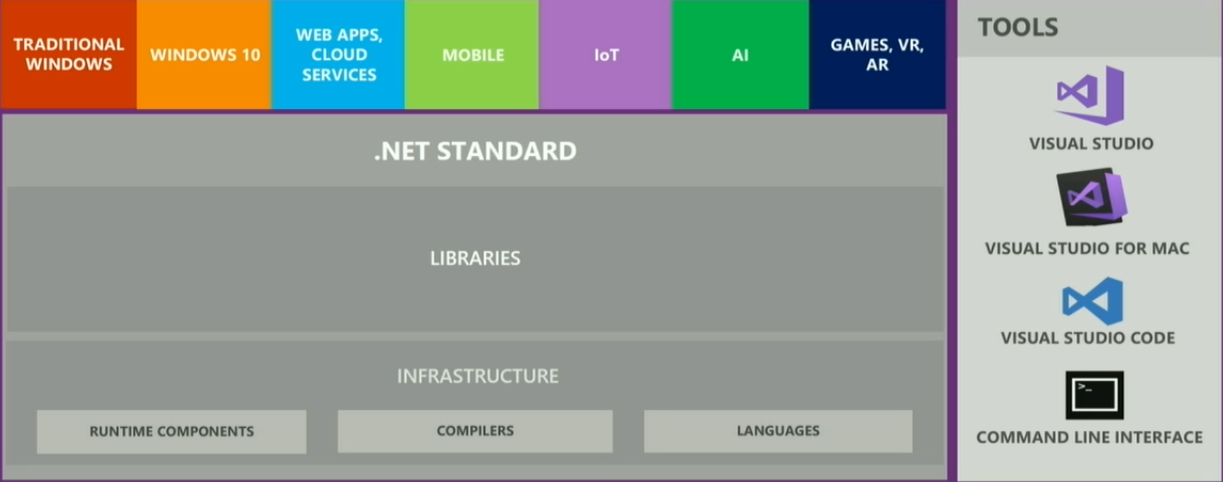
**.Net Standard 2.0**

**Introduction**

The **.NET Standard** is a formal specification of .NET APIs that are intended to be available on all .NET implementations.The motivation behind the .NET Standard is establishing greater uniformity in the .NET ecosystem.

The .NET Standard enables the following key scenarios:

* Defines uniform set of BCL APIs for all .NET implementations to implement, independent of workload.
* Enables developers to produce portable libraries that are usable across .NET implementations, using this same set of APIs.
* Reduces or even eliminates conditional compilation of shared source due to .NET APIs, only for OS APIs.



**Background**

Portable Class library is a reusable library project where we specify the .Net Platforms we are planning to target during creation of library.

**Advantage**: allowed code sharing between multiple platforms.

**Disadvantage**:

* Adding a new platform support requires re-compiling against the new set of target platforms and removing the API references that are not part of new intersection of base class libraries and does not have any control on the target platforms which were still evolving separately.
* There are many Verticals for .Net Platforms with similar base class libraries.

We need app/library is compatible with all the .Net technologies (ex. asp.net core, asp.net, xamarin) so you can reuse the same piece of code in more than one product without a need of recompiling it.

**Requirement**

With above problems below are the requirements laid down.

* 100% source and binary compatible surface area for:
  + Classic .NET Framework and Xamarin assemblies (mscorlib and friends)
  + Existing portable class libraries (both, mscorlib as well as System.Runtime)
* Binary compatibility refers to binding compat, but an understanding that behavioral compatibility may not be 100% possible.
* We’ll not necessarily bring all the types that .NET Framework has, but that we’ll bring types in their entirety without subsetting members.
* Focus on APIs that are available everywhere, but allow for platform specific extensions.

**Solution**

We can create NetStandard, and is something you should have in mind when you are developing a Nuget package. But let’s be honest, writing and publishing a Nuget package is not what most of us do on daily basis. So we need a framework, that you can use it in your app, if the app is .Net Standard 2.0 compatible.

