ASP.NET Core is a cross-platform, high-performance, [open-source](https://github.com/dotnet/aspnetcore) framework for building modern, cloud-enabled, Internet-connected apps.

With ASP.NET Core, you can:

* Build web apps and services, [Internet of Things (IoT)](https://www.microsoft.com/internet-of-things/) apps, and mobile backends.
* Use your favorite development tools on Windows, macOS, and Linux.
* Deploy to the cloud or on-premises.
* Run on [.NET](https://learn.microsoft.com/en-us/dotnet/core/introduction).

| **ASP.NET Core** | **ASP.NET 4.x** |
| --- | --- |
| Build for Windows, macOS, or Linux | Build for Windows |
| [Razor Pages](https://learn.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-9.0) is the recommended approach to create a Web UI as of ASP.NET Core 2.x. See also [MVC](https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-9.0), [Web API](https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-9.0), and [SignalR](https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-9.0). | Use [Web Forms](https://learn.microsoft.com/en-us/aspnet/web-forms), [SignalR](https://learn.microsoft.com/en-us/aspnet/signalr), [MVC](https://learn.microsoft.com/en-us/aspnet/mvc), [Web API](https://learn.microsoft.com/en-us/aspnet/web-api/), [WebHooks](https://learn.microsoft.com/en-us/aspnet/webhooks/), or [Web Pages](https://learn.microsoft.com/en-us/aspnet/web-pages) |
| Multiple versions per machine | One version per machine |
| Develop with [Visual Studio](https://visualstudio.microsoft.com/vs/), [Visual Studio for Mac](https://visualstudio.microsoft.com/vs/mac/), or [Visual Studio Code](https://code.visualstudio.com/) using C# or F# | Develop with [Visual Studio](https://visualstudio.microsoft.com/vs/) using C#, VB, or F# |
| Higher performance than ASP.NET 4.x | Good performance |
| [Use .NET Core runtime](https://learn.microsoft.com/en-us/dotnet/standard/choosing-core-framework-server) | Use .NET Framework runtime |

**Program.cs**

ASP.NET Core apps created with the web templates contain the application startup code in the Program.cs file. The Program.cs file is where:

* Services required by the app are configured.
* The app's request handling pipeline is defined as a series of [middleware components](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-9.0).

**Dependency injection (services)**

ASP.NET Core features built-in [dependency injection (DI)](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-9.0) that makes configured services available throughout an app. Services are added to the DI container with [WebApplicationBuilder.Services](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.webapplicationbuilder.services#microsoft-aspnetcore-builder-webapplicationbuilder-services), builder.Services in the preceding code. When the [WebApplicationBuilder](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.webapplicationbuilder) is instantiated, many [framework-provided services](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-9.0#framework-provided-services) are added automatically.

**Middleware**

The request handling pipeline is composed as a series of middleware components. Each component performs operations on an [HttpContext](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/http-context?view=aspnetcore-9.0) and either invokes the next middleware in the pipeline or terminates the request.

By convention, a middleware component is added to the pipeline by invoking a Use{Feature} extension method.

**Host**

On startup, an ASP.NET Core app builds a *host*. The host encapsulates all of the app's resources, such as:

* An HTTP server implementation
* Middleware components
* Logging
* Dependency injection (DI) services
* Configuration

There are three different hosts capable of running an ASP.NET Core app:

* [ASP.NET Core WebApplication](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis/webapplication?view=aspnetcore-9.0), also known as the [Minimal Host](https://learn.microsoft.com/en-us/aspnet/core/migration/50-to-60?view=aspnetcore-9.0#new-hosting-model)
* [.NET Generic Host](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/host/generic-host?view=aspnetcore-9.0) combined with ASP.NET Core's [ConfigureWebHostDefaults](https://learn.microsoft.com/en-us/dotnet/api/microsoft.extensions.hosting.generichostbuilderextensions.configurewebhostdefaults)
* [ASP.NET Core WebHost](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/host/web-host?view=aspnetcore-9.0)

The ASP.NET Core [WebApplication](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.webapplication) and [WebApplicationBuilder](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.webapplicationbuilder) types are recommended and are used in all the ASP.NET Core templates. WebApplication behaves similarly to the .NET Generic Host and exposes many of the same interfaces but requires fewer callbacks to configure. The ASP.NET Core [WebHost](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.webhost) is available only for backward compatibility.

