.NET

Week 11 - .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_Connection=Tr
ue;"
  },

  "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 services)
      {
         services.AddControllers();
         services.AddSwaggerGen();
      }
```

```
        // Configure HTTP pipeline
        public void Configure(WebApplication app, IWebHostEnvironment 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(args);

                var startup = new Startup(builder.Configuration);
                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

Middleware	Purpose
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**

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 request : {context.
Request.Method} {context.Request.Path}");

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

         Console.WriteLine($"Logging middleware outgoing request : {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 IApplicationBuilder build
er)
      {
         return builder.UseMiddleware<LoggingMiddleware>();
      }
   }
```

To use it

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

3. Shorthand method

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

            await next();

            Console.WriteLine($"Response: {context.Response.StatusCode}");
         });
```

4. For specific route

```
app.UseWhen(context ⇒ context.Request.Path == "/WeatherForecast/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**

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, AppSettingsService>();
```

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>();
```

Dependency is injected 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());
    }
}
```

What is 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
- Background services
- 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.Configuration.GetSecti
on("DatabaseSettings"));
```

4. Inject it from controller

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

      public ValuesController(IOptions<DatabaseSettings> databaseSettings)
      {
         _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.Get(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 var 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(EndpointFilterInvocationC
ontext context, EndpointFilterDelegate next)
        {
            foreach(var arg in context.Arguments)
            {
                if (arg is null) continue;

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

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

To use it

```
app.MapPost("/createBook", BookEndpoints.CreateBook).AddEndpointFilter<
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(EndpointFilterInvocationC
ontext context, EndpointFilterDelegate next)
      {
         foreach(var arg in context.Arguments)
         {
            if (arg is null) continue;

            if (!MiniValidator.TryValidate(arg, out var errors))
            {
```

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

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

To apply it

```
app.MapPost("/createBook", BookEndpoints.CreateBook).AddEndpointFilter<
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: 500);
return Results.ValidationProblem(errorsDictionary);
```

To use it globally use middleware `UseExceptionHandler()`

Ex:

```
app.UseExceptionHandler("/error");
app.MapGet("/error", (HttpContext httpContext) ⇒
        {
            var exceptionHandler = httpContext.Features.Get<Microsoft.AspNet
Core.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."
    ]
  }
}
```