GitHub ref: <https://github.com/dotnet/aspnetcore>

ASP.Net Core

[Versions history 2](#_Toc67511448)

[Useful links/literature 3](#_Toc67511449)

[General features 4](#_Toc67511450)

[1 Base infrastructure 5](#_Toc67511451)

[1.1 Entry point 5](#_Toc67511452)

[1.2 Startup 6](#_Toc67511453)

[IApplicationBuilder utility methods 7](#_Toc67511454)

[1.3 Processing pipeline 7](#_Toc67511455)

[Middleware 7](#_Toc67511456)

[1.4 Environment 9](#_Toc67511457)

[1.5 Dependency Injection 10](#_Toc67511458)

[Default mechanism 10](#_Toc67511459)

[2 Middleware components 10](#_Toc67511460)

[2.1 Static files – app.UseStaticFiles() 10](#_Toc67511461)

[2.2 Viewing web app directory – app.UseDirectoryBrowser() 11](#_Toc67511462)

[2.3 Error handling 11](#_Toc67511463)

[2.3.1 Viewing actual error - UseDeveloperExceptionPage 11](#_Toc67511464)

[2.3.2 Error redirection – UseExceptionHandler 12](#_Toc67511465)

[2.3.3 Status code processing – UseStatusCodePages 12](#_Toc67511466)

[2.4 HTTPS components 13](#_Toc67511467)

[2.4.1 HTTPS redirection – UseHttpsRedirection 13](#_Toc67511468)

[2.4.2 HTTP Strict Transport Security Protocol (HSTS) – UseHsts 13](#_Toc67511469)

[2.5 Other components 14](#_Toc67511470)

[4 Configuration 15](#_Toc67511471)

[4.1 Default Configuration 15](#_Toc67511472)

[4.2 Configuration providers 16](#_Toc67511473)

[4.3 Configuraiton sections 16](#_Toc67511474)

[4.4 Custom configuration classes 17](#_Toc67511475)

[4.5 IOptions (Preferable way) 18](#_Toc67511476)

[5 HTTPContext 18](#_Toc67511477)

[5.1 Sessions 19](#_Toc67511478)

[5.1 Useful members/methods 19](#_Toc67511479)

[6 Logging 20](#_Toc67511480)

[6.1 Custom Logger 20](#_Toc67511481)

# Versions history

Link: <https://dotnet.microsoft.com/download/dotnet-core>

Asp.net core version (as ASP.Net) inherits version of .net core

|  |  |  |  |
| --- | --- | --- | --- |
| **Version Number** | **Release Date** | **End of Support** | **Supported Visual Studio Version(s)** |
| 1.0 | 2016-06-27 | 2019-06-27 | [Visual Studio](https://en.wikipedia.org/wiki/Microsoft_Visual_Studio) 2015, 2017 |
| 1.1 | 2016-11-18 | 2019-06-27 | Visual Studio 2015, 2017 |
| 2.0 | 2017-08-14 | 2018-10-01 | Visual Studio 2017 |
| 2.1 | 2018-05-30 | 2021-08-21[[6]](https://en.wikipedia.org/wiki/ASP.NET_Core#cite_note-6) | Visual Studio 2017 |
| 2.2 | 2018-12-04 | 2019-12-23[[8]](https://en.wikipedia.org/wiki/ASP.NET_Core#cite_note-:0-8) | Visual Studio 2017 15.9 and 2019 16.0 preview 1 |
| 3.0 | 2019-09-23 | 2020-03-03[[8]](https://en.wikipedia.org/wiki/ASP.NET_Core#cite_note-:0-8) | Visual Studio 2017 and 2019 |
| 3.1 (ASP.NET Core) | 2019-12-03 |  | Visual Studio 2019 |
| 5.0 (ASP.NET) | 2021-01-12 |  | Visual Studio 2019 (v16.8) |

Current latest version of ASP.Net is 5.0 (.NET 5.0) on 05.02.2021

# Useful links/literature

1. Metanit: <https://metanit.com/sharp/aspnet5/1.1.php>

# General features

* Новый легковесный и модульный конвейер HTTP-запросов
* Возможность развертывать приложение как на IIS, так и в рамках своего собственного процесса
* Использование платформы .NET Core и ее функциональности
* Распространение пакетов платформы через NuGet
* Интегрированная поддержка для создания и использования пакетов NuGet
* Единый стек веб-разработки, сочетающий Web UI и Web API
* Конфигурация для упрощенного использования в облаке
* Встроенная поддержка для внедрения зависимостей
* Расширяемость
* Кроссплатформенность: возможность разработки и развертывания приложений ASP.NET на Windows, Mac и Linux
* Развитие как open source, открытость к изменениям

