

# Introducing .NET Standard



Immo

September 26th, 2016

**Questions?** Check out the [.NET Standard FAQ](#). You can find the latest version of the compatibility matrix [here](#).

In my last post, I talked about how we want to [make porting to .NET Core easier](#). In this post, I'll focus on how we're making this plan a reality with .NET Standard. We'll cover which APIs we plan to include, how cross-framework compatibility will work, and what all of this means for .NET Core.

If you're interested in details, this post is for you. But don't worry if you don't have time or you're not interested in details: you can just read the TL;DR section.

## For the impatient: TL;DR

.NET Standard solves the code sharing problem for .NET developers across all platforms by bringing all the APIs that you expect and love across the environments that you need: desktop applications, mobile apps & games, and cloud services:

- .NET Standard is a set of APIs that all .NET platforms have to implement. This unifies the .NET platforms and prevents future fragmentation.
- .NET Standard 2.0 will be implemented by .NET Framework, .NET Core, and Xamarin. For .NET Core, this will add many of the existing APIs that have been requested.
- .NET Standard 2.0 includes a compatibility shim for .NET Framework binaries, significantly increasing the set of libraries that you can reference from your .NET Standard libraries.
- .NET Standard will replace Portable Class Libraries (PCLs) as the tooling story for building multi-platform .NET libraries.
- You can see the .NET Standard API definition in the [dotnet/standard](#) repo on GitHub.

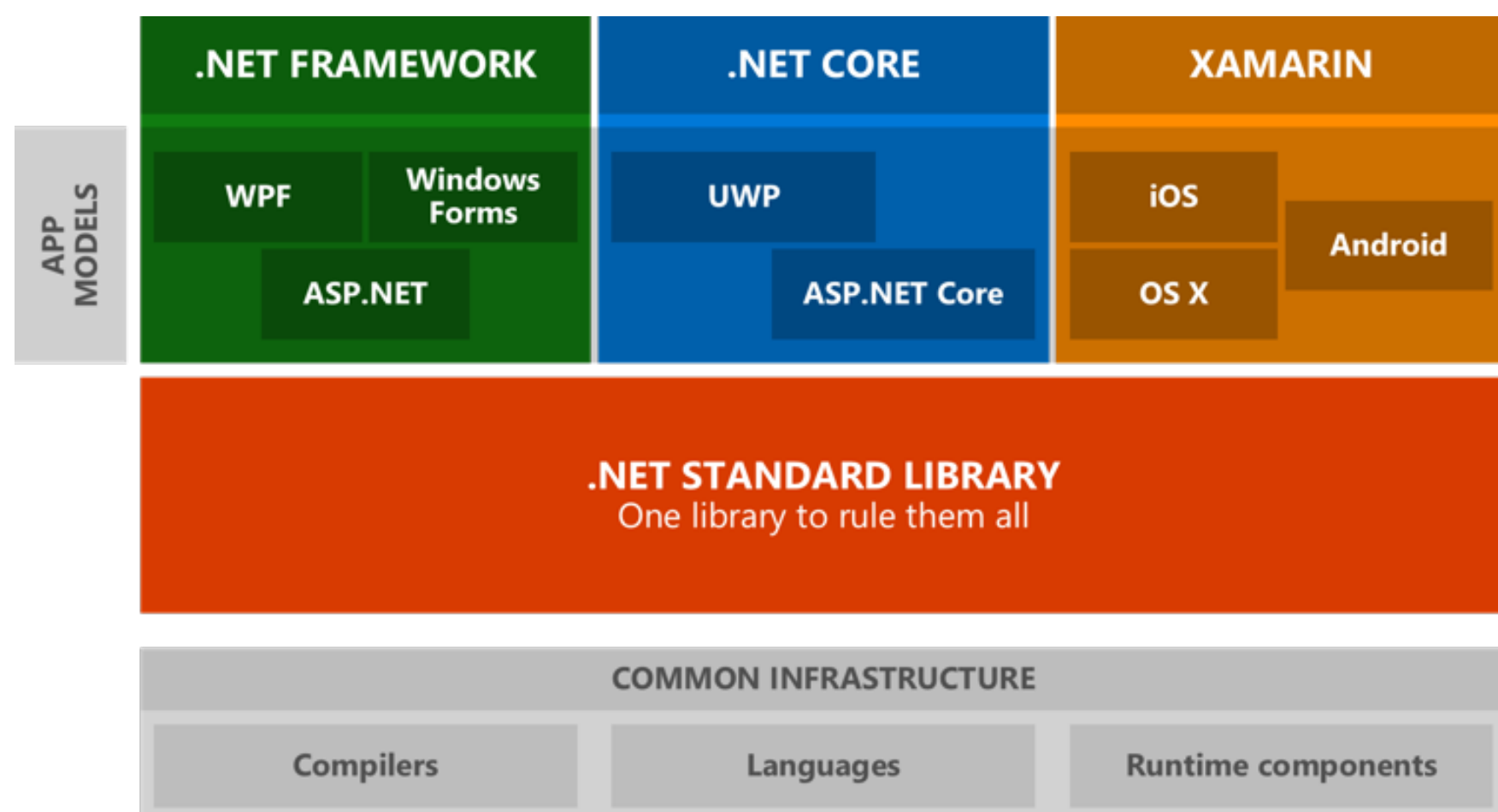
## Why do we need a standard?

