block’ the thread. This method simply delays the inevitable. The difference between this method and Task.Run is that the latter runs the method on a different thread while this does not.
    3. Dataflow: The Task-Parallel-Library provides dataflow components, these are components that help in passing stuff around in an application. In a traditional model we wait for an event or a timer and then execute a method when awaiting a response, like if we have to load images from disk as they become available we may set a timer or design our own event handler all while managing control flow. Dataflow components provide a pipeline that manages all of this, we define method to be executed when the resource becomes available and uses dataflow blocks, these are data structures that buffer and process data. 3 kinds of blocks, source blocks (sender of data), target blocks(receiver ) and propagator blocks (both).
       1. A source block links with a target block, it sends data to it and the target block can accept, decline or postpone the message. It can also decline permanently, in which case the source block unlinks from it. All messages of a source must be processed by targets otherwise a deadlock can occur when it tries sending its next one.
       2. Exceptions in a dataflow block are collected in an AggregateException object. The exception surfaces when we do <dataflow object>.Completion.Wait();
       3. There are a few predefined dataflow block types:
          1. BufferBlock<T>: Simple dataflow block that can act as a source and a target.

var bufferBlock = new BufferBlock<int>();

bufferBlock.Post(1);

bufferBlock.Post(2);

//await bufferBlock.SendAsync(3); to send asynchronously.

Console.WriteLine(bufferBlock.Receive());

Console.WriteLine(bufferBlock.Receive());

Console.WriteLine(bufferBlock.Receive());

prints

1

2

* + - * 1. BroadcastBlock<T>: Unlike BufferBlock, values that are received by a receiver aren’t removed from the datablock, although .Receive() only receives the latest value.
        2. WriteOnceBlock<T>: As the name implies, as soon as a single thread/caller successfully posts a value to this datablock, it prohibits any other caller to write a value. Its receiver acts like a BroadcastBlock, i.e., values read aren’t removed. But a single WriteOnceBlock can only have a single value for its lifetime.

var writeOnceBlock = new WriteOnceBlock<string>(null);

// Post several messages to the block in parallel. The first

// message to be received is written to the block.

// Subsequent messages are discarded.

Parallel.Invoke(

() => writeOnceBlock.Post("Message 1"),

() => writeOnceBlock.Post("Message 2"),

() => writeOnceBlock.Post("Message 3"));

//can print 1 or 2 or 3 and will send that value.

* + - * 1. ActionBlock<T>: Just like BufferBlock, but instead of directly returning the values, this datablock applies a function to the data and the data is made available to be received after the function completes (when the function’s Task is set to Completed state). It accepts a delegate Action<T> or Func<T,Task>. This dataflow block can only act as a source and hence its object doesn’t have a Receive method.

For example:

var actionBlock = new ActionBlock<int>(n => Console.WriteLine(n));

actionBlock.Post(2); //sends data to the action block but doesn’t wait for it to //finish.

actionBlock.Complete(); //Sets the ActionBlock to Completed state and now //the function inside the action block starts processing all the values.

actionBlock.Completion.Wait(); //waits for the ActionBlock to complete //processing all values.

* + - * 1. TransformBlock<T,K>: Just like ActionBlock, except its object can act as a receiver/target too. Also unlike action block, it has another type required, K. K is the return value type. It can be either Task<K> in which case the processing of the function is asynchronous and the function is complete when Task<K> is set to completed state or it can be K in which case it is complete when K type value is returned. This is why it only accepts a delegate of either Func<T,Task<K>> or Func<T,K>.

Syntax similar to Action Block just now the function must return something and its object has a Receive method as well.

* + - * 1. TransformManyBlock<T,K>: Just like TransformBlock except K can be either Task<IEnumerable<K>> or IEnumerable<K>. Then the receiver receives each value of this IEnumerable on each call.

For example:

var transformManyBlock = new TransformManyBlock<string, char>(

s => s.ToCharArray());

transformManyBlock.Post(“Hello”);

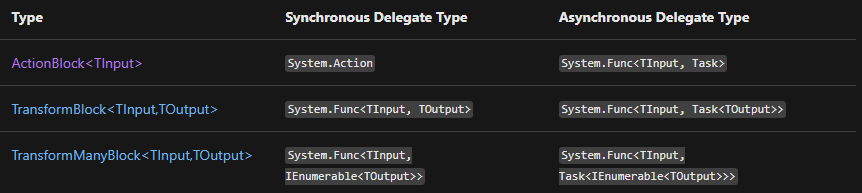
transformManyBlock.Post(“Big”);

transformManyBlock.Receive(); //receives H

transformManyBlock.Receive(); //receives e

…

and so on



* + - * 1. BatchBlock<T>: Takes a given number of values before sending them out as an IEnumerable<T>. If there aren’t enough values and complete is called on it then it will just send however many it has. If there are more values then it will send each ‘batch’ on each Receive call. There are 2 modes, Greedy and Non greedy, greedy is more performant than the latter and is the default. In greedy mode it accepts incoming messages as they come and when the limit is reached it sends them out at once, it non greedy mode it doesn’t accept but postpones accepting of messages until the limit is reached and then it sends them out at once.

For example:

var batchBlock = new BatchBlock<int>(10);

// Post several values to the block.

for (int i = 0; i < 13; i++)

{

batchBlock.Post(i);

}

// Set the block to the completed state. This causes

// the block to propagate out any remaining

// values as a final batch.

batchBlock.Complete();

// Print the sum of both batches.

Console.WriteLine("The sum of the elements in batch 1 is {0}.",

batchBlock.Receive().Sum());

Console.WriteLine("The sum of the elements in batch 2 is {0}.",

batchBlock.Receive().Sum());

prints 45 and then 33

* + - * 1. JoinBlock<T,K> or JoinBlock<T,K,L>: Kinda like BatchBlock, but it only accepts 2 or 3 values before forming their batch. The batch is returned as a tuple. It has greedy/non greedy modes too. The T,K and L classes must implement ITargetBlock<T>. Primitive types implement this by default.

For example:

var joinBlock = new JoinBlock<int, int, char>();

// Post two values to each target of the join.

joinBlock.Target1.Post(3);

joinBlock.Target1.Post(6);

joinBlock.Target2.Post(5);

joinBlock.Target2.Post(4);

joinBlock.Target3.Post('+');

joinBlock.Target3.Post('-');

var data= joinBlock.Receive(); //puts (3,5,’+’) in data

data= joinBlock.Receive(); //puts (6,4,’-‘)

* + - * 1. BatchedJoinBlock<T,K> or BatchedJoinBlock<T,K,L>: A combination of JoinBlock and BatchBlock, returns a Tuple(IList(T),IList(K)) or Tuple(IList(T),IList(K),IList(L)). The BatchedJoinBlock takes a given number of values as well, the tuple has 2 lists and the block accepts the given values and forms the lists after the limit is reached.

Func<int, int> DoWork = n =>

{

if (n < 0)

throw new ArgumentOutOfRangeException();

return n;

};

// Create a BatchedJoinBlock<int, Exception> object that holds

// seven elements per batch.

var batchedJoinBlock = new BatchedJoinBlock<int, Exception>(7);

// Post several items to the block.

foreach (int i in new int[] { 5, 6, -7, -22, 13, 55, 0 })

{

try

{

// Post the result of the worker to the

// first target of the block.

batchedJoinBlock.Target1.Post(DoWork(i));

}

catch (ArgumentOutOfRangeException e)

{

// If an error occurred, post the Exception to the

// second target of the block.

batchedJoinBlock.Target2.Post(e);

}

}

// Read the results from the block.

var results = batchedJoinBlock.Receive();

// Print the results to the console.

// Print the results.

foreach (int n in results.Item1)

{

Console.WriteLine(n);

}

// Print failures.

foreach (Exception e in results.Item2)

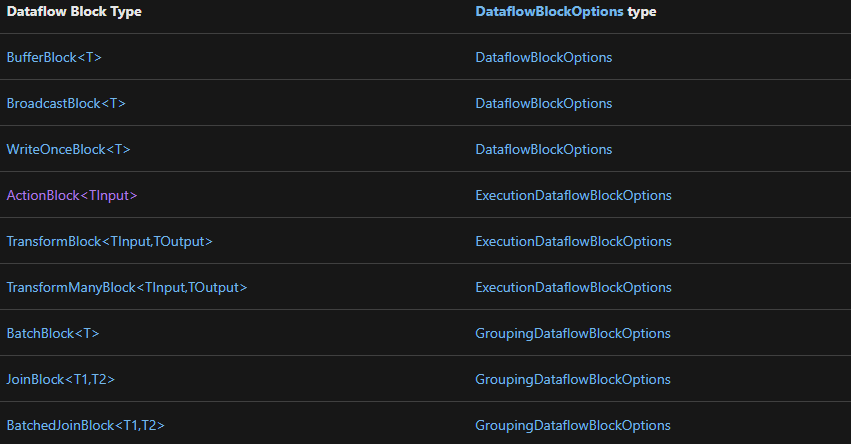
{

Console.WriteLine(e.Message);

}

prints 5 6 13 55 0 then 2 error lines

* + - 1. All the predefined dataflow blocks accept DataflowBlockOptions or its derivative classes. This is used to configure their behavior.



* + - 1. We can configure the default task scheduler, number of messages per task, degree of parallelism,cancellation and greedy/non greedy behavior using the DataFlowBlockOptions.
      2. MaxDegreeOfParallelism: Exclusive to each dataflow block, this property is by default at 1 and can be Unbounded or >1. This defines how many messages should be processed concurrently using different threads. The dataflow block may use a lesser than defined degree of parallelism but never more. Unbounded means it will process the max no. of messages at once, the limit is calculated automatically. Even if a block may process many messages it will emit them in the same order they were received. This option is exclusive, meaning 2 dataflow blocks with 1 degree might still run in parallel to each other.
      3. Number of Messages Per Task: By default set to unbounded, if set to any other value, a single block will only assign given number of messages to each Task object.
  1. Struct: Structures in C# have value semantics, while classes have reference semantics. Meaning, when we store an instance of a struct in a variable, a copy is stored in the variable and the original remains unaffected. If we modify a field of the struct then a new copy of the other values is created and the modified field receives the new value and the variable gets a new struct in it. In classes, since they have reference semantics, the variable holds a reference, it modifies the reference, etc. so copies aren’t created. This also means, when a variable gets a copy of a class instance, it is a reference copy and if it modifies any field then the original variable’s fields also get modified.
     1. Readonly struct: struct can have a readonly modifier to declare that it is immutable, all its data members must be read-only as well. They can be init-only too.

For example:

public readonly struct Coords

{

public Coords(double x, double y)

{

X = x;

Y = y;

}

public double X { get; init; }

public double Y { get; init; }

public override string ToString() => $"({X}, {Y})";

}

* + 1. Methods in struct can be readonly as well, this disallows the method from modifying the instance. However, it can still call non-readonly methods which can modify the instance, in which case the modified instance is copied and original instance remains unaffected. Static methods can’t be readonly.
    2. Even if we use initonly properties, a ‘with’ expression can still modify it at the time of object initialization.

For example:

public readonly struct Coords

{

public Coords(double x)

{

X = x;

}

public double X { get; init; }

}

var p1= new Coords(0);

var p2= p1 with {X=3};

* + 1. C#10 allows structs to have parameterless ctor. Same with initializing field or property at its declaration. Structs can’t inherit other structs/classes or be their parent. They can use interfaces though.
    2. default(<struct type>): will not invoke the parameterless ctor, it will instead assign default value to value types and null to reference types and return that instance.

• Abstract class & method: Abstract classes are those which are inherited instead of implemented. Abstract classes allow abstract methods, which are methods same as interface methods but they can not have bodies even in c# 8.0. And they must be overridden by child classes.  
  
public abstract class PP

{

Public abstract void ok();

}

Public Class hey: PP

{

Public override void ok(){}

}

• Virtual methods: Same as abstract methods but they don’t need abstract classes and can have their own bodies, which can be overridden in child classes.

• Partial: using partial keyword we can split the implementation of a class, struct, method or interface across multiple files.

Public partial class PP

{

Public Int a {get;set;}

Partial void display();  
}

Public partial class PP

{

Partial void display()

{}

}

Works, even if the implementation is in a different file. They should just be in the same namespace. Partial methods can only return void, be private so can’t be virtual and can’t have out .

• Method overloading: Same as in c++.

• C# supports single line or rather single object call functions just like dart.   
public int pp() => 2; pp can be called with pp().

• enums: Same as in c++ but they have access modifiers and use [Flags].  
<access type> enum <enumname>

{}

This works but only if we want single values to be returned.   
  
[Flags]

<enum>{}

Will allow some methods to return multiple values, so   
if   
enum one{a=1,b=2, c=3}

[Flags] enum two{a=1, b=2,c=3}

And we do Console.WriteLine((one.a | one.b).ToString()); we will get 3 as in bitwise or.

But for two.a | two.b we will get (a,b) tuple instead.

• Events:

• Delegates: They are like function pointers in c++. Though they have to be declared like a function first, also they can’t be static but they work for instance methods. Some delegates are provided by default. Like public delegate TResult Func<in T,out TResult>(T arg); and public delegate void Action<in T1,in T2>(T1 arg1, T2 arg2);

The TResult in Func is the return type, Essentially this func<> only takes 1 argument but 2 types, 1st is the type of arg and 2nd is the return type.

class Program

{

static void Main(string[] args)

{

Dp dp = ppo;

Console.WriteLine(dp()); //2

}

public delegate int Dp();

static public int ppo()=> 2;

}

Works.

Here dp can be used to hold reference to any function that has the same signature.

•Multicast delegates: These Delegates can hold reference to multiple methods. And call them one by one, We can add a method or remove it from the invocation list.

class Program

{

static void Main(string[] args)

{

Dp dp = ppo;

Dp dp2 = pp2;

Dp dp3 = dp;

dp3 += dp2;

Console.WriteLine(dp3());

Console.WriteLine(dp3.GetInvocationList().Length);

dp3 -= dp2;

Console.WriteLine(dp3());

}

public delegate int Dp();

static public int ppo()=> 2;

static public int pp2() => 3;

}

Prints 3\n2\n2

• Covariance and Contravariance in delegates: Covariance is allowing a less derived type (base class) to be assigned to a more derived type (child class). We use ‘in’ keyword for this in generics.  
Contravariance is allowing a more derived type (child class) to be assigned to a less derived type (base clas). We use ‘out’ keyword for this.   
Like

Public delegate <T> DD<in K,out T>(K arg);   
This delegate accepts T type or a type that is more derived (child class) of T type. It also accepts K type argument which can be K type or less derived (base class) of K type.

• Generic Delegates: Just like generics in classes, they can be used with delegates and methods.

class Program

{

static void Main(string[] args)

{

Dp<int> dp = Ppo;

Console.WriteLine(dp(2));

}

public delegate int Dp<T>(T pp);

static public int Ppo<T>(T pp)

{

return Convert.ToInt32(pp);

}

}

• Expression: Expressions encapsulate delegates and are processed at compile time. Their use is with LINQ Queries and as such it is in LINQ namespace, if we use normal delegates then they can only perform tasks only at runtime which means SQL queries will fetch entire tables before processing them. But Expressions convert those delegates into relevant instructions at compile time allowing SQL queries to send the instructions and get processed tables.

To use an expression:

Func<int, bool> isEven = s => s/2==0;

is a normal func, to convert it to an expression, simply

Expression<Func<int,bool>> isEvenExpr = s => s/2==0;

// for an action delegate, Expression<Action<int>> printEven = s => //Console.WriteLine(s);

Now to use the expression,

Func<int,bool> isEvenCompiled= isEvenExpr.Compile();

bool result = isEvenCompiled(2); //true is stored

This is a simple way of creating an expression, which converts into an Expression Tree at compile time. The Expression Tree can be manually created but is a tedious process. An expression tree is like a binary tree with constants, return types, param types, etc. as separate types.

• Operator overloading: We overload operators like so,

Public static <return type> operator<op type>(<args>)

{}

Return type is usually the class itself.

• Finalizers: Destructors from c++. Can’t be used in structs, can’t be overloaded, invoked or inherited.

~<classname>(){}

• String interpolation: In C# we operate on strings using $”<string> {variable}”. So int a; string b=$”here a is {a}”; Escape ‘{‘ with another to include it in a string, like “{{“ prints { , same for ‘}’.

• Composite Formatting: String interpolation is a simplified version of composite formatting permitted by String.Format(); This works kinda like how Python does it.

String a=String.Format(“Yo {0} ho {1}”,0,1); works.

We can also have the same index more than once.

Format item syntax: {index,alignment:formatString}. The last 2 are optional but index is necessary.

For ex. {0:X} formats the element as hexadecimal, E for exponent and N for natural number.

{0,-20}, puts the element 20 characters away from the left as the alignment is right aligned and negative values makes it left aligned.

• Verbatim: Used to create raw strings. We can use $ as well.

Verbatim can also be used to make multi line raw strings .

int pp = 2;

String a = @$"You're mom ""ok"" at {pp}";

String b=@”Line 1

Line 2

Line3”;

Console.WriteLine(a);

a and b both work and print the expected output.

• is: Check types.  
• as: Casting just like Convert but doesn’t throw instead returns null. Can’t be used with ReadLine.

• when: When is when. It means to provide a second layer of filtering and stuff to do if that happens.  
It can be used in try catch blocks and switch statements.  
For ex.:

Try

{

Throw;

}

Catch() when IamExpectingThisException()

{dostuff}

Finally{}

Normally, to achieve this we would use if else inside the catch block but that means every exception of the type needs to stop the program and then check it, with this however, it will only stop the program if a check fails. So for an exception that is expected or rather can be dealt with we do this.

• Records: They are like classes but are always immutable (idk what this means because we can change the value of the members). They override the ToString, comparator and other operators by their selves so we can simply put data in a record and call such methods.

Public record Student{

public int id{

Set;

Get;

}

public Student(int test) => Id = test;

}

Student s1=Student(2);

• with: Only used with records to instantiate a new record with the values of an old record with some values changed. We use ‘,’ to separate elements.

Student s2=s1 with {

Id=4

};

• nameOf(<Variable>): Returns the name of variable as string.

• typeOf(<Variable>): Returns the name of type as string. We can store the ‘type’ in a Type object.

For classes and their properties we can use Reflection to get the type.

Type t=typeof(<class>); (or call this.GetType() inside the class)

PropertyInfo elem=t.GetProperty(“<property name>”);

Elem.Name; // will return the type of the property.

• <obj>.GetType(): Returns the type of variable as string.

• ??: Null-coalescing operator, just like in dart, if the value to it’s left is null it returns stuff given to the right.

• Exceptions: We throw exceptions with ‘throw’. All exceptions are instances of a class from System.Exception. To re-throw we throw; from within a catch block, no need for exception argument.

We handle exceptions with, try-catch-[when]-finally

Try

{

Throw new NullReferenceException(“yolo”);

}

Catch (NullReferenceException e)

{//

}

If we print e, we’ll get the exception and our given string. Basically the exception classes are like,

<exceptionName>(<paramName:>,<message:>);

• Lambda function:

<static/instance> <async/sync> (<args>) => {stuff} or <static/instance> <async/sync> (<args>)=> expression; Unlike Python or c++ or even dart, these lambdas suck ass (a little less with c#9 and 10). They have to be assigned to delegates because without delegates they can’t be used. In functions that accept delegates they work. Anonymous methods need lambda operators.

The static keyword can be used to prevent lambda from capturing variables of the enclosing scope (it will still catch static methods and variables).

class Program

{

static void Main(string[] args)

{

int result = 0;

Func<int, int, string> thisSucks = (x, y) => $"{x} + {y}";

no(thisSucks);

}

public delegate int Dp<T>(T pp);

public static void no(Func<int, int, string> pp)

{

Console.WriteLine(pp(1, 2));

}

}

Func<T,K,L> is used to hold an anonymous method with L return type , T and K as the arg types.

To assign an anonymous function to a function that accepts an action delegate, we can do:

List<int> pp=new List<int>{2,3,4};

pp.foreach(delegate(int item){}); // foreach accepts an action delegate, so we can use an anonymous function created with delegate(<T> item){}; Otherwise we can simply use single line functions without a delegate , item => <doStuff>;

i.e, when an argument accepts an Action<T..>(), we can use delegate(Titem){}.

For Func<T,K,Z>(), we can use delegate(Kitem,Zitem){ return Titem;}.

Int sum=pp.Aggregate<int>(delegate(int b, int c){ return a+b;}); //Works, though arrow function doesn’t work here.

more ways to use lambdas (even async ones) here: https://anthonygiretti.com/2021/05/01/c-make-your-delegates-asynchronous-from-synchronous-delegates/

•StringBuilder: As learnt in dart, unless a string needs to be displayed , all operations should be performed on stringbuffers. In c# they are called StringBuilders and are there for the same reason.

Usage:

Var x=new System.Text.StringBuilder();

x.AppendLine(“stuff”);

x.toString();

• Loops: for, while, do-while and foreach( var x in y){}.

• Casting: Implicit conversion means a less derived class can hold the value of a more derived one. Like Long can hold int values and so on.  
Explicit conversion means we need to use methods like Int64.TryParse() or Convert.ToInt64();

C-style casts work as well.

• Query keywords: C# support SQL type query keywords to be performed on containers. This is part of LINQ.  
from, where, select, group, into, orderby, join, left, ascending, descending, on, equals, by, in.   
Meh I’ll read up on them if I have the time.  
But for now, this is a simple way to use them:

using System;

using System.Collections.Generic;

using System.Linq;

class Program

{

static void Main(string[] args)

{

List<int> one = new List<int>() { 1, 2, 3 };

IEnumerable<int> two =

from elem in one

where elem > 2

select elem;

foreach (var elem in two)

Console.WriteLine(elem);

}

}

• Object Initializer: We can use ‘with’ with objects but we can also do it directly to initialize objects.  
Like:

Public class A

{

Int a;

}

A one=new A(){a=2};

A two=new A{a=3}; //This works as well. We can bypass constructors as well.

If A was a record we could do this,

A a=new A with {a=4};

• System.Collection.Generic: Different types of data types that are already built. Non-generic types like ArrayList are in the System.Collection namespace. Non-generic types also have most of these types with the same names. Thread-safe collection types also exist for most of them but have different names. Refer to official doc.  
List<T>: Normal List.

ArrayList: Any type dynamic list. Deprecated in favor of List.

Array: Same as arraylist but only for single type, deprecated.

Queue<T>

Stack<T>

LinkedList<T>: Doubly linked list

HashSet<T>

SortedSet<T>

Observable<T>: For events

Dictionary<T,K>

• Method Group: It is the name for a set of methods with same name, i.e. method with method overloading. Using overload resolution (deciding which method is invoked in an overload) the compiler can convert a method group to a method call.

•Conversions:

Char to int: char.GetNumericValue;

String to char: char.Parse; or char.TryParse; //though why would we, just string[elem] works.

Char to string: char.ToString();

• LINQ: Language Integrated Query. This is kind of like Algorithm in c++ except it doesn’t have algorithms but many useful methods, it also includes SQL Queries. Some of the useful methods are:

Char.IsDigit(<char>): char is digit or nah

String.All(char.isDigit): String only has digits or nah

String.Any(char.isDigit): String has any digit or nah

<dict>.First(x=>x.Value==<value>).Key: returns first key where the value is same to the provided value.

• StringReader: Does what its named, reads a string, used to read string line by line.

String a=@”Line1  
Line 2  
Line 3”

StringReader b=new StringReader(a);

String? Item=b.ReadLine();

While(item!=null)

{

Console.WriteLine(item);

Item=b.ReadLine();

}

Works.

• using: Unlike C++, the using declarative here has a bit more power. It can be used to create blocks with classes which expose IDisposable or IAsyncDisposable as it automatically calls their Dispose method at the end of the scope. We can use using without the braces from c# 8.0.

Using(var a=new StringReader(“Lol\nas\n”))

{

Do stuff with a.

}

* Using static <namespace>; All static methods and props in the namespace can be accessed in the file without mentioning the class/struct name.
* Global using <namespace>; The namespace is opened for all the files in the compilation, given the files are used in this file. Can use static modifier, and should appear before other using statements. Typically placed in a source file.
* Type Alias: using name=<namespace/object/etc.> can alias types, namespaces and other defitions much like typedef in c++.

• Namespace: namespace MyName{…} , or namespaces for the same file (as in C++) namespace {…}

* namespace Mynamespace; Just writing this line at the top of a c# file means all the classes/methods/props etc. are part of this namespace.

• Extension methods: C# supports extension methods just like dart. To implement extension method on instance of a class , simply create a static class and put a method in it with it’s parameters being ‘this <classname> object,<…params>’

Class ABC{…}

Static class XYZ

{

Public static void Method1(this ABC abc){…}

}

ABC abc=new ABC();

//And we can call,

abc.Method1(); //and this will work

• Environment: Use this object to access environment properties. Like Command Line Args, current directory etc.

• Timer: System.Timers.Timer to use a timer, now for periodic functions we use Timer.Interval. But there’s an alternative to that, sync + async function assigned to a different thread <https://stackoverflow.com/questions/22449518/how-to-use-async-and-await-in-timer/22453097> >

namespace ConsoleApp1

{

sealed class Program

{

void run()

{

CancellationTokenSource cancellation = new CancellationTokenSource(

TimeSpan.FromSeconds(8));

Console.WriteLine("Starting action loop.");

RepeatActionEvery(() => Console.WriteLine("Action"),

TimeSpan.FromSeconds(1), cancellation.Token).Wait();

Console.WriteLine("Finished action loop.");

}

public static async Task RepeatActionEvery(Action action,

TimeSpan interval, CancellationToken cancellationToken)

{

while (true)

{

action();

Task task = Task.Delay(interval, cancellationToken);

try

{

await task;

}

catch (TaskCanceledException)

{

return;

}

}

}

static void Main(string[] args)

{

new Program().run();

}

}

}

• Environment.FailFast(String): Used to terminate the application immediately, there is a memory dump created and windows event viewer receives an event as well.

**Entity Framework**

An O/RM (Object-Relational Mapper) which is basically a framework that helps us write database code in c#. It’s a replacement to ADO.NET, since ADO.NET had all the conversion manually done by developers whereas EF just does that by itself.

•DbContext: In the System.Data.Entity namespace exists the most basic class needed for EF methods. Can be installed through NuGet PM in VS2019. We use this abstract class to extend our own Database context classes.

Like:

public class SContext: DbContext

{

public string connStr;

public SContext(DbContextOptions<SContext> options, string connString):base(options)

{

connStr=connString;

}

public DbSet<Student> Students { get; set; }

public DbSet<StudentAddress> StudentAddresss { get; set; }

}

Where Student and StudentAddress are 2 different classes. These are called Entities.

Here, DbContextOptions are used by EF internally to create connection or database itself based on given parameters. DbContextOptions are necessary for this class, however there are 2 ways of creating it.

1. Pass DbContextOptions through constructor. To pass DbContextOptions to constructor we can use .net core’s Dependency Injection, i.e., services.AddDbContext<SContext>(options=>options.UseSqlServer(“<connStr>”)); or we can create it manually,

var optionsBuilder= new DbContextOptionsBuilder<SContext>();

optionsBuilder.UseSqlServer(“<connStr>”);

var db= new SContext(optionsBuilder.options, “<connStr>”);

1. Override the OnConfiguring method. As it is visible, we pass a connection string but never use it. Normally, we either pass connection string and override onConfiguring method or pass options only.

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder){

if(!optionsBuilder.IsConfigured)

optionsBuilder.UseSqlServer(<connStr>);

}

• entity: From DbSet<TEntity>, basically when we declare a DbSet with TEntity type, the class that is TEntity is the table. And any properties in the class become the columns.

public class Student

{

// scalar properties

public int StudentID { get; set; }

public string StudentName { get; set; }

public DateTime? DateOfBirth { get; set; }

public byte[] Photo { get; set; }

public decimal Height { get; set; }

public float Weight { get; set; }

//reference navigation properties

public Grade Grade { get; set; }

}

public class Grade

{

public int GradeId { get; set; }

public string GradeName { get; set; }

public string Section { get; set; }

public ICollection<Student> Students { get; set; }

}

• Migrations: Migrations define how the database model changes in each revision, we can directly modify the database to have a structure/ change in structure but migrations make it programmatical and also help undo any changes we don’t want.

Add-migration <optional configuration class location> <migration name>: adds a migration by creating a .cs and a .Designer.cs file, both define the new change. The .cs file contains 2 methods in a class that extends Migration class, Up and Down. Whatever we change to the db is added in the up method and whatever change up is doing is negated in the down method. Both methods provide a MigrationBuilder object that is used to modify the db. We can create our own changes as well, such as creating a Stored Procedure and deleting it and then passing the string containing the Sql query/procedure to migrationBuilderObject.Sql(<String>); All migrations are added sequentially, i.e any new migration added is added at the end of the folder.

If optional configuration class is provided, i.e., there are multiple dbs and we want to only add migration of a specific db we specify it’s configuration class’ location.   
For example:   
Add-Migration -Configuration Somenamespace.somefolder.Configuration SecondDbMigration1.

It works the same way for remove and update as well.

Remove-migration <optional configuration class location> : Removes the last migration from the migrations folder if it is unapplied/reverted.

Update-database <optional configuration class location> <migration-name>: Migration name is optional, if it is provided then updates the database to match the migration. If the given migration is new and unapplied then from the migrations folder all the unapplied migrations till the given file (including the given file) are added, the Up method of these files are called. If the given migration has already been applied and there are more migrations after this one then those migrations are reverted (excluding this file), the Down method of all these files are called. If the given migration name is the current applied migration, then nothing happens. If migration name is not provided then all the unapplied migration files are applied.

• Automatic Migration: We can set ef to automatically migrate on finding changes in the code.

Syntax:

In PM console:

enable-migrations -EnableAutomaticMigrations:$true

Then we can add entries to the new database using the seed method inside Configuration.cs in migrations folder.

Then in DbContext class’ constructor,

Database.SetInitializer(new MigrateDatabaseToLatestVersion<DbContextClass>, Configuration);

• Enabling migration for a configuration: By default, any db will have the same migration folder. So if we run update database then it will apply migration to the other dbs as well. To avoid that, we create a different migration folder and configuration file.

Enable-Migrations -ContextTypeName SomeNamespace.somefolder.TheDbContextClass -MigrationsDirectory SomeFolderToStoreThisDbMigrations

This will generate a folder and put a new configuration.cs in it. Any migrations done to this configuration will come in it’s folder.

This Configuration is the class in the Configuration.cs.

• dotnet ef <databasename> drop to drop(delete) a database.

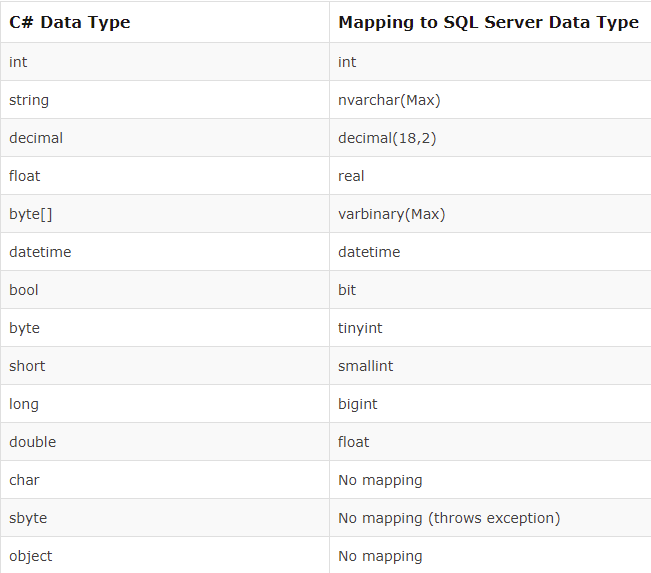
•Scalar and Navigation properties: All properties of an entity that are of primitive type are scalar properties, meaning they make columns in the table. If an entity has another class as a property then that instead creates another table and maps 1 to many relationship between the first and the second entity by creating a ForeignKey in the first entity automatically. This is called a reference navigation property. Collection Navigation Property is the 2nd type of Navigation property and it is created when an entity has a generic collection of an entity type.

• POCO and DynamicProxy: 2 type of entities, the ones above were POCO(Plain Old CLR Object) .

Dynamic Proxy entities are the POCOs that have a class with public access, non-abstract, navigation properties public virtual and context.Configuration.ProxyCreationEnabled=false; set in the Context class.

•Methods: Many methods such as Add, Remove, Update, Set (like insert) and the most important, SaveChanges. Called on context object (object of the class extending DbContext).

• Defaults: By default EFcore will create a database with the name of the entity + ‘s’/’es’ , “Students” for Student class for example.   
Any property in an entity having ‘Id’ in any case will become its primary key.  
Any property with <reference property name> + ‘id’ (in any case) / <reference property name> + <reference entity’s primary key name> will become foreign key if a reference property with <reference property name> exists.  
Only properties with getter and setters defined will be mapped to the database.  
If Database name is not provided within the base constructor call when DbContext is extended, EF creates the db with {NameSpace}.{Classname}.

• Mappings: 

• One-To-Many Relationships:  
The following naming schemes will create one to many relationships from class A to B.

1. Class A

{

Public B B{get;set;}

}

Class B

{

}

1. Class A

{

}

Class B

{

Public ICollection<A> As{get;set;}

}

1. Class A

{

Public B B{get;set;}

}

Class B

{

Public ICollection<A> As{get;set;}

}

1. Class A

{

Public int Bid{get;set;}

Public B B{get;set;}

}

Class B

{

Public int Bid{get;set;}

Public ICollection<A> As{get;set;}

}

•One-To-One: Only 1 way to create this type of relationship.

Class A

{

Public int Aid{get;set;}

Public B b{get;set;}

}

Class B

{

Public int Aid{get;set;}

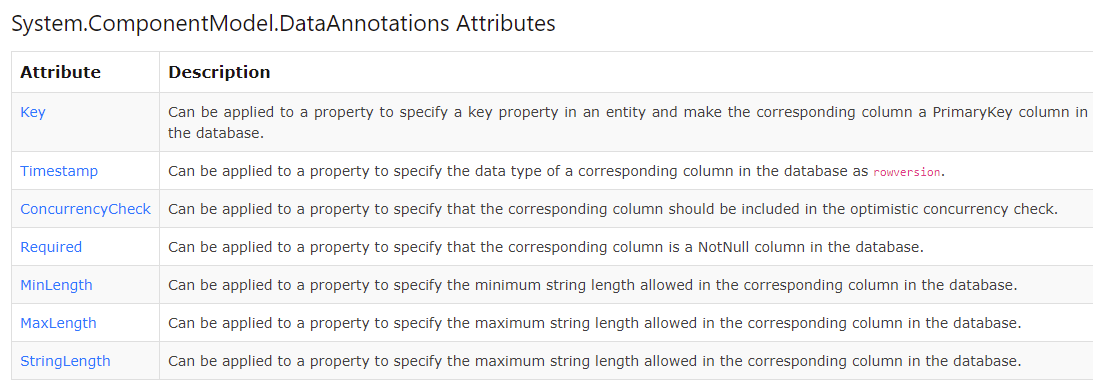
Public A A{get;set;}

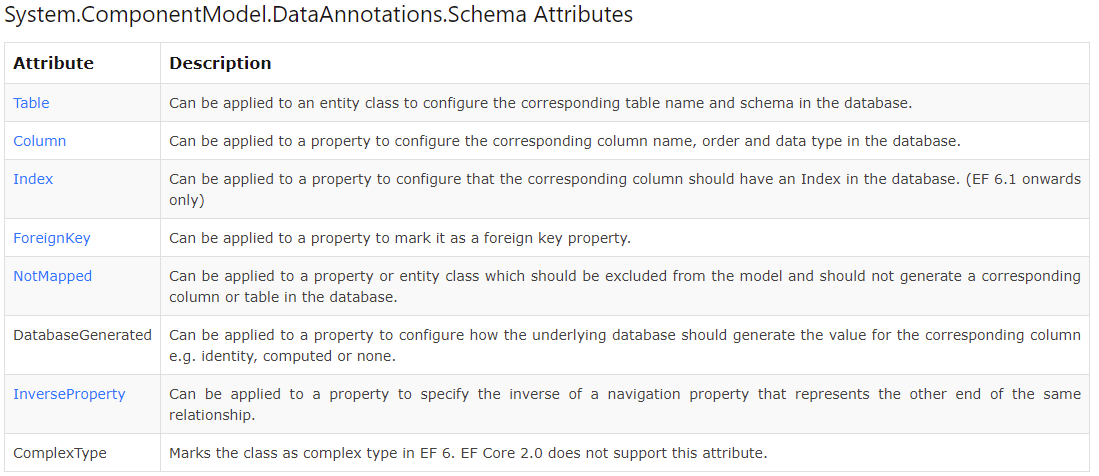
}

• Data Annotation Attributes: Stuff within [ ] , these are .Net attributes which can be provided to manually override ef defaults.

All under the System.ComponentModel.DataAnnotations; and System.ComponentModel.DataAnnotations.Schema; namespaces.

They are:





Used like this,

[Table(“StudentTable”,Schema=”admin”)]

Public class Student{}

Now when Student entity will be used by the EF to create table, the custom name will be given to it in the db.

The schema property is optional but if given, gives the admin schema to StudentTable.

[Key] gives primary key attribute to a property, if multiple keys are given then composite primary key is created instead.

•Fluent API: We can use EF’s Fluent API to override default conventions, much like annotations but more powerful. EF prioritises Fluent API over Annotations.

We override the following method of the DbContext for the same,

**protected override void OnModelCreating(ModelBuilder modelBuilder)**

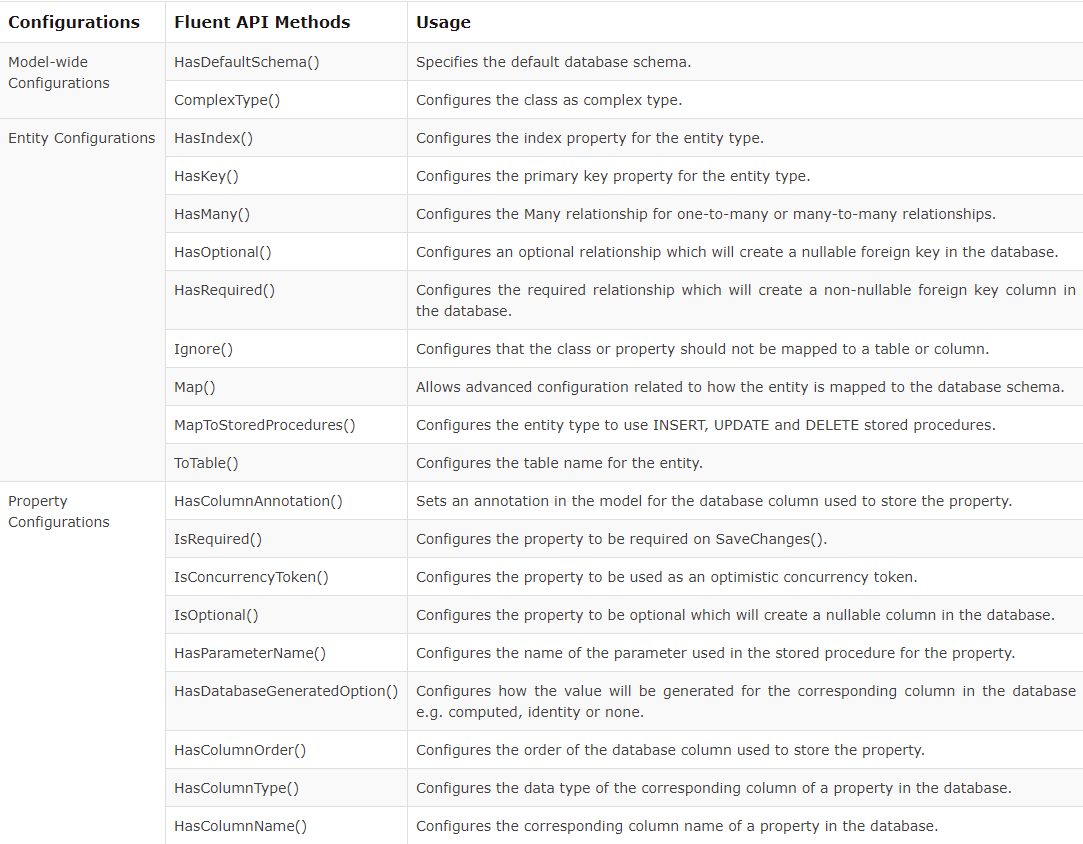
{

//Write Fluent API configurations here

}

1. …modelBuilder. HasDefaultSchema("notdbo"); to change schema name. By default schema name is ‘dbo’.

• FluentAPI Methods:



We can use them like,

modelBuilder.HasDefaultSchema(“Admin”); sets the default schema to the given string for all entities.

modelBuilder.Entity<TEntity>().ToTable(<Tablename>,<optional schema name>); is the same as Table() annotation.

• CRUD: Create, Read, Update and Delete for EF is like follows:

For

Class A

{…}

Class B : DbContext{…}

B context=new B();

A a=new A();

**Create:**

Context.Add<A>(a);

Or

Context.add.Departments(a);

Context.saveChanges();

SaveChanges() actually applies the stuff to the db;

**Read:**

IEnumerable<A> values= Context.A.toList(); //gets all A

**Update:**

2 ways to go about it.

Connected approach, i.e when the scope is same for fetch and update.

A oldA=context.A.Where(x=>x.name==”Ooga”).FirstOrDefault();

Or   
A oldA.context.A.Find(<TiD>); finds the db object with key == TiD, i.e TiD type and keyType should be same.

oldA.name=”lol”;

context.SaveChanges(); //works and updates oldA in db.

Disconnected approach, i.e when a given value is not fetched from the db or is fetched in another scope.

Void doStuff(A a)

{

Context.A.Update(a);

Or

Context.Update<A>(a);

}

**Delete:**

Same as in update, there is connected and disconnected approach.  
Connected:

A oldA=context.A.Find(2);

Context.Remove<A>(oldA);

Disconnected:

Void doStuff(A a)=> context.A.Remove(oldA);

• Types of Loading for connected objects, i.e tables connecting other tables: There are 3 main ways to load full connected objects, i.e if we query context for TEntity and want any T2Entity inside it, like another class we have to specify loading techniques.

For:

Class A

{

List<A2> aObjects{get;set;}

}

Class A2

{

A aObject{get;set;}

}

Class B : DbContext{…}

B context=new B();

A a=new A();

Var newObjs= new List<A2>{new A2(), new A2(), new A2()};

a.aObjects=newObjs;

for(int i=0;i< a.aOjbects.Length;++i)

a.aObjects[i].aObject=new A();

context.Add<A>(a);

**Eager Loading:** Loading all the connected objects at once. Gets all the objects plus the ‘included’ objects present in the object.

IEnumerable<A> aObj=context.A.Include(x=>x.aObjects).toList();

**Explicit Loading:** Loads the single entity into the object given in Entry(<obj>). Unlike eager loading it queries for each entity separately and when requested.

IEnumerable<A> aObjs=context.A.ToList();

Foreach(var obj in aObjs)

context.Entry(obj).Collection(x=> x.aObjects).Load();

foreach(var obj2 in obj.aObjects)

…

when the object in the object is not a collection type we use:

IEnumerable<A2> a2Objs=context.A2.ToList();

Foreach(var obj in a2Objs)

Context.entry(obj).Reference(x=>x.aObject).Load();

Var aObj=obj.aObject;

…

**Lazy Loading:** I think it is the default type of loading now and is the most cleanest since there is no special command to fetch the queries, instead they are lazily loaded automatically when they are requested. Just like explicit loading but automatic now.

To use this we simple specify this in the onConfiguring method of our class that extends DbContext.

optionsBuilder.useLazyLoadingProxies();

and now we don’t need to load explicitly as it is done automatically.

IEnumerable<A> aObjs=context.A.ToList();

Foreach(var obj in aObjs)

Foreach(var obj2 in obj.aObjecs)

…

//works

• Modes of SQL Query Execution:  
  
**Immediate:** In immediate mode of execution, the database is hit for a query right at the time of object creation, i.e if context.T.tolist() is called then all the entities of T will be requested from the database right then . Any method that relies on data to be present in the object containing the entity requests data from the sql server.   
the toList() method returns a list of all the entities so it is an immediate mode of execution. IEnumerable<T> (.toArray()) and List<T> (.toList()) can be used to hold the values obtained through immediate mode of execution. The difference between IEnumerable and List is that IEnumerable is immutable whilst list is not.

**Differed:** In differed mode of execution, the database is hit for a query whenever data is needed. Unlike toList which requires all data, differed does not need any data in the container. But on the downside every time any query is performed it hits the db again.

Using no method on context.T; returns an object that can be stored in IEnumerable<T> or IQueryable<T> . The difference between IQueryable and IEnumerable is that IQueryable performs any query on the sql server so the sql server returns data after performing the query whereas in IEnumerable all data is requested and then query is performed on the client. But for both the cases the requests are only performed when the query is made.

• Raw SQL using EF: We can use raw sql in ef using the fromSQL(<query>, <params>) method or the ExecuteSqlCommand(<query>,<params>) method on an entity in a context. FromSQL() is used to when query returns something, and ExecuteSQLCommand() is used when there is no value to return.

FromSQL() has a few limitations and the one most basic limitation is that the query must return the same number of columns as properties in the entity class that holds it.

Ex.  
SqlParameter param=(“@id”,2);

Var aList= context.T.FromSQL(“Select \* from Books where id=@id”,param);

Or

Var aList=context.T.FromSQL(“Select \* from Books where id={0}”,2);

//We use the SqlParameter object to avoid sql injection.

• Stored Procedure: We can use FromSQL() and ExecuteSQLCommand() to execute Stored Procedures by simply calling them inside these methods, like context.Books.FromSQL(“exec GetAllBooks”).toList();

We can create Stored Procedures by creating string in the Up Method holding the SP and passing it to migrationBuilder.Sql(<string>); And remove the same by creating a “Drop <SP name>” and passing it to the migrationbuilder in Down.

• QueryBuilder: SSMS Can be used to generate queries by using the UI.

• Transactions: Transaction are a group of SQL queries. By default ef uses saveChanges() to manage transactions, if there is an error in a query before the saveChanges() is called then the transaction is rolled back, i.e no changes are applied to the db. We can have manual transactions by using the Database.BeginTransaction method on the context, it extends the IDisposable class so we can use using. These manual transactions can hold multiple default transactions. It is also preferred to use try catch blocks so that rollback can be applied.

For example:   
var context=new BookContext();

(Using var txn=context.Database.BeginTransaction())  
{

Try{

Context.T.Add(…);

Context.saveChanges();

Context.Y.Add(…);

Context.saveChanges();

Txn.commit(); //commits the transaction.

}

Catch(exception e)

{

Txn.rollback();

}

}

Here even though saveChanges would apply changes to the db, if there is any exception then they are rolled back to match the state at the start of the scope.

• Concurrency Conflitcs: Concurrency conflict occurs when a user makes change to a row of a db but before he hits save another user fetches the row on a different client. He has the old state and when he updates the updated row he will overwrite the changes of the first user.

To manage concurrency conflicts:

1. Use row locks. Causes performance loss and is complex to implement, also not supported by Entity Framework. Called Pessimistic Concurrency.
2. Optimistic Concurrency: Allows conflicts to happen and decides which one wins. Has 3 types:

Track changes and update non-conflicting columns.

Client wins, the last update wins.

Store wins, the first update wins and until it is complete the later clients are alerted.

Using ConcurrencyCheck attribute isn’t recommended, instead TimestampAttribute and isRowVersion are recommended.

To implement store wins:

Model Class:

…

[Timestamp]

Public byte[] ConcurrencyToken {get;set;}

which generates

b.Property<byte[]>("ConcurrencyToken")

.IsConcurrencyToken()

.ValueGeneratedOnAddOrUpdate()

.HasColumnType("rowversion");

In modelBuilder.