# 1 Base infrastructure

## 1.1 Entry point

When you start ASP.Net Core app it starts from Program.cs

Example:

public class Program

{

public static void Main(string[] args)

{

CreateHostBuilder(args).Build().Run();

}

public static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.UseStartup<Startup>();

});

}

This class creates and configures our Web Server. Main parts of Program class:

1. Main method – entry point method which creates host builder, build host (**IHost)** and run it
2. Host.CreateDefaultBuilder(args) – creates IHostBuilder which do following:
   1. Set root directory
   2. Set Host configuration (environment variables with prefix "DOTNET\_" are loaded + args of command string)
   3. Set App configuration (via loading appsettings.json)
   4. Set Logging providers
3. ConfigureWebHostDefaults – configure host params
   1. Load environment params with “ASPNETCORE\_" prefix
   2. Run and configure Kestrel web server (or make IIS integration)
   3. Add Host Filtering component (to configure Kestrel addresses)
4. webBuilder.UseStartup<Startup>() – set “Start class” (class whose instance will be used to start requests handling)
5. CreateHostBuilder(args).Build().Run() – initiate build using builders from 2nd step + launches web app.

## 

## 1.2 Startup

public class Startup

{

// This method gets called by the runtime. Use this method to add services to the container.

// For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?LinkID=398940

public void ConfigureServices(IServiceCollection services)

{

}

// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseRouting();

app.UseStaticFiles();

app.UseEndpoints(endpoints =>

{

endpoints.MapGet("/", async context =>

{

await context.Response.WriteAsync("Hello World!");

});

endpoints.MapGet("/test", async context =>

{

await context.Response.WriteAsync("Hello World! <b> test bold text</b> (from localhost/test");

});

});

}

}

Startup – main class of the ASP.Net Core app, it’s entry point which setup various services and any other infrastructure to handle incoming requests.

Startup class is specified in Program.cs and this class must meet following requirements to be selected as application entry point:

1. There must be **ConfigureServices(IServiceCollection services)** method – first method which is invoked by ASP.Net, this methods registers all necessary services which will be used in app.
2. There must be **Configure()** method
3. There can be constructor

**Constructor**

There can be injected following:

* IConfiguration – configuration of the application (from appsettings.json)
* IWebHostEnvironment – environment info.

This info can be saved into local fields and used in Configure()

**ConfigureServices(IServiceCollection services)**

* This method contains all services registrations
* All registrations are places into IserviceCollection
* There can be extension methods to register services e.g. services.AddControllersWithViews which adds MVC services
  + Names of such extension methods should beginning from “Add”

**Configure(…)**

Configure(IApplicationBuilder app, IWebHostEnvironment env) – this method configures the way how a request will be processed (via configuring required ‘app’ parameter)

Parameters:

1. app – using methods of this object you can setup necessary components which will used in request processing;
2. env – non-required parameter which contains info about environment within application is working.

There can be transferred any service registred in ConfigureServices method.

### IApplicationBuilder utility methods

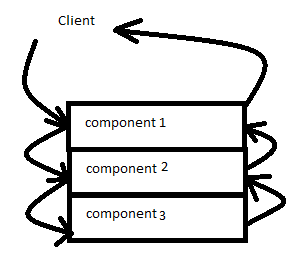
Useful methods:

1. UseEndpoints – setup addresses and handlers for them
2. UseDeveloperExceptionPage – shows detailed error info (can be used for development purposes)

## 1.3 Processing pipeline

### Middleware

Request processing mechanism is based on middleware. I.e. there are some n middleware components which process incoming request: Client’s request comes inside middleware components from 1 to n and then exits components from n to 1.



