.NET core is free open source and cross platform framework and created for building cloud based applications.

There are no boundaries for the platform.

2000 -> launched first .NET framework (.net 1.0)

Plays a major role in field of software development.

People love Microsoft products as they are easy to use and easy to learn.

2016 -> Microsoft came with .NET core 1.0

People always love ASP .NET because it is working over WWW.

But now in ASP .net core it is open source and cross platform

**Open source:**

Whose source code is made available for use or modification to other developers.

Developed as a public collaboration and made freely available

Example : Windows is not available freely while LINUX is available without any restrictions

The software being redistributed must be distributed to anyone else without any restrictions

**Features of ASP .NET**

Cross platform

Open source

Runs our app over Linux, Windows, Mac ( wherever you want)

Fast development – fast work over browser

Work in your editors ( not only visual studio but also visual studio code)

Based on MVC architecture

What is ASP .NET core?

New version of ASP .NET

Developed by Microsoft

Open source

Cross platform

Completely rewritten from scratch ( it was initially launched as ASP .Net 5 and renamed to ASP .NET Core 1.0)

Cloud based and internet connected applications

Contains modular components(every task divided into tasks) with minimal overhead, so you there is flexibility while constructing solutions.

Before ASP .net, Microsoft already had ASP .net Web forms and ASP .net MVC.

Latest version now : ASP .net Core 7.0 ( launched in feb 24 2023)

Asp .net core 6 is based on .net 6

From .NET 5.0, the word core is dropped from its name. (.net 6.0 onwards it is just ASP.net)

**Difference between .Net and .Net core framework**

What is .net framework?

Framework created by Microsoft that developers can use to create applications more easily. Framework is a bunch of code that programmer can call without having to write it explicitly

Basically a collection of libraries

There are pre-coded solutions that manage the execution of programs written specifically for the framework

|  |  |
| --- | --- |
| **.Net framework** | **.Net Core framework** |
| Old framework | New framework |
| Not open source | Open source |
| Only for windows | Cross platform (windows, linux, mac) |
| Application developed is dependent on the .net framework installed on windows | No dependency on the .net in the system |
| Doesn’t support dependency injection(have to implement from scratch) | Supports dependency injection |
| Have WinForms, ASP .net, WPF for developing both desktop and web apps. | Doesn’t support desktop application. Focusing on web, windows mobile, windows store and windows universal apps(that can run on any system |
| A single packaged installation and runtime environment for windows( the libraries that don’t use are also installed) | As it is cross-platform, so installed independently for each platform |
| No support for microservices | Has support microservices |
| Has support for WCF services, and implements and works with RESTful services | No support for WCF services. And have ro create a REST API. |
| Less scalable and provides low performance compared to .Net core | Provides high scalability and performance |
| Mobile development is not supported | Has support for mobile apps. It is compatible with Xamarin and other open source platforms for mobile applications. |
| Too heavy for CLI | Provides a light weight CLI |
| Libraries of framework are packed and shipped together. | .net core is shipped as collection of Nuget Packages |

**Is .net core or .net framework better?**

Depends on the project requirement and what the project demands

Prefer .net core if :

1. Demands cross platform integration
2. Requires the development of microservices
3. Depends heavily on CLI as .net core is suitable for CLI

Prefer .net framework if

1. When applications are already running on .net framework
2. When applications require technologies like workflow, web forms or WCF that are not available in .net core
3. When applications are built to run on Windows alone.

**MVC with ASP .net core**

Three basic components of MVC:

