.net core

Created by | Dakshil Gorasiya

.NET Core (now simply called **.NET**) is **Microsoft's modern, open-source, cross-platform framework for building diverse applications, including web apps, microservices, cloud services, and desktop apps**, that run on Windows, macOS, and Linux.

It's a successor to the Windows-only .NET Framework, offering high performance, modularity, and flexibility through features like ASP.NET Core for web development, Blazor for web UIs, and support for languages like C#, F#, and VB.NET.

Folder structure of asp.net core project

Controllers

- Handles **HTTP requests** (GET, POST, PUT, DELETE)
- Acts as the **entry point** of the API
- Should contain **minimal business logic**
- Inherit ControllerBase
- Calls service

Models

- Represent data structure
- Entity models

Data

- Migration files
- DbContext.cs

Services

- Contains business rules
- Keep controller clean
- Use repository for db opertaions

Repository

- Interact with Database
- No business logic

DTOs

- Prevents exposing **database entities**
- Controls what data goes in/out of API

Middleware

- Global exception handling
- Logging
- Request/response manipulation

Properties

- Defines **how the app runs locally**

- Ports, environment, startup URL

wwwroot

- static files

Program.cs

Its job is

1. Create Host
2. Register Services (Dependency Injection)
3. Configure HTTP Request Pipeline (Middleware)

It first create builder

```
var builder = WebApplication.CreateBuilder(args);
```

Then it register services using DI

```
builder.Services.AddControllers();
builder.Services.AddDbContext<AppDbContext>();
builder.Services.AddScoped<IUserService, UserService>();
builder.Services.AddAuthentication();
builder.Services.AddAuthorization();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
```

Then build app

```
var app = builder.Build();
```

Configure middleware pipeline

```
app.UseSwagger();
app.UseSwaggerUI();
app.UseHttpsRedirection();
app.UseAuthentication();
app.UseAuthorization();
app.MapControllers();
```

Run app

```
app.Run();
```

appsettings.json

Used for configuration

Ex:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=.;Database=DemoDb;Trusted_Co
nnection=True;"
  },

  "Jwt": {
    "Key": "super-secret-key",
    "Issuer": "https://localhost:5001",
    "Audience": "https://localhost:5001",
    "ExpiryMinutes": 60
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },

  "AllowedHosts": "*"
}
```

To access value

```
var key = builder.Configuration["Jwt:Key"];
```

```
builder.Services.Configure<JwtSettings>(
    builder.Configuration.GetSection("Jwt"));

public class JwtSettings
{
    public string Key { get; set; }
    public string Issuer { get; set; }
}
```

Environments

Environments define **where** and **how** your application is running, so the framework can change behavior **without changing code**.
Three environment are given by .net

1. Development
2. Staging

3. Production

In code to check environment we can use

```
app.Environment.IsDevelopment()
app.Environment.IsStaging()
app.Environment.IsProduction()
```

Environment is set by Properties → environmentVariables →ASPNETCORE_ENVIRONMENT

To create environment we can I write any string in it and then to check environment

```
app.Environment.IsEnvironment("QA")
```

We can create specific appsettings for different environment which will be loaded accoring to environment

Ex: appsettings.QA.json

Priority while loading configuration

1. appsettings.json
2. appsettings.{Environment}.json
3. Environment variables
4. Command-line args

Last one wins

To get current environment

```
var envName = app.Environment.EnvironmentName;
```

When hosted it completly ignore lauchsettings.json and load app in production environment

Minimal hosting model

The **Minimal Hosting Model** is a **simplified startup model** introduced in **.NET 6**, where:

- `Startup.cs` is removed
- Everything is configured in `Program.cs`
- Much less boilerplate
- Easier to read and maintain

To use startup.cs

1. Create Startup.cs

```
namespace WebApplication1
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // Register services
        public void ConfigureServices(IServiceCollection serv
ices)
        {
            services.AddControllers();
            services.AddSwaggerGen();
        }

        // Configure HTTP pipeline
        public void Configure(WebApplication app, IWebHostEnv
ironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseSwagger();
                app.UseSwaggerUI();
            }

            app.UseHttpsRedirection();
            app.UseAuthorization();

            app.MapControllers();
        }
    }
}
```

2. Use it in Program.cs

```

    namespace WebApplication1
    {
        public class Program
        {
            public static void Main(string[] args)
            {
                var builder = WebApplication.CreateBuilder(ar
gs);

                var startup = new Startup(builder.Configurati
on);
                startup.ConfigureServices(builder.Services);
                var app = builder.Build();

                startup.Configure(app, app.Environment);

                app.Run();
            }
        }
    }
```

Middleware

In **ASP.NET Core**, **middleware** is software that is assembled into an **application pipeline** to handle **HTTP requests and responses**.

Each middleware can:

- **Receives an HTTP request**
- Can **perform operations** on it
- Can **pass it to the next middleware** in the pipeline
- Optionally, **perform operations on the response** before sending it back to the client

How middleware pipeline work

1. Request come enters into middleware pipeline
2. Controller executes
3. Response pass through middleware pipeline in reverse orede

Build in middleware in asp.net core

Middleware	Purpose
`UseRouting()`	Determines which endpoint will handle the request
UseEndPoints()	Execute determined action method
`UseAuthentication()`	Checks if the user is authenticated
`UseAuthorization()`	Checks if the user has permission
`UseStaticFiles()`	Serves static files (CSS, JS, images)
`UseCors()`	Handles Cross-Origin Resource Sharing (CORS)
`UseExceptionHandler()`	Handles errors globally
`UseHttpsRedirection()`	Redirects HTTP to HTTPS
MapControllers()	Does **both routing and endpoint execution**
UseCors()	Enables Cross Origin Resource Sharing

To create custom middleware

1. By creating a class

```
public class LoggingMiddleware
    {
        private readonly RequestDelegate _next;

        public LoggingMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public async Task InvokeAsync(HttpContext context)
        {
            Console.WriteLine($"Logging middleware Incoming r
equest : {context.Request.Method} {context.Request.Path}");

            await _next(context); // To call next middleware

            Console.WriteLine($"Logging middleware outgoing r
equest : {context.Response.StatusCode}");
        }
    }
```

To inject it