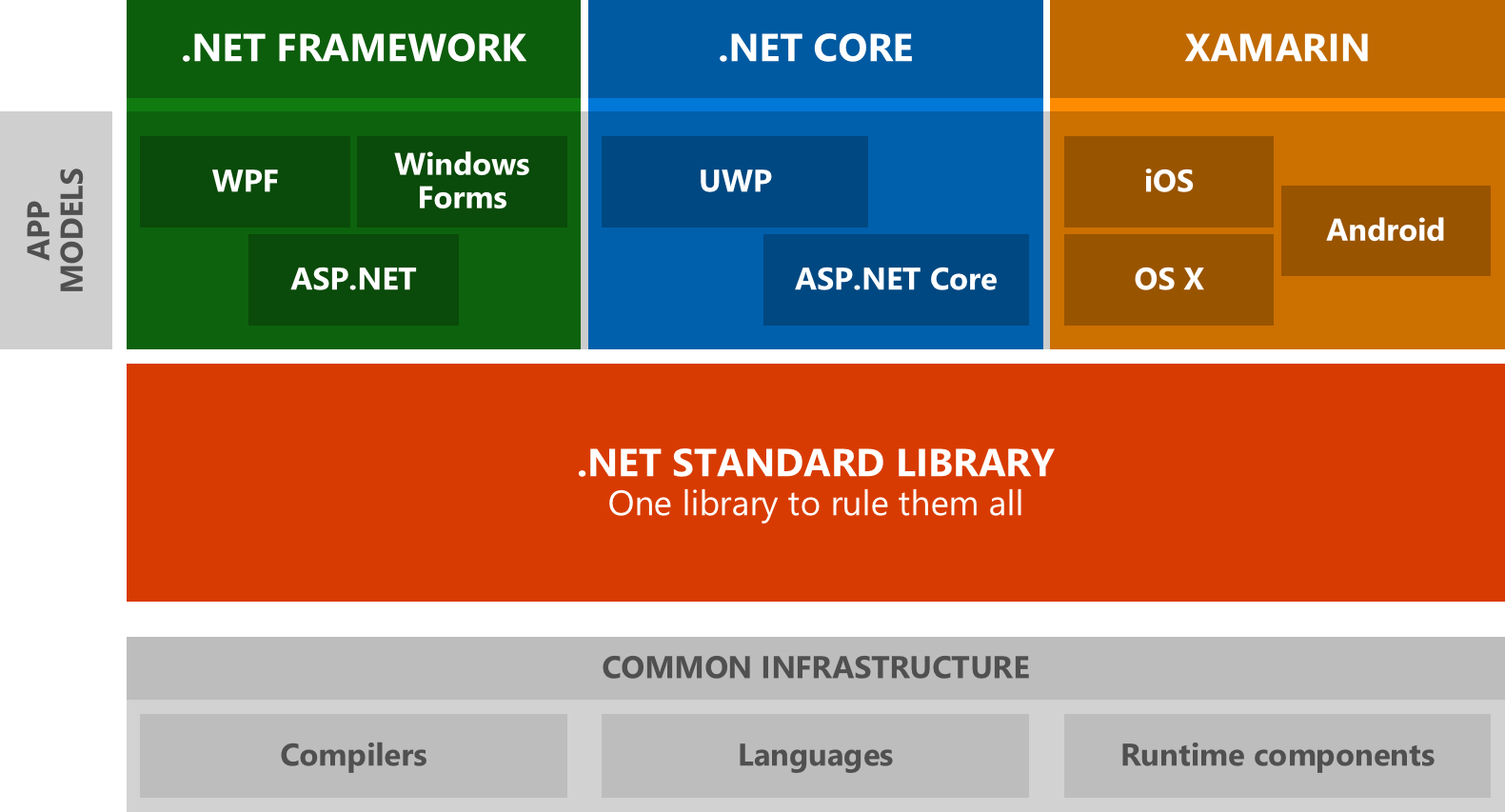
.Net Standard is not a framework or platform of its own. It does not have implementations or a runtime, just defines a specification what different .Net platforms has to implement to remain .Net Standard complaint. Eg.. .Net Core is a framework with runtime that implements .NetStandard.

**Features of .NetStandard**

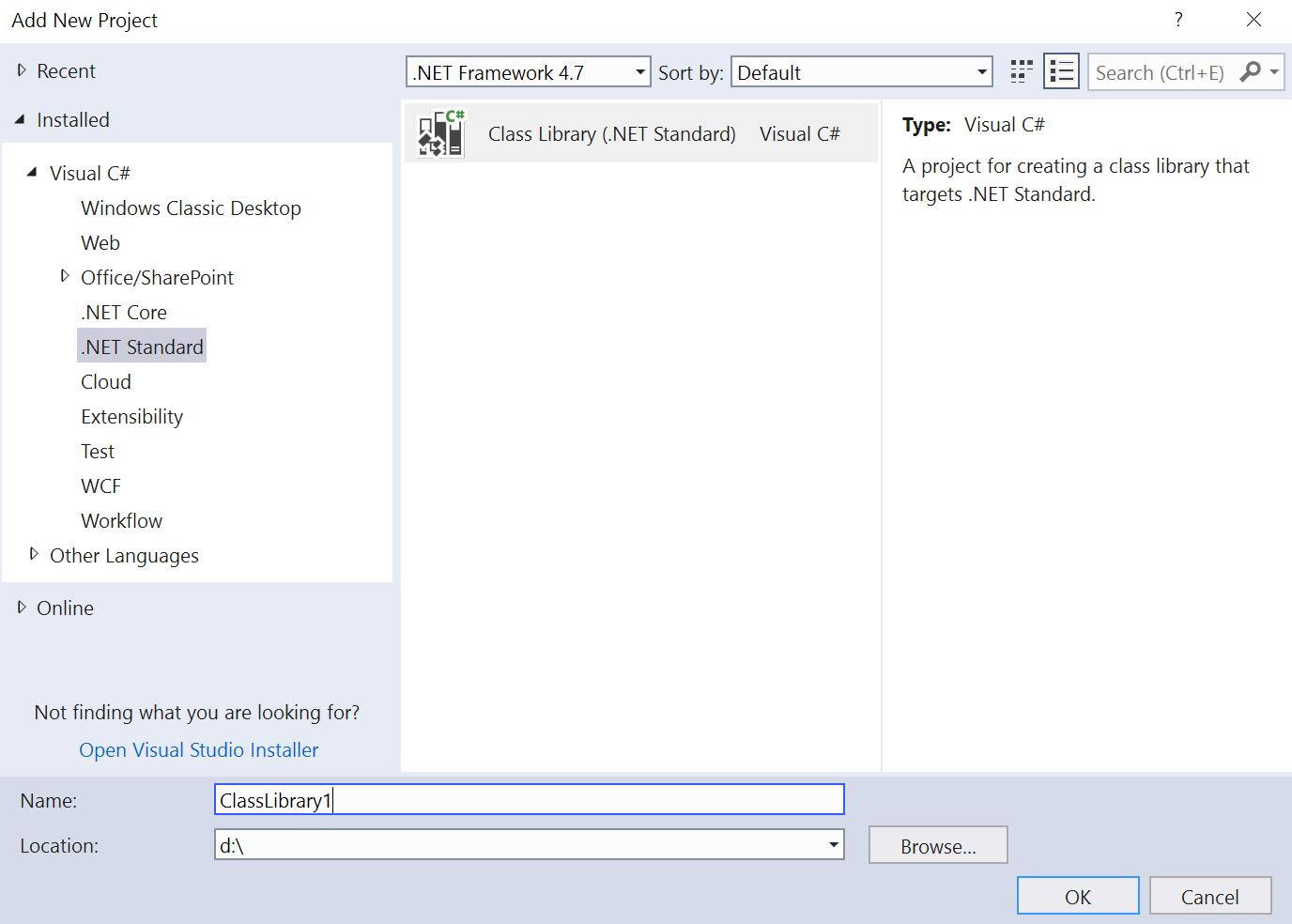
* **.NET Standard is for sharing code**: It replaces Portable Class Libraries (PCLs) as the tool for building .NET libraries that work everywhere.
* **Much bigger API Surface**: We have more than doubled the set of available APIs from 13k in [.NET Standard 1.6](https://github.com/dotnet/standard/blob/master/docs/versions/netstandard1.6.md) to 32k in [.NET Standard 2.0](https://github.com/dotnet/standard/blob/master/docs/versions/netstandard2.0.md). Most of them are existing .NET Framework APIs. These additions make it much easier to port existing code to .NET Standard,
* **.NET Framework compatibility mode**: Compatibility mode is added, that allows .NET Standard projects to reference .NET Framework libraries.

