Asp.Net

**What is ASP.NET?**

.NET is a developer platform made up of tools, programming languages, and libraries for building many different types of applications.

* ASP.NET extends the [.NET developer platform](https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet) with tools and libraries specifically for building web apps.

**Uses of Asp.net**:

* ASP.NET is a free web framework for building great websites and web applications using HTML, CSS, and JavaScript. You can also create Web APIs and use real-time technologies like Web Sockets.
* To build the apps using MVC architectural pattern.
* Asp.net offers three frameworks for creating website applications i.e., web forms, mvc, web pages.

**Web APIs**

* ASP.NET Web API is a framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers and mobile devices. ASP.NET Web API is an ideal platform for building RESTful applications on the .NET Framework.

**Uses of Web API**

* Web API is a programming interface/application type that provides communication or interaction between software applications. Web API is often used to provide an interface for websites and client applications to have data access. Web APIs can be used to access data from a database and save data back to the database.
* **MVC:**
* The MVC is an application development pattern or design pattern which separates an application into three main components. Or
* MVC is an architectural pattern which divides an app into 3 logical units: Model, view, controller.
* All user interactions and requests are routed to the controller.
* The controller works with the model for fetching /updating data from the datastore.
* The controller decides which view to display to the user. The data required by the view is provided by the controller after interaction with the model.
* 1.Model
* 2.View
* 3.Controller
* It is popular to design applications.
* Mvc design pattern helps to enforce separation of concerns to help you avoid making presentation logic, data access logic together.

**Model**: manages the behaviour and data where we stored. It is usually a collection of the basic objects or classes used in application.-----or-----

The Model component corresponds to all the data-related logic that the user works with. It can represent either the data that is being transferred between the view and controller components. It can add or retrieve data from the database. It responds to the controller’s request because the controller can’t interact with the database by itself. The model interacts with the database and gives the required data back to the controller.

* Data related.
* Consists of classes/objects with properties.
* Uses SQL statements.
* Supplies the controller with list of objects.
* **View:**
* It manages the display of data. It gives presentation to the user like css html from webpages.-----or----
* A view interacts with users by providing Ui elements to accept and display data.
* MVC uses razor as the view engine.
* Razor combines html with c# to provide presentation elements with dynamic programming to manipulate the same.
* The C# code is separated from the HTML using @symbol.
* Since C# is the programming language, view pages will have an extension of .cshtml.
* **Controller:**
* It handles page events and navigation between pages. Controller is the component which handles user interaction. It works with the model and selects the view to render the web page. In an MVC application, the view only displays information whereas the controller handles and responds to the user input and requests.
* **-**-----------------(or)-------------------
* A controller is a class which inherits from Microsoft . Asp. Net Core . MVC. Controller class.
* It contains public methods called action methods. These methods are used to process any action performed by the user i.e., they handle the user requests.
* Action methods most commonly return a sub type of the IActionResult Interface.
* The IActionResult interfaces represents the result of an action method.
* An action can result in displaying a view, downloading a file, returning Json data and so on.
* Helpers are methods used to return instances of the action result child classes. Developers can invoke these methods instead of creating instances.

**HTTP Methods:**

**The MVC framework includes httpget, post,put,delete,options patch.we can apply one or more action verbs to an action method to handle different http request.if we don’t apply any action verbs to an action method, then it will handle httpget requests by default.**

The following HTTP methods are most commonly used in a REST based architecture.

1. GET − Provides a read only access to a resource.(or) To retrieve the information from the server parameters will be appended in the querystring.
2. PUT − Used to update a new resource.
3. DELETE − Used to remove a resource.
4. POST − Used to update an existing resource or create a new resource.

or

Crud operations:

C:create

R:read

U:Update

D:delete.