```
app.UseMiddleware<LoggingMiddleware>();
```

2. By creating extension method a class

```
public static class LoggingMiddlewareExtension
    {
        public static IApplicationBuilder UseLogging(this IAp
plicationBuilder builder)
        {
            return builder.UseMiddleware<LoggingMiddleware>
();
        }
    }
```

To use it

```
app.UseLogging();
```

3. Shorthand method

```
app.Use(async (context, next) =>
            {
                Console.WriteLine($"Request: {context.Reques
t.Path}");

                await next();
                Console.WriteLine($"Response: {context.Respon
se.StatusCode}");
            });
```

4. For specific route

```
app.UseWhen(context => context.Request.Path == "/WeatherForec
ast/hello", helloApp =>
            {
                helloApp.Use(async (context, next) =>
                {
                    Console.WriteLine("Hello called");
                    await next();
                });
            });
```

Dependency Injection

Instead of a class **creating is own dependencies**, they are **provided (injected)** from outside is known as dependency injection

Ex:

```
public class OrderService
{
    private readonly IEmailService _email;

    public OrderService(IEmailService email)
    {
        _email = email;
    }
}
```

Advantages:

- Loose coupling
- Easy testing
- Easy to swap implementations

ASP.NET Core has a **built-in DI container**.

It works in **3 steps**:

1. **Register services**
2. **Resolve services**
3. **Inject services**

`builder.Services` has an `IServiceCollection` a container to **register services** that your app will use.

Method used to inject services

- `AddControllers()` → Adds MVC controllers and routing services.
- `AddAuthorization()` → Adds services needed for policy-based or role-based authorization.
- `AddScoped<TInterface, TImplementation>()` → Registers a service with scoped lifetime.

`ServiceProvider` is the **built DI container** in ASP.NET Core.

After you finish adding services to `builder.Services` (the `IServiceCollection`), ASP.NET Core **builds a `ServiceProvider`** internally.

It's the thing that actually **creates instances of your services** when requested.

Singleton

One instance for entire application

use cases

- App configuration
- In-memory caching
- Third-party API clients
- Logging helpers
- ML model loaded once

```
builder.Services.AddSingleton<IAppSettingsService, AppSetting
sService>();
```

Scoped

One instance per HTTP request

use cases

- Business logic services
- Database access
- User-specific operations
- Unit of Work pattern

```
builder.Services.AddScoped<IUserService, UserService>();
```

Transient

New instance every time requested

use cases

- Email sender
- SMS sender
- Password hashing
- Lightweight helpers
- Validation services

```
builder.Services.AddTransient<IEmailService, EmailService>();
```

Ways to resolve dependency

1. using constructor

Order does not matter in constructor parameter

```
[ApiController]
[Route("api/users")]
public class UsersController : ControllerBase
{
    private readonly IUserService _userService;

    public UsersController(IUserService userService)
    {
        _userService = userService;
    }

    [HttpGet]
    public IActionResult Get()
    {
        return Ok(_userService.GetUsers());
    }
}
```

2. Using method

```
public IActionResult GetUser(IUserService userService)
```

3. Using service provider

```
private readonly IServiceProvider _serviceProvider;
public ValuesController(IServiceProvider serviceProvider)
        {
            _serviceProvider = serviceProvider;
        }
[HttpGet("SendEmail")]
        public IActionResult SendEmail()
        {
            var emailService = _serviceProvider.GetService<IE
mailService>();
            emailService.Send("HI");
            return Ok();
        }
```

4. Using ActivatorUtilities

```
[HttpGet("GetSettings")]
        public IActionResult GetSettings()
        {
            var userSetting = ActivatorUtilities.CreateInstan
ce<SettingsService>(_serviceProvider);
            return Ok(userSetting.AppName);
        }
```

What if two service is registered

```
builder.Services.AddScoped<IMessageService, EmailService>();
builder.Services.AddScoped<IMessageService, SmsService>();
```

It will get last service only

```
public MyController(IMessageService service);
```

It will get both service

```
public MyController(IEnumerable<IMessageService> services)
```

Services can be injected into

- Middleware
- Filters
- Minimal API endpoints

Built-in services

Built-in services are services that **ASP.NET Core automatically registers** in the DI container.

Service	Purpose	Lifetime
`ILogger<T>`	Logging	Singleton
`IConfiguration`	Config access	Singleton
`IWebHostEnvironment`	Env info	Singleton
`IHttpContextAccessor`	Request access	Singleton

Option pattern

The Options Pattern in ASP.NET Core is a way to bind configuration settings (from appsettings.json, environment variables, etc.) to strongly-typed classes and inject them via Dependency Injection (DI).

Why use the Options Pattern?

- Avoid accessing configuration via `IConfiguration` everywhere in code.
- Provides **strongly-typed access** to settings.
- Makes it easier to **validate settings**.
- Works seamlessly with **DI**.

To use it

1. Add setting in appsetting.json

```
"DatabaseSettings": {
    "ConnectionString": "MyConnectionString",
    "Database": "MySQL"
  }
```

2. Create a strongly typed class

```
public class DatabaseSettings
    {
        public string? ConnectionString { get; set; }
        public string? Database { get; set; }
    }
```

3. Register the option in Program.cs

```
builder.Services.Configure<DatabaseSettings>(builder.Configur
ation.GetSection("DatabaseSettings"));
```

4. Inject it from controller

```
public class ValuesController : ControllerBase
    {
        private readonly DatabaseSettings _databaseSettings;

        public ValuesController(IOptions<DatabaseSettings> da
tabaseSettings)
        {
            _databaseSettings = databaseSettings.Value;
        }
    }
```

IOptions<> is singlton so if value is changed when app is running it will not reflect

To use latest value we can use IOptionsSnapshot<>

To get notified about updates we can use IOptionsMonitor<>

In which we set OnChange method

> Don’t set OnChange in controller as controller as scoped so method will be collected by GC we can create a different class and register it as singleton and use it once Program.cs so instance of it is created.

Routing

ASP.NET Core support two type of routing

1. Conventional routing
2. Attribute based routing

Conventional routing

- Routes are **defined centrally** (usually in `Program.cs`)
- Controllers/actions are matched using **URL patterns**
- Follows **naming conventions**

Advantages

- Centralized route configuration
- Clean for **simple CRUD apps**
- Easy to change route pattern globally

Disadvantages

- Hard to understand routes by just looking at controller
- Not ideal for REST APIs
- Less flexible for custom URLs

Ex:

`?` defines id as optional parameter

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

We can call MapControllerRoute in case if there is multiple matching patter the pattern which appear before will be considered

