ASP.NET Core

Introduction to ASP.NET Core
ASP.NET Core Project Structure
Routing and Controllers
Razor Pages & Views
Entity Framework Core Basics
Bibliography

ASP.NET Core Programming: Beginner

(Integrative-Generative AI Edition)

Student Price Book Center

คำนำ

การพัฒนาเว็บแอปพลิเคชันสมัยใหม่จำเป็นต้องอาศัยเทคโนโลยีที่มีประสิทธิภาพและสามารถรองรับการทำงานข้ามแพลตฟอร์ม **ASP.NET Core** ถือเป็นหนึ่งในเฟรมเวิร์กที่ตอบโจทย์เหล่านี้ได้อย่างครบถ้วน ด้วยสถาปัตยกรรมที่เบาแต่ทรงพลัง การจัดการ request-response pipeline อย่างมีประสิทธิภาพ และการสนับสนุนฟีเจอร์สมัยใหม่ของ .NET 6/7/8 ทำให้ ASP.NET Core เป็นตัวเลือกยอดนิยมสำหรับนักพัฒนาเว็บที่ต้องการสร้างแอปพลิเคชันแบบ dynamic, secure, และ scalable

หนังสือเล่มนี้ถูกออกแบบมาสำหรับผู้เริ่มต้นที่ต้องการเรียนรู้ ASP.NET Core ตั้งแต่พื้นฐานไปจนถึงแนวปฏิบัติจริง โดยเริ่มจาก **บทที่ 1 – Introduction to ASP.NET Core** ซึ่งอธิบายประวัติและวิวัฒนาการของ ASP.NET, เจาะลึก ASP.NET Core กับ .NET 6/7/8, เครื่องมือและ IDE/Editor ที่นิยมใช้, การติดตั้งและตั้งค่าเครื่องมือพัฒนา รวมถึงการสร้างและรันโปรเจกต์ตัวแรกด้วย dotnet run พร้อมตัวอย่างโปรแกรมพื้นฐานเพื่อสร้างความเข้าใจตั้งแต่เริ่มต้น

ต่อเนื่องด้วย **บทที่ 2 – ASP.NET Core Project Structure** ผู้อ่านจะได้เรียนรู้โครงสร้างโปรเจกต์ ASP.NET Core อย่างละเอียด เช่น โฟลเดอร์ wwwroot, Controllers, Models, Views, และ Pages พร้อมแนวคิดของ Middleware และ Request Pipeline นอกจากนี้ยังอธิบายการใช้งาน **Dependency Injection (DI)** เบื้องต้น และการตั้งค่าผ่านไฟล์ appsettings.json ซึ่งเป็นพื้นฐานสำคัญสำหรับการสร้างแอปพลิเคชันที่มีโครงสร้างชัดเจน

บทที่ 3 – Routing and Controllers และ **บทที่ 4 – Razor Pages & Views** มุ่งเน้นไปที่การสร้าง logic และ UI ของเว็บแอปพลิเคชัน บทเรียนเหล่านี้ครอบคลุมการกำหนด Route ทั้งแบบ Attribute Routing และ Convention-based, การสร้าง Controller และ Action, การส่งข้อมูลจาก Controller ไปยัง View, การใช้ Razor Syntax, Layout, Partial View, รวมถึง ViewData, ViewBag, TempData และการจัดการ Form ด้วย Model Binding ซึ่งทั้งหมดนี้ช่วยให้ผู้เรียนสามารถสร้างเว็บแอปพลิเคชันแบบ dynamic ได้อย่างมีประสิทธิภาพ

สุดท้าย **บทที่ 5 – Entity Framework Core Basics** จะพาผู้อ่านเข้าสู่การจัดการฐานข้อมูลใน ASP.NET Core โดยเริ่มจากการแนะนำ ORM และ EF Core, การเชื่อมต่อฐานข้อมูลเช่น SQL Server และ SQLite, การสร้าง Model และ DbContext, การทำงานกับ Migration และตัวอย่างบูรณาการที่รวมทุกแนวคิดเข้าด้วยกัน ซึ่งช่วยให้ผู้เรียนเข้าใจการประมวลผลข้อมูล, การจัดการ database schema, และการสร้างแอปพลิเคชันที่สามารถขยายตัวได้

หนังสือเล่มนี้เน้นการเรียนรู้แบบ step-by-step และตัวอย่างเชิงปฏิบัติจริง เพื่อให้ผู้อ่านสามารถประยุกต์ใช้ความรู้ที่ได้สร้างเว็บแอปพลิเคชันได้จริง ทั้งยังสร้างความเข้าใจเชิงลึกตั้งแต่โครงสร้างโปรเจกต์, Routing, Razor Pages, Controllers, ไปจนถึง EF Core ซึ่งเป็นพื้นฐานสำคัญของการพัฒนาเว็บแอปพลิเคชันด้วย ASP.NET Core

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 1

Introduction to ASP.NET Core

(Introduction to ASP.NET Core)

เนื้อหา

บทนำ: บทที่ 1 – Introduction to ASP.NET Core

ASP.NET Core เป็นเฟรมเวิร์กสำหรับพัฒนาเว็บแอปพลิเคชันที่ได้รับความนิยมสูงในโลกของ .NET โดยมีรากฐานมาจาก ASP.NET รุ่นดั้งเดิมที่เริ่มต้นด้วย Web Forms ซึ่งเป็นโมเดลการพัฒนาเว็บเชิง UI แบบ event-driven ที่ช่วยให้โปรแกรมเมอร์สามารถสร้างหน้าเว็บแบบ dynamic ได้ง่าย ต่อมา ASP.NET ได้พัฒนาเป็นสถาปัตยกรรม MVC (Model-View-Controller) ที่แยกส่วนของการทำงานอย่างชัดเจน ทำให้สามารถจัดการโค้ดได้สะดวกและเหมาะสมกับการพัฒนาแอปพลิเคชันขนาดใหญ่ การมาถึงของ ASP.NET Core ถือเป็นก้าวสำคัญของวิวัฒนาการนี้ ด้วยความสามารถในการรันบนหลายแพลตฟอร์ม (cross-platform) ประสิทธิภาพสูง และสถาปัตยกรรมที่เบาและยืดหยุ่นมากขึ้น

ในยุคของ .NET 6, .NET 7 และ .NET 8, ASP.NET Core ได้รับการปรับปรุงให้รองรับฟีเจอร์ใหม่ ๆ ทั้งในเรื่องของการทำงานแบบ asynchronous, การจัดการ dependency injection, และการกำหนดค่าโปรเจกต์ที่ง่ายขึ้น สำหรับผู้เริ่มต้น การทำความเข้าใจโครงสร้างโปรเจกต์เป็นเรื่องสำคัญ โดยทั่วไปโปรเจกต์ ASP.NET Core ประกอบด้วยไฟล์หลัก ๆ เช่น Program.cs สำหรับตั้งค่าและรันแอป, Startup.cs สำหรับกำหนดบริการและ middleware, และ appsettings.json สำหรับเก็บค่าการตั้งค่าต่าง ๆ ของแอปพลิเคชัน