1. Model -> business entities
2. View -> Presentation logic
3. Controller -> Business logic

Backbone of MVC: Controller

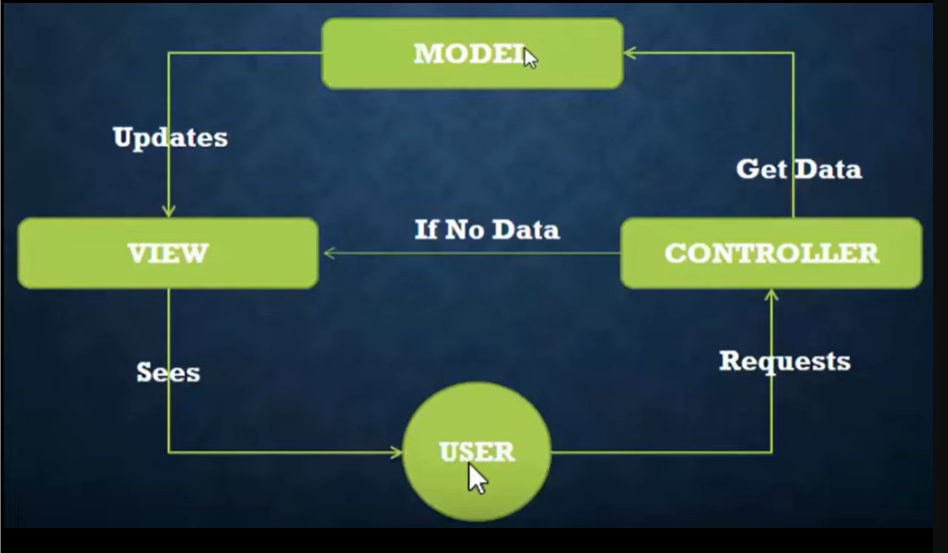
Separation of concern( work is done in different module)

It is architectural design pattern

MVC is not a programming language, not a framework, it is a design pattern.

Advantages of Separation of concern:

1. Allows work on individual pieces of the system in isolation
2. Facilitates reusability
3. Ensures the maintainability of a system
4. Ensures Extensibility
5. Enables user to understand the system better



A diagram of a workflow

Description automatically generated

**Directory Structure**

**Root**

**│**

**├── appsettings.json**

**├── Program.cs**

**├── MySolution.sln**

**│**

**├── Controllers**

**│ └── HomeController.cs**

**│**

**├── Models**

**│ └── MyModel.cs**

**│**

**├── Views**

**│ ├── Home**

**│ │ └── Index.cshtml**

**│ └── Shared**

**│ └── \_Layout.cshtml**

**│**

**├── Areas**

**│ └── Identity**

**│ └── Pages**

**│ ├── Account**

**│ │ ├── Login.cshtml**

**│ │ └── Register.cshtml**

**│ └── \_ViewStart.cshtml**

**│**

**├── Data**

**│ ├── ApplicationDbContext.cs**

**│ └── Migrations**

**│ └── [MigrationFiles].cs**

**│**

**└── wwwroot**

**├── css**

**│ └── site.css**

**└── js**

**└── site.js**

launchSettings.json

* In the properties folder in ASP .net core
* Contains settings that control how our web app is started on our development machine
* Settings in this file are going to be used when we run the .NET core application either from visual studio or by .net core cLI
* Only used in the local development machine
* This is not required when we publish our asp .net core application to production server

Wwwroot

* Contains static files (like CSS, JavaScript, images) that are served directly to clients. These files are publicly accessible.

Appsettings.json

* It provides a convenient way to externalize configuration from your code, allowing you to change settings without modifying the application's source code.
* Contains database connection strings used by the application.

Program.cs

* Entry point of application
* Contains the Main method
* Here we configure the web host for your ASP .net core application
* WebApplication.CreateBuilder(args): This static method creates a WebApplication builder instance, which is used to configure the application.
* builder.Services: This property is used to configure services (dependencies) that the application will use. In this example, AddControllersWithViews() configures MVC services.
* builder.Build(): This method builds the WebApplication instance, finalizing the configuration.
* app.Environment: This property provides access to the application's environment, which allows you to determine whether the application is running in development, staging, or production.
* Also starts the middle wares here before running the application

using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.Hosting;

var builder = WebApplication.CreateBuilder(args);

// Configure services and the app's request pipeline

builder.Services.AddControllersWithViews();

var app = builder.Build(); // returns instance of Web application

// Configure the HTTP request pipeline

if (app.Environment.IsDevelopment()) // checks if hosting env is develpment

{

app.UseDeveloperExceptionPage(); //adds middleware to pipeline

}

else

{

app.UseExceptionHandler("/Home/Error"); //to handle exceptions

app.UseHsts();

}

app.UseHttpsRedirection(); // automatically redirects HTTP requests to HTTPS.

app.UseStaticFiles(); //adds static files from wwwroot

app.UseRouting(); //enables endpoint routing

app.UseAuthorization(); //enables authorisation

app.MapControllerRoute(

name: "default",

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

app.Run();

* Considered as middleware

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(

name: "default",

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

**MIDDLEWARES IN ASP .net**

Generally client sends a request to Web application and and the application send sthe response to the client.