**Broad Platform Support**

* + .NET Framework 4.6.1
  + .NET Core 2.0
  + Mono 5.4
  + Xamarin.iOS 10.14
  + Xamarin.Mac 3.8
  + Xamarin.Android 7.5
  + UWP is work in progress and will ship later this year.

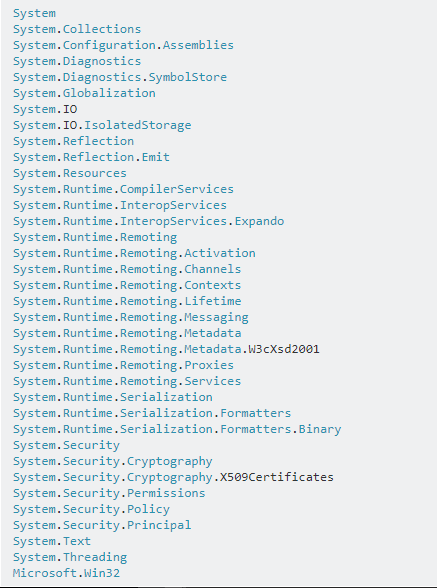
****

# **How do I use .NET Standard?**



**Note**

Microsoft's Common Object Runtime Library (**MSCorlib**) and it is the primary assembly for the Framework Common Library.It contains the following namespaces**:**

****

**How does .NET Standard work?**

*.NET Standard is represented by*

* The NuGet package NetStandard.Library which contains
* The reference assembly netstandard.dll

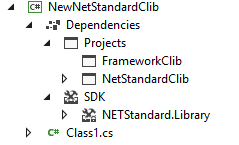
*At build time*

* .NET Standard bridges references to existing .NET Framework and PCL assemblies via type forwarding.

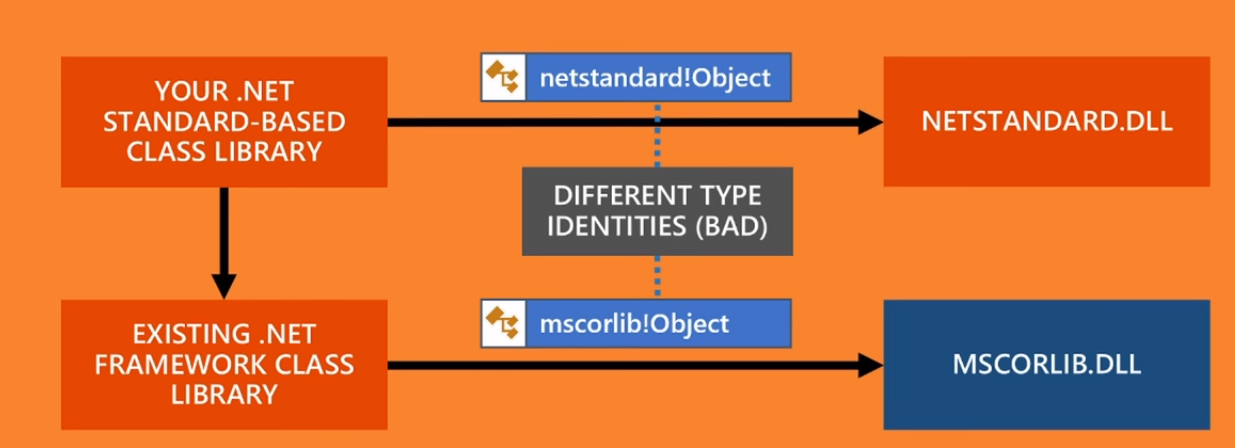
*At runtime*

* Each platform provides an implementation for netstandard.dll that type forwards to its implementation

