* **.net core**

<https://www.tutorialsteacher.com/core/dotnet-core>  
It is a cross-platform framework that runs on Windows, macOS, and Linux operating systems. .NET Core Framework can be used to build different types of applications such as mobile, desktop, web, cloud, IoT, machine learning, microservices, game, etc.

There are some limitations with the .NET Framework. For example, it only runs on the Windows platform. That is why .net core is introduced.

* **wwwroot:**

By default, the wwwroot folder in the ASP.NET Core project is treated as a web root folder. Static files can be stored in any folder under the web root and accessed with a relative path to that root.

* **Program.cs:**

ASP.NET Core web application is actually a console project which starts executing from the entry point public static void Main() in Program class where we can create a host for the web application.

**Kestrel** is an open-source, cross-platform web server for ASP.NET Core. It is designed to be used behind proxy because it has not yet matured to be exposed as a full-fledge web server.

* **Startup.cs: (**<https://www.tutorialsteacher.com/core/aspnet-core-startup>**)**

Startup class includes two public methods: ConfigureServices and Configure.

**ConfigureServices()**

The Dependency Injection pattern is used heavely in ASP.NET Core architecture. It includes built-in IoC container to provide dependent objects using constructors.

The ConfigureServices method is a place where you can register your dependent classes with the built-in IoC container. After registering dependent class, it can be used anywhere in the application. You just need to include it in the parameter of the constructor of a class where you want to use it. The IoC container will inject it automatically.

ASP.NET Core refers dependent class as a Service. So, whenever you read "Service" then understand it as a class which is going to be used in some other class.

ConfigureServices method includes IServiceCollection parameter to register services to the IoC container.

**Configure()**

The Configure method is a place where you can configure application request pipeline for your application using IApplicationBuilder instance that is provided by the built-in IoC container.

ASP.NET Core introduced the middleware components to define a request pipeline, which will be executed on every request. You include only those middleware components which are required by your application and thus increase the performance of your application.

* **Dependency Injection:**<https://www.tutorialsteacher.com/core/dependency-injection-in-aspnet-core>

DI is a way to registers dependency services so that It can be used anywhere in the application.

ASP.NET Core injects objects of dependency classes through constructor or method by using built-in IoC container.

IoC Container (a.k.a. DI Container) is a framework for implementing automatic dependency injection. It manages object creation and it's life-time, and also injects dependencies to the class.

**Service lifetime:**

<https://stackoverflow.com/questions/38138100/addtransient-addscoped-and-addsingleton-services-differences>

* **Request Processing:**<https://www.tutorialsteacher.com/core/aspnet-core-middleware>
* **Env variable:**

ASP.NET Core uses an environment variable called ASPNETCORE\_ENVIRONMENT to indicate the runtime environment. The value of this variable can be anything as per your need but typically it can be Development, Staging, or Production. The value is case insensitive in Windows and Mac OS but it is case sensitive on Linux.

The IHostingEnvironment service includes EnvironmentName property which contains the value of ASPNETCORE\_ENVIRONMENT variable. ASP.NET Core also includes extension methods to check the environment such as IsDevelopment(), IsStating(), IsEnvironment() and IsProduction().

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsEnvironment("Development"))

{

// code to be executed in development environment

}

if (env.IsDevelopment())

{

// code to be executed in development environment

}

if (env.IsProduction())

{

// code to be executed in production environment

}

}

* **Exception handling in Startup.cs:**

<https://www.tutorialsteacher.com/core/aspnet-core-exception-handling>

* <https://www.tutorialsteacher.com/core/aspnet-core-static-file>

ASP.NET Core application cannot serve static files by default. We must include Microsoft.AspNetCore.StaticFiles middleware in the request pipeline.

The app.UseStaticFiles() method adds StaticFiles middleware into the request pipeline. The UseStaticFiles is an extension method included in the StaticFiles middleware so that we can easily configure it.

Order of middleware is very important. app.UseDefaultFiles() should be added before app.UseStaticFiles() in the request pipeline.

