## **.NET Overview**

## **Runtime Targets and TFMs: Building for Specific .NET Platforms**

In a .NET project, a crucial setting in your project file (.csproj) is the **Target Framework Moniker (TFM)**. The TFM specifies the particular .NET runtime and its associated set of APIs that your project will build against. This is configured within the <TargetFramework> element.

**Examples of TFMs:**

* net8.0: Targets .NET 8
* net7.0, net6.0, net5.0: For previous .NET versions
* netcoreapp3.1: For .NET Core 3.1 (an older generation of .NET)
* net48: For .NET Framework 4.8 (the legacy Windows-only .NET)
* netstandard2.0: For .NET Standard 2.0 (which we'll discuss in detail)

**Example in .csproj:**

| <Project Sdk="Microsoft.NET.Sdk">   <PropertyGroup>  <TargetFramework>net8.0</TargetFramework>  </PropertyGroup>  </Project> |
| --- |

When you build your project, the TFM is encoded into the output assembly. An assembly built for a specific TFM can generally run on **newer (but not older)** runtimes of the *same .NET generation*. For instance, a library built for net6.0 will run on .NET 6, .NET 7, and .NET 8.

### **Multitargeting**

Sometimes, you might need your project to run on multiple, different .NET runtimes (e.g., both modern .NET and the older .NET Framework). You can achieve this through **multitargeting** by using the <TargetFrameworks> (plural) element and separating TFMs with semicolons:

| <PropertyGroup>  <TargetFrameworks>net8.0;net48</TargetFrameworks> </PropertyGroup> |
| --- |

When you multitarget, the compiler generates a separate output assembly for each specified target framework. This provides maximum compatibility but increases build complexity.

### **.NET Standard: The API Specification for Libraries**

Building libraries that support a wide range of .NET platforms can be challenging because different runtimes (.NET, .NET Framework, Xamarin/Mono) have overlapping but not identical sets of APIs. No single runtime is a pure superset of the others.

**.NET Standard** was introduced to solve this. It's **not a runtime itself**, but rather a **specification or a contract** that defines a minimum baseline of APIs (types and members) that are guaranteed to be available across a range of .NET runtimes. Think of it like an interface that different .NET runtimes implement. By targeting .NET Standard, you write a library once, and it can run on any .NET runtime that *implements* that version of .NET Standard.

#### **.NET Standard 2.0**

**.NET Standard 2.0** is the most widely adopted and useful version. A library targeting netstandard2.0 is highly compatible:

* It runs without modification on modern .NET (.NET 5+ down to .NET Core 2.0).
* It runs on .NET Framework (version 4.6.1 and later).
* It supports legacy platforms like UWP (from 10.0.16299+) and Mono 5.4+ (used by older Xamarin versions).

To target .NET Standard 2.0, you would use:

| <TargetFramework>netstandard2.0</TargetFramework> |
| --- |

Most of the common .NET APIs are available in .NET Standard 2.0.

#### **Other .NET Standards**

* **.NET Standard 2.1**: A superset of 2.0, but it is **not supported by .NET Framework**. This makes it less broadly useful for libraries needing to support legacy environments, typically only for modern .NET Core 3+ and Mono 6.4+.
* **Older .NET Standards (1.x)**: These versions (1.0, 1.1, etc.) are largely obsolete. They support very old runtimes but lack thousands of APIs present in 2.0, making them impractical for modern development.

### **.NET Framework and .NET Compatibility**

While .NET Standard aims for compatibility, sometimes you encounter older libraries that are *only* available for .NET Framework. Modern .NET 5+ and .NET Core projects **can reference .NET Framework assemblies**, but with caveats:

* An exception will be thrown at runtime if the .NET Framework assembly attempts to call an API that is not supported by the current .NET runtime.
* Complex dependencies often fail to resolve, making this approach most reliable for simple cases, like an assembly that merely wraps an unmanaged DLL.

