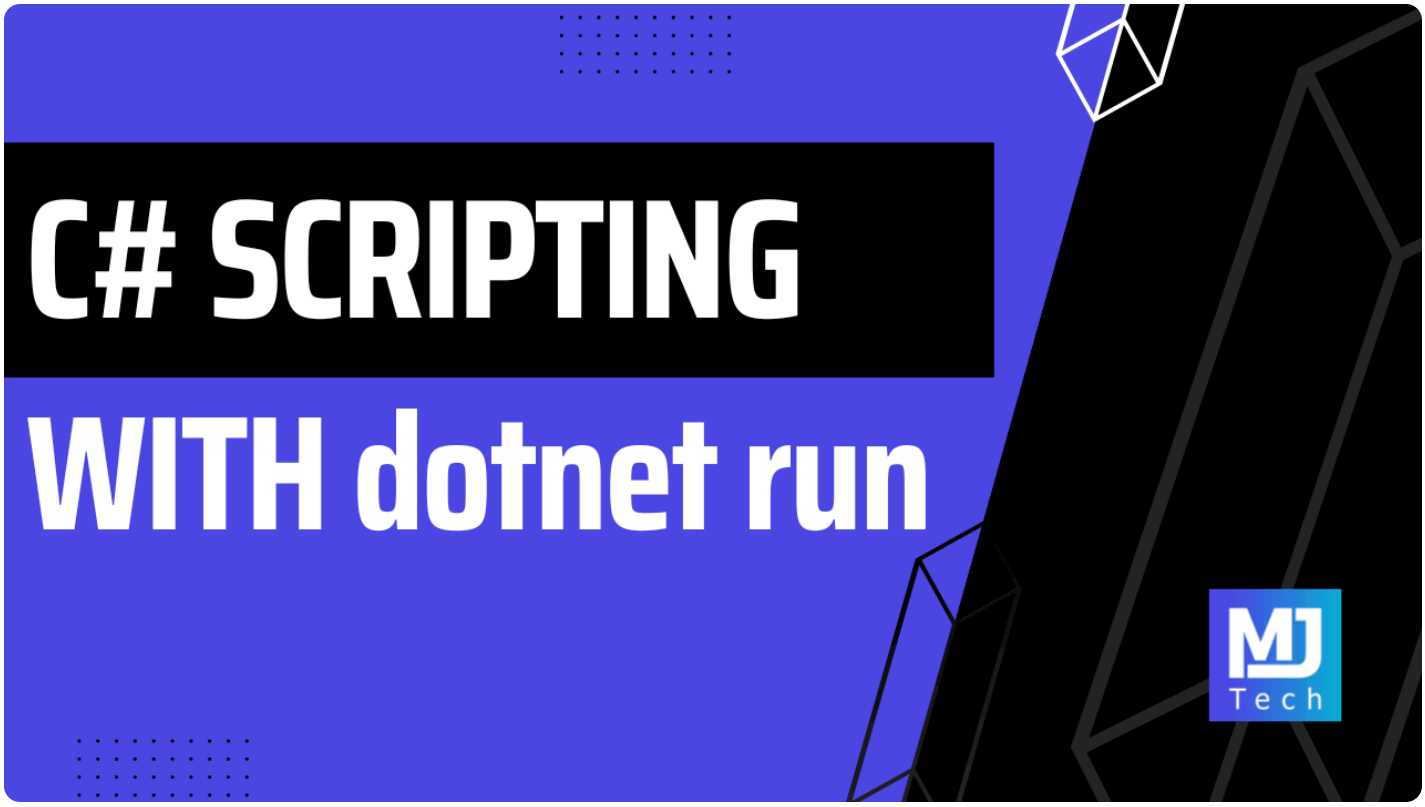

Run C# Scripts With dotnet run app.cs (No Project Files Needed)

4 min read · JUNE 14, 2025

Nick Chapsas' Dometrain is celebrating 2 years of teaching .NET developers, and they are offering their **"From Zero to Hero: REST APIs in .NET"** course for free. Until the end of June, use the link below, and the course is yours to keep for 1 month. **Get it for free**.

.NET 10 just got a whole lot more lightweight.

You can now run a C# file directly with:

```
dotnet run app.cs
```

That's it. No `.csproj`. No `Program.cs`. No solution files. Just a single C# file.

This new feature, introduced in .NET 10 Preview 4, is a big step toward making C# more script-friendly, especially for quick utilities, dev tooling, and CLI-based workflows.

Why This Matters

For years, C# has been perceived as heavyweight for small scripts. Compare that to Python, Bash, or even JavaScript, where you can just write a file and run it.

That barrier is now gone.

You can now:

- Reference NuGet packages inline
- Share minimal reproducible examples without scaffolding a project

And it runs on **any OS** with the .NET SDK installed.