บทแรกนี้จะพาผู้อ่านเข้าสู่การสร้างโปรเจกต์ ASP.NET Core แรกของตนเอง เริ่มจากการใช้งานคำสั่ง dotnet run เพื่อรันแอปพลิเคชันพื้นฐาน ทำความเข้าใจกับโครงสร้างไฟล์และ flow การทำงานของแอปฯ เพื่อเป็นพื้นฐานที่แข็งแรงสำหรับบทต่อ ๆ ไปในการพัฒนาเว็บแอปพลิเคชันที่ซับซ้อนและทันสมัย

Introduction to ASP.NET Core

- ประวัติและวิวัฒนาการของ ASP.NET ⟶ Web Forms ⟶ MVC ⟶ Core
- ASP.NET Core กับ .NET 6/7/8
- โครงสร้างโปรเจกต์ (Program.cs, Startup.cs, appsettings.json)
- การรันโปรเจกต์แรกด้วย dotnet run

---

1. ประวัติและวิวัฒนาการของ ASP.NET ⟶ Web Forms ⟶ MVC ⟶ Core

☐ **จุดเริ่มต้นของ ASP.NET**

- ASP.NET ถูกเปิดตัวครั้งแรกพร้อมกับ **.NET Framework 1.0 (ปี 2002)**
- ใช้ร่วมกับ **IIS (Internet Information Services)** ของ Microsoft
- สร้างเว็บแอปแบบ **Server-Side Rendering (SSR)** โดยใช้ **Web Forms**

☐ **ASP.NET Web Forms (2002–2010)**

- ใช้ **Drag & Drop + Event-driven Model** คล้าย Windows Forms
- โค้ดอยู่ในรูปแบบ **Code-Behind (C# หรือ VB.NET)**
- ปัญหา:
  - tightly coupled ระหว่าง UI กับ Logic
  - HTML ที่ถูก Generate ไม่ยืดหยุ่น
  - ทำให้การทำงานกับ Frontend (HTML/CSS/JS) ยุ่งยาก

☐ **ASP.NET MVC (2009–2016)**

- เปิดตัวปี 2009 บน **.NET Framework 3.5/4.0**
- นำ **Model-View-Controller (MVC)** มาใช้ ⟶ Separation of Concerns (SoC)
- รองรับ **Unit Testing** ได้ง่าย
- เปิดให้ Dev เขียน HTML, JavaScript เอง ไม่ต้องพึ่ง Control แบบ Web Forms
- ข้อจำกัด:
  - ยังผูกติดกับ **System.Web.dll** และ IIS
  - ไม่ Cross-platform

☐ **ASP.NET Core (2016–ปัจจุบัน)**

- เปิดตัวพร้อม **.NET Core 1.0 (2016)**
- **Open-source** และ **Cross-platform** (Windows, Linux, macOS)
- ใช้ **Kestrel Web Server** (ไม่พึ่ง IIS โดยตรง แต่ทำงานร่วมได้)
- Modular + Lightweight ⟶ ใช้ NuGet package ตามที่ต้องการ
- Performance สูงกว่า ASP.NET MVC / Web Forms อย่างชัดเจน
- สามารถทำงานได้หลายรูปแบบ:
  - **MVC + Razor Pages**
  - **Web API**
  - **Blazor (WebAssembly/Server)**
  - **Minimal APIs (ตั้งแต่ .NET 6 ขึ้นไป)**

---

2. ASP.NET Core กับ .NET 6/7/8

☐ **.NET 5 เป็นต้นมา ⟶ Unified Platform**

- ก่อนหน้านี้มี **.NET Framework, .NET Core, Xamarin** ⟶ แยกกัน
- Microsoft รวมทั้งหมดเป็น **.NET (Unified Platform)**

☐ **ASP.NET Core บน .NET 6 (LTS – 2021)**

- แนะนำ **Minimal APIs** ⟶ โค้ดน้อยลง
- ใช้ **Top-level statements** ใน Program.cs
- รองรับ **C# 10**

ตัวอย่าง Minimal API:

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

app.MapGet("/", () => "Hello ASP.NET Core 6!");

app.Run();
```

☐ **ASP.NET Core บน .NET 7 (STS – 2022)**

- ปรับปรุง **Minimal APIs** ให้รองรับ Filter, Binding ที่ซับซ้อน
- Performance ดีขึ้นมาก
- C# 11

☐ **ASP.NET Core บน .NET 8 (LTS – 2023)**

- รองรับ **Native AOT (Ahead-Of-Time compilation)** ⟶ Build เป็น Executable เล็กและเร็ว
- ปรับปรุง **Blazor United** ⟶ รวม Blazor Server + WebAssembly + SSR

- รองรับ **C# 12**

สรุป:

- ถ้าจะใช้ในโปรเจกต์จริง **.NET 6 หรือ .NET 8 (LTS)** เหมาะที่สุด
- .NET 7 เป็น **STS (Short-Term Support)** หมดซัพพอร์ตเร็วกว่าครับ

---

3. โครงสร้างโปรเจกต์ (Program.cs, Startup.cs, appsettings.json)

เมื่อสร้างโปรเจกต์ ASP.NET Core (dotnet new webapi) จะได้โครงสร้าง เช่น:

```
MyWebApp/
├── Controllers/
│   └── WeatherForecastController.cs
├── Properties/
│   └── launchSettings.json
├── appsettings.json
├── Program.cs
├── Startup.cs (เฉพาะ .NET 5 หรือต่ำกว่า / หรือถ้า Dev แยกเอง)
└── MyWebApp.csproj
```

Program.cs (ตั้งแต่ .NET 6 ⟶ ใช้ Minimal Hosting Model)

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

// Register Services (DI Container)
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Middleware pipeline
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

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

app.MapControllers(); // Map API Controllers

app.Run();
```

- builder.Services ⟶ สำหรับลงทะเบียน Service (DI, EF Core, Identity ฯลฯ)
- app.Use...() ⟶ Middleware Pipeline เช่น Authentication, Logging, Routing
- app.Map...() ⟶ Endpoint Mapping เช่น Controller, Razor Page, Blazor

Startup.cs (ใช้บ่อยใน .NET 3.1/5)

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseRouting();
        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
}
```

ใน .NET 6/7/8 โค้ด Startup.cs ถูกย่อเหลือใน Program.cs

☐ **appsettings.json**

ใช้เก็บ **Configuration** เช่น Connection String, Logging, Custom Settings

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "ConnectionStrings": {
    "DefaultConnection": "Server=.;Database=MyDb;Trusted_Connection=True;"
  },
  "AllowedHosts": "*"
}
```

---

4. การรันโปรเจกต์แรกด้วย dotnet run

☐ **ขั้นตอน**

1. ติดตั้ง **.NET SDK 6/7/8**
2. dotnet --version

   (จะแสดงเวอร์ชันเช่น 8.0.100)
3. สร้างโปรเจกต์ใหม่ (เช่น Web API)
4. dotnet new webapi -o MyWebApp
5. cd MyWebApp
6. รันโปรเจกต์
7. dotnet run

   Output:

   Building...