```
app.MapControllerRoute(
                name: "value",
                pattern: "api/value/{action}/{id:int?}",
                defaults: new { controller = "values" }
                );

app.MapControllerRoute(
                name: "valueMultiple",
                pattern: "api/value/{action}/{*path}",
                defaults: new { controller = "values" }
                );
```

We can use constraints in parameter

```
app.MapControllerRoute(
    name: "product",
    pattern: "products/{id:int}");
```

We can give default value to parameter

```
pattern: "{controller=Home}/{action=Index}/{id=1}"
```

To catch all parameters

```
pattern: "docs/{*path}"
```

If we are not using {controller} or {action } in pattern we must give defaults

```
app.MapControllerRoute(
                name: "valueMultiple",
                pattern: "api/value/{action}/{*path}",
                defaults: new { controller = "values" }
                );
```

`MapDefaultControllerRoute` is same as

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

Attribute based routing

Attribute Routing is where we define the **route directly on the controller or action** using attributes like:

- `[Route("...")]`
- `[HttpGet("...")]`
- `[HttpPost("...")]`, `[HttpPut("...")]`, `[HttpDelete("...")]`

Instead of relying on **central route templates** (`MapControllerRoute`), the route is **self-contained** with the action.

Advantages

- Very **clear and readable**
- Best for **RESTful APIs**

- Fine-grained control over URLs
- Supports versioning easily

Disadvantages

- Routes are scattered across controllers
- Large apps may become harder to refactor

To use it we must use `app.MapControllers();` in Program.cs

To define route for entire controller

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

controller will be replaced by controller name

To make get api

```
[HttpGet("GetData")]
```

To use parameter

```
[HttpGet("{id}")]
```

To give default value

```
[HttpGet("[action]/{id=5}")]
```

To use constraints

```
[HttpGet("getiint/{id:int:min(4):max(7)}")]
```

Other constrains

Constraint	Example	Matches
`int`	`{id:int}`	`/123`
`bool`	`{flag:bool}`	`/true` or `/false`
`datetime`	`{date:datetime}`	`/2026-01-21`
`decimal`	`{price:decimal}`	`/12.5`
`double`	`{value:double}`	`/3.14`
`long`	`{id:long}`	`/1234567890`
`min(value)`	`{id:int:min(1)}`	id ≥ 1
`max(value)`	`{id:int:max(100)}`	id ≤ 100
`range(min,max)`	`{id:int:range(1,10)}`	1 ≤ id ≤ 10
`length(n)`	`{name:length(5)}`	exact 5 characters
`minlength(n)`	`{name:minlength(3)}`	≥3 characters
`maxlength(n)`	`{name:maxlength(10)}`	≤10 characters
`alpha`	`{letter:alpha}`	only letters
`regex(pattern)`	`{code:regex(^[A-Z]{3}\d{3}$)}`	e.g., ABC123

To ignore controller's route use ~ or /

```
[Route("~/")]  // Root of application /
[Route("/home")] // /home
Route("/home/index")] // /home/index
public IActionResult Home()
{
         return Ok("HOME");
}
```

Minimal API

Minimal APIs are a **lightweight way** to build HTTP APIs in ASP.NET Core with *minimal code and configuration*.
They let you define routes and handlers **without controllers** or boilerplate.
Recommended for **fast HTTP APIs**, microservices, prototypes, and small services.

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/hello", () =>
            {
                return Results.Ok();
            });
app.Run();
```

Minimal APIs bind route / query / body parameters automatically by *type and name*.
Ex:

code

```
app.MapGet("/products/{id}", (int id) => …);
```

Minimal api support dependency injection

```
builder.Services.AddSingleton<IProductService, ProductService
>();
app.MapGet("/products", (IProductService service) => service.
GetAll());
```

To add route constraints

```
app.MapGet("/products/{id:int}", (int id) => ProductService.G
et(id));
```

Binding

Binding is how the framework takes data from an **HTTP request** and turns it into **C# objects** you can use.

Binding Rules

1. **Route parameters** → automatically bound if names match.
2. **Query parameters** → automatically bound if names match.
3. **Request body** → only **one parameter** can be inferred from body. Use `[FromBody]` if needed.
4. **Services** → automatically bound if type is registered in DI. Use `[FromServices]` for clarity.

```
app.MapPost("/books/{id}",
    ([FromRoute] Guid id,
     [FromQuery] string tag,
     [FromBody] Book book,
     [FromServices] IBookRepository repo) =>
{
    repo.AddBook(book);
    return Results.Ok(new { id, tag, book });
});
```

Priority if same name is there

- **Route values** (`[FromRoute]`) – values in the URL path take the highest priority.
- **Query string** (`[FromQuery]`) – next in line.

- **Request body** (`[FromBody]`) – only **one inferred body parameter** allowed.
- **Form data** (`[FromForm]`) – usually for HTML forms.
- **Header** (`[FromHeader]`) – only if explicitly specified.
- **Services** (`[FromServices]`) – explicitly injected services, not part of HTTP request.