Minimal Example

Here's a simple script that prints today's date:

```
Console.WriteLine($"Today is {DateTime.Now:dddd, MMM dd yyyy}");
```

Run it:

```
dotnet run app.cs
```

Output:

```
Today is Saturday, Jun 14 2025
```

That's it. No boilerplate, no boring `Main()` method. Just top-level programs and C# code.

Referencing NuGet Packages

You can do this inline:

```
#:package Flurl.Http@4.0.2

using Flurl.Http;

var response = await "https://api.github.com"
    .WithHeader("Accept", "application/vnd.github.v3+json")
    .WithHeader("User-Agent", "dotnet-script")
    .GetAsync();

Console.WriteLine($"Status code: {response.StatusCode}");

Console.WriteLine(await response.GetJsonAsync<object>());
```

To run it:

```
dotnet run fetch.cs
```

Behind the scenes, the compiler downloads and restores NuGet dependencies automatically.

Real-World Use Case: Seeding SQL Data

Here's a script I recently used to seed some test data into my Postgres database.

```
#:package Dapper@2.1.66
#:package Npgsql@9.0.3

using Dapper;
using Npgsql;
```

```
using var connection = new NpgsqlConnection(connectionString);
await connection.OpenAsync();

using var transaction = connection.BeginTransaction();

Console.WriteLine("Creating tables...");

await connection.ExecuteAsync(@"
    CREATE TABLE IF NOT EXISTS users (
        id SERIAL PRIMARY KEY,
        name TEXT NOT NULL
    );
");

Console.WriteLine("Inserting users...");

for (int i = 1; i <= 10_000; i++)
{
    await connection.ExecuteAsync(
        "INSERT INTO users (name) VALUES (@Name);",
        new { Name = $"User {i}" });

    if (i % 1000 == 0)
    {
        Console.WriteLine($"Inserted {i} users...");
    }
}

transaction.Commit();

Console.WriteLine("Done!");
```

Why did I write this as a script? I didn't want to clutter my app with throwaway seed logic. I just needed a quick way to populate my database with test data. This script does exactly that, and I can run it with:

File-Level Directives: The Magic Behind It

The real power comes from file-level directives. These let you configure your app without leaving the .cs file:

Package References

```
#:package Dapper@2.1.66
#:package Npgsql@9.0.3
```

SDK Selection

```
#:sdk Microsoft.NET.Sdk.Web
```

This tells .NET to treat your file as a web application, enabling ASP.NET Core features:

```
#:sdk Microsoft.NET.Sdk.Web
#:package Microsoft.AspNetCore.OpenApi@9.*

var builder = WebApplication.CreateBuilder();

builder.Services.AddOpenApi();

var app = builder.Build();

app.MapOpenApi();

app.MapGet("/", () => "Hello from a file-based API!");
```

You now have a running web API. No project file. No `Startup.cs`. Just C# that does what you want.

MSBuild Properties

You can also set MSBuild properties directly in the file:

```
#:property LangVersion preview
#:property Nullable enable
```

When Your Script Grows Up

The brilliant part? When your file-based app gets complex enough to need project structure, converting is seamless:

```
dotnet project convert api.cs
```

This creates:

- A new folder named after your file
- A proper `.csproj` file with all your directives converted to MSBuild properties
- Your code moved to `api.cs` (or `Program.cs` if you prefer)
- Everything ready for full project development

Given our API example above, the generated `.csproj` looks like:

```
    <TargetFramework>net10.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.OpenApi" Version="9.*" />
  </ItemGroup>
</Project>
```

Your file-based app evolves naturally into a project-based app. No need to rewrite or restructure everything. This makes it easy to start small and grow as needed, without losing the simplicity of the initial script.

Takeaway

The bottom line is this: C# just became significantly more approachable. The barrier to entry dropped from "learn project files and MSBuild" to "write C# and run it."

For experienced developers, this is a productivity boost for scripting and prototyping. For newcomers, this removes the biggest stumbling block to getting started with C#.

The best part? Microsoft didn't create a separate scripting language or runtime. They made regular C# easier to use. Your file-based apps are real .NET applications that can grow into full projects when needed.

The ceremony is dead. Long live practical C#.

Whenever you're ready, there are 4 ways I can help you:

1. **Pragmatic Clean Architecture:** Join 4,000+ students in this comprehensive course that will teach you the system I use to ship production-ready applications using Clean Architecture. Learn how to apply the best practices of modern software architecture.

2. **Modular Monolith Architecture:** Join 2,000+ engineers in this in-depth course that will transform the way you build modern systems. You will learn the best practices for applying the Modular Monolith architecture in a real-world scenario.

3. **(NEW) Pragmatic REST APIs:** Join 1,100+ students in this course that will teach you how to build production-ready REST APIs using the latest ASP.NET Core features and best practices. It includes a fully functional UI application that we'll integrate with the REST API.

4. **Patreon Community:** Join a community of 1,000+ engineers and software architects. You will also unlock access to the source code I use in my YouTube videos, early access to future videos, and exclusive discounts for my courses.

Milan Jovanović

Contact