   Now listening on: https://localhost:7113

   Now listening on: http://localhost:5189

   Application started. Press Ctrl+C to shut down.
8. เปิด Browser ⟶ https://localhost:7113/swagger
   - จะเห็น Swagger UI ที่ ASP.NET Core Generate มาให้

☐ **สรุปบทที่ 1:**

- ASP.NET มีวิวัฒนาการจาก Web Forms ⟶ MVC ⟶ Core
- ASP.NET Core เป็น Cross-platform, Open-source, Modular และ Performance สูง
- .NET 6 และ .NET 8 เป็นเวอร์ชัน LTS ที่เหมาะสำหรับ Production
- โครงสร้างโปรเจกต์ใหม่ใช้ Program.cs แทน Startup.cs
- Config ต่าง ๆ อยู่ใน appsettings.json
- การรันโปรเจกต์แรกใช้ dotnet run และสามารถดูผลลัพธ์ได้ที่ Swagger UI

บทที่ 1 (Introduction to ASP.NET Core) แบบเชิงลึก

1) ประวัติและวิวัฒนาการ (ละเอียดเชิงเทคนิค)

Timeline แบบย่อ (เพื่อเข้าใจมุมมองเชิงสถาปัตยกรรม)

- **ASP.NET (2002)** บน .NET Framework — model แบบ server-side rendering, ผูกกับ IIS และ System.Web (ใหญ่ หนัก)
- **Web Forms** — event-driven, code-behind; เหมาะกับคนที่มาจาก WinForms แต่สร้าง HTML ที่ไม่ค่อยคุมได้
- **ASP.NET MVC (2009)** — นำแนวคิด MVC เข้ามา แยก concerns, ดีต่อ unit testing และควบคุม HTML ได้มากขึ้น แต่ยังผูกติด System.Web และ Windows/IIS
- **ASP.NET Core (2016 ⟶ ปัจจุบัน)** — เขียนใหม่เป็น cross-platform, modular, lightweight, ใช้ Kestrel เป็น web server, open-source, แยกตัวเป็น NuGet packages มากขึ้น ⟶ เหมาะกับ Cloud/Containers/High-performance

สาเหตุของการเปลี่ยนแปลงเชิงสถาปัตยกรรม

- ขนาดของ System.Web ใหญ่และ monolithic ⟶ เปลี่ยนเป็น modular middleware pipeline เพื่อโหลดเฉพาะสิ่งที่ต้องการ
- ต้องการ cross-platform ⟶ port ไป Linux/Containers ⟶ Kestrel ถูกออกแบบใหม่เป็น high-performance async server
- ต้องการการปรับแต่งง่าย (dependency injection, configuration providers, logging) ⟶ built-in DI + IConfiguration + Logging abstraction

สถาปัตยกรรมหลัก (ภาพรวม)

- **Hosting layer**: Kestrel (process) ⟵⟶ (optionally) Reverse Proxy (IIS/NGINX)
- **Middleware pipeline**: a chain of delegates (UseXxx) ที่สลับ request/response — ordering สำคัญ (Exception⟶Routing⟶Auth⟶Endpoints)
- **Dependency Injection**: built-in container (IServiceCollection/IServiceProvider) — service lifetimes (Transient/Scoped/Singleton)

- **Configuration**: Provider-based (appsettings.json, env vars, command-line, user secrets, Azure Key Vault ฯลฯ)
- **Logging**: abstraction (ILogger) รองรับ providers ต่าง ๆ (Console, EventLog, Seq, Serilog)

2) ASP.NET Core กับ .NET 6 / 7 / 8 — จุดเปลี่ยนและฟีเจอร์สำคัญ

สรุปเป็นประเด็นสำคัญที่มีผลต่อการพัฒนาและโครงสร้างโปรเจกต์

.NET 6 (LTS) — จุดเปลี่ยนสำคัญ

- **Minimal Hosting Model** (Top-level Program.cs) — ทำให้โค้ดสั้นลง: WebApplicationBuilder + WebApplication
- **Minimal APIs** — เขียน HTTP endpoints แบบ function-style แทน controller ได้ เหมาะกับ microservices / small APIs
- **Hot Reload** (Dev productivity)
- **C# 10 features** (global using, file-scoped namespace)

ตัวอย่าง minimal API (.NET 6+):

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

if (app.Environment.IsDevelopment()) app.UseSwagger().UseSwaggerUI();
app.MapGet("/hello", () => "Hello from minimal API!");

app.Run();
```

.NET 7 (STS)

- ปรับปรุง performance หลายจุด (JIT, GC, HTTP stack)
- Minimal APIs มีฟีเจอร์ binding/filters มากขึ้น
- Short-term support ⟶ เปลืองถ้าอยาก long-term stability

.NET 8 (LTS)

- **Native AOT** (Ahead-of-Time compilation) ⟶ สร้าง native executables ขนาดเล็ก เริ่มใช้ใน scenarios ที่ต้องการ cold-start ต่ำ (serverless, tiny service)
- ปรับปรุง Blazor (แนวทาง Blazor United / integration)
- เพิ่มความสามารถ runtime/SDK หลายด้าน และ C# 12 features

- ยังใช้ minimal hosting model เหมือนเดิม แต่มี performance / publishing options เพิ่มขึ้น (single-file, native AOT)

ข้อแนะนำ: สำหรับโปรเจกต์ production ขนาดใหญ่ ให้ใช้ LTS (เช่น .NET 6 หรือ .NET 8) เพื่อ support ระยะยาว

3) โครงสร้างโปรเจกต์ — เจาะลึกไฟล์สำคัญและความหมายเชิงลึก

ตัวอย่างโครงสร้าง (webapi template, .NET 7/8):

```
MyWebApp/
├── Controllers/
│   └── WeatherForecastController.cs
├── Properties/
│   └── launchSettings.json
├── appsettings.json
├── appsettings.Development.json
├── Program.cs
├── MyWebApp.csproj
└── Dockerfile (ถ้ามี)
```

Program.cs (Minimal Hosting — อธิบายบรรทัดต่อบรรทัด)

```
var builder = WebApplication.CreateBuilder(args);
// builder.Configuration : IConfiguration (รวม provider ต่าง ๆ)
// builder.Services : IServiceCollection (DI registration)
// builder.Environment : IWebHostEnvironment

builder.Services.AddControllers();
builder.Services.AddDbContext<MyDbContext>(opts => ...);
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Middleware pipeline — Execution order matters!
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseSwagger();
```

