**.NET Core Fundamentals**

**.NET Core Overview**

.NET Core is an open-source, cross-platform framework developed by Microsoft for building modern, cloud-based, and high-performance applications. It is the successor to the .NET Framework and provides several advantages, including:

* **Cross-Platform Support**: Runs on Windows, macOS, and Linux.
* **Modular Architecture**: Uses NuGet packages for dependencies.
* **High Performance**: Optimized runtime and garbage collection.
* **Microservices & Cloud Readiness**: Suitable for Docker, Kubernetes, and cloud deployments.
* **Open Source**: Actively developed and maintained on GitHub.

**ASP.NET Core**

ASP.NET Core is a web framework built on .NET Core for developing web applications and APIs. Key features include:

* **Unified MVC & Web API**: Uses a single controller-based architecture.
* **Dependency Injection**: Built-in support for managing dependencies.
* **Middleware Pipeline**: Customizable request-processing pipeline.
* **Cross-Platform Hosting**: Can be hosted on IIS, Kestrel, or Docker.
* **Security & Performance**: Supports authentication, authorization, and optimized execution.

**Project Structure**

A typical ASP.NET Core project consists of the following directories and files:

* **wwwroot/**: Static files (CSS, JavaScript, images, etc.).
* **Controllers/**: Handles HTTP requests and responses.
* **Models/**: Represents data and business logic.
* **Views/**: UI templates (for MVC applications).
* **appsettings.json**: Configuration settings.
* **Program.cs**: Application entry point.
* **Startup.cs**: Configures services and middleware.
* **launchSettings.json**: Development and debugging settings.

A screenshot of a computer

Description automatically generated

**wwwroot Folder**

The wwwroot folder is used to store static assets such as:

* CSS files
* JavaScript files
* Images
* Fonts This folder is exposed to clients, and files inside can be directly accessed via URLs.

**Program.cs**

Program.cs is the entry point of an ASP.NET Core application. It:

* Configures the host.
* Calls CreateHostBuilder() to configure services and middleware.
* Uses WebApplication to build and run the application.

Example:   
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/", () => "Hello, World!");

app.Run();

**Startup.cs (Removed from .Net 6)**

Startup.cs defines the request pipeline and services. It contains:

* ConfigureServices(): Registers dependencies like Entity Framework, authentication, etc.
* Configure(): Defines middleware like routing, authentication, and exception handling.

Example :

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers();

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

app.UseRouting();

app.UseEndpoints(endpoints => endpoints.MapControllers());

}

}

Note: Startup.cs is optional in .NET 6+ as configurations are done in Program.cs

**launchSettings.json**

Located in Properties/, this file defines environment-specific settings for running and debugging the application. Example:

Example:

"profiles": {

"MVCWebApp": {

"commandName": "Project",

"dotnetRunMessages": true,

"launchBrowser": true,

"applicationUrl": "http://localhost:5190",

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

},

"IIS Express": {

"commandName": "IISExpress",

"launchBrowser": true,

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

}

}

The applicationUrl property determines the URL on which the application will run during development.

The ASPNETCORE\_ENVIRONMENT setting defines the environment (e.g., Development, Production).

**appSettings.json**

This file contains configuration settings for the application, such as:

* Connection strings
* Logging settings
* API keys
* Custom application settings

Example:

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"AllowedHosts": "\*"

}

**ASP.NET Core Request Processing Pipeline**

**Middleware**

Middleware is software that is executed in the request pipeline before reaching the endpoint (such as a controller action). Middleware components handle requests and responses and can perform tasks such as authentication, logging, exception handling, and routing.

