Rockford Lhotka

2019年1月11日

« CSLA .NET Version 4.9 New Features | Main | Remaining Relevant as a .NET Developer »

Migrating from .NET to .NET Standard

During 2018 I gave a talk at some VS Live events discussing how one might migrate existing .NET Framework enterprise apps/code to .NET Core. In this talk I have some assumptions I think are reasonable:

- Most of us can't do a "big bang" rewrite of our apps/code all in one shot
  - It'll take months or years to migrate from .NET to .NET Core
  - During this time it is necessary to maintain the existing code while working on the new code
- A lot of existing code is still on .NET 2, 3, and 4
- We're talking Windows Forms, WPF, and ASP.NET code - lots of variety
  - In *most* cases business logic is embedded in the UI - code-behind forms/pages or in controllers
  - Editorial observation: More people *should* be using CSLA to gain separation of concerns: keep their business logic in a separate and reusable layer from the UI or data access 😃
- In most cases you are not just migrating from .NET Framework to .NET Core, but also modernizing/rewriting the UI to also be modern
  - Replacing Windows Forms and WPF with ASP.NET Core Razor Pages or MVC, or Xamarin Forms
  - Maybe *upgrading* Windows Forms or WPF to the new .NET Core 3.0 support once it is available

Several people have asked if I'd blog the gist of my presentation, so here it is.

In summary:

- Step 0: Understand .NET Core vs .NET Standard
- Step 1: Get to .NET 4.6.1 or Higher
- Step 2: Separation of Concerns
- Step 3: Move Business Code to Shared Library
- Step 4: Create .NET Standard Project
- Step 5: Mitigate Dependency Conflicts
- Step 6: Mitigate Code Conflicts
- Step 7: Have a Glass of Bourbon

The code used in my talk and this post is the Net2NetStandard solution on GitHub.

Step 0: Understand .NET Core vs .NET Standard

I've encountered a lot of confusion between .NET Core and .NET Standard and .NET Framework. It is important to have a good understanding of these terms before moving forward at all, so you end up in the right place.

- .NET Framework is the "legacy" .NET implementation we've been using since 2002, and the long-term goal is to move off .NET Framework onto something more modern
- .NET Core is a new implementation of .NET that currently supports two types of UI: console and web server. .NET Core 3 is slated to also support Windows Forms and WPF UI frameworks. It does not currently support Xamarin (iOS, Android, Mac, Linux), or WebAssembly (mono-wasm/Blazor).
- .NET Standard is an *interface* against which you can write code, and that interface is *implemented* by .NET Framework 4.6.1+ and by .NET Core 2+ and by Xamarin (and by mono and mono-wasm). If you write your code against .NET Standard, then your compiled DLL can be deployed to .NET Framework, .NET Core, Xamarin, and other .NET implementations.

As a result, my recommendation is that you should always get as much of your code into .NET Standard as possible, because the resulting compiled DLL can run essentially anywhere.

Home | Lhotka.net | CSLA .NET

On this page....