```
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();        // Redirect HTTP -> HTTPS
app.UseRouting();                 // Adds route matching
app.UseAuthentication();          // Authentication must appear before Authorization
app.UseAuthorization();
app.MapControllers();             // Map controller endpoints
app.Run();
```

ประเด็นสำคัญ

- builder.Configuration ถูกสร้างขึ้นก่อน Build() และรวม providers ตามลำดับ (see next section)
- app.UseXxx() เป็นการเรียงลำดับ middleware — ถ้าสลับตำแหน่ง จะเปลี่ยน behavior (เช่น Authorization อยู่ก่อน Routing ⟶ จะไม่ทำงาน)
- app.Map...() คือการเพิ่ม endpoints ที่จะถูกเรียกเมื่อ routing ตรงกัน

Startup.cs (Legacy style)

- ใน .NET 5 หรือต้องการแยกความรับผิดชอบ คุณอาจยังคงใช้ Startup:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services) { ... }
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env) { ... }
}
```

- ใน .NET 6+ คุณสามารถกลับไปใช้ Startup pattern ได้ (for clarity), แต่ default แนะนำใช้ Program.cs เดียว

appsettings.json / Configuration Providers (ลำดับและ precedence)

- **Providers ที่มีโดย default (เรียงลำดับถูกโหลด):**
    1. appsettings.json
    2. appsettings.{Environment}.json (e.g., appsettings.Development.json)
    3. User secrets (เฉพาะสภาพแวดล้อม dev)
    4. Environment variables
    5. Command-line arguments
- **Precedence:** ไฟล์ที่ถูกโหลดหลังจะ override ค่าเดิม — ค่า CLI > Env Vars > appsettings.{ENV} > appsettings.json

- **IConfiguration** รองรับการเข้าถึงแบบ key-path ("Logging:LogLevel:Default") และ binding ไปยัง POCO

ตัวอย่าง appsettings.json:

```
{
  "Logging": {
    "LogLevel": { "Default": "Information", "Microsoft": "Warning" }
  },
  "ConnectionStrings": { "Default": "Server=.;Database=MyDb;Trusted_Connection=True;" },
  "MyOptions": { "FeatureEnabled": true, "MaxItems": 100 }
}
```

การ bind:

```
builder.Services.Configure<MyOptions>(builder.Configuration.GetSection("MyOptions"));
// หรือ
var options = builder.Configuration.GetSection("MyOptions").Get<MyOptions>();
```

Secrets & Secure config

- **User Secrets** (dotnet user-secrets) — สำหรับ dev machine (ไม่ commit to git)
    - dotnet user-secrets init แล้ว dotnet user-secrets set "ConnectionStrings:Default" "..."
- **Production**: ใช้ environment variables / Azure Key Vault / HashiCorp Vault / secrets manager ของ cloud
- อย่าเก็บ secret ใน appsettings.json ที่จะ commit ขึ้น repo

launchSettings.json (dev-only)

- กำหนด profiles (IIS Express, Project) และ environment variables ระหว่าง debugging (เช่น ASPNETCORE_ENVIRONMENT)

  ตัวอย่าง:

```
{
  "profiles": {
    "MyWebApp": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7113;http://localhost:5189",
      "environmentVariables": { "ASPNETCORE_ENVIRONMENT": "Development" }
    }
```

```
  }
}
```

.csproj (SDK-style) — สิ่งควรรู้

- ตัวอย่าง:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>
</Project>
```

- Microsoft.NET.Sdk.Web จะเพิ่ม defaults สำหรับ web (Kestrel, web hosting)
- global.json ใช้ล็อค SDK เวอร์ชันถ้าต้องการ

DI (IServiceCollection) — lifetimes และ pitfalls

- **Transient**: สร้าง instance ใหม่ทุกครั้งที่ขอ (use for lightweight stateless services)
- **Scoped**: หนึ่ง instance ต่อ HTTP request — ใช้สำหรับ DbContext, unit-of-work
- **Singleton**: หนึ่ง instance ตลอด lifetime ของ app — **ระวัง** ห้าม inject Scoped service เข้าใน Singleton (จะเกิด captive dependency)
- **Common pitfalls**:
  - ไม่ควร new DbContext เอง — ให้ DI จัดการ
  - ระวัง stateful singleton (จะทำให้ race conditions)

4) การรันโปรเจกต์แรกด้วย dotnet run — ลึกจนเข้าใจทุกขั้นตอน

คำสั่งพื้นฐาน

```
dotnet new webapi -n MyWebApp
cd MyWebApp
dotnet run
```

กระบวนการภายในของ dotnet run

1. **dotnet run** จะทำ dotnet build (implicit) ⟶ restore (ถ้ายังไม่ได้) ⟶ สร้าง assembly (DLL/EXE)
2. แล้วจะเรียก dotnet <assembly>.dll (หรือรัน native exe ถ้า publish แบบ self-contained/native AOT)
3. โปรเซสจะเริ่ม host (Kestrel) และ binding port ตาม ASPNETCORE_URLS / --urls / launchSettings.json / default (localhost:xxxxx)
4. คุณจะเห็น log เช่น:

Building...

info: Microsoft.Hosting.Lifetime[14]

Now listening on: https://localhost:7113

info: Microsoft.Hosting.Lifetime[14]

Now listening on: http://localhost:5189

info: Microsoft.Hosting.Lifetime[0]

Application started. Press Ctrl+C to shut down.

การกำหนดพอร์ต / URL / HTTPS

- ผ่าน environment variable:
  - ASPNETCORE_URLS="http://0.0.0.0:5000;https://0.0.0.0:5001" dotnet run
- หรือ argument:
  - dotnet run --urls "http://0.0.0.0:5000"
- **HTTPS Dev Certificate**: ติดตั้ง/เชื่อถือในเครื่อง dev:
  - dotnet dev-certs https --trust (จะสร้างและ trust certificate บนเครื่อง dev)
  - ถ้าไม่มี cert, Kestrel อาจ fallback ไป HTTP หรือ fail ขึ้นกับ config

Environment & ASPNETCORE_ENVIRONMENT

- ค่าที่ common: Development, Staging, Production
- พฤติกรรมที่ต่างกัน:
  - Development ⟶ DeveloperExceptionPage, Hot reload enabled, logs verbose
  - Production ⟶ default error handler, stricter CORS, stricter cookie settings
- ตั้งค่า:
  - ASPNETCORE_ENVIRONMENT=Development dotnet run
  - หรือใน launchSettings.json สำหรับ debugging

Hot Reload / dotnet watch

- Developer productivity:
  - dotnet watch run ⟶ จะคอยดูการเปลี่ยนแปลงไฟล์และ reload (Hot Reload) โดยไม่ต้อง rebuild full cycle
- ข้อควรระวัง: hot reload ดีสำหรับ code changes แต่ schema changes หรือ DI changes บางครั้งต้อง restart full

Publishing & Run in Production

- Build for production:
  - dotnet publish -c Release -o ./publish
  - ผลลัพธ์: publish folder ที่เอาไปรันด้วย dotnet MyWebApp.dll หรือ build native AOT (--self-contained / -p:PublishAot=true)

- Containerization:
    - ใช้ multi-stage Dockerfile (build -> publish -> runtime image เช่น mcr.microsoft.com/dotnet/aspnet:8.0)

Graceful shutdown

- Host listens for SIGINT/SIGTERM ⟶ ส่ง cancellation token ให้ hosted services ⟶ ใช้ IHostApplicationLifetime หรือ CancellationToken ใน background tasks เพื่อปิดตัวอย่างเป็นระเบียบ

5) ตัวอย่างการใช้งานจริง (snippet รวมแนวปฏิบัติ)

Program.cs ตัวอย่าง (ครบเครื่อง)

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

// Configuration & Options
builder.Services.Configure<MyOptions>(builder.Configuration.GetSection("MyOptions"));

// Add DbContext (scoped)
builder.Services.AddDbContext<MyDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("Default")));

// Add Authentication & Authorization
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
       .AddJwtBearer(...);
builder.Services.AddAuthorization();

// Add HttpClient factory (best practice)
builder.Services.AddHttpClient("github", c => { c.BaseAddress = new
Uri("https://api.github.com/"); });

// Add controllers & swagger
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

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

