# ASP.NET Core

## Introduction to .NET Core

* .NET Core is an open-source, cross-platform framework developed by Microsoft for building modern applications, including web, cloud, IoT, and mobile backends.
* It is a modular, lightweight, and high-performance successor to the .NET Framework.

### Key Benefits

| Benefit | Description |
| --- | --- |
| Cross-Platform | Runs on Windows, Linux, and macOS (unlike .NET Framework which is Windows-only) |
| Faster Performance | Optimized for speed and scalability |
| Microservices & Containers | Works well with Docker and Kubernetes |
| Lightweight & Modular | Includes only needed dependencies |
| Open Source | Actively developed by Microsoft and the community |
| Future-Proof | .NET Core (now .NET 5+) is the new standard |
| Better CLI Support | Powerful dotnet CLI for development |
| Side-by-Side Deployment | Run multiple .NET versions on the same machine |

## 

## ASP.NET Core Project Structure

### Program.cs

* The entry point of the application
* Configures the web host and logging
* Calls Startup to set up services and request handling

### Startup.cs

* Defines the application’s services and middleware pipeline
* Contains:
  + ConfigureServices – Registers services (DI, authentication, DB, etc.)
  + Configure – Sets up middleware (routing, error handling, CORS, etc.)

### launchSettings.json

* Defines launch configurations for different environments (IIS, Kestrel, etc.)
* Controls settings like URLs, environment variables, and profiles
* This file is in the **Properties** folder of an ASP.NET Core project:  
  📁 **/Properties/launchSettings.json**
* Specifies **application URLs** and **SSL settings**.

## Key Features of launchSettings.json

* Defines launch configurations for different environments like **IIS Express, Kestrel, or command-line execution**.
* Specifies **application URLs** and **SSL settings**.
* Allows setting up **environment variables**, such as ASPNETCORE\_ENVIRONMENT.
* Determines **whether the browser should open automatically** when running the app.
* Helps configure **debugging settings**.
* Sections Explained
* iisSettings (IIS Express Configuration)
* A computer screen shot of a code

  AI-generated content may be incorrect.
  + **windowsAuthentication**: Enables or disables Windows authentication (true or false).
  + **anonymousAuthentication**: Enables or disables anonymous authentication.
  + **iisExpress.applicationUrl**: Defines the base URL when running with IIS Express.
  + **sslPort**: Specifies the HTTPS port for SSL.
* **profiles (Launch Profiles)**

**A computer screen with many text

AI-generated content may be incorrect.**

** commandName: "Project" → Runs the application as a standalone .NET project (Kestrel).**

** dotnetRunMessages: true → Displays detailed messages when running dotnet run.**

** launchBrowser: true → Automatically launches a browser on application start.**

** applicationUrl: "https://localhost:5001;http://localhost:5000"**

* **Defines the URLs the app listens on.**
* **Supports both HTTP (http://localhost:5000) and HTTPS (https://localhost:5001).**

** environmentVariables:**

* **"ASPNETCORE\_ENVIRONMENT": "Development" → Sets the environment to Development.**

### appSettings.json

* Stores configuration settings like connection strings, logging, and API keys
* Supports environment-specific configurations (e.g., appSettings.Development.json)

### wwwroot Folder

* Stores static files like CSS, JavaScript, images, and fonts
* Files inside this folder are directly accessible via the browser
* Can be customized to serve static files from different locations:

A screen shot of a computer

AI-generated content may be incorrect.

This code enables your application to serve static files from a custom folder named "Customwwwroot" instead of the default "wwwroot" folder. The PhysicalFileProvider handles the file system access, and the empty RequestPath means these files will be served from the root URL.

## Middleware

### 

* **Handle Requests & Responses**: Middleware components are software units that process incoming requests and outgoing responses in an application pipeline.
* **Sequential Processing**: They execute in the order they are added to the pipeline, processing requests before passing them to the next middleware.
* **Conditional Execution**: Each middleware can decide whether to:
  + Pass the request to the next component.
  + Short-circuit the pipeline and generate a response immediately.
* **Pre & Post Processing**: Middleware can perform tasks both **before** and **after** calling the next middleware.
* **Examples**: Authentication, logging, exception handling, request compression, CORS, etc.