As explained in detail in the post [Introducing .NET Core](#), the .NET platform was forked quite a bit over the years. On the one hand, this is actually a really good thing. It allowed tailoring .NET to fit the needs that a single platform wouldn't have been able to. For example, the .NET Compact Framework was created to fit into the (fairly) restrictive footprint of phones in the 2000 era. The same is true today: Unity (a fork of Mono) runs on more than 20 platforms. Being able to fork and customize is an important capability for any technology that requires reach.

But on the other hand, this forking poses a massive problem for developers writing code for multiple .NET platforms because there isn't a unified class library to target:

There are currently three major flavors of .NET, which means you have to master three different base class libraries in order to write code that works across all of them. Since the industry is much more diverse now than when .NET was originally created it's safe to assume that we're not done with creating new .NET platforms. Either Microsoft or someone else will build new flavors of .NET in order to support new operating systems or to tailor it for specific device capabilities.

This is where the .NET Standard comes in:



For developers, this means they only have to master one base class library. Libraries targeting .NET Standard will be able to run on all .NET platforms. And platform providers don't have to guess which APIs they need to offer in order to consume the libraries available on NuGet.

**Applications.** In the context of applications you don't use .NET Standard directly. However, you still benefit indirectly. First of all, .NET Standard makes sure that all .NET platforms share the same API shape for the base class library. Once you learn how to use it in your desktop application you know how to use it in your mobile application or your cloud service. Secondly, with .NET Standard most class libraries will become available everywhere, which means the consistency at the base layer will also apply to the larger .NET library ecosystem.

**Portable Class Libraries.** Let's contrast this with how Portable Class Libraries (PCL) work today. With PCLs, you select the platforms you want to run on and the tooling presents you with the resulting API set you can use. So while the tooling helps you to produce binaries that work on multiple platforms, it still forces you to think about different base class libraries. With .NET Standard you have a single base class library. Everything in it will be supported across all .NET platforms — current ones as well as future ones. Another key aspect is that the API availability in .NET Standard is very predictable: higher version equals more APIs. With PCLs, that's not necessarily the case: the set of available APIs is the result of the intersection between the selected platforms, which doesn't always produce an API surface you can easily predict.

**Consistency in APIs.** If you compare .NET Framework, .NET Core, and Xamarin/Mono, you'll notice that .NET Core offers the smallest API surface (excluding OS-specific APIs). The first inconsistency is having drastic differences in the availability of foundational APIs (such as networking- and crypto APIs). The second problem .NET Core introduced was having differences in the API shape of core pieces, especially in reflection. Both inconsistencies are the primary reason why porting code to .NET Core is much harder than it should be. By creating the .NET Standard we're codifying the requirement of having consistent APIs across all .NET platforms, and this includes availability as well as the shape of the APIs.