* **Layout Views:**
* This is the default master page of the application. It contains the common layout like header, footer, logos and scripts which may be shared across all view pages in an app.
* The default layout is named\_Layout.cshtml.
* Location-: views/Shared/\_Layout.cshtml.
* Common HTML structures such as scripts and stylesheets are also frequently used by many pages within the app.
* Layouts reduce duplicate design and code in views.
* **Layout Page HTML structure:**
* The layout page for the app is specified using the layout property, in the \_viewstart.cshtmlfile.
* If you want to use different layout pages on razor pages, you can use the same syntax at the page level and define the new master page individually.
* @{
* Layout = ”\_Layout”;
* }
* **Partial Views:**
* A partial view is a view that is rendered within over views.
* A partial view cannot run in the browser independently. It is used as a part of other views.
* Need for partial views:
* Break up views into smaller, reusable \_components.
* Reduce the repetition of common content/scripts across views.
* Partial views are not to be used to maintain common layout elements. This must be done in the layout file.
* Partial view names begin with an underscore (\_) as a naming convention.
* **View Files Summary:**
* \_Layout.cshtml: The default layout view.
* \_validationScriptpartial.cshtml: partial view with scripts used for client-side validations.
* Error.cshtml: The default error page used by developers to handle exceptions.
* \_viewImports.cshtml: common namespaces for import.
* \_viewstart.cshtml: specify the layout file an application level.
* **HTML Helpers:**
* Html helpers are methods used in Razor views to generate html content.
* Consider a Html markup to generate a hyperlink.
* <a href=”/home/about”>about US</a>
* The above markup will be generated by the following helper:
* Html.ActionLink{“About Us”,”About”}
* The main usage of Html helpers over plain html is to perform model binding and pass route parameters easily
* @Html.TextBoxFor(model=>model.ProceductName) will generate an input element whose value is bound with the productName property of the model class used.
* **Tag Helpers:**
* Html helpers are methods used in razor views to generate a html content, whereas tag helpers are attribute which attach themselves to HTML elements.
* Using tag helpers generates cleaner markup as developers need not intersperse c# with Html.
* Consider a html markup to generates a hyperlink:
* <a href=”/home/about”>about us</a>
* The above markup will be generated by the following helper:

<a asp-controller=”home” asp\_action=”about”>about Us</a>

* **Commands:**
* Asp-controller: used to specify the controller’s name.
* Asp-action: used to specify the action name.
* Asp-route (value): specifies the route parameter in the value placeholder.
* Asp-route: specifies the name of the route.
* Asp-for: specifies the property to bind with the UI element.
* **Program.Cs Files:**
* This file is responsible for running the application.
* Var builder=web.application.createBuilder(args)-initializes anew instance of the webapplicationBuilder class with preconfigured defaults.
* Var app=builder.Build() method configures a host with a set of default options such as internally it configures kestrel, iis integration, content root directory and other configuration.
* The app.Run() method runs an application and block the calling thread until the host shutdown.
* A middle ware component is added to the pipeline by invoking a USE (feature) extension method.

**StartUp Class:**

* It is described by its name: startup. It is the entry point of the application. It configures the request pipeline which handles all requests made to the application.
* The inception of startup class is in owin(open web interface for .net) application that is specification to reduce dependency of application on server.
* Startup class is mandatory in asp.net core application. It can have any access modifier (public, private, internal). This is the entry point of the asp.net application. It contains application configuration related items.
* The asp.net core application is a console app, and we have to configure a web host to start listening. The program class does this configuration.

1. **public** **class** Program
2. {
3. **public** **static** **void** Main(string[] args)
4. {
5. **var** host = **new** WebHostBuilder()
6. .UseKestrel()
7. .UseContentRoot(Directory.GetCurrentDirectory())
8. .UseStartup<Startup>()
9. .Build();
10. host.Run();
11. }
12. }

The web host is created in the main method of the Program class and here we have configuration of startup class using method "UseStartup". So, it is not necessary that class name is "Startup".

1. **public** **class** Startup
2. {
3. **public** **void** Configure(IApplicationBuilder app)
4. {

We can also specify constructor of the startup class. The startup class has constructor with one or three parameters.