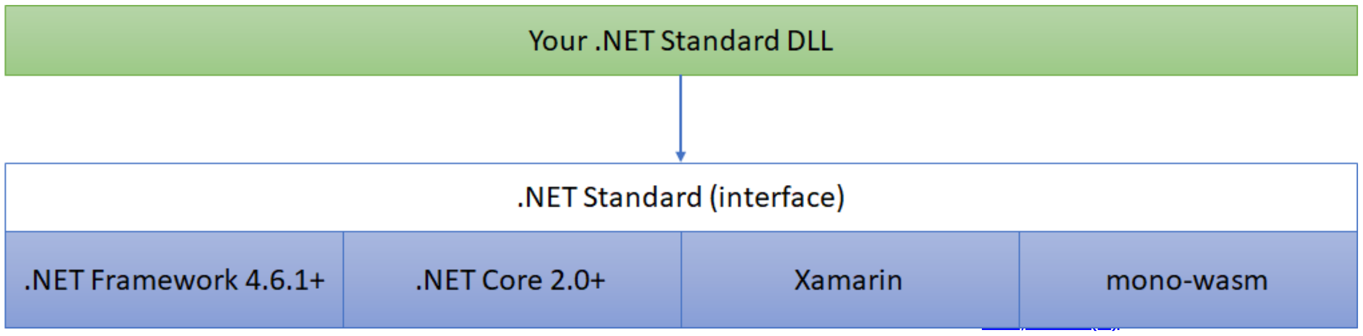
Migrating from .NET to .NET Standard

Search

Search

Archives

六月, 2020 (1)
九月, 2019 (3)
八月, 2019 (2)
六月, 2019 (1)
四月, 2019 (2)
一月, 2019 (1)
十二月, 2018 (1)
十一月, 2018 (1)
十月, 2018 (1)
九月, 2018 (3)
八月, 2018 (3)
六月, 2018 (4)
五月, 2018 (1)
四月, 2018 (3)
三月, 2018 (4)
十二月, 2017 (1)
十一月, 2017 (2)
十月, 2017 (1)
九月, 2017 (3)
八月, 2017 (1)
七月, 2017 (1)
六月, 2017 (1)
五月, 2017 (1)
四月, 2017 (2)
三月, 2017 (1)
二月, 2017 (2)
一月, 2017 (2)
十二月, 2016 (5)
十一月, 2016 (2)
八月, 2016 (4)
七月, 2016 (2)
六月, 2016 (4)
五月, 2016 (3)
四月, 2016 (4)
三月, 2016 (1)
二月, 2016 (7)
一月, 2016 (4)
十二月, 2015 (4)
十一月, 2015 (2)
十月, 2015 (2)
九月, 2015 (3)
八月, 2015 (3)
七月, 2015 (2)
六月, 2015 (2)
五月, 2015 (1)
二月, 2015 (1)
一月, 2015 (1)
十月, 2014 (1)
八月, 2014 (2)



If all you do is get your code to .NET Core, that currently blocks you from reusing that code on .NET Framework, Xamarin, WebAssembly, and other .NET implementations.

All *that* said, it is important to understand that your *UI* code will almost certainly be .NET platform specific. In other words, you'll choose to write a console app, a web site, a mobile app, or a desktop app *in a specific implementation of .NET*. So your UI is not portable or reusable to the same degree as non-UI code.

Your non-UI code should always be built with .NET Standard so it is as portable as possible, enabling reuse of that code in current and future .NET implementations and UI technologies.

This is why my talk (and this post) are about how to get to .NET Standard, not .NET Core. .NET Standard gets you to .NET Core plus Xamarin and other platforms.

Step 1: Get to .NET 4.6.1 or Higher

Version 4.6.1 of the .NET Framework is special, because this is the earliest version that is compatible with .NET Standard 2.0. In reality you'll probably want to get to 4.7.1 or whatever version exists when you start this journey, but the minimum bar is 4.6.1.

Basically, if your existing code won't run on .NET 4.6.1, you'll need to take whatever steps are necessary to get from your older unsupported version (2? 3? 3.5? 4.0? 4.5?) to 4.6.1 or higher.

Fortunately this is *usually* not that difficult, because Microsoft has done a good job of minimizing breaking changes and preserving backward compatibility over time.

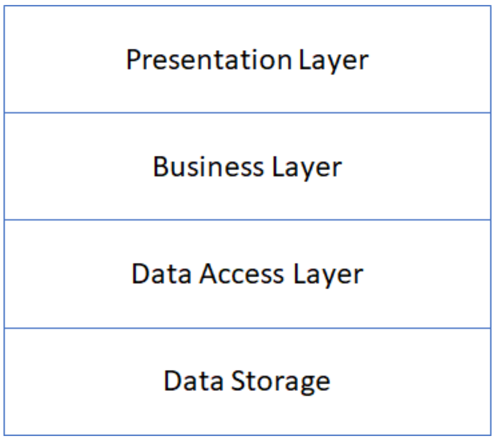
Step 2: Separation of Concerns

This is almost certainly the hardest step: if your existing code is "typical" it probably has tons of non-UI logic in button click or lostfocus event handlers, postback handlers, or controller methods. People have "enjoyed" this style of coding since VB3 back in the early 1990's and it persists through today.

The problem is that moving the *UI* to .NET Standard is a whole different thing from moving *business logic* or even *data access logic* to .NET Standard. Yes, .NET Core 3.0 is planned to have Windows Forms and WPF support, so that should help. But I suspect for most people the migration from .NET Framework to .NET Core ultimately means rewriting the UI into something more modern.

As a result, any code embedded in the UI or presentation layer needs to be cleaned up. You need to apply the concept of separation of concerns and get non-UI code out of the UI. That means no business or data access logic in code-behind or controllers or viewmodels. The goal should be (in my view) that all business logic (validation, calculations, manipulation, rules, authorization) is in a separate business layer, and all data access logic is in *its* own layer.

