Dotnet core

.NET Core is a cross-platform, open-source framework developed by Microsoft for building modern applications. It is the successor of the .NET Framework and provides better performance, scalability, and flexibility.

ASP.NET Core is a modern, high-performance framework for building web applications and APIs using .NET Core. It is faster, more flexible, and designed for dependency injection, middleware pipeline, and cloud integration

Project folder:

wwwroot Folder:
The wwwroot folder is the default location for static files such as CSS, JavaScript, images, and fonts.

Program.cs
This is the entry point of an ASP.NET Core application.
In .NET 6+, `Program.cs` uses the Minimal Hosting Model, combining both `Startup.cs` and `Program.cs` logic.

Startup.cs *(Optional in .NET 6+)*
Configures services and middleware (Not used in .NET 6+).
launchSettings.json
This file is located inside the Properties folder and is used to configure profiles for launching the application.
It defines environment variables, command-line arguments, and hosting settings for different environments (e.g., IIS Express, Kestrel).

appSettings.json
The configuration file for the application.
Stores database connection strings, API keys, logging settings, and other configurations

Middleware

Middleware is a software component that processes HTTP requests and responses in an ASP.NET Core application.

It is executed in a pipeline (sequential order) before the request reaches the controller and after the response is sent.

Common middleware components:

Authentication & Authorization

Exception Handling

Logging

Routing

Static Files

Routing

Routing maps incoming HTTP requests to the appropriate controller and action method.

It can be conventional routing (based on predefined patterns) or attribute routing (defined on controllers or actions).

Filters

Filters allow custom processing before or after executing an action method.

They help in logging, authentication, authorization, exception handling, caching, etc.

Types of Filters:

Authorization Filters – Runs before the request is handled

Action Filters – Runs before/after action execution

Exception Filters – Handles exceptions globally.

Result Filters – Runs before/after the result is executed.

Controller Initialization

When a request is received, ASP.NET Core initializes a controller.

The controller is created using dependency injection and then executes an action method.

Constructor injection can be used to pass services to a controller.

Built-in IoC Container

ASP.NET Core has a built-in Inversion of Control (IoC) container for Dependency Injection (DI).

it helps in managing dependencies and promoting loosely coupled code.

The IoC container resolves dependencies automatically when requested.

Registering Application Service

To use DI, we register services in the IoC container inside `Program.cs`.

Services can be registered using different lifetimes (Transient, Scoped, Singleton).

Understanding Service Lifetime

ASP.NET Core provides three types of service lifetimes

Extension Methods for Registration

Instead of writing all service registrations in `Program.cs`, we can create extension methods for better organization.

It keeps `Program.cs` clean and readable.

Constructor Injection

Dependencies are injected into a class through its constructor.

This helps in keeping code clean and testable.

Built-in IoC Container – ASP.NET Core has a built-in dependency injection system.

Registering Application Service – Services must be registered in `Program.cs`.

Understanding Service Lifetime – Three lifetimes: Transient, Scoped, and Singleton.

Transient – A new instance is created every time it is requested.

Best for lightweight, stateless services.

Example: A logging service that writes logs without maintaining state.

builder.Services.AddTransient<IMyService, MyService>();

Scoped – A single instance is created per request (HTTP request in a web app).

Best for database contexts or services that need to maintain state within a request.

Example: A service fetching user session details that should remain consistent throughout a request.

builder.Services.AddScoped<IMyService, MyService>();

Singleton – Only one instance exists for the lifetime of the application.

Best for shared resources like configuration settings or caching mechanisms.

Example: A service that maintains application-wide configuration or a cache manager.

builder.Services.AddSingleton<IMyService, MyService>();

Extension Methods for Registration – Used to keep service registrations modular.

Constructor Injection – Dependencies are injected via the constructor.

`UseDeveloperExceptionPage`

This middleware displays detailed error information in the browser when an unhandled exception occurs.

It is used only in the development environment to help developers debug issues quickly.

Shows detailed error messages (only for development).

```
if (app.Environment.IsDevelopment())
 {
app.UseDeveloperExceptionPage();
 }
```

`UseExceptionHandler`

This middleware handles exceptions globally and provides a custom error handling mechanism for production.

It ensures that users do not see technical error details and can be redirected to a friendly error page.

Logging API

ASP.NET Core provides a built-in Logging API to capture and manage logs.

It supports different log levels like `Trace`, `Debug`, `Information`, `Warning`, `Error`, and `Critical`.

The `ILogger` interface is used for logging messages

Logging Providers

Logging providers determine where logs are stored (console, files, database, etc.).

ASP.NET Core supports multiple built-in logging providers