The [WebApplicationBuilder.Build](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.webapplicationbuilder.build) method configures a host with a set of default options, such as:

* Use [Kestrel](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-9.0&tabs=windows#servers) as the web server and enable IIS integration.
* Load [configuration](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-9.0) from appsettings.json, environment variables, command line arguments, and other configuration sources.
* Send logging output to the console and debug providers.

**Non-web scenarios**

The Generic Host enables other types of apps to use cross-cutting framework extensions, such as logging, dependency injection (DI), configuration, and app lifetime management. For more information, see [.NET Generic Host in ASP.NET Core](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/host/generic-host?view=aspnetcore-9.0) and [Background tasks with hosted services in ASP.NET Core](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-9.0).

**Servers**

An ASP.NET Core app uses an HTTP server implementation to listen for HTTP requests. The server surfaces requests to the app as a set of [request features](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/request-features?view=aspnetcore-9.0) composed into an [HttpContext](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.httpcontext).

* [Windows](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-9.0&tabs=windows#tabpanel_1_windows)
* [macOS](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-9.0&tabs=windows#tabpanel_1_macos)
* [Linux](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-9.0&tabs=windows#tabpanel_1_linux)

ASP.NET Core provides the following server implementations:

* *Kestrel* is a cross-platform web server. Kestrel is often run in a reverse proxy configuration using [IIS](https://www.iis.net/). In ASP.NET Core 2.0 and later, Kestrel can be run as a public-facing edge server exposed directly to the Internet.
* *IIS HTTP Server* is a server for Windows that uses IIS. With this server, the ASP.NET Core app and IIS run in the same process.
* *HTTP.sys* is a server for Windows that isn't used with IIS.

Dotnet new console – dotnet restore will fire implicitly

Dotnet build – dotnet restore will fire implicitly

Dotnet run – dotnet restore will fire implicitly

Dotnet run –no-restore

.NET Core 2.1 includes enhancements and new features in the following areas:

## Tooling

dotnet watch -  provides a file system watcher that waits for a file to change before executing a designated set of commands.

dotnet dev-certs generates and manages certificates used during development in ASP.NET Core applications.

dotnet user-secrets manages the secrets in a user secret store in ASP.NET Core applications.

dotnet sql-cache creates a table and indexes in a Microsoft SQL Server database to be used for distributed caching.

dotnet ef is a tool for managing databases, [DbContext](https://docs.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.dbcontext) objects, and migrations in Entity Framework Core applications.

### **Global Tools**

.NET Core 2.1 supports Global Tools -- that is, custom tools that are available globally from the command line.

dotnet tool install -g dotnetsay

dotnet tool install

dotnet tool update

dotnet tool uninstall

dotnet tool list

## Roll forward

All .NET Core applications starting with the .NET Core 2.0 automatically roll forward to the latest minor version installed on a system.

Starting with .NET Core 2.0, if the version of .NET Core that an application was built with is not present at runtime, the application automatically runs against the latest installed minor version of .NET Core.

## Deployment

### **Self-contained application servicing**

dotnet publish now publishes self-contained applications with a serviced runtime version. When you publish a self-contained application with the .NET Core 2.1 SDK (v 2.1.300), your application includes the latest serviced runtime version known by that SDK. When you upgrade to the latest SDK, you’ll publish with the latest .NET Core runtime version. This applies for .NET Core 1.0 runtimes and later

## Windows Compatibility Pack

When you port existing code from the .NET Framework to .NET Core, you can use the [Windows Compatibility Pack](https://www.nuget.org/packages/Microsoft.Windows.Compatibility). It provides access to 20,000 more APIs than are available in .NET Core. These APIs include types in the [System.Drawing](https://docs.microsoft.com/en-us/dotnet/api/system.drawing) namespace, the [EventLog](https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.eventlog) class, WMI, Performance Counters, Windows Services, and the Windows registry types and members.