### **Reference Assemblies**

When you target a TFM (like net8.0 or netstandard2.0), your project implicitly references a set of **reference assemblies**. These assemblies:

* Exist solely for the compiler's benefit during compilation.
* Contain only API signatures (types and members), not actual compiled code.
* Ensure that your code is checked against the correct API surface without requiring the full runtime to be installed for compilation.

At runtime, the "real" assemblies (containing the actual implementation) are loaded based on the specific runtime and platform your application is running on. This mechanism helps manage versioning and cross-platform compatibility effectively.

### **Runtime and C# Language Versions**

By default, the **C# language version** used by the compiler is determined by your project's TargetFramework. Newer C# language features often rely on new types or runtime capabilities introduced in later .NET versions.

| **Runtime Target** | **Default C# Version** |
| --- | --- |
| .NET 8 | C# 12 |
| .NET 7 | C# 11 |
| .NET 6 | C# 10 |
| .NET 5 | C# 9 |
| .NET Core 3.x & 2.x | C# 8 |
| .NET Framework / .NET Standard 2.0 | C# 7.3 |

You can override this default using the <LangVersion> element in your project file. However, using a newer C# version with an older runtime might result in compilation errors or runtime failures if the language feature relies on types unavailable in that older runtime.

## **The CLR and BCL: Core Components of .NET**

The .NET platform is built upon two core components:

* **Common Language Runtime (CLR):** The execution engine of .NET. It performs tasks like garbage collection, exception handling, and Just-In-Time (JIT) compilation of Intermediate Language (IL) code into machine code.
* **Base Class Library (BCL):** A vast collection of reusable types and APIs that provide fundamental functionalities. These are organized into namespaces.

Let's briefly survey some key areas within the BCL:

* **System Types:** Fundamental types in the System namespace (e.g., int, string, Exception, DateTime, Guid), math functions (Math), random numbers (Random), and type conversions (Convert).
* **Text Processing:** System.Text for StringBuilder (mutable strings) and text encodings (UTF-8). System.Text.RegularExpressions for pattern matching.
* **Collections:** Various data structures (lists, dictionaries) for managing items, found in System.Collections (non-generic), System.Collections.Generic (generic), System.Collections.Immutable, System.Collections.Frozen, etc.
* **Querying (LINQ):** Language-Integrated Query provides a consistent way to query local and remote data. Essential types are in System.Linq, System.Linq.Expressions, and System.Xml.Linq.
* **XML and JSON:** Comprehensive support for parsing, creating, and manipulating XML (System.Xml, System.Xml.Linq) and JSON (System.Text.Json, System.Text.Json.Nodes).
* **Diagnostics:** Tools for logging, assertions, interacting with processes, and performance monitoring (System.Diagnostics).
* **Concurrency and Asynchrony:** Support for multithreading, asynchronous operations (async/await), tasks, and parallelism (System.Threading, System.Threading.Tasks).
* **Streams and Input/Output (I/O):** A stream-based model for reading and writing data to files, networks, etc. (System.IO).
* **Networking:** APIs for standard network protocols like HTTP, TCP/IP, and SMTP (System.Net, System.Net.Http, System.Net.Mail, System.Net.Sockets).
* **Assemblies, Reflection, and Attributes:** Mechanisms for inspecting program metadata at runtime, dynamically invoking methods, and creating custom attributes (System.Reflection, System.Reflection.Emit).
* **Dynamic Programming:** Support for dynamic objects and interoperability with dynamic languages (System.Dynamic).
* **Cryptography:** Comprehensive support for hashing and encryption protocols (System.Security.Cryptography).
* **Advanced Threading:** Low-level threading constructs like signaling and locks (System.Threading).
* **Parallel Programming:** Libraries for leveraging multicore processors (System.Threading.Tasks, PLINQ).
* **Span and Memory:** Performance-optimized types for memory management that reduce garbage collector load (System, System.Memory).
* **Native and COM Interoperability:** Tools for calling functions in unmanaged DLLs and interacting with COM components (System.Runtime.InteropServices).
* **Serialization:** Systems for saving and restoring objects to binary or text formats (e.g., JSON, XML serializers).
* **The Roslyn Compiler:** The C# compiler itself is written in C# and its libraries (Roslyn) are available for code analysis and refactoring.