Validation

validation means checking the incoming request data is **correct + safe** before processing it

asp.net provide System.ComponentModel.DataAnnotations to enforce validation rules

In DTO we can write rules like this

```
using System.ComponentModel.DataAnnotations;

public class Book
{
    public Guid Id { get; set; }

    [Required]
    public string Title { get; set; }

    [MinLength(3)]
    public string Author { get; set; }

    [Range(1, 10000)]
    public int Price { get; set; }
}
```

In controller [ApiController] automatically validates and returns 400

In minimal api we have to do it manually

To validate in minimal api install nuget package `MiniValidation`

```
app.MapPost("/books", (Book book) =>
{
    if (!MiniValidation.MiniValidator.TryValidate(book, out v
ar errors))
        return Results.ValidationProblem(errors);

    return Results.Ok(book);
});
```

Other avaliable options

Attribute	Use / Meaning
`[Required]`	Value must not be null/empty
`[MinLength(n)]`	Minimum length for string/array/list
`[MaxLength(n)]`	Maximum length for string/array/list
`[StringLength(max)]`	Max length (optionally min too)
`[Range(min, max)]`	Numeric range validation
`[EmailAddress]`	Must be a valid email
`[Phone]`	Must be a valid phone number
`[Url]`	Must be a valid URL
`[RegularExpression("pattern")]`	Must match regex pattern
`[Compare("OtherProperty")]`	Must match another property (password confirm)
`[CreditCard]`	Valid credit card format

We can also create filter to validate every request

```
using MiniValidation;

namespace MinimalAPIDemo.Filters
{
    public class GlobalValidationFilter : IEndpointFilter
    {
        public async ValueTask<object?> InvokeAsync(EndpointF
ilterInvocationContext context, EndpointFilterDelegate next)
        {
            foreach(var arg in context.Arguments)
            {
                if (arg is null) continue;

                if (!MiniValidator.TryValidate(arg, out var e
rrors))
                {
                    return Results.ValidationProblem(errors);
                }
            }

            return await next(context);
        }
    }
}
```

To use it

```
app.MapPost("/createBook", BookEndpoints.CreateBook).AddEndpo
intFilter<GlobalValidationFilter>();
```

Filters

In minimal API we can use `IEndpointFilter` to create Endpoint filter
Endpoint Filters are a way to run code **before or after a Minimal API endpoint executes**. They act as a lightweight, per-endpoint pipeline, similar to middleware but **specific to an endpoint**.

- **Purpose**:
  - Validation
  - Logging
  - Authorization (custom checks)
  - Exception handling
  - Modifying request/response
- **Key Features**:
  1. Can be applied **per endpoint**.
  2. Can short-circuit the pipeline (stop execution and return a response).
  3. Can access **route parameters, query, body, services**.
  4. Runs **after global middleware** but **before endpoint logic**.
  5. Can be **reused** across multiple endpoints.

To create a endpoint filter

```
public class GlobalValidationFilter : IEndpointFilter
    {
        public async ValueTask<object?> InvokeAsync(EndpointFilterInvocationContext context, EndpointFilterDelegate next)
        {
            foreach(var arg in context.Arguments)
            {
                if (arg is null) continue;
                if (!MiniValidator.TryValidate(arg, out var e
rrors))
                {
                    return Results.ValidationProblem(errors);
                }
            }

            return await next(context);
        }
    }
```

To apply it

```
app.MapPost("/createBook", BookEndpoints.CreateBook).AddEndpo
intFilter<GlobalValidationFilter>();
```

Problem Details

In **ASP.NET Core Web API**, **Problem Details** means a **standard JSON format for error responses**.

It is based on an internet standard called **RFC 7807** (also written as `application/problem+json`).

Ex:

```
{
  "type": "https://tools.ietf.org/html/rfc9110#section-15.5.
1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "name": ["The name field is required."]
  },
  "traceId": "00-3d8c9b..."
}
```

To use it in asp.net core

```
return Results.Problem("Something went wrong", statusCode: 50
0);
return Results.ValidationProblem(errorsDictionary);
```

To use it globally use middleware `UseExceptionHandler()`

Ex:

```
app.UseExceptionHandler("/error");
app.MapGet("/error", (HttpContext httpContext) =>
            {
                var exceptionHandler = httpContext.Features.G
et<Microsoft.AspNetCore.Diagnostics.IExceptionHandlerFeature>
();
                var exception = exceptionHandler?.Error;

                return Results.Problem(
                    detail: exception?.Message,
                    title: "An unexpected error occur",
                    statusCode: 500,
                    instance: httpContext.Request.Path
                    );
            });
```

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.6.
1",
  "title": "An unexpected error occur",
  "status": 500,
  "instance": "/error"
}
```

```
app.MapGet("/validationerror", () =>
            {
                var errors = new Dictionary<string, string[]>
                {
                    { "Name", new[] { "Name is required." }
},
                    { "Age", new[] { "Age must be > 0." } }
                };

                return Results.ValidationProblem(errors);
            });
```

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.
1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "Name": [
      "Name is required."
    ],
    "Age": [
      "Age must be > 0."
    ]
  }
}
```

Rate Limiting

Rate limiting is a technique to **control how many requests a client can make to your API in a specific time period**. It protects your application from overload, abuse, and bad actors (like bots or DoS attacks).

Why Use It?

- **Preventing Abuse**: Rate limiting helps protect an app from abuse by limiting the number of requests a user or client can make in a given time period. This is particularly important for public APIs.
- **Ensuring Fair Usage**: By setting limits, all users have fair access to resources, preventing users from monopolizing the system.
- **Protecting Resources**: Rate limiting helps prevent server overload by controlling the number of requests that can be processed, thus protecting the backend resources from being overwhelmed.
- **Enhancing Security**: It can mitigate the risk of Denial of Service (DoS) attacks by limiting the rate at which requests are processed, making it harder for attackers to flood a system.
- **Improving Performance**: By controlling the rate of incoming requests, optimal performance and responsiveness of an app can be maintained, ensuring a better user experience.
- **Cost Management**: For services that incur costs based on usage, rate limiting can help manage and predict expenses by controlling the volume of requests processed.

Types of rate limiting

Limiter	What It Does
Fixed Window	Allows X requests per time window (e.g., 100 requests per minute).
Sliding Window	A rolling window — smoother control than fixed.
Token Bucket	Replenishes “tokens” over time — good for bursts.
Concurrency Limiter	Limits how many requests can be *handled at the same time*.