## JIT compiler improvements

.NET Core incorporates a new JIT compiler technology called tiered compilation (also known as adaptive optimization) that can significantly improve performance. Tiered compilation is an opt-in setting.

One of the important tasks performed by the JIT compiler is optimizing code execution. For little-used code paths, however, the compiler may spend more time optimizing code than the runtime spends running unoptimized code. Tiered compilation introduces two stages in JIT compilation:

* A **first tier**, which generates code as quickly as possible.
* A **second tier**, which generates optimized code for those methods that are executed frequently. The second tier of compilation is performed in parallel for enhanced performance.

## API changes

### **Span<T> and Memory<T>**

.NET Core 2.1 includes some new types that make working with arrays and other types of memory much more efficient.

* [System.Span<T>](https://docs.microsoft.com/en-us/dotnet/api/system.span-1) and [System.ReadOnlySpan<T>](https://docs.microsoft.com/en-us/dotnet/api/system.readonlyspan-1).
* [System.Memory<T>](https://docs.microsoft.com/en-us/dotnet/api/system.memory-1) and [System.ReadOnlyMemory<T>](https://docs.microsoft.com/en-us/dotnet/api/system.readonlymemory-1).

### **Brotli compression**

.NET Core 2.1 adds support for Brotli compression and decompression. Brotli is a general-purpose lossless compression algorithm that is defined in [RFC 7932](https://www.ietf.org/rfc/rfc7932.txt) and is supported by most web browsers and major web servers.

### **Sockets improvements**

.NET Core includes a new type, [System.Net.Http.SocketsHttpHandler](https://docs.microsoft.com/en-us/dotnet/api/system.net.http.socketshttphandler), and a rewritten [System.Net.Http.HttpMessageHandler](https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpmessagehandler), that form the basis of higher-level networking APIs.[System.Net.Http.SocketsHttpHandler](https://docs.microsoft.com/en-us/dotnet/api/system.net.http.socketshttphandler), for example, is the basis of the [HttpClient](https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclient) implementation. In previous versions of .NET Core, higher-level APIs were based on native networking implementations.

The sockets implementation introduced in .NET Core 2.1 has a number of advantages:

* A significant performance improvement when compared with the previous implementation.
* Elimination of platform dependencies, which simplifies deployment and servicing.
* Consistent behavior across all .NET Core platforms.

### **New cryptography APIs and cryptography improvements**

.NET Core 2.1 includes numerous enhancements to the cryptography APIs:

* [System.Security.Cryptography.Pkcs.SignedCms](https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.pkcs.signedcms) is available in the System.Security.Cryptography.Pkcs package. The implementation is the same as the [SignedCms](https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.pkcs.signedcms)class in the .NET Framework.
* New overloads of the [X509Certificate.GetCertHash](https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.x509certificates.x509certificate.getcerthash) and [X509Certificate.GetCertHashString](https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.x509certificates.x509certificate.getcerthashstring)methods accept a hash algorithm identifier to enable callers to get certificate thumbprint values using algorithms other than SHA-1.

# **Hosting .NET Core**

Hosting the .NET Core runtime is an advanced scenario and, in most cases, .NET Core developers don't need to worry about hosting because .NET Core build processes provide a default host to run .NET Core applications. In some specialized circumstances, though, it can be useful to explicitly host the .NET Core runtime, either as a means of invoking managed code in a native process or in order to gain more control over how the runtime works.

Deployment

1. Framework Dependent Deployment - For an FDD, you deploy only your app and any third-party dependencies. You don't have to deploy .NET Core, since your app will use the version of .NET Core that's present on the target system. This is the default deployment model for .NET Core apps.

