.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);

         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