### Built-in Middleware Components

| Middleware | Description | Recommended Order |
| --- | --- | --- |
| Authentication | Provides authentication support | Before HttpContext.User is needed. Terminal for OAuth callbacks |
| Authorization | Provides authorization support | Immediately after Authentication Middleware |
| Cookie Policy | Tracks consent for storing personal information and enforces minimum standards for cookie fields | Before middleware that issues cookies |
| CORS | Configures Cross-Origin Resource Sharing | Before components that use CORS |
| DeveloperExceptionPage | Generates error information page for Development environment | Before components that generate errors |
| Diagnostics | Provides exception handling, status code pages, and default web pages | Before components that generate errors |
| Forwarded Headers | Forwards proxied headers onto the current request | Before components that consume the updated fields |
| Health Check | Checks the health of an app and its dependencies | Terminal if a request matches a health check endpoint |
| Header Propagation | Propagates HTTP headers from incoming request to outgoing HTTP Client requests | - |
| HTTP Logging | Logs HTTP Requests and Responses | At the beginning of the middleware pipeline |
| HTTP Method Override | Allows an incoming POST request to override the method | Before components that consume the updated method |
| HTTPS Redirection | Redirects all HTTP requests to HTTPS | Before components that consume the URL |
| HSTS | Adds HTTP Strict Transport Security header | Before responses are sent, after URL-modifying components |
| MVC | Processes requests with MVC/Razor Pages | Terminal if a request matches a route |
| Output Caching | Provides support for caching responses based on configuration | Before components that require caching |
| Response Caching | Provides support for caching responses | Before components that require caching |
| Request Decompression | Provides support for decompressing requests | Before components that read the request body |
| Response Compression | Provides support for compressing responses | Before components that require compression |
| Request Localization | Provides localization support | Before localization sensitive components |
| Request Timeouts | Configures request timeouts (global and per endpoint) | After UseExceptionHandler, UseDeveloperExceptionPage, and UseRouting |
| Endpoint Routing | Defines and constrains request routes | Terminal for matching routes |
| SPA | Handles requests by returning default page for Single Page Applications | Late in the chain |
| Session | Provides support for managing user sessions | Before components that require Session |
| Static Files | Provides support for serving static files and directory browsing | Terminal if a request matches a file |
| URL Rewrite | Provides support for rewriting URLs and redirecting requests | Before components that consume the URL |
| W3CLogging | Generates server access logs in W3C Extended Log File Format | At the beginning of the middleware pipeline |
| WebSockets | Enables the WebSockets protocol | Before components that accept WebSocket requests |

### Order of Middleware

The order in which middleware components are added in the Configure method of Startup.cs defines the sequence in which they process requests and responses:

1. Middleware added first processes the request first and the response last
2. Middleware added last processes, the request last and the response first

A diagram of a custom software

AI-generated content may be incorrect.

It’s critical to arrange middleware in the correct order to ensure proper functioning of the application.

### Short Circuit in Middleware

* **Definition**: A middleware component can **stop** request processing by not calling the next middleware in the pipeline.
* **How It Works**:
  + It **does not** call the next delegate in the pipeline.
  + It **handles the request** itself and directly generates a response.
* **Use Cases**:
  + **Authorization**: Blocks unauthorized access before reaching other components.
  + **Static Files**: Serves files (CSS, JS, images) without unnecessary processing.
  + **Custom Endpoints**: Handles specific requests (e.g., maintenance mode, health checks).

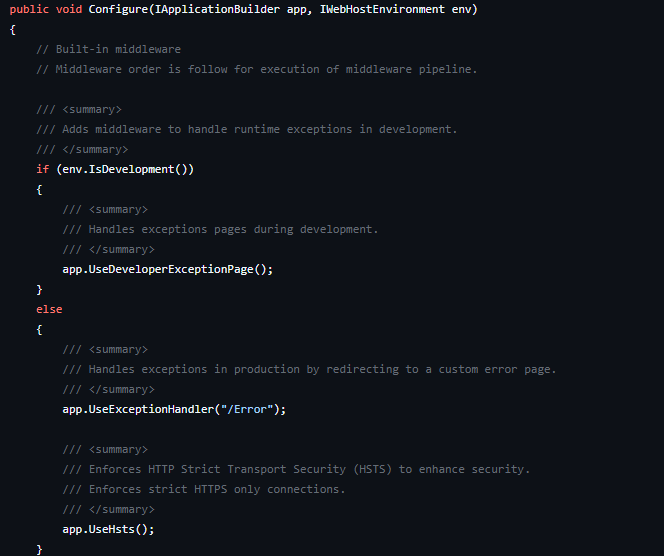
### Dependency of Current Middleware on Earlier Middleware