七月, 2013 (2)
五月, 2013 (3)
四月, 2013 (2)
三月, 2013 (3)
二月, 2013 (7)
一月, 2013 (4)
十二月, 2012 (3)
十一月, 2012 (3)
十月, 2012 (7)
九月, 2012 (1)
八月, 2012 (4)
七月, 2012 (3)
六月, 2012 (5)
五月, 2012 (4)
四月, 2012 (6)
三月, 2012 (10)
二月, 2012 (2)
一月, 2012 (2)
十二月, 2011 (4)
十一月, 2011 (6)
十月, 2011 (14)
九月, 2011 (5)
八月, 2011 (3)
六月, 2011 (2)
五月, 2011 (1)
四月, 2011 (3)
三月, 2011 (6)
二月, 2011 (3)
一月, 2011 (6)
十二月, 2010 (3)
十一月, 2010 (8)
十月, 2010 (6)
九月, 2010 (6)
八月, 2010 (7)
七月, 2010 (8)
六月, 2010 (6)
五月, 2010 (8)
四月, 2010 (13)
三月, 2010 (7)
二月, 2010 (5)
一月, 2010 (9)
十二月, 2009 (6)
十一月, 2009 (8)
十月, 2009 (11)
九月, 2009 (5)
八月, 2009 (5)
七月, 2009 (10)
六月, 2009 (5)
五月, 2009 (7)
四月, 2009 (7)
三月, 2009 (11)
二月, 2009 (6)



In short, you'll have a much easier time migrating code outside the UI to .NET Standard than any code *inside* the UI.

Step 3: Move Business Code to Shared Library

Now we get to the fun part. This step is in some ways the simplest and yet the most scary.

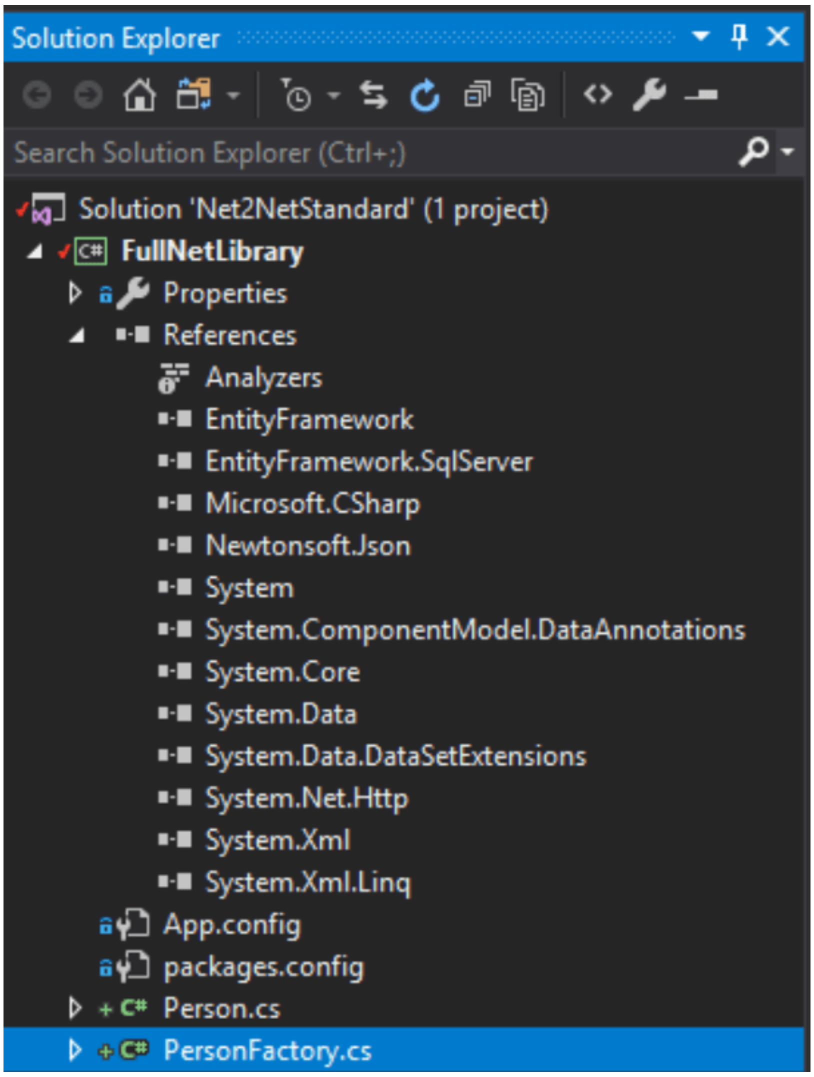
Right now your code is in a .NET Framework Class Library project. That means it compiles specifically for the .NET Framework, and uses .NET Framework specific dependency references. And this is your *existing, running code*, so we want to minimize risk in changing it, because changes to this code and existing references and even the `csproj` file will have a direct impact on your production environment.