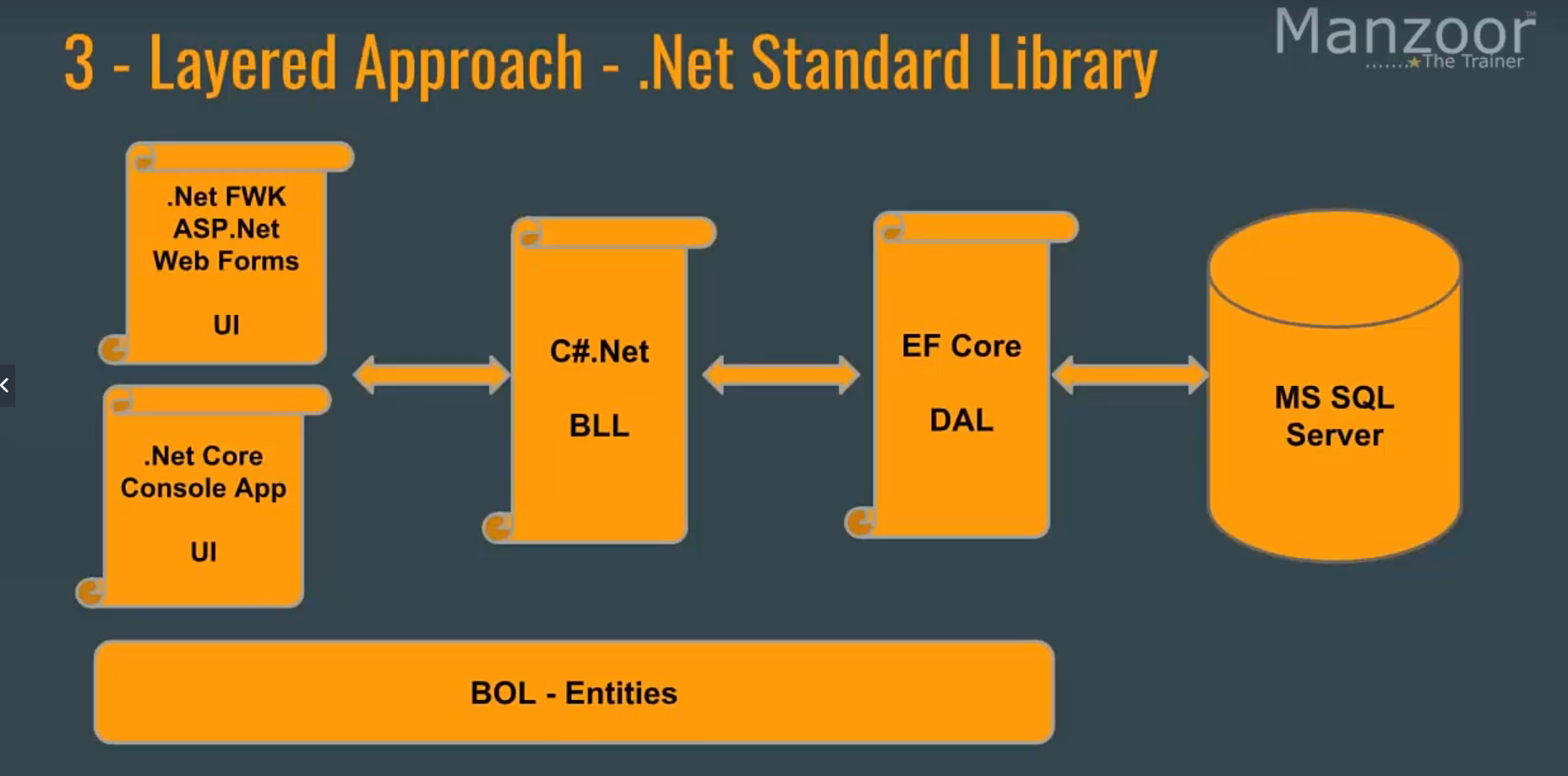
Or

DbContextclass’ onModelCreating:

modelBuilder.Entity<Model>().Property<byte[]>(“ConcurrencyToken”).IsRowVersion();

EF throws a DbUpdateConcurrencyException when another user updates the row that was already retrieved as each user to retrieve the row gets a token and until that token is presented back it will not permit changes.

More changes specific to SQLite or SQLserver are needed, refer docs for the same.

•

What it really means is,

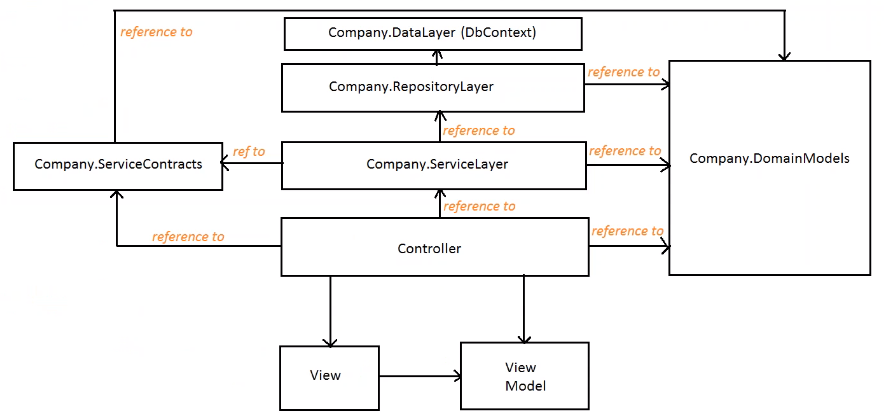
1. Have a db.
2. Access the db using a Class Library (call it <soln>.DAL) . References BOL.
3. Create models and have them in <soln>.BOL CL.
4. Create a BLL <soln>.BLL ,i.e., a class that requests data from DAL and also passes to it. References BOL and DAL.
5. Finally create the MVC class, set it as startup project. References BOL and BLL. The models inside are actually ViewModels since they are used by the views.

There’s also Service Pattern. The only difference is that instead of the BLL we have an interface project and then a project that has a class that implements it. We call this class in the MVC.

And lastly there’s Repository Pattern.

As it was with Service Pattern, this pattern also adds onto the previous pattern ,i.e., the Service Pattern.

Here we divide DAL into 3 classes, 1st layer is the core DbContext project, the second layer is an interface for accessing and performing queries on the dbcontext project and the third layer is the repository class that implements the second. Then we simply access the interface in the service layer and the service layer’s interface in the MVC. We implement Dependency Injection for the repository classes in the MVC as well.



This is the final structure (With repositoryContracts interface just above RepositoryLayer but missing from the figure).

1. **ASP.Net Core**

Current .Net core is Microsoft’s .Net 5.0 and later on. There used to be .Net core up until .Net core 3.1, it existed separately from .Net, but from .Net 5.0 it is now integrated into .Net. In other words, .Net core is no longer updated, it’s .Net itself but the features of core are available in it as well.

ASP.Net is MS’ AIO software development framework, it can create front-end using Blazor (for Client-sided web UI) and Razor Pages (for server-sided web ui) based web apps or used in conjunction with popular front end technologies as their backend creating RESTful HTTP Services Web API, or as RealTime app or as RPC app.   
That is, it is a full-stack in itself but can be used to fill in parts as well.

ASP.Net Core is the name for ASP.Net 5.0 and forward.

* 1. ASP.NET core uses Kestrel server by default to handle requests. It can use either in-process(same as IIS worker)/out-of-process(not same) hosting model to host its Kestrel server. The alternative to Kestrel is Http.sys which has different features but it is only available on windows. We can check the server type, running port etc. at runtime using <IApplicationBuilder object>.ServerFeatures.
  2. We can write custom server as well, they must at minimum implement IHttpRequestFeature and IHttpResponseFeature.
  3. launchsettings.json isn’t used by the app itself but by the debuggers that launch it. We can manually give a launch profile using –launch-profile <name> to dotnet run, given the <name> exists in launchsettings.json..
  4. Host: On startup, asp.net core app builds a host. The host starts and contains all the resources of the app like HTTP Server, Middleware Components, Logging, DI services and Configuration. The reason a single object holds all the app’s interdependent resources is lifetime management, because of which app can have controlled startup and graceful shutdown.

3 types of Hosts:

.NET Generic Host

ASP.NET Core Web Host

WebApplicationBuilder

Minimal API is the modern and recommended way of working. It covers 98% of features of generic host but is faster. The WebApplicationBuilder is a part of Minimal API. In terms of configuration, WebApplicationBuilder is mostly similar to Generic Host.

To start a generic host:

Main(string[] args) {

Host.CreateDefaultBuilder(args).ConfigureWebHostDefaults(webBuilder=>{

webBuilder.UseStartup<Startup>(); // The Startup is the Startup class.

}).Build().Run();

}

To use WebApplicationBuilder,

var builder= WebApplication.CreateBuilder(args); //args is provided global //namespace.

* + 1. The createdefaultbuilder loads configuration from appsettings.json, appsettings.{Environment}.json, env vars, cli args and loads all other json files, sends logging data to console and debug providers, enables scope validation and dependency validation when env is dev and sets the content root to the path returned by GetCurrentDirectory.
    2. The ConfigureWebHostDefaults uses Kestrel as web server, loads host config from env vars prefixed with ASPNETCORE\_, adds host filtering middleware, conditionally adds forwarded headers middleware and enables IIS integration.
    3. For a non-HTTP workload, we use

Host.CreateDefaultBuilder(args).ConfigureServices((hostContext,services)=>{

Services.AddHostedService<Worker>();

}).Build().Run();

* + 1. In Generic host, we use a CreateHostBuilder static method which returns an IHostBuilder, that is, we call this method in the main instead of directly calling the host. This is for both, semantic reasons and programmatic reasons, the Entity Framework Core Tools expect to find a CreateHostBuilder method.
    2. The host provides IHostAppilcationLifetime, IHostLifetime and IHostEnvironment/ IWebHostEnvironment services.
    3. IHostApplicationLifetime:

Is used to handle post-startup and graceful shutdown tasks.

To use it,

Internal class myLifetimeEventHostedService: IHostedService

{

Private readonly IHostApplicationLifetime \_appLifetime;

Private readonly ILogger \_logger;

Public myLifetimeEventHostedService(IHostApplicationLifetime appLifetime, ILogger< myLifetimeEventHostedService > logger)=> (\_appLifetime,\_logger)=(appLifetime,logger);

public Task StartAsync(CancellationToken cancellationToken)

{

\_appLifetime.ApplicationStarted.Register(OnStarted);

\_appLifetime.ApplicationStopping.Register(OnStopping);

\_appLifetime.ApplicationStopped.Register(OnStopped);

return Task.CompletedTask;

}

public Task StopAsync(CancellationToken cancellationToken)

{

return Task.CompletedTask;

}

private void OnStarted()

{

\_logger.LogInformation("OnStarted has been called.");

// Perform post-startup activities here

}

private void OnStopping()

{

\_logger.LogInformation("OnStopping has been called.");

// Perform on-stopping activities here

}

private void OnStopped()

{

\_logger.LogInformation("OnStopped has been called.");

// Perform post-stopped activities here

}

}

We can use the On.. methods to do something.

* + 1. IHostLifetime controls when host starts and stops.
    2. IHostEnvironment service is used to get env related info.
    3. Host configuration can be used for the properties of the IHostEnviroment implementation. To do so we need to use ConfigureHostConfiguration() method on the IHostBuilder.

For example:

using Microsoft.Extensions.Configuration;

Host.CreateDefaultBuilder(args)

.ConfigureHostConfiguration(configHost =>

{

configHost.SetBasePath(Directory.GetCurrentDirectory());

configHost.AddJsonFile("hostsettings.json", optional: true);

configHost.AddEnvironmentVariables(prefix: "PREFIX\_");

configHost.AddCommandLine(args);

});

We can Configure host multiple times and the changes are additive.

* + 1. App Configuration is created by calling ConfigureAppConfiguration on IHostBuilder.
    2. We can modify settings using both env vars and methods of host (some values can only be set through env vars).

They are:

UseContentRoot("c:\\content-root") To set different content root

UseEnvironment("Development") To set env name

Host.CreateDefaultBuilder(args)

.ConfigureServices((hostContext, services) =>

{

services.Configure<HostOptions>(option =>

{

option.ShutdownTimeout = System.TimeSpan.FromSeconds(20);

});

}); to set shutdown timeout (default 5 secs).

* + 1. ConfigureWebHostDefaults methods:

webBuilder.CaptureStartupErrors(true); //if false errors in startup result in host exiting, if true host captures them and tries to start again.

webBuilder.UseSetting(WebHostDefaults.DetailedErrorsKey, "true"); //if true captures detailed errors.

webBuilder.UseSetting(WebHostDefaults.HostingStartupAssembliesKey, "assembly1;assembly2"); //to load external assemblies on top of the app assemblies.

webBuilder.UseSetting(WebHostDefaults.HostingStartupExcludeAssembliesKey, "assembly1;assembly2"); //to exclude loading of assemblies.

webBuilder.UseSetting("https\_port", "8080"); //define custom https port

webBuilder.UseStartup<Startup>(); // startup class

webBuilder.UseStartup("StartupAssemblyName"); // a startup assembly

webBuilder.UseUrls("http://\*:5000;http://localhost:5001;https://hostname:5002"); //listen on these urls for requests.

webBuilder.UseWebRoot("public"); //relative path to the content root directory.

* 1. Minimal API: Introduced in .net 6.0, these api provide new way to create a host, new routing and new project structures. If you are new to my docs, read this section after finishing the doc.
     1. WebApplication: Now apps can be created using a single file with only 4 lines. Just like node js.

In a new .cs file, or a file created with dotnet new web,

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();

That’s it.

Or

var app = WebApplication.Create(args);

app.MapGet("/", () => "Hello World!");

app.MapGet("/oops", () => "Oops! An error happened.");

app.Run();

//..Create creates a WebApplication with defaults.

//app.Run(); starts the server and blocks the thread.

* + - 1. We can define url in app.Run here,

app.Run("http://localhost:3000"); //runs the server on the given host //and port. This port must also be specified in Properties/launchSettings.json.

app.Urls.Add("http://localhost:4000");

//Multiple ports can be specified too.

app.Urls.Add("http://\*:3000"); //listen on all interfaces with port 3000.

app.Urls.Add("http://0.0.0.0:3000"); //same

app.Urls.Add("http://+:3000"); //same

* + - 1. The difference between app.Run(<url>) and app.Urls.Add is, app.Run can specify only a single url, while app.Urls.Add can specify multiple listener urls.
      2. ‘https’ can be used too, but first make the host os trust the dev certs. <https://docs.microsoft.com/en-us/aspnet/core/security/enforcing-ssl?view=aspnetcore-6.0&tabs=visual-studio#trust>

or

In appsetting.json,

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"AllowedHosts": "\*",

"Kestrel": {

"Certificates": {

"Default": {

"Path": "cert.pem",

"KeyPath": "key.pem"

}

}

}

}

We can specify custom certificate here.

or

In .cs file,

builder.Configuration["Kestrel:Certificates:Default:Path"] = "cert.pem";

builder.Configuration["Kestrel:Certificates:Default:KeyPath"] = "key.pem";

var app = builder.Build();

…

or

Use the Certificate API in the .cs file,

using System.Security.Cryptography.X509Certificates;

var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(options =>

{

options.ConfigureHttpsDefaults(httpsOptions =>

{

var certPath = Path.Combine(builder.Environment.ContentRootPath, "cert.pem");

var keyPath = Path.Combine(builder.Environment.ContentRootPath, "key.pem");

httpsOptions.ServerCertificate = X509Certificate2.CreateFromPemFile(certPath,

keyPath);

});

});

var app = builder.Build();

…

* + - 1. The other routing mechanisms like app.MapGet, app.MapPost etc. still work the same way as .net 5.0.
    1. Exception Handler work the same way,

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/oops");

}

* + 1. Environment vars:

var port = Environment.GetEnvironmentVariable("PORT") ?? "3000";

Simply using Environment object will work.

* + 1. Configuration: The <WebApplication obj> provides the IConfiguration object.

var message = app.Configuration["HelloKey"] ?? "Hello";

* + 1. Configuration Providers: The <WebApplicationBuilder obj> provides the ability to add Configuration Providers.

var builder = WebApplication.CreateBuilder(args);

builder.Configuration.AddIniFile("appsettings.ini");

var app = builder.Build();

* + 1. Logging: Same as in Config,

app.Logger.LogInformation("The app started");

* + - 1. HTTP Logging: Added in .net 6, this middleware is used to log HTTP requests and responses, it affects performance so it should be used only in dev env. By default logs are logged at Information level.

<IApplicationBuilder obj>.UseHttpLogging();

It can be configured as well,

<IServiceCollection obj>.AddHttpLogging(logging=>{

logging.LoggingFields = HttpLoggingFields.All;

logging.RequestHeaders.Add("My-Request-Header");

logging.ResponseHeaders.Add("My-Response-Header");

logging.MediaTypeOptions.AddText("application/javascript");

logging.RequestBodyLogLimit = 4096;

logging.ResponseBodyLogLimit = 4096;

});

* + - 1. W3C Logging: Same as HTTP Logging but outputs in W3C format, it has its own type of configuration. Add with ..UseW3Clogging(); and configure in services with ..AddW3Clogging(logging=>{…});
    1. Logging Providers: Same as in Config Providers,

var builder = WebApplication.CreateBuilder(args);

// Configure JSON logging to the console.

builder.Logging.AddJsonConsole();

var app = builder.Build();

* + 1. WebApplicationBuilder: We can define server configuration using CreateBuilder method on WebApplication.

var builder = WebApplication.CreateBuilder(new WebApplicationOptions

{

ApplicationName = typeof(Program).Assembly.FullName,

ContentRootPath = Directory.GetCurrentDirectory(),

EnvironmentName = Environments.Staging,

WebRootPath = "customwwwroot"

});

var app= builder.Build();

* + 1. DI: Services in the DI can be added by using <WebApplicationBuilder obj>.services.

var builder = WebApplication.CreateBuilder(args);

// Add the memory cache services.

builder.Services.AddMemoryCache();

// Add a custom scoped service.

builder.Services.AddScoped<ITodoRepository, TodoRepository>();

var app = builder.Build();

* + 1. Configuring HostBuilder: The same WebApplicationBuilder can configure both IHostBuilder and IWebHostBuilder.

var builder = WebApplication.CreateBuilder(args);

builder.Host.ConfigureHostOptions(o => o.ShutdownTimeout = TimeSpan.FromSeconds(30));

builder.WebHost.UseHttpSys();

builder.WebHost.UseWebRoot("webroot");

* + 1. Middleware:

var app = WebApplication.Create(args);

// Setup the file server to serve static files.

app.UseFileServer();

app.MapGet("/", () => "Hello World!");

app.Run();

* + 1. Routing:

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/", () => "This is a GET");

app.MapPost("/", () => "This is a POST");

app.MapPut("/", () => "This is a PUT");

app.MapDelete("/", () => "This is a DELETE");

app.MapMethods("/options-or-head", new[] { "OPTIONS", "HEAD" },

() => "This is an options or head request ");

app.Run();

* + - 1. app.Map<Verb> can take lambda function, static methods or other methods to invoke when request url matches.
      2. app.Map<Verb> can chain .WithName(<string>); to define internal route name which can be used by link generators. This name must be globally unique. WithName and WithTags are also picked up by OpenAPI to generate docs.

For example:

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/hello", () => "Hello named route")

.WithName("hi");

app.MapGet("/", (LinkGenerator linker) =>

$"The link to the hello route is {linker.GetPathByName("hi", values: null)}");

app.Run();

// ‘/’ endpoint will return The link to the hello route is /hello.

// linker.GetPathByName returns null if a route with the given name //isn’t found.

* + - 1. Route Parameters:

app.MapGet("/users/{userId}/books/{bookId}",

(int userId, int bookId) => $"The user id is {userId} and book id is {bookId}");

On /users/5/books/6 or any userId and bookId that can be converted to int, invokes this route and passes the values to the handler function. Returns an exception if binding fails.

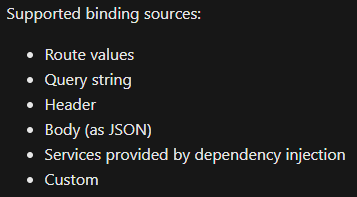
Route Constraints here work the same way as in .net 5.0.

* + - 1. Wildcard/Catch all routes:

app.MapGet("/posts/{\*rest}", (string rest) => $"Routing to {rest}");

Catches all routes like /posts/hello.

* + - 1. Model Binding: FormBinding is removed.



For example:

builder.Services.AddSingleton<Service>();

var app = builder.Build();

app.MapGet("/{id}", (int id, int page, Service service) => { });

Here, id is inferred from route value, page is inferred from query string and service is provided by DI. (This is how Action methods in .net 5.0 work too).

GET, HEAD, OPTIONS & DELETE don’t automatically bind from body, either use [FromBody] on params or read body from HttpRequest object.

* + - 1. Custom Model Binding: A Type can define either TryParse or BindAsync to allow model binding on it,

app.MapGet("/map", (Point point) => $"Point: {point.X}, {point.Y}");

app.Run();

//where Point is,

public class Point

{

public double X { get; set; }

public double Y { get; set; }

public static bool TryParse(string? value, IFormatProvider? provider,

out Point? point)

{

// Format is "(12.3,10.1)"

var trimmedValue = value?.TrimStart('(').TrimEnd(')');

var segments = trimmedValue?.Split(',',

StringSplitOptions.RemoveEmptyEntries | StringSplitOptions.TrimEntries);

if (segments?.Length == 2

&& double.TryParse(segments[0], out var x)

&& double.TryParse(segments[1], out var y))

{

point = new Point { X = x, Y = y };

return true;

}

point = null;

return false;

}

}

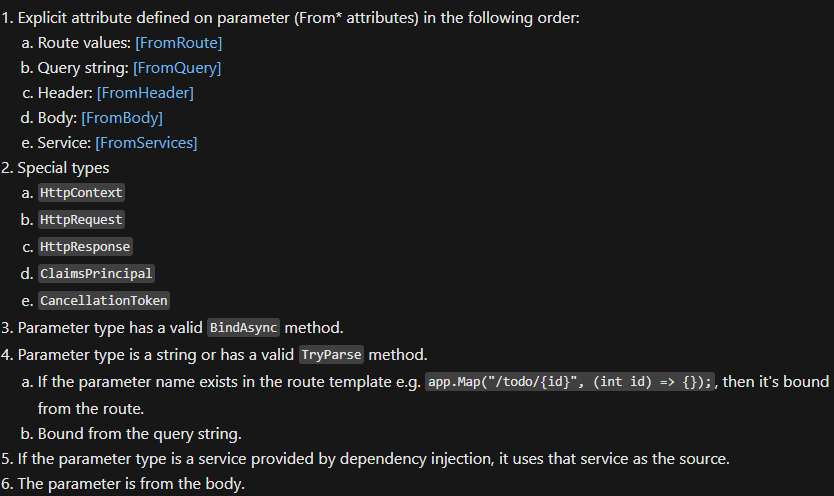
//TryParse must either have (string value, T out result) or (string value, //IFormatProvider provider, T out result) parameters.

// GET /map?Point=12.3,10.1 will invoke the Map method and the //model binding will succeed.

//Alternatively, we can use public static ValueTask<T?> BindAsync(HttpContext context, ParameterInfo parameter) or public static ValueTask<T?> BindAsync(HttpContext context);

* + - 1. Model Binding Precedence:

From the highest to the lowest priority:



* + 1. Return Types: Route Handlers can return either IResult, string or T. Or their Task or ValueTask versions. For T, the JSON serializer will serialize the return value and set content-type to ‘application/json’.

To use the built-in Result object,

app.MapGet("/api/todoitems/{id}", async (int id, TodoDb db) =>

await db.Todos.FindAsync(id)

is Todo todo

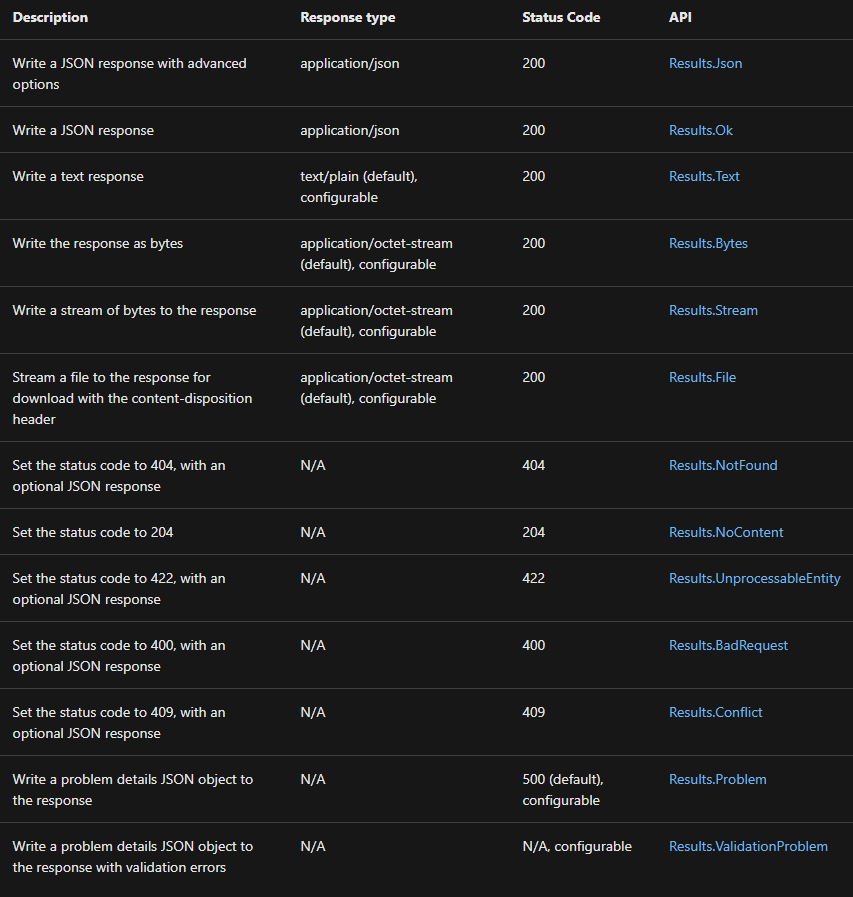
? Results.Ok(todo)

: Results.NotFound())

.Produces<Todo>(200)

.Produces(404);

Results.StatusCode or Results.Text or Results.Json or Results.Redirect or Results.File etc. are there as well.



* + 1. Custom Result: We can add the extension method to Microsoft.AspNetCore.Http.IResultExtensions namespace.

using System.Net.Mime;

using System.Text;

static class ResultsExtensions

{

public static IResult Html(this IResultExtensions resultExtensions, string html)

{

ArgumentNullException.ThrowIfNull(resultExtensions, nameof(resultExtensions));

return new HtmlResult(html);

}

}

class HtmlResult : IResult

{

private readonly string \_html;

public HtmlResult(string html)

{

\_html = html;

}

public Task ExecuteAsync(HttpContext httpContext)

{

httpContext.Response.ContentType = MediaTypeNames.Text.Html;

httpContext.Response.ContentLength = Encoding.UTF8.GetByteCount(\_html);

return httpContext.Response.WriteAsync(\_html);

}

}

* + 1. JSON Serializer Configuration: The body binding source uses System.Text.Json by default and this can’t be changed only configured.

builder.Services.Configure<JsonOptions>(options =>

{

options.SerializerOptions.IncludeFields = true;

});

* + 1. Authorization:

using Microsoft.AspNetCore.Authorization;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization(o => o.AddPolicy("AdminsOnly",

b => b.RequireClaim("admin", "true")));

var app = builder.Build();

app.UseAuthorization();

app.MapGet("/auth", [Authorize] () => "This endpoint requires authorization.");

//or app.MapGet("/auth", () => "This endpoint requires authorization.").RequireAuthorization();

app.Run();

We can define policy as well, [Authorize(“mypolicy”)] or the same for RequireAuthorization.

This also shows how attributes can be used on route handler methods.

* + 1. Cors:

using Microsoft.AspNetCore.Cors;

const string MyAllowSpecificOrigins = "\_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>

{

options.AddPolicy(name: MyAllowSpecificOrigins,

builder =>

{

builder.WithOrigins("http://example.com",

"http://www.contoso.com");

});

});

var app = builder.Build();

app.UseCors();

app.MapGet("/cors", [EnableCors(MyAllowSpecificOrigins)] () =>

"This endpoint allows cross origin requests!");

app.MapGet("/cors2", () => "This endpoint allows cross origin requests!")

.RequireCors(MyAllowSpecificOrigins);

* + 1. OpenAPI:

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new() { Title = builder.Environment.ApplicationName,

Version = "v1" });

});

var app = builder.Build();

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json",

$"{builder.Environment.ApplicationName} v1"));

}

app.MapGet("/swag", () => "Hello Swagger!");

app.MapGet("/skipme", () => "Skipping Swagger.")

.ExcludeFromDescription(); //to skip from swagger doc

app.Run();

* + 1. Use the HttpRequest object:

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/upload", async (HttpRequest req) =>

{

if (!req.HasFormContentType)

{

return Results.BadRequest();

}

var form = await req.ReadFormAsync();

var file = form.Files["file"];

if (file is null)

{

return Results.BadRequest();

}

var uploads = Path.Combine("uploads", file.FileName);

await using var fileStream = File.OpenWrite(uploads);

await using var uploadStream = file.OpenReadStream();

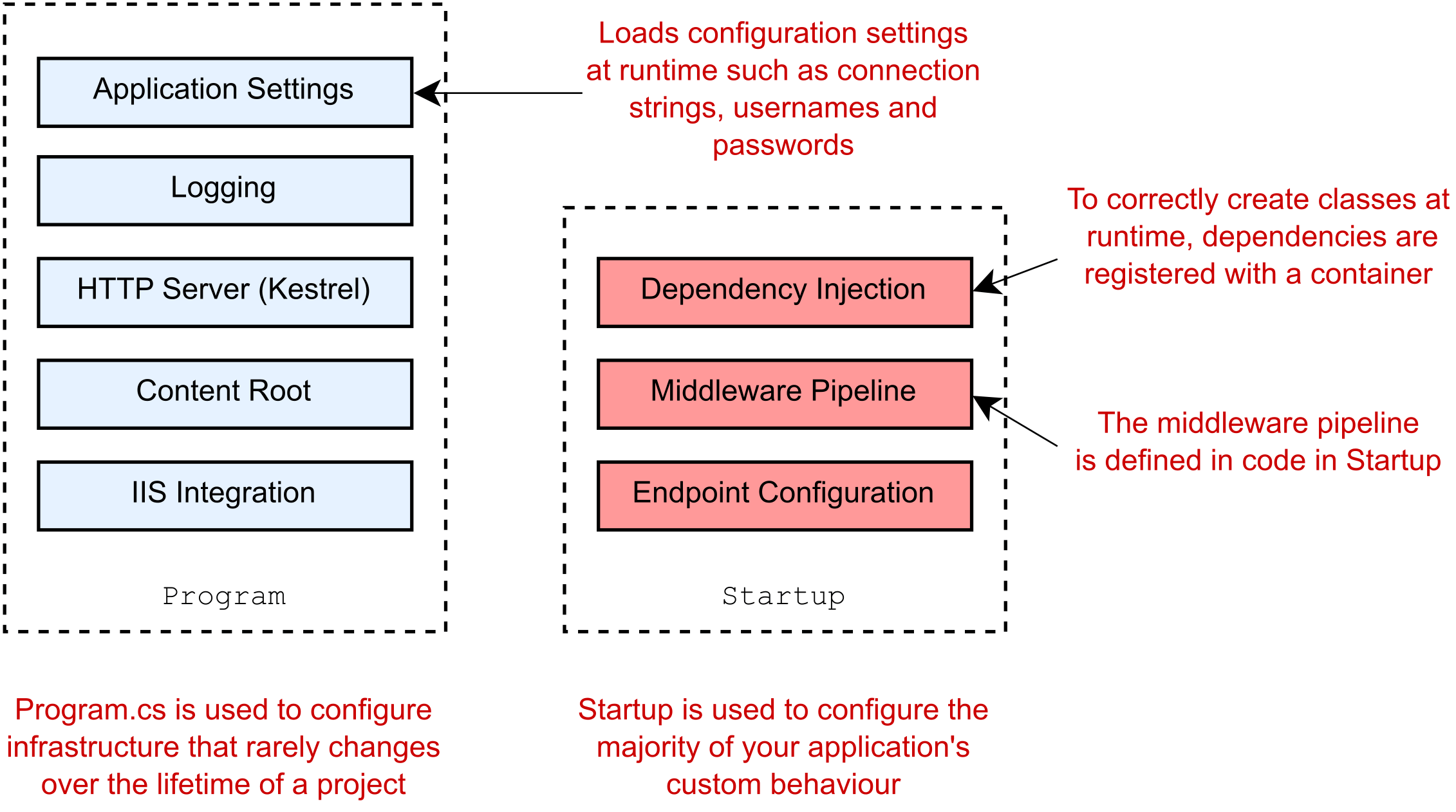
await uploadStream.CopyToAsync(fileStream);

return Results.NoContent();

})

.Accepts<IFormFile>("multipart/form-data");

app.Run();

3.6 General structure (Generic Host):

• Startup Class: This is the class where all the services are configured and middleware pipeline defined.

It should have 2 methods and a constructor,

<classname>(IConfiguration configuration){…}

public void **ConfigureServices**(IServiceCollection services){…}

public void **Configure**(IApplicationBuilder app){…}

* The constructor receives the configuration object from the Host, it is optional however. ASP.NET core uses constructor injection on startup class as well, 3 of the available interfaces are, IWebHostEnvironment, IHostEnvironment and IConfiguration.
* The ConfigureServices method is called before the Configure method and the startup class itself is called after the Host is done configuring things.
* In ConfigureServices, the order matters. Services added later can work with services added before but not vice-versa. This is the same behavior as in Configure method.

• Multiple Startup: The host can specify multiple Startup classes with its UseStartup method, if a classname ends with the environment name then that class is used instead of the other startup classes.

• Dependency Injection: .Net core has a DI framework that makes configured services available throughout the app.

The services are added to the DI container first, one way to add them is through the **ConfigureServices** method, <IServiceCollection Object>.Add<X> is the syntax where X is the name of the service, many services are pre-defined and only need initialization like AddDbContext.

* Then these services are available throughout the app, for example, if any controller needs a service, it can ask for it in it’s constructor and if the service is initialized and exists in the DI container then it is passed automatically using Constructor Injection.
* The **Configure** method itself can use DI container, any service in the DI container can be requested by the Configure method, even our own dependencies, given they are added to the container.
* We can add our own dependency and then pass it’s implementation through the DI container.

To create our own **dependency**:

First create an interface

Public Interface myInterface {…}

Then implement it

Public class someClass : myInterface{…}

Then simply use any of the ConfigureServices(…) method,

{

Services.AddScoped<myInterface,someClass>();

}

And then simply use myInterface in any controller and ASP.NET core will inject the implementation through constructor injection.

* The AddScoped method adds a scoped dependency, meaning the implementation class will initialize at every HTTP request.
* The dependency class can use the DI container as well, i.e., it can have it’s own dependencies and they will be given through constructor injection as well.
* To add a related group of services to the DI container we can use extension method on the IServiceCollection.

First create an extension method,

Public static class myExtension

{

Public static IServiceCollection AddConfig(

This IServiceCollection services, <some interfaces >,

)

{

Services.Add<X>;

}

}

Then use it like in a ConfigureServices like so,

Services.AddConfig(<the implementations for requested interfaces>).AddMyDependencyGroup();

AddConfig is just a method name and for convention all extension methods must use Add suffix. Furthermore, they must be in Microsoft.Extensions.DependencyInjection namespace.

The extension method doesn’t get access to DI container.

* To add custom dependencies to the DI container we can use any of the 3 methods of IServiceCollection.

AddScoped: Scoped objects are same for every requesting class but differ across requests.

AddTransient: Transient objects are different for any requesting class and across requests.

AddSingleton: Singleton objects are the same throughout the app’s lifetime.

Regardless of the Add method used, the disposing of objects is handled by framework automatically and they mustn’t be disposed manually.

However, if a service is injected like so,

Services.AddSingleton(new myService());

Then we must handle disposing it manually.

* We can use the DI container even in main,

First, create host.

Var host= Host….Build();

Then use,

Using(var serviceScope=host.Services.CreateScope())

{

Var services=serviceScope.serviceProvider;

…do Stuff with services. For example, seeding the database;

Using(var somedbContext= new XDbContext(services.GetRequiredService<DbContextOptions<XDbContext>>())){

New Someseederclass(somedbContext);

}

}

The GetService method should be avoided as much as possible.

* We can request services from the DI container using HttpContext.RequestServices. Using constructor injection is preferred but this is an alternative.
* There are more than 250 services registered in the DI by the framework automatically.

• Middleware: The request handling pipeline is composed of a series of middleware components, these components perform task on an HttpContext Object which holds the request and then either pass it to the next component or terminate the request.

One way to define the middleware is through the **Configure** method. <IApplicationBuilder Object>.Use<X> where X is the component to be invoked is the synstax. Pre-defined components include UseRouting(), UseStaticFiles() etc.

Example:

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseExceptionHandler("/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapRazorPages();

});

}

* Order of invocation matters here.
* Each middleware component in the request performs some task then can either invoke the next component or terminate the middleware pipeline, this short-circuits the pipeline and the component that does so is called terminal middleware.
* Middleware components are also called Request Delegates as they handle requests, each request delegate can perform work before or after another and they all use either Run, Map or Use extension method.
* The Request-Response flow. It is preferred to use exception-handling request delegates before any other request delegate.  
  
* We can define a middleware component either in-line or through classes (explained how in IStartupFilter).
* To define an in-line middleware component,

**Configure(**IApplicationBuilder app**){**

…

App.Use(async(context, next)=>{

..do something with the HttpContext object which has the request data.

//optionally,

Await next.Invoke();

}

);

**}**

If we don’t call next.invoke then this will be a terminal middleware component,i.e., the HttpContext object will be returned.

* The HttpContext object has a Response method, this is where we modify the Response that will be ultimately sent back. Each middleware component can read the Request from the HttpContext object and can write to the Response object. However, it is an exception if a middleware component writes to a Response object after it has been sent back, i.e., when the middleware

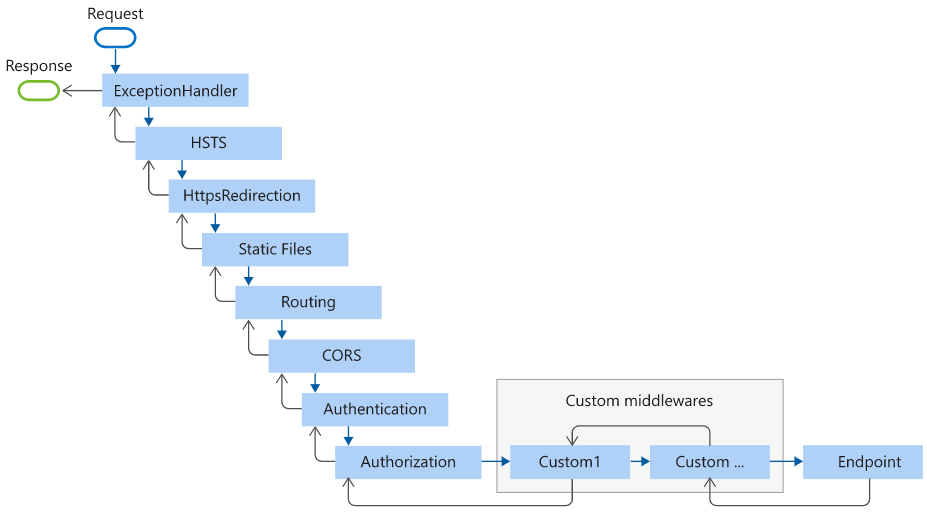
component before it short-circuited.

* If we are going to short-circuit, we can use Run method instead, it runs a component and is always terminal.

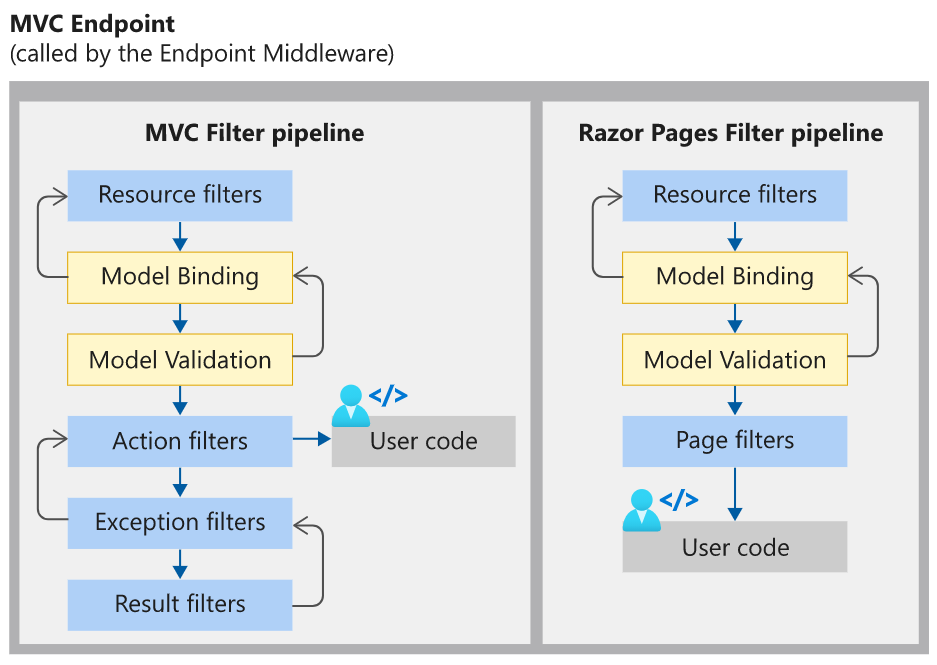
Syntax:

<IApplicationBuilder object>.Run(async (context)=>{…});

* Typical request processing pipeline for MVC apps and Razor pages.



At the endpoint middleware, the filter pipeline begins. This is different between MVC app and Razor Pages.



* Typical middleware pipeline, in code:
* public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
* {
* if (env.IsDevelopment()) //check if app is running in dev env
* {
* app.UseDeveloperExceptionPage(); // runtime app errors
* //are shown through dev exception page.
* app.UseDatabaseErrorPage(); // same but for db errors.
* }
* else
* {
* app.UseExceptionHandler("/Error"); // catches exception
* app.UseHsts(); //Adds Strict-Transport-Security header
* }
* app.UseHttpsRedirection(); //redirects http requests to https
* app.UseStaticFiles(); //returns static files and short-circuits.
* // app.UseCookiePolicy(); //conforms the app to GDPR regulations.
* app.UseRouting(); //route requests
* // app.UseRequestLocalization();
* // app.UseCors();
* app.UseAuthentication(); //self explainatory
* app.UseAuthorization(); //self explainatory
* // app.UseSession(); // establishes and maintains session state.
* // app.UseResponseCompression();
* // app.UseResponseCaching();
* app.UseEndpoints(endpoints => //defines routing
* {
* endpoints.MapRazorPages();
* endpoints.MapControllerRoute(
* name: "default",
* pattern: "{controller=Home}/{action=Index}/{id?}");
* });

}

* The Map method on IApplicationBuilder can be used to map url requests to different components if the request url matches. This is called Branching.

For example,

Configure(…){

App.Map(‘/map1’,myMethod);

}

Where myMethod is,

Public void myMethod(IApplicationBuilder app)

{ …do Something with app.

}

If the request url ends with /map1 then this method gets invoked and matched path segments are removed from HttpRequest.Path and appended to HttpRequest.PathBase.

* We can nest Map methods and also use multi segments as well. Like ‘/map/something/’.

* There’s also MapWhen, syntax similar to Run but return type must be a bool. The predicate given to it can use the HttpContext to Map based on more than just the url.
* Map and MapWhen do not rejoin the branches, if a branch occurs then the branch must handle the entire pipeline from the point it got branched off of the main middleware pipeline.
* UseWhen method is just like MapWhen but the branch rejoins the main branch after it ends and when it doesn’t short-circuit.
* There are a lot of built-in middleware we can readily use (as presented in the diagram above).
* We can do Route-To-Code for basic JSON APIs directly in .net core. Basically we can skip all the controllers, action methods, model binding etc. and let a simple method handle a request.

To do so,

<IApplicationBuilder obj>.UseEndpoints(endpoints=>{

endpoints.MapGet(‘/hello/{name:alpha}’, async context=> {

var repository = context.RequestServices.GetService<UserRepository>(); // to use DI

var name= context.Request.RouteValues[‘name’];

if(context.Request.HasJsonContentType())

{

var weather= await context.Request.ReadFromJsonAsync<Weather>();

context.Response.StatusCode=(int)HttpStatusCode.Accepted;

context.Response.WriteAsJsonAsync(name);

}

else{

context.Response.StatusCode=400;

}

})

}).RequireAuthorization();

The method could have been MapPost for post requests.

The RequireAuthorization does what it’s named, UseAuthorization doesn’t work on endpoints like these. We can use AllowAnonymous instead as well.

We can configure the JsonSerializer options with

<IServiceCollection obj>.Configure<JsonOptions>(o=>{

o.SerializerOptions.WriteIndented=true;

});

* Custom Map or Route based APIs must be put before managed endpoints like MapRazorPages.
* To create a middleware (without startup filter),

Conventional Method:

..MyMiddlewareClass

{

…get the RequestDelegate obj from DI

public async Task InvokeAsync(HttpContext obj, …some obj from DI)

{

… short circuit or await requestDelegateObj(obj);

}

}

IMiddleware method:

…create the middleware class normally but extend IMiddleware

Then to add them, creat extension method on IApplicationBuilder,

public static class MiddlewareExtensions

{

public static IApplicationBuilder UseConventionalMiddleware(

this IApplicationBuilder builder)

{

return builder.UseMiddleware<MyMiddlewareClass>();

}

public static IApplicationBuilder UseIMiddleware(

this IApplicationBuilder builder)

{

return builder.UseMiddleware<MyIMiddlewareClass>();

}

}

The IMiddleware middleware must be added in **ConfigureSevices,**

<IServiceCollection obj>.AddTransient<MyIMiddlewareClass>();

and lastly,

<IApplicationBuilder obj>.UseConventionalMiddleware();

<IApplicationBuilder obj>.UseIMiddleware();

* There’s one more method of creating middlewares, using IMiddlewareFactory.

• The host can be used to directly configure the DI container and the Middleware pipeline, by simply calling

Main(string[] args) {

Host.CreateDefaultBuilder(args).ConfigureAppConfiguration((hostingContext, config)=>{…}).ConfigureWebHostDefaults(webBuilder=>{

webBuilder.ConfigureServices(services=>{…}).Configure(app=>{…});

}).Build().Run();

}

Here ConfigureAppConfiguration can be used to modify app configuration.

• IStartupFilter: This filter is used to add defaults to the beginning of the middleware pipeline by ASP.NET core, we as well can use it to add middleware components, and we can define if it will be invoked before or after all the components are invoked.

To use it, first we create a **middleware.**

Public class myMiddleWare{

Private readonly RequestDelegate \_next;

Public myMiddleWare(RequestDelegate next)=> (\_next)=(next);

Public async Task Invoke(HttpContext httpContext){

//…do Stuff with httpContext which has the request data.

Return \_next(httpContext);

}

}

Then create StartupFilter,

Public class myStartupFilter: IStartupFilter{

Public Action<IApplicationBuilder> Configure(Action<IApplicationBuilder> next){

Return builder => {

builder.UseMiddleware<myMiddleWare>();

next(builder);

}

}

}

To use this startupfilter and middleware, <IServiceCollection Object>.AddTransient<IStartupFilter,myStartupFiler>();

Now this middleware component can be put in the startup class but that beats the point of it’s existence, to use it we define it in the host itself.

Host.CreateDefaultBuilder(args).ConfigureAppConfiguration((\_,\_\_)=>{…}).ConfigureServices(services=>{ services.AddTransient….}).ConfigureWebHostDefaults((\_)=>{…}).Build().Run();

This ensures the startup filter is invoked before the custom middleware pipeline, to make it so that startup filter is invoked after the custom middleware pipeline we simply need to put the ConfigureServices((\_)=>{…}) after ConfigureWebHostDefaults. That is to say, order matters.

• Configuration: Configuration in this framework is given by configuration providers that read data as key-value pairs from variety of sources such as, json setting files, env vars, cli args, custom providers, dir files and in-memory .net objects. The CreateDefaultBuilder of IHostBuilder provides default configuration of the app, in the given order.

Creates IConfiguration object

Reads appsettings.json using JSON config provider.

Reads appsettings.<Environment>.json

Reads App secrets when app runs in dev env.

Reads env vars using env var config prov.

Reads cli args using cli config prov.

Order matters as the last value assigned to a key is only used.

* We can check the config provs at runtime by using IConfiguration object’s Providers.ToList() method.
* How to read loaded Configuration,

Given this appsettings.json,

{

"Position": {

"Title": "Editor",

"Name": "Joe Smith"

},

"MyKey": "My appsettings.json Value",

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Information"

}

},

"AllowedHosts": "\*"

}

We use <IConfiguration object> to read values from it,

<IConfiguration object>[“MyKey”]

<IConfiguration object>[“Position:Title”]

<IConfiguration object>[“Logging:LogLevel:Default”]

And so on. Alternatively, we can use GetValue method.