When to use which rate limiter

1. Fixed window

- Simple public APIs
- CRUD APIs (GET/POST/PUT/DELETE)

Advantages: Easy to implement, low overhead,

Disadvantages: Burst problem at boundary

2. Sliding window

- APIs where smooth traffic matters
- Payment APIs
- Search APIs

Advantages: smoother than fixed window, prevents boundary issue

Disadvantages: More memory/CPU than fixed window

3. Token bucket

- APIs that allow **bursts sometimes**
- Mobile apps (users may retry quickly)
- Real-time apps
- Chat / messaging APIs
- Upload APIs

Advantages: Allows controlled burst

Disadvantages: If configured wrong can allow big burst

4. Concurrency

→ Best for heavy endpoints

- report generation
- PDF export
- ML prediction
- file processing
- DB-heavy queries

Advantages: Directly protects server resources

Disadvantages: Can exploit fast endpoints

To add fixed window rate limiter

```
builder.Services.AddRateLimiter(options =>
            {
                options.AddFixedWindowLimiter("fixed", limiterOptions =>
                {
                    limiterOptions.PermitLimit = 5;
                    limiterOptions.Window = TimeSpan.FromMinutes(2);
                    limiterOptions.QueueLimit = 2;
                    limiterOptions.QueueProcessingOrder = System.Threading.RateLimiting.QueueProcessingOrder.OldestFirst;
                });
            });
```

To add sliding window rate limiter

```
options.AddSlidingWindowLimiter("sliding", limiterOptions =>
                {
                    limiterOptions.PermitLimit = 6;
                    limiterOptions.SegmentsPerWindow = 2;
                    limiterOptions.Window = TimeSpan.FromMinu
tes(1);
                    limiterOptions.QueueLimit = 2;
                    limiterOptions.QueueProcessingOrder = Sys
tem.Threading.RateLimiting.QueueProcessingOrder.OldestFirst;
                });
```

To add token bucket rate limiter

```
options.AddTokenBucketLimiter("tokenBucket", limiterOptions =
>
                {
                    limiterOptions.TokenLimit = 10;
                    limiterOptions.AutoReplenishment = true;
                    limiterOptions.TokensPerPeriod = 5;
                    limiterOptions.ReplenishmentPeriod = Time
Span.FromSeconds(30);
                    limiterOptions.QueueLimit = 2;
                    limiterOptions.QueueProcessingOrder = Sys
tem.Threading.RateLimiting.QueueProcessingOrder.OldestFirst;
                });
```

To add concurrency rate limiter

```
options.AddConcurrencyLimiter("concurrency", limiterOptions =
>
```

```
                {
                    limiterOptions.PermitLimit = 2;
                    limiterOptions.QueueLimit = 0;
                    limiterOptions.QueueProcessingOrder = Sys
tem.Threading.RateLimiting.QueueProcessingOrder.OldestFirst;
                });
```

Add middleware

```
app.UseRateLimiter();
```

To add rate limiter to a specific endpoint

```
app.MapGet("/api/resource", () => "This endpoint is rate limi
ted")
   .RequireRateLimiting("fixed"); // Apply specific policy to
an endpoint
```

To enable globally

```
app.MapControllers().RequireRateLimiting("fixed");
```

To add limiter per ip

```
options.AddPolicy("fixed_per_ip", context =>
                {
                    var clientIp = context.Connection.RemoteI
pAddress?.ToString() ?? "Unknown";
```

```
                    return RateLimitPartition.GetFixedWindowL
imiter(clientIp, key =>
                    {
                        return new FixedWindowRateLimiterOpti
ons
                        {
                            PermitLimit = 2,
                            Window = TimeSpan.FromMinutes(1),
                            QueueLimit = 2,
                            QueueProcessingOrder = QueueProce
ssingOrder.OldestFirst,
                        };
                    });
                });
```

To add global limiter

```
options.GlobalLimiter = PartitionedRateLimiter.Create<HttpCon
text, string>(context =>
                {
                    var ip = context.Connection.RemoteIpAddre
ss?.ToString() ?? "Unknown";

                    return RateLimitPartition.GetFixedWindowL
imiter(ip, _ =>
                        new FixedWindowRateLimiterOptions
                        {
                            PermitLimit = 2,
                            Window = TimeSpan.FromMinutes(1),
                            QueueLimit = 0,
                            QueueProcessingOrder = QueueProce
ssingOrder.OldestFirst,
                        });
                });
```

To set what to do on rejection

```
options.RejectionStatusCode = StatusCodes.Status429TooManyReq
uests;

options.OnRejected = async (context, token) =>
                {
                    context.HttpContext.Response.ContentType
= "application/json";

                    await context.HttpContext.Response.WriteA
sync(
                        """
                        {
                          "message": "Too many requests. Plea
se try again later."
                        }
                        """
                    );
                };
```

Filters

Filters are reusable components that run before or after specific stages of request processing in **MVC / Web API** pipeline.

Types of filter

1. Authorization filter

2. Resource filter
3. Action filter
4. Exception filter
5. Result filter

Execution Order

1. Authorization Filter
2. Resource Filter (before)
3. Model Binding
4. Action Filter (before)
5. Action executes
6. Action Filter (after)
7. Resource Filter (after)
8. Result Filter (before result executes)
9. Result executes (returns response to client)
10. Result Filter (after result executes)
11. Exception Filter (if exception occurs)

1. Authorization filter

- Checks if the **user/request is authorized** to execute the action.
- Can **short-circuit** the pipeline by returning a `context.Result` if authorization fails.

Use case

- JWT token validation
- Role-based access

- API key validation

Ex:

```
public class JwtAuthorizeFilter : IAuthorizationFilter
    {
        public void OnAuthorization(AuthorizationFilterContex
t context)
        {
            // ....
        }
    }
```

2. Resource Filter

- Runs **after authorization**, **before model binding**, and again **after the action executes**.
- Can **pre-process** the request or **short-circuit** before model binding.
- Good for caching, request validation, logging, or global pre/post logic.