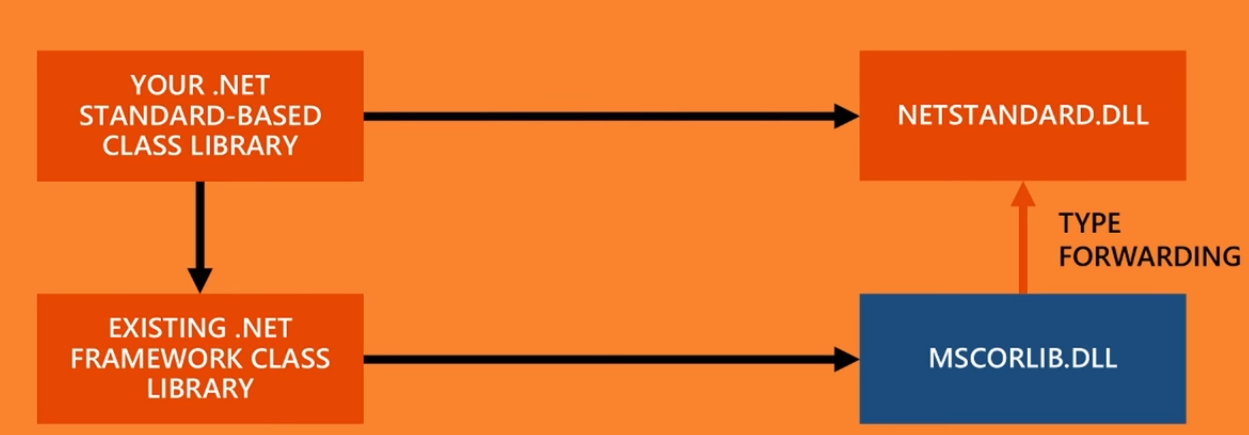
For example if we have NetStandard target application and is referencing NetStandard.dll and NetFramework compiled application as shown in below screenshot.



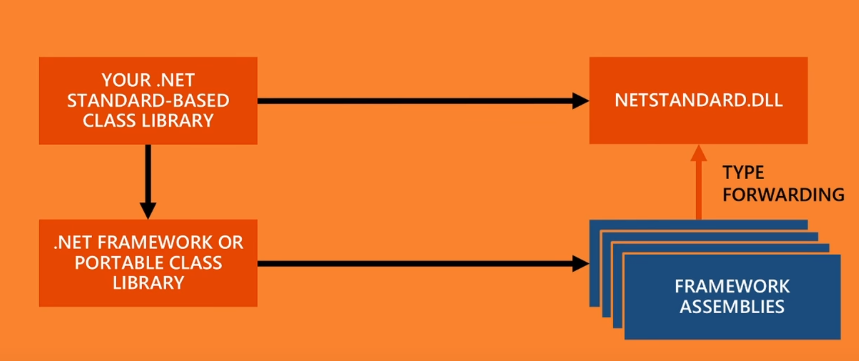
If we need to send object of type NetStandard to FrameWorkClass library, there will type compatibility issue. Netstandard!Object can not be passed to mscorlib!Object . NetFramework always reference MSCORLIB directly.



The above problem statement is resolved by type forwarding to its implementation.

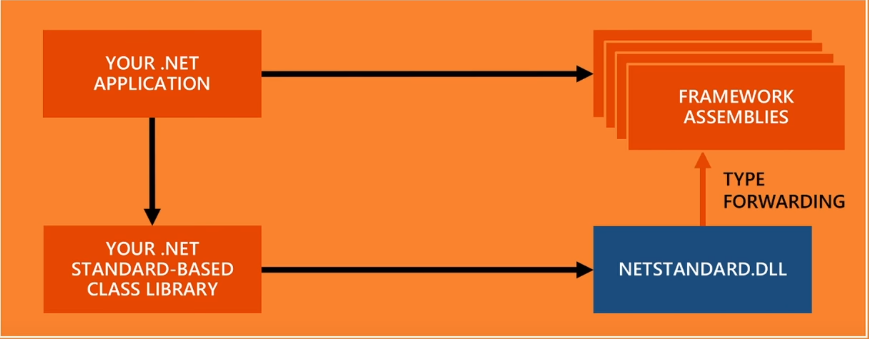


And this is done for all platform specific libraries.

****

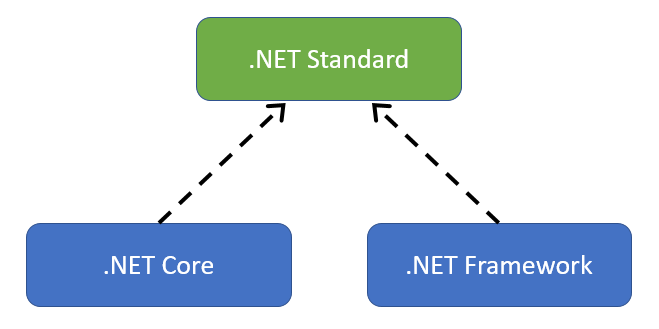
**Note**

1. Each .NetStandard project is shipped with mscorlib.dll facade which is only interface and when Net standard tries to use Net framework class objects, since target application is netstandard2.0 , Netframework references mscorlib facade, which inturn type forwards to Netstandard.dll. This makes compatibility between NetStandard based project and NetFramework.