Facts about Middleware components:

* They are configured in Startup.Configure
* Their order in Startup.Configure determines order in request processing pipeline
* Every component can process request before next components invocation, then invoke next component and finally execute some code after next components invocation.
* Objects for custom middleware classes are singletones.
* Component which performs writing in response must be only one and last (it can’t invoke next component)

Variants of adding new middleware component:

* Inline component i.e. method which implements delegate:

public delegate Task RequestDelegate(HttpContext context);

* Separate class

Components are including into pipeline via IApplicationBuilder methods:

1. **Run()** – only for inline components or lambda functions

app.Run(async (context) =>

{

await context.Response.WriteAsync("Hello World!");

});

1. **Use()** (or custom extension **UseXXX()**) – it invokes method with 2 params: http context and object which will be used to invoke next component.

app.Use(async (context, next) =>

{

x = x \* 2; // processing before all next components

await next.Invoke(); // invoke next component

x = x \* 2; // processing after all next components

await context.Response.WriteAsync($"Result: {x}");

});

1. **Map()** (or custom extensions **MapXXX()**) – includes certain component for certain request path

app.Map("/index", Index);

..

}

private static void Index(IApplicationBuilder app)

{

app.Run(async context =>

{

await context.Response.WriteAsync("Index");

});

}

This method can also simplify components mapping with the same base request path

app.Map("/home", home =>

{

home.Map("/index", Index);

home.Map("/about", About);

});

1. **MapWhen(Func<HttpContext, bool>)** – includes component in pipeline if certain condition is met.

app.MapWhen(context =>

{

return context.Request.Query.ContainsKey("id") &&

context.Request.Query["id"] == "5";

}

, HandleId);

. . . . .

private static void HandleId(IApplicationBuilder app)

{

app.Run(async context =>

{

await context.Response.WriteAsync("id is equal to 5");

});

}

1. **UseMiddleware(Type, object[])** (or generic version **UseMiddleware<T>(object[])**) – includes middleware component which presents as separate class:

//Class must contain:

//1. Constructor with RequestDelegate param

//2. InvokeAsync (or Invoke) method with HttpContext parameter and return Task

//3. It can contain additional parameters in constructor

public class TestMiddleware

{

private readonly RequestDelegate \_next;

//Class must contain constructor with RequestDelegate

public TestMiddleware(RequestDelegate next, string additionalParameter)

{

this.\_next = next;

}

public async Task InvokeAsync(HttpContext context)

{

// do something

await \_next.Invoke(context);

// do something

}

}

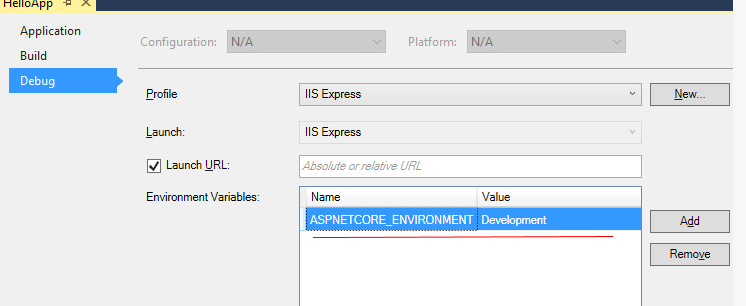
Usage example: app.UseMiddleware<IApplicationBuilder>("data for additional parameter");

## 1.4 Environment

IWebHostEnvironment – contains info about current environment:

1. ApplicationName – application name.
2. EnvironmentName – contains env description e.g. Development, Staging, Production or custom.
3. ContentRootPath – root folder of web app.
4. WebRootPath – folder for static files (by default it’s ‘appFolder\wwwroot’).
5. ContentRootFileProvider – provider for reading files from ContentRoot folder.
6. WebRootFileProvider – the same provider but for WebRoot folder.

Example of setting Environment name in app properties:



Env settings can be set in **launchSettings.json** (in properties folder)

Env name can be checked via IsEnvironment(string envName), IsDevelopment(), IsStaging() and etc methods.

## 1.5 Dependency Injection

### Default mechanism

Standard DI is represented by **IServiceProvider**.

Services registered in **Startup.ConfigureServices** via **IServiceCollection**.

public void ConfigureServices(IServiceCollection services)

{

var servicesAmount = services.Count;

}

**ServiceDescriptor** is item of ServiceCollection which have properties:

1. Lifetime – life time scope of the service
2. ServiceType – type of absraction
3. ImplementationType – type of implementation

Services registration options:

* services.AddTransient<TimeService>();
* services.AddTransient<IMessageSender, EmailMessageSender>();
* make extenstion for IServiceCollection – AddTimeService()
* services.AddTransient<TimeService>(factoryMethod) – you can use factory to setup certain type of instance in dependance on various conditions (e.g. switch between types by certain value in Conriguration)