1. **public** Startup(IHostingEnvironment env)
2. {
4. }
6. **public** Startup(IApplicationBuilder appenv, IHostingEnvironment env,  ILoggerFactory loggerFactory)
7. {
9. }

*IApplicationBuilder* is an interface that contains properties and methods related to current environment. It is used to get the environment variables in application.

*IHostingEnvironment* is an interface that contains information related to the web hosting environment on which application is running. Using this interface method, we can change behaviour of application.

*IloggerFactory*is an interface that provides configuration for the logging system in Asp.net Core. It also creates the instance of logging system.

The startup class contains two methods: Configure Services and Configure.

**ConfigureServices** **Method**

Declaration of this method is not mandatory in startup class. This method is used to configure services that are used by the application. When the application is requested for the first time, it calls Configure Services method. This method must be declared with a public access modifier, so that environment will be able to read the content from metadata.

ASP.net core has built-in support for Dependency Injection. We can add services to DI container using this method. Following are ways to define ConfigureServices method in startup class.

1. **public** **void** ConfigureServices(IServiceCollection services)
2. {
3. services.AddMvc();
4. }

In case of specific class or library of project that would like to add to DI container, we will use the IserviceCollection. In the above example, I have just add MVC service to the ConfigureServices method.

**Configure Method**

This method is used to define how the application will respond on each HTTP request i.e. we can control the ASP.net pipeline. This method is also used to configure middleware in HTTP pipeline. This method accept IApplicationBuilder as a parameter. This method may accept some optional parameter such as IHostingEnvironment and ILoggerFactory. Whenever any service is added to ConfigureServices method, it is available to use in this method.

1. **public** **void** Configure(IApplicationBuilder app)
2. {
3. app.UseMvc();
5. app.Run(context => {
6. **return** context.Response.WriteAsync("Hello Readers!");
7. });
8. }
9. }
10. **public** **void** ConfigureServices(IServiceCollection services)
11. {
13. }
14. }

**Summary**

The Startup class is mandatory, and it is the entry point of the application. With the help of this class, we can configure the environment in our ASP.net Core application. We can use Constructor and two different methods: ConfigureServices and Configure for setting up the environment. This class creates services and injects services as dependencies so the rest of the application can use these dependencies. The ConfigureServices used to register the service and Configure method allow us to add middleware and services to the HTTP pipeline. This is the reason ConfigureServices method calls before Configure method.

* **Routing:**
* Routing refers to matching the urls of incoming requests to controller actions.
* Asp.net core uses the routing middleware to perform routing.
* Routes are typically defined at stratup in program.cs or using attributes on controller actions.
* Route specification in the program.cs file.
* App.UseRouting(); // configure routing using middleware
* App.UseAuthorization();
* App.MapControllerRoute(
* Name:”default”,
* pattern:”(controller=Home)/(action=index}/{id?”);// route pattern, first controller name with default value of home, followed by action name with default value of index.
* App.run();

**Attribute based routes**

Attribute routing applied at action level.

Public class Mycontroller: controller

{

[route(“home”)]

[route(“home/Authenticate”]}

Public IActionResult Login(string user,string password){}

}

The above example shows 2 route patterns mapped to the login action.

* **Validation:**
* Validating user input in razor pages:
* Razor pages includes a robust validation framework that validate the inbound model properties on the client-side and on the server.
* The key components of the input validation framework are:
* 1. Data annotation attributes
* 2.Tag helpers.
* 3.Jquery Unobtrusive validation
* 4.Model state
* 5.Route Constraints.
* Data Annotation Attributes: The primary building blovks of the validation framework is a set of attributes that inherit from validationAttribute.

Validating user input in Razor pages:

Attribute Description

Compare Used to specify another form field that the value should be compared to for equality.

MaxLength Sets the maximum number of character /bytes/items/ that can be accepted.