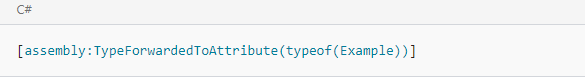
**Note**

1. Each Framework project is shipped with netstandard.dll facade which is only interface and when NetFramework tries to use NetStandard class objects, since target application is NetFramework4.6.2 , netstandard based project references netstandard.dll facade, which inturn type forwards to mscorlib.dll. This makes compatibility between NetFramework based project and NetStandard based project.



**What is Type Forwarding?**

Type forwarding is a **Common Language Runtime (CLR)** feature that allows us to move a type from its original assembly to another assembly in such a way that there is no need to recompile the applications referring to the original assembly.



Suppose you have created an assembly named StudentLibrary which contains a class named EnrollmentClass. Let's say, some of your applications refer to the StudentLibrary assembly and utilizes EnrollmentClass. In a later phase, you decide to move EnrollmentClass from StudentLibrary to a newly created assembly called EnrollmentAdvancedLibrary. If you ship a new version of StudentLibrary (which now doesn't have EnrollmentClass), along with EnrollmentAdvancedLibrary, then your existing applications looking for EnrollmentClass in StudentLibrary will not find the EnrollmentClass and end up with errors.

.NET Standard defines a set of reference assemblies for each version that are used to compile .NET Standard assemblies against.

For .NET Standard 2.0, the most notable reference assembly is netstandard.dll. If you reference System.Object, the compiler will emit IL code referencing [netstandard]System.Object.

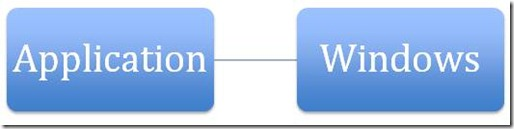
Any platform that "conforms" to .NET Standard 2.0 then has an implementation assembly netstandard.dll that either contains the type or contains type forwarding definitions. So for .NET Framework, there can be a netstandard.dll that contains a type forward to [mscorlib]System.Object. Another platform may have a different netstandard.dll that for instance forwards to [System.Runtime]System.Object.

In addition to netstandard.dll there are a few more libraries that are there to support .NET Standard 1.0-1.6 and a few other type forwarding dlls that form a compatibility shim for .NET Framework applications.

**What is Compat Shim?**

A Shim is a small library which transparently intercepts an API, changes the parameters passed, handles the operation itself, or redirects the operation elsewhere. Shims can also be used for running programs on different software platforms than they were developed for.

In this particular case, the two objects are the application program and Windows.



After the shim is applied, the application interacts with Windows indirectly; the shim code is injected and can modify the request to Windows, the response from Windows, or both.



Specifically, it leverages the nature of linking to redirect API calls from Windows to alternative code --- the Shim. Calls to external binary files take place through the Import Address Table (IAT) .

****

A compat shim makes .NetFramework libraries usable on other platforms and with NetStandard, with caveats.

**Platform specific APIs & .NET Standard**

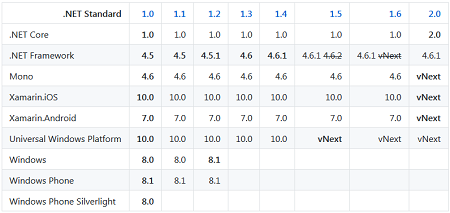
.NET Standard (mostly) only contains APIs that will work everywhere

* We generally avoid adding large chunks of APIs that don’t work everywhere
* A small set of APIs will throw PlatformNotSupportedException

Platform specific APIs sit on top of .NET Standard & you can add references to them. And these are less portable

*Examples*: Registry, Reflection Emit, Access Control, Windows Identity.

**.Net Standard Versions and Platform Support**



On the other hand, .NET Standard 2.0 adds many APIs that .NET Framework 4.6.1 already supports. The delta looks as follows:

* .NET Standard 2.0 adds 14,994 APIs that .NET Framework 4.6.1 already supports
* .NET Standard 2.0 only has 43 APIs that .NET Framework 4.6.1 doesn't support
* .NET Standard 2.0 will be a strict superset of .NET Standard 1.6. In other words, no breaking changes will happen between .NET Standard 2.0 and 1.x.
* .NET Framework 4.6.1 will allow referencing binaries that are compiled against .NET Standard 2.0.

Considering the number of APIs that .NET Framework 4.6.1 will not support is low and that these are all brand-new APIs with low adoption we believe this is a much better trade-off.

You can use API Port to scan a given application to make sure no code in your application depends on these APIs.

# **.NET Portability Analyzer**

To work out if you can upgrade from .NET Framework to .NET Core and to find what you need to change, use the **.NET Portability Analyzer** extension for Visual Studio.

This is a fantastic tool that scans your projects and checks all the APIs being used against the configured framework targets. It can identify which APIs are being used that are incompatible with a given target and even suggests newer alternative APIs in some cases. In my experience, the suggestions are quite rare, so you'll be on your own in most cases.