```
// Middleware ordering - critical
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.UseRouting();
app.UseAuthentication(); // before Authorization
app.UseAuthorization();

app.MapControllers();

app.Run();
```

6) Best practices & gotchas (สรุปเชิงปฏิบัติ)

- **Config & Secrets**: user-secrets / env vars / KeyVault — อย่า commit secrets
- **DI**: ให้ใช้ scoped สำหรับ DbContext; ห้าม inject scoped into singleton
- **HttpClient**: อย่า new HttpClient() เอง — ใช้ IHttpClientFactory
- **Middleware ordering**: Authentication ⟶ Authorization ต้องวางถูกตำแหน่งก่อน MapControllers()
- **Health & Observability**: ใช้ Health Checks, Metrics (Prometheus), Distributed Tracing (OpenTelemetry) และ structured logging (Serilog)
- **Minimal API vs Controllers**: ใช้ Minimal API สำหรับ services เล็ก ๆ หรือ microservices; ใช้ Controllers + Filters สำหรับแอปขนาดใหญ่ที่ต้องการ structure และ testability
- **Production hosting**: ใน container ให้ expose Kestrel ตรง ๆ และใช้ reverse proxy (optional) ถ้าจำเป็น (NGINX/ALB) — avoid in-process IIS unless you need Windows-only features

---

ประวัติและวิวัฒนาการของ ASP.NET

ประวัติและวิวัฒนาการของ ASP.NET แบบเจาะลึกเชิงเทคนิค ตั้งแต่ **Web Forms ⟶ MVC ⟶ Core** ให้ครบทุกมุมมอง

1 ASP.NET (Web Forms) — จุดเริ่มต้น (2002)

ASP.NET Web Forms ถูกเปิดตัวพร้อม **.NET Framework 1.0**

- **Model**: Event-driven, Code-behind
- **ลักษณะเด่น**:
  - คล้าย Windows Forms ⟶ Developers ที่มาจาก desktop development จะคุ้นเคย
  - มี **Server Controls** เช่น <asp:Button> <asp:GridView> ⟶ Generate HTML อัตโนมัติ
  - ViewState เก็บสถานะของ UI controls ระหว่าง postback
- **ข้อดี**:
  - เขียนเว็บเร็วโดยไม่ต้องยุ่งกับ HTML/JS มาก
  - Event model ทำให้ logic คล้าย desktop app
- **ข้อจำกัด**:
  - HTML output มัก **ไม่สะอาด** / ไม่เหมาะกับ Frontend modern
  - Code-behind tightly coupled ⟶ ยากต่อ testing และ maintenance
  - ผูกติดกับ **IIS/Windows**

สรุป: เหมาะสำหรับ intranet หรือแอปที่เน้น rapid development แต่ไม่ยืดหยุ่นสำหรับ web standards หรือ cross-platform

---

2 ASP.NET MVC — Separation of Concerns (2009)

ASP.NET MVC เปิดตัวปี 2009

- นำแนวคิด **Model-View-Controller (MVC)** มาใช้ ⟶ แยก:
  - **Model**: Data & Business Logic
  - **View**: UI (Razor / HTML)
  - **Controller**: รับ request, เลือก view, ส่ง data ⟶ response

ลักษณะเด่น:

- **Unit Testing Friendly**: Controller แยกจาก UI ⟶ เขียน test ได้ง่าย
- **Full control over HTML**: ใช้ Razor syntax (@Model.Name)
- **Routing**: สามารถกำหนด URL patterns ได้อิสระ (Attribute Routing / Convention-based Routing)

ข้อดี:

- เหมาะกับ web apps ที่ต้องการ maintainability และ flexibility
- สนับสนุน TDD/BDD ได้ดี

ข้อจำกัด:

- ยังผูกติดกับ **System.Web.dll** ⟶ ขนาดใหญ่, performance ต่ำกว่า Core
- ไม่ cross-platform ⟶ ทำงานเฉพาะ Windows/IIS

สรุป: เหมาะสำหรับแอปขนาดกลางถึงใหญ่ ต้องการ separation of concerns และ testability

---

3 ASP.NET Core — Modern, Cross-platform (2016–ปัจจุบัน)

ASP.NET Core เป็นการเขียนใหม่ของ ASP.NET

- **Cross-platform**: Windows, Linux, macOS
- **Open-source** (hosted on GitHub)
- ใช้ **Kestrel Web Server** ⟶ Lightweight, High Performance, Async-first
- **Modular**: NuGet packages ⟶ โหลดเฉพาะสิ่งที่ต้องการ (ลดขนาด / improve performance)
- **Unified framework**: รองรับ Web API, MVC, Razor Pages, Blazor, Minimal APIs

ลักษณะเด่น:

- **Middleware Pipeline** ⟶ app.UseXxx() เพื่อจัดการ request/response
- **Dependency Injection** ในตัว ⟶ register services และ inject ใน controller / middleware
- **Configuration** ผ่าน IConfiguration + appsettings.json / environment variables / secrets
- **Performance**: ดีขึ้นกว่า MVC / Web Forms อย่างชัดเจน
- **Cross-platform hosting**: ทำงานบน containers, cloud, serverless

ตัวอย่างการเรียกใช้งาน Minimal API (.NET 6+):

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

app.MapGet("/", () => "Hello ASP.NET Core!");
app.Run();
```

ข้อดี:

- เหมาะกับ microservices, cloud apps, high-performance web apps
- Flexible, modern development, รองรับ DevOps / CI/CD

สรุป:

- จาก tightly-coupled ⟶ modular ⟶ cross-platform
- Web Forms ⟶ MVC ⟶ Core แสดงการพัฒนา **จาก rapid dev / Windows-only ⟶ testable / flexible ⟶ high-performance / cloud-ready**