* **aspnet-core-logging :**

<https://www.tutorialsteacher.com/core/aspnet-core-logging>

**MVC**

* MVC stands for Model, View, and Controller. MVC separates an application into three components - Model, View, and Controller.

<https://www.tutorialsteacher.com/mvc/mvc-architecture>

* Routing:

<https://dotnettutorials.net/lesson/routing-asp-net-core-mvc/>

<https://dotnettutorials.net/lesson/custom-routing-inasp-net-core-mvc/>

<https://dotnettutorials.net/lesson/attribute-routing-in-asp-net-core-mvc/>

Routing is the process of directing an HTTP request to a controller.

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller=Home}/{action=Index}/{id?}");

});

namespace FirstAppDemo.Controllers {

[Route("[controller]")]

public class AboutController {

[Route ("")]

public string Phone() {

return "+49-333-3333333";

}

[Route("[action]")]

public string Country() {

return "Germany";

}

}

}

**Action result:** The ActionResult class is the base for all the action results. Action Result is actually a data type, This return type has many other derived types.

You might be wondering what is the advantage of using something that produces an ActionResult.

The typical advantage is that it is just a formal way to encapsulate the decision of the controller.

The controller decides what to do next, either return a string or HTML or return a model object that might be serialized into JSON etc.

All that the controller needs to do is make that decision and the controller does not have to write directly into the response the results of its decision.

It just needs to return the decision and then it is the framework that will take a result and understand how to transform that result into something that can be sent back over HTTP.

1. All the public methods in the Controller class are called Action methods.
2. The Action method has the following restrictions.  
       - Action method must be public. It cannot be private or protected.  
       - Action method cannot be overloaded.  
       - Action method cannot be a static method.
3. ActionResult is a base class of all the result type which returns from Action method.
4. The base Controller class contains methods that returns appropriate result type e.g. View(), Content(), File(), JavaScript() etc.
5. The Action method can include [Nullable](https://www.tutorialsteacher.com/csharp/csharp-nullable-types) type parameters.

**Controller:**

The Controller in MVC architecture handles any incoming URL request. The Controller is a class, derived from the base class System.Web.Mvc.Controller. Controller class contains public methods called Action methods. Controller and its action method handles incoming browser requests, retrieves necessary model data and returns appropriate responses.

In ASP.NET MVC, every controller class name must end with a word "Controller". For example, the home page controller name must be HomeController, and for the student page, it must be the StudentController. Also, every controller class must be located in the Controller folder of the MVC folder structure.

Points to Remember :

1. The Controller handles incoming URL requests. MVC routing sends requests to the appropriate controller and action method based on URL and configured Routes.
2. All the public methods in the Controller class are called Action methods.
3. The Controller class must be derived from System.Web.Mvc.Controller class.
4. The Controller class name must end with "Controller".
5. A new controller can be created using different scaffolding templates. You can create a custom scaffolding template also.

**Action Method:**

MVC 5 includes the following action selector attributes:

* ActionName
* NonAction
* ActionVerbs

NonAction attribute:

Use the NonAction attribute when you want public method in a controller but do not want to treat it as an action method.

ActionName attribute:

The ActionName attribute allows us to specify a different action name than the method name, as shown below.

ActionVerbs:

The ActionVerbs selector is to handle different type of Http requests. The MVC framework includes HttpGet, HttpPost, HttpPut, HttpDelete, HttpOptions, and HttpPatch action verbs.

You can apply one or more action verbs to an action method to handle different HTTP requests. If you don't apply any action verbs to an action method, then it will handle HttpGet request by default.

Multiple action verbs: [AcceptVerbs("Get", "Post")]

**Layout view:**

An application may contain a specific UI portion that remains the same throughout the application, such as header, left navigation bar, right bar, or footer section. ASP.NET MVC introduced a Layout view which contains these common UI portions so that we don't have to write the same code in every page.