* There’s also GetValue, GetSection, GetChildren and Exists methods on IConfiguration. Section gets a section, children gets subsections inside a section and exists check if any of these method have an actual value.
* The JSON prov reads changes to appsettings.json and restarts the app if it detects changes because of the default reloadOnChange key being true.
* It is recommended to use Options pattern. It is basically just creating a class for the keys of json files. Since the key is known we can manually create them as const attributes. It is elaborated in its own section.
* Only env vars with their key(name) that match the prefix are read. By default only DOTNET\_ and ASPNETCORE\_ are prefixes.
* We can define a custom prefix for env vars. < IHostBuilder obj>.CreateDefaultBuilder(args).ConfigureAppConfiguration((hostContext, config)=>{

Config.AddEnvironmentVariables(prefix: “Some prefix”);

})…

We can use the same method to add/override env vars for the app.

* Appsettings and its equivalent in env vars.
* {
* "SmtpServer": "smtp.example.com",
* "Logging": [
* {
* "Name": "ToEmail",
* "Level": "Critical",
* "Args": {
* "FromAddress": "MySystem@example.com",
* "ToAddress": "SRE@example.com"
* }
* },
* {
* "Name": "ToConsole",
* "Level": "Information"
* }
* ]

}

setx SmtpServer=smtp.example.com

setx Logging\_\_0\_\_Name=ToEmail

setx Logging\_\_0\_\_Level=Critical

setx Logging\_\_0\_\_Args\_\_FromAddress=MySystem@example.com

setx Logging\_\_0\_\_Args\_\_ToAddress=SRE@example.com

setx Logging\_\_1\_\_Name=ToConsole

setx Logging\_\_1\_\_Level=Information

* Env vars set in launchSettings.json override system env vars.

Like,

“applicationUrl”:”some urls”

Sets the ASPNETCORE\_URLS env var and overrides those of system.

* To display env vars being used,

Var host = …Build();

Var config= host.Services.GetRequiredService<IConfiguration>();

And read values from config.AsEnumerable();

* We can use cli args to set env vars using either =, /,-- or – and any mix.

Dotnet run MyKey=”something” Position:Title=CMD Position:XYZ=zxc

Dotnet run /MyKey ”something” /Position:Title=CMD /Position:XYZ=zxc

Dotnet run --MyKey=”something” --Position:Title=CMD --Position:XYZ=zxc

Dotnet run -MyKey ”something” --Position:Title=CMD --Position:XYZ=zxc

Single ‘-‘ keys require usage of Switch Mapping.

* SwitchMapping: Switch Mappings allow key name to be replaced. The key names that are replaced are from the cli args. These are required if the key in cli args begin with single ‘-‘.

To do it,

…ConfigureAppConfiguration((…)=>{

Var switchMappings=new Dictionary<string,string>(){

{‘-someKey’,’someAnotherKey’},

{‘-someKey2’,’someAnotherKey2’},

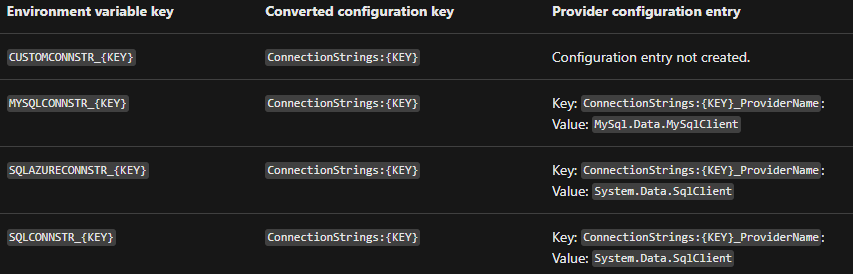
};

Config.AddCommandLine(args, switchMappings);

})…

Any cli args that have the matching key have their key replaced with one from switchMappings.

* Keys are always case sensitive and values are always strings. Values are non null.
* Connection Strings are automatically converted if they are found in the given format in env vars.



* There are various config providers, some of them are, json config prov, ini config prov and xml config prov.

To use ini config prov,

…ConfigureAppConfiguration((…)=>{

Config.Sources.Clear(); //clears the configs.

Config.AddIniFile(‘<ini file name from root dir>.ini’,optional: <bool>, reloadOnChange: <bool>);

Config.AddEnvironmentVariables();

If(args!=null)

{

Config.AddCommandLine(args);

}

})…

Here, first config sources are cleared, then ini files are loaded, then env vars are loaded and then cli args are loaded, in the order they are called. They can override previous config provs values if the key match.

* Config.AddJsonFile(…) syntax similar to ini loads custom json files.
* Config.AddXmlFile(…)
* Config.AddKeyPerFile(directoryPath: <Path object for path to a folder>,optional:<bool>); adds all files to the config, the filenames are keys and their content are values.
* Config.AddInMemoryCollection(<dict object>); adds dict to config.
* Sample appsettings.json,

{

"Kestrel": {

"Endpoints": {

"Https": {

"Url": "https://localhost:9999"

}

}

},

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Information"

}

},

"AllowedHosts": "\*"

}

The kestrel key overrides cli args and app given values but not env vars.

* We can load configuration from database using EFCore.

To do so, we will have to create our own **Configuration Provider**

First create dbContext class with the model class for our data.

Then,

// using Microsoft.EntityFrameworkCore;

// using Microsoft.Extensions.Configuration;

Public class myConfSource: IConfigurationSource

{

Private readonly Action<DbContextOptionsBuilder> \_options;

Public myConfSource(Action<DbContextOptionsBuilder> options) =>(\_options)=(options);

public IConfigurationProvider Build(IConfigurationBuilder builder)

{

Return new MyConfigurationProvider(\_options);

}

}

public class MyConfigurationProvider : ConfigurationProvider

{

Private readonly Action<DbContextOptionsBuilder> \_options;

public MyConfigurationProvider(Action<DbContextOptionsBuilder> options)=> (\_options)=(options);

public override void Load()

{

Var builder = DbContextOptionsBuilder<MyDbContextClass>();

\_options(builder);

using(var dbContext= new MyDbContextClass(builder.Options))

{

dbContext.Database.EnsureCreated();

Data= !dbContext.Values.Any() ? CreateAndSaveDefaultValues(dbContext) : dbContext.Values.ToDictionary(c=> c.<something> , c=> c.<Something else>);

}

}

private static IDictionary<string,string> CreateAndSaveDefaultValues(MyDbContextClass obj){

//Create value, store in db and return.

}

}

Now create an ExtensionMethod on IConfigurationBuilder,

// using Microsoft.EntityFrameworkCore;

// using Microsoft.Extensions.Configuration;

public static class EntityFrameworkExtensions

{

public static IConfigurationBuilder AddEFConfiguration(

this IConfigurationBuilder builder,

Action<DbContextOptionsBuilder> optionsAction)

{

return builder.Add(new myConfSource(optionsAction));

}

}

Now to use it,

..ConfigureAppConfiguration((…)=>{

config.AddEFConfiguration(options=> options.UseInMemoryDatabase(“SomeDb”));

})

The config here gets passed a database to use, we can use any db.

• Options pattern: This pattern is used to convert configuration data into strongly-typed models and to validate the values.

To start with it,

Assume this is the content of appSettings.json,

{"Position": {

"Title": "Editor",

"Name": "Joe Smith"

}

}

Fort this json we create this options class,

public class PositionOptions

{

public const string Position = "Position";

public string Title { get; set; }

public string Name { get; set; }

}

Fields are not bound so Position variable doesn’t get binded.

To use it,

var positionOptions = new PositionOptions();

Configuration.GetSection(PositionOptions.Position).Bind(positionOptions);

* To bind a class to a json file,

Var obj= new PositionOptions ();

<IConfiguration object>.GetSection(obj.Position).Bind(obj);

Or var obj2=<IConfiguration object>.GetSection(PositionOptions.Position).Get< PositionOptions >();

This reads a part of the appsettings.json file and binds its value to obj.

and we can use obj/obj2.Title or Name anywhere and the corresponding value from the configuration will be retrieved.

* We can use DI container instead of binding,

<IServiceCollection object>.Configure<MyOptionsClass>(<IConfiguration obj>.GetSection(<some section name>));

And now to access the class from DI container, simply use,

Public class XYZ: Controller

{

Private readonly MyOptionsClass \_obj;

XYZ(IOptions<MyOptionsClass> options)=>(\_obj)=(options);

}

Here, changes to json config files do not cause restart of the app, to get that functionality use IOptionsSnapshot instead.

* Options has 3 interfaces,

IOptions<TOptions>: Is a singleton, Does not read config after app has started, and does not support named options. Can be injected into any service.

IOptionsSnapshot<TOptions>: Does read config at each http request, is a Scoped service so cannot be injected into singleton services and supports named options.

IOptionsMonitor<TOptions>: Is used to retrieve options and manage options notifications for TOptions instances. Is a singleton and supports everything.

Snapshot and Monitor have a difference that Snapshot reads updated values at ever httprequest/ when the object is created while Monitor reads values whenever object is requested. Monitor is better for singleton and Snapshot is better for transient and scoped dependencies.

* Named Options:

For this json,

{

"TopItem": {

"Month": {

"Name": "Green Widget",

"Model": "GW46"

},

"Year": {

"Name": "Orange Gadget",

"Model": "OG35"

}

}

}

We use a single class,

public class TopItemSettings

{

public const string Month = "Month";

public const string Year = "Year";

public string Name { get; set; }

public string Model { get; set; }

}

<IServiceCollection object>.Configure<TopItemSettings>(TopItemSettings.Month,

Configuration.GetSection("TopItem:Month"));

<IServiceCollection object>.Configure<TopItemSettings>(TopItemSettings.Year,

Configuration.GetSection("TopItem:Year"));

Here the sections are given by name.

* We can configure options as well.

<IServiceCollection object>.AddOptions<MyOptions>("optionalName")

.Configure<Service1, Service2, Service3, Service4, Service5>(

(o, s, s2, s3, s4, s5) =>

o.Property = DoSomethingWith(s, s2, s3, s4, s5));

MyOptions is the options class that is added to the DI container while we can use upto 5 services from the DI container to configure it.

* Options validation:

For the given json file,

{

"MyConfig": {

"Key1": "My Key One",

"Key2": 10,

"Key3": 32

}

}

public class MyConfigOptions

{

public const string MyConfig = "MyConfig";

[RegularExpression(@"^[a-zA-Z''-'\s]{1,40}$")]

public string Key1 { get; set; }

[Range(0, 1000,

ErrorMessage = "Value for {0} must be between {1} and {2}.")]

public int Key2 { get; set; }

public int Key3 { get; set; }

}

We use DataAnnotations to validate keys being bound to the model.

//using Microsoft.Extensions.Options.DataAnnotations

<IServiceCollection object>.AddOptions<MyConfigOptions>()

.Bind(Configuration.GetSection(MyConfigOptions.MyConfig))

.ValidateDataAnnotations();

to actually validate the model.

It will not throw until a service actually calls <IOptions object>.Value.

The exception thrown is of OptionsValidationException type.

We can add .Validate method to the ValidateDataAnnotations() method, it takes a delegate.

..Validate(config=> {

…some check on config (of type MyConfigOptions) values.

return bool;

}, “<Failure message>”);

* Even more complex validation can be done by extending IValidateOptions<TOptions class>.

public class myValidationClass : IValidateOptions<MyConfigOptions>

{

private MyConfigOptions \_config;

public myValidationClass(IConfiguration config)

{

\_config=config.GetSection(MyConfigOptions.SomeSection).Get<MyConfigOptions>();

}

public ValidateOptionsResult Validate(string name, MyConfigOptions options)

{

//check somethings of \_config and options and return //ValidateOptionsResult.Fail(“<failure message>”)/Success;

}

}

To add this validation,

<IServiceCollection object>.Configure… bind normally.

<IServiceCollection object>.TryAddEnumerable(ServiceDescriptor.Singleton<IValidateOptions<MyConfigOptions>, myValidationClass>());

* Options post-configuration: We can add post-configuration to configure the options object after all configurations have occurred.

To do so,

<IServiceCollection object>.PostConfigure<MyOptionsClass>(<optional named options>, myOptions=>{ myOptions.SomeOption=’some new value’; });

<optional named options> can be used to post configure only some named option.

PostConfigureAll to configure all configuration instances.

* IOptions can be used by **Configure** method as it is inserted into the DI container in ConfigureServices which is before it.

• Environments: The ‘environment’ value can be anything but these 3 values are the main ones and are supported by framework methods.

‘Development’, ‘Staging’ and ‘Production’.

The environment value can be set by using env vars and launchsettings.json,

DOTNET\_ENVIRONMENT and ASPNETCORE\_ENVIRONMENT are the env var keys for it and ASPNETCORE\_ENVIRONMENT overrides the former if both are found.

launcSettings.json sets the ASPNETCORE\_ENVIRONMENT to development if it is used, otherwise if environment is not defined through env vars or launchsettings then Production is used.

* launchSettings overrides ASPNETCORE\_ENVIRONMENT’s value.
* To check env in **Configure** method,

<IWebHostEnvironment object>.IsDevelopment()/IsProduction()/IsStaging/IsEnvironment(“some environment name”) returns bool.

* environment value isn’t case sensitive in windows and mac but is in linux..
* Sample launchSettings.json,
* {
* "iisSettings": {
* "windowsAuthentication": false,
* "anonymousAuthentication": true,
* "iisExpress": {
* "applicationUrl": "http://localhost:64645",
* "sslPort": 44366
* }
* },
* "profiles": {
* "IIS Express": {
* "commandName": "IISExpress",
* "launchBrowser": true,
* "environmentVariables": {
* "ASPNETCORE\_ENVIRONMENT": "Development"
* }
* },
* "EnvironmentsSample": {
* "commandName": "Project",
* "launchBrowser": true,
* "applicationUrl": "https://localhost:5001;http://localhost:5000",
* "environmentVariables": {
* "ASPNETCORE\_ENVIRONMENT": "Development"
* }
* }
* }
* }

This file gives 2 profiles to be used by the app. We can select one using VS’s green run button’s dropdown or by using –launch-profile “<profile name>” with dotnet run in cli.

* VSCode uses launch.json instead.
* We can have environment bases startup apps,

Firstly define Startup classes with classname having their environment names as suffix. Like StartupProduction. And a default Startup class as well.

Then in IHostBuilder,

var assemblyName= typeof(Startup).GetTypeInfo().Assembly.Fullname;

…

webBuilder.UseStartup(assemblyName);

…

* Similarly, we can have env based **Configure** and **ConfigureServices** methods in startup classes, Configure{Env name}Services and Configure{Env name} is their syntax. Depending on the env the appropriate Config is opened.

• Logging: ASP.Net core has built-in logging API and supports 3rd party ones as well. Available providers are Console, Debug, Windows Event Tracing, Windows Event Log, TraceSource, Azure App Service and Azure Application Insights.

To use a basic logger, first create ILogger object in controller then use ILogger<[ControllerName]> in constructor parameter to get it from DI and then <logger object>.Log<Information/Warning/etc.> (LoggingEvent.<some event type>, “<message>”,<optional params>); to log events.

This ILogger object is the interface for all logger implementations so even if the logging provider API changes it will not affect controllers as long as it implements ILogger.

* Multiple logging providers can be used at the same time.
* The CreateDefaultBuilder in IHostBuilder adds the following log providers,

Console, Debug, EventSource and EventLog.

* We can add/remove logging providers using the IHostBuilder itself.

To do so,

Host.CreateDefaultBuilder(args).ConfigureLogging(logging=>{

logging.ClearProviders(); // clears all default logging providers

logging.AddConsole(); //adds console logging provider.

})…

* Logger Configuration: It is set by appsettings.{Environment}.json’s ‘Logging’ section.

{

“Logging”:

{

“LogLevel”:

{

“Default”:”Information”, //default log level for namespaces.

“Microsoft”:”Warning”,

“Microsoft.Hosting.Lifetime”:”Information”

}

}

}

LogLevel dictates that any namespace with the given category will automatically log at the given level, given their classes/methods don’t give their own log levels.

Microsoft means Microsoft.something… namespaces, and same for ..lifetime.

* ..AddEventLog(eventLogSettings => { eventLogSettings.SourceName/LogName/MachineName = ‘something’ }); adds event logger. The settings are by default ‘Application’/’.NET Runtime’/’ automatically retrieved Machine name’ for the respective fields.
* Available log levels,



* Sample appsettings,json for multiple logging providers,

{

"Logging": {

"LogLevel": { // No provider, LogLevel applies to all the enabled providers.

"Default": "Error",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Warning"

},

"Debug": { // Debug provider.

"LogLevel": {

"Default": "Information" // Overrides preceding LogLevel:Default setting.

}

},

"Console": {

"IncludeScopes": true,

"LogLevel": {

"Microsoft.AspNetCore.Mvc.Razor.Internal": "Warning",

"Microsoft.AspNetCore.Mvc.Razor.Razor": "Debug",

"Microsoft.AspNetCore.Mvc.Razor": "Error",

"Default": "Information"

}

},

"EventSource": {

"LogLevel": {

"Microsoft": "Information"

}

},

"EventLog": {

"LogLevel": {

"Microsoft": "Information"

}

},

"AzureAppServicesFile": {

"IncludeScopes": true,

"LogLevel": {

"Default": "Warning"

}

},

"AzureAppServicesBlob": {

"IncludeScopes": true,

"LogLevel": {

"Microsoft": "Information"

}

},

"ApplicationInsights": {

"LogLevel": {

"Default": "Information"

}

}

}

}

* settings in Logging.{provider}.LogLevel override settings of Logging.LogLevel.
* For convention we include fully qualified type name(namespace.class) in the log message.

We can explicitly set the category as well,

Instead of using ILogger<T> in the constructor, request ILoggerFactory then,

…

\_logger= <ILoggerFactory object>.CreateLogger(“MyCategory”);

…

This category is the one we can define in appsettings.json.

* If the default log level is not set then Information is the Loglevel for all logs.
* We can change the default log level, i.e., when it is not set in appsettings or through env vars by,

…ConfigureLogging((…)=>{

logging.SetMinimumLevel(LogLevel.SomeLevel);

})…

* LogFilter: A log filter filters logs, it is a function that returns a bool and only prints log if that is true.

To set one,

…ConfigureLogging((…)=>{

logging.AddFilter((provider,category,logLevel) {

// do something and return a bool.

});

})…

* Log Scopes: A scope can group a set of logs (though we can just do it without it). Only Console and AzureAppServices{File/Blob} support it.

To use it,

using (<ILogger object>.BeginScope(“message”))

{

//log messages normally.

}

* Default logging providers don’t write logs to files.
* PerfView utility helps collecting and viewing logs.
* Azure App Service can write logs to text files to azure app’s file system and/or to blob store in Azure Storage.

To do so,

Host.CreateDefaultBuilder(args)

.ConfigureLogging(logging => logging.AddAzureWebAppDiagnostics())

.ConfigureServices(serviceCollection => serviceCollection

.Configure<AzureFileLoggerOptions>(options =>

{

options.FileName = "azure-diagnostics-";

options.FileSizeLimit = 50 \* 1024;

options.RetainedFileCountLimit = 5;

})

.Configure<AzureBlobLoggerOptions>(options =>

{

options.BlobName = "log.txt";

}))

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.UseStartup<Startup>();

});

* Logging can also be used by console apps (non host),

To do so,

* using var loggerFactory = LoggerFactory.Create(builder =>
* {
* builder
* .AddFilter("Microsoft", LogLevel.Warning)
* .AddFilter("System", LogLevel.Warning)
* .AddFilter("LoggingConsoleApp.Program", LogLevel.Debug)
  + - .AddConsole()
    - .AddEventLog();
* });
* ILogger logger = loggerFactory.CreateLogger<Program>();
* logger.LogInformation("Example log message");

where ‘Program’ is classname.

* There’s another way of adding logger to a service (other than constructor injection), by using DI container.

<IServiceCollection object>.AddSingleton<IMyService>((container)=>{

var logger=container.GetRequiredService<ILogger<MyService>>();

return new MyService() {Logger= logger};

});

* To create logs in main,
* var logger = host.Services.GetRequiredService<ILogger<Program>>();
* logger.LogInformation("Host created.");
* We can inject ‘ILogger<Startup> logger’ into **Configure** method but not into **ConfigureServices** or Startup’s constructor, because logger isn’t initialized until **ConfigureServices** finished setting up DI container.
* Don’t use async operations with logs.

• Error Handling: We can handle errors and exceptions in ASP.NET core using, developer exception page, custom error page, static status code page and startup exception handling.

The developer exception page (<IApplicationBuilder object>.UseDeveloperExtensionPage()) displays verbose error information in case of an exception but should only be used in dev env.

* To configure a custom exception handler page, use,

<IApplicationBuilder object>.UseExceptionHandler(‘<route>’);

On exception, .net redirects to the given route page with the same method as the HTTP request came in, so the controller for the route must be able to accept all the methods or define contents based on HTTP methods.

* IExceptionHandlerPathFeature is an object in HttpContext that contains error information,

To get it,

HttpContext.Features.Get<IExceptionHandlerPathFeature>();

Returs nullable object.

* Exception handler lambda: Alternative to routing to an exception page we can use a lambda,

…app.UseExceptionHandler(errorApp=>{

//errorApp is an IApplicationBuilder, this is an exception pipeline.

// do stuff, write content.Response here etc.

});

* <IApplicationBuilder object>.UseStatusCodePages(); , enables displaying of static text only responses that have error code (from 400 to 599) in them. By default, it is upto the browser to display the status codes but using this middleware component .net core responds with the status code instead.

This method has an overload that accepts a content type string and a format string, for example:

app.UseStatusCodePages(

"text/plain", "Status code page, status code: {0}");

Another overload is a lambda,

app.UseStatusCodePages(

async context=> {

//context is of type StatusCodeContext

}

);

Here context contains HttpContext and we can write a response and/or read the request data on/from it. This is a separate pipeline.

* <IApplicationBuilder object>.UseStatusCodePagesWithRedirects(‘formatted string route with {0} as error code’) Redirects on error statuscodes to the given route and passes it the error code. That route typically returns a normal page with status code 200.
* <IApplicationBuilder object>.UseStatusCodePagesWithReExecute(); Same as WithRedirects except it returns the status code normally but writes response body taken from the given route.

For Example:

app.UseStatusCodePagesWithReExecute("/MyStatusCode2", "?code={0}");

This method must come before app.UseRouting();

* Sample page to read the error of ReExecute,

[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]

public class MyStatusCode2Model : PageModel

{

public string RequestId { get; set; }

public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

public string ErrorStatusCode { get; set; }

public string OriginalURL { get; set; }

public bool ShowOriginalURL => !string.IsNullOrEmpty(OriginalURL);

public void OnGet(string code)

{

RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;

ErrorStatusCode = code;

var statusCodeReExecuteFeature = HttpContext.Features.Get<

IStatusCodeReExecuteFeature>();

if (statusCodeReExecuteFeature != null)

{

OriginalURL =

statusCodeReExecuteFeature.OriginalPathBase

+ statusCodeReExecuteFeature.OriginalPath

+ statusCodeReExecuteFeature.OriginalQueryString;

}

}

}

* To disable status code pages for a given controller method, use
* // using Microsoft.AspNetCore.Diagnostics;
* var statusCodePagesFeature = HttpContext.Features.Get<IStatusCodePagesFeature>();
* if (statusCodePagesFeature != null)
* {
* statusCodePagesFeature.Enabled = false;
* }
* HTTP Server also handles exceptions. These are the exceptions raised by the app before a response is sent and when they aren’t caught by anything, in that case 500 – Internal Server Error is sent.
* <IServiceCollection object>.AddDatabaseDeveloperPageExceptionFilter(); generates DB related exception’s pages when they occur. Only works in dev env.
* Exception Filters, part of Filters, handle exceptions on per-controller or per-action basis. UseExceptionHandler is preferred over them.

• HTTP Request: An implementation IHttpClientFactory is available for creating HttpClient instances, this factory allows us to make http requests to external servers.

* To use it,

<IServiceCollection object>.AddHttpClient();

Then request it from the DI container using IHttpClientFactory,

* Perform a basic httpGET request.

…async Task someMethod()

{

var request= new HttpRequestMessage(HttpMethod.Get, <full url>);

request.Headers.Add(<header name>, <header value>);

var client = <IHttpClientFactory object>.CreateClient();

var response = await client.SendAsync(request);

if(response.IsSuccessStatusCode)

{

using var responseStream = await response.Content.ReadAsStreamAsync();

//do something with response data.

}

else

{

//put some error

}}

* Named Clients:

…AddHttpClient(“<someName>”,obj=>{

obj.BaseAddress= new Uri(‘domain’);

obj.DefaultRequestHeaders.Add(…);

});

To use it,

…

var request=new HttpRequestMessage(HttpMethod.Get,”<subdirectory and not full url>”) ;

var client = <IHttpClientFactory object>.CreateClient(‘<someName>’);

var response= await client.SendAsync(request);

* Typed Clients:

First create a class that has an HttpClient,

public class GitHubService

{

public HttpClient Client { get; }

public GitHubService(HttpClient client)

{

client.BaseAddress = new Uri("https://api.github.com/");

// GitHub API versioning

client.DefaultRequestHeaders.Add("Accept",

"application/vnd.github.v3+json");

// GitHub requires a user-agent

client.DefaultRequestHeaders.Add("User-Agent",

"HttpClientFactory-Sample");

Client = client;

}

}

Then,

…AddHttpClient<GithubService>();

Instead of creating and getting an IHttpClientFactory object, now we request for GithubService directly from DI container.

Typed clients also support an overload,

…AddHttpClient<GithubService>(c=>

{

c.BaseAddress=….

}

);

* For post requests, one way to send body is by using StringContent(<JsonSerializer.Serialize bject>, Encoding.UTF8, ‘application/json’); on <IHttpClientFactory object>.PostAsync();
* Other HttpMethods such as PUT, DELETE and PATCH are supported as well.
* Delgating Handlers: We can chain multiple HttpClients to handle HTTP requests, they make a middleware of their own.

To create a custom HttpDelegating handler,

public class myHandler: DelegatingHandler

{

protected override async Task<HttpResponseMessage> SendAsync(HttpRequestMessage request, CancellationToken \_)

{

if(!request.Headers.Contains(‘myKey’))

{

return new HttpResponseMessage(HttpStatusCode.BadRequest)

{

Content= new StringContent(“Doesn’t have myKey”);

};

}

return await base.SendAsync(request,\_);

}

}

To use it,

<IServiceCollection Object>.AddTransient<myHandler>();

<IServiceCollection Object>.AddTransient<myHandler2>();

<IServiceCollection Object>.AddHttpClient(…).AddHttpMessageHandler<myHandler>(). AddHttpMessageHandler<myHandler2>();

And simply use IHttpClientFactory object anywhere.

Here, when a request is sent it first reaches the default HttpClient handler then myHandler and then myHandler2, inversely when a response comes it first comes into myHandler2, then myHandler and lastly default HttpClient.

This is how a middleware of HttpClients is formed.

* Handle transient faults: We can handle exceptions rasied by transient handlers such as HttpRequestException, HTTP 5xx and HTTP 408 exceptions with AddTransientHttpErrorPolicy.

For example:

…

<IServiceCollection obj>.AddHttpClient<UnreliableEndpointCallerService>()

.AddTransientHttpErrorPolicy(p =>

p.WaitAndRetryAsync(3, \_ => TimeSpan.FromMilliseconds(600)));

makes it so that on the known exceptions handler retries 3 times with 600ms delay between each retry.

We can chain multiple AddTransientHttpErrorPolicy objects, if the eception is raised even after first’s handling then the next one handles it.

* CircuitBreaker policy:

…p.CircuitBreakerAsync(5, TimeSpan.FromSeconds(30)));…

1st arg is retryCount and 2nd arg is time to wait. If a request fails 5 times, block any requests for given timespan, (blocks all client requests in the app through the same IHttpClientFactory object)

* Dynamic Policy selection: For each request we can select a different policy.

Policies are rules/configs on the HttpClient.

To use it,

var firstPolicy = Policy.TimeoutAsync<HttpResponseMessage>(

TimeSpan.FromSeconds(10));

var secondPolicy = Policy.TimeoutAsync<HttpResponseMessage>(

TimeSpan.FromSeconds(30));

services.AddHttpClient("conditionalpolicy")

.AddPolicyHandler(request =>

request.Method == HttpMethod.Get ? firstPolicy: secondPolicy);

applies first policy if request method is HttpGet, secondPolicy otherwise. Both affect the Timeout for the httpClient.

* We can set timeout duration for underlying HttpMessageHanlder(the longer a handler remains active the lesser requests need to open new connections but the resilient to dns changes it becomes). Default duration is 2 minutes, to set it,

…AddHttpClient().SetHandlerLifetime(<timespan>);

* By default, HttpMessageHandler shares cookies between requests.

To disable automatic cookie handling,

services.AddHttpClient("configured-disable-automatic-cookies")

.ConfigurePrimaryHttpMessageHandler(() =>

{

return new HttpClientHandler()

{

UseCookies = false,

};

});

HttpClientHandler() above can take many more fields,

AllowAutoRedirect=bool,

UseDefaultCredentials=bool and so on.

* We can use IHttpClientFactory in our own defined services as well.
* Header Propagation: Appends given headers to all outgoing requests.

To use it,

<IServiceCollection Object>.AddHttpClient();

<IServiceCollection Object>.AddHeaderPropagation(o=> {

o.Headers.Add(‘someheader’);

});

then,

<IApplicationBuilder object>.UseHeaderPropagation(); //after UseHttpsRedirection //and before UseRouting();

• Content root: This is the base path for the entire application, it has all exes, dlls, content files, db files and web root.

• Web Root: Web Root is the path which has static resource files which webpages can use to display stuff. The default web root path is {content root}/wwwroot. We can specify a different web root in ConfigureWebHostDefaults of Host.

* To enable access to static files,

<IApplicationBuilder object>.UseStaticFiles(); //before UseRouting.

Will enable direct static file access.

For example:  
wwwroot/images/MyImage.jpg

is accessible as

localhost:1234/images/MyImage.jpg

* It takes an overload of a StaticFilesOptions object,

// using Microsoft.Extensions.FileProviders;

// using System.IO;

…new StaticFilesOptions(){

FileProvider = new PhysicalFileProvider(

Path.Combine(env.ContentRootPath, "MyStaticFiles")),

RequestPath = "/StaticFiles"

}…

allows static files to be served from the given directory.

To access its contents,

localhost:1234/StaticFiles/<path>

* It can set response as well,

const string cacheMaxAge = "604800";

app.UseStaticFiles(new StaticFileOptions

{

OnPrepareResponse = ctx =>

{

// using Microsoft.AspNetCore.Http;

ctx.Context.Response.Headers.Append(

"Cache-Control", $"public, max-age={cacheMaxAge}");

}

});

* It can set content type based MIME content type responses using FileExtensionContentTypeProvider.

To use it,

var provider = new FileExtensionContentTypeProvider();

provider.Mappings[".myapp"] = "application/x-msdownload";

provider.Mappings[".htm3"] = "text/html";

provider.Mappings[".image"] = "image/png";

provider.Mappings[".rtf"] = "application/x-msdownload";

provider.Mappings.Remove(".mp4"); // Remove MP4 videos mapping.

Then

…

app.UseStaticFiles(…   
{

ContentTypeProvider= provider;

}  
);

…

when a file of a mapped type is returned then MIME content type is set to the given string to the provider. It will replace any default mappings, like .rpf is already mapped in FileExtensionContentTypeProvider. Remove removes the mime type configuration for an extension.

* StaticFileOptions also has ServeUnknownFileTypes = <bool> and DefaultContentType= <string> parameters. This allows Static File middleware to return files of all types, it understands about 400 file content types and if a requested file isn’t of one of those types then the request is pass to the next middleware component. If no middleware handles the request then 404 is response is returned.
* Generally we use UseStaticFiles before Authentication and Authorization middleware components to serve the files without the request being authorized, but we can use it afterwards to enable them. However, the wwwroot files only use authentication and not authorization.
* It is recommended to use FallbackPolicy if authentication and authorization are used on static files.

To do so,

<IServiceCollection object>.AddAuthorization(options =>

{

options.FallbackPolicy = new AuthorizationPolicyBuilder()

.RequireAuthenticatedUser()

.Build();

});

Fallback policy applies auth policy to all methods/controllers that don’t specify their own, here we make sure the request is always made by an authenticated user or the controller/method uses [AllowAnonymous].

* Directory browsing allows listing contents of a directory,

To enable it,

<IServiceCollection object>.AddDirectoryBrowser();

and then

<IApplicationBuilder object>.UserDirectoryBrowser((new DirectoryBrowserOptions

{

FileProvider = new PhysicalFileProvider(

Path.Combine(env.WebRootPath, "images")),

RequestPath = "/MyImages"

});

where the RequestPath and FileProvider are same as UseStaticFiles. (In this case it is wwwRoot/images folder)

Now when <hostname>/<RequestPath> url is sent then a webpage is returned with all contents of the directory.

* <IApplicationBuilder obj>. UseDefaultFiles();

returns contents of a default file which is when no file path is given in the URL (like https://localhost:1234), should be used before UseStaticFiles.

By default, it looks for

default.htm

default.html

index.htm

index.html

files in wwwroot in that order, whichever is found first is returned with its name appended to the URL.

Has an overload that takes DefaultFilesOptions object.

var options = new DefaultFilesOptions();

options.DefaultFileNames.Clear();

options.DefaultFileNames.Add("mydefault.html");

app.UseDefaultFiles(options);

app.UseStaticFiles();

clears default filenames and adds mydefault, it looks for this page when no file path is given to the URL.

* <IApplicationBuilder obj>.UseFileServer();

Combines the effects of UseStaticFiles , UseDefaultFiles and optionally UseDirectoryBrowser()

To use it,

<IApplicationBuilder obj>.UseFileServer(enableDirectoryBrowsing: true);

If directory browsing is set to true then <IServiceCollection object>.AddDirectoryBrowser(); must be used as well.

One overload takes a lambda,

// using Microsoft.Extensions.FileProviders;

// using System.IO;

<IApplicationBuilder obj>.UseFileServer(new FileServerOptions{

FileProvider = new PhysicalFileProvider(

Path.Combine(env.ContentRootPath, "MyStaticFiles")),

RequestPath = "/StaticFiles",

EnableDirectoryBrowsing = true

});

This does the following,

<hostname>/ -> <hostname>/index.cshtml does what UseDefaultFiles() would,

<hostname>/StaticFiles -> <hostname>/StaticFiles/ appends ‘/’ as directories need to have ‘/’ at the end, for ‘StaticFiles’ it appends that automatically but any deeper directories require ‘/’ at the end.

<hostname>/StaticFiles/MyImages/image.jpg returns the image, since it isn’t a directory it doesn’t need ‘/’ at the end.

* It is recommended to not use any of the file sharing methods as it can be a security hazard. Even so if it is used then we must try not to put any code file, including .cs and .cshtml inside the web root.

• Routing: A ‘route’ is a URL pattern mapped to a handler, this handler can be a Razor page, an action method in a controller or a middleware.

Routing is responsible for matching incoming HTTP request to app’s endpoints. When a request arrives, the URL is given to the endpoints and the first endpoint to accept it handles the entire request.

Basic:

…

<IApplicationBuilder object>.UseRouting(); //to enable url matching

<IApplicationBuilder object>.UseEndpoints(endpoints=>{

endpoints.MapGet(‘/’, async context => {

await context.Response.WriteAsync(‘Hi’);

})

});

Responds with ‘Hi’ when a request arrives at ‘/’ and is of type GET. Since this ‘map’ short circuits the response returns.

By default ‘/’ is <https://localhost:1234/> or rather <hostname>/,

By default, the hostname is prefixed automatically, i.e., ‘https://localhost:1234’ is prefixed automatically.

endpoints is of type ‘IEndpointRouteBuilder’.

There are other Map<type> methods as well, like MapRazorPages, MapControllers, MapPost etc.

* Route Constraints:

For this map,

MapGet(‘/hi/{name:alpha}’, <delegate>);

The first arg is called the route template,

a URL like ‘<host>/hi/abdul will be caught by this endpoint.

In {name:alpha}, ‘name’ is parameter name and ‘alpha’ is a Route Constraint as it limits what type of value is acceptable.

* Health check endpoint:

…

<IEndpointRouteBuilder>.MapHealthChecks(‘someURL’).RequireAuthorization();

Any URL that matches triggers Health Check endpoint and RequireAuthorization chains authorization to the endpoint. This is basically a url to check if the user is authorized.

RequireAuthorization here is called endpoint metadata, as it is ‘extra data’.

* UseEndpoints is always a terminal endpoint (only if a match is found).
* HttpContext has .GetEndpoint(); method that returns endpoint data.

Endpoint data is null if HttpContext object (though app.Use or app.Run etc.) is accessed before UseRouting. Endpoint is also null if routing system never finds a match, i.e.,

for this code:

// Location 1: before routing runs, endpoint is always null here

app.Use(next => context =>

{

Console.WriteLine($"1. Endpoint: {context.GetEndpoint()?.DisplayName ?? "(null)"}");

return next(context);

});

app.UseRouting();

// Location 2: after routing runs, endpoint will be non-null if routing found a match

app.Use(next => context =>

{

Console.WriteLine($"2. Endpoint: {context.GetEndpoint()?.DisplayName ?? "(null)"}");

return next(context);

});

app.UseEndpoints(endpoints =>

{

// Location 3: runs when this endpoint matches

endpoints.MapGet("/", context =>

{

Console.WriteLine(

$"3. Endpoint: {context.GetEndpoint()?.DisplayName ?? "(null)"}");

return Task.CompletedTask;

}).WithDisplayName("Hello");

});

// Location 4: runs after UseEndpoints - will only run if there was no match

app.Use(next => context =>

{

Console.WriteLine($"4. Endpoint: {context.GetEndpoint()?.DisplayName ?? "(null)"}");

return next(context);

});

if ‘/’ is url then 1 – null, 2 – hello and 3- hello is output else 1 2 4 are null.

This is to say, the UseRouting goes ahead and maps routing, and knows which/if any endpoint can handle the url.

* The core principle behind placement of UseRouting and UseEndpoints in middleware pipeline is,

Middleware components that run before UseRouting modify data routing depends upon, such as request data. Components like UseHttpMethodOverride/UsePathBase/UseRewriter etc.

Components that run after UseRouting and before UseEndpoints inspect the metadata and perform security decisions like UseAuthentication/UseAuthorization/UseCors.

Typically UseEndpoints are placed at the end of the middleware pipeline as they get processed data and can now directly use controllers/pages to handle it.

* UseEndpoints vs app.Use (Routing vs Terminal middleware): The basic diff is that UseEndpoints is straightforward, directly matches url and automatically interfaces with auth middleware whereas terminal middleware requires manually defining what executes what and manual interfacing with auth system.
* All endpoints are processed at once, there is no priority. That is, regardless of how we set up multiple UseEndpoints in middleware, they all are requested at once for matching the url. However, there is still route template precedence that assigns each route template a value based on how specific it is.

That is, an endpoint with /home/0 will have higher precedence that /home/{id:int} as it is more specific.

The order of precedence is,

More segments mean more specific

Literal text segment is more specific than a parameter segment

A complex segment is equal to a parameter segment with a constraint

Catch-all parameters are least specific.

If 2 endpoints match a url and have the same order of precedence then an ambiguous match exception is thrown.

* LinkGenerator: A singleton service in the DI that provides link generation/ url generation , it is used internally too, to separate endpoints and the URLs that access them.

A LinkGenerator object has,

GetPathByAction

GetUriByAction

GetPathByPage

GetUriByPage

The GetPath methods generate a URI containing an absolute path, the GetUri methods generate an absolute URI.

These methods only work if the uri they are pointing to is understood by the routing system. We don’t give them full hostname, scheme etc. ,only page/action + controller name and they look up the endpoints to see if address is picked up by one then the url of the endpoint is returned.

GetPathByAddress

GetUriByAddress

These methods are used to generate absolute path/uri based on provided values

For example:

public class ProductsLinkMiddleware

{

private readonly LinkGenerator \_linkGenerator;

public ProductsLinkMiddleware(RequestDelegate next, LinkGenerator linkGenerator)

{

\_linkGenerator = linkGenerator;

}

public async Task InvokeAsync(HttpContext httpContext)

{

var url = \_linkGenerator.GetPathByAction("ListProducts", "Store");

httpContext.Response.ContentType = "text/plain";

await httpContext.Response.WriteAsync($"Go to {url} to see our products.");

}

}

gets the url for controller = ListProducts and action= Store .

This terminal middleware uses LinkGenerator.

* Ambient and Explicit values for link generator:

GetPathByAction(null, null, new {id=17}); looks for a controller and method that takes id=17 and generates a url for it. This is an ambient value.

GetPathByAction("ListProducts", "Store", new {id=17}); knows where to look so it is an explicit value.

* Url.Action(‘action name’,<optional params>): Url returns an IUrlHelper while Action generates a URL by searching for the given action method with given parameters in the controllers.
* Url.Page(…): Same as Url.Action but searches for a Page.
* Route Value Invalidation: Ambient and explicit values are passed through a complex process in the url generation process that generates a url based on complicated ruleset.

The basic way to generate urls is,

{controller}/{action}/{params} is the basic way it looks at a url.

Required value names (all controllers and their methods) are combined with route parameters and each is processed left to right. If a left value in invalid then all the right values are invalidated.

* Route template reference: {…} in routes define route parameters that are bound if the route is matched, so if ‘…{id:int}’ is used and the request url is …/20 then 20 will be bound to ‘id’ parameter.
* Multiple {…} must be separated by literal values, like {…}p{…}.
* Route parameters must have a name and may have additional attributes,

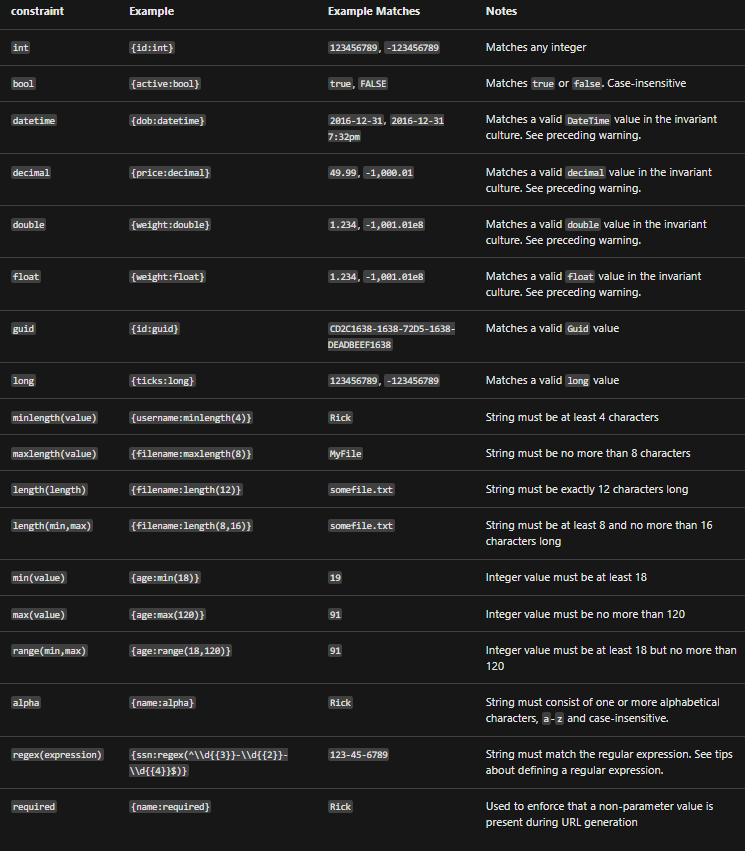
{id} and {id:int} are both valid and mean the same thing.

* Literal Text (the opposite of route template), is not case sensitive, must match the text in the url .
* To escape ‘{‘ use {{, same for }.
* catch-all: abc/{\*\*slug}, catches any url that has abc/ and anything after it. The \*\* is the catch all operator and ‘slug’ is the parameter name, for semantic reasons it’s called slug, a slug in a url is usually the end part of it’s path.

We can alternatively use single ‘\*’ but doing so generates ascii based url, like if pattern is abc/{\*slug} and url is abc/some/path then it returns abc/some%2Fpath, ‘\*\*’ preserves it and returns ‘abc/some/path’.

* Optional Route Parameter: /home/{something}.{else?} , catches /home/abc and also catches /home/abc.xyz.
* Default Route Parameter Value: {<param>=<value>}, so {something=else}, if a value isn’t specified then this value is used.
* Route Constraint: {<param>:<constraint/transformers>}, The ‘:’ constraints the param value to be of that type in the url. The constraint can be type or an inbuilt func like minlength(<int>). Multiple inline constraints can be specified with multiple ‘:’.

Types of Route Constraints:



regex(‘<regex value>’) is also a constraint.

Generally regex value in .net must start with ^ and end with & so that it matches entire route with the regex value.

* If regex is used, a timeout must be set on it. Otherwise it can be a security risk.
* Don’t use route constraints for input validation as that is not their intended use case.
* We can define custom constraints using IRouteConstraint. However their use is not recommended as the same effect can be achieved by checking values in actions/pages etc.
* Parameter transformers: {<param>:<Parameter transformer>}, transforms the value by passing it to the given transformer.

To use a transformer,

{<parameterName>:<transformerName>};

For example,

routes.MapControllerRoute(

name: "default",

template: "{controller:slugify=Home}/{action:slugify=Index}/{id?}");

where slugify is our custom transformer.

First a URL arrives then the transformer is applied to it and then the transformed URL is finally received by the function expecting the route.

* We can define custom parameter transformers by extending IOutboundTransformer,

To do so,

public class myTransformer : IOutboundParameterTransformer

{

public string TransformOutbound(object value)

{

if (value == null) { return null; }

return value.ToString()+”hoooo”;

}

}

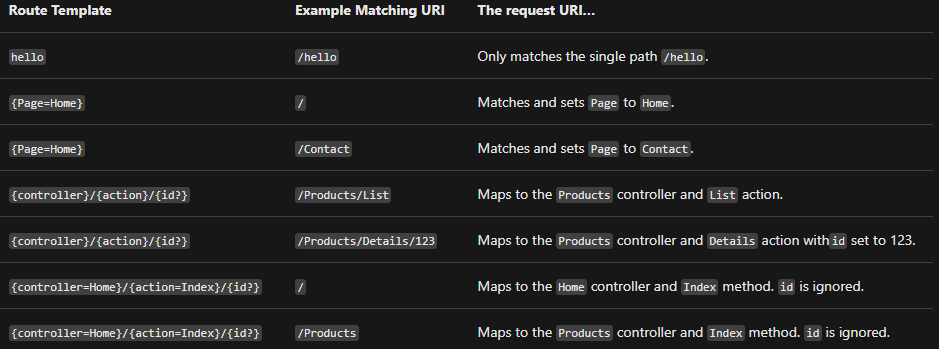
Then map the transformer,

<IServiceCollection object>.AddRouting(o=> {

o.ConstraintMap[“myTransfomer”]= typeof(myTransformer);

});

* URI and route constraints examples:



* Complex Segments:

For a route parameter {“/a{b}c{d}”},

In a URL,

First the first literal is searched for, from right to left. In this case it is c,

Everything to it’s right is now matched with {d}.

The next literal is a, now everything after a till c is now in {b}.

There is no remaining text so this is the match.

For example:

/aholaamigacnoooo url will, have parameters,

d=noooo

b=holaamiga

It parses from right to left.

For a url,

/aabcd,

d=d

b=b

But ‘a’ is still left at the start so this is not a match for the route parameter.

The matching algo is non-greedy, it matches the smallest amount of text possible in each text. If a delimiter is found in a parameter value then that fails the match.

We can use Regex to perform complex matching as well.

* RequireHost and [Host]: Enables host matching, normally hostname is not visible to the endpoints and route constraints can’t be applied to them but using either of these that can be done.

RequireHost:

<IEndpointRouteBuilder object>.someMethods.RequireHost(‘<subdomain>.<second level domain>.<top level domain>:<port>’);

will allow endpoint to work only if host matches.

Any of the values can be something or ‘\*’, ‘\*’ means any value is accepted.

Any value can be omitted and it means that part isn’t present in the url as well.

[Host(‘<subdomain>.<second level domain>.<top level domain>:<port>’)] attribute, same as RequireHost

can be applied to controllers and methods, if both have it then method’s attribute is used.

• Model Binding: Converts data from HTTP requests into strongly typed models/ method parameters etc.

For example,

[HttpGet("{id}")]

public ActionResult<Pet> GetById(int id, bool dogsOnly)

{…}

and a request like so,

<hostname>/api/pets/2?DogsOnly=true

Model binding looks at the parameter list for api/pets and gets the first param ‘id’ and its type ‘int’, then goes through the available sources and looks for a value that can be an integer or is named id (case insensitive), here url contains ‘2’ so it takes it and then does the same for dogsOnly.

Then it applies model validation for each property and sets the result in ControllerBase/PageModel.ModelState and finally ModelState.IsValid contains the result.