The Net2NetStandard solution is intentionally stripped down to the bare minimum. My talk is often a 20 minute lightning talk, so the demo needs to be concise, and this qualifies. The start point is a .NET Framework Class Library project with some existing production code. That code uses Newtonsoft.Json and Entity Framework, with NuGet references to both dependencies.

一月, 2009 (9)
十二月, 2008 (5)
十一月, 2008 (4)
十月, 2008 (7)
九月, 2008 (8)
八月, 2008 (11)
七月, 2008 (11)
六月, 2008 (10)
五月, 2008 (6)
四月, 2008 (8)
三月, 2008 (9)
二月, 2008 (6)
一月, 2008 (6)
十二月, 2007 (6)
十一月, 2007 (9)
十月, 2007 (7)
九月, 2007 (5)
八月, 2007 (8)
七月, 2007 (6)
六月, 2007 (8)
五月, 2007 (7)
四月, 2007 (9)
三月, 2007 (8)
二月, 2007 (5)
一月, 2007 (9)
十二月, 2006 (4)
十一月, 2006 (3)
十月, 2006 (4)
九月, 2006 (9)
八月, 2006 (4)
七月, 2006 (9)
六月, 2006 (4)
五月, 2006 (10)
四月, 2006 (4)
三月, 2006 (11)
二月, 2006 (3)
一月, 2006 (13)
十二月, 2005 (6)
十一月, 2005 (7)
十月, 2005 (4)
九月, 2005 (9)
八月, 2005 (6)
七月, 2005 (7)
六月, 2005 (5)
五月, 2005 (4)
四月, 2005 (7)
三月, 2005 (16)
二月, 2005 (17)
一月, 2005 (17)
十二月, 2004 (13)
十一月, 2004 (7)
十月, 2004 (14)
九月, 2004 (11)
八月, 2004 (7)
七月, 2004 (3)
六月, 2004 (6)
五月, 2004 (3)
四月, 2004 (2)
三月, 2004 (1)
二月, 2004 (5)

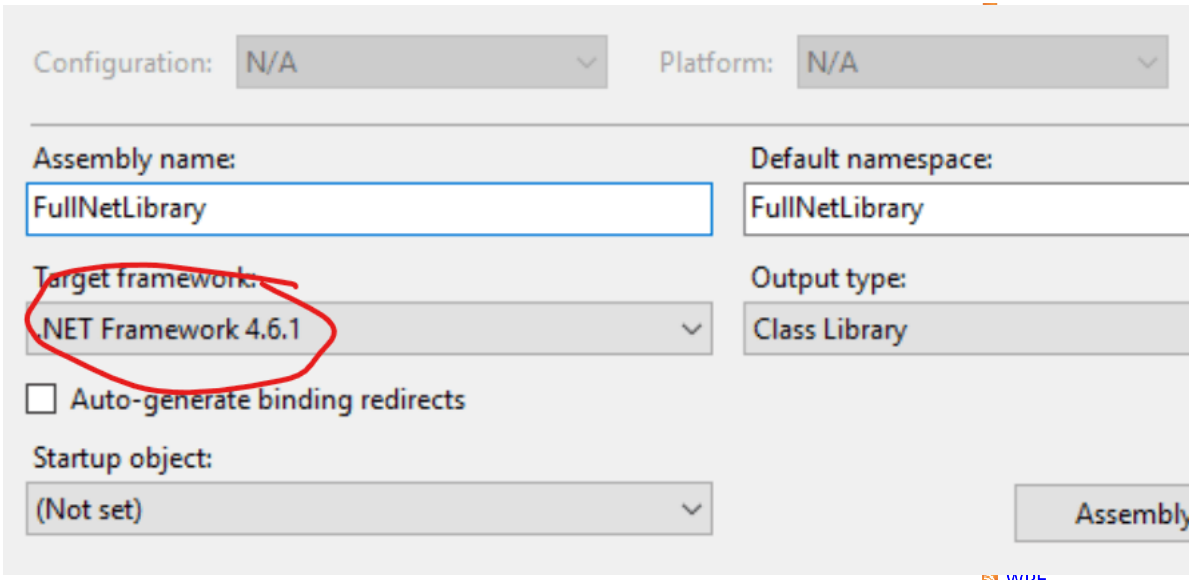
Categories

AspNetCore



```
Solution 'Net2NetStandard' (1 project)
  FullNetLibrary
    Properties
    References
      Analyzers
      EntityFramework
      EntityFramework.SqlServer
      Microsoft.CSharp
      Newtonsoft.Json
      System
      System.ComponentModel.DataAnnotations
      System.Core
      System.Data
      System.Data.DataSetExtensions
      System.Net.Http
      System.Xml
      System.Xml.Linq
    App.config
    packages.config
    Person.cs
    PersonFactory.cs
```