The request is processed by the application and the response is sent back to the client.

Between the client and the app there is a concept called the HTTP pipeline.

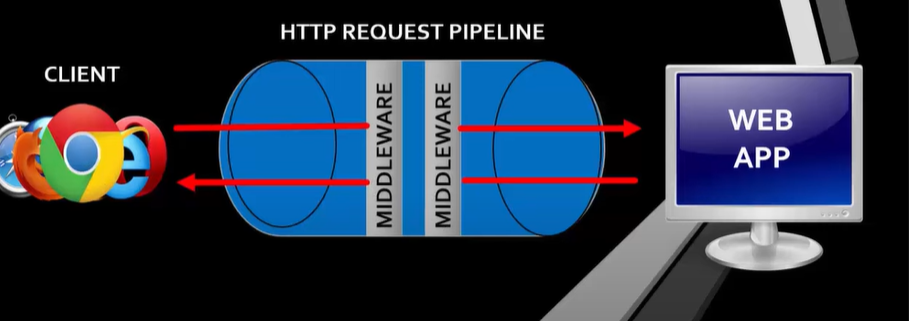
The request send by the client or the response from the application goes from this HTTP pipeline itself.

In this http pipeline we define the middleware

In Program.cs we define the middlewares

They are simple functions or logics written in middlewares

Example : client requests a page that needs authorisation, then the check of authorisation can be defined in the middle ware. Middle ware checks if the request asked by a particular user is authorised user or not. If he is a authorised user then the resource is accessed otherwise he gets an error as response



Middleware is nothing but a class which is executed on every request

It controls how our application responds to HTTP request

They are software components that are assembled to application pipeline to handle requests and responses

Also controls how the application looks when there is an error(what page should be displayed to the user)

Authentication and authorisation to perform specific actions

Each piece of middle ware is an object and each piece has a very specific, focused and limited role

Custom middleware

app.Run(async (context) =>

{

await context.Response.WriteAsync("Welcome to MyAPP");

});

Order of middleware is important.

If we use app.Run to define middleware, the subsequent middleware will not be executed.

So if we app.Use then we can execute the other middleware

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

{

await context.Response.WriteAsync("Welcome to MyAPP\n");

await next(context); // pass the context to next middleware

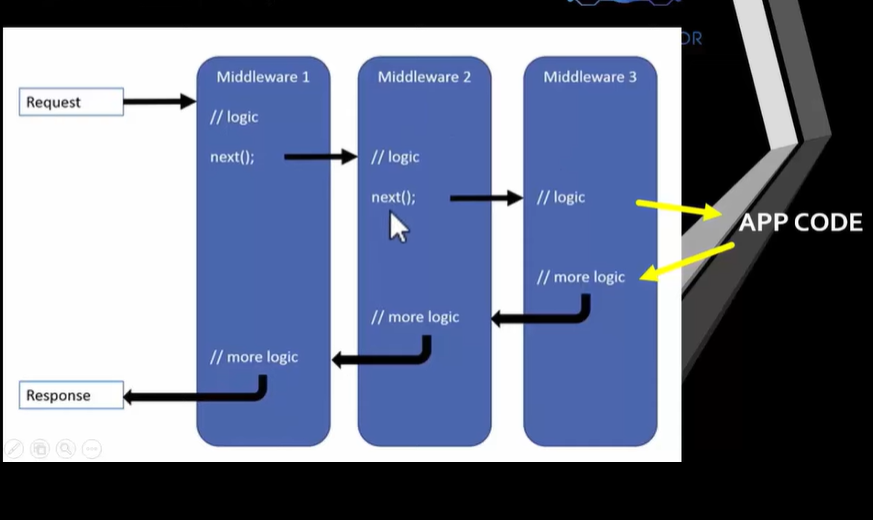
});

app.Run(async (context) =>

{

await context.Response.WriteAsync("By Palla Nikhita");

});