Once you have installed the .NET Portability Analyzer. Go to the Analyze menu, and select Portability Analyzer Settings to configure the options. You can set the output directory, select the output formats (Json, HTML, Excel), and the target platforms.

<https://docs.microsoft.com/en-us/dotnet/core/porting/>

## **.NET Core 2.0**

.NET Core is newer and features many improvements over .NET Framework. Microsoft released .NET Core 1.0 in June 2016, a little over a year ago, and just released .NET Core 2.0.

If you started a new project in the last 12 months, particularly an ASP.NET Core project, you may be using .NET Core.

It has these features:

* Cross-platform (the OS kind of platform)
  + It runs on Windows, Linux, and macOS
* Installation is standalone and side-by-side
  + You can run different versions of .NET Core for different applications on the same computer
* High Performance
  + Improvements over .NET Framework in Collections, Math, Regex, Strings, Files, and more.
* Open Source
  + Part of the .NET Foundation and generally licensed under either an MIT or Apache 2 license
* v2.0 implements .NET Standard 2.0

How is .NET Standard different from .NET Core?

* .NET Standard is a specification that covers which APIs a .NET platform has to implement.
* .NET Core is a concrete .NET platform and implements the .NET Standard.

## **Compatibility Mode**

.NET Standard 2.0 also adds a new compatibility mode that allows projects to target .NET Framework libraries, provided the libraries only use APIs in .NET Standard 2.0. This enables usage of many older .NET Framework libraries that were previously unusable in .NET Core projects and .NET Standard libraries.

Approximately 70% of NuGet packages targeting .NET Framework can be targeted under compatibility mode. Previously, any projects or libraries that had dependencies of these packages could not be upgraded.

**Inclusive Principles**

* **Ubiqutous APIs.** APIs in the .NET Standard must be implemented by all .NET platforms. Thus, we're only interested in standardizing APIs that are universal in nature and thus should be available everywhere.
* **Mature APIs.** APIs that are part of the .NET Standard can only be versioned when the standard itself is versioned. Thus, we generally only standardize APIs that are mature enough that they don't have to be updated frequently.
* **Runtime-specific APIs.** If the APIs have to be implemented by the runtime, they don't benefit from being libraries on top of .NET Standard. Good examples of such APIs include primitive types, reflection, GC, and code-gen intrinsics (SIMD).
* **Widely-used APIs.** In order to enable a vibrant .NET ecosystem, it's important to have a common vocabulary of types that library authors can rely on. Thus, it's beneficial to add widely used APIs to the .NET Standard as it simplifies building reusable libraries.

**Few libraries that are not supported in .NetStandard2.0**

* The AppDomain type is part of .NET Standard
* The primary reason we expose this type in .NET Standard is because the usage is fairly high and typically not associated with creating new app domains but for interacting with the current app domain, such as registering an unhandled exception handler or asking for the application's base directory.
* No Remoting dll's are part of .Net Standard but MarshalByRefObject
* NET Standard will contain the abstractions (DbProvider, DbProviderFactories, DbConnection, IDbConnection etc.) as well as the general ADO.NET facilities (DataSet, DataTable etc.) APIs. Providers are not part of .Net Standard as they remain platform specific.
* JSON.NET not part of .NET Standard
* "Type" for example, methods, properties, etc is not added to .net Standard. It remains platform specific.
* Unity is not supported currently.
* Extensions exist outside of the .NET Standard and must be manually referenced.
* .NET Standard you are only getting what is in the BCL, not what we know as the Framework Class Library (FCL). Frameworks in this case are application platforms like ASP.NET (System.Web and OWin based), WinForms, WPF, WCF etc. For .NET Core ASP.NET 5 is an Application Framework that lives on top of .NET Standard.

**Converting .NetFramework or .NetCore Classlibrary to .NetStandard**

Replace csproj content with below code.

<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>

<TargetFramework>netstandard2.0</TargetFramework>

</PropertyGroup>

</Project>

**Xamarin with NetStandard**

<Project Sdk="Microsoft.NET.Sdk">  
 <PropertyGroup>  
 <TargetFramework>netstandard1.4</TargetFramework> <PackageTargetFallback>$(PackageTargetFallback);portable-win+net45+wp8+win81+wpa8</PackageTargetFallback>  
 </PropertyGroup>  
 Other stuff here  
 </Project>

Highlighted code tells VS that it will be accepting/compatible of these other frameworks.

**EF6 in NetCore2.0 application**

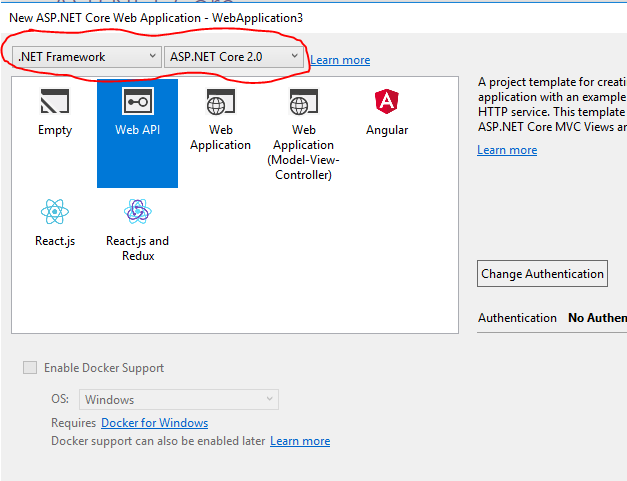
EF6 basically runs on .NetFramework4.6.2 or in .NetStandard but i guess there are issues . We can use both in .netstandard project but best recommended to use latest .NetCore EF.