Layout views are shared with multiple views, so it must be stored in the Shared folder. By default, a layout view \_Layout.cshtml is created when you Create MVC application using Visual Studio.  Prefixing the underscore \_ before layout view name is a common naming convention

@{

ViewBag.Title = "Home Page";

Layout = "~/Views/Shared/\_myLayoutPage.cshtml";

}

**Partial View:**

A partial view is a reusable portion of a web page. It is .cshtml or .vbhtml file that contains HTML code. It can be used in one or more Views or Layout Views. You can use the same partial view at multiple places and eliminates the redundant code.

<**partial** **name**="~/Views/Shared/\_PartialHeader.cshtml" />

**ViewBag:**

The ViewBag in ASP.NET MVC is used to transfer temporary data (which is not included in the model) from the controller to the view.

Internally, it is a dynamic type property of the ControllerBase class which is the base class of the Controller class.

ViewBag only transfers data from controller to view, not visa-versa. ViewBag values will be null if redirection occurs. Internally, ViewBag is a wrapper around ViewData. It will throw a runtime exception, if the ViewBag property name matches with the key of ViewData.

ViewBag Limitations:

* ViewBag doesn't require typecasting while retrieving values from it. This can throw a run-time exception if the wrong method is used on the value.
* ViewBag is a dynamic type and skips compile-time checking. So, ViewBag property names must match in controller and view while writing it manually.
* **<h1>**@ViewBag.Header**</h1>**
* @{
* var student = ViewBag.Student;
* }

**ViewData:**

ViewData is similar to ViewBag, which transfers data from Controller to View. ViewData is of Dictionary type, whereas ViewBag is of dynamic type. However, both store data in the same dictionary internally.

ViewData only transfers data from controller to view, not vice-versa. It is valid only during the current request.

<ul>

@foreach (var std in ViewData["students"] as IList<Student>)

{

<li>

@std.StudentName

</li>

}

</ul>

Points to Remember :

* ViewData transfers data from the Controller to View, not vice-versa.
* ViewData is a dictionary type.
* ViewData's life only lasts during the current HTTP request. ViewData values will be cleared if redirection occurs.
* ViewData value must be typecast to an appropriate type before using it.
* ViewBag internally inserts data into ViewData dictionary. So the key of ViewData and property of ViewBag must NOT match.

**TempData:**

<https://www.tutorialsteacher.com/mvc/tempdata-in-asp.net-mvc>

TempData is used to transfer data from view to controller, controller to view, or from one action method to another action method of the same or a different controller.

TempData stores the data temporarily and automatically removes it after retrieving a value.

TempData is a property in the ControllerBase class. So, it is available in any controller or view in the ASP.NET MVC application.

* <https://dotnettutorials.net/lesson/strongly-typed-view-asp-net-core-mvc/>
* <https://dotnettutorials.net/lesson/view-model-asp-net-core-mvc/>
* **Filters:**

<https://www.tutorialsteacher.com/mvc/filters-in-asp.net-mvc>

<https://www.tutorialsteacher.com/articles/create-custom-filters>

<https://www.tutorialsteacher.com/mvc/action-filters-in-mvc>

* **Area:**

<https://www.tutorialsteacher.com/mvc/area-in-asp.net-mvc>

* **Sections:**

<https://dotnettutorials.net/lesson/sections-in-layout-view-in-asp-net-core-mvc/>

* **ViewStart:**

<https://dotnettutorials.net/lesson/viewstart-in-asp-net-core-mvc/>

* **View import:**

<https://dotnettutorials.net/lesson/viewimports-in-asp-net-core-mvc/>

* **Tag helpers:**

<https://dotnettutorials.net/lesson/tag-helpers-in-asp-net-core-mvc/>

* **Image tag helper: (asp-append-version)**

<https://dotnettutorials.net/lesson/image-tag-helper-asp-net-core/>

* **HTTP stands for HyperText Transport Protocol.**

Media type (aka MIME type) specifies the format of the data as type/subtype e.g. text/html, text/xml, application/json, image/jpeg etc. In HTTP request, MIME type is specified in the request header using Accept and Content-Type attribute.