Importantly, this project is already targeting .NET Framework 4.6.1.


Configuration: N/A Platform: N/A

Assembly name: FullNetLibrary
Default namespace: FullNetLibrary
Target framework: .NET Framework 4.6.1
Output type: Class Library
Auto-generate binding redirects ☐
Startup object: (Not set)

What we want to do is get the code from this project into a location where it can continue to be used to build the existing .NET Framework DLL *and also* build a .NET Standard DLL. And we want to do this without duplicating the code or files, as that would make maintainability much harder.

Fortunately Visual Studio includes a feature called *Shared Projects* that solves this issue. A Shared Project is not a normal project at all, it is nothing more than a location to store code files. Those code files are then pulled into a *real* project at compile time *as though they were part of that real project*.

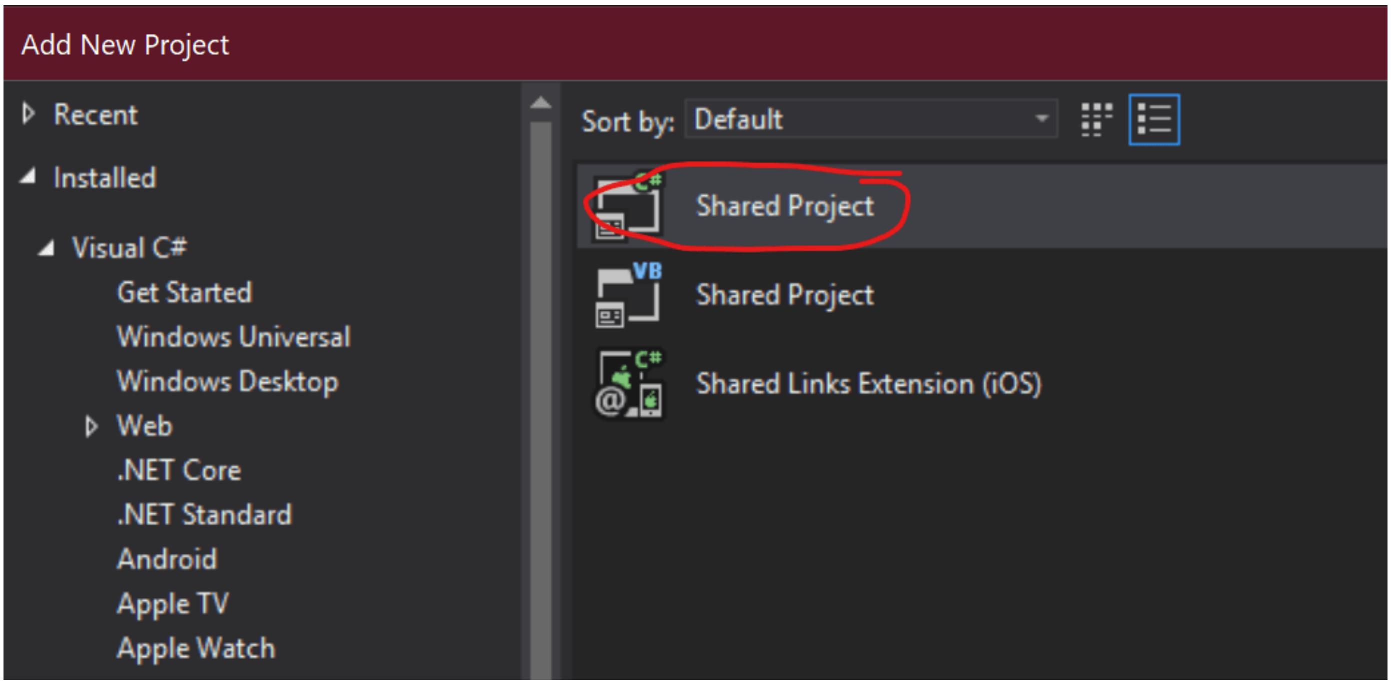
To see this in action, add a new C# Shared Project to the solution.