* Model Binding doesn’t only work for parameters of action methods/page handler methods but also for public properties of a controller/pageModel class, given they are using an attribute that allows them to use Model Binding.
* To enable model Binding for public properties,

[BindProperty]

<public prop>

//or

[BindProperties(SupportsGet=true)] //applies to all public props

<controller>

By default, GET requests don’t invoke model binding for props but we can enable that by using SupportsGet. A ‘Name’ param to BindProperty or BindProperties makes model binding look for that named source.

* Model Binding Sources:  
  1. Form Fields

2. Request Body (controllers need [ApiController] attribute)

3. Route Data

4. Query String params

5. Uploaded files

Sources are scanned in that order. 3,4 are used only for simple types. 5 is bound only to target that use IFormFile or IEnumerable<IFormFile>.

* To define custom source for a target we use [From…] attributes.

[FromQuery]: When a complex parameter uses this attribute then the properties in the controller of this method cannot use any attribute, their attributes are ignored.

[FromRoute]

[FromForm]

[FromBody]

[FromHeader]

and [FromServices] for getting from DI.

These attrs can have Name param to denote prop name, and are only applied to methods not controllers.

* To define custom sources for Model Binding, we need to create a class that implements IValueProvider, a class that implements IValueProviderFactory and then,

<IServiceCollection obj>.AddRazorPages().AddMvcOptions(options=>{

options.ValueProviderFactories.Add(new myvalueproviderFactory());

//or

options.Insert(<int>, new myvalue….());

options.ModelMetadataDetailsProviders.Add(

new ExcludeBindingMetadataProvider(typeof(System.Version)));

options.ModelMetadataDetailsProviders.Add(

new SuppressChildValidationMetadataProvider(typeof(System.Guid)));

});

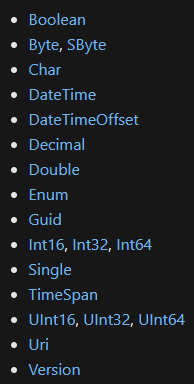
Add adds the source at the end of the default source list, Insert can specify the index to insert it at.

Here ExcludeBindingMetadataProvider excludes the given type from model binding globally, similarly SuppressChildValidationMetadataProvider excludes the given type from going into Model Validation globally.

* Model State error isn’t created if a value for a model property isn’t found in a source, instead, nullable objects get null and non-nullable objects get default() value (like 0 for int). For complex types the default constructor is used.

To store a ModelState error if they props aren’t found we use [BindRequired] attribute, it doesn’t work with JSON/XML in request body as that is handled by input formatters.

* If a value is found but is of invalid type then ModelState is flagged as invalid and stores default/null in props or automatically sends an HTTP400 response if [ApiController] is used.
* Simple Types: Model Binding looks for types then names of params/props.



* Complex types: A complex type must have a public default parameterless constructor and public writable props. Model Binding looks for Prefix.property\_name then only property\_name if not found in sources.

Prefix= param/property name of the variable that has the complex type.

We can specify custom prefix with [Bind(Prefix=”myPrefix”)SomeType xyz], it would’ve looked for xyz.something but now looks for myPrefix.something.

* Bind Attributes: None of these apply to request bodies in JSON/XML since they are processed by input formatters.

[Bind]: Can be applied to a model class or an action method parameter of a class. Bind(“x,y,z”) selects only x, y and z property to be used for model binding.

[BindRequired]: Can be applied to model class props only. Gives ModelState error if binding fails for the prop.

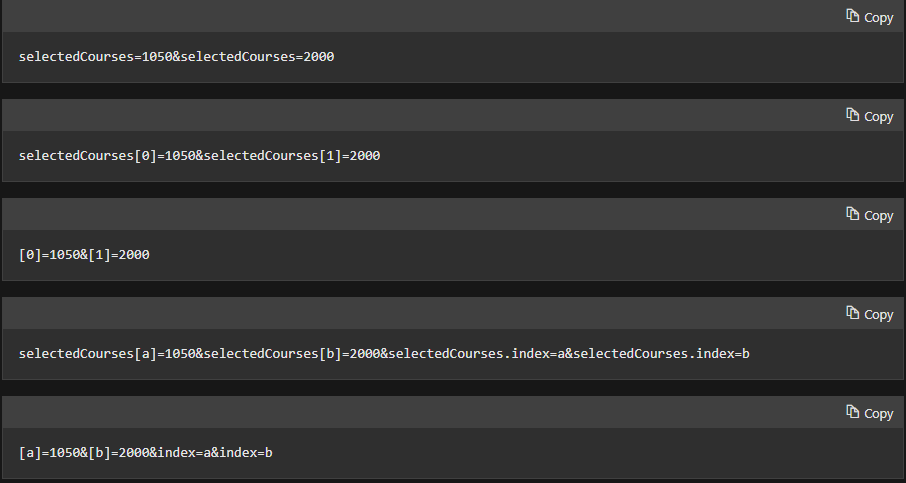
[BindNever]: Same as [BindRequired] but disallows binding of props.

* [ModelBinder] attribute: [ModelBinder(typeof(T))] applied to a param/prop looks for T in sources. We can also change Name to be looked for model binding using this attribute.
* For collections, model binding works like so,

For

public IActionResult OnPost(int? id, int[] selectedCourses) method

The possible formats are,



while for form data,

selectedCourses[]=1050&selectedCourses[]=2000

is the only format available.

Sets selectedCourses[0]=1050

selectedCourses[1]=2000

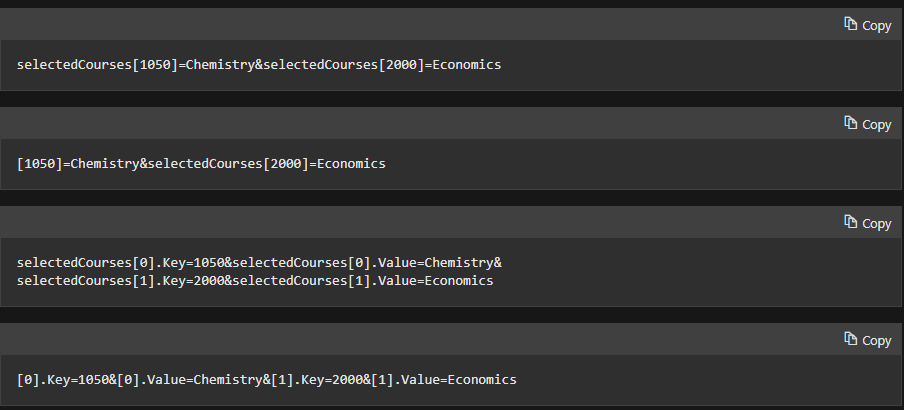
Here if index is numeric (0,1…) then it must not have gaps and must start from 0 otherwise all latter values are ignored. Like 0 then 2 would ignore 2.

* For Dictionary targets,

and this method

public IActionResult OnPost(int? id, Dictionary<int, string> selectedCourses)

The valid formats are:



* Record Type and Model Binding:

For this record,

public record Person([Required] string Name, [Range(0, 150)] int Age, [BindNever] int Id);

ModelBinding takes the record as a list of parameters instead of a single complex type.

To use records with model binding, they must

have same prop name as type for complex types (like ABC abc)

have exactly 1 public constructor

Record types allow us to create model object and validation attributes in 1 line.

Records can have a body and properties and a parametered constructor as long as the constructor applies attributes to all props and its params are named same as props.

For example,

public record Person

{

public Person([Required] string Name, [Range(0, 100)] int Age) => (this.Name, this.Age) = (Name, Age);

public string Name { get; set; }

public int Age { get; set; }

}

works.

Model attributes ([BindProperty] or [Required] etc.) get ignored if applied to record props.

* Input Formatters added like ..AddMvcOptions().AddXmlSerializerFormatters(); adds the input formatter to be used to parse request body given the input formatter accepts the content type given to Consumes attribute on an action method (in this case ‘application/xml’).

By default, JSON input formatter is included.

* Model Binding can be manually invoked for a given type by using TryUpdateModelAsync,

To use it,

myType t= new myType();

var result=await TryUpdateModelAsync<T>(t,”somename”, t=> t.someprop1, t.someprop2)

result is a bool that tells if the binding succeeded (true) or not. somename allows us to give a custom name and the delegate allows us to select props to be looked for.

* We can create our own Model Binder as well.

• Model Validation: After Model Binding is successful (like it found 0 for int id), model validation takes place and checks if the value is correct (like if 0 is in range 1-5). This happens before the action method is started.

* We can rerun model validation by,

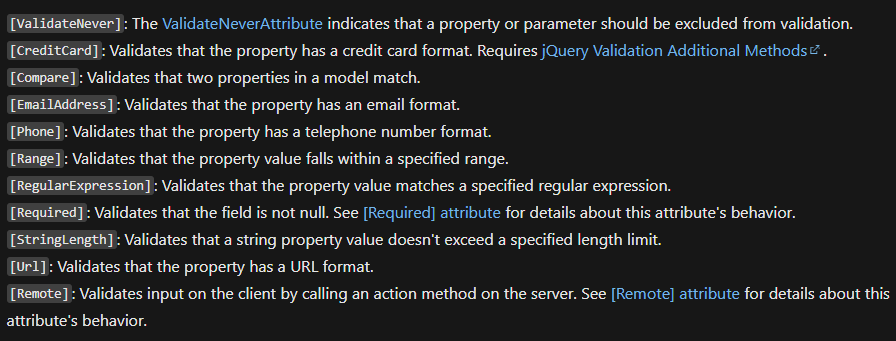
ModelState.ClearValidationState(nameOf(variable));

TryValidateModel(variable,nameof(variable));

returns a bool which tells if the validation succeeded.

* Validation Attributes: We apply these to props/params.

Built-in attributes are,



and more in System.ComponentModel.DataAnnotations, which are also applied to EF models.

All built-in attributes have an ErrorMessage parameter that allows us to provide a string with error text if validation fails.

* We use [Required] attribute to denote that there must be a value for the given prop/param. However, a non-nullable field is always valid as model binding inserts a default value. We can make it a nullable property to resolve this.

Model Binding can fail for non nullable fields and store error message,

To store custom error if model binding fails,

<IServiceCollection obj>.Add…AddMvcOptions(options=> {

options.MaxModelValidationErrors=50; options.ModelBindingMessageProvider.SetValueMustNotBeNullAccessor(

\_ => "The field is required.");

});

Here, MaxModelValidationErrors (default 200) stops running validation if that number of errors is reached.

* [Remote] attribute: Allows client to validate a field by using server side validation,

To implement Remote validation,

[Remote(action:”SomeActionMethod”, controller: “FirstController”, AdditionalFields= nameof(name)+’,’+nameof(papi))]

public string email {get;set;}

public string name{get;set;}

public string papi {get;set;}

then,

…FirstController…{

…

[AcceptVerbs(“GET”,”POST”)] //just like HttpGet and HttpPost

public IActionResult SomeActionMethod(string email, string name, string papi)

{

//check if email is correct

return Json(true); //if correct

//or

return Json(false); // or no return or return null otherwise

}

}

Now when email property is used by javascript or jquery remote method then it will call SomeActionMethod of FirstController.

The AdditionalFields pass the value of the fields to the method.

* We can create custom Validation by extending ValidationAttribute or IValidatableObject.
* We can disable client sided validation in razor pages by,

<IServiceCollection obj>.AddRazorPages()

.AddViewOptions(options =>

{

options.HtmlHelperOptions.ClientValidationEnabled = false;

});

• To specify multiple attributes, use ‘,’.

For example

[BindRequired, FromQuery] int age

• Tag Helpers: Defined later in the doc.

• Razor Pages: Page-based model, Server-rendered UI.

To enable them,

<IServiceCollection object>.AddRazorPages();

then,

<IApplicationBuilder object>.UseEndpoints(

endpoints=> {

endpoints.MapRazorPages();

}

);

MapRazorPages maps filenames to URLs,



* AddRazorPages takes an overload which provides a RazorPagesOptions object, used to set advanced configuration for RazorPages.
* WithRazorPagesAtContentRoot() can be chained to AddRazorPages method to make the root of the Razor Pages at content root (default is ContentRoot/Pages/ directory).
* WithRazorPagesRoot(‘<somepath>’) instead of WithRazorPagesAtContentRoot() to set root at custom location. These methods are quicker way of doing a part of what we can do with RazorPagesOptions object.
* Index.cshtml is the default page when URL doesn’t include a page.
* Basic cshtml page: This is a Razor Page, denoted by @page (MVC Action), the syntax is the same as a Razor view file for the most part, but unlike MVC apps Razor pages handle request directly and there isn’t a separate controller involved.

@page

<h1>Hello, world!</h1>

<h2>The time on the server is @DateTime.Now</h2>

Page that uses a PageModel,

@page

@using RazorPagesIntro.Pages

@model Index2Model

<h2>Separate page model</h2>

<p>

@Model.Message

</p>

using is the same thing as in .net, it opens namespace.

model imports the given model class.

@Model to access it’s properties.

The construction of the model class is handled by dependency injection and constructor injection.

PageModel for the above page,

using Microsoft.AspNetCore.Mvc.RazorPages;

using Microsoft.Extensions.Logging;

using System;

namespace RazorPagesIntro.Pages

{

public class Index2Model : PageModel

{

public string Message { get; private set; } = "PageModel in C#";

public void OnGet()

{

Message += $" Server time is { DateTime.Now }";

}

}

}

By convention, the cs file is named same as the page file that uses it.

i.e., for the page being Pages/myPage.cshtml the pagemodel file is called Pages/myPage.cshtml.cs.

And the PageModel classname is, <PageName>Model so for Index.cshtml it is IndexModel.

* HEAD requests are handled by an OnHead method. HEAD requests return no response body but only headers.

For example:

public void OnHead()

{

HttpContext.Response.Headers.Add("Head Test", "Handled by OnHead!");

}

* Razor pages use, Model Classes, PageModel Classes, cshtml files.
* The PageModel class:

Extends PageModel,  
constructor accepts objects from DI container.   
Methods use IActionResult (or async Task<IActionResult>)to return data, they are expected to return Page() or RedirectToPage(‘<page file location>’);

Methods are named according to their HTTPMethod, OnGet for Get, OnPost for Post and so on. For async, OnGetAsync, OnPostAsync and so on, Async suffix is optional.

Page() returns an instance of PageResult.

* There can be multiple handlers assigned to any HttpMethod, their names must be different however.
* RedirectToPage(‘file location’): If path starts with ‘.’ or without ‘/’ and ‘.’ then the url generated is relative,

For example:

‘./Index’ would assign ‘.’ with current dir and would link to Index.cshtml (yes the extension is optional). Same for ‘Index’. ‘../Index’ would go up 1 dir.

If it starts with ‘/’ then the base dir for Pages is returned,i.e., the absolute path,

‘/Index’ would link Index.cshtml from Pages/ folder (this is the base folder for pages),

The RedirectToPage supports an overload that let’s us define an Area,

RedirectToPage(<path>,new {area=”<somename>”});

* [BindProperty] attribute: Uses Model Binding on given action method. This is needed to allow model validation.
* ModelState.IsValid: Uses Model Validation in PageModel class.
* Razer pages can use ‘asp’ tag helpers and asp defined css attributes to enforce validation rules that are in Model class.

In cshtml file,

@addTagHelper \*,Microsoft.AspNetCore.Mvc.TagHelpers

<div asp-validation-summary=”ModelOnly”></div>

<span asp-validation-for=”<class>.<property>”></span>

validation summary renders errors and validation for enables validations taken of the given property in model class.

asp-page-handler=’hiData’

asp-page-handler attribute: Define which handler to call in the pagemodel class,

For example, if the attribute was used in submit method of form then it will replace the url of the form to send a POST request to the handler, the handler that will be invoked in the pagemodel class is

OnPosthiData(){}

I.e., it will search for OnPost<name><OptionalAsync> method.

These methods (OnPostXYZ) are called named handler methods.

* Layouts, Partials, Templates, TagHelpers, \_ViewStart.cshtml and \_ViewImports.cshml work similar to Razor view engine.
* @namespace sets the namespace of the page. That means @Model doesn’t need to include to include namespace anymore.
* Pages/\_ViewImports.cshtml : We can define common imports for all razor page files here, the imports used here (like @namespace, @addTagHelper etc.) are applied to all files automatically.
* When @namespace is specified in \_ViewImports then the page will get that namespace already assigned, if that namespace contains the model class then we can use it normally as if @namespace was defined in the page file else we can define @namespace in it and only add the remaining path.

For example,

ViewImports:

@namespace xyz.csz

Page file

@namespace <after csz>

* ViewData: This is used to pass data to a Razor Page from a controller, it uses the internal ViewDataDictionary. ViewBag aren’t supported in Razor Pages. TempData is supported.

We can use it as an attribute as well,

public class myClass: PageModel{

[ViewData]

public String title {get;} =”About”;

…

}

or normally in methods as,

ViewData[‘somekey’]=somevalue;

allows us to use @Model.title in the razor page after we import the pagemodel class.

We can also directly use @ViewData[‘title’] in the razor page to access the value, this is how we are to use it in the layout file.

* TempData: Similar to ViewData in both assignments(attribute exists as well) and usage (with TempData instead of ViewData). The difference is that TempData has Keep and Peek methods, normally TempData loses data after first time the data inside is read, the Peek method doesn’t trigger deletion and reads the method, the Keep method unsets the deletion of a key marked for deletion. TempData.Peek/Keep(“<key>”);

TempData is used as a cheap way for session state management as data persists through multiple requests.

By default, cookie-based TempData provider is used, it stores TempData in cookies instead of the server. The data is encrypted. Cookie limitations apply and hence it is only advised to use default TempData to store small amounts of data (<500bytes).

To use a session-based tempdata instead,

<IServiceCollection obj>.AddControllersWithViews()

.AddSessionStateTempDataProvider();

//or AddRazorPages() for razor pages.

then,

<IServiceCollection obj>.AddSession();

and lastly,

<IApplicationBuilder obj>.UseSession(); //just before UseEndpoints middleware.

* @page: The route to a page can be set with

@page “<route>” like @page “/xyz/zzz” (should start with ‘/’ or ‘~/’, ‘~’ means root)

To append a string to the path

@page “<item>” like @page “xyz” will append ‘/xyz’ to whatever the route is

We can append parameters to the route as well,

@page “{<paramname>}” will require page route to have value for parameter (follows the same rules as Route Paramters in .net core)

This also makes it so that the parameter isn’t a query parameter in the URL,, so for

@page “{id?}”

<host>/xyz/2 and <host>/xyz will be the url for it.

* To show configuration data in razor pages.

@page

@model Test5Model

@using Microsoft.Extensions.Configuration

@inject IConfiguration Configuration

Configuration value for 'MyKey': @Configuration["MyKey"]

or first add the Configuration values to DI Container using services.Configure

then

@page

@model SampleApp.Pages.Test3Model

@using Microsoft.Extensions.Options

@inject IOptions<MyOptions> optionsAccessor

<p><b>Option1:</b> @optionsAccessor.Value.Option1</p>

<p><b>Option2:</b> @optionsAccessor.Value.Option2</p>

* Filter: Razor Page filters allow us to run code after and before Razor pages. They also provide us with PageHandler..Context objects, which have access to HttpContext and can be used to set response headers (we can’t do this with middleware in Razor Pages).

These are the available Razor Page filters, and their time of invocation,

OnPageHandlerSelected: After a handler is selected, before model binding

OnPageHandlerExecuting: After model binding, before handler body executes.

OnPageHandlerExecuted: After the handler body, before it returns.

OnPageHandlerSelectionAsync: Same as Selected but async.

OnPageHandlerExecutionAsync: Same as executing but async. Must await next.Invoke() at end or will short-circuit.

Only define 1 of method types, async or sync, if both are defined then the async variant is used.

* To define a csutom Razor Page filter, extend either IPageFilter or IAsyncPageFilter (if using async handler filters)

public class myFilter: IAsyncPageFilter

{

….constructor , methods etc.,

public Task OnPageHandlerSelectionAsync(PageHandlerSelectedContext context)

{

var key = \_config["UserAgentID"];

context.HttpContext.Request.Headers.TryGetValue("user-agent",

out StringValues value);

ProcessUserAgent.Write(context.ActionDescriptor.DisplayName,

"SampleAsyncPageFilter.OnPageHandlerSelectionAsync",

value, key.ToString());

return Task.CompletedTask;

}

public async Task OnPageHandlerExecutionAsync(PageHandlerExecutingContext context,

PageHandlerExecutionDelegate next)

{

// Do post work.

await next.Invoke();

}

}

To enable this filter globally,

<IServiceCollection object>.AddRazorPages().AddMvcOptions(o=> {

o.Filters.Add(new myFilter(<constructor params>))

});

* We can apply a filter to only a set of pages as well,

<IServiceCollection object>.AddRazorPages(options =>

{

options.Conventions.AddFolderApplicationModelConvention(

"/Movies",

model => model.Filters.Add(new myFilter(<constructor params>)));

});

This applies the filter to pages inside ‘/Movies’.

* To use a filter only on a single PageModel class,

public class myPageModelClass : PageModel{

…

public override <the filter>(){

…

}

}

* Filter Attribute: The ResultFilterAttribute can be extended,

public class myHeadAttribute : ResultFilterAttribute{

…some props

public myHeadAttribute(<params>)…

public override void OnResultExecuting(ResultExecutingContext context)

{

…do something, this filter also has access to httpcontext and can set headers.

}

}

We can alternatively use OnResultExecutionAsync for async tasks.

Then to use it,

[myHead(<params>)]

public class myPageModelClass: PageModel

{}

* AuthorizeAttribute is available as filter attribute.
* Routing: Razor page apps use route and app model provider conventions to route URLs to pages.

Routing Order: Routes specify an Order for route matching

Order can be an int with value from -1,0,1,2,…n (default is 0 for any route), the lower the value the earlier in the route matching queue it is. If 2 or more routes have same Order then the most specific route is matched first, if they are identical in that regard as well then the one declared earlier is used.

Setting route order isn’t advised.

To set conventions,

<IServiceCollection object>.AddRazorPages(o=> {

o.Conventions.Add(…);

}

);

Route Model Convention:

public class GlobalTemplatePageRouteModelConvention

: IPageRouteModelConvention

{

public void Apply(PageRouteModel model)

{

var selectorCount = model.Selectors.Count;

for (var i = 0; i < selectorCount; i++)

{

var selector = model.Selectors[i];

model.Selectors.Add(new SelectorModel

{

AttributeRouteModel = new AttributeRouteModel

{

Order = 1,

Template = AttributeRouteModel.CombineTemplates(

selector.AttributeRouteModel.Template,

"{globalTemplate?}"),

}

});

}

}

}

appends {globalTemplate} parameter to all the routes.

Order of AttributeRouteModel is 1.

then simply options.Conventions.Add(new GlobalTemplatePageRouteModelConvention());

App Model Convention:

This is used to add filters (and some other things ) to all the pages.

public class GlobalHeaderPageApplicationModelConvention

: IPageApplicationModelConvention

{

public void Apply(PageApplicationModel model)

{

model.Filters.Add(new myHeadAttribute(

"GlobalHeader", new string[] { "Global Header Value" }));

}

}

Set it globally by using options.Conventions.Add(…);

Handler Model Convention:

Apply something to all the handlers

public class GlobalPageHandlerModelConvention

: IPageHandlerModelConvention

{

public void Apply(PageHandlerModel model)

{

// Access the PageHandlerModel

}

}

Folder Route Model Convention:

Apply an IPageRouteModelConvention on all the pages in a given folder,

options.Conventions.AddFolderRouteModelConvention("/OtherPages", model =>

{

var selectorCount = model.Selectors.Count;

for (var i = 0; i < selectorCount; i++)

{

var selector = model.Selectors[i];

model.Selectors.Add(new SelectorModel

{

AttributeRouteModel = new AttributeRouteModel

{

Order = 2,

Template = AttributeRouteModel.CombineTemplates(

selector.AttributeRouteModel.Template,

"{otherPagesTemplate?}"),

}

});

}

});

This code adds {otherPagesTemplate?} route parameter to all the pages in the /OtherPages.

model is the same Object that is available in Page Route Model Convention, so we can use it

model.Filters.Add(new myHeadAttribute(

"OtherPagesHeader", new string[] { "OtherPages Header Value" }));

like so as well

Page Route Model Convention:

options.Conventions. AddPageRouteModelConvention(…)

Syntax same as Folder Route but the only difference being that it applies IPageRouteModelConvention to the given page only.

Parameter Transformers:

We can use conventions to apply Parameter transformers to multiple page routes, it doesn’t transform parameters for app generated urls, only those that come in request.

options.Conventions.Add(

new PageRouteTransformerConvention(

new myParameterTransformer()));

AddPageRoute:

Use this method to map another URL to a page,

options.Conventions.AddPageRoute("/Contact", "TheContactPage/{text?}");

where /Contact is it’s default page route, it can now be accessed using either of the paths.

Configure a Filter:

model.filters.Add is quite long, a shorter way to access and configure filters is

options.Conventions.ConfigureFilter(model =>

{

if (model.RelativePath.Contains("OtherPages/Page2"))

{

return new myHeadAttribute(

"OtherPagesPage2Header",

new string[] { "OtherPages/Page2 Header Value" });

}

return new EmptyFilter();

});

Filter Factory:

Used to apply filters to all Razor Pages,

public class myFilterFactory : IFilterFactory

{

// Implement IFilterFactory

public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)

{

return new myFilter();

}

private class AddHeaderFilter : IResultFilter

{

public void OnResultExecuting(ResultExecutingContext context)

{

context.HttpContext.Response.Headers.Add(

"FilterFactoryHeader",

new string[]

{

"Filter Factory Header Value 1",

"Filter Factory Header Value 2"

});

}

public void OnResultExecuted(ResultExecutedContext context)

{

}

}

public bool IsReusable

{

get

{

return false;

}

}

}

Yes, it uses a nested class.

options.Conventions.ConfigureFilter(new myFilterFactory());

* MVC Action Filters are ignored by Razor Pages.
* Return a Partial View:

public IActionResult OnGetPartial() =>

Partial("<partial view name>");

• MVC: ASP.NET Core MVC is a framework for building webapps using the Model-View-Controller design pattern.

When a request arrives it is passed to controller which then uses models to build views and return the said views or any other data.

Model shouldn’t depend on anything, controller depends on both the other and view depends only upon model.

* Views: Views are built on Razor View Engine, i.e., cshtml files. Just like Razor Pages these views use Razor markup for syntax in the cshtml files. A Razor page builds the html file in the server and sends it to the client.

Views that are specific to a controller are created in the Views/<ControllerName> folder, views that are shared among controllers are created in Views/Shared folder

Razor Markup: Starts with @ symbol and we can place Razor Code Blocks after them which run c# code inside them.

cshtml file

@{

ViewData[“Title”]=”Yo”

var abc=2;

var address=ViewData[“Address”] as Address; //using models from //ViewData

}

<div> @ViewData[“Title”]</div>

<div> @abc</div>

We can read variables by using @ before them.

Views support DI. It is inadvised to use the DI container in Views since Controllers are supposed to give all the data.

Configuration injection: Read Configuration values.

@using Microsoft.Extensions.Configuration

@inject IConfiguration Configuration

@{

var myVal= Configuration[“key:value:value”];

}

Service Injection: Inject services from DI.

@using something

@inject IService service

Overriding Services:

Any service with a variable name that shadows an already inserted object’s name overrides it.

@using System.Threading.Tasks

@using ViewInjectSample.Helpers

@inject MyHtmlHelper Html

//overrides @Html Html helper method.

* Controllers: A controller is used to define a set of Actions, an Action (or action method) is a method on a controller which handles requests.

A Controller class inherits Microsoft.AspNetCore.Mvc.Controller. However, this isn’t necessary but doing so provides access to controller helper methods.

By convention, controller classes have ‘Controller’ suffix on their classnames.

A Controller class cannot have [NonController] attribute.

Action Method: Methods of a controller (except the ones with [NonAction] attribute) have,

Parameters which are bound to request data and are validated using model binding.

If the parameter is a model and ModelBinding is used then ModelState.IsValid can be invoked in the method to get a bool that tells if model binding and validation succeeded.

They usually return IActionResult but can return anything.

Alternatively we can return,

Empty response body return types:

HTTP Status Code helper methods, like BadRequest(); would return a response with status code 400. Other helpers are NotFound() and Ok(). All of these have overloads but if Overloads are used then Content Negotiation takes place.

Redirect, LocalRedirect, RedirectToAction, or RedirectToRoute. To redirect the request.

Predefined Content type non-empty response body return types:

FormattedResponse: Returns a JSON or a similar data exchange format to represent an Object in a specific manner. Like Json() or File() or PhysicalFile() etc.

View: Return a view (html file)

Returns View() which is an instance of ViewResult which is a type of ActionResult.

If View() is returned then the view with the <controller name - “Controller”> is searched for in the Views/<Controller name - “Controller”> folder and then in the Views/Shared folder if not found. This process is called View Discovery.

Overloads:

View(“<ViewName like “Order” or an absolute path that starts with ‘/’ or a relative path that starts with ‘.’ >”) select custom View to search and return.

View(<model class>); //same as View() but passes model to the view.

View(“<ViewName>”, <model Class>) combines the other overloads.

We can pass data to views using the model class, ViewBag, ViewData and TempData.

Since the Model is being given to a View it is called a ViewModel. It ensures strongly typed data is being passed.

Non-empty response body formatted in a content-type negotiated with the client:

This category is aka Content Negotiation. This applies whenever the method returns an ObjectResult or something other than an IActionResult.

BadRequest() and Ok() if overloaded perform it. CreatedAtRoute always performs it.

What it basically means is that the .net core tries to serialize the return result into one of the types defined by the Accept header. If there’s no Accept header then content negotiation never takes place. Inversely, if an Accept Header is given and a return value is not the types required by it then it is serialized into that value by inbuilt formatters instead.

If an inbuilt formatter cannot be found to produce response in the required media type then Http Status Code 406 Not Acceptable is returned if MvcOptions.ReturnHttpNotAcceptable is set to true, else the first formatter that can produce a response is used instead.

Action Methods can have same names, but they must have different HTTP attributes and signatures.

..method{

ControllerContext.ActionDescriptor // access Route/Controller values.

}

Controllers use DI container through constructor injection. We can directly use the DI container in action methods as well,

public IActionResult Index([FromServices] ISomething something){…}

gets something from the DI container.

We can use Options pattern with Controllers as well.

* ViewData: ViewDataDictionary object has string keys and string values (converts all types to strings). Keys are case insensitive. Values need to be casted when read if they are not of string type.

To set them,

ViewData[“something”]=something;

in controller/view.

Alternatively, we can use [ViewData] attribute on properties of models and controllers to insert them automatically in the ViewData.

* ViewBag: is a DynamicViewData object, similar to ViewData but values are dynamic type and not string so casting is required. Keys are accessed with ‘.’ instead of […].

To set them,

ViewBag.something=something;

in controller/view,

To read them,

@ViewBag.something

in view.

We can use both, ViewData and ViewBag in views.

* Model: To use a model in a view that is passed by a controller use

@model <namespace>.<classname>

and then access the class attributes using @Model in the view.

Defining the namespace and class is optional, even if we don’t define them we can still use @Model to access the passed model’s attributes, however we will not get any intellisense or compilation protection and the compiler won’t know the type of the model. @model defines a Strongly Typed Model and we get compilation support because of it.

It is recommended to have a separate Model class from a ViewModel class.

* Partial View: A Razor Markup file (.cshtml) that renders an HTML output within another Razor Markup file’s rendered output. Partial views don’t have an ‘@page’ directive at start.

It is placed inside the Views Folder for MVC and Pages folder for Razor Pages.

In MVC a returned ViewResult can be a Partial View, in Razor Pages a PageModel can return a PartialViewResult for partial views.

Partial Views can use other Partial views.

By convention, partial view filenames start with ‘\_’.

To use a partial view inside a view,

Partial Tag Helper:

Renders content asynchronously.

In the View,

<partial name = “<viewname/filename/absolute/relativepath>”>

Opens the partial view at the location of the tag.

The difference between filename and viewname is that filename includes the extension and that means the partial view is searched for only in the same folder whereas viewname uses Partial View Discovery.

Asynchonous HTML Helper:

Uses HTML Helper to render a partial view,

@await Html.PartialAsync(“<viewname/filename/absolute/relativepath>”)

PartialAsync returns an IHtmlContent wrapped inside a Task.

Alternatively,

@{

await Html.RenderPartialAsync("<viewname/filename/absolute/relativepath>");

}

This method streams the rendered output directly to the response instead of first giving it to the server and letting it unpack in the given view file then sending the html. It sends the output directly with the response and lets the client do the unpacking.

It can potentially give faster rendering performance.

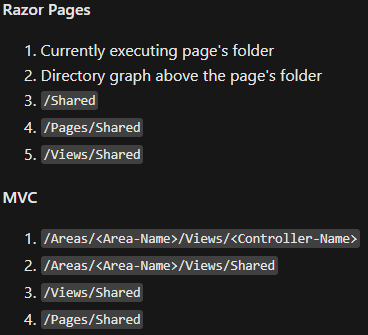
These Html Helpers also have their sync variants but usage is not recommended.

When a partial view is referenced, it receives a copy of parent’s ViewData, any changes made to it aren’t visible in the parent.

The benefit of Html Helpers over Tag helpers is that they can pass a model to the partial view or a ViewDataDictionary object (to override the default ViewData being passed) or both,

@await Html.PartialAsync(“<viewname etc.>”, <model>);

* Partial View Discovery: When a partial view is referenced without a file extension, the following locations are searched in the given order for the given file.



* Routing: Routing in Controllers is used to map URL requests with action methods, there are 3 ways of routing an MVC app.

Conventional Routing

Attribute-based Routing

Mixed Routing

Conventional Routing: We define the routing in the middleware.

<IApplicationBuilder object>.UseEndpoints( endpoints=>

endpoints.MapControllerRoute(

name:”somename”,

pattern: “<Route URL template like {controller=Home}/{action=Index}/id?>”

);

);

MapControllerRoute maps a single route.

And links a valid route like <hostname>/xyz/abc/1 to xyzController class and abc action method then passes 1 as ‘id’ to the method.

The pattern takes a Route Template Reference (defined earlier in this doc).

‘controller’ and ‘action’ parameter here, are reserved routing names and link controller and action. ‘id’ is optional and if model binding is used and url doesn’t specify it then 0 or null (if accepted) are passed.

endpoints.MapDefaultControllerRoute(); maps default routes, it basically is shorthand for

endpoints.MapControllerRoute("default", "{controller=Home}/{action=Index}/{id?}");

Order is automatically assigned in conventional Routing. Although ordering isn’t guaranteed and all endpoints are processed at once.

If MapControllerRoute maps to a specific action method only, it’s called a dedicated conventional Route. If a dedicated conventional route and conventional route get triggered for the same url then route template precedence is used.

There’s also MapAreaRoute and MapAreaControllerRoute which allow mapping Areas with URLs.

Route names such as ‘default’ etc. are used not in URL matching but in URL generation. Hence, they must be unique. Endpoint name and Route name are the same thing.

Attribute based routing: We use attributes to define routes. It is recommended to use these with REST APIs.

To enable it,

endpoints.MapControllers();

and that’s it.

In routing, route templates are used to define route to an action method or controller.

Http verb attributes(yes, a method with [HttpPost] and a method without are 2 generate 2 different routes) and [Route] are the templates in ASP.NET Core.

Example:

public class HomeController : Controller

{

[Route("")]

[Route("Home")]

[Route("Home/Index")]

[Route("Home/Index/{id?}")]

public IActionResult Index(int? id)

{

return ControllerContext.MyDisplayRouteInfo(id);

}

[Route("Home/About")]

[Route("Home/About/{id?}")]

public IActionResult About(int? id)

{

return ControllerContext.MyDisplayRouteInfo(id);

}

}

Route attribute defines which routes will trigger the given action method. With this routing, action method names are irrelevant to the mapping.

Token Replacement: In attribute based routing this is used to .net to create routes for the given action methods and controllers. The tokens are [area], [controller] and [action].

public class HomeController : Controller

{

[Route("")]

[Route("Home")]

[Route("[controller]/[action]")]

public IActionResult Index()

{

return ControllerContext.MyDisplayRouteInfo();

}

[Route("[controller]/[action]")]

public IActionResult About()

{

return ControllerContext.MyDisplayRouteInfo();

}

}

Here, [controller] is replaced by ‘Home’ and [action] is replaced by ‘Index’ for Index method and ‘About’ for About method. The URL must give <hostname>/Home/Index to match to it. Here action method names are relevant.

In the above code, if HomeController itself had an attribute

[Route("[controller]/[action]")]

<controller>

then it would apply the route to all the methods.

Applying Route Template to a controller is the same as applying that template to all the methods, i.e., essentially, controller cannot have a template of its own but it can apply a template to all the methods, and also child classes. A route template applied to a controller is also applied to it’s child controller classes. Multiple route templates can be applied to a controller/ action methods to generate multiple routes for the urls.

However, methods can still apply attributes but the interaction has a rule,

[Route("[controller]/[action]")]

<controller>{

[Route("Home/method1")]

…method1

[Route("~/Home/method2")]

[Route("/Home/method2")]

…method2

}

The URLs required to reach method1 is <hostname>/Controller/method1/Home/method1

<hostname>/Controller/method1

while the URLs required for method2 is

<hostname>/Controller/method2

<hostname>/Home/method2

That is,

If a route template starts with ‘/’ or ‘~/’ then it is not appended to the definition given by the controller, otherwise it is.

HttpVerb attributes have overloads,

[Http<Verb>(<Route template reference>)]

[Http<Verb>(<Route template reference>,Name=”RouteName”)]

For example,

[HttpGet(“{id:int}”)]

..method..

requires id of type int at the end of the URL to access this route.

If a URL doesn’t get caught by any URL then 404 is returned while if model binding fails (as in id:int getting ‘abc’) then status 400 is returned.

…

[HttpGet("/products3")]

public IActionResult ListProducts()

{

return ControllerContext.MyDisplayRouteInfo();

}

[HttpPost("/products3")]

public IActionResult CreateProduct(MyProduct myProduct)

{

return ControllerContext.MyDisplayRouteInfo(myProduct.Name);

}

…

have the same url, however since their HttpVerbs are different, different request types invoke different methods.

It is recommended to use HttpVerb attributes in REST APIs.

Attribute based routes can also use transformers, to use a transformer,

<IServiceCollection object>.AddControllersWithViews(options =>

{

options.Conventions.Add(new RouteTokenTransformerConvention(

new myTransformer()));

});

And then use define attribute routes normally.

Custom Route Attribute:

public class MyApiControllerAttribute : Attribute, IRouteTemplateProvider

{

public string Template => "api/[controller]";

public int? Order => 2;

public string Name { get; set; }

}

and then just use [MyApiController] as an attribute on methods/controllers to apply the template. Setting Order is inadvised.

IControllerModelConvention can be used to apply application model convention to routes. To use them,

public class NamespaceRoutingConvention : Attribute, IControllerModelConvention

{

private readonly string \_baseNamespace;

public NamespaceRoutingConvention(string baseNamespace)

{

\_baseNamespace = baseNamespace;

}

public void Apply(ControllerModel controller)

{

var hasRouteAttributes = controller.Selectors.Any(selector =>

selector.AttributeRouteModel != null);

if (hasRouteAttributes)

{

return;

}

var namespc = controller.ControllerType.Namespace;

if (namespc == null)

return;

var template = new StringBuilder();

template.Append(namespc, \_baseNamespace.Length + 1,

namespc.Length - \_baseNamespace.Length - 1);

template.Replace('.', '/');

template.Append("/[controller]/[action]/{id?}");

foreach (var selector in controller.Selectors)

{

selector.AttributeRouteModel = new AttributeRouteModel()

{

Template = template.ToString()

};

}

}

}

then options.Conventions.Add(

new NamespaceRoutingConvention(typeof(Startup).Namespace));

in AddControllersWithView method.

This applies the above convention to all routes except Attribute based routes.

Since it extends Attribute it can be directly applied as well,

[NamespaceRoutingConvention(“<namespace>”)]

<controller/methods>…

Now the URL to access the given controller is <namespace with ‘.’ replaced by ‘/’>/[controller]/[action]

Mixed Routing: A single app can use both types of routing, but a single controller that is attribute routed cannot be reached through conventional routing and vice versa. Both use same routing engine.

In MVC and Razor Pages, these route parameter names are reserved:

action

area

controller

handler

page

In Razor pages these are also reserved,

using

namespace

inject

section

inherits

model

addTagHelper

removeTagHelper

* Supported HTTP Verb attributes

[HttpGet]

[HttpPost]

[HttpPut]

[HttpDelete]

[HttpHead]

[HttpPatch]

* Areas: MVC feature used to group related functionality together, giving them their own namespace, routing etc. Areas allow us to have multiple controllers with same name as long as they are in diff areas.

To create an area:

endpoints.MapAreaControllerRoute("blog\_route", "Blog",

"Manage/{controller}/{action}/{id?}");

or

endpoints.MapControllerRoute("blog\_route", "Manage/{controller}/{action}/{id?}",

defaults: new { area = "Blog" }, constraints: new { area = "Blog" });

here “Blog” is the area name.

<area> value being null or “” is the same thing as not defining any area, the routes are mapped without an [Area()].

To use an area,

[Area(“<areaName>”)]

<controller/action>

puts the given controller(meaning all methods in it)/action under that area. If [Area(…)] is not defined then the controller/method doesn’t belong to any area and hence will not be picked up by area routes.

When a method inside an Area uses URL Generation through Url.Action or LinkGenerator the area route is the ambient value and will get inserted into the generated url, explicitly set the area to null or the target method’s area.

More details in a section below.

• DbContext: To initialize DbContext in .net core,

<IServiceCollection object>. AddDbContext<MyDbContextClass>(options=> options.UseSqlServer(“<connectionString>”));

Here, MyDbContextClass has a constructor that accepts an Options parameter, which we initialize in the call.

* UseInMemorDatabase() is used to use an in memory database instead.

• Session and State Management: Many techniques can be used to persist data between server and client.

Cookies: Limited, max size of a cookie is 4 KiB

Session State: In .net core, a session state is maintained by using a session id cookie, this cookie is set for the given client and is only valid for the browser the client connected from. It is sent with each request. The session data in the server is stored as cache and is by default only valid for 20 mins since the last request by a client. It is advised to not store sensitive or user-specific data in session data.

TempData: (defined in razor pages section)

Query Strings: Small amount of data can be passed around by using these in the url.

HttpContext.Items: It is a collection used to pass data between middleware components, we use it like ViewData dictionary object except it is commonly used to pass data between middleware components. It is reset on each new request.

Response Cache: (explained later in the doc)

* Configure Session State: To set up an in-memory session state:

<IServiceCollection obj>.AddDistributedMemoryCache();

<IServiceCollection obj>.AddSession(option=> {

option.IdleTimeout= Timespan.FromSeconds(10);

option.Cookie.HttpOnly=true;

option.Cookie.IsEssential=true;

});

options is an object of SessionOptions.

We can use the ..Cookie to define a lot more options such as Cookie name etc.

<IApplicationBuilder obj>.UseSession(); //just before UseEndpoint middleware.

HttpContext.Session to get the value of the Session state. Only available after UseSession middleware component is used.

ISession.Get(ISession, String) , similarly there’s GetInt32, GetString and their Set methods to set value in it.

• Layout: Layout is a razor view file that has the most common elements in it. It is used to provide a common ‘layout’ for other pages to use. Filename is usually \_Layout.cshtml and is stored in ‘Pages/Shared/’ or ‘Views/Shared/’ depending on if the app is Razor Pages app or MVC app. Layouts can use partial views and can do whatever a razor view can, even reference another layout.

A sample layout file:

<!DOCTYPE html>

<html>

<head> hello</head>

<body>

<h1>

@RenderBody()

</h1>

<h2>

@RenderSection(“<section name>”,required: <bool>)

<h2>

</body>

</html>

@RenderBody() is replaced by the content of the page that uses this layout.

@RenderSection() is replaced by the content of the section. Required parameter allows it to throw if the section isn’t found.

To use the layout file,

…myPage

@{

Layout= “\_Layout”; //’\_Layout’ is the layout file name. It can have //absolute/relative/filename and uses the same discovery process used for //views

}

Layout file can call @IgnoreBody() or @IgnoreSection() method to disable rendering of body/section from the view that uses it.

* Section: A Razor Section is a part of a view that has some content, it’s like a partial view except it is simpler to use. They are defined in the file that uses the layout and used in the layout file. Sections can call partials inside them.

To define one, Syntax:

@section <section Name>{

..cshtml content

}

To use one, Syntax:

In the layout file that is used by a view that defined the section,

@RenderSection(“<section Name>”,required: <bool>)

* ViewImports: Views and pages can import many namespaces, for describing common imports, \_ViewImports.cshtml is used. Any import inside it is imported into the view/page that uses it. It is generally placed in ‘Views/’ or ‘Pages/’ folder, and is automatically applied to all the views/pages in directories and files below and around it. Multiple \_ViewImports.cshtml may exist, and they combine. (Not placed in shared folder)

The supported tag helpers, and their combination rules,

@addTagHelper : all run, in order of specification

@removeTagHelper : same as addTagHelper.

@tagHelperPrefix : The viewimports that is closest to the view and has this will override all tagHelperPrefix of the other viewimports.

@using: all run, duplicate ignored.

@model: same as tagHelperPrefix.

@inherits: same as tagHelperPrefix.

@inject: same as tagHelperPrefix but only overrides tags with similar property name.

Sample \_ViewImports.cshtml file,

@using WebApplication1

@using WebApplication1.Models

@using WebApplication1.Models.AccountViewModels

@using WebApplication1.Models.ManageViewModels

@using Microsoft.AspNetCore.Identity

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

* ViewStart: Code that needs to run before any view (except in layout files and partial views) is placed in it. \_ViewStart.cshtml follows the same rules for file placement and combination as ViewImports.

For ex:

\_ViewStart.cshtml

@{

Layout=”layout”;

}

will apply this layout to all the views in files and directories below and around it.

• Areas: They are a .net feature that help organize related functionality into a group.

We use areas to split big regions of an app, an area has its own controllers,views and pages so it is like a component for the app.

To setup an area,

create an Area (right click Project in Solution Explorer -> Add -> Area (and give it some name))

Now assign any controller inside/outside the area the [Area(<area name>)] attribute.

Lastly, route to the area.

endpoints.MapControllerRoute(

name: "MyArea",

pattern: "{area:exists}/{controller=Home}/{action=Index}/{id?}");

Here, exists constraint mean that the given areaname in the url must exist.

Alternatively, use

endpoints.MapAreaControllerRoute(

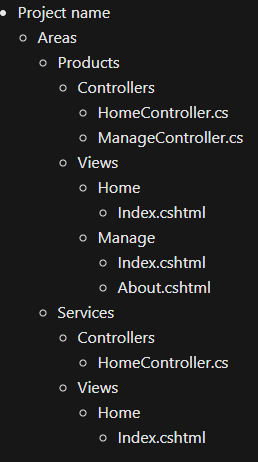
name: "MyAreaServices",

areaName: "myAreaName",

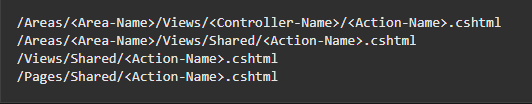
pattern: "Services/{controller=Home}/{action=Index}/{id?}");

to route to an area.

Sample folder structure for 2 areas:



* For an area, view discovery works in this order.



* ViewStart and ViewImport is applied to the entire app (including areas) only if it is in the application root folder (that is the folder containing Startup.cs).
* The area view discovery can be modified using

<IServiceCollection obj>.Configure<RazorViewEngineOptions>(options =>

{

options.AreaViewLocationFormats.Clear();

options.AreaViewLocationFormats.Add("/MyAreas/{2}/Views/{1}/{0}.cshtml");

options.AreaViewLocationFormats.Add("/MyAreas/{2}/Views/Shared/{0}.cshtml");

options.AreaViewLocationFormats.Add("/Views/Shared/{0}.cshtml");

});

* In RazorPages, Areas/<area name>/Pages is required to be structure.

• Razor Syntax: The syntax used by the razor files, stored in cshtml files. The razor part of the cshtml files, is processed by the server while the non razor part is sent as-is.

Syntax for razor part:

@ <statement> //called Implicit Razor Expressions.

@@ to escape @

@ in email is not parsed.

Implicit Razor Expressions must not have spaces, unless they have have await.

Statements can’t have <> (as in <int>)

@(…) //called Explicit Razor Expressions (ERE)

can have spaces.

can be used to concatenate, [xyz@(abc.c)](mailto:xyz@(abc.c)) will replace @(abc.c) but [xyz@abc.c](mailto:xyz@abc.c) will be parsed as an email.

can have <>.

* Expression Encoding: C# expr that evaluate to a string are HTML encoded, i.e., they are passed through ToString and encoded. If they evaluate to IHtmlContent then they are rendered through IHtmlContent.Write() method.