## **Application Layers: Building User Interfaces with .NET**

.NET supports building various types of applications, primarily categorized as **thin-client** (web applications) and **rich-client** (desktop or mobile applications).

### **Thin-Client Applications (Web)**

* **ASP.NET Core:** This is the modern, cross-platform (Windows, Linux, macOS) framework for building:
  + **Websites:** Using the Model-View-Controller (MVC) pattern or Razor Pages.
  + **REST-based Web APIs:** For services consumed by other applications.
  + **Microservices.**
  + **Single-Page Applications (SPAs):** Integrates well with client-side frameworks like React and Angular.
  + **Blazor:** Allows writing client-side web code directly in C# instead of JavaScript, running either on the server (Blazor Server) or in the browser via WebAssembly (Blazor WebAssembly).

ASP.NET Core is lightweight, modular, and not dependent on older ASP.NET Web Forms technologies. Its advantages include zero client-side deployment, platform independence for the client (just needs a web browser), and easy updates.

### **Rich-Client Applications (Desktop and Mobile)**

For applications that users download and install, .NET offers several UI technologies:

#### **Windows Desktop (Windows-only)**

* **WPF (Windows Presentation Foundation):** Introduced in 2006, WPF is a powerful UI framework that renders controls using DirectX.  
  + **Advantages:** Supports sophisticated graphics, 3D, multimedia, true transparency, DPI-awareness, flexible layout, reliable data binding, and declarative UI in XAML for better separation of concerns. It's excellent for modern UIs and complex business applications.
  + **Disadvantages:** Has a steeper learning curve due to its complexity.
  + **Namespaces:** System.Windows and its subnamespaces (except System.Windows.Forms).
* **Windows Forms:** Shipped with the first version of .NET Framework (2000), Windows Forms is a simpler, more mature technology built on GDI+ and Win32 controls.  
  + **Advantages:** Relatively simple to learn, good for traditional Windows applications, and widely used for legacy applications.
  + **Disadvantages:** Less capable graphically (flickering, no true transparency), poor DPI-awareness by default, limited layout flexibility, and often difficult to customize complex controls.
  + **Namespaces:** System.Windows.Forms and System.Drawing.

#### **UWP (Universal Windows Platform) and WinUI 3 (Windows App SDK)**

* **UWP:** Designed for "Universal" Windows 10+ apps (desktop, Xbox, HoloLens). It uses XAML and is somewhat similar to WPF.  
  + **Key Differences:** Primarily distributed via the Windows Store, runs in a sandbox (limited file access, no admin elevation), and relies on WinRT APIs tied to specific Windows OS versions.
  + **Limitations:** Its sandbox and OS version dependency limited its popularity.
* **WinUI 3 (Windows App SDK):** Microsoft's successor to UWP, addressing its limitations.  
  + **Improvements:** WinRT APIs are part of the runtime (not OS-dependent), better integration with Windows Desktop, can run outside the Windows Store sandbox, and runs atop the latest .NET.
  + **Current Status:** Has not yet gained the widespread popularity of classic desktop APIs, and still requires a separate end-user download. It currently does not support Xbox or HoloLens.

#### **MAUI (Multi-platform App UI, formerly Xamarin)**

* **MAUI:** Allows you to develop **cross-platform mobile apps** in C# targeting iOS and Android. It also supports cross-platform desktop applications for macOS (via Catalyst) and Windows (via Windows App SDK).
* **Mono Integration:** Historically, Xamarin used the Mono runtime. Since .NET 6, Mono's public interface merged with .NET, making Mono effectively an implementation of .NET itself.
* **Features:** Includes a unified project interface, hot reloading, and support for Blazor Desktop/hybrid apps.

**Third-Party Alternatives:** Beyond Microsoft's offerings, libraries like Avalonia provide cross-platform UI development (including Linux support), sometimes offering more flexibility than MAUI in certain desktop scenarios.