Use case:

- Validate required headers or query params
- Short-circuit request early
- Implement caching

Ex:

```
public class LogResourceFilter : IResourceFilter
    {
        public void OnResourceExecuted(ResourceExecutedContex
t context)
        {
            Console.WriteLine($"Resouce filter after : {conte
xt.HttpContext.Request.Path}");
```

```
        }

        public void OnResourceExecuting(ResourceExecutingCont
ext context)
        {
            Console.WriteLine($"Resouce filter before : {cont
ext.HttpContext.Request.Path}");

            // To short circuit it
            //context.Result = new BadRequestObjectResult(new
            //{
            //    message = "Invalid data"
            //});
        }
    }
```

3. Action Filter

- Wraps **action method execution**.
- Runs **before and after the action executes**.
- Can inspect or modify **action arguments** and **results**.
- Can also **short-circuit** using `context.Result`.

Use case:

- Logging requests
- Input validation
- Measuring performance of actions
- Transforming action result

Ex:

```
public class LogActionFilter : IActionFilter
    {
        private Stopwatch _stopwatch = new();
        public void OnActionExecuted(ActionExecutedContext context)
        {
            _stopwatch.Stop();
            Console.WriteLine($"Action '{context.ActionDescriptor.DisplayName}' executed in {_stopwatch.ElapsedMilliseconds} ms");
        }

        public void OnActionExecuting(ActionExecutingContext context)
        {
            _stopwatch.Start();
            Console.WriteLine($"Action '{context.ActionDescriptor.DisplayName}' executing at {DateTime.Now}");
        }
    }
```

4. Exception Filter

- Runs **only when an exception occurs** in action or action filters.
- Can handle the exception, return a custom result, and **prevent propagation** using `context.ExceptionHandled = true`.

Use case:

- Return **custom JSON errors**
- Centralized exception logging
- Transform exceptions into user-friendly messages

Ex:

```
public class ExceptionFilter : IExceptionFilter
    {
        public void OnException(ExceptionContext context)
        {
            var ex = context.Exception;
            Console.WriteLine($"Exception: {ex.Message}");

            context.Result = new ObjectResult(new
            {
                message = "Something went wrong!",
                error = ex.Message
            })
            {
                StatusCode = 500
            };

            context.ExceptionHandled = true;
        }
    }
```

5. Result filter

- Wraps the **execution of the action result**.
- Runs **before and after result execution** (like before `Ok()`, `Json()`, `View()`).
- Cannot inspect model binding; works with **result objects**.

Use case:

- Add custom headers
- Wrap API responses in a standard format
- Log responses

Ex:

```
public class HeaderAdderResultFilter : IResultFilter
    {
        public void OnResultExecuted(ResultExecutedContext co
ntext)
        {
            if (context.Result is ObjectResult objectResult)
            {
                string json = JsonSerializer.Serialize(object
Result.Value);
                Console.WriteLine($"Action Result executed:
{json}");
            }
            else
            {
                Console.WriteLine("Action Result executed, bu
t not an ObjectResult");
            }
        }

        public void OnResultExecuting(ResultExecutingContext
context)
        {
            context.HttpContext.Response.Headers.Add("X-Versi
on", "1");
            Console.WriteLine("Header added");
        }
    }
```

Ways to apply filters

1. Apply globally

```
builder.Services.AddControllers(options =>
            {
                options.Filters.Add<LogResourceFilter>();
                options.Filters.Add<LogActionFilter>();
                options.Filters.Add<ExceptionFilter>();
            });
```

2. Apply at controller level

```
[ServiceFilter(typeof(HeaderAdderResultFilter))]
```

3. Apply at action level

```
[ServiceFilter(typeof(HeaderAdderResultFilter))]
```

4. Using TypeFilter we can pass argument to constructor

```
[TypeFilter(typeof(RoleFilter), Arguments = new object[] { "A
dmin" })]
```

Complete flow

> HTTP Request → Kestrel → Middleware 1 → Middleware 2 → ... → Routing → Endpoint selection
> → Authorization Filters → Resource Filters → Model Binding → Validation → Action Filters
> → Action Execution → Result Filters → Result Execution → Response Middleware → Kestrel → HTTP Response

AOT

NativeAOT = Ahead-of-Time compilation to native machine code

NativeAOT compiles .NET apps into a self-contained native executable at publish time, without requiring a JIT compiler at runtime.

So instead of

C# → IL → JIT → Native (runtime)

We do

C# → Native executable (at publish time)

Difference between build and publish

Feature	`dotnet build`	`dotnet publish`
Purpose	Compile & check for errors	Compile + prepare for deployment
Output	Only DLLs/EXEs in `bin` folder	DLLs/EXEs + dependencies + config in `publish` folder
Includes dependencies	No (only references)	Yes (all required files)
Ready to run/deploy	No	Yes
Use in Docker	Rarely (you'd still need publish)	Yes (copy `publish` folder into container)

Advantages of AOT

- Very fast startup
- Low memory usage
- Single native executable

Disadvantages of AOT

- No JIT at runtime
- Limited reflection
- No dynamic code generation

How it works

1. C# code
2. Roslyn compiler → IL
3. Trimmer runs (remove unused code)
4. AOT compiler analyzes reachable code
5. Native machine code generated
6. OS-specific executable produced

Assumption by NativeAOT

“All code that will ever execute is known at publish time”

So

```
Type.GetType("MyClass");
```

- Type may not exist at runtime
- Trimmer might remove it
- AOT compiler never compiled it

What NativeAOT removes

- JIT compiler

- IL interpreter
- Dynamic code generation
- Reflection emit

What works well

- Minimal APIs
- Simple controllers
- JSON serialization (System.Text.Json)
- Kestrel
- Middleware pipeline

What struggles

- Heavy reflection
- MVC view engines

What improves

- Startup time (huge)
- Memory usages
- Predictable latency

What does NOT improve

- Long-running throughput (JIT may outperform)
- Heavy CPU-bound tasks

EF Core relies heavily on reflection, runtime code generation, and open-world assumptions — all of which conflict with NativeAOT's closed-world model. so EF Core may struggle with NativeAOT

How to enable NativeAOT

```
<PropertyGroup>
  <PublishAot>true</PublishAot>
  <TrimMode>link</TrimMode> <!-- optional but recommended -->
  <RuntimeIdentifier>win-x64</RuntimeIdentifier> <!-- or linu
x-x64 -->
</PropertyGroup>
```

Trimming

Trimming = **analyzing your code and removing parts of assemblies that are never used**.