Registered services can be accessed:

* in Startup.Configure() as a parameter
* in middleware component Invoke() as a parameter
* in custom middleware class via constructor parameter (only singletones) or parameter in Invoke method
* in class constructor as a parameter (class must be also registered)
* via HttpContext.RequestServices
* via IApplicationBuilder.ApplicationServices

Lifecycle types:

* Transient – creates obj instance on every type request
* SingleTone – creates obj instance only one time and use it for all type requests
* Scoped – creates obj instance per http request

Notes:

1. ASP.Net Core adds some services by default e.g: **ILogger<T>, ILoggerFactory, IWebHostEnvironment**
2. Registered services can be added to **Startup.Configure** as a parameters.

# 2 Middleware components

## 2.1 Static files – app.UseStaticFiles()

Static files – any file (js, image, css and etc) which can be requested via special url: <http://websitename.com/contentFolderRelativePathToTheFile>.

Component configuration:

1. Add middleware – app.UseStaticFiles();
2. Add ‘wwwroot’ folder (or overridden folder) where static files will present

Facts:

1. Default ‘wwwroot’ folder can be overridden in Program.cs using **webBuilder.UseWebRoot("test")**;
2. Static files folder can be matched with certain relative URL using **StaticFileOptions** parameter

Example:

app.UseStaticFiles(); // process all requests to 'wwwroot'

app.UseStaticFiles(new StaticFileOptions() //process requests to 'wwwroot/html'

{

FileProvider = new PhysicalFileProvider(Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot\html")),

RequestPath = new PathString("/pages")

});

## 2.2 Viewing web app directory – app.UseDirectoryBrowser()

This component enables viewing certain folders of web application

Component configuration:

1. Add middleware – app.UseDirectoryBrowser()

Facts:

1. Folder can be matched with certain url using **DirectoryBrowserOptions** param which is consumed by UseDirectoryBrowser() (note: Microsoft.Extensions.FileProviders reference must be added).

Example:

app.UseDirectoryBrowser(new DirectoryBrowserOptions()

{

FileProvider = new PhysicalFileProvider(Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot\html")),

RequestPath = new PathString("/pages")

});

## 2.3 Error handling

### 2.3.1 Viewing actual error - UseDeveloperExceptionPage

This component enables viewing detailed information about occurred error (used within dev env).

Component configuration:

1. Add middleware – app.UseDeveloperExceptionPage();

Facts:

1. Better to use it when env is development
2. Must be registered among first components in pipeline

Example:

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

### 2.3.2 Error redirection – UseExceptionHandler

This components redirects current request on another address in case of error occuring

Configuration:

1. Register component with specifying redirection address – app.UseExceptionHandler("/error");

Facts:

1. Must be registered among first components in pipeline

Example:

app.UseExceptionHandler("/error");

app.Map("/error", ap => ap.Run(async context =>

{

await context.Response.WriteAsync("DivideByZeroException occured!");

}));

### 2.3.3 Status code processing – UseStatusCodePages

Component allows to process certain error status codes.

Several variants of configuration:

1. Show error in dependance on status code (message can be setup) - app.**UseStatusCodePages**("text/plain", "Error. Status code : {0}");
2. Don’t use it as it’s override status codes on 30x – app.**UseStatusCodePagesWithRedirects**("/error?code={0}");
3. Redirects on certain address + configures parameter – app.**UseStatusCodePagesWithReExecute**("/error", "?code={0}");

Example:

// обработка ошибок HTTP

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

app.Map("/error", ap => ap.Run(async context =>

{

await context.Response.WriteAsync($"Err: {context.Request.Query["code"]}");

}));

## 2.4 HTTPS components

HTTPS can be configured in project properties and can be enabled for Debug purposes: Project options -> Debug -> Enable SSL

### 2.4.1 HTTPS redirection – UseHttpsRedirection

Enable redirection from http to https

Configuration:

1. Register middleware component - app.**UseHttpsRedirection**();
2. (Optionally) register services to configure the redirection – services.**AddHttpsRedirection**

Example:

//register component

app.UseHttpsRedirection();

//register services

services.AddHttpsRedirection(options =>

{

options.RedirectStatusCode = StatusCodes.Status307TemporaryRedirect;

options.HttpsPort = 44344;

});

### 2.4.2 HTTP Strict Transport Security Protocol (HSTS) – UseHsts

Add special header which tells client that he must make requests via https (this is needed because even with httpsRedirection client can still make first request via http which can cause some sort of attacks)

