.NET Core

Version .NET 9 · verified 2026-07-29

A practical guide to modern .NET backend concepts — Minimal APIs, DI, configuration, middleware, and logging.

Contents

Mental model

An ASP.NET Core application starts with a builder that sets up configuration, dependency injection, and logging. Building it creates a host, which configures a middleware pipeline to process every incoming HTTP request. Modern Minimal APIs eliminate controllers, routing directly to functions and injecting dependencies on demand.

Minimal APIs & routing

API	Use
`app.MapGet(path, fn)`	handle GET requests
`app.MapPost(path, fn)`	handle POST requests
`Results.Ok(v)`	HTTP 200 with JSON body
`Results.NotFound()`	HTTP 404 response
`[FromQuery]`	bind from URL query string

Minimal APIs map HTTP verbs directly to delegates. Route parameters are parsed automatically, and services are injected into the delegate parameters if they are registered in the container.

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

app.MapGet("/users/{id}", (int id) =>
{
  if (id <= 0) return Results.BadRequest();
  return Results.Ok(new { Id = id, Name = "Sam" });
```

```
});


app.Run();
```

> **Gotcha:** return types must be `IResult` to control the status code. Returning an object directly always results in a 200 OK.

Dependency injection lifetimes

Lifetime	Meaning
`AddTransient`	new instance every time
`AddScoped`	one instance per HTTP request
`AddSingleton`	one instance forever
`IServiceProvider`	the resolved container

Services are registered on `builder.Services`. Do not do expensive work in constructors, as dependencies may be created frequently. Avoid capturing a scoped service inside a singleton, as it forces the scoped service to live forever.

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


app.MapGet("/users", (IUserService svc) =>
{
```

```
  return svc.GetAll();
});
```

> **Warning:** injecting a scoped service (like EF Core's `DbContext`) into a singleton (like a background worker) throws `InvalidOperationException` at runtime.

Configuration

API	Use
`appsettings.json`	default config file
`builder.Configuration`	reads combined settings
`GetValue<T>(key)`	read single typed value
`GetSection(key)`	read nested section

Configuration is built by layering sources: appsettings, environment variables, and command-line arguments. Later sources override earlier ones. Use the Options pattern to bind sections to strongly typed classes.

```
// { "Api": { "Key": "abc" } }
var key = builder.Configuration["Api:Key"];


app.MapGet("/config", (IConfiguration cfg) =>
{
  var max = cfg.GetValue<int>("MaxItems");
```

```
  return new { Max = max };
});
```

> **Gotcha:** `IConfiguration` returns `null` if a key does not exist, but `GetValue<T>` returns `default(T)` — an unset `GetValue<int>` becomes `0`, not an error.

The Options pattern

Interface	Use
`IOptions<T>`	singleton, reads config once
`IOptionsSnapshot<T>`	scoped, updates on change
`IOptionsMonitor<T>`	singleton, alerts on change

Binding configuration to a C# record or class provides type safety and removes magic strings from the codebase. Register it using `Configure<T>`.

```
builder.Services.Configure<AppOptions>(
  builder.Configuration.GetSection("App"));

app.MapGet("/settings",
  (IOptions<AppOptions> opts) =>
{
  return opts.Value.Title;
});
```

```
class AppOptions { public string Title = ""; }
```

Middleware pipeline

API	Use
`app.Use(fn)`	adds middleware, calls `next`
`app.Run(fn)`	terminal middleware, no `next`
`HttpContext`	request and response data
`next(context)`	passes control to next step

Middleware processes the `HttpContext` sequentially. Order matters: if `UseAuthentication` comes after `UseAuthorization`, authorization fails because the user is not yet authenticated.

```
app.Use(async (context, next) =>
{
  var start = DateTime.UtcNow;
  await next(context); // let the rest run
  var duration = DateTime.UtcNow - start;
  Console.WriteLine(duration.TotalMilliseconds);
});

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

> **Gotcha:** writing to the response body locks the response headers. Trying to set a header or status code after writing the body throws an exception.

Logging

Level	Purpose
`Trace`	diagnostic details, noisy
`Debug`	for local development
`Information`	business flows (default)
`Warning`	recoverable issues
`Error` / `Critical`	failures requiring attention

Inject `ILogger<T>` into services or endpoints. .NET logging is structured — log message templates capture parameter values separately from the message string, allowing log aggregators to query on specific variables.

```
app.MapGet("/divide", (ILogger<Program> log) =>
{
  int x = 10;
  int y = 0;
  log.LogInformation("Dividing {X} by {Y}", x, y);
  return x / y;
});
```

> **Tip:** use semantic names in the log template (like `{UserId}`), not string interpolation (`$"User {id}"`), so external log sinks can index the parameter by its name.

Further reading

- ASP.NET Core fundamentals
- Minimal APIs overview
- Dependency injection in ASP.NET Core
- Configuration in ASP.NET Core