- Removes **unused methods, properties, classes, and even whole assemblies**
- Reduces **app size** and **startup memory**
- Mandatory for **NativeAOT**, because AOT requires a **closed-world model**

How Trimming Works Internally

Trimming is like a **static analyzer / linker**:

1. Starts at **entry points** (Main method, public APIs)
2. Builds a **call graph** of all reachable code
3. Marks **used methods, types, properties**
4. Removes **everything else**

Trim mode

Mode	Description
link	Aggressive trimming, smaller size, riskier
copyused	Safer, trims unused assemblies only

To preserve some property from trimmer

```
app.MapGet("/user/{id}", ([DynamicallyAccessedMembers(Dynamic
allyAccessedMemberTypes.PublicProperties)] User user) =>
{
    return user.Name;
});
```

- Preserves all public properties of `User` even if trimmer wants to remove them.
- Required if `User` is deserialized dynamically, e.g., from JSON.

We use this attribute where we use dynamic type not where we define it.

Profile guided optimization

PGO = a compiler/runtime technique that **uses runtime profiling data to optimize code generation**.

- Instead of blindly compiling IL → machine code, .NET **observes how the app is actually used**.
- Then it **reorders code, inlines hot paths, and optimizes branches** based on real usage.

How PGO Works in .NET

PGO has two phases:

1 Instrumentation Phase

- The app is compiled with instrumentation enabled
- When you run your app normally, it collects runtime data:
    - Hot methods
    - Frequently taken branches

    - Frequently executed loops
- Generates a profile file (`.pgc` or similar)

2 Optimization Phase

- Use the profile to recompile the app or the native binary
- The compiler:
    - Inlines hot methods
    - Reorders branches for better CPU prediction
    - Precompiles frequently executed loops
    - Improves startup time and throughput

Types of PGO in .NET

1 JIT PGO (Dynamic)

- Available in .NET Core 3.1+ and .NET 8
- Uses runtime profile at execution time
- Collects data as the JIT compiles IL
- Optimizes hot methods dynamically

2 Crossgen2 / NativeAOT PGO (Static)

- NativeAOT cannot use JIT at runtime
- Use PGO profile at compile/publish time
- Produces a pre-optimized native binary

To use JIT PGO set two environment variable

set DOTNET_JitProfileOptimization=1
set DOTNET_JitProfileOptimizationPath=C:\PGODemoProfiles

To use NativeAOT PGO

1. Set GenerateProfileData to true in .csproj

```
<PropertyGroup>
  <PublishAot>true</PublishAot>
  <RuntimeIdentifier>win-x64</RuntimeIdentifier>
  <GenerateProfileData>true</GenerateProfileData> <!-- Enable
s instrumentation -->
</PropertyGroup>
```

2. Publish the app use and use it to simulate actual workflow
3. Change .csproj to use generated profile data

```
<PropertyGroup>
  <PublishAot>true</PublishAot>
  <UseProfileData>true</UseProfileData>
  <RuntimeIdentifier>win-x64</RuntimeIdentifier>
</PropertyGroup>
```

Benchmark

BenchmarkDotNet is a powerful open-source .NET library to measure and compare performance precisely. It helps you write benchmarks and get reliable performance reports without writing timing logic yourself.

Internal Working — How It Actually Benchmarks

BenchmarkDotNet doesn't just call your method once. Under the hood it does:

1. Warmup phase

It runs the method enough times to allow JIT and CPU to "settle." This avoids measuring cold JIT costs.

2. Iteration selection

It determines how many times to call your method to get stable measurements.

3. Multiple runs

It runs **multiple iterations in multiple processes** and averages results. This avoids side-effects like GC pauses.

4. Stats & output

It calculates mean, error, standard deviation — and outputs detailed summary tables.

How to use it

1. Install nuget package : BenchmarkDotNet
2. Define Benchmark class

```
public class LinqVsLoopBenchmark
    {
        private List<int> numbers;

        [GlobalSetup]
        public void Setup()
        {
            numbers = Enumerable.Range(1, 10000).ToList();
        }

        [Benchmark]
        public int ForLoop()
        {
            int count = 0;
            for (int i = 0; i < numbers.Count; i++)
            {
                if (numbers[i] > 5000)
                {
                    count++;
                }
            }
            return count;
        }

        [Benchmark]
        public int Linq()
        {
            return numbers.Count(x => x > 5000);
        }
    }
```

[GlobalSetup] : Runs **once** before benchmarking
[Benchmark] : Method to measure

3. Run benchmark

```
BenchmarkRunner.Run<LinqVsLoopBenchmark>();
```

4. Run in release mode

Sample output:

```
| Method  | Mean      | Error    | StdDev    |
|-------- |----------:|---------:|----------:|
| ForLoop |  16.02 us | 0.473 us |  1.388 us |
| Linq    | 169.46 us | 4.395 us | 12.679 us |
```

Behind the scene of Benchmark library

1. Use reflection to find method with attribute like [GlobalSetup], [Benchmark]
2. Code generation

It does not directly run method but generate new c# program at runtime

It ensure no reflection on runtime

3. Process creation

It will create 3 separate process for warmup, measurement, verification

4. Warmup phase

Methods are executed repeatedly to avoid JIT compilation

5. Iteration Calibration

Runs method few time to approximate calculate execution time to determine iteration time

Method is very fast → run 1,000,000 times

Method is slow → run 100 times

6. Measurement phase

Now the *real* benchmarking begins.
Multiple iteration

Use high resolution timers

7. Statistical analysis

Calculate mean, median, stddev, outliner

What output on terminal mean

Log Section	Phase	Purpose
OverheadJitting	Warmup / calibration	Measure harness overhead
WorkloadJitting	Warmup / calibration	Measure method JIT timing
WorkloadPilot	Iteration calibration	Decide optimal iteration counts
OverheadWarmup	Warmup	Warmup harness overhead
OverheadActual	Preparation	Measure exact overhead before measurement
WorkloadWarmup	Warmup	Stabilize CPU, JIT, caches

Log Section	Phase	Purpose
WorkloadActual	Measurement	Actual timing of your method
WorkloadResult	Aggregation / reporting	Final processed results

> Harness overhead is the time **BenchmarkDotNet itself takes** to run the benchmark code, without your actual method doing anything meaningful.