EF6.0 is updated to EF6.1.3 recently to support in .netstandard projects. We can install through nuget package as "EntityFrameWork".

EFCore cannot run in .NetFramework4.6.2. We can install through nuget package as "Microsoft.Entity.FrameworkCore.EFCore". This is for .NetCore or .netStandard based. For more details refer to below link.

<https://docs.microsoft.com/en-us/ef/efcore-and-ef6/choosing>

<https://github.com/aspnet/EntityFramework6/issues/271>



**Encryption and Decryption in .NetStandard2.0**

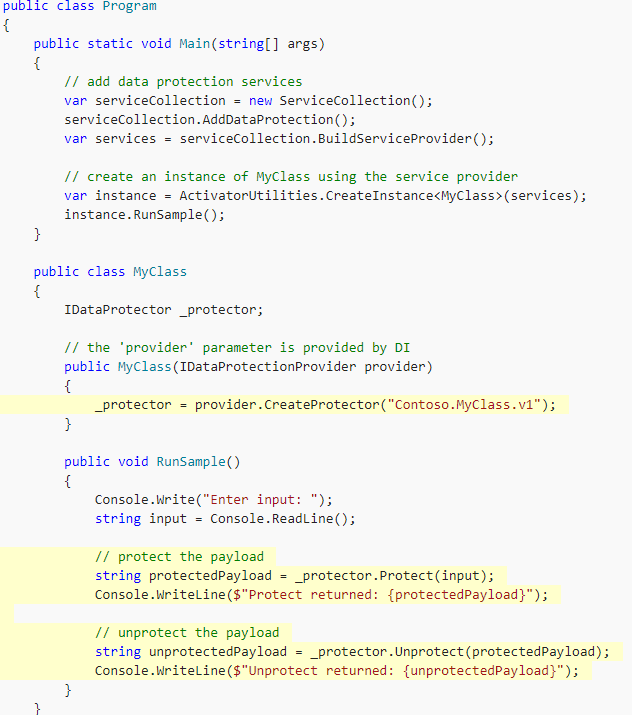
There's a replacement in NetStandard Core for the machine keys.

The ASP.NET Core data protection stack is designed to serve as the long-term replacement for the security elements in .NET Framework web application. It was designed to address many drawbacks of the old cryptographic and providing better solution for VS2017 applications.

*Links*:

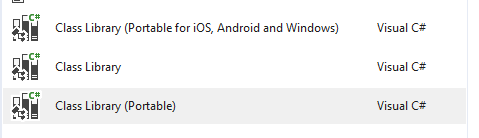
<https://docs.microsoft.com/en-us/aspnet/core/security/data-protection/>

<https://docs.microsoft.com/en-us/aspnet/core/security/data-protection/using-data-protection>



# **In VS2015 how to reference NetStandard?**

* In Visual Studio 2015 .net core is executable project type and targeting .netstandard for an executable project type is not supported.
* **WinForms**: we can reference ClassLibrary (Portable) with target as .netstandard 1.4 which targets Framework 4.6.2
* **Console**: we can reference ClassLibrary (Portable) with target as .netstandard 1.4 which targets Framework 4.6.2
* **WPF**: we can reference ClassLibrary (Portable) with target as .netstandard 1.4 which targets Framework 4.6.2



**Note**:

1. If we add NetStandard2.0 in VS2015, we cannot target class library to Net standard2.0 version nor we can use assemblies in NetStandard.dll.
2. If we add Netstandard.dll to .NetFramework in VS2015 or VS2017, this does not give us any advantage of netstandard.dll because netstandard.dll references all the assemblies which NetFramework4.6.2 has.

# **References:**

<https://github.com/dotnet/corefx/issues/18217>

<https://docs.microsoft.com/en-us/aspnet/core/>

<http://www.michael-whelan.net/porting-dotnet-framework-library-to-dotnet-core/>

<https://www.stevejgordon.co.uk/migrating-full-net-framework-net-core>

<http://www.natemcmaster.com/blog/2017/03/09/vs2015-to-vs2017-upgrade/>

<https://docs.microsoft.com/en-us/aspnet/core/migration/1x-to-2x/identity-2x#identitycookieoptions-instances>

<https://docs.microsoft.com/en-us/ef/core/extensions/entityframeworkcore-detached>

<https://docs.microsoft.com/en-us/aspnet/core/security/authentication/>

<https://www.danielcrabtree.com/blog/314/upgrading-to-net-core-and-net-standard-made-easy>

<https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity-configuration?tabs=aspnetcore1x>

<https://github.com/dotnet/cli-migrate>

<https://github.com/dotnet/cli/issues/4770>

<https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-migrate>

<https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/migrations>

<https://github.com/dotnet/cli/issues/2245>

<https://github.com/dotnet/cli/issues/5444>

<https://github.com/dotnet/standard/issues/375>

<https://www.nuget.org/packages/NETStandard.Library.NETFramework/2.0.0-preview1-25305-02>