**Versioning and Tooling.** As I mentioned in [Introducing .NET Core](#) our goal with .NET Core was to lay the foundation for a portable .NET platform that can unify APIs in shape and implementation. We intended it to be the next version of portable class libraries. Unfortunately, it didn't result in a great tooling experience. Since our goal was to represent any .NET platform we had to break it up into smaller NuGet packages. This works reasonably well if all these components can be deployed with the application because you can update them independently. However, when you target an abstract specification, such as PCLs or the .NET Standard, this story doesn't work so well because there is a very specific combination of versions that will allow you to run on the right set of platforms. In order to avoid that issue, we've defined .NET Standard as a single NuGet package. Since it only represents the set of required APIs, there is no need to break it up any further because all .NET platforms have to support it in its entirety anyways. The only important dimension is its version, which acts like an API level: the higher the version, the more APIs you have, but the lower the version, the more .NET platforms have already implemented it.

To summarize, we need .NET Standard for two reasons:

1. **Driving force for consistency.** We want to have an agreed upon set of required APIs that all .NET platforms have to implement in order to gain access to the .NET library ecosystem.
2. **Foundation for great cross-platform tooling.** We want a simplified tooling experience that allows you to target the commonality of all .NET platforms by choosing a single version number.

## What's new in .NET Standard 2.0?

When [we shipped .NET Core 1.0](#), we also introduced .NET Standard. There are multiple versions of the .NET Standard in order to represent the API availability across all current platforms. The following table shows which version of an existing platform is compatible with a given version of .NET Standard:

| .NET Platform              | .NET Standard |     |       |     |       |       |       |       |
|----------------------------|---------------|-----|-------|-----|-------|-------|-------|-------|
|                            | 1.0           | 1.1 | 1.2   | 1.3 | 1.4   | 1.5   | 1.6   | 2.0   |
| .NET Core                  | →             | →   | →     | →   | →     | →     | 1.0   | vNext |
| .NET Framework             | →             | 4.5 | 4.5.1 | 4.6 | 4.6.1 | 4.6.2 | vNext | 4.6.1 |
| Xamarin.iOS                | →             | →   | →     | →   | →     | →     | →     | vNext |
| Xamarin.Android            | →             | →   | →     | →   | →     | →     | →     | vNext |
| Universal Windows Platform | →             | →   | →     | →   | 10.0  | →     | →     | vNext |
| Windows                    | →             | 8.0 | 8.1   |     |       |       |       |       |
| Windows Phone              | →             | →   | 8.1   |     |       |       |       |       |
| Windows Phone Silverlight  | 8.0           |     |       |     |       |       |       |       |

The arrows indicate that the platform supports a higher version of .NET Standard. For instance, .NET Core 1.0 supports the .NET Standard version 1.6, which is why there are arrows pointing to the right for the lower versions 1.0 – 1.5.

You can use this table to understand what the highest version of .NET Standard is that you can target, based on which .NET platforms you intend to run on. For instance, if you want to run on .NET Framework 4.5 and .NET Core 1.0, you can at most target .NET Standard 1.1.

You can also see which platforms will support .NET Standard 2.0:

- We'll ship updated versions of .NET Core, Xamarin, and UWP that will add all the

necessary APIs for supporting .NET Standard 2.0.

- .NET Framework 4.6.1 already implements all the APIs that are part of .NET Standard 2.0. Note that this version appears twice; I'll cover later why that is and how it works.

.NET Standard is also compatible with Portable Class Libraries. The mapping from PCL profiles to .NET Standard versions is listed in [our documentation](#).

From a library targeting .NET Standard you'll be able to reference two kinds of other libraries:

- **.NET Standard**, if their version is lower or equal to the version you're targeting.
- **Portable Class Libraries**, if their profile can be mapped to a .NET Standard version and that version is lower or equal to the version you're targeting.

Graphically, this looks as follows:

Unfortunately, the adoption of PCLs and .NET Standard on NuGet isn't as high as it would need to be in order to be a friction free experience. This is how many times a given target occurs in packages on NuGet.org:

| Target         | Occurrences |
|----------------|-------------|
| .NET Framework | 46,894      |
| .NET Standard  | 1,886       |
| Portable       | 4,501       |

As you can see, it's quite clear that the vast majority of class libraries on NuGet are targeting .NET Framework. However, we know that a large number of these libraries are only using APIs we'll expose in .NET Standard 2.0.