Use [MemoryDiagnoser] to enable memory measurement

To run for all value of N

```
[Params(10, 100, 1000)]
public int N;
```

To run for multiple runtime

```
[SimpleJob(RuntimeMoniker.Net70, baseline: true)]
[SimpleJob(RuntimeMoniker.Net80)]
```

To export result

```
[HtmlExporter]
[CsvExporter]
```

System.text.json

Source generator

What is a Source Generator?

- A **C# compiler feature** (Roslyn)
- Runs **at compile time**
- **Generates C# code automatically**
- No reflection
- No runtime cost

Why System.Text.Json Uses Source Generators

Default `System.Text.Json`:

- Uses **reflection** at runtime
- Reflection = slower
- allocations
- trimming/AOT unfriendly

Source generator:

- Generates **serialization code at compile time**
- Faster
- Less memory
- NativeAOT & trimming safe

Default behavior of System.text.json

Serialization

1. JsonSerializer.Serialize(obj)
2. Inspect object type using reflection
3. Discover properties & metadata

4. Read property values via reflection
5. Write JSON tokens

Deserialization

1. JsonSerializer.Deserialize<T>(json)
2. Inspect target type T using reflection
3. Create object instance dynamically
4. Match JSON properties to .NET properties
5. Set values via reflection

Behavior of source generator

1. Detection: It detect `[JsonSerializable(typeof(T))]` and register it
2. Code generation: At compile time code(serialize and deserialize) will be generated for all marked classes
3. Usage: When we use context in serialize it use pre generated methods instead of reflections

How to use it

1. Create context class for intended model

```
using System.Text.Json.Serialization;

[JsonSerializable(typeof(User))]
public partial class AppJsonContext : JsonSerializerContext
{
}
```

2. Use context class while serializing it

```
var deserializedUser =
    JsonSerializer.Deserialize(
        json,
        AppJsonContext.Default.User
    );
```

Polymorphism

If base class reference holding derived type object and we try to serialize/deserialize it only property of base class is processed

So to serialize/deserialize properly we set attribute to inform serializer about derived types

`[JsonPolymorphic]`

To enable polymorphism

Ex:

```
[JsonPolymorphic(TypeDiscriminatorPropertyName = "$type")]
public abstract class Animal
{
    public string Name { get; set; }
}
```

`TypeDiscriminatorPropertyName` define discriminator property name default is `$type`

`[JsonDerivedType]`

`JsonDerivedType` **maps a derived CLR type to a discriminator value.**

When $type is dog, use Dog class.

Ex:

```
[JsonDerivedType(typeof(Dog), "dog")]
[JsonDerivedType(typeof(Cat), "cat")]
public abstract class Animal
{
    public string Name { get; set; }
}
```

In case of multi level inheritance use these attributes to class which will be used in serialize and deserialize function

Internal Working

Serialization Flow

1. Runtime object type is detected
2. Serializer checks:
    - Is polymorphism configured?
3. Writes:
    - Discriminator
    - Derived properties

Deserialization Flow

1. JSON read
2. Discriminator value found
3. Matches discriminator → derived type
4. Creates correct object
5. Populates properties

Why it is not enabled by default

Because of security aspects with deserialization

While deserializing constructor run attacker might try to use it to exploit system by injecting code in it

Attacker might try to send extremely deep object which cause memory exhaustion or system crash

UTF8JsonReader/Writer

System.text.json

- Built-in JSON library in .NET (Core 3.0+)
- Faster & more memory-efficient than Newtonsoft.Json
- Designed for:
  - **Span-based**
  - **UTF-8 first**
  - **Low allocation**

`Utf8JsonReader`

- **Forward-only**
- **Read-only**
- **High-performance**
- Works on `ReadOnlySpan<byte>` (UTF-8 bytes)
- Does **not allocate strings** unless you ask for them

`Utf8JsonWriter`

- **Forward-only**
- **Write-only**
- Writes UTF-8 JSON directly to:
    - `IBufferWriter<byte>`
    - `Stream`
- No intermediate DOM objects

When should we use `Utf8JsonReader` instead of `JsonSerializer`?

Use it when:

- Large JSON files
- High throughput APIs
- Partial JSON parsing

Why is `Utf8JsonWriter` faster than serializing objects?

- No reflection
- No property discovery
- No intermediate object graph
- Writes raw UTF-8 bytes

To write json using Utf8JsonWriter

```
using var stream = new MemoryStream();
            var writer = new Utf8JsonWriter(stream, new JsonW
riterOptions
            {
                Indented = true,
            });
            writer.WriteStartObject();

            writer.WriteNumber("id", 1);
            writer.WriteString("name", "Dakshil");

            writer.WriteEndObject();

            writer.Flush();

            string json = Encoding.UTF8.GetString(stream.ToAr
ray());
```

To create array inside it

```
writer.WriteStartArray("skills");
writer.WriteStringValue("C#");
writer.WriteStringValue(".NET");
writer.WriteStringValue("ASP.NET Core");
writer.WriteEndArray();
```

To create nested object

```
writer.WriteStartObject("address");
writer.WriteString("street", "a");
writer.WriteString("city", "b");
writer.WriteString("state", "c");
writer.WriteEndObject();
```

To read json

```
ReadOnlySpan<byte> data = Encoding.UTF8.GetBytes(json);
    var reader = new Utf8JsonReader(data);
    while (reader.Read())
    {
        switch (reader.TokenType)
        {
            case JsonTokenType.PropertyName:
                Console.Write($"Property: {reader.GetString
()} → ");
                break;

            case JsonTokenType.String:
                Console.WriteLine(reader.GetString());
                break;

            case JsonTokenType.Number:
                Console.WriteLine(reader.GetInt32());
                break;
        }
    }
```