<https://github.com/dotnet/sdk/issues/1267>

<https://github.com/dotnet/cli/issues/6738>

<https://github.com/dotnet/cli/issues/6737>

<https://github.com/dotnet/core/issues/686>

<https://www.pluralsight.com/courses/entity-framework-core-getting-started?aid=7010a000002BWq6AAG&promo=&oid=&utm_source=google&utm_medium=ppc&utm_campaign=APAC_Dynamic&utm_content=&utm_term=&gclid=EAIaIQobChMIg6TKkbmJ1wIVA5GPCh0XMQ-9EAAYASAAEgK4dvD_BwE>

<https://docs.microsoft.com/en-us/aspnet/core/data/entity-framework-6>

<https://github.com/dotnet/cli/pull/6818>

<https://github.com/dotnet/cli/pull/7019>

<https://github.com/dotnet/roslyn/issues/12918>

<https://github.com/dotnet/core>

<https://github.com/aspnet/Mvc/blob/dev/Mvc.sln>

<https://github.com/aspnet/Home>

<https://github.com/aspnet/DotNetTools>

<https://www.danielcrabtree.com/blog/314/upgrading-to-net-core-and-net-standard-made-easy>

<https://github.com/dotnet/standard/blob/master/docs/versions.md>

<https://docs.microsoft.com/en-us/dotnet/core/deploying/deploy-with-cli>

<https://docs.microsoft.com/en-us/dotnet/core/rid-catalog>

<https://github.com/dotnet/cli/issues/6237>

<https://github.com/dotnet/cli/issues/6760>

<https://github.com/dotnet/cli/issues/6237>

<https://github.com/dotnet/coreclr/blob/master/Documentation/workflow/UsingYourBuild.md#change-project-to-be-self-contained>

<https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-publish?tabs=netcore2x>

<https://github.com/dotnet/cli-migrate>

<https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/migrations>

<https://github.com/aspnet/home/issues/2022>

<https://blogs.msdn.microsoft.com/dotnet/2017/08/14/announcing-net-standard-2-0/>

<https://docs.microsoft.com/en-us/dotnet/core/tools/?tabs=netcore2x>

<https://weblog.west-wind.com/posts/2017/Sep/14/Accepting-Raw-Request-Body-Content-in-ASPNET-Core-API-Controllers>

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/request-features>

<http://www.c-sharpcorner.com/article/what-is-net-standard/>

<http://www.codedigest.com/quick-start/9/what-is-netstandard>

<https://github.com/dotnet/standard/blob/master/docs/versions/netstandard2.0.md>

<https://elanderson.net/2016/10/create-a-net-standard-library-for-use-with-full-net-framework/>

<https://github.com/dotnet/corefx/issues/18217>

<https://github.com/dotnet/standard/blob/master/netstandard/src/ApiCompatBaseline.net461.txt>

<https://docs.microsoft.com/en-us/dotnet/core/tools/global-json>

<https://www.nuget.org/packages/NETStandard.Library/>

<https://docs.microsoft.com/en-us/dotnet/standard/net-standard>

<https://docs.microsoft.com/en-us/dotnet/project-json>

<https://github.com/IdentityServer/IdentityServer4/issues/926>

<https://stackoverflow.com/questions/41524744/structuring-an-asp-net-core-project-to-reference-net-standard-libraries>

<https://stackoverflow.com/questions/44464937/compatibility-shim-used-by-net-standard-2-0>

<https://github.com/dotnet/standard/tree/master/docs/netstandard-20>

<https://github.com/dotnet/standard/issues/497>

<https://blogs.infosupport.com/net-core-and-net-standard-2-0/>

<https://github.com/dotnet/cli/issues/1657>

<https://github.com/NuGet/NuGet.Client/pull/418>

<https://github.com/dotnet/cli/issues/3512>

<https://channel9.msdn.com/Blogs/MVP-VisualStudio-Dev/EF-Core-Quick-Starts-ASPNET-Core-in-Visual-Studio-2017>

<https://github.com/dotnet/core/blob/master/Documentation/core-repos.md>

<https://github.com/dotnet/corefx>

<https://docs.microsoft.com/en-us/dotnet/api/>

<https://andrewlock.net/understanding-net-core-netstandard-and-asp-net-core/>

<https://github.com/dotnet/corefx/blob/master/Documentation/architecture/net-platform-standard.md>

<https://github.com/dotnet/standard/blob/master/docs/faq.md>

<https://github.com/dotnet/standard/blob/master/docs/versions.md>

<https://github.com/dotnet/standard/blob/master/docs/review-board/README.md>

<https://github.com/dotnet/standard/blob/master/docs/metaphor.md>

<https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming>

<https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/data-parallelism-task-parallel-library>

<https://docs.microsoft.com/en-us/dotnet/standard/threading/the-managed-thread-pool>

<https://www.codeproject.com/Articles/747594/Understanding-Task-Parallelism-Framework-TPL-using>

<https://www.codeproject.com/Articles/67999/Forwarding-a-type-from-one-assembly-to-another-Typ>

<https://www.codeproject.com/Articles/27268/Type-Forwarding-in-NET>

<https://blogs.technet.microsoft.com/askperf/2011/06/17/demystifying-shims-or-using-the-app-compat-toolkit-to-make-your-old-stuff-work-with-your-new-stuff/>