In .NET Standard 2.0, we'll make it possible for libraries that target .NET Standard to *also* reference existing .NET Framework binaries through a compatibility shim:

Of course, this will only work for cases where the .NET Framework library uses APIs that are available for .NET Standard. That's why this isn't the preferred way of building libraries you intend to use across different .NET platforms. However, this compatibility shim provides a bridge that enables you to convert your libraries to .NET Standard without having to give up referencing existing libraries that haven't been converted yet.

If you want to learn more about how the compatibility shim works, take a look at the [specification for .NET Standard 2.0](#).

## .NET Standard 2.0 breaking change: adding .NET Framework 4.6.1 compatibility

A standard is only as useful as there are platforms implementing it. At the same time, we want to make the .NET Standard meaningful and useful in and of itself, because that's the API surface that is available to libraries targeting the standard:

- **.NET Framework**. .NET Framework 4.6.1 has the highest adoption, which makes it the most attractive version of .NET Framework to target. Hence, we want to make sure that it can implement .NET Standard 2.0.
- **.NET Core**. As mentioned above, .NET Core has a much smaller API set than .NET Framework or Xamarin. Supporting .NET Standard 2.0 means that we need to extend the surface area significantly. Since .NET Core doesn't ship with the OS but with the app, supporting .NET Standard 2.0 only requires updates to the SDK and our NuGet packages.



- **Xamarin.** Xamarin already supports most of the APIs that are part of .NET Standard. Updating works similar to .NET Core — we hope we can update Xamarin to include all APIs that are currently missing. In fact, the majority of them were already added to the stable Cycle 8 release/Mono 4.6.0.

The table listed earlier shows which versions of .NET Framework supports which version of .NET Standard:

|                | 1.4   | 1.5   | 1.6   | 2.0   |
|----------------|-------|-------|-------|-------|
| .NET Framework | 4.6.1 | 4.6.2 | vNext | 4.6.1 |

Following normal versioning rules one would expect that .NET Standard 2.0 would only be supported by a newer version of .NET Framework, given that the latest version of .NET Framework (4.6.2) only supports .NET Standard 1.5. This would mean that the libraries compiled against .NET Standard 2.0 would not run on the vast majority of .NET Framework installations.

In order to allow .NET Framework 4.6.1 to support .NET Standard 2.0, we had to remove all the APIs from .NET Standard that were introduced in [.NET Standard 1.5 and 1.6](#).

You may wonder what the impact of that decision is. We ran an analysis of all packages on NuGet.org that target .NET Standard 1.5 or later and use any of these APIs. At the time of this writing we only found six non-Microsoft owned packages that do. We'll reach out to those package owners and work with them to mitigate the issue. From looking at their usages, it's clear that their calls can be replaced with APIs that are coming with .NET Standard 2.0.

In order for these package owners to support .NET Standard 1.5, 1.6 and 2.0, they will need to cross-compile to target these versions specifically. Alternatively, they can choose to target .NET Standard 2.0 and higher given the broad set of platforms that support it.

## What's in .NET Standard?

In order to decide which APIs will be part of .NET Standard we used the following process:

- **Input.** We start with all the APIs that are available in both .NET Framework and in Xamarin.
- **Assessment.** We classify all these APIs into one of two buckets:
  1. **Required.** APIs that we want all platforms to provide and we believe can be implemented cross-platform, we label as *required*.
  2. **Optional.** APIs that are platform-specific or are part of legacy technologies we label as *optional*.

Optional APIs aren't part of .NET Standard but are available as separate NuGet packages. We try to build these as libraries targeting .NET Standard so that their implementation can be consumed from any platform, but that might not always be feasible for platform specific APIs (e.g. Windows registry).

In order to make some APIs optional we may have to remove other APIs that are part of the required API set. For example, we decided that **AppDomain** is in .NET Standard while Code Access Security (CAS) is a legacy component. This requires us to remove all members from **AppDomain** that use types that are part of CAS, such as overloads on **CreateDomain** that accept **Evidence**.

The .NET Standard API set, as well as our proposal for optional APIs will be reviewed by the [.NET Standard's review body](#).

Here is the high-level summary of the API surface of .NET Standard 2.0:

If you want to look at the specific API set of .NET Standard 2.0, you can take a look at the [.NET Standard GitHub repository](#). Please note that .NET Standard 2.0 is a work in progress, which means some APIs might be added, while some might be removed.