Deploying an FDD has a number of advantages:

* You don't have to define the target operating systems that your .NET Core app will run on in advance. Because .NET Core uses a common PE file format for executables and libraries regardless of operating system, .NET Core can execute your app regardless of the underlying operating system. For more information on the PE file format, see [.NET Assembly File Format](https://docs.microsoft.com/en-us/dotnet/standard/assembly-format).
* The size of your deployment package is small. You only deploy your app and its dependencies, not .NET Core itself.
* Multiple apps use the same .NET Core installation, which reduces both disk space and memory usage on host systems.

There are also a few disadvantages:

* Your app can run only if the version of .NET Core that you target, or a later version, is already installed on the host system.
* It's possible for the .NET Core runtime and libraries to change without your knowledge in future releases. In rare cases, this may change the behavior of your app.

## Self-contained deployments (SCD)

For a self-contained deployment, you deploy your app and any required third-party dependencies along with the version of .NET Core that you used to build the app. Creating an SCD doesn't include the [native dependencies of .NET Core](https://github.com/dotnet/core/blob/master/Documentation/prereqs.md) on various platforms, so these must be present before the app runs.

FDD and SCD deployments use separate host executables, so you can sign a host executable for an SCD with your publisher signature.

### **Why deploy a self-contained deployment?**

Deploying a Self-contained deployment has two major advantages:

* You have sole control of the version of .NET Core that is deployed with your app. .NET Core can be serviced only by you.
* You can be assured that the target system can run your .NET Core app, since you're providing the version of .NET Core that it will run on.

It also has a number of disadvantages:

* Because .NET Core is included in your deployment package, you must select the target platforms for which you build deployment packages in advance.
* The size of your deployment package is relatively large, since you have to include .NET Core as well as your app and its third-party dependencies.
* Deploying numerous self-contained .NET Core apps to a system can consume significant amounts of disk space, since each app duplicates .NET Core files.

IHostingEnvironment

UseIISIntegration

 wwwroot folder

ASP.NET Core Module (ANCM) lets you run ASP.NET Core applications behind IIS and it works only with Kestrel; it isn’t compatible with WebListener. ANCM is a native IIS module that hooks into the IIS pipeline and redirects traffic to the backend ASP.NET Core application. ASP.NET Core applications run in a process separate from the IIS worker process, ANCM also does process management. ANCM starts the process for the ASP.NET Core application when the first request comes in and restarts it when it crashes. In short, it sits in IIS and routes the request for ASP.NET Core application to Kestral.

|  |
| --- |
| What are technologies discontinued in .NET Core? |
|  | * **Reflection** * **Appdomain** * **Remoting** * **Binary serialization** * **Sandboxing** |

* **global.json**

can define solution level settings in global.json file 

* **launchsettings.json**

can define project specific settings associated with each profile Visual Studio is configured to launch the application, including any environment variables that should be used. You can define framework for your project for compliation and debugging for specific profiles. 

* **appsettings.json**

to store custom application setting, DB connection strings,Logging etc 

* **bundleconfig.json**

can define the configuration for bundling and minification for the project. 

* **project.json**

storing all project level configuration settings 

* **bower.json**

Bower is a package manager for the web. Bower manages components that contain HTML, CSS, JavaScript, fonts or even image files. Bower installs the right versions of the packages you need and their dependencies

It is use to build up the HTTP pipeline via webHostBuilder.Use() chaining it all together with WebHostBuilder.Build() by using the builder pattern. It is available within the Microsoft.AspNet.Hosting namespace.The purpose of the Build method is to build the required services and a Microsoft.AspNetCore.Hosting.IWebHost which hosts a web application.

DotNet core 2.0 Servers

1.Http.Sys – Windows only ,support Windows authentication

2.Kestrel- Cross platfrom,highly optimized and recommended

DotNet Core Deployment Strategy

* 1. IIS on Windows
  2. Kestrel on Linux -AWS
  3. Azure App service- CI/CD using TFS or git
  4. Docker-