เจาะลึก ASP.NET Core กับ .NET 6/7/8

เจาะลึก **ASP.NET Core กับ .NET 6/7/8** แบบเชิงเทคนิค ครอบคลุมความแตกต่าง, ฟีเจอร์สำคัญ, การเปลี่ยนแปลงสถาปัตยกรรม และสิ่งที่นักพัฒนาควรรู้

---

1 Overview: ASP.NET Core + .NET 6/7/8

- **ASP.NET Core** = Framework สำหรับสร้าง **Web, APIs, Blazor, Minimal APIs** บน **Cross-platform**
- **.NET 6/7/8** = Unified Platform ของ .NET (รวม .NET Core, .NET Framework, Xamarin)
- **Key points**:
  - Minimal Hosting Model (Program.cs เดียว)
  - Performance สูง (Kestrel, async-first, pipeline optimized)
  - Cross-platform / Container-friendly / Cloud-ready

---

2 .NET 6 (LTS – Long-Term Support)

จุดเด่น:

- **Minimal APIs** ⟶ เขียน HTTP endpoints แบบ function-style
- **Top-level statements** ⟶ โค้ดใน Program.cs สั้นลง, no need Startup.cs
- **C# 10** features ⟶ global using, file-scoped namespace, record structs
- **Hot Reload** ⟶ Dev productivity: เปลี่ยนโค้ดแล้ว reload อัตโนมัติ

ตัวอย่าง Minimal API

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

if (app.Environment.IsDevelopment())
    app.UseSwagger().UseSwaggerUI();

app.MapGet("/", () => "Hello .NET 6 ASP.NET Core!");
app.Run();
```

Notes:

- LTS ⟶ เหมาะสำหรับ Production / Enterprise

- Middleware pipeline ใช้ app.Use... ⟶ app.Map...

---

3 .NET 7 (STS – Short-Term Support)

จุดเด่น:

- ปรับปรุง **Minimal APIs** ⟶ advanced routing, filters, model binding
- Performance boost: JIT, GC, HTTP stack
- ใช้ **C# 11** features เช่น raw string literals, generic attributes
- STS ⟶ รองรับระยะสั้น (~18 เดือน) ⟶ ไม่เหมาะสำหรับ long-term production

Notes:

- เหมาะกับ experimentation / fast-moving projects
- ใช้ features ใหม่ได้เร็ว แต่ต้อง upgrade เป็น LTS ต่อไป

---

4 .NET 8 (LTS – 2023)

จุดเด่น:

- **Native AOT (Ahead-of-Time compilation)** ⟶ สร้าง native executable ขนาดเล็ก, cold-start ต่ำ
- **Blazor United** ⟶ รวม Blazor Server + WASM + SSR ในโปรเจกต์เดียว
- **Performance**: further improved Kestrel, GC, Hot reload, minimal APIs
- **C# 12** support ⟶ improved lambda, pattern matching, collection literals

ตัวอย่าง Native AOT

```
dotnet publish -c Release -r win-x64 -p:PublishAot=true
```

- Output = standalone executable ไม่ต้องมี .NET runtime ติดตั้งในเครื่อง

---

5 เปรียบเทียบ .NET 6 / 7 / 8 (เชิง Developer)

Feature / Aspect	.NET 6 (LTS)	.NET 7 (STS)	.NET 8 (LTS)
Support	Long-term	Short-term	Long-term
Minimal APIs	Basic	Advanced filters & binding	Enhanced + Native AOT ready
Hot Reload			Improved
Performance	High	Higher	Highest
Language Version	C# 10	C# 11	C# 12

Feature / Aspect	.NET 6 (LTS)	.NET 7 (STS)	.NET 8 (LTS)
Blazor	Server / WASM	Server / WASM	Blazor United
Native AOT	□	□	□ (production ready)
Recommended for Production	□	⚠□ (short-term)	□

6□ Key takeaways สำหรับ ASP.NET Core Developers

1. เลือก **LTS สำหรับ Production** ⟶ .NET 6 หรือ .NET 8
2. **Minimal Hosting Model** ⟶ program.cs เดียว, DI, Middleware pipeline
3. **Native AOT (.NET 8)** ⟶ ดีสำหรับ microservices / serverless / container startup
4. **Hot Reload + Dev productivity** ⟶ ทุกเวอร์ชันใหม่ ทำให้ iteration เร็วขึ้น
5. **Cross-platform & Containers** ⟶ Core + .NET 6/8 สามารถ deploy บน Linux/Windows/macOS และ Docker ได้

เครื่องมือและ IDE/Editor ที่นิยมใช้พัฒนา ASP.NET / ASP.NET Core

---

1□ Integrated Development Environment (IDE) หลัก

1. Visual Studio (Windows / Mac)

- **เวอร์ชันที่นิยมใช้**
  - **Visual Studio 2022 (Windows)** ⟶ Full-featured, รองรับ .NET 6/7/8, Debugging, Profiling, Designer
  - **Visual Studio for Mac** ⟶ ASP.NET Core, Blazor, Mobile development (Xamarin/MAUI)
- **จุดเด่น**
  - GUI Designer สำหรับ Windows Forms, WPF
  - Advanced Debugger, IntelliSense, Code Refactoring
  - Integrated Package Manager (NuGet)
  - Support Live Unit Testing, Hot Reload
  - Built-in Git support
- **เหมาะกับ**: Enterprise apps, Full-stack dev, มีทีมใหญ่

---

2. Visual Studio Code (Cross-platform)

- **จุดเด่น**
  - Lightweight, Cross-platform (Windows/Linux/macOS)
  - Extensions: C# (OmniSharp), NuGet Package Manager, Debugger for .NET Core, REST Client, GitLens
  - Hot Reload + dotnet CLI integration
  - ใช้ได้ทั้ง Minimal APIs, Blazor, MVC, Web API
- **เหมาะกับ**: Dev ที่ชอบ lightweight IDE, Linux/Mac, containerized workflow

---

3. JetBrains Rider

- Cross-platform .NET IDE (Windows/Linux/macOS)
- **จุดเด่น**
  - Fast code analysis, refactoring, debugging
  - Support .NET Core, ASP.NET Core, Blazor
  - Integrated Unit Testing (xUnit, NUnit, MSTest)
  - Git integration, Database tools
- **ข้อจำกัด**: ต้องซื้อ License

---

2 Command-line Tools (CLI)

dotnet CLI

- **คำสั่งสำคัญ**
  - dotnet new webapi ⟶ สร้างโปรเจกต์ Web API
  - dotnet run ⟶ รันโปรเจกต์
  - dotnet build ⟶ Build project
  - dotnet publish ⟶ เตรียม deploy
  - dotnet add package <package> ⟶ Add NuGet package
  - dotnet restore ⟶ Restore dependencies
  - dotnet watch run ⟶ Auto reload / Hot reload
- **ข้อดี**
  - Cross-platform, เหมาะกับ DevOps, CI/CD, Docker
  - Minimal API / Headless development ได้ดี

NuGet Package Manager

- ใช้งานผ่าน
  - Visual Studio GUI
  - CLI: dotnet add package <PackageName>
- **Purpose**: Download / update libraries, middleware, EF Core, Logging, Authentication

---

3 Database / ORM Tools

1. Entity Framework Core Tools

- **CLI commands**
  - dotnet ef migrations add <MigrationName> ⟶ สร้าง migration
  - dotnet ef database update ⟶ Apply migration
  - dotnet ef dbcontext scaffold ⟶ Scaffold DB
- Integrated กับ Visual Studio / VS Code

2. SQL Server Management Studio (SSMS)

- GUI สำหรับ SQL Server
- ใช้ร่วมกับ EF Core หรือ ADO.NET

3. Azure Data Studio

- Cross-platform, modern GUI
- รองรับ SQL Server, PostgreSQL, MySQL

---

4 Debugging / Profiling / Monitoring

1. Visual Studio Debugger

- Breakpoints, Conditional Breakpoints, Watch Window, Call Stack, Immediate Window

2. dotnet-trace / dotnet-dump

- CLI tools สำหรับ performance tracing / memory dump analysis

3. Application Insights (Azure)

- Logging, telemetry, distributed tracing, metrics

4. Serilog + Seq / ELK Stack

- Structured logging
- Real-time monitoring

---

5 Frontend / UI Tools สำหรับ ASP.NET Core

1. Razor Pages / Blazor

- Razor syntax editor ⟶ Visual Studio / VS Code

- Hot Reload support

2. JavaScript / CSS Frameworks

- Node.js + npm / yarn / pnpm
- SPA frameworks: React, Angular, Vue ⟶ integrate ผ่าน npm run build / wwwroot

---

6 Containerization / DevOps

1. Docker

- Multi-stage builds for ASP.NET Core apps
- Run cross-platform containers ⟶ Linux / Windows

2. Kubernetes

- Orchestrate multiple ASP.NET Core microservices

3. CI/CD Tools

- GitHub Actions, Azure DevOps, GitLab CI
- Build ⟶ Test ⟶ Publish ⟶ Deploy

---

7 Testing Tools

Unit / Integration Testing

- xUnit, NUnit, MSTest ⟶ Visual Studio / Rider / CLI
- TestServer for ASP.NET Core ⟶ run integration tests without real HTTP server

API Testing

- Postman / Insomnia
- Swagger UI ⟶ auto-generated endpoints testing

---

8 Summary / Recommendation

Tool Category	Recommended Tool(s)	Notes
IDE	Visual Studio 2022 (Windows), VS Code (cross-platform), Rider	Depends on team size & platform
CLI	dotnet CLI, NuGet	Essential for DevOps / minimal APIs
Database / ORM	EF Core CLI, SSMS, Azure Data Studio	EF Core + CLI = migrate DB easily
Debug & Profiling	VS Debugger, dotnet-trace/dump, App	Production monitoring /

Tool Category	Recommended Tool(s)	Notes
	Insights	memory analysis
Frontend / UI	Razor / Blazor, npm frameworks	SPA integration via wwwroot
Containerization & CI/CD	Docker, Kubernetes, Azure DevOps, GitHub Actions	For cloud-native / microservices
Testing	xUnit / NUnit / MSTest, Postman, Swagger UI	Unit + Integration + API testing

---

การติดตั้งเครื่องมือพัฒนา ASP.NET Core

---

1 ติดตั้ง .NET SDK

Windows / macOS / Linux

1. เข้าเว็บทางการ: https://dotnet.microsoft.com/en-us/download
2. เลือก **.NET 8 SDK (LTS)** หรือเวอร์ชันที่ต้องการ (.NET 6 LTS, .NET 7 STS)
3. ดาวน์โหลดและติดตั้งตาม OS

ตรวจสอบการติดตั้ง