For example:

@(“<p>Hello</p>”)

will be encoded as &lt;p&gt;Hello&lt;/p&gt;

will be parsed by browser as <p>Hello</p>.

@Html.Raw(“<p>Hello</p>”) will directly send the content as html. However, it is inadvised to use this as it won’t be sanitized and is a security threat.

* Razor Code Blocks: @{…} All features of ERE plus can have multiple lines and more c# syntaxes. Though they aren’t rendered unless they have markup, we use them to declare variable which are visible outside the blocks as well.

Local Functions: In code blocks functions can be used as well,

..cshtml file

@{

void myName(string pp)

{

<p> Hello @pp</p>

}

myName(“yo”)

}

will replace the code block with <p>Hello yo</p>.

The part of Code blocks that have html transition into html while the rest is C# so it has both languages.

* Razor text tag: <text> </text> is used to render html content without any html tags, i.e., raw text. Any text inside it is parsed and then laid out as-is in the html doc.
* Explicit line transition: @: , prepend this to a line inside the code block to get it rendered as html (after getting parsed for any razor parts in it).

@{

@: Name: @abc.xyz

}

will render Name: something in the html doc.

* Control Structures: @if, else, else if and @switch.

@if(…)

{…}

else if(…)

{…}

else

{…}

@switch(val)

{

case 1:

…

default:

…

}

* Loop: @for(…){…} The code block part works as code blocks.

@foreach, @while, @do while.

* @using (…){…} :This uses the using of c# instead of razor’s using directive.
* @try, catch, finally: Same syntax as if else if else.
* @lock: locks section.

@lock(name)

{

…

}

* Comments: Inside code blocks, c# comment style, outside code blocks, html comment style.
* Razor Comment: @\* … \*@ . This covers both c# and html elements and is removed in the final html file.
* Razor Internals:

For a code block,

@{

var abc= “yo”;

}

<p>Yo @abc</abc>

Razor generates the following class,

public class \_Views\_Something\_cshtml : RazorPage<dynamic>

{

public override async Task ExecuteAsync()

{

var output = "yo";

WriteLiteral("/r/n<p>Yo:");

Write(output);

WriteLiteral("</p>");

}

}

Razor Directives: These are applied to this generated class,

@attribute [<attribute>]

@code {…} //only applies to razor components, adds given fields/props/methods //to a component

@functions {…} // same as code but for generated class. Use @code for razor //components. Both can have markups.

@implements <interface> extends the given interface on the generated class.

@inherits <classname> extends the given class on the generated class.

@model and @inherits can be used in the same view.

@inherits <class<TModel>> TModel passes the type of Model

@inject <IserviceName> : to inject a service from DI into the generated class.

@layout: only used in razor components and for Blazor layouts.

@model <namespace.class>: Specify the type of Model passed to a view/page. When this is specified the generated class extends RazorPage<TModel> instead of dynamic.

@namespace <somenamespace>: sets the namespace of the generated class, depending on what this is the @model must be specified. When multiple import views have different namespaces then the file closest to the the view/page is used to set the namespace.

@page: In .cshtml files it indicates the file is a Razor page. In razor components(.razor) it indicates the razor component will handle request directly.

@preservewhitespace: Only applicable in .razor files, takes bool value, removes preceding/succeeding whitespaces from codeblocks, elements and RenderFragment.

@section: For sections.

@using <namespace>: Opens the namespace for the view.

…and a few more but they are limited to .razor files.

* Templated Razor Delegates: @<tag>…</tag> Allows us to place dynamically generated content inside the tags.

For example:

@{

Func<dynamic,object> petTemplate= @<p>pet is @item</p>;

var pets= new List<String> {

“abc”,

“b”,

};

}

where item is provided by the Func delegate.

@foreach(var pet in pets)

{

@petTemplate(pet)

}

will work as intended.

Or use the HtmlContentBuilder.

@using Microsoft.AspNetCore.Html

@functions {

public static IHtmlContent Repeat(IEnumerable<dynamic> items, int times,

Func<dynamic, IHtmlContent> template)

{

var html = new HtmlContentBuilder();

foreach (var item in items)

{

for (var i = 0; i < times; i++)

{

html.AppendHtml(template(item));

}

}

return html;

}

}

<ul>

@Repeat(pets,3, @<p>@item</p>)

</ul>

* Tag Helpers: The @addTagHelper adds a tag helper to a view, remove removes it and @tagHelperPrefix specifies a tag prefix to enable Tag Helper support and to make Tag Helper usage explicit.
* We can inspect the generated razor class in compiled file directory.

For Pages/Index.cshtml that is obj/Debug/netcoreapp2.1/Razor/Pages/Index.g.cshtml.cs

• Razor Class Library: Created as a new project Visual Studio RCL provides all the razor files, views, components, page models etc. That can then be reused across apps. Apps can include RCL and override whatever it brings, if a view/partial view/page is found in both, app and RCL then the ones in app take precedence.

After creating an RCL project, simply add it as reference in the webapp project. That’s it, the file structure in the RCL directly applies over the webapp’s file structure, so any layouts/shared files etc. go into the folder they would as if they were created in the webapp.

* If RCL uses Razor Pages then enable Razor pages in **ConfigureServices** and **Configure** methods.
* If the RCL needs static assets then include them in the wwwroot folder of the RCL (after creating the folder).
* dotnet pack to pack the RCL.
* Typescript integration is supported as well. (<https://docs.microsoft.com/en-us/aspnet/core/razor-pages/ui-class?view=aspnetcore-5.0&tabs=visual-studio#typescript-integration-1>)
* The static files of the RCL go into ‘\_content/<PACKAGE ID>/’ directory, by default it is \_content/<assemblyname like xyz.class.lib>/’ but if a <PackageId> is specified in its project file then that is used instead.

Also, use <IApplicationBuilder obj>.UseStaticFiles();

• Tag Helpers: They provide ability to modify HTML elements in razor files. They are like @Model or other methods except they provide predefined functionality.

To include them,

..cshtml file

@addTagHelper <helpers to include>, <namespace.class of taghelper>

Like,

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

where \* means all tag helpers.

We can selectively include-all as well,

@addTagHelper Microsoft.AspNetCore.M\*, Microsoft.AspNetCore.Mvc.TagHelpers

would include all the tag helpers inside the Microsoft.AspNetCore. namespace where ‘M’ is the starting letter of the namespace/class.

* @removeTagHelper has same syntax as addTagHelper, removes tag helpers defined with the first arg.

We can disable tag helper for a given HTML element using ‘!’.

For example:

<!p asp-for=’Email’ class=’aaaaa’>yo </!p>

* @tagHelperPrefix : Tag helpers only work on elements that have the given prefix.

For example,

..cshtml file

@tagHelperPrefix th:

<th:p asp-for=”text”>yo</p>

<p asp-for=”text”>yo</p>

The tag helpers will only work for the first line and not for the 2nd one.

* Tag helpers don’t allow c# in the attribute declaration area of html elements.

<input asp-for="LastName"

@(Model?.LicenseId == null ? "disabled" : string.Empty) />

invalid

<input asp-for="LastName"

disabled="@(Model?.LicenseId == null)" />

valid

* Html Helper vs Tag Helper:

For Html Helper:

@Html.Label("FirstName", "First Name:", new {@class="caption"})

The tag helper is,

<label class="caption" asp-for="FirstName"></label>

* Creating our own Tag Helper: We can use the ITagHelper but TagHelper is set up to allow most tasks.

public class EmailTagHelper: TagHelper

{

public override void Process(TagHelperContext context, TagHelperOutput output)

{

output.TagName = "a"; // Replaces <email> with <a> tag

}

}

And that’s it. Use it like any other tag helper and it would replace the tags it does.

By default, a tag helper replaces the tag with the class’ root name (which is <classname – “TagHelper”>) so output.TagName replaces ‘email’ tag. Alternatively if class was named “email” then that would work as well.

We can use ProcessAsync for async task.

Attribute Parsing: The tag helper class takes attributes from the tag it is targeted at and assigns them to their PascalCase counterparts in the class.

For example:

<email mail-to=”myMail”>yo</email>

would need,

public class EmailTagHelper: TagHelper

{

private const String domain=”@id.com”;

public String MailTo{get;set;}

public override void Process(TagHelperContext context, TagHelperOutput output)

{

output.TagName = "a"; // Replaces <email> with <a> tag

var addr= MailTo+ domain;

output.Attribute.SetAttribute(“href”,”<mailto:>”+addr);

output.Content.SetContent(addr);

}

}

would generate

<a href="mailto:myMail@id.com">[myMail@id.com</a](mailto:myMail@id.com%3c/a)>

The kebab-case attribute passes it’s value into the variable with it’s PascalCase variant in the class. The SetAttribute and SetContent are self-explainatory.

..Attributes.RemoveAll(“<attr>”) removes the given attr from the entire html string

..PreContent.SetHtmlContent(“<start tag>”);

..PostContent.SetHtmlContent(“</end tag>”);

Custom name for tag:

[HtmlTargetElement(‘<target element name such as ‘email’>’, TagStructure= <optionally define TagStructure like TagStructure.WithoutEndTag>)]

…myTagHelperClass

The generated tag is self-closing if the tag that uses it is self-closing, otherwise for otherwise. This is called it’s TagMode.

..WithoutEndTag makes sure it is always without an end tag.

The [HtmlTargetElement(…)] has many overloads, one is [HtmlTargetElement(Attributes= “<attr name>”)]

This will pick up any tag that has the given attr.

Multiple of these mean any one of them must be true for a line to be parsed by this helper.

While,

[HtmlTargetElement(“<tagname>”,Attributes= “<attr name>” )]

means the tag name and attr name must match.

[HtmlTargetElement(Attributes= nameof(xyz))]

…myHelperTagClass..

{

public bool xyz;

…

}

This is used to set the attribute needed to invoke the helper tag same as a variable name in it (yes it reads the variable name from below it).

We can pass class objects to the tag helper as well.

@{

XYZ xyz= new XYZ();

}

<email my-object=”xyz”>

will pass xyz object to,

…taghelper class

{

public XYZ MyObject{get;set;}

}

* Tag Helper Conflicts: When one tag helper modifies a part of a document, it stores it in the cache first, when the next tag helper tries to override the same part it modifies the raw part instead of the cached part and stores it in the cache overriding the previous tag helper’s cache.

To check if a tag has been written to another tag helper,

output.Content.IsModified //returns bool.

We can set the ‘Order’ property to define how much priority one tag helper has,

..myTagHelper…{

public override int Order{

get {<someintval>;}

}

}

* Partial Tag Helper: These can be used to speed up loading.
* Pre-built Tag helpers exist for a lot of HTML elements.
* Tag helper Component: They are just like conditional tag helpers.

• Application Parts: These allow us to share an assembly containing controllers/views/razor pages etc. to another app. We can load these classes into a different project.

To use them,

First add the other project as reference,

then,

using System.Reflection;

var assembly= typeof(MySharedController).Assembly;

<IServiceCollection obj>.AddControllersWithViews().AddApplicationPart(assymbly).AddRazorRuntimeCompilation();

<IServiceCollection obj>.Configure<MvcRazorRuntimeCompilationOptions>(options=>{ options.FileProviders.Add(new EmbeddedFileProvider(assembly))});

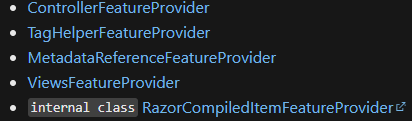
or by using AssemblyPart (a kind of initialized ApplicationPart),

var assembly= typeof(MySharedController).GetTypeInfo().Assembly;

var part= new AssemblyPart(assembly);

<IServiceCollection obj>.AddControllersWithViews().ConfigureApplicationPartsManager(apm=> apm.ApplicationParts.Add(part));

* Feature Providers: Feature (the controllers/views etc. of an application part) providers are used to get the imported application parts.



To use them,

..myControllerClass…{

myControllerClass(ApplicationPartManager apm) => (\_apm)=(apm);

..myMethod(){

var controllerFeature=new ControllerFeature();

\_apm.PopulateFeature(controllerFeature);

var list= controllerFeature.Controllers.ToList();

…

}

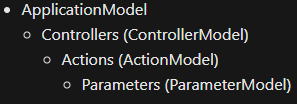
}

APM is available in the DI.

This list will have all the controllers that were imported by APM.

• Application Model: MVC apps in .net core define an app model, this object has all the components describing the app. We can modify it by using conventions/providers to define which classes are controllers, which methods are action methods and so on.

It has this structure:



* Providers are used internally and have more priority than conventions, however they are too complex for most use-cases.
* Convention: Used to modify the application model,

To use it,

using Microsoft.AspNetCore.Mvc.ApplicationModels;

first define the convention,

public class ApplicationDescription : IApplicationModelConvention

{

private readonly string \_description;

public ApplicationDescription(string description)

{

\_description = description;

}

public void Apply(ApplicationModel application)

{

application.Properties["description"] = \_description;

}

}

then, add it,

<IServiceCollection obj>.AddMvc(options=> {

options.Conventions.Add(new ApplicationDescription(“Yolo”));

});

and then use it,

…myControllerClass…

{

..somemethod(){

…

var prop=ControllerContext.ActionDescriptor.Properties[“description”];

}

}

here “description” is a custom property, these override existing props with same name. Here, the custom property is applied to the whole application model so any controller can access it, to limit it to a controller,

…myConventionAttribute: Attribute, IControllerModelConvention {

…

public void Apply(ControllerModel controllerModel)

{

controllerModel.Properties[“desc”]=”something”;

}

}

and to use it,

[myConvention()]

…myController…{

..somemethod(){

…// use it same way application model’s action descriptor was used.

}

}

Yes, we don’t apply this convention to the <IServiceCollection obj>, if we do that then all the controllers will get the convention automatically.

Similarly there’s ActionModel which is applied to action methods.

For ParameterModel,

…Apply(ParameterModel model)

{

if(model.BindingInfo==null)

{

model.BindingInfo=new BindingInfo();

}

model.BindingInfo.BindingSource= BindingSource.Path;

}

then apply this convention to a parameter,

like someMethod([myParameterConvention] int id)

The BindingSource makes it so that the only way the value would bind to ‘id’ is through path only.

<hostname>/SomeController/someMethod/123 //will work

<hostname>/SomeController/someMethod?id=123 //won’t work.

If we apply this convention to the <IServiceCollection obj> then all parameters in all action methods will get it.

* <ActionModel obj>.ActionName = xyz; to set the action method name to be used by routing. When we apply this attribute convention then this xyz will be name of the action method in the routing.
* Sample Convention to affect routing:

using Microsoft.AspNetCore.Mvc.ApplicationModels;

using System.Linq;

namespace AppModelSample.Conventions

{

public class NamespaceRoutingConvention : IApplicationModelConvention

{

public void Apply(ApplicationModel application)

{

foreach (var controller in application.Controllers)

{

var hasAttributeRouteModels = controller.Selectors

.Any(selector => selector.AttributeRouteModel != null);

if (!hasAttributeRouteModels

&& controller.ControllerName.Contains("Namespace")) // affect one controller in this sample

{

// Replace the . in the namespace with a / to create the attribute route

// Ex: MySite.Admin namespace will correspond to MySite/Admin attribute route

// Then attach [controller], [action] and optional {id?} token.

// [Controller] and [action] is replaced with the controller and action

// name to generate the final template

controller.Selectors[0].AttributeRouteModel = new AttributeRouteModel()

{

Template = controller.ControllerType.Namespace.Replace('.', '/') + "/[controller]/[action]/{id?}"

};

}

}

// You can continue to put attribute route templates for the controller actions depending on the way you want them to behave

}

}

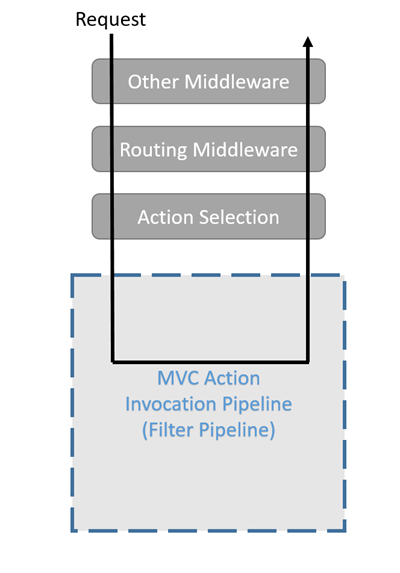
}

then simply add it normally to the <IServiceCollection object>

* Add convention to middleware:

<IServiceCollection obj>.Configure<MvcOptions>(c => c.Conventions.Add({CONVENTION}));

• Filters: Filters allow code to be run before or after specific stages in the middleware pipeline. They don’t directly work with Razor Components. Filters also make a pipeline, known as filter pipeline and interact with middleware pipeline.



For example:

public class MySampleActionFilter : IActionFilter

{

public void OnActionExecuting(ActionExecutingContext context)

{ MyDebug.Write(MethodBase.GetCurrentMethod(), context.HttpContext.Request.Path);

}

public void OnActionExecuted(ActionExecutedContext context)

{

MyDebug.Write(MethodBase.GetCurrentMethod(), context.HttpContext.Request.Path);

}

}

methods can be async as well, though they would instead need IAsyncActionFilter interface.

For a given filter type, if both async and sync version exist then async one is used.

* Filter Types:

Authorization Filter: Run first. These are the only filter than run even before Resource filters.

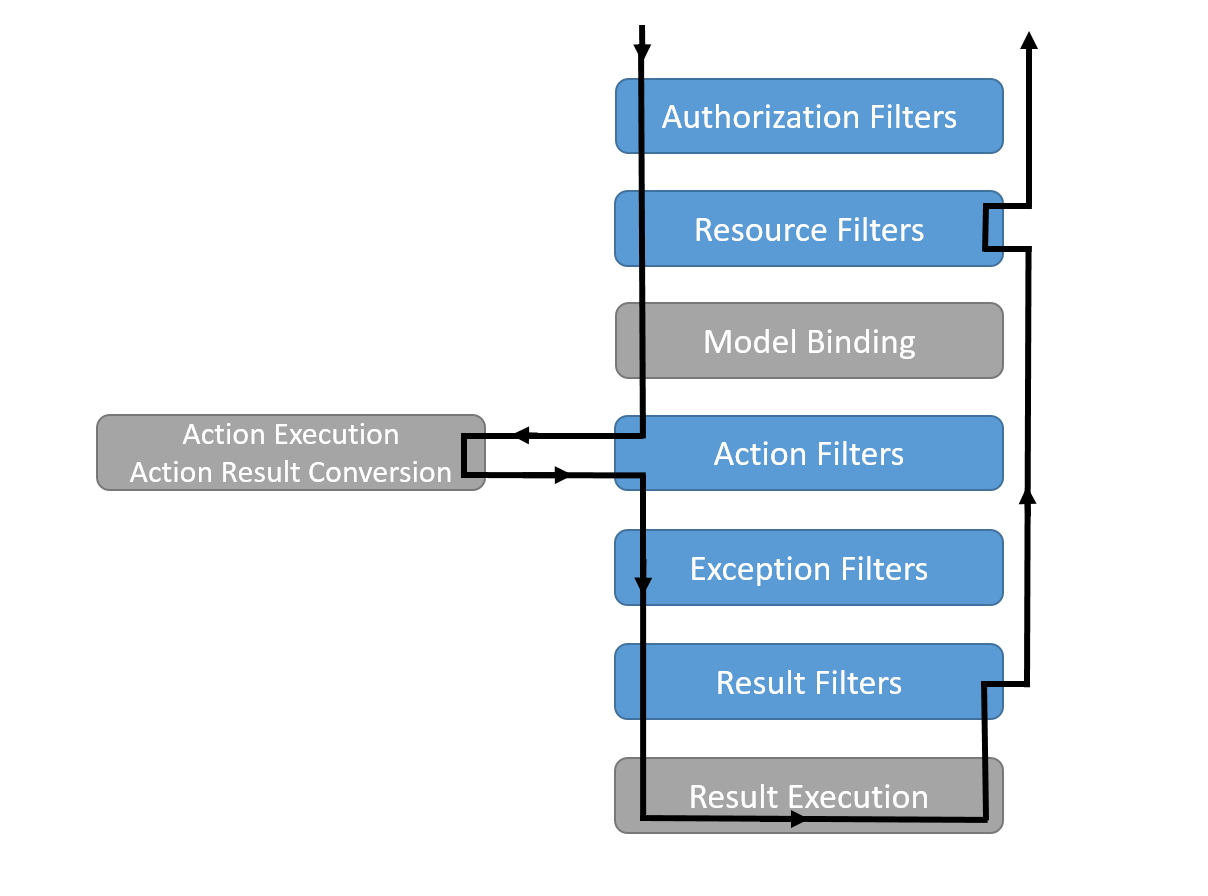
Resource Filter: Has OnResourceExecuting method which runs code before the rest of the filter pipeline and OnResourceExecuted which runs after the entire pipeline.

Action Filter: Aren’t supported by Razor Pages, can run before/after an action method and modify arguments/results.

Exception Filter: Global exception filter.

Result Filter: Just like action filter but only runs after the action has completed successfully. IAlwaysRunResultFilter and IAsyncAlwaysRunResultFilter implement IResultFilter internally, these are used to run code for all results, auth,exception and resource filters are all caught between these.

* Resource Filter: It has a special way of working, its methods take a ResourceExecutingContext object, if <ResourceExecutingContext obj>.Result is set then that short-circuits the pipeline. Resource Filters have the least Order.
* General Filter Invocation pipeline:



* Built-in filter attributes: Rather than needing to explicitly extend a filter interface and attribute separately, we can use these.

They are:

ActionFilterAttribute

ExceptionFilterAttribute

ResultFilterAttribute

FormatFilterAttribute

ServiceFilterAttribute

TypeFilterAttribute

For example:

public class AddHeaderAttribute : ResultFilterAttribute

{

private readonly string \_name;

private readonly string \_value;

public AddHeaderAttribute(string name, string value)

{

\_name = name;

\_value = value;

}

public override void OnResultExecuting(ResultExecutingContext context)

{

context.HttpContext.Response.Headers.Add( \_name, new string[] { \_value });

base.OnResultExecuting(context);

}

}

This filter requires arguments, those are given as:

[AddHeader(“Author”, “some author”)]

<controller>…

* Attributes are one way to apply filters, another is by adding them as services.

<IServiceCollection obj>.AddScoped<MyFilter>();

This is also used to use constructor injection with the filter class (it can access DI)

To apply the service filter,

[ServiceFilter(typeof(MyFilter))]

…action method…

* Filter scopes:

Filters can be applies at one of the 3 scopes,

As an attribute on controller actions. Filters Attributes can’t be applied to Razor Page Handler methods.

As an attribute on controllers/Razor Pages.

Globally, for all controllers, actions and Razor Pages.

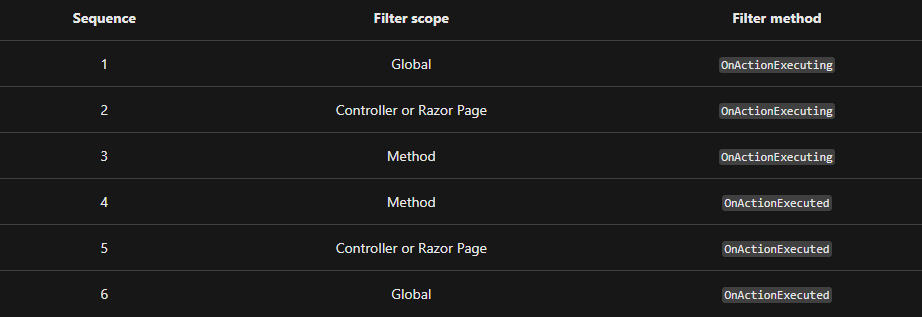
To do so,

<IServiceCollection obj>.AddControllersWithViews(options=> {

});

* Filter Nesting: When there are multiple filters for a given stage of the pipeline, their scopes determine their ‘Order’ of execution. This creates a filter ‘nesting’.

This is the sequence,



This sequence can be overridden by using IOrderedFilter, this is basically just passing Order property to any filter (even custom ones without explicit constructors). The lower order value filter runs its before method before any other filter with higher order value and after method after any other filter with higher order value.

For global filters we pass order like so,

…

options.Filters.Add(typeof(MyActionFilter), <some int value for the order>);

…

* Controller filters: Every class that inherits from Controller base class has OnActionExecuting, OnActionExecutionAsync and OnActionExecuted. Overriding these essentially creates a controller filter automatically in the controller. The first 2 methods are the same except the latter is async (and must await next()) and run before any action filters, the 3rd method runs after an action method and its action filters have been executed.
* Dependency Injection in Filters: Since most Filters are instantiated on controllers/methods directly and are Filter Attributes they can’t use DI.

If we want to use DI on filters, we need to create normal non-attribute extending filters (like extending only IResultFilter).

Then, use ServiceFilterAttribute, TypeFilterAttribute or IFilterFactory.

ServiceFilterAttribute: This requires the the filter into DI and gets the same instance as attribute from there.

<IServiceCollection obj>.AddScoped<MyFilter>();

then,

[ServiceFilter(typeof(MyFilter))]

<controller/method>

Since our filter is added as scoped dependency it must only depend on dependencies with same or larger scopes.

We can set the IsReusable parameter of ServiceFilter to let .net core know that the ServiceFilter may be reused outside its request scope. When this parameter is set the filter must only have singleton dependencies, if any.

TypeFilterAttribute: Just like ServiceFilterAttribute but it doesn’t need the filter to be added to the DI container. It can also pass custom non-DI arguments, and also use IsReusable.

Syntax:

[TypeFilter(typeof(MyFilter),Arguments= new object[] {“some arg”, “some arg 2 ”})]

<controller/method>

The rest of the args are searched in the DI container.

IFilterFactory:

This filter is applied conditionally, if any other filter (all filters are of type IFilterMetadata) is able to be casted to an IFilterFactory then the filter is applied otherwise not. This Is defined through the CreateInstance method.

public class AddHeaderWithFactoryAttribute : Attribute, IFilterFactory

{

public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)

{

return new InternalAddHeaderFilter();

}

private class InternalAddHeaderFilter : IResultFilter

{

public void OnResultExecuting(ResultExecutingContext context)

{

context.HttpContext.Response.Headers.Add(

"Internal", new string[] { "My header" });

}

public void OnResultExecuted(ResultExecutedContext context)

{

}

}

public bool IsReusable

{

get

{

return false;

}

}

}

and use,

[AddHeaderWithFactory]  
<controller/action>

* Filter on IFilterFactory: TypeFilterAttribute implements IFilterFactory, hence it can be used like so,

public myFilterAttribute: TypeFilterAttribue

{

public myFilterAttribute():base(typeof(myFilterImpl)){…}

class myFilterImpl: IActionFilter

{…}

public bool isReusable(){…}

}

Then we can use it as,

[myFilter]

<controller/method>

or

[TypeFilter(typeof(myFilterAttribute))]

<controller/method>

or

[ServiceFilter(typeof(myFilterAttribute))] //requires setting it up as service

<controller/method>

This attribute filter has another benefit, the constructor of myFilterAttribute can use DI container.

* MiddlewareFIlter: We can insert middleware components into the filter pipeline.

To do so,

..myMiddlewareClass…

{…}

[MiddlewareFilter(typeof(myMiddlewareClass))]

<controller/method>

This filter runs the middleware component twice, before model binding and after the rest of the pipeline, i.e., the same as a ResourceFilter.

* We pass ‘type’ of a filter instead of an instance everywhere because filter is a singleton and is not thread safe.

• View Components: Similar to partial views, i.e., they are also used to render a chunk of a response but have a lot more features. They can be used with both Razor Pages and MVC.

We can create ViewComponent in 3 ways,

Deriving from ViewComponent

Using [ViewComponent] attribute on a class, or deriving from one such class.

We can give this attribute a parameter named ‘Name’ to give our name to ViewComponent.

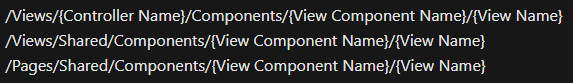
Creating a class whose name ends with ‘ViewComponent’.

We can use the attribute on a ViewComponent class to change its name.

Any way, ViewComponent classes support DI but not filters.

ViewComponent class has either an Invoke or an InvokeAsync method and return IViewComponentResult. Parameters of the method come from invocation and not model binding. ViewComponent typically returns a ViewResult() after initializing a viewmodel and passes it that. They aren’t directly reachable using URLs and can only be used internally.

* The ViewResult() it returns by default searches for a view file here,



and the filename it searches for is Default.cshtml.

* We can add more search paths,

<IServiceCollection obj>.AddMvc().AddRazorOptions(options=> {

options.ViewLocationFormats.Add(“/{0}.cshtml”);

}).SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);

Here {0} is automatically replaced by “Components/{View Component Name}/{View Name}”.

* To use a ViewComponent call this in a view,

@await Component.InvokeAsync(“<ViewComponent name – ‘ViewComponent’>”, new {<param1>=<some val>, <param2>…})

Here, the param list is passed to the InvokeAsync method of the viewcomponent.

* ViewComponent as a TagHelper: We can do so by,

@addTagHelper \*, MyViewComponentClass

then,

<vc:view-component-class prop-1=”someval” prop-2=”some val 2”>

</vc>

* ViewComponent as an IActionResult: To return ViewComponent from an action method,

…myActionMethod...

{…

return ViewComponent(“<ViewComponent name – ‘ViewComponent’>”, new {<param1>=<some val>, <param2>…});

;

…}

We will await it if it has an async Invoke method.

• File Uploading: .net core supports uploading one or more files using buffered model binding for smaller files and unbuffered streaming for larger files.

* Security steps to follow on file uploads:

1. Upload files to a separate folder and preferably separate drive than the app.
2. Use safe file name determined by the app and not take it from the client or derive it from the upload file name.
3. Allow only known file extensions to be uploaded.
4. Perform the client-side checks on the server as well.
5. Set max file size.
6. Make sure file doesn’t overwrite another file, unless intentional.
7. Run a virus/malware scanner on the uploaded content.
8. Disable file execute permission on upload file location.

* Storage Locations:

Database: For small file uploads, faster than file system.

Physical Storage: Uses File system, for larger files.

Cloud Data Storage Service

* File Upload Scenarios: 2 types,

1. Buffering: The entire file is read and stored onto the memory (if file size <=64KB) and then into disk (after exceeding default file size limit). Site crashes if the file size is too big.
2. Streaming: File is directly processed and saved onto the disk.

* Send a file to the server:

Use a Post request and set encoding type to ‘multipart/form-data’.

Give data in the body. For HTML and Razor refer docs (

<https://docs.microsoft.com/en-us/aspnet/core/mvc/models/file-uploads?view=aspnetcore-5.0#upload-small-files-with-buffered-model-binding-to-physical-storage> ).

* To save the file,

Buffered Method:

public async Task<IActionResult> OnPostUploadAsync(List<IFormFile> files)

{

long size = files.Sum(f => f.Length);

foreach (var formFile in files)

{

if (formFile.Length > 0)

{

var filePath = Path.Combine(\_config["StoredFilesPath"],

Path.GetRandomFileName());

using (var stream = System.IO.File.Create(filePath))

{

await formFile.CopyToAsync(stream);

}

}

}

// Process uploaded files

// Don't rely on or trust the FileName property without validation.

return Ok(new { count = files.Count, size });

}

Here parameter can be IFormFile, IFormFileCollection, IEnumerable<IFormFile> or list. Path.GetRandomFileName() generates a safe filename.

//Streaming method too complex to define here.

* To save file in razor pages into db using EFCore,

For this file class,

public class AppFile

{

public int Id { get; set; }

public byte[] Content { get; set; }

}

Using this model and using model binding which is automatically performed,

public class BufferedSingleFileUploadDbModel : PageModel

{

...

[BindProperty]

public BufferedSingleFileUploadDb FileUpload { get; set; }

...

}

public class BufferedSingleFileUploadDb

{

[Required]

[Display(Name="File")]

public IFormFile FormFile { get; set; }

}

Lastly, to save it to db,

public async Task<IActionResult> OnPostUploadAsync()

{

using (var memoryStream = new MemoryStream())

{

await FileUpload.FormFile.CopyToAsync(memoryStream);

// Upload the file if less than 2 MB

if (memoryStream.Length < 2097152)

{

var file = new AppFile()

{

Content = memoryStream.ToArray()

};

\_dbContext.File.Add(file);

await \_dbContext.SaveChangesAsync();

}

else

{

ModelState.AddModelError("File", "The file is too large.");

}

}

return Page();

}

* Signature Validation: We should match the extension of the file with the content of the file, the first few bytes of a file are used to determine it’s type. That is done like so,

For jpeg,

private static readonly Dictionary<string, List<byte[]>> \_fileSignature =

new Dictionary<string, List<byte[]>>

{

{ ".jpeg", new List<byte[]>

{

new byte[] { 0xFF, 0xD8, 0xFF, 0xE0 },

new byte[] { 0xFF, 0xD8, 0xFF, 0xE2 },

new byte[] { 0xFF, 0xD8, 0xFF, 0xE3 },

}

},

};

using (var reader = new BinaryReader(uploadedFileData))

{

var signatures = \_fileSignature[ext];

var headerBytes = reader.ReadBytes(signatures.Max(m => m.Length));

return signatures.Any(signature =>

headerBytes.Take(signature.Length).SequenceEqual(signature));

}

More sigs can be found at: <https://www.filesignatures.net/>

* Configure multipart body length: Default is 128MB.

<IServiceCollection obj>.Configure<FormOptions>(options =>

{

// Set the limit to 256 MB

options.MultipartBodyLengthLimit = 268435456;

});

For razor pages,

<IServiceCollection obj>.AddRazorPages(options =>

{

options.Conventions

.AddPageApplicationModelConvention("/FileUploadPage",

model.Filters.Add(

new RequestFormLimitsAttribute()

{

// Set the limit to 256 MB

MultipartBodyLengthLimit = 268435456

});

});

or

[RequestFormLimits(MultipartBodyLengthLimit = 268435456)]

<PageModel/action method>

* Kestrel max request body size: Default is 28.6MB.

webBuilder.ConfigureKestrel((context, options) =>

{

// Handle requests up to 50 MB

options.Limits.MaxRequestBodySize = 52428800;

})

For Razor Pages,

<IServiceCollection obj>.AddRazorPages(options =>

{

options.Conventions

.AddPageApplicationModelConvention("/FileUploadPage",

model =>

{

// Handle requests up to 50 MB

model.Filters.Add(

new RequestSizeLimitAttribute(52428800));

});

});

or

[RequestSizeLimit(52428800)]

<PageModel/action method>

• WebAPI: We can create RESTful Services with .net Core. REST means there’s no state maintained between requests and each request is as if it’s new.

A webapi controller extends ControllerBase. An MVC app extends Controller class which extends ControllerBase class, so an MVC app can also act as a webapi but it is better suited for handling ViewResults.

* Attributes: All MVC attributes are supported. [ApiController] attribute can be applied to the controller to enable some api specific behaviors defined by .net core.

We can apply ApiController to the entire assembly (app) by specifying it on the startup’s namespace.

[assembly: ApiController]

namespace xyz

{

public class Startup

{…

}

}

ApiController makes attribute routing a requirement. Actions can’t be reached through conventional routes defined by UseEndpoints, UseMvc or UseMvcWithDefaultRoute.

ApiController automatically returns HTTP400 response if model validation fails.

To disable it,

<IServiceCollection obj>.AddControllers()

.ConfigureApiBehaviorOptions(options =>

{

options.SuppressConsumesConstraintForFormFileParameters = true;

options.SuppressInferBindingSourcesForParameters = true;

options.SuppressModelStateInvalidFilter = true;

options.SuppressMapClientErrors = true;

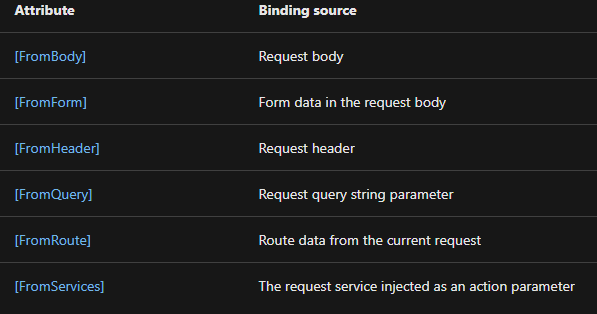
options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =

"https://httpstatuses.com/404";

});

* Binding Source Parameter: Any one of these attributes applied to a parameter in an action method means .net core should look for the value of parameter in the given source.

They are:



By default, the [ApiController] uses complex object model binder and applies inference rules to determine default data sources for parameters, i.e., it uses one of these attributes automatically if we don’t define them manually.

The rules are:

[FromBody]: This is applied to any user-defined type. A single action method can only have a single FromBody per default rules.

[FromForm]: For parameters of IFormFile/IFormFileCollection types. If this is inferred then it is also inferred that the request content type is ‘multipart/form-data’. To disable it

… options.SuppressConsumesConstraintForFormFileParameters = true;

…

[FromRoute]: If the parameter name matches a name in a route template for the action method.

[FromQuery]: For all other parameters.

Disable automatic inference:

... options.SuppressInferBindingSourcesForParameters = true;

…

* When we return NotFound() from a method then a json ProblemDetails body is automatically added as response along with HTTP404 status code.

To disable the body,

…options.SuppressMapClientErrors = true;…

[FromServices] isn’t a normal binding source, it simply provides services from DI container.

* Consumes attribute: Use this attribute to define and limit which request content type is supported by an action method.

[Consumes(“application/xml”)]

<action method>

requires Content-Type header to be ‘application/xml’. Missing or different value for the header returns HTTP415

This attribute influences routing, hence it can be used to separate routing to 2 same url methods based on different Content-Type values.

* Action Methods Return Types: We can Return IActionResult, IActionResult<T>, ActionResult, ActionResult<T> or T. All these can be async too.

With IActionResult we generally return BadRequestResult (http400), NotFoundResult(404), CreatedAtAction(201) and OkObjectResult(200). All these have shorthands/already initialized instances like BadRequest();

Since IActionResult can return many types or results we use [ProducesResponseType(StatusCodes.StatusSomething, Type= sometype)] on the action method to denote what we are returning.

If an action method with a Model return type returns null then HttpStatusCode 204 No Content is returned.

Content Negotiation takes place as normal (described in controllers section).

ActionResult allows us to just use [ProducesResponseType(<status code>)] and the type is inferred automatically. We can also return T and T will be casted to new ObjectResult(T); given we use ActionResult<T>.

To return custom http status code results, return new HttpResponseMessage(HttpStatusCode.something);

or

request.CreateResponse(HttpStatusCode.something,obj);

or

new StatusCodeResult(<int>);

or

Problem();

* Output/Response Formatting: Action Results return values that are formatted and have the appropriate Content-Type header set automatically. For example,

return Content(<obj>); returns object that is serialized to a string with Content-Type as ‘text/plain’

return Ok(<obj>), json serialized content with Content-Type as ‘application/json’.

If requrest is coming from a browser or if Accept Header contains ‘\*/\* then content is returned in JSON, unless configured. To respond according to Accept Header,

<IServiceCollection obj>.AddControllers(options =>

{

options.RespectBrowserAcceptHeader = true; // false by default

});

Inbuilt Formatter: Formatters like XMLFormatters can be enabled by using,

<IServiceCollection obj>.AddControllers()

.AddXmlSerializerFormatters();

Now if Accept Header of a request has XML type then this formatter will be used.

For better JSON formatting than System.Text.Json, use NewtonsoftJson nuget package.

First, get it from nuger pm, then

<IServiceCollection obj>.AddRazorPages/AddControllers/AddControllersWithViews().AddNewtonsoftJson();

..AddNewtonsoftJson(options=> {

…

});

to define global json formatting, or

… return Json(<obj>, new JsonSerializerSettings{

…

});

to define per-action basis serialization options.

If [Produces(“<content type>”)] attribute filter is specified on a controller/action method then the action method(s) will only return the content formatted in the given content type using its formatter.

To remove output formatter,

using Microsoft.AspNetCore.Mvc.Formatters;

<IServiceCollection obj>.AddControllers(options =>

{

options.OutputFormatters.RemoveType<StringOutputFormatter>();

options.OutputFormatters.RemoveType<HttpNoContentOutputFormatter>();

});

Removing the StringOutputFormatter disables returning of text/plain objects, either 406 Not Acceptable is returned or JSON/XML formatter are used.

Removing HttpNoContentOutputFormatter means instead of returning 204, null object is returned through the JSON/XML formatter.

FormatFilter: This filter can be used to check the existence of ‘format’ value in the route URL and pass it to the route system.

For example:

[Route("api/[controller]")]

[ApiController]

[FormatFilter]

public class ProductsController : ControllerBase

{…

[HttpGet("{id}.{format?}")]

public Product Get(int id)

{…}

…

}

Makes the output formatted in the given ‘format’ using its formatter. Kinda like [Produces()] but more powerful.



* Custom Formatters: We can define custom formatters for both serializing data (output formatters), i.e. data sent to the client and for deserializing data (input formatters), i.e., data received from the client.

To create an input formatter:

public class myInputFormatter: TextInputFormatter

{

public myInputFormatter()

{

SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("text/vcard"));

SupportedEncodings.Add(Encoding.UTF8);

SupportedEncodings.Add(Encoding.Unicode);

//use the constructor to set the supported types.

}

protectected override bool CanReadType(Type type)

{

//self descriptive,

//sample return type

return typeof(<user defined type>).IsAssignableFrom(type) ||

typeof(IEnumerable<user defined type>).IsAssignableFrom(type);

}

public override async Task<InputFormatterResult> ReadRequestBodyAsync (InputFormatterContext context, Encoding encoding )

{

//context has HttpContext, read the values from it and write to custom //model and return it using InputFormatterResult.SuccessAsync(<obj>);

}

}

This class cannot use constructor injection, however we can pass objects using HttpContext.

OutputFormatter, almost same working except the methods are named CanWriteType and WriteResponseBodyAsync, the WriteResponseBody returns void and there we read from the context and write to the HttpContext.Response.

Sometimes CanWriteResult is used instead of CanWriteType, it is done when the action method returns a model class/ there may be derived classes returned at runtime/ the derived class is to be known at runtime.

To use the custom formatter,

<IServiceCollection obj>.AddControllers(options =>

{

options.InputFormatters.Insert(0,new myInputFormatter());

//same for output

});

Formatters are evaluated in the order we insert them.

Sample Input Formatter:

public class VcardInputFormatter : TextInputFormatter

{

public VcardInputFormatter()

{

SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("text/vcard"));

SupportedEncodings.Add(Encoding.UTF8);

SupportedEncodings.Add(Encoding.Unicode);

}

protected override bool CanReadType(Type type)

{

return type == typeof(Contact);

}

public override async Task<InputFormatterResult> ReadRequestBodyAsync(

InputFormatterContext context, Encoding effectiveEncoding)

{

var httpContext = context.HttpContext;

var serviceProvider = httpContext.RequestServices;

var logger = serviceProvider.GetRequiredService<ILogger<VcardInputFormatter>>();

using var reader = new StreamReader(httpContext.Request.Body, effectiveEncoding);

string nameLine = null;

try

{

await ReadLineAsync("BEGIN:VCARD", reader, context, logger);

await ReadLineAsync("VERSION:", reader, context, logger);

nameLine = await ReadLineAsync("N:", reader, context, logger);

var split = nameLine.Split(";".ToCharArray());

var contact = new Contact

{

LastName = split[0].Substring(2),

FirstName = split[1]

};

await ReadLineAsync("FN:", reader, context, logger);

await ReadLineAsync("END:VCARD", reader, context, logger);

logger.LogInformation("nameLine = {nameLine}", nameLine);

return await InputFormatterResult.SuccessAsync(contact);

}

catch

{

logger.LogError("Read failed: nameLine = {nameLine}", nameLine);

return await InputFormatterResult.FailureAsync();

}

}

private static async Task<string> ReadLineAsync(

string expectedText, StreamReader reader, InputFormatterContext context,

ILogger logger)

{

var line = await reader.ReadLineAsync();

if (!line.StartsWith(expectedText))

{

var errorMessage = $"Looked for '{expectedText}' and got '{line}'";

context.ModelState.TryAddModelError(context.ModelName, errorMessage);

logger.LogError(errorMessage);

throw new Exception(errorMessage);

}

return line;

}

}

* JSON PATCH requests and the PATCH HTTPRequest Type: A PATCH request method provides a json body with the update type, value and target resource. Then the resource is updated with the changes. PUT is alike to PATCH except PUT replaces the entire resource and PATCH just updates part of it.

For this JSON,

{

"customerName": "John",

"orders": [

{

"orderName": "Order0",

"orderType": null

},

{

"orderName": "Order1",

"orderType": null

}

]

}

A PATCH request could be,

[

{

"op": "add",

"path": "/customerName",

"value": "Barry"

},

{

"op": "add",

"path": "/orders/-",

"value": {

"orderName": "Order2",

"orderType": null

}

}

]

Which would give this result,

{

"customerName": "Barry",

"orders": [

{

"orderName": "Order0",

"orderType": null

},

{

"orderName": "Order1",

"orderType": null

},

{

"orderName": "Order2",

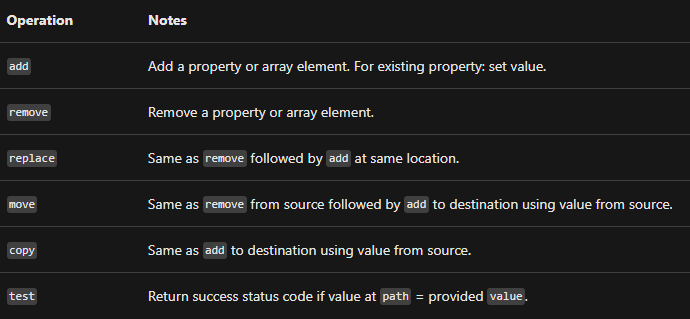
"orderType": null

}

]

}

‘op’: Operation Type, these are the possible values



‘path’: Path to resource in the JSON object, slashes between levels. Like, ‘/orders/’ where ‘/’ is root. Arrays can have index or ‘-‘ which means last element.

‘value’: if op type needs a value then provide a value.

To enable JSON PATCH support, get the NewtonSoftJson nuget package.

Then,

<IServiceCollection obj>.AddRazorPages/AddControllers/AddControllersWithViews().AddNewtonsoftJson();

We can enable the support for some formatters only by creating a new ServiceCollection. Check <https://docs.microsoft.com/en-us/aspnet/core/web-api/jsonpatch?view=aspnetcore-5.0#json-patch-addnewtonsoftjson-and-systemtextjson> for more info.

Sample HTTPPatch action method:

[HttpPatch]

public IActionResult JsonPatchWithModelState(

[FromBody] JsonPatchDocument<Customer> patchDoc)

{

if (patchDoc != null)

{

var customer = CreateCustomer();

patchDoc.ApplyTo(customer, ModelState);

if (!ModelState.IsValid)

{

return BadRequest(ModelState);

}

return new ObjectResult(customer);

}

else

{

return BadRequest(ModelState);

}

}

* Analyzer: WebAPI analyzer works with controllers that have [ApiController] attribute , to enable it, specify this in property in the project file

<PropertyGroup>

<IncludeOpenAPIAnalyzers>true</IncludeOpenAPIAnalyzers>

</PropertyGroup>

Then most common issues with the controllers will be visible in the OpenAPI generated doc.

* WebAPI Conventions: These are used to define documentation for multiple actions or controllers. They are just like [ProducesResponseType] attribute but better. These are generated using our attributes at runtime. ApiExplorer is used to generate docs by OpenAPI, they read conventions and so do API Analyer. A convention is a static type with methods which can define response types and naming requirements on actions.

We can only use 1 convention with 1 action method.

For example,

[HttpPut("{id}")]

[ApiConventionMethod(typeof(DefaultApiConventions),

nameof(DefaultApiConventions.Put))]

public IActionResult Update(string id, Contact contact){

…

}

Here, [HttpPut] isn’t a convention but a filter.

The [ApiConventionMethod] attribute adds,

[ProducesDefaultResponseType]

[ProducesResponseType(StatusCodes.Status204NoContent)]

[ProducesResponseType(StatusCodes.Status404NotFound)]

[ProducesResponseType(StatusCodes.Status400BadRequest)]

Even here, applying this attribute to all action methods is the same as applying it to the controller. Applying it to the assembly applies it to all controllers.

We can create our own conventions by defining a static class with static methods and applying some ApiConvention<Name/Type>Match attributes. Then applying the class as attribute on action methods/controllers.

* Exception Handling in WebAPI: For development environments we can use DeveloperExceptionPage but for production apps we should have a custom route for it,

First,   
use <IApplicationBuilder obj>.UseExceptionHandler(‘<route>’);

then,

[Route(‘<route>’)]

<action method>

Here, context.Error object won’t be null if an exception was caught.