## Can I still use platform-specific APIs?

One of the biggest challenges in creating an experience for multi-platform class libraries is to avoid only having the lowest-common denominator while also making sure you don't accidentally create libraries that are much less portable than you intend to.

In PCLs we've solved the problem by having multiple profiles, each representing the intersection of a set of platforms. The benefit is that this allows you to max out the API surface between a set of targets. The .NET Standard represents the set of APIs that all .NET platforms have to implement.

This brings up the question how we model APIs that cannot be implemented on all platforms:

- **Runtime specific APIs.** For example, the ability to generate and run code on the fly using reflection emit. This cannot work on .NET platforms that do not have a JIT compiler, such as .NET Native on UWP or via Xamarin's iOS tool chain.
- **Operating system specific APIs.** In .NET we've exposed many APIs from Win32 in order to make them easier to consume. A good example is the Windows registry. The implementation depends on the underlying Win32 APIs that don't have equivalents on other operating systems.

We have a couple of options for these APIs:

1. **Make the API unavailable.** You cannot use APIs that do not work across all .NET platforms.
2. **Make the API available but throw `PlatformNotSupportedException`.** This would mean that we expose all APIs regardless of whether they are supported everywhere or not. Platforms that do not support them provide the APIs but throw `PlatformNotSupportedException`.
3. **Emulate the API.** Mono implements the registry as an API over `.ini` files. While that doesn't work for apps that use the registry to read information about the OS, it works quite well for the cases where the application simply uses the registry to store its own state and user settings.

We believe the best option is a combination. As mentioned above we want the .NET Standard to represent the set of APIs that all .NET platforms are required to implement. We want to make this set sensible to implement while ensuring popular APIs are present so that writing cross-platform libraries is easy and intuitive.

Our general strategy for dealing with technologies that are only available on some .NET platforms is to make them NuGet packages that sit above the .NET Standard. So if you create a .NET Standard-based library, it'll not reference these APIs by default. You'll have to add a NuGet package that brings them in.

This strategy works well for APIs that are self-contained and thus can be moved into a separate package. For cases where individual members or types cannot be implemented everywhere, we'll use the second and third approach: platforms have to have these members but they can decide to throw or emulate them.

Let's look at a few examples and how we plan on modelling them:

- **Registry.** The Windows registry is a self-contained component that will be provided as a separate NuGet package (e.g. `Microsoft.Win32.Registry`). You'll be able to consume it from .NET Core, but it will only work on Windows. Calling registry APIs from any other OS will result in `PlatformNotSupportedException`. You're expected to guard your calls appropriately or making sure your code will only ever run on Windows. We're considering improving our tooling to help you with detecting these cases.
- **AppDomain.** The `AppDomain` type has many APIs that aren't tied to creating app domains, such as getting the list of loaded assemblies or registering an unhandled exception handler. These APIs are heavily used throughout the .NET library ecosystem. For this case, we decided it's much better to add this type to .NET Standard and let the few APIs that deal with app domain creation throw exceptions on platforms that don't support that, such as .NET Core.
- **Reflection Emit.** Reflection emit is reasonably self-contained and thus we plan on following the model as Registry, above. There are other APIs that logically depend on being able to emit code, such as the expression tree's `Compile` method or the ability to compile regexes. In some cases we'll emulate their behavior (e.g. interpreting expression trees instead of compiling them) while in other cases we'll throw (e.g. when compiling regexes).

In general, you can always work around APIs that are unavailable in .NET Standard by targeting specific .NET platforms, like you do today. We're thinking about ways how we can improve our tooling to make the transitions between being platform-specific and being platform-agnostic more fluid so that you can always choose the best option for your situation and not being cornered by earlier design choices.

To summarize:

- We'll expose concepts that might not be available on all .NET platforms.
- We generally make them individual packages that you have to explicitly reference.
- In rare cases, individual members might throw exceptions.

The goal is to make .NET Standard-based libraries as powerful and as expressive as possible while making sure you're aware of cases where you take dependencies on technologies that might not work everywhere.