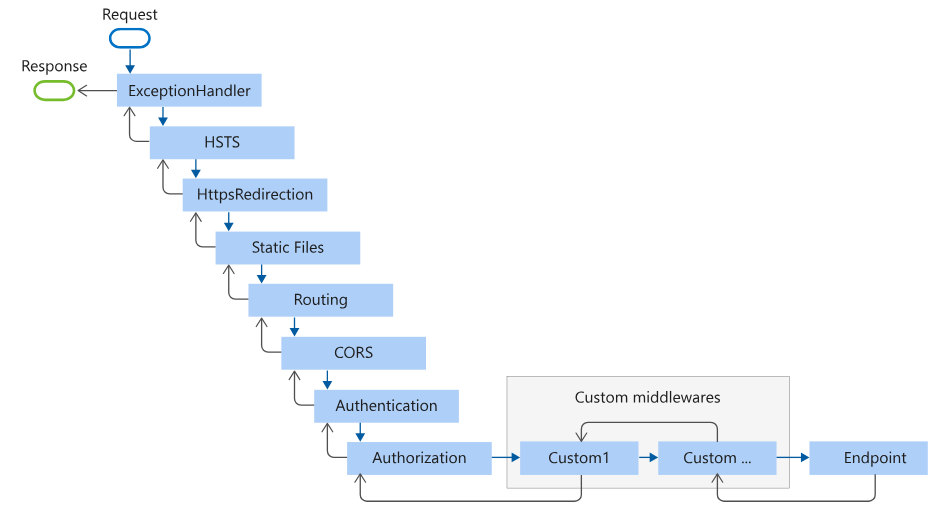
**Key Features of Middleware:**

* Executes in the order they are added in Program.cs.
* Can modify requests before passing them to the next component.
* Can short-circuit the pipeline to prevent further processing.
* Can modify responses before sending them to the client.

Flow of multiple middlewares :



Middleware order:



**Routing**

* Routing maps incoming HTTP requests to appropriate controller actions.
* Defined in Program.cs or directly on controllers using attributes.

**Types of Routing**:

* **Conventional Routing**: Centralized route configuration in Program.cs

Example:

app.MapControllerRoute(

name: "default",

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

* **Attribute Routing**: Defined at the controller or action level using attributes like [Route], [HttpGet], etc.

Example :

[Route("api/products")]

[ApiController]

public class ProductsController : ControllerBase

{

[HttpGet]

public IActionResult GetProducts()

{

return Ok(new [] { "Product1", "Product2" });

}

}

**Route Parameters**:

* **Required**: Defined as {parameterName}, e.g., {id}.
* **Optional**: Defined as {parameterName?}, e.g., {id?}.
* **Constraints**: Enforce specific formats, e.g., {id:int} (integer).

**Filters**

Filters allow code execution before or after an action method runs. They help in tasks such as authentication, caching, and logging.

**Types of Filters:**

* **Authorization Filters**: Handle authentication and authorization.

Run before any other filter and determine whether the user is authorized to access the requested resource.

Use Case: Check if the user is authenticated before proceeding with the action.

* **Action Filters**: Execute before or after an action method.

Run before and after an action method executes. Typically used for logging, validation, or performance tracking.

Use Case: Log method execution time or validate model data.

* **Exception Filters**: Handle errors.

Handle exceptions thrown during action execution and return a custom error response.

Use Case: Return a standardized error response for unhandled exceptions.

* **Result Filters**: Modify response results.

Run before and after the result is executed. Used for modifying or processing the response.

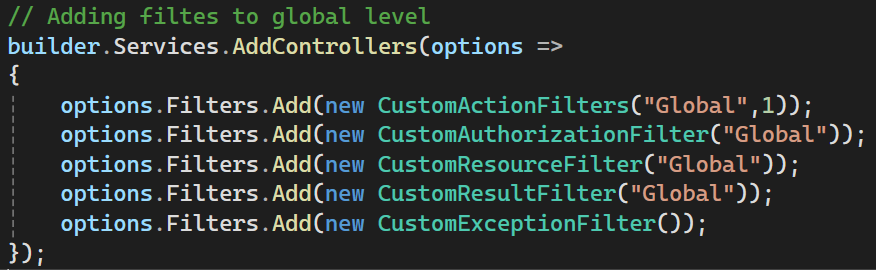
Use Case: Modify the response before it is sent to the client.

* **Resource Filters**: Execute before and after the rest of the pipeline, useful for caching and resource initialization.

Run after authorization filters but before model binding, useful for caching or resource initialization.

Use Case: Initialize resources or cache data before executing the action.

**Example of Global Filter Registration in Program.cs :**



**Controller Initialization**

Controllers in ASP.NET Core handle HTTP requests and return responses. The initialization process includes:

1. Resolving dependencies via dependency injection.
2. Creating an instance of the controller.
3. Executing any constructor logic before processing requests.

Example:

[ApiController]

[Route("api/users")]

public class UsersController : ControllerBase

{

private readonly IUserService \_userService;

public UsersController(IUserService userService)

{

\_userService = userService;

}

[HttpGet]

public IActionResult GetUsers()

{

var users = \_userService.GetAllUsers();

return Ok(users);

}

}

**Action Method**

An action method processes incoming HTTP requests and returns responses. Action methods can:

* Return various result types (JsonResult, ObjectResult, ContentResult, etc.).
* Use model binding to handle input data from query strings, route parameters, or request bodies.
* Be decorated with attributes like [HttpGet], [HttpPost], [Authorize], [FromQuery], [FromBody], etc.