We can create an exception filter as well, and apply it to all the action methods, to do so

public class HttpResponseException : Exception

{

public int Status { get; set; } = 500;

public object Value { get; set; }

}

public class HttpResponseExceptionFilter : IActionFilter, IOrderedFilter

{

public int Order { get; } = int.MaxValue - 10;

public void OnActionExecuting(ActionExecutingContext context) { }

public void OnActionExecuted(ActionExecutedContext context)

{

if (context.Exception is HttpResponseException exception)

{

context.Result = new ObjectResult(exception.Value)

{

StatusCode = exception.Status,

};

context.ExceptionHandled = true;

}

}

}

And then simply,

<IServiceCollection obj>.AddControllers(options =>

options.Filters.Add(new HttpResponseExceptionFilter()));

Client Error Response: Generally, an error is returned with an HTTP Status Code of 400 or greater, for webAPI controllers, the MVC framework transforms an error into a ProblemDetails object. We can configure it manually using 2 ways,

1. Define a class and extend ProblemDetailsFactory, then add it as a transient service with <IServiceCollection obj>.AddTransient<ProblemDetailsFactory, myProblemFactory>();
2. Use ClientErrorMapping property,

<IServiceCollection obj>.AddControllers()

.ConfigureApiBehaviorOptions(options =>

{

…

options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =

"https://httpstatuses.com/404";

});

• SignalR: This open-source library adds real-time web functionality to our apps. Supports WebSockets, Server-Sent events and Long Polling for handling RTC (real time communication).

It uses hubs to communicate, hubs are like pipelines and we specify method names to invoke in string along with parameters and their names which are then serialized at sender and deserialized at receiver.

The way Real-time communication works is, the hub has method that forwards messages from 1 user to another, it also takes the method name that it will invoke in the client that receives the message. The client connects to the hub url and then sends a message with the intended user to the hub, the hub then forwards the message to the given user. The client has to specify the method name to be invoked in the hub as well.

To enable SignalR,

<IServiceCollection obj>.AddSignalR();

then,

<IApplicationBuilder obje>.UseEndpoints(endpoints =>

{

endpoints.MapHub<ChatHub>("/chathub");

});

where ChatHub is,

public class ChatHub: Hub

{

public Task SendMessage(string user, string message)

{

return Clients.All.SendAsync("ReceiveMessage", user, message);

}

public Task SendMessageToCaller(string user, string message)

{

return Clients.Caller.SendAsync("ReceiveMessage", user, message);

}

public Task SendMessageToGroup(string user, string message)

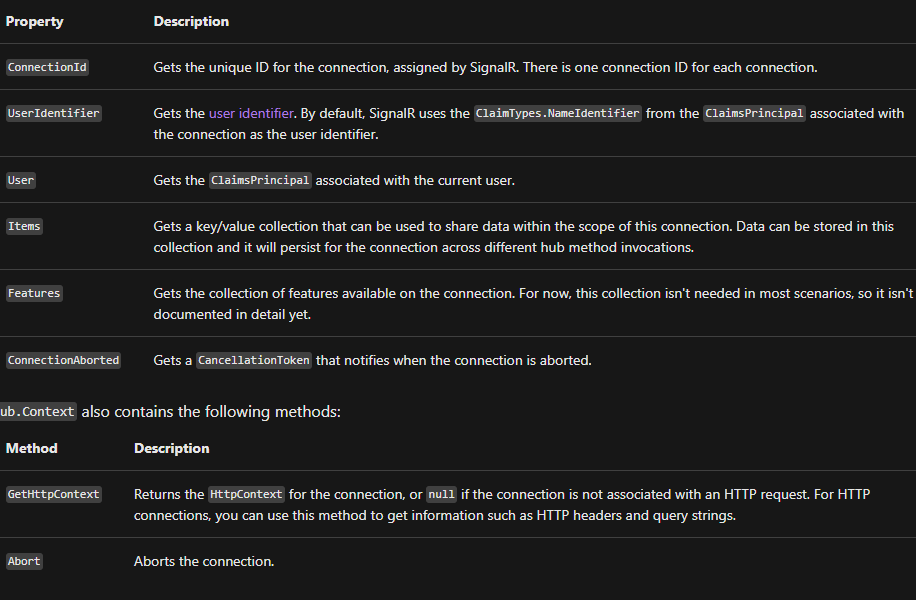
{

return Clients.Group("SignalR Users").SendAsync("ReceiveMessage", user, message);

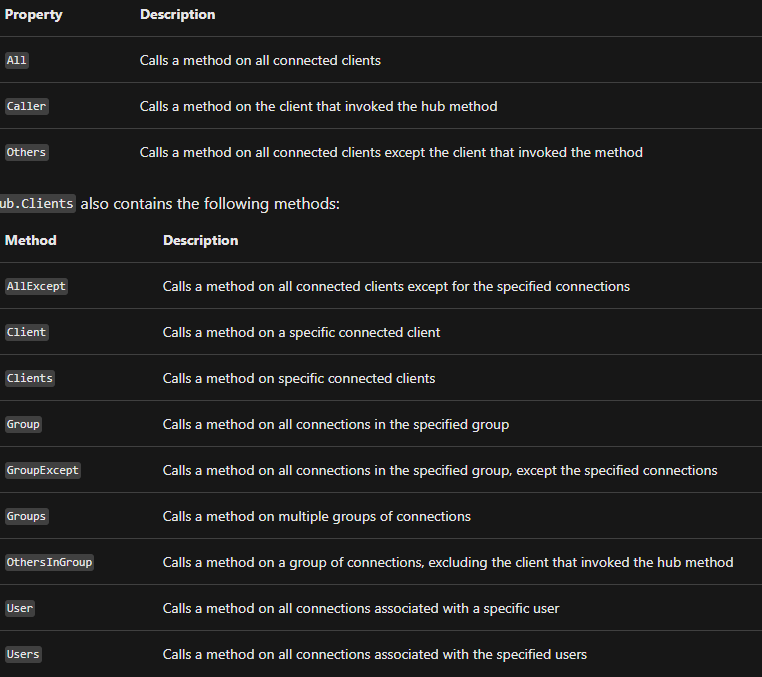
}

}

A Hub is a transient service, i.e., instances are reset at each request, so don’t store state in them.

* Hub class has a Context property, it has these following methods and properties

It also has a Clients object with these props and attrs,



* Client.Group(“<group name>”).SendAsync(“<method name in the client>”, username, message); sends the message to a group, similar methods exist and the syntax is almost the same for all of them.

When a Hub uses SendAsync, it provides a Reciever method name, the user/group/etc. specified get the message as response on the url the request came from and the receiver method in the client with the given name here gets the response.

* Strongly-typed hubs: Since Hubs as is rely on us specifying the receiving method’s name in string, it is error prone. We can use Hub<T> to resolve this,

public interface IChatClient

{

Task ReceiveMessage(string user, string message);

//this is an async method but interfaces don’t need to know that

}

public class StronglyTypedChatHub : Hub<IChatClient>

{

public async Task SendMessage(string user, string message)

{

await Clients.All.ReceiveMessage(user, message);

}

public Task SendMessageToCaller(string user, string message)

{

return Clients.Caller.ReceiveMessage(user, message);

}

}

* HubMethodName attribute can be used to change name of a hub method.

[HubMethodName(“<new name>”)]

<hub method>

* Hub Event Handler: OnDisconnected and OnConnected can be used to do stuff when a client (dis)connects from the hub.

To override,

…myHubClass…

{

…

public override async Task OnConnectedAsync()

{

await Groups.AddToGroupAsync(Context.ConnectionId, "SignalR Users");

await base.OnConnectedAsync();

}

public override async Task OnDisconnectedAsync(Exception exception)

{

await Groups.RemoveFromGroupAsync(Context.ConnectionId, "SignalR Users");

await base.OnDisconnectedAsync(exception);

}

}

In OnDisconnected, the Exception object will be null of connection.stop() was called by the client else it will have an exception.

* Custom Exception: By default SignalR sends some message to the client if there’s an exception caught. We can override the exception like so,

public Task ThrowException()

{

throw new HubException("This error will be sent to the client!");

}

* Use Hub object from outside the hub,

IHubContext<T,K> is a service in the DI that has the hub object, here T is the hub class. K is only required when the Hub<K> is used.

This service can be retrieved in the controllers or middleware (using app.Use(… context.RequestServices.GetRequiredService<IHubContext<T>>…)) or in IHost(get them same way as middleware)

Since we aren’t inside Hub class, we don’t have access to Caller, ConnectionId and the methods alike.

* Users: A single user can have multiple connections, all of them will get the message whenever we send them to the User. A user is identified with Context.UserIdentifier. This is the same object as ClaimTypes.NameIdentifier from the ClaimsPrincipal.

To send a message to a single user,

…

await Clients.User(<string user>).SendAsync(“<method>”,<string message>);

…

Multiple users can be ‘grouped’,

We group them with,

…anyMethod…

await Groups.AddToGroupAsync(Context.ConnectionId, <string group name>);

For remove it’s just RemoveFromGroupAsync with the same syntax.

To send a message to a group,

await Clients.Group(<string groupname>).SendAsync(“<RecieveMethod name>”,<string message>);

Group membership isn’t preserved when a connection reconnects, nor is it feasible to count the no. of connections.

* Hub Optimizations: Instead of raw parameters, get a custom object.

public class MyClass

{

public string user {get; set;}

public string message {get; set;}

public string param3 {get;set;}

}

For example,

public async myMethod(MyClass obj)

{

…do stuff with obj

}

invoking for .net is, Clients.All.SendAsync(“<RecieveMethod name>”, new {user= “abc”, message= “xxx”});

and invoking for js is, connection.invoke(“<RecieveMethod name>”,{user: “abc”, message: “xxx”});

param3 isn’t given and it won’t throw an exception, it will be null for myMethod’s obj parameter though.

* Hub Filters: Just like MVC filters but for hubs, we can define either global or per-hub filters, if both are used then global filters run before local filters. Filters are always invoked here in the order of addition.

Add filters:

<IServiceCollection obj>.AddSignalR(options=> {

//these are global filters.

options.AddFilter<myFilter>();

}).AddHubOptions<myHubClass>(options=> {

//these are local/per-hub filters

options.AddFilter<myFilter2>();

//or

options.AddFilter(typeof(myFilter3)); //type activated.aaa

//or

options.AddFilter(new myFilter4); //adds the filter as a singleton, all instances of //this hub use the same instance

//any of the methods can be used to add a hub filter.

});

To create a filter,

public class myFilter: IHubFilter

{

public async ValueTask<object> InvokeMethodAsync(

HubInvocationContext invocationContext, Func<HubInvocationContext, ValueTask<object>> next)

{

Console.WriteLine($"Calling hub method '{invocationContext.HubMethodName}'");

try

{

return await next(invocationContext);

}

catch (Exception ex)

{

Console.WriteLine($"Exception calling '{invocationContext.HubMethodName}': {ex}");

throw;

}

}

// Optional method

public Task OnConnectedAsync(HubLifetimeContext context, Func<HubLifetimeContext, Task> next)

{

return next(context);

}

// Optional method

public Task OnDisconnectedAsync(

HubLifetimeContext context, Exception exception, Func<HubLifetimeContext, Exception, Task> next)

{

return next(context, exception);

}

}

Filters run like middleware, where calling next calls the next filter in queue and not calling it shorts the filter pipeline.

Instead of directly returning a next , we can also take the value of next and then return it after processing. var obj= await next(…) and use the processed value from it. This does the following,

Order of processing: this func -> the next filter -> next filter and so on -> at last this func again, with the processed value now in obj and then we can use it and at last return obj to complete the filter chain.

This may be how filters in MVC work too (unsure).

Recommended way to use hub filters is by using attributes along with filters, to do so,

create a hub filter, add it in AddFilter,

create an attribute class, myXYZAttribute,

use the attribute on a hub method,

[myXYZ (…)] //’Attribute’ has to beremoved as .net removes that from the //classname

…myHubMethod…

then in the filter class,

…InvokeAsync…{

var myXYZ = (myXYZAttribute)Attribute.GetCustomAttribute(

invocationContext.HubMethod, typeof(myXYZAttribute));

… //use myXYZ object

}

* Client: SignalR clients are there in many frameworks, for .net we use Microsoft.AspNetCore.SignalR.Client package.

To use a .net SignalR client,

HubConnection connection= new HubConnectionBuilder().WithUrl(“<hub url like [http://localhost:53353/ChatHub>”).Build](http://localhost:53353/ChatHub%3e)();

await connection.StartAsync();

connection.Closed += async (error)=> {

…. //same as OnDisconnected method in a Hub, the error object is null if we //call connection.Close but not null if an exception has occured/connection //terminated

// This method can be non-async too, in which case we return Task.CompletedTask;

//we can use manual reconnection here too, in case it failed and error is non //null, by using await connection.StartAsync(); and returning void.

}

connection.On<string,string>(“<RecieveMethod name like RecieveMessage which the hub will use in SendAsync as well>”, (user,message) => {

…

//do stuff with user which has connectionId and message

});

await connection.InvokeAsync(“<method name in the hub like SendMessage >”, “<some user id>”, “<some message>”);

That’s it.

* Client Reconnect: <HubBuilder obj>.WithUrl(…).WithAutomaticReconnect().Build(); to configure enable reconnection on loss. By default it waits 0,2,10 and then 30 secs before trying reconnection.

While reconnecting, <HubConnection obj>.State is HubConnectionState.Reconnecting.

<HubConnection obj>.Reconnecting += error => {…}

can be used to handle reconnecting phases.

..Reconnected += connectionId=> {…}

can be used to handle successful reconnected phases, here connectionId is supplied.

Reconnection doesn’t cover initial starting phase, if there’s an error there then that needs to be handled differently.

…WithAutomaticReconnect(new [] {TimeSpan.FromSeconds(2), TimeSpan.FromSeconds(5)});

the anonymous array class tells the retry wait and retry count. For this one, after failing first time it will wait 2 seconds then attempt reconnecting, then 5 seconds and then it will not retry again, as opposed to 0 , 2 , 10 and 30 seconds it would by default.

…WithAutomaticReconnect(new RetryPolicy());

RetryPolicy is a class that implements IRetryPolicy.

public class RetryPolicy: IRetryPolicy

{

private readonly Random \_random = new Random();

public TimeSpan? NextRetryDelay(RetryContext retryContext)

{

// If we've been reconnecting for less than 60 seconds so far,

// wait between 0 and 10 seconds before the next reconnect attempt.

if (retryContext.ElapsedTime < TimeSpan.FromSeconds(60))

{

return TimeSpan.FromSeconds(\_random.NextDouble() \* 10);

}

else

{

// If we've been reconnecting for more than 60 seconds so far, stop reconnecting.

return null;

}

}

}

* Client Initial phase failures: We can use a method like so to handle initial failures,

public static async Task<bool> ConnectWithRetryAsync(HubConnection connection, CancellationToken token)

{

// Keep trying to until we can start or the token is canceled.

while (true)

{

try

{

await connection.StartAsync(token);

Debug.Assert(connection.State == HubConnectionState.Connected);

return true;

}

catch when (token.IsCancellationRequested)

{

return false;

}

catch

{

// Failed to connect, trying again in 5000 ms.

Debug.Assert(connection.State == HubConnectionState.Disconnected);

await Task.Delay(5000);

}

}

}

* For Hosting SignalR based apps, Redis Backplane or Azure App Service can be used. Reason for their need is that unlike traditional HTTP services, SignalR needs to keep the connections alive, and a single user can have multiple connections (diff devices) so the server can reach it’s limit quite fast. Another reason is that when a signalR app is scaled out horizontally it can’t keep the users in sync and that means each one would act like its own hub. To solve the latter we can use a backplane, a connector service, this service holds the data for SignalR and syncs stuff. To solve both the issues we can use AAS, as it provides a server which will handle the connections and also acts as a backplane.
* Serializers: SignalR supports 2 protocols for encoding messages, JSON and MessagePack.

To use JSON serialization,

<IServiceCollection obj>.AddSignalR(hubOptions=>

{

…

}

).AddJsonProtocol(options=> {

…//use options to configure the json serializer.

});

This same method is also available on the <HubConnectionBuilder obj> to configure it for the .net client.

MessagePack: It is a fast and compact binary serialization format, the demerit to it is that the messages are unreadable in network traces and logs as the bytes need to be parsed.

To use MessagePack in the hub, first get Microsoft.AspNetCore.SignalR.Protocols.MessagePack package,

then,

<IServiceCollection obj>. AddSignalR()

.AddMessagePackProtocol();

This enables MessagePack, JSON is enabled by default and with this MessagePack is enabled as well.

To configure MessagePack Serializer,

…AddMessagePackProtocol(options =>

{

options.SerializerOptions = MessagePackSerializerOptions.Standard

.WithResolver(new CustomResolver())

.WithSecurity(MessagePackSecurity.UntrustedData);

});

In a .net client we enable SignalR by, first getting the same package.

then we enable it with,

new HubConnectionBuilder()

.WithUrl("/chathub")

.AddMessagePackProtocol()

.Build();

here as well, the AddMessagePackProtocol takes a delegate with the same options.

Unlike in Hub, a client can only use 1 type of serializer, using messagepack replaces the default Json serializer.

MessagePack property names are case sensitive, DateTime must be in UTC before sending, and lastly messagepack doesn’t automatically convert type of a property into the receiving type, we must send values in the type they will be received.

* Configuration: …hubOptions has methods that can be used to configure SignalR.

HTTP configuration can be made by using,

…endpoints.MapHub<MyHub>(“<route>”, options=>

{

…

}

)

For .net client the configurations can be applied by using

..WithUrl(“<url>”, options=>{

…

}).Build();

* Auth: SignalR communication can also use authentication and authorization the same way other .net apps do.

First, enable auth by UseAuthentication and UseAuthorization in middleware pipeline.

Auth Methods:

Cookie Auth: This auth type is automatically used by SignalR when the client is a browser. For non-browser clients or to manually set cokie we use Cookies property of options object in …WithUrl of client.

Bearer Token Auth: The client can use an ‘access token’ to do auth as well, this token is sent at every HTTP request and the server uses it to validate and identify the user. This is done by using AccessTokenProvider property of options object in ..WithUrl of .net client, it needs an async function that generates an access token for each request. For browser clients , SignalR uses query params but for others it uses HTTP headers to set the token.

Bearer Authentication is preferred over cookie auth since it covers more type of clients.

On server, Bearer token auth is set up by using JWT Bearer middleware,

To set it up,

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

services.AddIdentity<ApplicationUser, IdentityRole>()

.AddEntityFrameworkStores<ApplicationDbContext>()

.AddDefaultTokenProviders();

services.AddAuthentication(options =>

{

// Identity made Cookie authentication the default.

// However, we want JWT Bearer Auth to be the default.

options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddJwtBearer(options =>

{

// Configure the Authority to the expected value for your authentication provider

// This ensures the token is appropriately validated

options.Authority = /\* TODO: Insert Authority URL here \*/;

// We have to hook the OnMessageReceived event in order to

// allow the JWT authentication handler to read the access

// token from the query string when a WebSocket or

// Server-Sent Events request comes in.

// Sending the access token in the query string is required due to

// a limitation in Browser APIs. We restrict it to only calls to the

// SignalR hub in this code.

// See https://docs.microsoft.com/aspnet/core/signalr/security#access-token-logging

// for more information about security considerations when using

// the query string to transmit the access token.

options.Events = new JwtBearerEvents

{

OnMessageReceived = context =>

{

var accessToken = context.Request.Query["access\_token"];

// If the request is for our hub...

var path = context.HttpContext.Request.Path;

if (!string.IsNullOrEmpty(accessToken) &&

(path.StartsWithSegments("/hubs/chat")))

{

// Read the token out of the query string

context.Token = accessToken;

}

return Task.CompletedTask;

}

};

});

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

services.AddSignalR();

// Change to use Name as the user identifier for SignalR

// WARNING: This requires that the source of your JWT token

// ensures that the Name claim is unique!

// If the Name claim isn't unique, users could receive messages

// intended for a different user!

services.AddSingleton<IUserIdProvider, NameUserIdProvider>();

// Change to use email as the user identifier for SignalR

// services.AddSingleton<IUserIdProvider, EmailBasedUserIdProvider>();

// WARNING: use \*either\* the NameUserIdProvider \*or\* the

// EmailBasedUserIdProvider, but do not use both.

}

When using Identity Server, use

using Microsoft.AspNetCore.Authentication.JwtBearer;

using Microsoft.Extensions.Options;

public class ConfigureJwtBearerOptions : IPostConfigureOptions<JwtBearerOptions>

{

public void PostConfigure(string name, JwtBearerOptions options)

{

var originalOnMessageReceived = options.Events.OnMessageReceived;

options.Events.OnMessageReceived = async context =>

{

await originalOnMessageReceived(context);

if (string.IsNullOrEmpty(context.Token))

{

var accessToken = context.Request.Query["access\_token"];

var path = context.HttpContext.Request.Path;

if (!string.IsNullOrEmpty(accessToken) &&

path.StartsWithSegments("/hubs"))

{

context.Token = accessToken;

}

}

};

}

}

then,

services.AddAuthentication()

.AddIdentityServerJwt();

services.TryAddEnumerable(

ServiceDescriptor.Singleton<IPostConfigureOptions<JwtBearerOptions>,

ConfigureJwtBearerOptions>());

The UserIdProvider is defined as follows,

public class EmailBasedUserIdProvider : IUserIdProvider

{

public virtual string GetUserId(HubConnectionContext connection)

{

return connection.User?.FindFirst(ClaimTypes.Email)?.Value;

}

}

And to generate claim type for identity database we define

// create a new user

var user = new ApplicationUser { UserName = Input.Email, Email = Input.Email };

var result = await \_userManager.CreateAsync(user, Input.Password);

// add the email claim and value for this user

await \_userManager.AddClaimAsync(user, new Claim(ClaimTypes.Email, Input.Email));

finally,

services.AddSingleton<IUserIdProvider, EmailBasedUserIdProvider>();

The UserIdProvider must provide unique values for each user else message intended for 1 user can go to other.

To finally use Auth on methods (so that only authenticated users can access them),

[Authorize]

…myMethod

//or

[Authorize(“<authorization policy name>”)]

…myMethod2

To create the policy,

[Authorize]

public class ChatHub : Hub

{

public void SendMessage(string message)

{

}

[Authorize("DomainRestricted")]

public void BanUser(string username)

{

}

[Authorize("DomainRestricted")]

public void ViewUserHistory(string username)

{

}

}

public class DomainRestrictedRequirement :

AuthorizationHandler<DomainRestrictedRequirement, HubInvocationContext>,

IAuthorizationRequirement

{

protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,

DomainRestrictedRequirement requirement,

HubInvocationContext resource)

{

if (IsUserAllowedToDoThis(resource.HubMethodName, context.User.Identity.Name) &&

context.User.Identity.Name.EndsWith("@microsoft.com"))

{

context.Succeed(requirement);

}

return Task.CompletedTask;

}

private bool IsUserAllowedToDoThis(string hubMethodName,

string currentUsername)

{

return !(currentUsername.Equals("asdf42@microsoft.com") &&

hubMethodName.Equals("banUser", StringComparison.OrdinalIgnoreCase));

}

}

We also have to add the policy to the AddAuthorization,

services

.AddAuthorization(options =>

{

options.AddPolicy("DomainRestricted", policy =>

{

policy.Requirements.Add(new DomainRestrictedRequirement());

});

});

* CORS with SignalR: We should use Cors, to do so the following is required by SignalR,

<IApplicationBuilder obj>.UseCors(builder =>

{

builder.WithOrigins("https://example.com")

.AllowAnyHeader()

.WithMethods("GET", "POST")

.AllowCredentials();

});

* Streaming: SignalR supports streaming, i.e., sending/receiving fragments of data over time rather than instantly.

To set up Server-To-Client streaming, simply return IAsyncEnumerable<T>, ChannelIReader<T>, Task<IAsyncEnumerable<T>> or Task<ChannelReader<T>>

For example,

public async IAsyncEnumerable<int> Counter(

int count,

int delay,

[EnumeratorCancellation]

CancellationToken cancellationToken)

{

for (var i = 0; i < count; i++)

{

// Check the cancellation token regularly so that the server will stop

// producing items if the client disconnects.

cancellationToken.ThrowIfCancellationRequested();

yield return i;

// Use the cancellationToken in other APIs that accept cancellation

// tokens so the cancellation can flow down to them.

await Task.Delay(delay, cancellationToken);

}

}

Here, whenever an object is written to the ChannelWriter, it is sent to the client. At the end it is completed and tells the client that the stream is closed.

To use a ChannelReader and a ChannelWriter,

public ChannelReader<int> Counter(

int count,

int delay,

CancellationToken cancellationToken)

{

var channel = Channel.CreateUnbounded<int>();

// We don't want to await WriteItemsAsync, otherwise we'd end up waiting

// for all the items to be written before returning the channel back to

// the client.

\_ = WriteItemsAsync(channel.Writer, count, delay, cancellationToken);

return channel.Reader;

}

private async Task WriteItemsAsync(

ChannelWriter<int> writer,

int count,

int delay,

CancellationToken cancellationToken)

{

Exception localException = null;

try

{

for (var i = 0; i < count; i++)

{

await writer.WriteAsync(i, cancellationToken);

// Use the cancellationToken in other APIs that accept cancellation

// tokens so the cancellation can flow down to them.

await Task.Delay(delay, cancellationToken);

}

}

catch (Exception ex)

{

localException = ex;

}

finally

{

writer.Complete(localException);

}

}

A hub acts like a client when,

public async Task UploadStream(ChannelReader<string> stream)

{

while (await stream.WaitToReadAsync())

{

while (stream.TryRead(out var item))

{

// do something with the stream item

Console.WriteLine(item);

}

}

}

a parameter is of type ChannelReader<T> or IAsyncEnumerable<T>

For a .net client to a .net SignalR server,

Client as server: hubConnection using StreamAsync or StreamAsChannelAsync start server-to-client streaming. They also take <T> which specifies the T of IAsyncEnumerable<T> or ChannelReader<T>.

For example:

var cancellationTokenSource = new CancellationTokenSource();

var stream = hubConnection.StreamAsync<int>(

"Counter", 10, 500, cancellationTokenSource.Token);

await foreach (var count in stream)

{

Console.WriteLine($"{count}");

}

Console.WriteLine("Streaming completed");

or

// Call "Cancel" on this CancellationTokenSource to send a cancellation message to

// the server, which will trigger the corresponding token in the hub method.

var cancellationTokenSource = new CancellationTokenSource();

var channel = await hubConnection.StreamAsChannelAsync<int>(

"Counter", 10, 500, cancellationTokenSource.Token);

// Wait asynchronously for data to become available

while (await channel.WaitToReadAsync())

{

// Read all currently available data synchronously, before waiting for more data

while (channel.TryRead(out var count))

{

Console.WriteLine($"{count}");

}

}

Console.WriteLine("Streaming completed");

Here “Counter” is the method name for the hub.

Client as Client: When SendAsync, InvokeAsync or StreamAsChannelAsync gets an IAsyncEnumerable<T> or ChannelReader<T> argument.

async IAsyncEnumerable<string> clientStreamData()

{

for (var i = 0; i < 5; i++)

{

var data = await FetchSomeData();

yield return data;

}

//After the for loop has completed and the local function exits the stream completion will be sent.

}

await connection.SendAsync("UploadStream", clientStreamData());

or

var channel = Channel.CreateBounded<string>(10);

await connection.SendAsync("UploadStream", channel.Reader);

await channel.Writer.WriteAsync("some data");

await channel.Writer.WriteAsync("some more data");

channel.Writer.Complete();

* WebSockets with SignalR: WebSockets enable fast 2-way persistent communication over TCP connections.

To enable it,

<IApplicationBuilder obj>.UseWebSockets();

//Place it just before UseEndpoints.

Can also take an argument,

var webSocketOptions= new WebSocketOptions(){

KeepAliveInterval=<Timespan>;

}

webSocketOptions.AllowedOrigins.Add(“<url>”);

…UseWebSockets(webSocketOptions);

By default all origins are accepted but defining any makes it only allow websocket connection from that origin.

To accept a WebSocket request,

public class WebSocketController : ControllerBase

{

[HttpGet("/ws")]

public async Task Get()

{

if (HttpContext.WebSockets.IsWebSocketRequest)

{

using WebSocket webSocket = await

HttpContext.WebSockets.AcceptWebSocketAsync();

await Echo(HttpContext, webSocket);

}

else

{

HttpContext.Response.StatusCode = (int)HttpStatusCode.BadRequest;

}

}

…

}

we can do so in the middleware as well

app.Use(async (context, next) =>

{

if (context.Request.Path == "/ws")

{

if (context.WebSockets.IsWebSocketRequest)

{

using (WebSocket webSocket = await context.WebSockets.AcceptWebSocketAsync())

{

await Echo(context, webSocket);

}

}

else

{

context.Response.StatusCode = (int) HttpStatusCode.BadRequest;

}

}

else

{

await next();

}

});

where Echo is

private async Task Echo(HttpContext context, WebSocket webSocket)

{

var buffer = new byte[1024 \* 4];

WebSocketReceiveResult result = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer), CancellationToken.None);

while (!result.CloseStatus.HasValue)

{

await webSocket.SendAsync(new ArraySegment<byte>(buffer, 0, result.Count), result.MessageType, result.EndOfMessage, CancellationToken.None);

result = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer), CancellationToken.None);

}

await webSocket.CloseAsync(result.CloseStatus.Value, result.CloseStatusDescription, CancellationToken.None);

}

Echo method here displays how we can Receive and Send messages using WebSocket.

Client sends a Disconnect message when it disconnects, which will not arrive if the connection is dropped. To handle this, use timers.

For other Client WebSocket setups, look at any package for them on the client.

3.1. Performance:

* + 1. Use Caching. Cache large objects that are frequently used. Pool buffers by using ArrayPool<T> to store large arrays.
    2. Understand Hot-Code paths: Code path that is accessed frequently. They limit app-scale out and performance. Don’t allocate many ephemeral objects on hot code paths.
    3. Avoid sync working (but don’t use Task.Run to make sync tasks async either), use async as much as possible. Don’t use Task.Wait or Task.Result since they are blocking in nature. Don’t acquire locks in common code paths, and don’t use Task.run and await it right away. Make entire Controller/API architecture async.

public class GoodStreamReaderController : Controller

{

[HttpGet("/contoso")]

public async Task<ActionResult<ContosoData>> Get()

{

var json = await new StreamReader(Request.Body).ReadToEndAsync();

return JsonSerializer.Deserialize<ContosoData>(json);

//or

// return await JsonSerializer.DeserializeAsync<ContosoData>(Request.Body);

}

}

Use methods like these to read a request data instead of normal ReadToEnd();

Prefer ReadFormAsync (HttpContext.Request.ReadFormAsync) over Request.Form

* + 1. Avoid reading large request/response bodies onto the memory. >85KB objects end up in Large Object Heap and that slows the app down.
    2. Use pagination if a huge amount of data is to be sent to the client, i.e., send only part of the data at once and design pages with Indexes/Page Numbers or the like for the same. Don’t retrieve any more data than necessary in an action method..
    3. For async enumerable lists such as a database entry list, use IAsyncEnumerable<T> return type with Controllers. Or IEnumerable<T> obj.ToListAsync();
    4. Minimize network round trips, try to return all data in a single call (pagination is a different thing than this).
    5. Use no-tracking queries when requesting entries in EF Core, these queries are more efficient than their counterparts.
    6. Do filter and aggregate LINQ queries (like .where or .Select or .Sum methods) so that filtering is performed by the database.

There are other EF Core optimizations like DbContext Pooling and Explicitly compiled queries.

* + 1. Pool HTTP connections with HttpClientFactory. We shouldn’t create and dispose HttpClient instances directly as they leave sockets open for a while after being disposed, HttpClientFactory reuses them instead of opening new sockets. Or create an HttpClient object and try to use the same for multiple requests.
    2. Middleware components shouldn’t have long-running tasks in them. For that matter, no long-running task should be awaited by an HTTP request, use background services or Azure Functions to run these tasks and respond to the client in a separate response or use real-time communication.

Middleware components/Action methods etc. shouldn’t write to HttpContext after the response body is sent or has started, to check if the response has been sent or is in sending, HttpContext.Response.HasStarted can be used which returns a bool for the same. Alternatively, we can use the OnStarting event handler,

context.Response.OnStarting(() =>

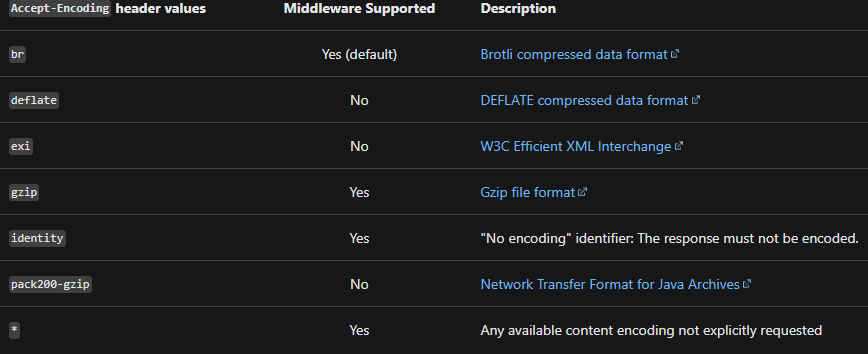
{

context.Response.Headers["someheader"] = "somevalue";

return Task.CompletedTask;

});

* + 1. Use bundling (combine multiple files into one) and minifying (reduce the size of files) and consider tools like Webpack for even better asset management.
    2. Use response compression. If we compress responses then that saves network bandwidth by a lot.
       1. When a client supports response compression it will give a value to Accept-Encoding header.



* + - 1. The server in turn uses, Content-Encoding header to tell which encoding was used, Content-Type to specify the MIME type and Vary header to specify what content should be cached. Content-Length and Content-MD5 headers are removed.
      2. To enable response caching, we use Microsoft.AspNetCore.ResponseCompression package.

then,

<IServiceCollection obj>.AddResponseCompression();

then,

<IApplicationBuilder obj>.UseResponseCompression();

Should be placed before any middleware that returns a response.

* + - 1. By default, Brotli (br) and Gzip (gzip) compression providers are already enabled by ResponseCompression middleware if any other provider (custom or inbuilt) isn’t explicitly added, in that case default compression providers must be added explicitly as well.
      2. To configure Brotli or Gzip compression providers (added explicitly or by default),

<IServiceCollection obj>.AddResponseCompression();

<IServiceCollection obj>.Configure<BrotliCompressionProviderOptions>(options=>{

…

}); //We can specify CompressionLevel

Similarly we can configure Gzip.

* + - 1. To add Custom Provider, we extend ICompressionProvider class

then,

public class myCompressionProvider:ICompressionProvider

{

public string EncodingName=”mycompression”; //required prop

public bool SupportsFlush=> true;

public Stream CreateStream(Stream outputStream)

{

//create custom compression stream wrapper

return outputStream;

}

}

then, to add it,

<IServiceCollection obj>.AddResponseCompression(options=>{

options.Providers.Add<BrotliCompressionProvider>();

options.Providers.Add<GzipCompressionProvider>();

options.Providers.Add<myCompressionProvider>();

options.MimeTypes= ResponseCompressionDefaults.MimeTypes.Concat(

new[] { "image/svg+xml" });

//options.EnableForHttps=true;

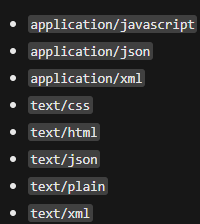
});

Now if a request has Accept-Encoding: mycompression then myCompressionProvider will be used.

EnableForHttps is false by default but if enable allows compression of files sent over Https.

The mimetype given above tell what the actual type of content is being returned in responses, .net core compresses default mime types. We can replace/append multiple mime types.

These are the supported ones



The image/svg+xml is also an inbuilt one for images.

* + 1. Try to minimize exceptions as they are expensive and don’t use them as means of nomal program flow.
    2. Don’t store IHttpContextAccessor.HttpContext in a field, instead store the IHttpContextAccessor object (to make it available in the DI look at HttpContext in the doc below). This allows us to view the HttpContext with each request.
    3. Use async Task instead of async void, as async void causes the method to end after the first await is reached.
    4. Copy HttpContext data if it has to reach multiple threads (parallel or otherwise) or be read outside the scope of an active action method (such as in Task.Run). The latter part also applies to capturing services from DI in an action method, all captured values are disposed when the method ends so using [FromServices] and passing those objects to Task.Run will cause exception if Task.Run doesn’t finish before the method is complete.

If [FromServices] is to be used and then the object passed to Task.Run then do it like this,

public IActionResult FireAndForget3([FromServices]IServiceScopeFactory

serviceScopeFactory)

{

\_ = Task.Run(async () =>

{

await Task.Delay(1000);

using (var scope = serviceScopeFactory.CreateScope())

{

var context = scope.ServiceProvider.GetRequiredService<ContosoDbContext>();

context.Contoso.Add(new Contoso());

await context.SaveChangesAsync();

}

});

return Accepted();

}

* + 1. Garbage Collector: The .net GC allocates heap segments (continuous ranges of memory) for each object, then it categorizes them into 1 of the 3 generations. Gen 0, Gen 1 and Gen 2. The lower the Gen the more frequently the object is garbage collected. Objects change gens based on their lifetime, a shorter lived object like an Http Request is Gen 0 as it is GC’d very frequently and a singleton is a Gen 2 object.
       1. GCing is an expensive task because it has to allocate and de-allocate objects.
       2. De-allocating large objects frequently would draw a lot more cpu.
       3. Large Objects (>=83KB) are stored in Large Object Heap, which is a special zone. Other objects are defragmented at each GC cycle (Compaction) but since moving large objects are time consuming they are in this special zone. Large Objects are categorized as Gen 2. When LOH is full, Gen 2 GC is triggered which is an even more expensive task.
       4. There are general limitations for memory usage per object and their lifetime, given above.
       5. There are 2 types of GC modes,

WorkStation GC: Faster GCing, uses more cpu and less memory as it GCs objects more faster.

Server GC: Default, uses more memory but lesser cpu as objects are GCed less frequently.

To change the mode,

<PropertyGroup>

<ServerGarbageCollection>true</ServerGarbageCollection>

</PropertyGroup>

in project file/runtimeconfig.json. True means server GC and false means former.

* + - 1. It is preferred to run workstation GC when running multiple apps on same machine using docker.
      2. GC can’t run on objects that are referenced such as an array of strings that stores a string at every request in a controller’s property. This object will quickly size out and this process is called a memory leak.
      3. Object Pooling: It is simply pooling multiple objects multiple times and storing them on an already allocated memory. To do so,

private static ArrayPool<int> apl= ArrayPool<int>.Create();

public class myPoolObj: IDisposable

{

public int[] arr{get; private set;}

public myPoolObj(int size){

arr=apl.Rent(size);

}

public void Dispose()

{

apl.Return(arr);

}

}

and use this myPoolObj everywhere along with calling dispose or using using.

In .net core, action methods can use,

…myActionMethod(int size)

{

var myPool=new myPoolObj(size);

//dostuff with myPool.arr;

//then

HttpContext.Response.RegisterForDispose(myPool);

return myPool.arr;

}

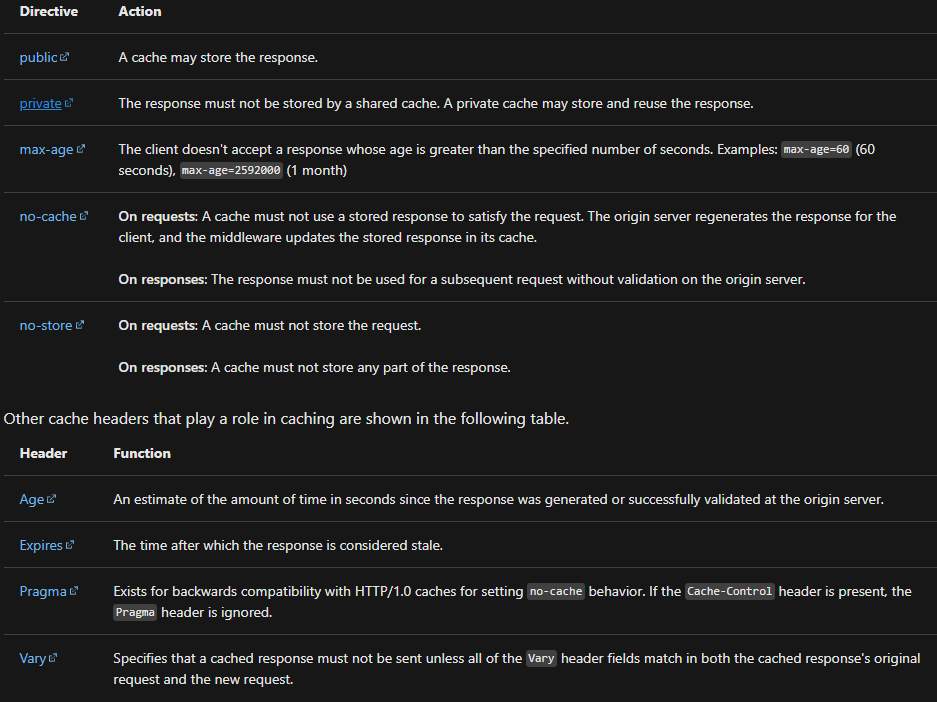
RegisterForDispose(<obj>) takes an obj that extends IDisposable and disposes it once the response is sent.

If large objects need to be stored frequently, and cannot be broken down into smaller units then use ArrayPool with Object Pooling.

* + 1. Response caching: It is used to reduce the number of requests to a web server and reduce direct processing needed.

3.1.19.1 The HTTP header ‘Cache-Control’ defines cache directives, these directives control the caching behavior.

Common directives are:



.Net respects these cache directives automatically when Response Caching Middleware is used.

3.1.19.2 We shouldn’t cache content that changes based on a users identity. Caching works best when the object changes infrequently and is expensive to generate. Apps should be written and tested to never depend on cached data.

3.1.19.3 Types of caching in .Net Core:

3.1.19.3.1 In-Memory Caching: Use server memory to serve cache data, useful for sticky sessions (Requests from a client are always routed to same server). We can use Cache Tag Helper to enable this.

3.1.19.3.1.1 To use it, we need Microsoft.Extensions.Caching.Memory, then

simply create an IMemoryCache field and get the same from DI in a constructor of a controller.

Then,

var obj2= <IMemoryCache obj>.GetOrCreate(“somekey”, entry=>{

entry.SetSlidingExpiration(TimeSpan.FromSeconds(3));

entry.AbsoluteExpirationRelativeToNow=TimeSpan.FromSeconds(30);

return someRequestedObj;

});

Looks up the somekey in the cache and if it isn’t found then stores it. The SlidingTime and AbsoluteTime, whichever hits 0 first will expire the entry in the cache.   
They work this way, if a cache request comes for the key then the sliding timer is set to 0 and this could go on for forever meaning the cached data can become stale and new data won’t be generated till requests keep coming. The absoluteTime will expire the entry no matter slidingtime remaining, it is measured from the time of cache creation.

3.1.19.3.1.2 MemoryCacheOptions: We can create a singleton MemoryCache object and impose SizeLimit on cache.

To do so,

using Microsoft.Extensions.Caching.Memory;

public class MyMemoryCache

{

public MemoryCache Cache { get; private set; }

public MyMemoryCache()

{

Cache = new MemoryCache(new MemoryCacheOptions

{

SizeLimit = 1024

});

}

}

Then add the class as singleton in <IServiceCollection obj>.AddSingleton<MyMemoryCache>();

By default, cache size has no limit and objects of any size and any count can be stored in it even after the system runs out of memory. This class helps prevent that, we set a SizeLimit with an arbitrary int value. The value means nothing to the package but when we are storing items using this class we have to define a ‘size’ of the item, which won’t be inserted if the limit has been reached here.

To store values in it

….controller that gets MyMemoryCache obj from DI

…action method(){

if(!cache.TryGetValue(“MyKey”, out string cacheEntry))

{

cacheEntry=”xyz”; // new value for cache Entry since it is not in cache.

var cacheEntryOptions= new MemoryCacheEntryOptions().SetSize(1).SetSlidingExpiration(…);

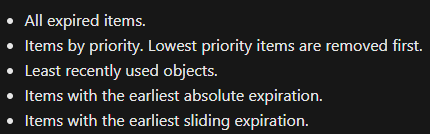
cache.Set(“MyKey”, cacheEntry, cacheEntryOptions);

}

}

Then, we have to call Compact or Remove every once a while, entries may expire but they aren’t automatically removed.

cache.Remove(“MyKey”); //to remove 1 item

cache.Compact(.50); //removes 50% of entries. This is the ordering.

3.1.19.3.2 Distributed Cache: Stores cache data in the memory of another server or redis backplane and the cache is shared with multiple servers. We can use Distributed Cache Tag Helper for this.

To add this service we use <IServiceCollection obj>.AddDistributedMemoryCache(options=>{…});

This type of cache is recommended since it stores data on an external server which provides resilience and coherence. We use IDistributedCache object for this type of cache. The actual implementation is made on the separate server itself/

3.1.19.3.3 ResponseCache Attribute: Specifies necessary parameters for setting appropriate headers in response caching.

3.1.19.3.4 VaryByHeader: Using [ResponseCache(VaryByHeader=”User-Agent”, Duration=30)] on an action method applies ‘Vary’ and Cache-Control header on response. The given attribute example presents this value in response headers

Cache-Control: public,max-age=30

Vary: User-Agent

3.1.19.3.5 NoStore: Using [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)] on an action method overrides most other properties for cache control sets the following,

Cache-Control: no-store,no-cache

Pragma: no-cache

If Location is None and NoStore is false then Cache-Control and Pragma are set to no-cache.

NoStore is typically set to true for error pages.

3.1.19.3.6: Location and Duration: To enable caching in the first place, Duration must be +ve and Location must be be Any(default, sets Cache-Control to public) or Client(private). Location being None sets Cache-Control and Pragma to no-cache.

3.1.19.3.6 Cache Profile: Allows us to specify a profile for ResponseCache to reduce duplication, if we use ResponseCache attribute on an action method then that overrides this global value.

To do so,

<IServiceCollection obj>.AddMvc(o=> {

o.CacheProfiles.Add(“somename”, new CacheProfile(){

Duration=30

});

});

and to use it,

[ResponseCache(CacheProfileName=”somename”)]

<action method>

3.1.19.4 Response Caching Middleware: We can configure response caching using this middleware.

3.1.19.4.1 To add it,

<IServiceCollection obj>.AddResponseCaching();

then,

<IApplicationBuilder obj>.UseResponseCaching(); //after UseCors and just //before any endpoint or middleware component modifying response.

then,

<IApplicationBuilder obj>.Use(async (context, next) =>

{

context.Response.GetTypedHeaders().CacheControl =

new Microsoft.Net.Http.Headers.CacheControlHeaderValue()

{

Public = true,

MaxAge = TimeSpan.FromSeconds(10)

};

context.Response.Headers[Microsoft.Net.Http.Headers.HeaderNames.Vary] =

new string[] { "Accept-Encoding" };

await next();

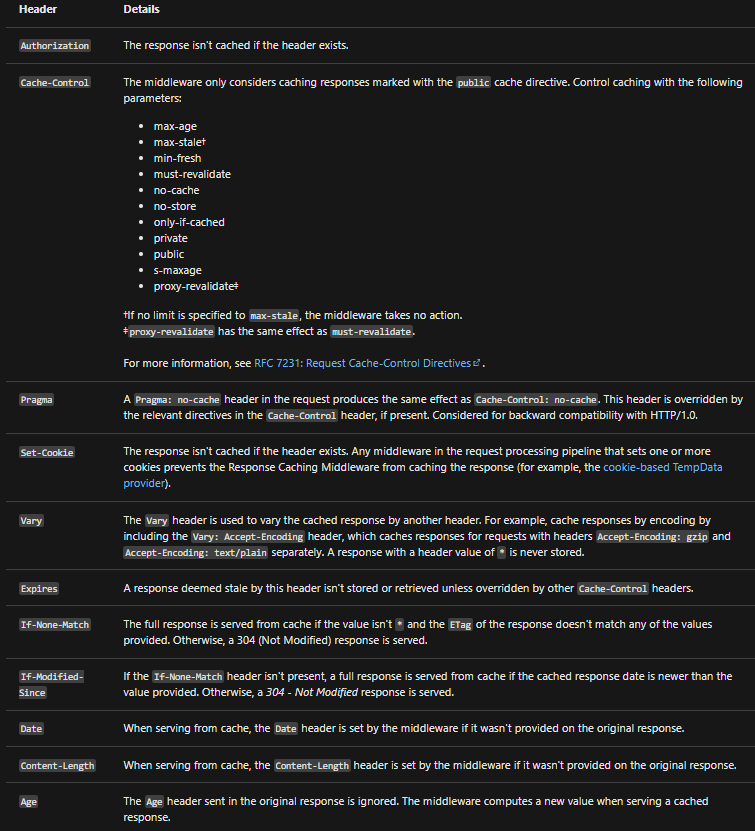
});

to configure response caching. Applies the config to all responses which are being returned with HTTP200.