Certain middleware components rely on the correct **execution order** for proper functionality:

* **Routing → Endpoint Middleware**
  + UseRouting() **must come before** UseEndpoints().
  + Ensures the request is routed before executing the final endpoint.
* **Authentication → Authorization**
  + UseAuthentication() **must execute before** UseAuthorization().
  + Ensures users are **authenticated** before checking access permissions.
* **CORS → Response Caching**
  + UseCors() **must come before** UseResponseCaching().
  + Prevents cached responses from violating **CORS** policies.
* **Routing → Output Caching**
  + UseRouting() **must come before** UseOutputCaching().
  + Ensures correct request processing before caching responses.

### Global Exception Handling Using Middleware

ASP.NET Core provides built-in middleware for handling exceptions:

****

### Middleware Methods

ASP.NET Core provides several methods for building middleware components:

| Method | Description |
| --- | --- |
| Run() | Adds a terminal middleware that doesn’t call the next middleware |
| Use() | Adds a middleware that can process the request and optionally call the next middleware |
| Next | A delegate that represents the next middleware in the pipeline |
| Map methods | Used to branch the middleware pipeline based on the request path |

## Routing

Routing is the process of matching HTTP requests to endpoint handlers.

### Middleware Used for Routing

ASP.NET Core 5 uses the following middleware for routing: - UseRouting(): Adds route matching to the middleware pipeline - UseEndpoints(): Adds endpoint execution to the middleware pipeline

### Types of Routing

ASP.NET Core supports two primary types of routing:

#### 1. Conventional-based Routing

Defined centrally, typically in the Startup.Configure method:

app.UseEndpoints(endpoints =>  
{  
 endpoints.MapControllerRoute(  
 name: "default",  
 pattern: "{controller=Home}/{action=Index}/{id?}");  
});

#### 2. Attribute-based Routing

Defined using attributes directly on controllers and actions:

[Route("api/[controller]")]  
**public** **class** ProductsController : ControllerBase  
{  
 [HttpGet("{id}")]  
 **public** IActionResult GetProduct(int id)  
 {  
 *// Action implementation*  
 }  
}

## Filters

Filters allow code to run before or after specific stages in the request processing pipeline.

### Filter Pipeline

The filter pipeline runs after ASP.NET Core selects an action to execute but before the action executes.

### Difference Between Middleware and Filters

| Aspect | Middleware | Filters |
| --- | --- | --- |
| Scope | Global to the application | Scoped to controllers or actions |
| Execution | Before and after all controller processing | Within the MVC action invocation pipeline |
| Configuration | Configured in Startup.Configure | Applied using attributes or registered globally |
| Access to | Limited action/controller context | Full access to controller/action context |

### Types of Filters

ASP.NET Core provides several types of filters that execute at different stages:

| Filter Type | Description | Execution Order |
| --- | --- | --- |
| Authorization | Runs first and determines if the user is authorized for the request | 1 |
| Resource | Runs after authorization and before model binding | 2 |
| Action | Runs after model binding but before action execution | 3 |
| Exception | Handles exceptions that occur during action execution | 4 |
| Result | Runs before and after the execution of action results | 5 |

### Order Within Filters

Filters can be set to run at three different stages: - Global filters (registered in Startup.ConfigureServices) - Controller level filters (applied to a controller) - Action level filters (applied to a specific action)

Within each scope, multiple filters are executed in order of their filter type (as listed in the table above), not in the order they are applied.

### Controller Initialization

### Request Routing to a Controller in ASP.NET Core

When a request is routed to a controller, the following steps occur:

1. **Controller Instantiation**
   * The framework creates an instance of the controller.
2. **Dependency Injection**
   * Any required dependencies are injected via **constructor injection**.
3. **Action Method Execution**
   * The appropriate action method is called based on the request route and parameters.
4. **Result Processing & Response**
   * The action method's result is processed and returned as an HTTP response.

### Action Methods

Action methods are public methods on controller classes that handle HTTP requests.