Header example:

Strict-Transport-Security: max-age=63072000; includeSubDomains; preload

It has 3 params:

1. max-age – maximum time length (required param)
2. includeSubDomains – if it’s set then this header is applied to all sub-domains (optional param)
3. preload – if set then special preload-list of domains will be used (domains which are safety)

Strict-Transport-Security header sets global policy of working with domains/sub-domains

Component registration:

app.UseHsts();

Configuration in ConfigureSevices

    services.AddHsts(options =>

    {

        options.Preload = true;

        options.IncludeSubDomains = true;

        options.MaxAge = TimeSpan.FromDays(60);

        options.ExcludedHosts.Add("us.example.com");

        options.ExcludedHosts.Add("www.example.com");

    });

Facts:

1. Not recommended to use it during development (because of header caching and issues because of that).

## 2.5 Other components

1. app.UseFileServer() – just a combination of StaticFiles, DirectoryBrowser and DefaultFiles components

# 4 Configuration

Configuration values can be obtained from:

* env variables
* console params
* .Net memory objects
* various files (xml, json)
* Azure
* custom sources and providers

Configuration in ASP.Net is represented by **IConfiguration**:

public interface IConfiguration

{

string this[string key] { get; set; } // get setting value by key

IEnumerable<IConfigurationSection> GetChildren(); // get sub-sections of current section

IChangeToken GetReloadToken(); // get special token for observing changes in configuration

IConfigurationSection GetSection(string key); // get section of a setting key

}

Also there is another interface **IConfigurationRoot**:

public interface IConfigurationRoot : IConfiguration

{

IEnumerable<IConfigurationProvider> Providers { get; }

void Reload();

}

**ConfigurationBuilder** is used to build cusom configurations (often in Startup ctor).

var builder = new ConfigurationBuilder()

.AddInMemoryCollection(new Dictionary<string, string>

{

{"name", "Tom"},

{"age", "31"}

})

.AddConfiguration(config);

To include default configuration (injected into Startup.ctor) use AddConfiguration extension (also need to register IConfiguration again to override default config.

## 4.1 Default Configuration

There is already intance for **IConfiguration** which is provided by ASP DI

Instance provides following providers:

1. JsonConfigurationProvider - for appsettings.json
2. JsonConfigurationProvider - for appsettings.{envname}.json
3. EnvironmentVariablesConfigurationProvider - for env variables
4. CommandLineConfigurationProvider – for command line params

Facts:

1. All providers are used when setting are requested by IConfiguration
2. Last registered provider will be used for duplicated settigns between several providers
3. Default Configuration can be added to custom via AddConfiguration (then IConfiguration must be re-registered in ConfigureServices)

## 4.2 Configuration providers

**Command line params**

Command line parameters can be specified by following:

1. VS -> project properties -> Debug -> Application arguments
2. via cmd command from app folder: *dotnet run name=Tom age=35*

**Environment params**

By default ASP infrastructure loads all env variables into configuration but you can manually add them via ConfigurationBuilder.AddEnvironmentVariables()

**Memomry configuration**

MemoryConfigurationProvider is used to add config values from app code:

var configuration = new ConfigurationBuilder()

.AddInMemoryCollection(new Dictionary<string, string>

{

{"color", "blue"},

{"text", "Hello ASP.NET 5"}

}).Build();

**JSON configuration**

Adding json file is supported by **JsonConfigurationProvider** or configBuilder extension - **AddJsonFile()**

New json file e.g. ‘test.json’ can be added and included in configuration via AddJsonFile(‘test.json’)

Nesting:

{

  "color": "red",

  "namespace": { "class": { "method": "AddJson" } }

}

Nested setting can be accessed via ‘:’ e.g.:

string text = AppConfiguration["namespace:class:method"];

**XML configuration**

Achieved via **XmlConfigurationProvider**

**Ini configuration**

Achieved via **IniConfigurationProvider**

**Custom configuration provider**

There must be 3 completed steps:

1. Implement configuration source (inherit from **IConfigurationSourceb)**
2. Implement connfiguration provider (inherit from **ConfigurationProvider**)
3. Optionally, make extension method for configurationBuidler.