```
dotnet --version
dotnet --list-sdks
dotnet --list-runtimes
```

- ควรเห็นเวอร์ชันที่ติดตั้ง เช่น 8.0.xxx

---

2 ติดตั้ง IDE / Editor

2.1 Visual Studio 2022 (Windows)

1. ดาวน์โหลดจาก https://visualstudio.microsoft.com/downloads/
2. เลือก **Community / Professional / Enterprise** ตาม license
3. ในตัว installer:
    - เลือก **ASP.NET and web development workload**
    - (Optional) เลือก **Azure development**, **Blazor WebAssembly** หรือ **Desktop development** ตามต้องการ
4. ติดตั้งเสร็จ เปิด VS ⟶ สร้างโปรเจกต์ ASP.NET Core Web API

2.2 Visual Studio Code (Cross-platform)

1. ดาวน์โหลดจาก https://code.visualstudio.com/

2. ติดตั้ง Extension:
    - **C# (OmniSharp)**
    - **NuGet Package Manager**
    - **Debugger for .NET Core**
    - (Optional) **REST Client**, **GitLens**
3. เปิด VS Code ⟶ Terminal ⟶ สร้างโปรเจกต์

dotnet new webapi -n MyWebApp

cd MyWebApp

dotnet run

2.3 JetBrains Rider (Cross-platform)

1. ดาวน์โหลดจาก https://www.jetbrains.com/rider/
2. ติดตั้งตาม OS
3. เปิดโปรเจกต์ ASP.NET Core ผ่าน **New Project ⟶ .NET 8 / 6 ⟶ ASP.NET Core Web API**

---

3 ติดตั้ง Database Tools

SQL Server Management Studio (Windows)

1. ดาวน์โหลด: https://aka.ms/ssmsfullsetup
2. ติดตั้ง และ connect กับ SQL Server instance
3. ใช้ EF Core CLI หรือ query database

Azure Data Studio (Cross-platform)

1. ดาวน์โหลด: https://learn.microsoft.com/en-us/sql/azure-data-studio/download-azure-data-studio
2. ติดตั้งแล้ว connect กับ SQL Server / PostgreSQL

EF Core CLI

dotnet tool install --global dotnet-ef

dotnet ef --version

---

4 ติดตั้ง Docker (Optional, สำหรับ Containerized Dev)

Windows

https://www.docker.com/products/docker-desktop/

- เปิด WSL 2 (ถ้าใช้ Windows) ⟶ เลือกใช้ Linux containers

macOS / Linux

- ดาวน์โหลด Docker Desktop / Docker Engine ตาม OS

ตรวจสอบ

docker --version

docker run hello-world

---

5 ติดตั้ง Git (Version Control)

git --version

- Windows: https://git-scm.com/download/win
- macOS: brew install git
- Linux: sudo apt install git หรือ sudo yum install git

---

6 ตรวจสอบว่าพร้อมใช้งาน