Example :

[HttpPost]

public IActionResult CreateUser([FromBody] UserDto userDto)

{

if (!ModelState.IsValid)

{

return BadRequest(ModelState);

}

var user = \_userService.CreateUser(userDto);

return Ok(“user created”);

}

Example of different types of return in action methods :

A black screen with white text

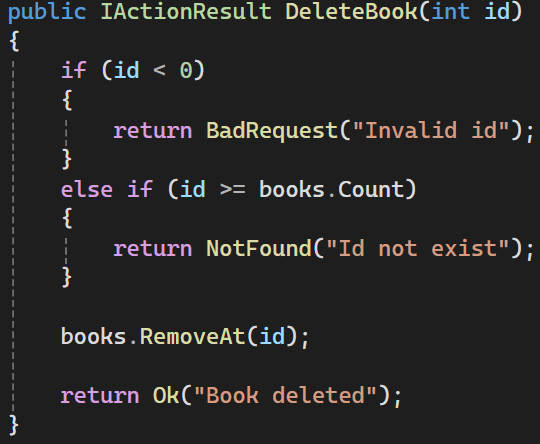
Description automatically generated

A black screen with white text

Description automatically generated

A black background with white text

Description automatically generated



**Dependency Injection**

Dependency Injection (DI) is a design pattern used to manage dependencies in an application. ASP.NET Core has a built-in Inversion of Control (IoC) container that facilitates DI, promoting modular, testable, and maintainable code.

DI allows objects to receive their dependencies from an external source rather than creating them internally, improving flexibility and reducing tight coupling between components.

**Built-in IoC Container**

ASP.NET Core provides a built-in IoC (Inversion of Control) container to manage dependency injection. This container is responsible for creating and managing service instances throughout the application.

**Features of ASP.NET Core IoC Container:**

* Registers dependencies in Program.cs.
* Supports different service lifetimes (Singleton, Scoped, Transient).
* Resolves dependencies automatically when needed.

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

**Registering Application Service**

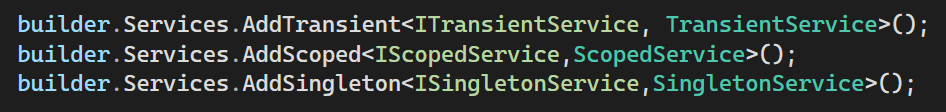
To use DI, services must be registered in the IoC container within Program.cs. ASP.NET Core provides various extension methods to register services, such as AddSingleton, AddScoped, and AddTransient.

Registered different types of services, which are resolved when requested in the application.

**Understanding Service Lifetime**

ASP.NET Core DI container provides three types of service lifetimes:

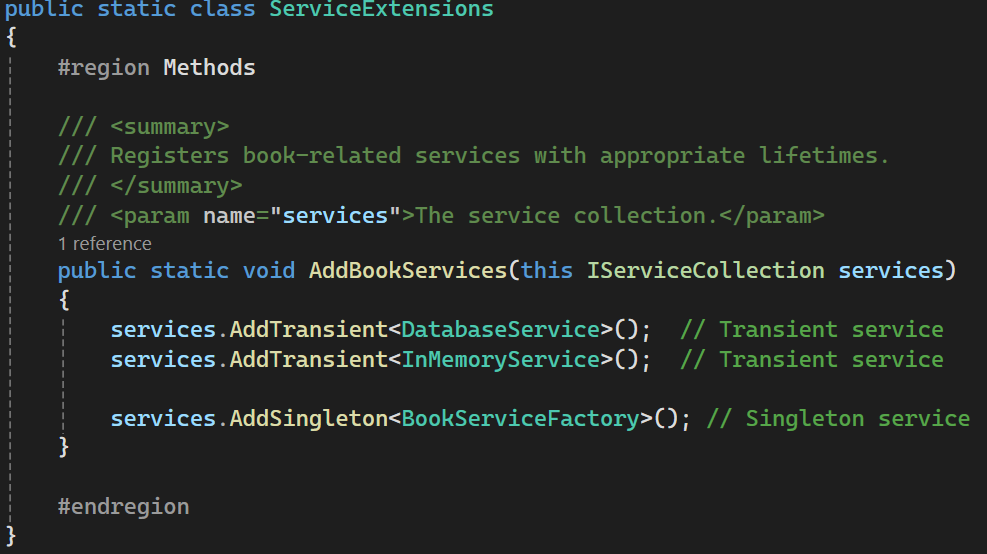
1. **Singleton** - The service is created once and shared throughout the application's lifetime.
2. **Scoped -** A new instance is created per request scope.
3. **Transient -** A new instance is created each time it is requested.