If an action method/controller uses a [ResponseCache] attribute then this isn’t applied to that one.

3.1.19.4.2 ...AddResponseCaching(options=>{…}) allows us to set some other properties like MaximumBodySize.

3.1.19.4.3 HTTP headers used by Response Caching Middleware:



3.1.19.5 Caching only works when,

Response has 200 status code

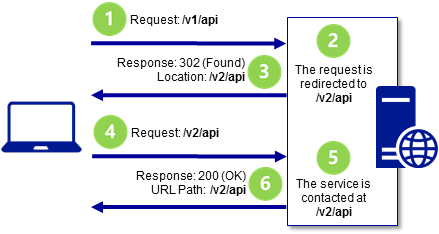
Request method is GET or HEAD

CSRF isn’t being used

…

* 1. Request and Response Operations: We can read request data and write to response directly using HttpRequest.BodyReader and HttpResponse.BodyWriter. These are Pipe objects, their stream counterparts are not recommended.
  2. URL Rewriting Middleware: It is the act of modifying request URL. It should be seldom used since it can reduce performance of an app.
     1. This is different from a URL Redirect, that is a client-side operation. A

URL redirect does this



Status Code 301 means the resource is Moved permanently and client can cache the url allowing it to skip step 1-3.

Status Code 302 means resource is Found and the redirection may be temporary hence it shouldn’t cache the url.

* + 1. URL Rewriting just means the request comes at a url and server internally uses another url mapped to the request url, the address bar on client remains the same. It is a server side operation.
    2. To use URL Redirection and URL Rewriting,

var options = new RewriteOptions()

.AddRedirect("redirect-rule/(.\*)", "redirected/$1")

.AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",

skipRemainingRules: true)

.AddApacheModRewrite(apacheModRewriteStreamReader)

.AddIISUrlRewrite(iisUrlRewriteStreamReader)

.Add(MethodRules.RedirectXmlFileRequests)

.Add(MethodRules.RewriteTextFileRequests)

.Add(new RedirectImageRequests(".png", "/png-images"))

.Add(new RedirectImageRequests(".jpg", "/jpg-images"));

<IApplicationBuilder obj>.UseRewriter(options);

…AddRedirect takes a 3rd arg for status code to be returned, default is 302.

…AddRedirectToHttps(<status code>,<port>); If both args are null then port is 443 and status code is 302. Redirects non https requests to https.

…AddRedirectToHttpsPermanent(); does the same but status code is 301 by default.

The ApacheModRewrite and IISUrlRewrite apply to Apache and IIS.

MethodRules.RedirectXmlFileRequests is an extension method,

public static void RedirectXmlFileRequests(RewriteContext context)

{

var request = context.HttpContext.Request;

// Because the client is redirecting back to the same app, stop

// processing if the request has already been redirected.

if (request.Path.StartsWithSegments(new PathString("/xmlfiles")))

{

return;

}

if (request.Path.Value.EndsWith(".xml", StringComparison.OrdinalIgnoreCase))

{

var response = context.HttpContext.Response;

response.StatusCode = (int) HttpStatusCode.MovedPermanently;

context.Result = RuleResult.EndResponse;

response.Headers[HeaderNames.Location] =

"/xmlfiles" + request.Path + request.QueryString;

}

}

Redirects requests for /xyz.xml to /xmlfiles/xyz.xml.

Similarly,

public static void RewriteTextFileRequests(RewriteContext context)

{

var request = context.HttpContext.Request;

if (request.Path.Value.EndsWith(".txt", StringComparison.OrdinalIgnoreCase))

{

context.Result = RuleResult.SkipRemainingRules;

request.Path = "/file.txt";

}

}

The context.Result takes a RuleResult, if it is ContinueRules (default) then it does so but if it’s EndResponse then stops applying rules and sends response back. It can also be SkipRemainingRules to stop applying rules but go to next middleware.

The RedirectImageRequests is a class that extends IRule, and has ApplyRule method which gets a RewriteContext object just like the extension methods for MethodRule.

* + 1. File Provider: We can access FileSystem through File Providers. We use IFileProvider to do so.
       1. Some uses are,
          1. IWebHostEnvironment which exposes Content root and web root as IFileProvider types.
          2. Static File Middleware uses File Providers to locate static files.
          3. Razor uses File Providers to locate pages and views.
       2. Classes that implement IFileProvider
          1. Composite File Provider: It is used to combine multiple IFileProvider instances to expose a single object to work with all of them.
          2. Manifest Embedded File Provider: Has to be installed with Microsoft.Extensions.FileProviders.Embedded package. It provides access to files embedded inside other assemblies.
          3. Physical File Provider: Used to access the physical file system, we specify a directory. To use it,

…get IFileProvider obj from DI using constructor injection then,

var file=new PhysicalFileProvider(“someLocation”);

We can get it in **ConfigureServices** as well by using env.ContentRootFileProvider.

* + - 1. IFileProvider.Watch: This method allows us to watch files/dirs for changes. Returns an IChangeToken,

For example:

var fileProvider=new PhysicalFileProvider(Directory.GetCurrentDirectory());

IChangeToken token= fileProvider.Watch(“xyz.txt”);

var tcs= new TaskCompletionSource<int>();

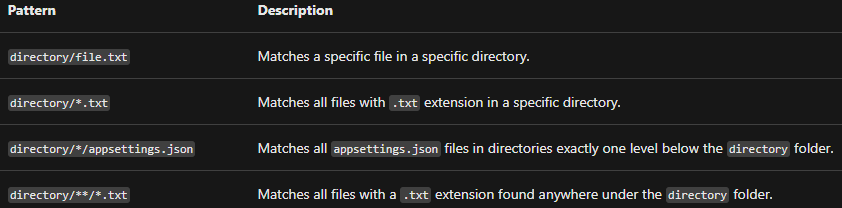
token.RegisterChangeCallback(state=> ((TaskCompletionSource<int>)state).TrySetResult(1),tcs);b

await tcs.Task.ConfigureAwait(false); //basically lets the awaiter of //this Task method not wait for it. It’s not useful in .net core.

DOTNET\_USE\_POLLING\_FILE\_WATCHER env var must be 1 or true if using the app in Docker. This polls the file watcher every 4 secs.

* + - 1. Glob Pattern: File system paths can use wildcard patterns, these are called glob (or globbing) patterns. It uses \* and \*\* to achieve the same.

Examples:



* + 1. HttpContext: This object has all the request and response data. We use the IHttpContextAccessor to work with it. The components inside it are defined as Feature Interfaces, we can even add/remove these features as needed. We can view these features in the Microsoft.AspNetCore.Http.Features object.
       1. To access the HttpContext in PageModel/Controller, directly call HttpContext object. To store it in a field, get and store the IHttpContextAccessor object.

To do so,

<IServiceCollection obj>.AddHttpContextAccessor();

Now IHttpContextAccessor is available in the DI.

* + - 1. To access HttpContext in Middleware, use HttpContext object as parameter of the Invoke or InvokeAsync method.
    1. IChangeToken is used to track changes in resources. It is used with file watchers, monitor configuration changes and the like.
    2. OWIN is supported by .NET . It allows multiple servers of different framework run together. It is not recommended for .net core only apps as it brings a performance penalty.
    3. Background Task: These are implemented as Hosted Services in .net core. IHostedService is used for the same. There are 3 ways to run a background service, on a timer, as a scoped service to access DI and queue these services sequentially.
       1. To add a hosted service,

<IServiceCollection obj>.AddHostedService<MyHostedService>();

Normally we would add this in Startup.cs and that means this service starts with the other services. To add this service after the app is configured,

…

Host.CreateDefaultBuilder(args).ConfigureWebHostDefaults(…).ConfigureServices(…//add it here).

or, to add it before the app is configured, call ConfigureServices(…) before ConfigureWebHostDefaults(…).

* + - 1. To create a Background Task,

We can extend BackgroundService class which internally implements IHostedService.

It has 3 methods, StartAsync, StopAsync and ExecuteAsync. All have a CancellationToken parameter.

StartAsync: Perform short-running async task here, the thread waits until this method is finished before starting up the service.

ExecuteAsync: Perform long-running async task here, the thread doesn’t wait unil this method is finished unless StopAsync is called which happens when app is asked to Shutdown, when that happens 5 seconds of grace period exists and after that app forces it to shut down.

When StartAsync is defined in the class extending BackgroundService then it must call executeAsync otherwise it will never be called. Either don’t override StartAsync or explicitly call ExecuteAsync in StartAsync in a class extending BackgroundService.

StopAsync: When this method is called the thread waits until this method is complete, it only waits for 5 seconds by default. This method calls ExecuteAsync and asks it to cancel tasks.

public class MyBackgroundService: BackgroundService

{

//constructor that can access even scoped services.

protected override async Task ExecuteAsync(..token)

{

..await and do stuff, pass the token and use while(!token.IsCancellationRequested){…} to keep running a task

}

public override async Task StopAsync(..token)

{

await base.StopAsync(token);

}

}

* + - 1. To request a BackgroundService in a scoped service like a controller, refer to <https://stackoverflow.com/a/49541063/13036358> . Basically, request IEnumerable<IHostedService> from DI in ctor, then search background.FirstOrDefault(w => w.GetType() == typeof(MyBackgroundService).

then,

if (backgroundService != null)

{

var type = backgroundService.GetType();

if (type == typeof(MyBackgroundService))

{

var obj = (MyBackgroundService) backgroundService;

//do stuff with it

}

}

* + - 1. We can directly use IHostedService as well,

To do so,

public class MyTimerService : IHostedService, IDisposable

{

private Time \_timer;

//constructor that can access DI.

public Task StartAsync(CancellationToken stopToken)

{

\_timer= new Timer(<somefunction>, null, TimeSpan.Zero, TimeSpan.FromSeconds(5));

return Task.CompletedTask;

}

public Task StopAsync(CancellationToken stopToken)

{

\_timer?.Change(Timeout.Infinite,0);

return Task.CompletedTask;

}

public void Dispose()

{

\_timer?.Dispose();

}

}

We can use Interlocked.Increment(ref executionCount); where executionCount is an int property. This increments an int value in an atomic way where only 1 thread updates the value.

* + - 1. Lastly, to queue background tasks, use Channel.Writer.WriteItem and Task.DequeAsync.
    1. Hosting Startup Assemblies: An IHostingStartup implementation adds enhancements to an app startup from another assembly. A class library added to our project contains a [HostingStartup] attribute on top of a namespace and class extending IHosting Startup which is then inserted automatically into our app’’s Startup class and adds features to it.
       1. For example, a class library,

[assembly: HostingStartup(typeof(HostingStartupPackage.ServiceKeyInjection))]

namespace HostingStartupPackage

{

public class ServiceKeyInjection : IHostingStartup

{

public void Configure(IWebHostBuilder builder)

{

builder.ConfigureAppConfiguration(config =>

{

var dict = new Dictionary<string, string>

{

{"DevAccount\_FromPackage", "DEV\_3333333-3333"},

{"ProdAccount\_FromPackage", "PROD\_4444444-4444"}

};

config.AddInMemoryCollection(dict);

});

}

}

}

And now simply adding this library to our app would add the given object to our app’s Startup class and we can access <IConfiguration obj>

[“DevAccount\_FromPackage”]

* + - 1. To enable hosting startup libraries to load,

…webBuilder.UseSetting(WebHostDefaults.HostingStartupAssembliesKey, “{Assembly1;Assembly2;…}”);

or

ASPNETCORE\_HOSTINGSTARTUPASSEMBLIES env var with the same values.

* + - 1. Activation: Hosting startup libraries can be added either at runtime or at compile time. The IHostingStartup library is not referenced by the main app.
         1. Runtime Store: The IHostingStartup class library is placed in the runtime store and then other apps can pick it up.

To do so,

first run the IHostingStartup project with

dotnet store --manifest {MANIFEST FILE} --runtime {RUNTIME IDENTIFIER} --output {OUTPUT LOCATION} --skip-optimization

For example,

dotnet store --manifest store.manifest.csproj --runtime win7-x64 --output ./deployment/store --skip-optimization

This will generate a manifest file,

Then,

dotnet publish {MANIFEST FILE}

the manifest file provided here is the one generated by dotnet store.

This will generate a <somename>.deps.json file,

Then remove ‘runtime’ section, and any ‘manifest’ sections from targets and libraries sections of the <somename>.deps.json file.

Then,

we store the .deps.json file here,

{ADDITIONAL DEPENDENCIES PATH}/shared/{SHARED FRAMEWORK NAME}/{SHARED FRAMEWORK VERSION}/{ENHANCEMENT ASSEMBLY NAME}.deps.json

For example,

deployment/additionalDeps/shared/Microsoft.AspNetCore.App/3.0.0/StartupDiagnostics.deps.json

Lastly, the runtime store is built and then deployed

using a powershell script, which can be found here https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/platform-specific-configuration?view=aspnetcore-5.0#runtime-store

* + - * 1. Compile Time: We have 2 ways here, to directly provide a dll to the main app or to simply include the IHostingStartup class library.

To add the dll, first build the IHostingStartup class library, then copy the generated .dll file from the output folder to the main app’s bin/Debug folder, then add the IHostingStartup class library’s assembly name in the ASPNETCORE\_HOSTINGSTARTUPASSEMBLIES env var and lastly add it in the main app’s csproj file like so

…

<ItemGroup>

<Reference Include=".\\bin\\Debug\\netcoreapp3.0\\HostingStartupLibrary.dll">

<HintPath>.\bin\Debug\netcoreapp3.0\HostingStartupLibrary.dll</HintPath>

<SpecificVersion>False</SpecificVersion>

</Reference>

</ItemGroup>

</Project>

* + - 1. To disable hosting startup libraries to load, by default any hosting startup library in the assembly with Startup.cs (our main app’s Startup.cs assembly) is automatically loaded,

…webBuilder.UseSetting(WebHostDefaults.PreventHostingStartupKey,”true”).UseStartup<Startup>();

…

or set ASPNETCORE\_PREVENTHOSTINGSTARTUP env var to true/1.

To exclude specific assemblies from loading

…webBuilder.UseSetting(WebHostDefaults.HostingStartupExcludeAssembliesKey, “{ASSEMBLY1;ASSEMBLY2;…}”).UseStartup<Startup>();

…

or

ASPNETCORE\_HOSTINGSTARTUPEXCLUDEASSEMBLIES env var with the same string.

If both env var and UseSetting are used for the same key then UseSetting’s value overrides the env var.

* + - 1. We can use Microsoft.AspNetCore.Hosting.Abstractions to add a dynamic hosting startup, i.e., an app doesn’t need to reference it at compile time. The way this would work is that a runnable app (not class library) with that package referenced will have the hosting startup class and then we will target that running app from another app which will receive the injection at runtime instead of compile time.
      2. Change priority of IHostingStartup: We can specify if the IHostingStartup class injects the configuration before our app’s Configuration (**Configure** method) or after it.

To do so,

In the IHostingStartup class,

…

public void Configure(IWebHostBuilder builder)

{

…

var ourConfig= new ConfigurationBuilder();

…//do stuff with it

builder.UseConfiguration(ourConfig); //to add it after the injected app //is done configuring

//or

builder.ConfigureAppConfiguration(ourConfig); //to add it before

}

3.4 Security: .Net Core contains many tools and libraries to deal with security. Authentication, authorization, data protection, HTTPS enforcement, app secrets, XSRF/CSRF prevention and CORS management. We can use built-in or external Identity providers

3.4.1 Authentication and Authorization: Authentication is used to provide credentials to a user, while authorization checks if those credentials are valid and permit user to do an action or not.

Authentication: An authentication service extends IAuthenticationService. It uses authentication handlers (aka schemes) to do authentication related tasks. These same schemes can be used by Authorization to authorize. Said schemes are added using

<IServiceCollection obj>.AddAuthentication(…);

and authentication is enabled using <IApplicationBuilder obj>.UseAuthentication(); just before UseAuthorization(), this adds the Authentication middleware which picks up these schemes and executes them.

For example,

<IServiceCollection obj>.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(JwtBearerDefaults.AuthenticationScheme, options => Configuration.Bind("JwtSettings", options))

.AddCookie(CookieAuthenticationDefaults.AuthenticationScheme, options => Configuration.Bind("CookieSettings", options));

By default there is no scheme enabled and just plainly using UseAuthentication will throw an exception.

The ..AddAuthentication() method takes a default scheme, like CookieAuthenticationDefaults.AuthenticationScheme.

* + - 1. Claims Based Authentication: .Net core uses this model for authentication instead of raw username/password. A claim is a piece of information that describes a part of Identity. Claims are like key-value pairs. Multiple claims and a unique signature make an authentication Token. A single token may have a username, email, etc. details.



* + - * 1. ClaimsPrincipal: Authentication is responsible for providing this object so that authorization can make permission decisions. This has to be generated by one of the schemes. The authentication scheme can select which handler generates the correct set of Claims.
        2. Authentication Scheme/Handler: Extends IAuthenticationHandler or AuthenticationHandler<TOptions> or RemoteAuthenticationHandler<TOptions> (used for remote authentication steps such as Google auth). Constructs an AuthenticationTicket (the authentication token as explained in Claims) object if authentication is successful else returns void or failure. The class also has challenge (when a user is unauthenticated and tries to access a resource) and forbid (when a user is unauthorized to access a resource) actions. When a user tries to challenge then Authorization invokes the challenge method of the scheme being used and similarly forbit method is used.
        3. Basic Cookie based Authentication without Identity: We can use a per-session cookie based authentication (resets after client browser closes),

First,

<IServiceCollection obj>.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme).AddCookie();

AddCookie’s default authentication scheme is different from the scheme CookieAuthenticationDefaults.AuthenticationScheme. By default it is CookieAuthenticationDefaults.AuthenticationScheme(“Cookies”).

Then, enable authentication and authorization middlewares.

Then enable Cookie Policy Middleware by

…

var cookiePolicyOptions = new CookiePolicyOptions

{

MinimumSameSitePolicy = SameSiteMode.Strict, //enables cors

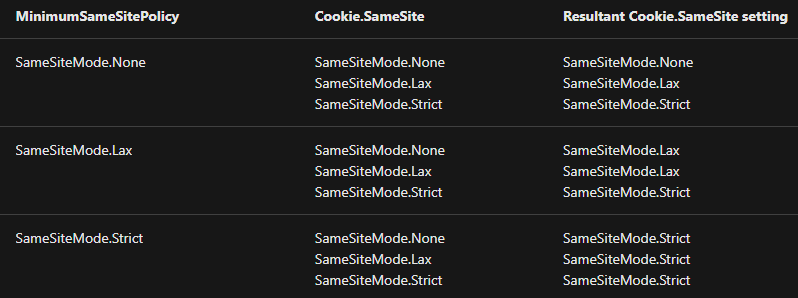
};

<IApplicationBuilder obj>.UseCookiePolicy(cookiePolicyOptions);

//At the top

…

If CookieAuthenticationOptions and Cookie Policy are both set for SameSite policy then this table is followed



This middleware only affects downstream components, the cookie policy applies the given policies to the cookies.

To create an authentication cookie

var claims = new List<Claim>

{

new Claim(ClaimTypes.Name, user.Email),

new Claim("FullName", user.FullName),

new Claim(ClaimTypes.Role, "Administrator"),

};

var claimsIdentity = new ClaimsIdentity(

claims, CookieAuthenticationDefaults.AuthenticationScheme);

var authProperties = new AuthenticationProperties

{

//AllowRefresh = <bool>,

// Refreshing the authentication session should be allowed.

//ExpiresUtc = DateTimeOffset.UtcNow.AddMinutes(10),

// The time at which the authentication ticket expires. A

// value set here overrides the ExpireTimeSpan option of

// CookieAuthenticationOptions set with AddCookie.

//IsPersistent = true,

// Whether the authentication session is persisted across

// multiple requests. When used with cookies, controls

// whether the cookie's lifetime is absolute (matching the

// lifetime of the authentication ticket) or session-based.

//IssuedUtc = <DateTimeOffset>,

// The time at which the authentication ticket was issued.

//RedirectUri = <string>

// The full path or absolute URI to be used as an http

// redirect response value.

};

await HttpContext.SignInAsync(

CookieAuthenticationDefaults.AuthenticationScheme,

new ClaimsPrincipal(claimsIdentity),

authProperties);

adds an encrypted key to the response headers.

await HttpContext.SignOutAsync(

CookieAuthenticationDefaults.AuthenticationScheme);

to logout.

ValidatePrincipal: This event can be used to track back-end changes on cookies. If a cookie isn’t revoked and client continues to use it despite being banned or logged out then it is problematic. To resolve this, we use this event and a handler for it, it brings a performance penalty since the method is triggered at every request. 1 way to implement the back-end tracking is tracking the changes in the database, to do so

var claims = new List<Claim>

{

new Claim(ClaimTypes.Name, user.Email),

new Claim("LastChanged", {Database Value})

};

var claimsIdentity = new ClaimsIdentity(

claims,

CookieAuthenticationDefaults.AuthenticationScheme);

await HttpContext.SignInAsync(

CookieAuthenticationDefaults.AuthenticationScheme,

new ClaimsPrincipal(claimsIdentity));

To implement ValidatePrincipal event handler,

public class CustomCookieAuthenticationEvents : CookieAuthenticationEvents

{

private readonly IUserRepository \_userRepository;

public CustomCookieAuthenticationEvents(IUserRepository userRepository)

{

// Get the database from registered DI services.

\_userRepository = userRepository;

}

public override async Task ValidatePrincipal(CookieValidatePrincipalContext context)

{

var userPrincipal = context.Principal;

// Look for the LastChanged claim.

var lastChanged = (from c in userPrincipal.Claims

where c.Type == "LastChanged"

select c.Value).FirstOrDefault();

if (string.IsNullOrEmpty(lastChanged) ||

!\_userRepository.ValidateLastChanged(lastChanged))

{

context.RejectPrincipal();

await context.HttpContext.SignOutAsync(

CookieAuthenticationDefaults.AuthenticationScheme);

}

}

}

then add it as a scoped service and provide it to the CookieOptions,

<IServiceCollection obj>.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)

.AddCookie(options =>

{

options.EventsType = typeof(CustomCookieAuthenticationEvents);

});

services.AddScoped<CustomCookieAuthenticationEvents>();

In case we want to update user data in the db and not cause a logout, we can use context.ReplacePrincipal and set context.ShouldRenew to true.

We can have a persistent cookie as well,

…

await HttpContext.SignInAsync(

CookieAuthenticationDefaults.AuthenticationScheme,

new ClaimsPrincipal(claimsIdentity),

new AuthenticationProperties

{

IsPersistent = true

ExpiresUtc = DateTime.UtcNow.AddMinutes(20)

});

The ExpiresUtc provides an absolute expiration time, if the browser is closed and cookie expires then it is automatically cleared. This option overrides ExpireTimeSpan of CookieAuthenticationOptions.

* + - * 1. External Authentication Providers: To use them,

<IServiceCollection>.AddAuthentication()

.AddMicrosoftAccount(microsoftOptions => { ... })

.AddGoogle(googleOptions => { ... })

.AddTwitter(twitterOptions => { ... })

.AddFacebook(facebookOptions => { ... });

It is recommended to use these along with a default username/password or Identity based authentication.

For example, setting up Google Auth:

First get Microsoft.AspNetCore.Authentication.Google package and get Client ID and Client Secret from Google Console,

then enable secret storage (explained later in doc) and store the Client ID and secret like so,

dotnet user-secrets set "Authentication\_\_Google\_\_ClientId" "<client-id>"

dotnet user-secrets set "Authentication\_\_Google\_\_ClientSecret" "<client-secret>"

then the **ConfigureServices** can be,

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(

Configuration.GetConnectionString("DefaultConnection")));

services.AddDefaultIdentity<IdentityUser>(options =>

options.SignIn.RequireConfirmedAccount = true)

.AddEntityFrameworkStores<ApplicationDbContext>();

services.AddRazorPages();

services.AddAuthentication()

.AddGoogle(options =>

{

IConfigurationSection googleAuthNSection =

Configuration.GetSection("Authentication:Google");

options.ClientId = googleAuthNSection["ClientId"];

options.ClientSecret = googleAuthNSection["ClientSecret"];

options.CallbackPath= ‘/xy/google’;

});

options.ClaimActions.MapJsonKey("urn:google:picture", "picture", "url");

options.ClaimActions.MapJsonKey("urn:google:locale", "locale", "string");

options.SaveTokens = true;

options.Events.OnCreatingTicket = ctx =>

{

List<AuthenticationToken> tokens = ctx.Properties.GetTokens().ToList();

tokens.Add(new AuthenticationToken()

{

Name = "TicketCreated",

Value = DateTime.UtcNow.ToString()

});

ctx.Properties.StoreTokens(tokens);

return Task.CompletedTask;

};

}

By default the callback uri of google auth is /signin-google by we change it with CallbackPath.

MapJsonKey reads keys/subkeys from the external provider’s JSON user data and passes them to Identity.

If SaveTokens is true then access and refresh tokens would be stored in the auth cookie.

These auth requests must be forwarded through proxy and load balancers. Check the doc for setting them up per server type.

We can have external providers without Identity as well,

// using Microsoft.AspNetCore.Authentication.Cookies;

// using Microsoft.AspNetCore.Authentication.Google;

// NuGet package Microsoft.AspNetCore.Authentication.Google

<IServiceCollection obj>

.AddAuthentication(options =>

{

options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = GoogleDefaults.AuthenticationScheme;

})

.AddCookie()

.AddGoogle(options =>

{

options.ClientId = Configuration["Authentication:Google:ClientId"];

options.ClientSecret = Configuration["Authentication:Google:ClientSecret"];

});

Then adding authentication and authorization middleware components and finally

public async Task<IActionResult> OnPostLogoutAsync()

{

await HttpContext.SignOutAsync();

return RedirectToPage();

}//action method

* + - * 1. Policy Scheme: Allows a single auth scheme to use multiple approaches and select any one dynamically. Any auth scheme that uses AuthenticationSchemeOptions uses policy schemes.

For example:

<IServiceCollection obj>.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)

.AddCookie(options => options.ForwardChallenge = "Google")

.AddGoogle(options => { });

Here ..options is an AuthenticationSchemeOptions type of object. ..ForwardChallenge means CookieAuthentication must forward ForbidAsync calls to “Google” auth scheme. That is, Cookie auth handler handles all calls but Google auth handler handles only 1 type of call.

More ..options.ForwardX methods are defined here:

<https://docs.microsoft.com/en-us/aspnet/core/security/authentication/policyschemes?view=aspnetcore-5.0>

* + - * 1. Certificate Authentication: Microsoft.AspNetCore.Authentication.Certificate contains implementation for Certificate Auth defined by IETF. This type of authentication happens at TLS level, it validates the security certificate.

To use it,

First we need to configure our server, <https://docs.microsoft.com/en-us/aspnet/core/security/authentication/certauth?view=aspnetcore-5.0#configure-your-server-to-require-certificates>

For Kestrel,

…

return Host.CreateDefaultBuilder(args)

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.UseStartup<Startup>();

webBuilder.ConfigureKestrel(o =>

{

o.ConfigureHttpsDefaults(o =>

o.ClientCertificateMode =

ClientCertificateMode.RequireCertificate);

});

});

…

Then reference the cert package, and

<IServiceCollection>.AddAuthentication(CertificateAuthenticationDefaults.AuthenticationScheme).AddCertificate(options=>{

options…. //There are many methods available to validate the cert

options.Events=new CertificateAuthenticationEvents{

OnCertificateValidated= context=> {

var validationService =

context.HttpContext.RequestServices

.GetRequiredService<ICertificateValidationService>();

//Service in the DI to provide additional validations on the //cert

if (validationService.ValidateCertificate(

context.ClientCertificate))

{

var claims= new[]{

new Claim(

ClaimTypes.NameIdentifier,

context.ClientCertificate.Subject,

ClaimValueTypes.String,

context.Options.ClaimsIssuer

),

new Claim(ClaimTypes.Name,

context.ClientCertificate.Subject,

ClaimValueTypes.String,

context.Options.ClaimsIssuer

)

};

context.Prinicpal=new ClaimsPrincipal(new ClaimsIdentity(claims, context.Scheme.name));

context.Success();

}

else{

context.Fail(“Some Reason ”);

}

return Task.CompletedTask;

};

};

}).AddCertificateCache(options.CacheSize = 1024;

options.CacheEntryExpiration = TimeSpan.FromMinutes(2);

);

..AddCertificateCache caches the auth result in memory for a given request address. We can implement ICertificateValidationCache and add the service as a singleton then use that instead as well.

There’s another event, OnAuthenticationFailed which is triggered when validation fails/ exception occurs.

The HttpContext.Request is null/empty at this stage so we cannot use it.

If auth fails this handler returns a Http403 automatically.

Then,

<IApplicationBuilder obj>.UseAuthentication();

Certificate Forwarding: In custom web proxies certificate is passed as a custom request header (like X-SSL-CERT), to configure the handler for it,

…

<IServiceCollection obj>.AddCertificateForwarding(options =>

{

options.CertificateHeader = "X-SSL-CERT";

options.HeaderConverter = (headerValue) =>

{

X509Certificate2 clientCertificate = null;

if(!string.IsNullOrWhiteSpace(headerValue))

{

byte[] bytes = StringToByteArray(headerValue);

clientCertificate = new X509Certificate2(bytes);

}

return clientCertificate;

};

});

//for NGINX/Kubernetes there are differences, refer to docs

…

private static byte[] StringToByteArray(string hex)

{

int NumberChars = hex.Length;

byte[] bytes = new byte[NumberChars / 2];

for (int i = 0; i < NumberChars; i += 2)

{

bytes[i / 2] = Convert.ToByte(hex.Substring(i, 2), 16);

}

return bytes;

}

then,

<IApplicationBuilder obj>.UseCertificateForwarding();

<IApplicationBuilder obj>.UseAuthentication();

<IApplicationBuilder obj>.UseAuthorization();

We can send our own certificate (so that it is validated in our custom validators). To do so, first we create (to create check docs) and store our own cert. Then, we do something like so in a controller,

…controller{

private readonly IHttpClientFactory \_clientFactory;

public ApiService(IHttpClientFactory clientFactory)

{

\_clientFactory = clientFactory;

}

private async Task<JsonDocument> GetApiDataWithNamedClient()

{

var client = \_clientFactory.CreateClient("namedClient");

var request = new HttpRequestMessage()

{

RequestUri = new Uri("https://localhost:44379/api/values"),

Method = HttpMethod.Get,

};

var response = await client.SendAsync(request);

if (response.IsSuccessStatusCode)

{

var responseContent = await response.Content.ReadAsStringAsync();

var data = JsonDocument.Parse(responseContent);

return data;

}

throw new ApplicationException($"Status code: {response.StatusCode}, Error: {response.ReasonPhrase}");

}

}

And we setup the **ConfigureServices** like so,

…ConfigureServices{

var clientCertificate =

new X509Certificate2(

Path.Combine(\_environment.ContentRootPath, "sts\_dev\_cert.pfx"), "1234");

var handler = new HttpClientHandler();

handler.ClientCertificates.Add(clientCertificate);

<IServiceCollection obj>.AddHttpClient("namedClient", c =>

{

}).ConfigurePrimaryHttpMessageHandler(() => handler);

}

SNI: Server Name Indication is used to separate hosts, this can be configures in .net core to allow different cert authentication for diff parts.

We can do the same as SNI with content renogitation but it is inadvised to do so.

Check docs for both.

* + - * 1. MFA: Multi-Factor Authentication can be set up in .net core. To do so, refer the doc.
      1. We can send email (for auth confirmation/ any thing) by extending IEmailSender. Then adding the child to the DI as a transitive service.
      2. Identity: It’s an API that manages users, passwords, profile data, roles, claims, tokens etc. An Identity needs access to a SQL server database to store the data. It is added by using either <IServiceCollection obj>.AddIdentity or ..AddDefaultIdentity.

Sample **ConfigureServices** for Identity based authentication,

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<ApplicationDbContext>(options =>

// options.UseSqlite(

options.UseSqlServer(

Configuration.GetConnectionString("DefaultConnection")));

services.AddDatabaseDeveloperPageExceptionFilter();

services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)

.AddEntityFrameworkStores<ApplicationDbContext>();

services.AddRazorPages();

services.Configure<IdentityOptions>(options =>

{

// Password settings.

options.Password.RequireDigit = true;

options.Password.RequireLowercase = true;

options.Password.RequireNonAlphanumeric = true;

options.Password.RequireUppercase = true;

options.Password.RequiredLength = 6;

options.Password.RequiredUniqueChars = 1;

// Lockout settings.

options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);

options.Lockout.MaxFailedAccessAttempts = 5;

options.Lockout.AllowedForNewUsers = true;

// User settings.

options.User.AllowedUserNameCharacters =

"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-.\_@+";

options.User.RequireUniqueEmail = false;

});

services.ConfigureApplicationCookie(options =>

{

// Cookie settings

options.Cookie.HttpOnly = true;

options.ExpireTimeSpan = TimeSpan.FromMinutes(5);

options.LoginPath = "/Identity/Account/Login";

options.AccessDeniedPath = "/Identity/Account/AccessDenied";

options.SlidingExpiration = true;

});

}

We can create a new Identity() object to store an identity.

* + - * 1. To use Identity in our controller,

we get the UserManager<Tuser> object and store it in the same property, TUser an IdentityUser.

This UserManager has many methods and is used for all tasks by the controller,

var username=await userManager.GetUserNameAsync(“xyz”);

var result = await userManager.PasswordSignInAsync(Input.Email,

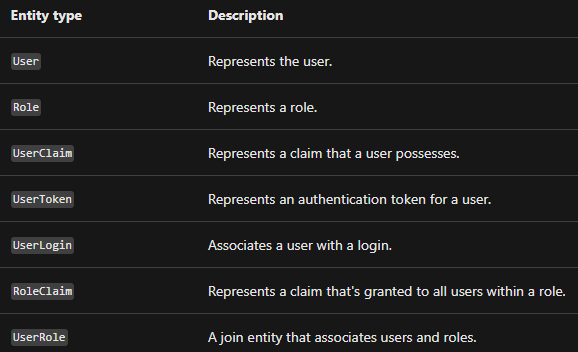
Input.Password, Input.RememberMe,

lockoutOnFailure: false); //to login

await userManager.CreateAsync(<TUser>,”somepassword”);

to create a user and so on.

* + - * 1. Entity types of Identity:



Each of these types are represented with classes with ‘Identity’+ same name.

A single User can have many UserClaims, RoleClaims, UserLogins, UserTokens and Roles. Each Role can have many Users as well, this ‘join’ is UserRole table in the db.

Sample dbcontext’s OnModelCreating method for setting up Identity:

https://docs.microsoft.com/en-us/aspnet/core/security/authentication/customize-identity-model?view=aspnetcore-5.0#default-model-configuration

* + - * 1. When a dbContext class is made for Identity data then it must extend IdentityDbContext,

For example:

public class ApplicationDbContext<TUser>

: IdentityDbContext<TUser, IdentityRole, string>

where TUser : IdentityUser

{

}

Here string is key type, it must be unique for each identity. There are many types of IdentityDbContexts overloaded by generics.

If our Identity object to be used in a dbcontext class doesn’t have ‘Roles’ and has only claims then we can instead use IdentityUserContext<TUser>.

ApplicationDbContext is a name that is used for convention by a dbContext class for identity.

If overriding OnModelCreating of the dbContext then call base.OnModelCreating(); first.

* + - * 1. To create our own Tuser,

public class myUser: IdentityUser  
{

…props

}

then simply

public class ApplicationDbContext : IdentityDbContext<myUser>

{

…

}

the props in myUser will be automatically mapped by EFCore and we don’t need to have a DbSet of either myUser or props inside it. We only need to migrate db if we make any changes to myUser.

To use this TUser and DbContext we can

<IServiceCollection obj>.AddDefaultIdentity<ApplicationUser>(options => options.SignIn.RequireConfirmedAccount = true)

.AddEntityFrameworkStores<ApplicationDbContext>() .AddDefaultUI();

or

<IServiceCollection obj>.AddAuthentication(o =>

{

o.DefaultScheme = IdentityConstants.ApplicationScheme;

o.DefaultSignInScheme = IdentityConstants.ExternalScheme;

})

.AddIdentityCookies(o => { });

services.AddIdentityCore<TUser>(o =>

{

o.Stores.MaxLengthForKeys = 128;

o.SignIn.RequireConfirmedAccount = true;

})

.AddDefaultUI()

.AddDefaultTokenProviders();

The AddDefaultUI method must be used if Identity isn’t scaffolded.

We can change the Primary Key as well, but it must be specified in the initial migration only.

To do so,

public class ApplicationDbContext

: IdentityDbContext<IdentityUser<Guid>, IdentityRole<Guid>, Guid>

{…}

where Guid is the class we uses as PK. This way of creating an IdentityDbContext can also be used as an alternative to creating custom Identity classes with only 1 prop like this one

public class ApplicationUser : IdentityUser<Guid>

{

public string CustomTag { get; set; }

}

and

public class ApplicationDbContext

: IdentityDbContext<ApplicationUser, IdentityRole<Guid>, Guid>

{…}

This doesn’t just apply to TUser but also to every other Identity class.

To add this class,

<IServiceCollection obj>.AddIdentity<ApplicationUser, IdentityRole<Guid>>()

.AddEntityFrameworkStores<ApplicationDbContext>()

.AddDefaultUI()

.AddDefaultTokenProviders();

We can’t use Composite PKs with Identity.

* + - * 1. To change table name or relations or even seed an Identity db,

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

modelBuilder.Entity<IdentityUser>(b =>

{

b.ToTable("MyUsers");

});

}

changes name of TUser to the given name.

* + - * 1. To use lazy-loading with DbContext,

<IServiceCollection obj>

.AddDbContext<ApplicationDbContext>(

b => b.UseSqlServer(connectionString)

.UseLazyLoadingProxies())

.AddDefaultIdentity<ApplicationUser>()

.AddEntityFrameworkStores<ApplicationDbContext>();

plus we need Microsoft.EntityFrameworkCore.Proxies package and the public entity types (props in classes like TUser) to have public virtual navigation props.

For example,

public class ApplicationUser : IdentityUser

{

public virtual ICollection<IdentityUserClaim<string>> Claims { get; set; }

public virtual ICollection<IdentityUserLogin<string>> Logins { get; set; }

public virtual ICollection<IdentityUserToken<string>> Tokens { get; set; }

public virtual ICollection<IdentityUserRole<string>> UserRoles { get; set; }

}

For this the modelBuilder will be,

…

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

modelBuilder.Entity<ApplicationUser>(b =>

{

// Each User can have many UserClaims

b.HasMany(e => e.Claims)

.WithOne()

.HasForeignKey(uc => uc.UserId)

.IsRequired();

// Each User can have many UserLogins

b.HasMany(e => e.Logins)

.WithOne()

.HasForeignKey(ul => ul.UserId)

.IsRequired();

// Each User can have many UserTokens

b.HasMany(e => e.Tokens)

.WithOne()

.HasForeignKey(ut => ut.UserId)

.IsRequired();

// Each User can have many entries in the UserRole join table

b.HasMany(e => e.UserRoles)

.WithOne()

.HasForeignKey(ur => ur.UserId)

.IsRequired();

});

}

* + - * 1. Scaffold Identity: We can add a new scaffolded identity, i.e., by adding a scaffolded item using Visual Studio we can generate most of Identity based authentication code in new or old projects.
        2. IdentityUser: Userdata model class with pre-defined properties, we can extend it to provide our own properties.

For example:

using System;

using Microsoft.AspNetCore.Identity;

namespace WebApp1.Areas.Identity.Data

{

public class WebApp1User : IdentityUser

{

[PersonalData]

public string Name { get; set; }

[PersonalData]

public DateTime DOB { get; set; }

}

}

* + - * 1. Identity Options: The IdentityOptions class is used for configuring the Identity system.

For example:

<IServiceCollection obj>.Configure<IdentityOptions>(options =>

{

//Default Lockout config

options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);

options.Lockout.MaxFailedAccessAttempts = 5;

options.Lockout.AllowedForNewUsers = true;

});

This method is called only after AddIdentity or AddDefaultIdentity.

Similarly options.Password.. is used to set password requirements and same for User and Tokens methods.

* + - * 1. We can configure App’s cookie using

<IServiceCollection obj>.ConfigureApplicationCookie(options =>

{

options.AccessDeniedPath = "/Identity/Account/AccessDenied";

options.Cookie.Name = "YourAppCookieName";

options.Cookie.HttpOnly = true;

options.ExpireTimeSpan = TimeSpan.FromMinutes(60);

options.LoginPath = "/Identity/Account/Login";

// ReturnUrlParameter requires

//using Microsoft.AspNetCore.Authentication.Cookies;

options.ReturnUrlParameter = CookieAuthenticationDefaults.ReturnUrlParameter;

options.SlidingExpiration = true;

});

* + - * 1. We can configure the Password Hasher using

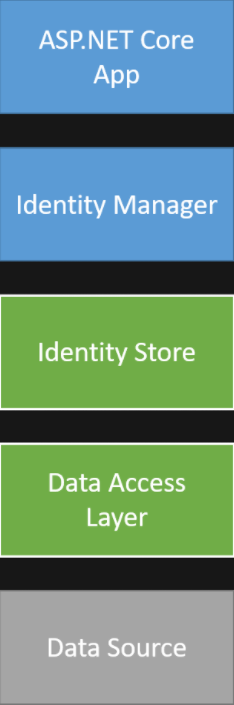
services.Configure<PasswordHasherOptions>(option =>

{

option.IterationCount = 12000;

});

* + - * 1. <IServiceCollection obj>.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme) shouldn’t be used when using Identity as Cookie Providers and Identity clash.
        2. Identity Architecture:



The Identity Manager is used to perform high level operations such as creating an IdentityUser in the db, these objects include UserManager, RoleManager and so on.

The Identity Store contains low level classes, they specify how Identity works in the back, like specifying how users and roles are persisted in the db. The DAL and Data Source (where Identity data is stored, like in SQL Server )along with Stores use Repository pattern.

We can modify the Data Source by using custom providers.

* + - 1. OpenID Authentication: This can be used to generate claims using OpenID connect client.

To use it,

public void ConfigureServices(IServiceCollection services)

{

services.AddAuthentication(options =>

{

options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;

})

.AddCookie()

.AddOpenIdConnect(options =>

{

options.SignInScheme = "Cookies";

options.Authority = "-your-identity-provider-";

options.RequireHttpsMetadata = true;

options.ClientId = "-your-clientid-";

options.ClientSecret = "-your-client-secret-from-user-secrets-or-keyvault";

options.ResponseType = "code";

options.UsePkce = true;

options.Scope.Add("profile");

options.SaveTokens = true;

options.GetClaimsFromUserInfoEndpoint = true;

options.ClaimActions.MapUniqueJsonKey("preferred\_username", "preferred\_username");

options.ClaimActions.MapUniqueJsonKey("gender", "gender");

});

MapUniqueJsonKey adds required claims taken from UserInfoEndpoint.

…options.TokenValidationParameters = new TokenValidationParameters

{

NameClaimType = "email",

// RoleClaimType = "role"

};

can be used to set Name and a Role claim.

In the **Configure** method, using JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();

will disable .net core from adding default namespaces to claims generated by OpenID Connect.

* + - * 1. Add Custom Claims: We use the IClaimsTransformation class,

To do so,

public class MyClaimsTransformation : IClaimsTransformation

{

public Task<ClaimsPrincipal> TransformAsync(ClaimsPrincipal principal)

{

ClaimsIdentity claimsIdentity = new ClaimsIdentity();

var claimType = "myNewClaim";

if (!principal.HasClaim(claim => claim.Type == claimType))

{

claimsIdentity.AddClaim(new Claim(claimType, "myClaimValue"));

}

principal.AddIdentity(claimsIdentity);

return Task.FromResult(principal);

}

}

then

<IServiceCollection obj>.AddTransient<IClaimsTransformation, MyClaimsTransformation>();

This adds the custom claim to the ClaimsPrincipal class.

* + 1. Authorization: Authorization refers to a process where we determine what a user can do. It requires Authentication to be set up.

3.3.11.1 To add authorization to a Razor Page,

First,

<IServiceCollection obj>.AddRazorPages(options=> {

options.Conventions.AuthorizePage(“/Contact”);

options.Conventions.AuthorizeFolder(“/Private”);

options.Conventions.AllowAnonymousToPage(“/Private/PublicPage”);

options.Conventions.AllowAnonymousToFolder(“/Private/P2”);

});

Applies [Authorize] and access type to the given page/folder in razor apps.

A page that is authorized can get anonymous access but the reverse is not true.

3.3.11.2 [Authorize] attribute applied to a controller/action method enables access to the resource only to authenticated users.

[AllowAnonymous] bypasses [Authorize] attribute. Neither of these attributes can be applied to razor page handlers as filter attributes cannot be applied to page handler methods, there is a workaround that can be seen here <https://docs.microsoft.com/en-us/aspnet/core/security/authorization/simple?view=aspnetcore-5.0#authorize-attribute-and-razor-pages> . These attributes can be applied to the PageModel class however.

To require authenticated users globally,

<IServiceCollection obj>.AddAuthorization(options=>{

options.FallbackPolicy= new AuthorizationPolicyBuilder().RequreAuthenticatedUser().Build();

});

The fallback policy is used by controllers/methods that don’t have [Authorize] attribute or [AllowAnonymous] attribute.

Another way of doing the same is by using Authorization filter,

<IServiceCollection obj>.AddControllers(config =>

{

// using Microsoft.AspNetCore.Mvc.Authorization;

// using Microsoft.AspNetCore.Authorization;

var policy = new AuthorizationPolicyBuilder()

.RequireAuthenticatedUser()

.Build();

config.Filters.Add(new AuthorizeFilter(policy));

});

3.3.11.3 Role-based Authorization:

[Authorize(Role=”xyz,abc”)]

<controller/action method>

limits access of the resource to the given roles only. The ‘,’ acts as an OR operator, role must be a OR b.

[Authorize(Role=”xyz”)]

[Authorize(Role=”abc”)]

…

requires a user to be of both the roles to access the resource. This is like an AND operator.

To enable Role services in Identity,

<IServiceCollection obj>.AddDefaultIdentity<IdentityUser>().AddRoles<IdentityRole>().AddEntityFrameworkStores<AppDbContext>();

3.3.11.4 Claims-based Authorization: Authorization by performing checks on the claims. These checks are performed by the policy so we have to build and register policy to do so.

For example:

<IServiceCollection obj>.AddAuthorization(options =>

{

options.AddPolicy("EmployeeOnly", policy => policy.RequireClaim("EmployeeNumber"));

});

We have declared a policy that has the given name and its handler returns a bool that is true if the given claim exists in the HttpRequest.

policy.RequireClaim("EmployeeNumber",”1”,”2”));

does the same as above + needs the value of the claim be 1,2.

Then call UseAuthentication and UseAuthorization in **Configure**.

To use it,

[Authorize(Policy = "EmployeeOnly")]

<action method/controller>

We can apply multiple policies to a resource and that means all of them must pass to enable access to the resource.

3.3.11.5 Policy-based Authorization: This is the recommended way to enable any type of authorization (role/claim/etc.). We register a policy and then we apply it on action methods/controllers and that’s simply it.

To use it,

<IServiceCollection obj>.AddAuthorization(options=>{

options.AddPolicy(“somepolicyname”, policy=> policy.Requirements.Add(new MinimumAgeRequirement(21));

});

where MinimumAgeRequitement is a class that extends IAuthorizationRequirement.

Multiple requirements can be added and all must pass

To apply this policy,

[Authorize(Policy=“somepolicyname”)]

<controller/pagemodel/action method>

they can’t be applied to page handlers, only to pagemodels.

3.3.11.5.1 IAuthorizationRequirement:

using Microsoft.AspNetCore.Authorization;

public class MinimumAgeRequirement : IAuthorizationRequirement

{

public int MinimumAge { get; }

public MinimumAgeRequirement(int minimumAge)

{

MinimumAge = minimumAge;

}

}

3.3.11.5.2 IAuthorizationHandler: This class receives the IAuthorizationRequirement object.

For example:

public class MinimumAgeHandler : AuthorizationHandler<MinimumAgeRequirement>

{

protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,

MinimumAgeRequirement requirement)

{

if (!context.User.HasClaim(c => c.Type == ClaimTypes.DateOfBirth &&

c.Issuer == "http://contoso.com"))

{

//TODO: Use the following if targeting a version of

//.NET Framework older than 4.6:

// return Task.FromResult(0);

return Task.CompletedTask;

}

var dateOfBirth = Convert.ToDateTime(

context.User.FindFirst(c => c.Type == ClaimTypes.DateOfBirth &&

c.Issuer == "http://contoso.com").Value);

int calculatedAge = DateTime.Today.Year - dateOfBirth.Year;

if (dateOfBirth > DateTime.Today.AddYears(-calculatedAge))

{

calculatedAge--;

}

if (calculatedAge >= requirement.MinimumAge)

{

context.Succeed(requirement);

}

//TODO: Use the following if targeting a version of

//.NET Framework older than 4.6:

// return Task.FromResult(0);

return Task.CompletedTask;

}

}

This handler only accepts MinimumAgeRequirement type of IAuthorizationRequirement.