## 4.3 Configuraiton sections

Example configuration:

{

  "ConnectionStrings": {

    "DefaultConnection": "Main database",

      "UsersContext": "Users database"

  }

}

Ways of sections reading:

1. Get section as a separate IConfiguration object and work with it:

IConfigurationSection connStrings = AppConfiguration.GetSection("ConnectionStrings");

string defaultConnection = connStrings.GetSection("DefaultConnection").Value;

1. Get section value by full path:

string con = AppConfiguration.GetSection("ConnectionStrings:DefaultConnection").Value;

1. string con = AppConfiguration["ConnectionStrings:DefaultConnection"];
2. string con = AppConfiguration.GetConnectionString("DefaultConnection");

## 4.4 Custom configuration classes

Steps:

1. Create simple class (like dto, with public members)
2. Create instance of a class and use it in Bind method of IConfiguration

(or use Get<T>() method to obtain object with filled in properties)

Facts:

1. Case of configuration keys and object fields are ignored

Complex configuration classes are allowed as well:

{

  "age": 18,

  "name": "Tom",

  "languages": [

    "English",

    "German",

    "Spanish"

  ],

  "company": {

    "title": "Microsoft",

    "country": "USA"

  }

}

Class:

public class Person

{

    public string Name { get; set; }

    public int Age { get; set; }

    public List<string> Languages { get; set; }

    public Company Company { get; set; }

}

public class Company

{

    public string Title { get; set; }

    public string Country { get; set; }

}

**Binding certain config sections**

Need to get section and use the same methods (Bind or Get<T>), e.g.:

Company company = AppConfiguration.GetSection("company").Get<Company>();

**Other sources**

XML can be also binded to classes

## 4.5 IOptions (Preferable way)

IOptions – another way of working with configuration, it automatically bind certain class with certain settings.

Example:

public PersonMiddleware(RequestDelegate next, IOptions<Person> options)

        {

            \_next = next;

            Person = options.Value;

        }

Configuration:

1. Certain type for IOptions must be registered in ConfigureServices:

  services.Configure<Person>(AppConfiguration);

or bind to specific section:

services.Configure<Company>(AppConfiguration.GetSection("company"));

Facts:

1. Settings object is requested via Value member of Options class
2. IOptions can be injected wherever where other DI objects are used

# 5 HTTPContext

HTTPContext – object which contains info about current request and data for future response

## 5.1 Sessions

Session – collection which stored user or other data within different user requests until session period is expired or user stops sending requests (which also causes session expirting).

Certain session binds to certain user request via unique sessionId which is stored in user cookies.

Sessions are worked over **IDistributedCache**

Session can be accessed via context:

context.Session

Sessions setup:

1. Register services in ConfigureServices

services.AddDistributedMemoryCache();

services.AddSession();

Note: there can be completed some configuration (cookie name, cookie or session timeout, area of cookie usage and etc)

1. Register middlewares in Configure

app.UseSession()

Session actions:

* View keys: Session.Keys
* Get session value by key: Session.GetString(“name”)
* Set session data: Session.SetString("name", "Tom");

Facts:

* Default life time of the session is 20 minutes (can be overridden)
* User request resets session timeout
* Session object created per user and per browser
* Default cookie name: AspNet.Session
* Complex object can be saved to session as serialized string (e.g. json)

## 5.2 Useful members/methods

Fields:

1. Items – (IDictionary<object, object>) contains data for current request. This data lives within a request and can be updated within the period.

context.Items["text"] = "Text from HttpContext.Items";

# 6 Logging

ASP.Net provides **ILogger<T>** for logging

T – type of the class whose name will be used in log message.

Logger levels: Trace, Debug, Information, Error, Critical, Warning, None

Logger creation:

1. DI requesting (ASP.Net register Ilogger by default)
2. LoggerFactory.Create – here logger can be configured

var loggerFactory = LoggerFactory.Create(builder =>

{

builder.AddDebug();

});

Logger providers:

1. Console – writes to console
2. Debug – logs can be viewed in VS-> output
3. EventSource – writes to ETW (Event Tracing for Windows)
4. EventLog – writes to Windows Event Log

Configuration:

1. appsettings.json or another file

Facts:

* ASP.Net provides default implementation for ILogger (by default)
* Logger cannot be injected in ConfigureServices or Startup ctor

## 6.1 Custom Logger

Steps:

1. Create logger class from **ILogger**
2. Create custom logger provider from **ILoggerProvider**
3. Register custom provider in loggerFactory: factory.AddProvider(new FileLoggerProvider(filePath));