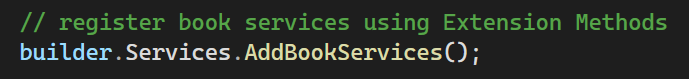
**Extension Methods for Registration**

To simplify service registration and improve maintainability, we can create extension methods for registering services.

Example :



Then use it in Program.cs:

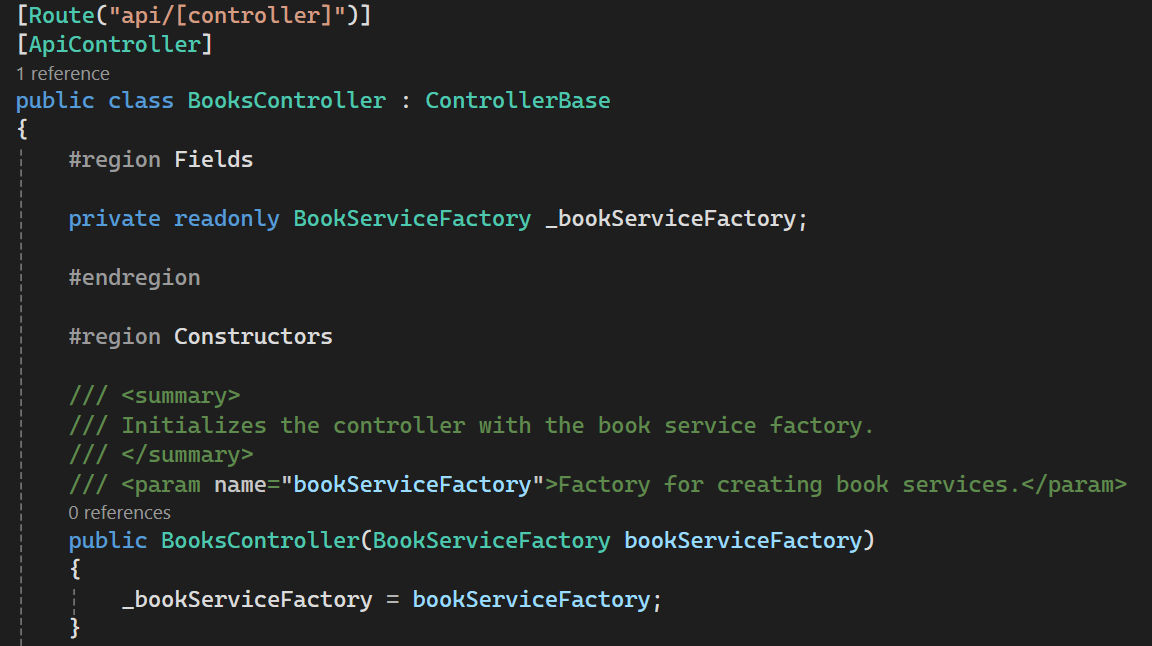


This improves code organization and reusability

**Constructor Injection**

Constructor Injection is the most common way to inject dependencies into classes. The IoC container automatically provides registered services through the constructor.

Example :



**Exception Handling**

**UseDeveloperExceptionPage**

**Purpose**: This middleware is used to display detailed error information during development to help developers diagnose issues quickly.

**When to Use**: It is only enabled in the **Development** environment.

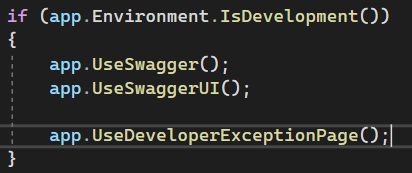
**When NOT to Use**: **In Production**: Exposing detailed error information in production can be a security risk, so it is **not enabled** in production environments.

**Behavior**:

It provides detailed error pages, including the exception's stack trace, request details, and source code, if available.

The developer can see exactly where and why an exception occurred in the application.

**Code Example**:



**Key Features**:

* Shows the exception details (e.g., message, stack trace).
* Displays request details, such as query parameters, headers, and more.
* Helps developers quickly identify issues and fix them during development.

**UseExceptionHandler**

**Purpose**: This middleware is used to handle exceptions in a controlled way, often used in production environments to avoid exposing sensitive information to users.