To handle multiple requirement types either specify them in the HandleRequirementAsync or specify no requirement,

… someHandler…

{

protected override Task HandleRequirementAsync (AuthorizationHandlerContext context)

{

var pendingRequirements= context.PendingRequirements.ToList();

…

}

}

A handler can set context.Succeed(<IAuthorizationRequirement>) to tell if the authorization succeeded or context.Fail or do nothing to tell if authorization failed (Task.Completed is simply used to tell the compiler to wait for awaits before it and then return void from the function).

The difference between context.fail and not doing anything is that context.fail guarantees the authorization will fail and can be configured to short-circuit the handler chain at it. Normally context.succeed, context.fail and not doing anything allow the chain to continue processing, this behavior can be configured by setting InvokeHandlersAfterFailure to false which stops the chain at first context.fail.

Lastly we must register the handler,

<IServiceCollection obj>.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();

We can bundle IAuthorizationHandler and IAuthorizationRequirement together and then just use that in a policy without registering the class in DI.

2 or more handlers can be registered for the same requirement, in that case if either of those handlers return success then success is considered.

The HandleRequirementAsync receives an AuthorizationHandlerContext object, in MVC/Razor Pages when authorization occurs due to a filter the context.Resource is of type AuthorizationFilterContext and has access to HttpContext, RouteData and everything provided by MVC/Razor Pages.

When endpoint routing is used the context.Resource is of type HttpContext and has access to underlying resource, like

…

var endpoint = httpContext.GetEndpoint();

var actionDescriptor = endpoint.Metadata.GetMetadata<ControllerActionDescriptor>();

…

The handler can access the DI using constructor injection, however if EF is being requested in the handler then it mustn’t be registered as a singleton.

3.3.11.5.3 IAuthorizationService: Paired with IAuthorizationHandler and IAuthorizationRequirements this service determines if authorization succeeded for an item. By default it determines success if all handlers succeed.  
<https://docs.microsoft.com/en-us/aspnet/core/security/authorization/policies?view=aspnetcore-5.0#iauthorizationservice>

on how to use it.

3.3.11.5.4 RequireAssertion: This method can be used to supply a func to define a policy (skip creating an authorization handler/ requirement).

<IServiceCollection obj>.AddAuthorization(options =>

{

options.AddPolicy("BadgeEntry", policy =>

policy.RequireAssertion(context =>

context.User.HasClaim(c =>

(c.Type == "BadgeId" ||

c.Type == "TemporaryBadgeId") &&

c.Issuer == "https://microsoftsecurity")));

});

3.3.11.5.5 IAuthorizationPolicyProvider: This class can be extended to provide our own policy provider, i.e., instead of manually adding each requirement to the AddAuthorization we can simply create this class and add it as singleton in the DI. We don’t even need to call AddAuthorization if we do so.

To do so,

<https://docs.microsoft.com/en-us/aspnet/core/security/authorization/iauthorizationpolicyprovider?view=aspnetcore-5.0>

Unlike raw AddAuthorization this also enables us to create custom AuthorizeAttribute which can be used to dynamically provide values to the policy requirements, to do so

internal class MinimumAgeAuthorizeAttribute : AuthorizeAttribute

{

const string POLICY\_PREFIX = "MinimumAge";

public MinimumAgeAuthorizeAttribute(int age) => Age = age;

// Get or set the Age property by manipulating the underlying Policy property

public int Age

{

get

{

if (int.TryParse(Policy.Substring(POLICY\_PREFIX.Length), out var age))

{

return age;

}

return default(int);

}

set

{

Policy = $"{POLICY\_PREFIX}{value.ToString()}";

}

}

}

Then we simply use this class instead of [Authorize] attribute, provide different values per controller etc.

3.3.11.6 IAuthorizationMiddlewareResultHandler: Extend this class to define how to handle the result of authorization.

To do so,

public class MyAuthorizationMiddlewareResultHandler : IAuthorizationMiddlewareResultHandler

{

private readonly AuthorizationMiddlewareResultHandler

DefaultHandler = new AuthorizationMiddlewareResultHandler();

public async Task HandleAsync(

RequestDelegate requestDelegate,

HttpContext httpContext,

AuthorizationPolicy authorizationPolicy,

PolicyAuthorizationResult policyAuthorizationResult)

{

// if the authorization was forbidden and the resource had specific requirements,

// provide a custom response.

if (Show404ForForbiddenResult(policyAuthorizationResult))

{

// Return a 404 to make it appear as if the resource does not exist.

httpContext.Response.StatusCode = (int)HttpStatusCode.NotFound;

return;

}

// Fallback to the default implementation.

await DefaultHandler.HandleAsync(requestDelegate, httpContext, authorizationPolicy,

policyAuthorizationResult);

}

bool Show404ForForbiddenResult(PolicyAuthorizationResult policyAuthorizationResult)

{

return policyAuthorizationResult.Forbidden &&

policyAuthorizationResult.AuthorizationFailure.FailedRequirements.OfType<

Show404Requirement>().Any();

}

}

public class Show404Requirement : IAuthorizationRequirement { }

Then, register the class in the DI.

3.3.11.7 Imperative Authorization: Instead of using [Authorize] or just AuthroizeAttribute based authroization which occurs before data binding or execution of the action method we can use IAuthorizationService to load it at our desired time.

To use it,

…controller…

{

//IAuthorizationService object from constructor

..IActionResult..{

if (Document == null)

{

return new NotFoundResult();

}

var authorizationResult = await \_authorizationService

.AuthorizeAsync(User, Document, "EditPolicy");

if (authorizationResult.Succeeded)

{

return Page();

}

else if (User.Identity.IsAuthenticated)

{

return new ForbidResult();

}

else

{

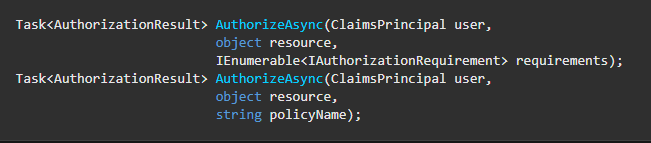
return new ChallengeResult();

}

}

}

where AuthorizeAsync can accept 2 overloads,



This approach allows an action method to decide how it wants to deal with authorization for a specific request.

We can use the AuthorizationHandler to create a resource based handler, instead of using Imperative Authorization. This allows us to write a requirement which can be then used by the action method.

To do so,

public class DocumentAuthorizationHandler :

AuthorizationHandler<SameAuthorRequirement, Document>

{

protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,

SameAuthorRequirement requirement,

Document resource)

{

if (context.User.Identity?.Name == resource.Author)

{

context.Succeed(requirement);

}

return Task.CompletedTask;

}

}

public class SameAuthorRequirement : IAuthorizationRequirement { }

then we add the handler and requirement to the DI. Then we add the policy for it.

We can use OperationAuthorizationRequirement for simple operations like CRUD. This requirement type can be used by resource handler and the action method.

* + 1. AuthenticationScheme: To use multiple authentication types,

For example:

<IServiceCollection obj>.AddAuthentication()

.AddCookie(options => {

options.LoginPath = "/Account/Unauthorized/";

options.AccessDeniedPath = "/Account/Forbidden/";

})

.AddJwtBearer(options => {

options.Audience = "http://localhost:5001/";

options.Authority = "http://localhost:5000/";

});

We are using 2 authentication handlers,

to use both,

[Authorize(AuthenticationSchemes = CookieAuthenticationDefaults.AuthenticationScheme + "," +

JwtBearerDefaults.AuthenticationScheme)]

…

We can select an authentication scheme with policy as well,

<IServiceCollection obj>.AddAuthorization(options =>

{

options.AddPolicy("Over18", policy =>

{

policy.AuthenticationSchemes.Add(JwtBearerDefaults.AuthenticationScheme);

policy.RequireAuthenticatedUser();

policy.Requirements.Add(new MinimumAgeRequirement());

});

});

We can declare multiple authentication schemes of same type if we use different scheme names for each.

<IServiceCollection obj>.AddAuthorization(options =>

{

var defaultAuthorizationPolicyBuilder = new AuthorizationPolicyBuilder(

JwtBearerDefaults.AuthenticationScheme,

"AzureAD");

defaultAuthorizationPolicyBuilder =

defaultAuthorizationPolicyBuilder.RequireAuthenticatedUser();

options.DefaultPolicy = defaultAuthorizationPolicyBuilder.Build();

});

like so.

* 1. We can create multiple instances of singleton services by

var obj=<IServiceCollection obj>.BuildServiceProvider();

BuildServiceProvider() essentially calls the **ConfigureServices** again (it only stores the singleton services and doesn’t call/use the same variable used for creating it to avoid infinite recursion) and returns an IServiceCollection object with the configured services. This object is scoped only to the variable that stores the result of BuildServiceProvider. The benefit of using this method is that it returns a fully configured IServiceCollection object, we can use that in the configuring IServiceCollection object of the **ConfigureServices.** It’s like an alternative to using **Configure** method toget a configured IServiceCollection obj (as this method is called after **ConfigureServices** isfinished).

* + 1. There are 2 ways of resolving dependencies manually, first is the one way given above but it only works in **ConfigureServices,** and the 2nd is by using IServiceProvider (which can be requested from the DI). In the **Configure** method, the IApplicationBuilder has an ApplicationServices method which is provides this object. The Startup’s ctor can ask it in its params too but only very few services are in the DI container by then. The first method or asking for those services in other classes ensure all services added to DI are available.
    2. To manually resolve dependencies in **ConfigureServices** we have 2 ways, the first is as given above, BuildServiceProvider and the other is by using AddSingleton/AddTransient/AddScoped. These methods have an overload which passes IServiceProvider to the methods.

For example:

<IServiceCollection obj>.AddSingleton<X>(sp=>{

var xy=sp.GetRequiredService<XY>;

return new X(xy);

});

* + 1. The IServiceProvider object has 2 methods to get services, GetRequiredService<T> and GetService<T>. The former throws an exception if the service isn’t found, the latter returns null.
    2. Manually resolving dependencies is considered anti-pattern and should be avoided.
  1. Data Protection: Involves API used for protecting data, in .net core this means protecting data that is sent to client and returned and making sure it is the desired data without being tampered. There are 5 main packages for 3 main audiences, Consumer APIs (for simple DP use cases), Configuration APIs (for slightly more advanced use cases like configuring settings for the DP) and Extensibility APIs (for defining everything in DP).

To protect data, there are 3 steps:

1. Create Data Protector
2. Call Protect method of it on the data meant to be protected.
3. Call Unprotect on already protected data to get plain text.
   * 1. The most basic protector is provided in DI by using AddDataProtection().

To use it,

<IServiceCollection obj>.AddDataProtection();

then in controller,

using Microsoft.AspNetCore.DataProtection;

…{

IDataProtector obj

…constructor(IDataProtectionProvider obj2){

obj=obj2.CreateProtector(‘<PurposeString>’)

…

}

}

The purpose string can be any string, it provides isolation between different protectors.

then simply use < IDataProtector obj>.protect(<string data or byte[] data>) to protect and < IDataProtector obj>.unprotect(<string or byte[]>) to unprotect. Unprotect throws CryptographicException if it fails.

* + 1. Purpose String: A unique string that seperates different protectors.

A protector can only protect/unprotect strings created using the same purpose string. By convention we use namespace.classname.version as purpose strings, this provides pretty good nomenclature for defining different purpose strings for different classes in different namespaces in different versions. The CreateProtector can also take a string array,

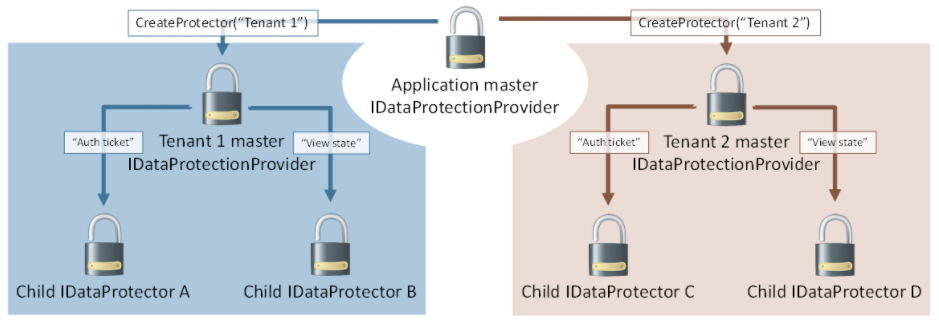
..CreateProtector([“xyz”,”abc”])

This is the same as with a single purpose string but provides modularity. It also allows multiple PurposeStrings to be chained,

For example:

..CreateProtector([“xyz”,”abc”]) is the same as ..CreateProtector(“xyz”).CreateProtector(”abc”).

This allows us to define multi-tenancy. That is, we can allow users or other sources to define purpose strings and nest them inside our purpose strings so now even if a user has a same purpose string in tenant 1 as another in tenant 2 they would not be the same.



The DP system already uses multi-tenancy where parents are app names, app identifiers etc.

Purpose String or array cannot be null or length 0.

* + 1. We can use ..AddOptins to configure DP.

<IServiceCollection obj>.AddDataProtection().AddKeyManagementOptions(o=> {…});

This may need the use of BuildServiceProvider.

Alternatively,

<IServiceCollection obj>.AddDbContext<DataProtectionDbContext>(…);

<IServiceCollection obj>.AddDataProtection();

<IServiceCollection obj>.AddOptions<KeyManagementOptions>().Configure<IServiceScopeFactory>((options, factory)=> {

options.XmlRespository= new CustomXmlRepository(factory);

});

We have access to the singleton services here without needing BuildServiceProvider. This factory object can be used like so

…

using (var scope = factory.CreateScope())

{

var context = scope.ServiceProvider.GetRequiredService<DataProtectionDbContext>();

…

}

…

* + 1. We can use IDataProtector in Main() method of a .net core app as well, to do so,

using Microsoft.AspNetCore.DataProtection;

using Microsoft.Extensions.DependencyInjection;

…Main(){

var serviceCollection = new ServiceCollection();

serviceCollection.AddDataProtection();

var services = serviceCollection.BuildServiceProvider();

var protector = services.GetDataProtector("Contoso.Example.v2");

…

}

Here GetDataProtector is an alternative/shorthand to CreateProtector and storing the IDataProtector. It supports all params supported by CreateProtector.

* + 1. Password Hashing: The .net core DP provides KeyDerivation.Pbkdf2 which uses the algo with the same name to hash passwords. The Rfc2898DeriveBytes method from .net is it’s predecessor and shouldn’t be used over it as it can be slower.

To use it,

var pass=”somepassword”;

byte[] salt= new byte[128/8];

using (var rngCsp= new RNGCryptoServiceProvider())

{

rngCsp.GetNonZeroBytes(salt);

}

//generates 128 bit salth using random sequence of non-zero values.

string hashed= Convert.ToBase64String(KeyDerivation.Pbkdf2(

password: pass,

salt: salt,

prf: KeyDerivationPrf.HMACSHA256,

iterationCount: 100000,

numBytesRequested: 256/8

));

* + 1. Time based data protector: We can use this API to allow self-expiring protected payloads, that they will be rendered invalid after a set duration.

To use it,

…controller{

ITimeLimitedDataProtector obj

…constructor (IDataProtectionProvider obj2)  
{

var protector= obj2.CreateProtector(…);

obj= protector.ToTimeLimitedDataProtector();

}

Then we simply use protect and unprotect, the only difference that now these protect and unprotect also have overloads for a DateTimeOffset objects which define expiry (we can store absolute expiry time by passing DateTimeOffset objects or relative expiry time by passing TimeSpan objects to these). The unprotect takes an out parameter of DateTimeOffset type which stores the remaining type of a protected payload. Unprotect will throw CryptographicException if the payload is expired.

* + 1. The Data Protector API uses purpose strings and keys to protect/unprotect data. The keys can be managed manually as well but by default are managed automatically, they can be revoked.

Like purpose strings, different keys can protect/unprotect their data only.

To revoke keys,

IKeyManager obj can be requested from the DI and obj.RevokeAllkeys(…); can be used to revoke all keys.

* + 1. Persisted Data Protector: While keys can be expired/revoked, their protected data can still be retrieved. This is done by using this protector. It is dangerous to use this as the data could also be tampered data.

We use IPersistedDataProtector for this.

It has this method

DangerousUnprotect(byte[] protectedData, bool ignoreRevocationErrors,

out bool requiresMigration, out bool wasRevoked) : byte[]

requireMigration is true if the key used to protect this payload is no longer the active default key, the wasRevoked is true if the key was revoked and will throw if ignoreRevocationErrors is set to false.

To use it,

<IServiceCollection obj>.AddDataProtection().PersistkeysToFileSystem(…).ProtectKeysWithDpapi();

then

…controller{

IPersistedDataProtector obj;

…constructor(IDataProtectionProvider obj2){

obj= obj2 as IPersistedDataProtector; //this can fail in which case obj //will be null

}

…

//then simply use obj.DangerousUnprotect(…);

}

* + 1. Configuring Data Protection:
       1. Persisting Keys: Keys are by default stored in %LOCALAPPDATA%\ASP.NET\DataProtection-Keys in windows and

$HOME/.aspnet/DataProtection-Keys in linux

but that can be changed,

To do so,

<IServiceCollection obj>.AddDataProtection().PersistKeysToFileSystem(<some dir>);

..PersistKeysToDbContext<T>(); persists keys in given dbcontext. The dbContext class must have

public DbSet<DataProtectionKey> DataProtectionKeys {get;set;}

where DataProtectionKey is provided by IDataProtectionKeyContext class (i.e., extend this class instead of DbContext)

* + - 1. ProtectKeysWith\* : These methods can be use to protect keys at rest (stored keys). For example,

..PersistKeysToFileSystem(…).ProtectKeysWithCertificate(new X509Certificate2(“cert.pfx’,Configuration[“Somethig”]));

Appending .UnprotectKeysWithAnyCertificate(<same param as ProtectKeysWithCertificate>) to this method allows DPAPI to unprotect the said keys, it can work even without this method as it uses the protected keys instead. But multiple UnprotectKeysWithAnyCertificate can be chained which allows DPAPI to rotate keys, i.e., use any cert to protect/unprotect a key which will be used in the protection/unprotection of data.

* + - 1. ..SetDefaultKeyLifetime(<TimeSpan obj>) allows us to set a default key lifetime (key used to (un)protect data and not the resting keys). By default this is 90 days.
      2. Share DPAPI across multiple apps: By default, the DPAPI each app by using purpose strings defined by its own version, package name, app name etc.

To allow multiple .net apps to (un)protect and work on the same data,

use ..SetApplicationName(“somename”) and set “somename” to be similar for all the apps.

and reference the same Data Protection package version for all the apps.

* + - 1. Disable automatic key generation: By default when a key is about to reach expiry, it is automatically rolled (new key is created and stored in its place with same amount of expiration time as was when the key was originally created).

We can disable this behavior by

..DisableAutomaticKeyGeneration();

* + - 1. It is advised to use different places to store each app’s keyring, if one app is compromised the other is as well if they use the same directory. Even if protection-at-rest is used.
      2. Changing Algorithm used by newly-generated keys: This can be done by

..UseCryptographicAlgorithms(new AuthenticatedEncryptorConfiguration(){

EncryptionAlgorithm= EncryptionAlgorithm.AES\_256\_CBC,

ValidationAlgorithm= ValidationAlgorithm.HMACSHA256

});

These are also the default algos.

We can provide custom algos as well, to do so

..UseCustomCryptographicAlgorithms(

new ManagedAuthenticatedEncryptorConfiguration()

{

// A type that subclasses SymmetricAlgorithm

EncryptionAlgorithmType = typeof(Aes),

// Specified in bits

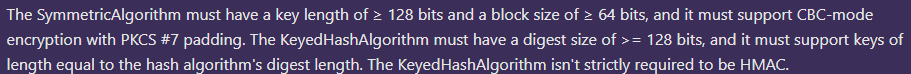
EncryptionAlgorithmKeySize = 256,

// A type that subclasses KeyedHashAlgorithm

ValidationAlgorithmType = typeof(HMACSHA256)

});

The custom algo must follow some rules. The EncryptionAlgorithmType and ValidationAlgorithmType take a class that has a public parameterless ctor (constructor) and implements SymmetricAlgorithm and KeyedHashAlgorithm.



* + - 1. When hosting in a Docker container, keys should be stored in a Docker Volume that persists beynd the container’s lifetime. We can also use Redis but the versions must support Redis Data Persistence.
      2. We can configure DPAPI on a machine-wide scale using Registry Keys and environment variables.
      3. Non-DI aware scenarios: For cases where we don’t want to / can’t rely on DI to get data protection APIs we can use

using Microsoft.AspNetCore.DataProtection;

var myObj=DataProtectionProvider.Create(<directory info obj>, config=> {…});

var protector=myObj.CreateProtector(…);

where config is used to configure the DP provider.

This DP provider doesn’t automatically encrypt files at rest, is expensive to create and maintain and doesn’t offer app isolation. That is, the only purpose strings here are the ones we define in CreateProtector and no hidden purpose string is used automatically on it.

* + 1. IAuthenticatedEncryptor: This interface defines the most basic block of a cryptographic system. We extend this class and override Encrypt and Decrypt methods to define how a value is encrypted or decrypted. Refer to docs on how to use it.
    2. Keys: The DPAPI manages the lifetime of master keys in following stages, Created, active, expired and revoked. A key can be in any of the stage. All keys except revoked keys are used to unprotect data, but only active keys are used to protect it. It may end up using expired keys to protect data when automatic key generation is disabled and no other key except revoked key is available. Manually deleting keys is not preferred since it renders the protected data of the key unusable.

Keys have an activation period of T+2 days and expiration period of T+90 days by default, i.e., it will remain in Created state for 2 days then it will become active in the keyring. If there isn’t a key to be used the DPAPI will create a new key instead of falling back to keys of other stages unless automatic key generation is disabled.

We can define custom lifetime with ..SetDefaultKeyLifetime(<timespan>);

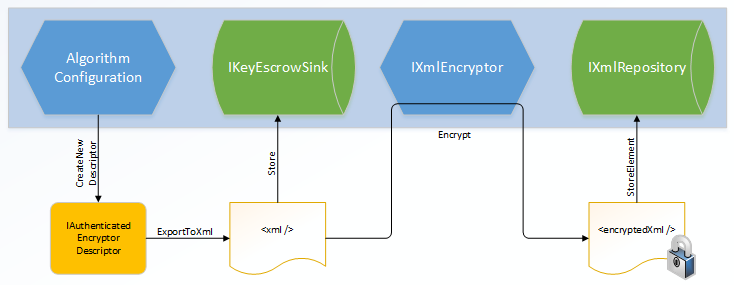
This overrides system-wide policy set for the same.

The IKeyManager can be requested from the DI. This object can be used to request all keys in the keyring, we can revoke all keys using it as well. Doing so makes DPAPI’s in-memory cache reset and it gets a new key from the backing store on next (un)protect operation. DPAPI stores a key in the cache for 24 hrs before requesting a new one from the store.

Ikey is used to define a key. IKeyManager operates on these said keys. XmlKeyManager is a concrete implementation of IKeyManager. It basically stores encrypted keydata as XML and uses IXmlRepository to store/retrieve data from an XML store. We can configure it by

<IServiceCollection obj>.Configure<KeyManagementOptions>(options => options.XmlRepository = new MyCustomXmlRepository());

IKeyEscrowSink is an interface that is used to escrow key data before it passes through IXmlEncryptor. It can be implemented to store the key outside the app and can then be picked up by an external authority.



The basic mechanism of encryption.

We can use IKeyManager.CreateNewKey to trigger creation of a new key.

* + 1. EphemeralDataProtectionProvider: We use this class to generate an IDataProtectionProvider object and then call CreateProtector to create an IDataProtector. This class is creates an ephemeral data protection provider, meaning the given object doesn’t use the physical memory but only stores keys in-memory (with a unique master key generated for each run) and doesn’t write anything to the backing store either. We can (un)protect payload using it but it will only work with the same instance.

var provider = new EphemeralDataProtectionProvider();

var protector = provider.CreateProtector(purpose);

…

* 1. Secret Manager: It is inadvisable to store secrets (connection strings, encryption keys etc.) in the same directory as the project. Nor should they be checked into source control. We can store these details in the environment variables and then use them in the project. Or we can use the secret manager provided by .net core (it is recommended to use this only in dev env and not prod).

To use it,

dotnet user-secrets init

from the terminal of project in visual studio.

This adds

<UserSecretsId>

in the project conf file with an auto-generated value.

or

right-click the project in visual studio and it will give an option to manage user secrets. This can be used to both, create the user secrets file and also manage it.   
  
We can specify values in the generated secrets.json, either through vs or manually. It is at %APPDATA%\Microsoft\UserSecrets\<user\_secrets\_id>\secrets.json in win and ~/.microsoft/usersecrets/<user\_secrets\_id>/secrets.json in linux.

The CLI can be used to set “keys:values” or remove and also to set –project <project dir>.

Then we need to register it,

in development environment ..CreateDefaultBuilder(args) on Host automatically calls AddUserSecrets. We can manually call it with

Host.ConfigureAppConfiguration((hostContext, builder) =>

{

// Add other providers for JSON, etc.

if (hostContext.HostingEnvironment.IsDevelopment())

{

builder.AddUserSecrets<Program>();

}

})…

Then to use the values provided simply use IConfiguration which now also reads the secrets.json.

3.7.1 We can also use configuration’s object graph binding feature to bind secret file’s content. To do so,

…

var moviesConfig =

Configuration.GetSection("Movies").Get<MovieSettings>();

for a secrets.json,

{

"Movies:ConnectionString": "Server=… ",

"Movies:ServiceApiKey": "12345"

}

where MovieSettings is,

public class MovieSettings

{

public string ConnectionString { get; set; }

public string ServiceApiKey { get; set; }

}

3.7.2 We can also store values as secret, to do so,

dotnet user-secrets set “key” “value”

This value isn’t stored in secrets.json and can be normally accessed by IConfiguration.

* 1. HTTPS: Don’t use [RequireHttps] attribute on WebAPI since it causes URL redirection which may not be understood by clients. To disable URL redirection, set ASPNETCORE\_URLS env var to a desired https url. Similarly HSTS shouldn’t be used with WebAPI since it’s a browser only instruction and other devices may not understand it.
     1. To use HTTPS

<IApplicationBuilder obj>.UseHttpsRedirection()

middleware to redirect requests.

<IApplicationBuilder obj>.UseHsts()

to send HSTS headers to clients.

Both must be used.

HSTS headers must be recognized by the browser, if they are then the browser ensures all comms take place over HTTPS. It is recommended to use the middleware only in prod env.

* + 1. HSTS options: We can configure HSTS by using

<IServiceCollection obj>.AddHsts(options =>

{

options.Preload = true;

options.IncludeSubDomains = true;

options.MaxAge = TimeSpan.FromDays(60);

options.ExcludedHosts.Add("example.com");

options.ExcludedHosts.Add("www.example.com");

});

The preload param allows browsers to directly use the HTTPS ports instead of HTTP ones. But for the preload to work, the webapp must be configured as given by <https://hstspreload.org/>

* + 1. HTTPS redirection requires an empty port to redirect the request to.

We can specify the port with,

ASPNETCORE\_HTTPS\_PORT env var

or

https\_port in appsettings.json

or

specify the port in the url given to ASPNETCORE\_URLS

or

<IServiceCollection obj>.AddHttpsRedirection(options =>

{

options.RedirectStatusCode = (int) HttpStatusCode.TemporaryRedirect; //Http307 status code or use ..PermanentRedirect to send Http308

options.HttpsPort = 5001;

});

This method is only needed to be called when these options are to be specified.

* + 1. Firefox and linux need a bit more HTTPS configuration to allow trusting dev certificates. Refer to <https://docs.microsoft.com/en-us/aspnet/core/security/enforcing-ssl?view=aspnetcore-5.0&tabs=visual-studio#trust-the-https-certificate-with-firefox-to-prevent-sec_error_inadequate_key_usage-error> for the same.
    2. For production certificates, <https://letsencrypt.org/> can be used to get certs. It is advised to not store prod certs in site directory.

To use a certificate, Get a .pfx cert file

dotnet dev-certs https -ep %USERPROFILE%\.aspnet\https\aspnetapp.pfx -p { password here }

dotnet dev-certs https --trust

For docker,

docker pull mcr.microsoft.com/dotnet/core/samples:aspnetapp

docker run --rm -it -p 8000:80 -p 8001:443 -e ASPNETCORE\_URLS="https://+;http://+" -e ASPNETCORE\_HTTPS\_PORT=8001 -e ASPNETCORE\_Kestrel\_\_Certificates\_\_Default\_\_Password="password" -e ASPNETCORE\_Kestrel\_\_Certificates\_\_Default\_\_Path=/https/aspnetapp.pfx -v %USERPROFILE%\.aspnet\https:/https/ mcr.microsoft.com/dotnet/core/samples:aspnetapp

Replace %USERPROFILE% with $env:USERPROFILE in Powershell for all commands.

In linux the %USERPROFILE will be replaced by ${HOME} for all commands

<https://github.com/dotnet/dotnet-docker/blob/main/samples/run-aspnetcore-https-development.md> for developing .netcore apps on docker.

* 1. GDPR Regulations: Follow GDPR regulations by adding cookie consent, to do so

<IServiceCollection obj>.Configure<CookiePolicyOptions>(options =>

{

// This lambda determines whether user consent for non-essential

// cookies is needed for a given request.

options.CheckConsentNeeded = context => true;

// requires using Microsoft.AspNetCore.Http;

options.MinimumSameSitePolicy = SameSiteMode.None;

});

and then

<IApplicationBuilder obj>.UseCookiePolicy(); //right before UseRouting and right after UseStaticFiles and UseHttpsRedirection.

Then enable cookie consent by requesting user.

* + 1. Encryption at rest must be enables as well. For DBs that support it like SQL Server, use Transparent Data Encryption (<https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/transparent-data-encryption?view=sql-server-ver15> ) . For local files and dbs that don’t support it, use disk encryption and encrypt the disk storing the files/db.
  1. CSRF/XSRF Protection: Inbuilt authentication providers like Cookie based or Token based auth provide protection against this attack, for token we must make sure to not store it in the cookie.

Shared hosting envs are vulnerable so we must host different apps in diff domains, xyz.abc.com and zzz.abc.com for example must be xyz.aaa.com and zzz.abc.com.

* + 1. We can use the Html.AntiforgeryToken() method in html forms but it isn’t required as MVC and Razor apps add the same automatically.
    2. To configure AntiForgery,

<IServiceCollection obj>.AddAntiforgery(options =>

{

// Set Cookie properties using CookieBuilder properties†.

options.FormFieldName = "AntiforgeryFieldname";

options.HeaderName = "X-CSRF-TOKEN-HEADERNAME";

options.SuppressXFrameOptionsHeader = false;

});

* + 1. To configure IAntiforgery,

<IApplicationBuilder obj>.Use(next => context =>

{

string path = context.Request.Path.Value;

if (

string.Equals(path, "/", StringComparison.OrdinalIgnoreCase) ||

string.Equals(path, "/index.html", StringComparison.OrdinalIgnoreCase))

{

// The request token can be sent as a JavaScript-readable cookie,

// and Angular uses it by default.

var tokens = antiforgery.GetAndStoreTokens(context);

context.Response.Cookies.Append("XSRF-TOKEN", tokens.RequestToken,

new CookieOptions() { HttpOnly = false });

}

return next(context);

});

Here we are generating a token and storing it in the response, done using a middleware.

* + 1. Validation: We can make antiforgery token required on a controller/action method by using [ValidateAntiForgeryToken] attribute. Or we can use [AutoValidateAntiforgeryToken] attribute, the difference between this attribute and the former is that this can be directly applied to a controller without using [IgnoreAntiForgeryToken] on GET methods as we would do with [ValidateAntiForgeryToken]. GET requests shouldn’t require the token hence if we use [ValidateAntiForgeryToken] on a controller we must use [IgnoreAntiForgeryToken] on GET action methods, the [AutoValidateAntiforgeryToken] doesn’t apply the same to GET, HEAD, OPTIONS and TRACE action methods.
    2. For more security, refresh the tokens after successful logins.
    3. It is recommended to send the AntiForgery token as a header than a cookie.
  1. Protection against Open Redirection attacks: If a URL redirection is performed and the redirecting url is plainly given in url/cookie/query parameter etc. then an attacker could modify that and supply the modified part with original url to user. We can prevent this.
     1. LocalRedirect: Return this instead of Redirect() method when redirecting, it will only redirect to local URLs
     2. Url.isLocalUrl(<string url>); returns a bool if the string url is of the same app.
  2. Basic rule of protection is, trust no data given by user or which can be accessed by user. Login infos, forms, images, files etc..

We follow the same with HTML pages, if an HTML page takes an input then it must validate,encode and clean/sanitize the input before storing/returning the data otherwise it poses an XSS vulnerability. All 3 steps are **important**.

If we have to display user input data then we can use any of the 3 available encoders,

UrlEncoder, JavascriptEncoder and HtmlEncoder. These can be configured to (dis)allow certain characters with

<IServiceCollection obj>.AddSingleton<HtmlEncoder>(

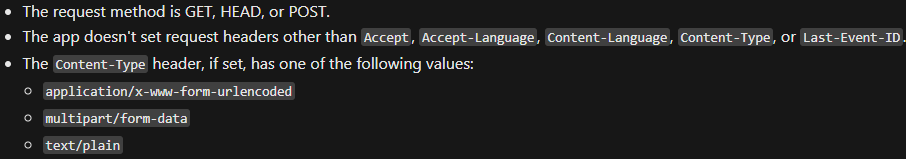
HtmlEncoder.Create(allowedRanges: new[] { UnicodeRanges.BasicLatin,

UnicodeRanges.CjkUnifiedIdeographs }));

Where UnicodeRanges provides ranges referenced from <https://www.unicode.org/charts/index.html>

* 1. CORS: Browsers prevent a client from making request to another domain, this restriction is called same-origin policy (2 domains are same origin if they have same schemes, hosts and ports. In https:// google.com/h and <https://google.com/y> , the scheme is https, port is 443 or as taken from DNS, the host is google.com. The host includes TLD, domain and sub level domain.). CORS is a standard that allows a server to relax the said policy.

The way CORS works is, first a pre-flight request is sent. This is nothing just browser sending an extra request first (even before sending actual http request headers or body), the pre-flight request is not sent iff (the request is directly sent instead but the response is discarded if the response headers aren’t the correct ones)



The ‘request method’ specified in the image refers to the actual http request’s method and not the pre-flight request’s method. The pre-flight request’s method is HTTP OPTIONS.

The server then returns a CORS response which has Access-Control-…-… keys in the header and HTTP 200 status code. If the pre-flight request is denied by the server’s Cors handler then these keys have empty values but the status code is the same, this lets the browser know that CORS has failed and hence disallows sending of actual request. Browser then shows an error in the console for the same.

If pre-flight request receives a response with the above mentioned headers set and Http200 then that means the server accepts the client/domain. Then it checks the header values and determines if the actual request should be sent or not.

If the pre-flight request was not sent then the actual request is sent directly and the same rules still apply but now the CORS response headers are appended to the actual response headers.

To enable CORS,

We can use a named policy/default policy.

or

We can use endpoint routing.

or

We can use [EnableCors] attribute.

Contrarily we can use [DisableCors] when we want a controller to not need it.

and then

<IApplicationBuilder obj>.UseCors();

This must be positioned before the UseResponseCaching() middleware.

When using endpoint routing, this method must be called between UseRouting and UseEndpoints.

That is, if we are using CORS we must define it in the DI (default and/or named policy) then we must add it in the middleware chain.

* + 1. To configure CORS:

string MyAllowSpecificOrigins = "\_myAllowSpecificOrigins";

<IServiceCollection obj>.AddCors(options =>

{

options.AddPolicy(name: MyAllowSpecificOrigins,

builder =>

{

builder.WithOrigins("http://example.com",

"http://www.contoso.com")

.AllowAnyHeader()

.AllowAnyMethod();

});

});

then,

<IApplicationBuilder obj>.UseCors(MyAllowSpecificOrigins);

Here, ..AddPolicy is used to add a named policy where MyAllowSpecificOrigins provides the name for the policy. The same name is also provided to the CORS middleware.’

The CorsPolicyBuilder provided in the lambda is used to actually configure the CORS middleware. It can be chained as well. The URL in the ..WithOrigins method must not end with ‘/’. The URL given to WithOrigins is the one Cors compares the incoming requests against, these are the URLS we want to pass the CORS.

We can use options.AddDefaultPolicy(builder=>{…}); instead as well, when this is used we don’t pass a policy name to UseCors either.

* + 1. Endpoint routing with CORS:

..UseCors(); //default policy

<IApplicationBuilder obj>.UseEndpoints(endpoints =>

{

endpoints.MapGet("/echo",

context => context.Response.WriteAsync("echo"))

.RequireCors(MyAllowSpecificOrigins);

endpoints.MapControllers()

.RequireCors(MyAllowSpecificOrigins);

endpoints.MapGet("/echo2",

context => context.Response.WriteAsync("echo2"));

endpoints.MapRazorPages();

});

Here we are using default policy for the rest of the system and named policy with endpoint routing cors. That is, MyAllowSpecificOrigins named policy is added to the <IServiceCollection obj> and then we pick it up here, while we use default policy for the rest of the middleware components.

If we don’t define the default policy for CORS then ..UseCORS() alone can’t enable CORS for the rest of the components.

* + 1. If default policy is defined in <IServiceCollection obj> and ..UseCors() is used then CORS is enabled and configured for all endpoints. It is enabled globally.

If default policy is not defined and ..UseCors() is used then CORS is enabled but is not configured, it means Cors middleware is enabled to be used but isn’t configured and hence it isn’t enabled globally. We can use Endpoint routing with cors or [EnableCors] attribute in this scenario.

[EnableCors] or RequireCors() with parameterless ctor uses the default policy, but it is inadvised to use these as the default policy is enabled globally anyway because of ..UseCors().

A named policy doesn’t enable Cors globally until ..UseCors(<policy name>) is used. It is advised to use

No default policy

Named policy

Endpoint routing with cors/Cors Attribute with named policy as parameter

..UseCors() //parameterless ctor.

.Net core responds to pre-flight requests automatically when CORS is enabled globally or [EnableCors] attribute is used, it doesn’t respond to the same with endpoint routing with Cors as of yet.

* + 1. [EnableCors]: This attribute can be applied to controller/ action method/page model and provides an alternative to enabling it globally. An overload of the attribute accepts a named policy string so we can apply named policies too, given we define them in <IServiceCollection obj> first and also use the ..UseCors() middleware. It is recommended to either use this or apply it globally but not both.
    2. [DisableCors]: Disables Cors for the given controller/action method/page model. It cannot disable cors when defined through endpoint routing with Cors.
    3. CorsPolicyBuilder options:

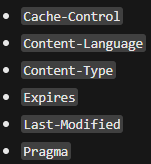
..AllowAnyOrigin(): allows any origins to pass the CORS.

..SetIsOriginAllowedToAllowWildcardSubdomains(): Accepts a lambda function that determines if the request can access the server.

..AllowAnyMethod(): Allows any HTTP method. The Access-Control-Allow-Methods header is set to \* meaning all methods are allowed.

..WithHeaders(‘abc’,’my header’…): Allows the given headers to be in the CORS request. The CORS request is declined if the defined headers aren’t in the request, it is also declined if any header other than the default headers + defined headers (in WithHeader) is received.

..AllowAnyHeader()

..WithExposedHeaders(‘abc’,’my header’…): By default, the browser doesn’t allow the client app to see all the response headers from the pre-flight or skipped pre-flight response. But this method defines headers which the browser must allow the client to see. By default, 

are the headers that browser does make visible.

..AllowCredentials(): allows credentials to be sent through CORS. By default they aren’t. This is a 2 way option, the client sending request must set <XMLHttpRequest obj>.withCredentials to true.

This method can’t be used along with AllowAnyOrigin() as it is prohibited by the CORS specification.

..SetPreflightMaxAge(<TimeSpan obj>): Sets the cache duration of the pre-flight request.

* + 1. Check the preflight request:

…

[HttpOptions("{id}")]

public IActionResult PreflightRoute(int id)

{

return NoContent();

}

//

[HttpOptions]

public IActionResult PreflightRoute()

{

return NoContent();

}

…

These action methods can handle the preflight requests as they are sent as normal requests but with HTTP Options method. We use these when using endpoint routing with Cors as we need to define them manually with that method of using cors.

These action methods only handle preflight requests sent to their route, so each controller can define them separately if needed.

* + 1. By default browsers don’t show HTTP Options requests. This behavior can be changed by

https://docs.microsoft.com/en-us/aspnet/core/security/cors?view=aspnetcore-5.0#display-options-requests-1

* 1. Share Cookies among apps: We do so for an SSO (single sign on) behavior.

To do so, when using Identity

<IServiceCollection obj>.AddDataProtection()

.PersistKeysToFileSystem("{PATH TO COMMON KEY RING FOLDER}")

.SetApplicationName("SharedCookieApp");

<IServiceCollection obj>.ConfigureApplicationCookie(options => {

options.Cookie.Name = ".AspNet.SharedCookie";

});

where all these values are same for all the apps.

If Identity is not being used then,

…AddDataProtection(…);

<IServiceCollection obj>.AddAuthentication("Identity.Application")

.AddCookie("Identity.Application", options =>

{

options.Cookie.Name = ".AspNet.SharedCookie";

});

..cookie.path can be modified to set different HttpRequest.PathBase for cookies, as cookies need the same base path to be shared this can be used to do the same.

To share the cookies across subdomains, we use

options.Cookie.Domain = ".contoso.com";

where ‘.contoso.com’; means any subdomain will share the cookie.

Lastly a common Identity database and schema must be referenced by all the apps sharing the cookie if Identity is being used.

* 1. SameSite: SameSite header affects what can be used by pages in <iframe> and other places, it is a way of combatting against CSRF attacks. The value must be ‘None’ for cross-site cookie use. <iframe> faces issues against Samesite=Lax and ‘Strict’.

…

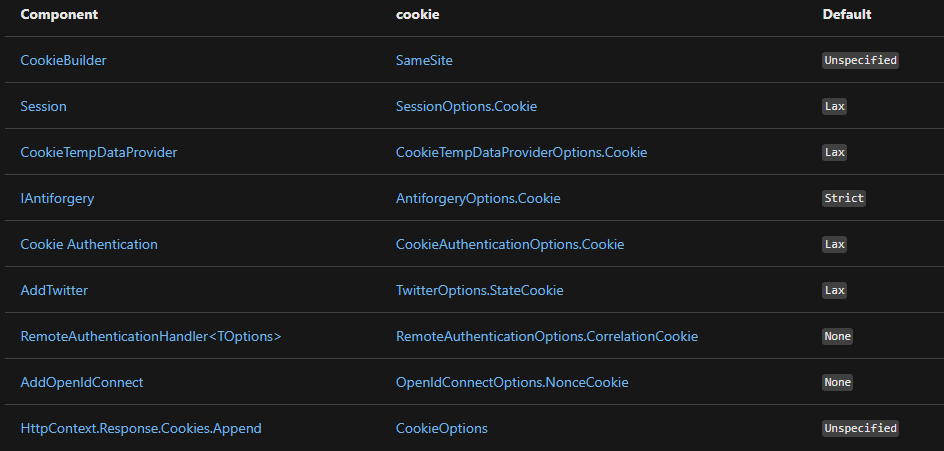
HttpContext.Response.Cookies.Append(

"name", "value",

new CookieOptions() { SameSite = SameSiteMode.Lax });

to change the value for a response.

Default values: By default the SameSite value is ‘Unspecified’ which is interpreted as ‘Lax’ for new browsers and ‘None’ for old ones.



* 1. IP Safelist: We can do so by defining a safe ip list and then (dis)allowing them in the app using filters.

• Swagger: A tool to generate documentation and help debugging by providing automatically generated UI.

OpenAPI is the specification and Swagger refers to a family of products that work with OpenAPI specification.

2 main OpenAPI implementation for .Net are SwashBuckle and NSwag.

To install SwashBuckle, search SwashBuckle.AspNetCore in nuget.

1. To add Swagger Doc Generator,

<IServiceCollection obj>.AddSwaggerGen();

then,

<IApplicationBuilder obj>.UseSwagger(); //Place it at the top of middleware //pipeline. This command enable Swagger’s JSON endpoint.

<IApplication Builder obj>.UseSwaggerUI(c=>{

c.SwaggerEndpoint(“/swagger/v1/swagger.json”,”MyAPI name”);

c.RoutePrefix=string.empty;

}); //Place it after UseSwagger(). This enables Swagger’s UI.

The swagger endpoint means <host>/<swagger endpoint>.json will open the generated doc in json. <host>/swagger opens swagger UI.

If we set RoutePrefix to anything then that is used to visit swagger UI instead, if its empty string then the swagger ui is at <host>/

If RoutePrefix is defined separately then the swagger json endpoint is <host>/<RoutePrefix>/<swagger endpoint>.json

1. Swagger relies on ApiExplorer service to discover routes and endpoints and generate doc for them, which happens automatically for AddMvc but not for AddMvcCore.
2. Add Author info using Swagger:

<IServiceCollection obj>.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new OpenApiInfo

{

Version = "v1",

Title = "ToDo API",

Description = "A simple example ASP.NET Core Web API",

TermsOfService = new Uri("https://example.com/terms"),

Contact = new OpenApiContact

{

Name = "Shayne Boyer",

Email = string.Empty,

Url = new Uri("https://twitter.com/spboyer"),

},

License = new OpenApiLicense

{

Name = "Use under LICX",

Url = new Uri("https://example.com/license"),

}

});

});

1. Adding ‘///’ comments and enclosing content within <summary> and </summary> allows swagger to take the content as definition for given action method.

Syntax:

///<summary>

/// My action method

///</summary>

<action method>

1. Encose within <remarks> and </remarks> creates action method’s documentation with the content.
2. Attributes are also picked up by swagger gen, like Produces/Consumers/Route etc.
3. Enclose content within <response code =”<HTTPStatusCode>”> and </response> to define meaning for each response code that an action method can return.
4. To add custom stylesheet for Swagger,

<IApplicationBuilder obj>.UseStaticFiles();

then

…

<IApplicationBuilder obj>.UseSwaggerUI(c=>

{

c.SwaggerEndpoint(…);

c.InjectStyleSheet(‘<somelocation>’); //it searches within wwwroot folder.

}

);

1. Use OpenAPI tools to generate docs automatically using a code generator.

**4. Net CLI**

We can use the cli to set env vars, cli args and specify launch options for any .net app.

In the .Net CLI

4.1 dotnet run: to run dotnet app.

dotnet run --project <PROJECT> to specify which project.

4.2 set <myKey>=”<some value>” : Sets env vars.

set <anotherKey>=<myKey> to copy vals

setx <myKey> <some value> only windows, sets persisting env vars. Use /M at end to set the env var in system env, without it env var is set in user env.

4.3 We can use : or \_\_ (double \_) as delimiters, both mean the same thing to .net. It is recommended to use \_\_ as : is not supported by Bash.

4.4 Similar to how we access json key values, ‘set’ accesses them.

4.5 Logging: We can set logging value as   
set Logging\_LogLevel\_Microsoft=Information

it means the same as Logging:LogLevel:Microsoft as if we are accessing the appsettings.json directly.

4.6 dotnet trace tool enables collection of traces of a running process.

To use it,

//After starting app

dotnet trace ps //shows processes and their pids

dotnet trace collect -p {PID} --providers <define providers, level and filters >

* 1. dotnet-aspnet-codegenerator: Runs the .net scaffolding engine, which basically generates areas/controllers/etc. and even defines their general layout such as parameters etc. Pretty useful tool, check docs on how to use it.
  2. dotnet watch: It’s a tool that runs a given command when source file changes. Any dotnet command can be run with dotnet watch, like dotnet watch –project WebApp run …

Now if we make changes to any file in the project and save, the watcher will restart the app and log messages in the console for the same.

* 1. dotnet dev-certs https –trust : Run this in the project’s terminal to add the HTTPS development cert to the trusted certs.

For security reasons, DOTNET\_GENERATE\_ASPNET\_CERTIFICATE env var must be set to false to disallow automatic cert generation which will be stored in the image/machine and pose security risk.

* 1. dotnet dev-certs https –clean cleans the trusted certificate cache.