```
dotnet --version          # ตรวจสอบ .NET SDK
code --version            # ตรวจสอบ VS Code
git --version             # ตรวจสอบ Git
docker --version          # ตรวจสอบ Docker (ถ้าใช้)
dotnet ef --version       # ตรวจสอบ EF Core CLI
```

Expected Output ตัวอย่าง

8.0.100

1.82.0

2.41.0

24.0.0

8.0.0

หลังติดตั้งเครื่องมือครบแล้ว คุณสามารถ:

1. สร้างโปรเจกต์ ASP.NET Core

dotnet new webapi -n MyWebApp

cd MyWebApp

dotnet run

2. เปิดใน Visual Studio / VS Code
3. Connect DB, ใช้ Docker, หรือ Debug ได้ทันที

Step-by-step ตัวอย่างติดตั้ง ASP.NET Core Development

การติดตั้งเครื่องมือพัฒนา ASP.NET Core แบบละเอียดทีละขั้นตอน (Windows เป็นตัวอย่างหลัก) พร้อมคำสั่งเช็คและคำแนะนำเชิงปฏิบัติ จะครอบคลุม IDE, .NET SDK, Database, Docker และ Git

---

Step 1: ติดตั้ง .NET SDK

1. เข้าเว็บทางการ .NET download
2. เลือก **.NET 8 SDK (LTS)** หรือ .NET 6 LTS ตามความต้องการ
3. ดาวน์โหลด **Installer สำหรับ Windows**
4. ดับเบิลคลิกไฟล์ .exe ⟶ เลือก **Next** ⟶ **Install**
5. รอจนติดตั้งเสร็จ ⟶ กด **Finish**

ตรวจสอบการติดตั้ง

dotnet --version

dotnet --list-sdks

dotnet --list-runtimes

- Output ตัวอย่าง:

8.0.100

8.0.0 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]

- แปลว่า SDK พร้อมใช้งาน

Step 2: ติดตั้ง IDE / Editor

2.1 Visual Studio 2022 (Windows)

1. ดาวน์โหลดจาก Visual Studio Downloads
2. เลือก **Community / Professional / Enterprise**
3. รัน installer ⟶ เลือก **ASP.NET and web development** workload
4. (Optional) เลือก Azure development / Desktop development
5. กด **Install** ⟶ รอจนเสร็จ
6. เปิด Visual Studio ⟶ สร้างโปรเจกต์ใหม่ **ASP.NET Core Web API**

2.2 Visual Studio Code (Cross-platform)

1. ดาวน์โหลดจาก VS Code
2. ติดตั้ง extension:
    - **C# (OmniSharp)** ⟶ Debug + IntelliSense
    - **NuGet Package Manager** ⟶ Manage packages
    - **REST Client** (optional) ⟶ Test API endpoints

3. เปิด Terminal ใน VS Code:

dotnet new webapi -n MyWebApp

cd MyWebApp

dotnet run

- เปิด browser ⟶ https://localhost:7113/swagger/index.html (สำหรับ Web API template)

2.3 JetBrains Rider (Cross-platform)

1. ดาวน์โหลด JetBrains Rider
2. ติดตั้งตาม OS
3. New Project ⟶ เลือก **ASP.NET Core Web API (.NET 6/8)**
4. Build ⟶ Run ⟶ Debug ได้ทันที

Step 3: ติดตั้ง Database Tools

3.1 SQL Server Management Studio (SSMS)

1. ดาวน์โหลดจาก https://aka.ms/ssmsfullsetup
2. ติดตั้งตามขั้นตอน ⟶ Finish
3. เปิด ⟶ Connect กับ SQL Server instance
4. สร้าง database และ test query

3.2 Azure Data Studio (Cross-platform)

1. ดาวน์โหลดจาก Azure Data Studio
2. ติดตั้ง ⟶ เปิด ⟶ Connect กับ SQL Server / PostgreSQL

3.3 EF Core CLI

dotnet tool install --global dotnet-ef

dotnet ef --version

- ใช้สำหรับ Migrations, Scaffold DB, Update database

Step 4: ติดตั้ง Docker (Optional)

1. ดาวน์โหลด Docker Desktop: https://www.docker.com/products/docker-desktop/
2. ติดตั้ง ⟶ Restart Windows (ถ้าจำเป็น)
3. เปิด Docker Desktop ⟶ ตรวจสอบว่า **Running**
4. ตรวจสอบ CLI:

docker --version

docker run hello-world

- Output: แสดงข้อความ confirm Docker พร้อมใช้งาน

Step 5: ติดตั้ง Git (Version Control)

- Windows: https://git-scm.com/download/win ⟶ ติดตั้ง
- macOS: brew install git
- Linux: sudo apt install git หรือ sudo yum install git

ตรวจสอบ:

git --version

- Output เช่น 2.41.0

Step 6: ตรวจสอบว่าเครื่องมือพร้อมใช้งาน

```
dotnet --version       # .NET SDK
code --version         # VS Code
git --version          # Git
docker --version       # Docker (optional)
dotnet ef --version    # EF Core CLI
```

Expected Output:

```
8.0.100
1.82.0
2.41.0
24.0.0
8.0.0
```

Step 7: สร้างโปรเจกต์ตัวอย่าง ASP.NET Core

1. เปิด Terminal / VS Code / VS
2. สร้างโปรเจกต์ Web API:

```
dotnet new webapi -n MyWebApp
cd MyWebApp
dotnet run
```

3. เปิด Browser ⟶ https://localhost:7113/swagger/index.html
4. ตรวจสอบ API endpoints พร้อมใช้งาน

☐ **สรุป Workflow หลังติดตั้งเสร็จ**

1. .NET SDK + CLI ⟶ Compile / Run / Publish

2. IDE (VS / VS Code / Rider) ⟶ Code + Debug + Refactor
3. Database Tools ⟶ Manage DB / Test query
4. Docker ⟶ Containerize / DevOps workflow
5. Git ⟶ Version control + CI/CD

โครงสร้างโปรเจกต์ ASP.NET Core

โครงสร้างโปรเจกต์ **ASP.NET Core** แบบละเอียดเจาะลึก โดยเฉพาะ **Program.cs, Startup.cs, appsettings.json** พร้อมตัวอย่างและสรุปบทบาทของแต่ละไฟล์

---

1 Overview ของโครงสร้างโปรเจกต์ ASP.NET Core

เมื่อสร้างโปรเจกต์ **ASP.NET Core Web API** (dotnet 6/7/8) จะได้ไฟล์และโฟลเดอร์หลักดังนี้:

```
MyWebApp/
├── Program.cs
├── Startup.cs          (optional ใน .NET 6+)
├── appsettings.json
├── appsettings.Development.json
├── Controllers/
│   └── WeatherForecastController.cs
├── Properties/
│   └── launchSettings.json
├── wwwroot/            (static files)
├── MyWebApp.csproj
└── ... (อื่นๆ เช่น Models, Services)
```

---

2 Program.cs

บทบาทหลัก

- Entry point ของโปรเจกต์
- กำหนด **Host**, **Middleware pipeline**, และ **Service registrations**
- ใน .NET 6/7/8 ⟶ ใช้ **Minimal Hosting Model** ⟶ สามารถเขียนโค้ดสั้น ไม่ต้องมี Startup.cs

ตัวอย่าง Program.cs (Minimal API style)

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