#### Where to Use Action Methods

* To handle specific HTTP verbs (GET, POST, PUT, DELETE)
* To implement CRUD operations
* To process form submissions
* To return views, JSON, files, or other response types

#### Types of Action Methods

### ASP.NET Core Action Result Types

|  |  |
| --- | --- |
| **Result Type** | **Description** |
| IActionResult | The base interface for all action results. Allows returning different result types dynamically. |
| ActionResult | A concrete implementation of IActionResult that can return multiple result types. |
| ContentResult | Returns a simple text response (string content with a specified content type). |
| EmptyResult | Represents a response with no content (HTTP 204 No Content). |
| JsonResult | Returns JSON-formatted data, commonly used for APIs. |
| PartialViewResult | Returns a partial view, used for rendering a section of a page. |
| ViewResult | Returns a full HTML view as a response. |
| ViewComponentResult | Returns a **view component** result, used for rendering reusable UI components. |

### Difference Between IActionResult and ActionResult

| IActionResult | ActionResult |
| --- | --- |
| Interface | Abstract class |
| More flexible for unit testing | Provides base implementation |
| Commonly used with dependency injection | Required when returning typed results |
| Used for action methods that might return different result types | Base class for specific result types like ViewResult, JsonResult |

## Dependency Injection (DI)

Dependency Injection is a technique where one object supplies the dependencies of another object.

### Why Use DI

* Decouples components, making them easier to test
* Allows for better code organization and separation of concerns
* Facilitates easier maintenance and updates
* Enables configuration of components in one central location
* Promotes the use of interfaces rather than concrete implementations

### Definition of Dependency and Injection

* **Dependency**: An object that another object depends on to perform its functions
* **Injection**: The process of providing the required dependencies to an object

### Advantages of DI

* Reduces class coupling
* Increases code reusability
* Improves code maintainability
* Facilitates unit testing through mocking
* Enables runtime configuration changes
* Enhances application flexibility

### Types of Dependency Injection

#### 1. Constructor Injection

Dependencies are provided through a class constructor.

**public** **class** ProductsController : ControllerBase  
{  
 **private** **readonly** IProductService \_productService;  
  
 **public** ProductsController(IProductService productService)  
 {  
 \_productService = productService;  
 }  
}

* **When to use**: For required dependencies that the class cannot function without
* **Advantages**: Clear dependency requirements, works well with immutable objects

#### 2. Property Injection

Dependencies are provided through public properties.

**public** **class** ProductsController : ControllerBase  
{  
 **public** IProductService ProductService { **get**; **set**; }  
}

* **When to use**: For optional dependencies
* **Advantages**: Allows for late binding of dependencies

#### 3. Method Injection

Dependencies are provided through method parameters.

**public** **class** ProductService  
{  
 **public** void ProcessProduct(Product product, ILogger logger)  
 {  
 logger.LogInformation("Processing product");  
 *// Process the product*  
 }  
}

* **When to use**: When different dependencies are needed for different operations
* **Advantages**: Very specific to the method’s requirements

### Built-in IoC Container

ASP.NET Core includes a built-in Inversion of Control (IoC) container for registering and resolving dependencies.

### Registering Application Services – 3 Steps

1. Create interfaces and implementations for your services
2. Register services in the ConfigureServices method in Startup.cs
3. Request dependencies via constructor injection in consuming classes

*// Step 1: Create interface and implementation*  
**public** **interface** IProductService  
{  
 Product GetProduct(int id);  
}  
  
**public** **class** ProductService : IProductService  
{  
 **public** Product GetProduct(int id) => **new** Product { Id = id };  
}  
  
*// Step 2: Register in Startup.ConfigureServices*  
**public** void ConfigureServices(IServiceCollection services)  
{  
 services.AddTransient<IProductService, ProductService>();  
}  
  
*// Step 3: Request via constructor injection*  
**public** **class** ProductsController : ControllerBase  
{  
 **private** **readonly** IProductService \_productService;  
  
 **public** ProductsController(IProductService productService)  
 {  
 \_productService = productService;  
 }  
}

### Understanding Service Lifetime

ASP.NET Core DI supports three service lifetimes:

#### 1. Transient

services.AddTransient<IMyService, MyService>();

* **Definition**: A new instance is created each time the service is requested
* **Where to use**: For lightweight, stateless services
* **Scope**: Each request to the service creates a new instance
* **Best for**: Services that are short-lived and don’t maintain state