## What does this mean for .NET Core?

We [designed .NET Core](#) so that its reference assemblies are the .NET portability story. This made it harder to add new APIs because adding them in .NET Core preempts the decision on whether these APIs are made available everywhere. Worse, due to versioning rules, it also means we have to decide which combination of APIs are made available in which order.

**Out-of-band delivery.** We've tried to work this around by making those APIs available "out-of-band" which means making them new components that can sit on top of the existing APIs. For technologies where this is easily possible, that's the preferred way because it also means any .NET developer can play with the APIs and give us feedback. We've done that for immutable collections with great success.

**Implications for runtime features.** However, for features that require runtime work, this is much harder because we can't just give you a NuGet package that will work. We also have to give you a way to get an updated runtime. That's harder on platforms that have a system wide runtime (such as .NET Framework) but is also harder in general because we have multiple runtimes for different purposes (e.g. JIT vs AOT). It's not practical to innovate across all these spectrums at once. The nice thing about .NET Core is that this platform is designed to be fully self-contained. So for the future, we're more likely to leverage this capability for experimentation and previewing.



**Splitting .NET Standard from .NET Core.** In order to be able to evolve .NET Core independently from other .NET platforms we've divorced the portability mechanism (which I referred to earlier) from .NET Core. .NET Standard is defined as an independent reference assembly that is satisfied by all .NET platforms. Each of the .NET platforms uses a different set of reference assemblies and thus can freely add new APIs in whatever cadence they choose. We can then, after the fact, make decisions around which of these APIs are added to .NET Standard and thus should become universally available.

Separating portability from .NET Core helps us to speed up development of .NET Core and makes experimentation of newer features much simpler. Instead of artificially trying to design features to sit on top of existing platforms, we can simply modify the layer that needs to be modified in order to support the feature. We can also add the APIs on the types they logically belong to instead of having to worry about whether that type has already shipped in other platforms.

Adding new APIs in .NET Core isn't a statement whether they will go into the .NET Standard but our goal for .NET Standard is to create and maintain consistency between the .NET platforms. So new members on types that are already part of the standard will be automatically considered when the standard is updated.

## As a library author, what should I do now?

As a library author, you should consider switching to .NET Standard because it will replace Portable Class Libraries for targeting multiple .NET platforms.

In case of .NET Standard 1.x the set of available APIs is very similar to PCLs. But .NET Standard 2.x will have a significantly bigger API set and will also allow you to depend on libraries targeting .NET Framework.

The key differences between PCLs and .NET Standard are:

- **Platform tie-in.** One challenge with PCLs is that while you target multiple platforms, it's still a specific set. This is especially true for NuGet packages as you have to list the platforms in the lib folder name, e.g. `portable-net45+win8`. This causes issues when new platforms show up that support the same APIs. .NET Standard doesn't have this problem because you target a version of the standard which doesn't include any platform information, e.g. `netstandard1.4`.
- **Platform availability.** PCLs currently support a wider range of platforms and not all profiles have a corresponding .NET Standard version. Take a look at the [documentation](#) for more details.
- **Library availability.** PCLs are designed to enforce that you cannot take dependencies on APIs and libraries that the selected platforms will not be able to run. Thus, PCL projects will only allow you to reference other PCLs that target a superset of the platforms your PCL is targeting. .NET Standard is similar, but it additionally allows referencing .NET Framework binaries, which are the de facto exchange currency in the library ecosystem. Thus, with .NET Standard 2.0 you'll have access to a much larger set of libraries.

In order to make an informed decision, I suggest you:

1. Use [API Port](#) to see how compatible your code base is with the various versions of .NET Standard.
2. Look at the [.NET Standard documentation](#) to ensure you can reach the platforms that are important to you.

For example, if you want to know whether you should wait for .NET Standard 2.0 you can check against both, .NET Standard 1.6 and .NET Standard 2.0 by downloading the [API Port](#) command line tool and run it against your libraries like so:



```
> apiport analyze -f C:srcmylibs -t ".NET Standard,Version=1.6"^
-t ".NET Standard,Version=2.0"
```

**Note:** .NET Standard 2.0 is still work in progress and therefore API availability is subject to change. I also suggest that you watch out for the APIs that are available in .NET Standard 1.6 but are [removed from .NET Standard 2.0](#).