**When to Use**: It is typically enabled in the **Production** environment.

**When NOT to Use**: **In Development**: It might obscure useful debugging information that is needed for development, so it is **not recommended** in development environments.

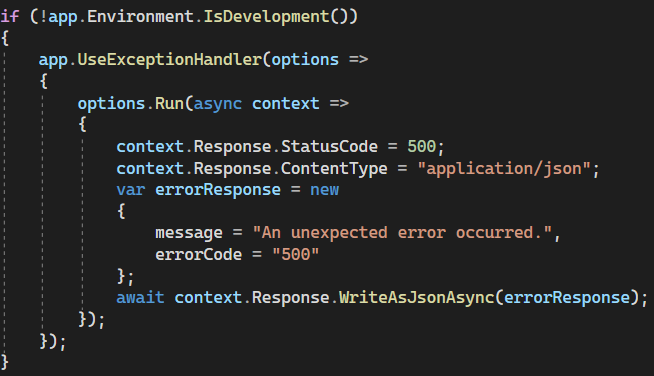
**Behavior**:

It provides a centralized way of catching unhandled exceptions.

Developers can define a custom error-handling route (e.g., a global error page or API error response).

Unlike UseDeveloperExceptionPage, it does not display detailed exception information to the end user but can return a generic error message or redirect to a custom error page.

**Code Example**:



**Key Features**:

* Redirects to a designated error handler route (e.g., /Home/Error).
* Ensures that no detailed exception data is exposed to the client in production.
* Can be configured to return a custom error page or a generic JSON response.

**Logging in ASP.NET Core**

**Logging API**

The **Logging API** in ASP.NET Core is a built-in framework for handling logging activities. It provides a simple interface to capture logs across various components of an application. Key components include:

**ILogger Interface**: The main interface used to log messages.

* Methods: LogInformation(), LogWarning(), LogError(), LogCritical(), LogDebug(), LogTrace(), etc.
* Example: ILogger allows you to capture different severity levels of logs and direct them to various outputs.

**ILoggerFactory Interface**: Responsible for creating instances of ILogger that can be used throughout the application.

* It helps in configuring loggers and adding logging providers.

**LogLevel Enum**: Defines the severity of the logs.

* Levels include:
  + Trace: Very detailed information, useful only for debugging.
  + Debug: Information useful for debugging but not for general application monitoring.
  + Information: Normal application flow, general information.
  + Warning: Warnings about potential issues that aren’t errors.
  + Error: Indicates failure or critical issues.
  + Critical: Logs of major failures requiring immediate attention.

**Loggers and Log Entries**:

* Loggers are used to capture log entries (e.g., Log.Information("Log message")).
* You can inject ILogger<T> into controllers, services, middleware, or other components to capture logs at runtime.

**Logging Providers**

**Logging Providers** are the destinations where log messages are sent. ASP.NET Core supports several built-in logging providers, and third-party providers can be added for more specialized logging solutions.

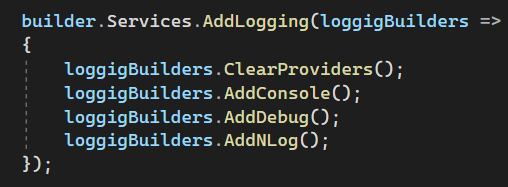
**Built-In Logging Providers:**

1. **Console Logger**:
   * Writes log messages to the console.
   * Useful for development or simple applications.
   * Configured using .AddConsole().
2. **Debug Logger**:
   * Sends log messages to the **Debug Output** window in development tools like Visual Studio.
   * Great for debugging applications in development.
   * Configured using .AddDebug().
3. **EventSource Logger**:
   * Writes logs to **EventSource** for monitoring purposes, often used for performance or diagnostic data collection.
   * More specialized and often used in production systems for tracking application events.
4. **File Logger** (requires third-party libraries like **Serilog**, **NLog**, etc.):
   * Writes logs to files. Useful for persistent storage and long-term log analysis.
   * To use, you need to integrate a third-party provider such as **Serilog** (Serilog.AspNetCore) or **NLog**.
   * Configured using .WriteTo.File("logfile.log") in the case of Serilog.

**Configuring Logging**

**Program.cs**: ASP.NET Core allows you to configure logging during the application setup in Program.cs:

* **AddLogging**: Adds one or more logging providers to the service container.



**appsettings.json**: You can configure logging behavior and levels through the appsettings.json file, specifying log levels and which providers to use.