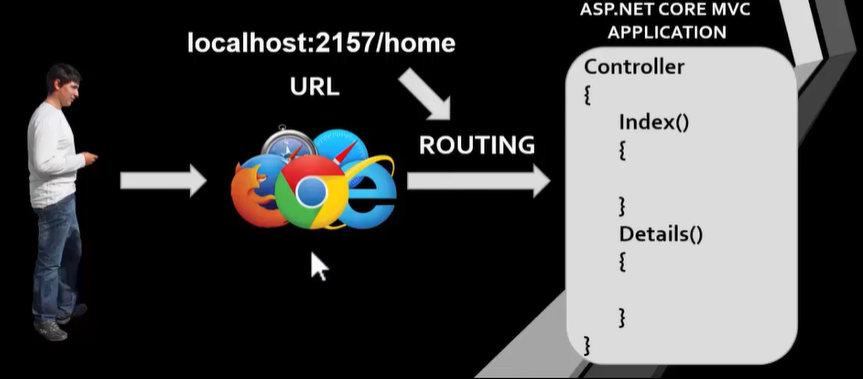


Middlewares defined using app.Run will never call subsequent middleware

To execute middleware in the pipleline and allows that middleware to to pass control to the next item in the pipeline

**ROUTING**

A mechanism in which it will inspect the incoming requests(URL) and then map that request to the controllers and their action methods



A computer screen with a diagram

Description automatically generated

Routing : URL + HTTP Methods(Get/post/put/delete)

Mapping : connecting the url to controller and action method

We can do this by adding the routing middleware to the request processing pipeline

After we configure the incoming requests i.e., URLs to the controllers action methods based on the routes configured in your application

Routes can be defined in two ways:

1. Convention based routing
2. Attribute based routing

**Convention based routing**

app.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}"); // id is optional so given with ?

To build routing from scratch

1. Add controllers and views to builder

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

var app = builder.Build();

1. It will call Home/Index by default

app.MapDefaultControllerRoute();

1. We can also define route ourselves

app.MapControllerRoute(

name: "default",

pattern: "{controller=Main}/{action=Index}/{id?}"

);

If we wont define the convention in Program.cs then the URL cant produce results as it cant identify the controller and action method.

Actual work is done by action method in controller.

Example : /home/details/2 -> then the logic is executed from the action method called details.

**Attribute Based routing**

[Route] attribute is used to define the routes

1. Add this statement in program.cs

builder.Services.AddControllersWithViews();

var app = builder.Build();

1. To configure attribute based routing

App.MapControllers();

With this middleware itself we build the attribute based routing

[Route("")]

public IActionResult Index()

{

return View();

}

The above route tells that whenever there is empty URL then this particular action method gets executed.

[Route("")]

[Route("Home")]

[Route("Home/Index")]

public IActionResult Index()

{

return View();

}

If the URL contains empty or Home or Home/index then the above action method gets executed

[Route("Home/Details/{id}")]

public IActionResult Details(int id)

{

return View(id);}

in the above route we can give id optional ( the the default value will be there and the action method gets executed

[Route("Home/Details/{id?}")]

Also we can give route on the class

Route can be given on controller or action method

Note : We can use both Conventional based and attribute based routing in a single application

app.MapControllerRoute( // for conventional based

name: "default",

pattern :"{controller=Home}/{action=About}/{id?}"

);

app.MapControllers(); //for attribute based

If we have both conventional and attribute based routing defined for an action , then what will be used? Then attribute based routing is given priority.

Attribute-based routing provides more control and flexibility in defining routes at the controller and action level, and it's often preferred for defining RESTful APIs or handling specific routing scenarios. However, you should use either conventional routing or attribute-based routing consistently within your application to avoid confusion and maintain readability.

The change in the name of controller and action name doesnot require the route template to be changed.

Token for controller -> [controller]

Token for action -> [action]

Both the above tokens can be used at a time

[Route("api/[controller]/[action]")]

public class YourController : Controller

{

[HttpGet]

public IActionResult Details(int id)

{

// Action logic

}

[HttpPost]

public IActionResult Update()

{

// Action logic

}

}

**Methods for routing**

app.UseEndpoints(endpoints =>

{

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

{

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

});

endpoints.MapPost("/Home", async (context) =>

{

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

});

endpoints.MapPut("/Home", async (context) =>

{

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

});

endpoints.MapDelete("/Home", async (context) =>

{

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

});

});

MapGet method is used to define an endpoint

An endpoint is something that can be selected, by matching the URL and HTTP method.

**Controller**

Backbone of MVC