## Summary

We’ve created .NET Standard so that sharing and re-using code between multiple .NET platforms becomes much easier.

With .NET Standard 2.0, we’re focusing on compatibility. In order to support .NET Standard 2.0 in .NET Core and UWP, we’ll be extending these platforms to include many more of the existing APIs. This also includes a compatibility shim that allows referencing binaries that were compiled against the .NET Framework.

Moving forward, we recommend that you use .NET Standard instead of Portable Class Libraries. The tooling for targeting .NET Standard 2.0 will ship in the same timeframe as the upcoming release of Visual Studio, code-named “Dev 15”. You’ll reference .NET Standard as a NuGet package. It will have first class support from Visual Studio, VS Code as well as Xamarin Studio.

You can follow our progress via our new [dotnet/standard](#) GitHub repository.

Please let us know what you think!



[Immo Landwerth](#)

Program Manager, .NET

Follow Immo



Posted in [Uncategorized](#) Tagged [.NET Core](#), [.net framework](#), [bcl](#), [nuget](#), [open source](#), [portable class libraries](#), [xamarin](#)

Log in to comment

Log In

Log In

- [efe özkel](#)

2019-03-05 06:45:15

thank you good post really like. [agario pvp](#)
- [Alexsa Denia](#)

2019-04-23 23:49:15

great post. [maspedia](#)

Relevant Links

[.NET Download](#)

[.NET Hello World](#)

[.NET Meetup Events](#)

[.NET Documentation](#)

[.NET API Browser](#)

[.NET SDKs](#)

.NET Application Architecture Guides

[Web apps with ASP.NET Core](#)

[Mobile apps with Xamarin.Forms](#)

[Microservices with Docker Containers](#)

[Modernizing existing .NET apps to the cloud](#)

Top bloggers



[Bertrand Le Roy](#)  
Senior Software Engineer



[The .NET Team](#)



[Rich Lander \[MSFT\]](#)  
Program Manager

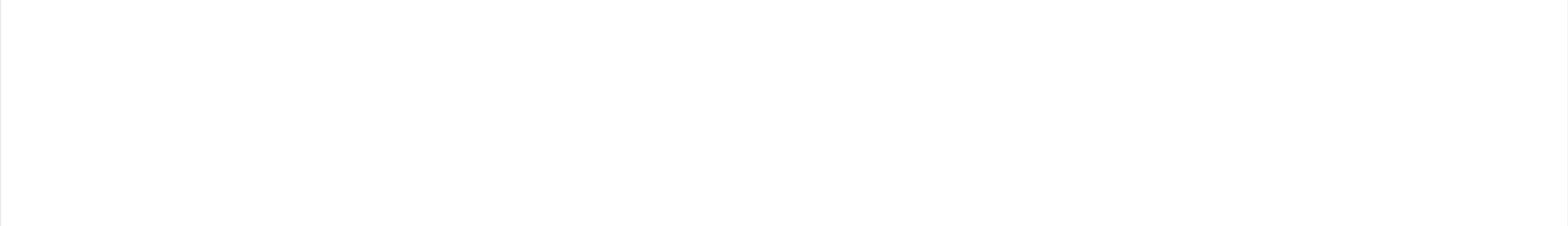


[Immo Landwerth \[MSFT\]](#)  
Program Manager



[CLR Team](#)

Twitter Feed



Stay informed



What's new

Surface Pro 6

Surface Laptop 2

Surface Go

Xbox One X

Xbox One S

VR & mixed reality

Windows 10 apps

Office apps

Microsoft Store

Account profile

Download Center

Microsoft Store support

Returns

Order tracking

Store locations

Buy online, pick up in store

Education

Microsoft in education

Office for students

Office 365 for schools

Deals for students & parents

Microsoft Azure in education

Enterprise

Azure

AppSource

Automotive

Government

Healthcare

Manufacturing

Financial services

Retail

Developer

Microsoft Visual Studio

Windows Dev Center

Developer Network

TechNet

Microsoft developer program

Channel 9

Office Dev Center

Microsoft Garage

Company

Careers

About Microsoft

Company news


Privacy at Microsoft

Investors

Diversity and inclusion

Accessibility

Security

 English (United States)