#### 2. Scoped

services.AddScoped<IMyService, MyService>();

* **Definition**: A new instance is created once per client request (connection)
* **Where to use**: For services that should maintain state within a request
* **Scope**: Per HTTP request in web applications
* **Best for**: Services that need to maintain state during a single request, like DbContext

#### 3. Singleton

services.AddSingleton<IMyService, MyService>();

* **Definition**: A single instance is created and shared throughout the application’s lifetime
* **Where to use**: For services that need to maintain state across requests
* **Scope**: Application-wide
* **Best for**: Services that are expensive to initialize or need to maintain global state

### Extension Methods for Registration

Extension methods provide a cleaner way to register services:

*// Extension method*  
**public** **static** **class** ServiceCollectionExtensions  
{  
 **public** **static** IServiceCollection AddProductServices(**this** IServiceCollection services)  
 {  
 services.AddScoped<IProductService, ProductService>();  
 services.AddScoped<IInventoryService, InventoryService>();  
 **return** services;  
 }  
}  
  
*// Usage in Startup.ConfigureServices*  
**public** void ConfigureServices(IServiceCollection services)  
{  
 services.AddProductServices();  
}

**Why use extension methods:** - Organizes related service registrations - Improves readability in the ConfigureServices method - Facilitates reuse across different applications - Encapsulates complex registration logic

Exception Handling

### UseDeveloperExceptionPage

This middleware captures synchronous and asynchronous exceptions from the pipeline and generates HTML error responses.

**if** (env.IsDevelopment())  
{  
 app.UseDeveloperExceptionPage();  
}

**Features:** - Displays detailed exception information - Shows stack trace - Shows source code where the exception occurred - Shows query string parameters - Shows cookies - **Only suitable for the development environment**

### UseExceptionHandler

This middleware catches exceptions, logs them, and re-executes the request in an alternate pipeline.

**if** (!env.IsDevelopment())  
{  
 app.UseExceptionHandler("/Error");  
}

**Features:** - Handles exceptions safely in production - Redirects to a dedicated error handling endpoint - Preserves the original status code - Doesn’t expose sensitive error details to clients

## Logging in ASP.NET Core

### Logging API

ASP.NET Core provides a built-in logging API through the Microsoft.Extensions.Logging namespace.

Key components: - ILogger<T>: Generic interface for logging with category name derived from T - ILoggerFactory: Creates loggers - ILoggerProvider: Creates logger instances

Logging Levels (in increasing order of severity): 1. Trace 2. Debug 3. Information 4. Warning 5. Error 6. Critical 7. None

### Logging Providers

ASP.NET Core includes several built-in logging providers:

| Provider | Package | Description |
| --- | --- | --- |
| Console | Microsoft.Extensions.Logging.Console | Logs to the console window |
| Debug | Microsoft.Extensions.Logging.Debug | Logs to the debugger output window |
| EventSource | Microsoft.Extensions.Logging.EventSource | Uses EventSource/EventListener |
| EventLog | Microsoft.Extensions.Logging.EventLog | Logs to Windows Event Log |
| TraceSource | Microsoft.Extensions.Logging.TraceSource | Uses System.Diagnostics.TraceSource |
| Azure App Service | Microsoft.Extensions.Logging.AzureAppServices | Logs to Azure App Service |

Configuring logging in Program.cs:

**public** **static** IHostBuilder CreateHostBuilder(string[] args) =>  
 Host.CreateDefaultBuilder(args)  
 .ConfigureLogging((hostingContext, logging) =>  
 {  
 logging.ClearProviders();  
 logging.AddConsole();  
 logging.AddDebug();  
 logging.AddEventSourceLogger();  
 *// Configure minimum log level*  
 logging.SetMinimumLevel(LogLevel.Information);  
 })  
 .ConfigureWebHostDefaults(webBuilder =>  
 {  
 webBuilder.UseStartup<Startup>();  
 });

Using the logger in a controller:

**public** **class** HomeController : Controller  
{  
 **private** **readonly** ILogger<HomeController> \_logger;  
  
 **public** HomeController(ILogger<HomeController> logger)  
 {  
 \_logger = logger;  
 }  
  
 **public** IActionResult Index()  
 {  
 \_logger.LogInformation("Index page visited at {time}", DateTime.Now);  
 **return** View();  
 }  
}