MinLength Sets the minimum number of characters/bytes/items that can be accepted.

pageRemote Enables client-side validation against a server-side resource, such as a database check to see if a username is already in use.

Range sets the minimum and maximum value of arrange.

Regular Expression checks the value against the specified regular expression.

Remote Enables client-side validation against a server-side resource, such as database check to see if a username is already in use.

Required Specifies that a value must be provided for this property.Note that non-nullable values types such as date time and numeric values are treated as required by default and do not need this attribute applied to them.

StringLength sets the maximum and optionally the minimum number of string characters allowed.

* **Client Side Validation:**
* Client-side validation support is provided by the jQuery unobtrusive library, developed by Microsoft.
* Client-side validation will not take place unless\_validationscriptspartial.cshtml is included in your form.
* Client-side validation works with special html5 data attributes emitted by tag helpers. To see how that works, here is a simple tag helper-based form featuring the properties above.
* **Server side Validation:**
* Because it is so easy to circumvent client-side validation,server-side validation is included as part of the asp.net core validation frameworks.
* Once property values have been bound, the framework looks for all validation attributes on those properties and executes them.
* Any failure results in an entry being added to a modelstate dictionary.
* This is made available via Modelstate which has a property named is valid that returns false if any of the validation tests fail.
* **State Management:**

**View Data:**

* Viewadata Is a container for data to be passed from the controller to the viewpage.
* View data is a dictionary of objects with a string-based key.
* The view data dictionary is automatically made available to the view. therefore, in order to reference values stored in it, you just refer to their item by key.
* Unboxing is necessary to retrieve data from the viewdata in its original form.
* **ViewBag:**
* Viewbag like viewdata is a container for data to be passed from the controller to the view page.
* Viewdata is a dynamic property.
* Unboxing is not necessary to retrieve data from the viewbag in its original form since it is dynamic in type.
* Both viewdata and viewbag are available only for the current request.
* **TempData:**
* Tempdata is a container for data to be passed from the controller to the viewpage.
* Like view data it is a dictionary of objects with a string-based key.
* Unboxing is necessary to retrieve data from the viewdata in its original form
* Tempdata can be persisted for the current request and the subsequent request unlike viewbag and viewdata which are scoped to the current request.
* Security:

Authentication:

Authentication is the process by which an app checks the identity of the user.

* The most common way to do this is by asking the user to enter credentials like username and password.
* Developers can use the asp.net core identity API to manage users, passwords, roles, claims, tokens, and other data.
* Users can create an account with the login information stored in the application identity provider database or they can use external login providers.
* Some external login providers include Facebook, google, Microsoft account and twitter.

**Authorization:**

* Once we determine who a user is the next step is to check what rights the user has
* AN admin user might have rights to create, edit, delete documents from a portal.
* A non admin user might have only read access.
* Roles can be used to group users with similar privileges. Authorization checks the role of the user to determine the level of access.
* If roles have been created for users in the app, the authorize attribute can include the same
* **Middleware :**
* A middleware is nothing but a component (class) which is executed on every request in asp.net core application.
* HttpHandlers and httpmodules were part of request pipeline.
* Middleware is similar to httphandlers and httpmodules where both needs to be configured and executed in each request.
* We can set the order of middleware execution in the request pipeline.Each middleware adds or modifies http request and optionally passes control to the next middleware component.
* Middlewares build the request pipeline.
* Configure Middleware:
* We can configure middleware in the configure method of the startup class using IApplicationBuilder instance.
* Run() is an extension method on IApplicationBuilder instance which adds a terminal middleware to the application’s request pipeline.
* Run() is an extension method on IApplicationBuilder and accepts a parameter of request Delegate.
* The requestDelegate is a delegate method which handles the request.
* App.Use() may call next middleware component in the pipeline.

**Dependency Injection:**

Dependency Injection is a software design pattern that specifies how components get holds of their dependencies. In this pattern, components are given their dependencies instead of coding them within the component.