Kubernetes
.NET Core
.NET Standard
Android
Architecture
ASP.NET MVC
AspNetCore
Blazor
Books
Bxf
CSLA .NET
dasBlog
Distributed OO
Docker
git
Git
gRPC
h5js
Hololens
iOS
JavaScript
Kubernetes
Magenic
Microservices
Microsoft .NET
mono
MonoDroid
MonoTouch
MOslo
News
Ooui
Programming
RabbitMQ
Service-Oriented
WPF
Xamarin
Xbox
Zune

About

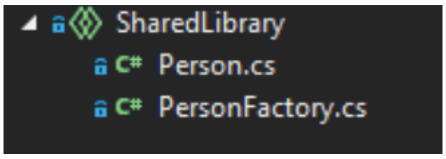
Powered by: newtelligence dasBlog 2.0.7226.0

Disclaimer
The opinions expressed herein are my own personal opinions and do



What you'll see in Solution Explorer is that this new project is missing common things like a References or Dependencies node, or a Properties folder. Again, this is not a normal project, it is nothing more than a placeholder to contain code files.

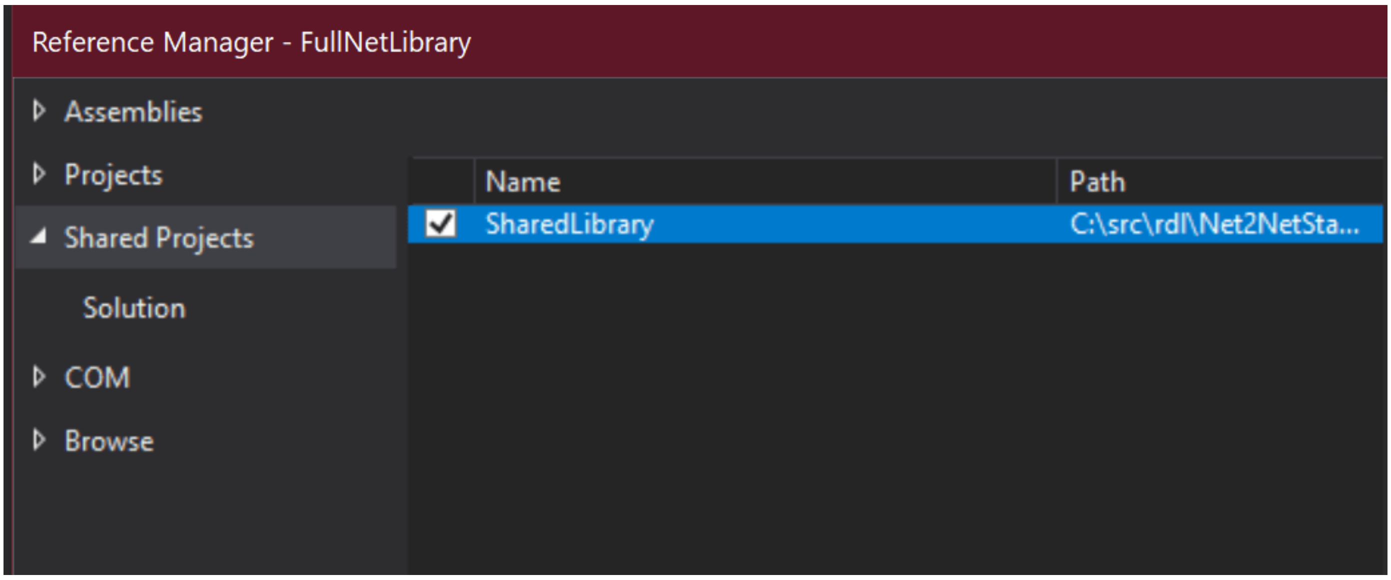
Next, select the source files from the .NET Framework project and drag-drop them into the new SharedLibrary project. That'll copy the files, so there's no risk here.



> Before proceeding with any real code, now is the time to make sure you've done a commit to source control so you have an easy way to revert in case something does go wrong!

However, this next step might make your heart race and palms sweat a little, because I want you to highlight *and delete* the source files from the original .NET Framework project. I know, this sounds scary, but trust me (and your backups).

And here's the key: go to the .NET Framework project and add a reference to the SharedProject.



At this point you can build the original .NET Framework project and you'll get *the exact same DLL output as before*. Zero changes to your existing code or build result. And yet your code is now in a *physical* location that'll enable forward movement.

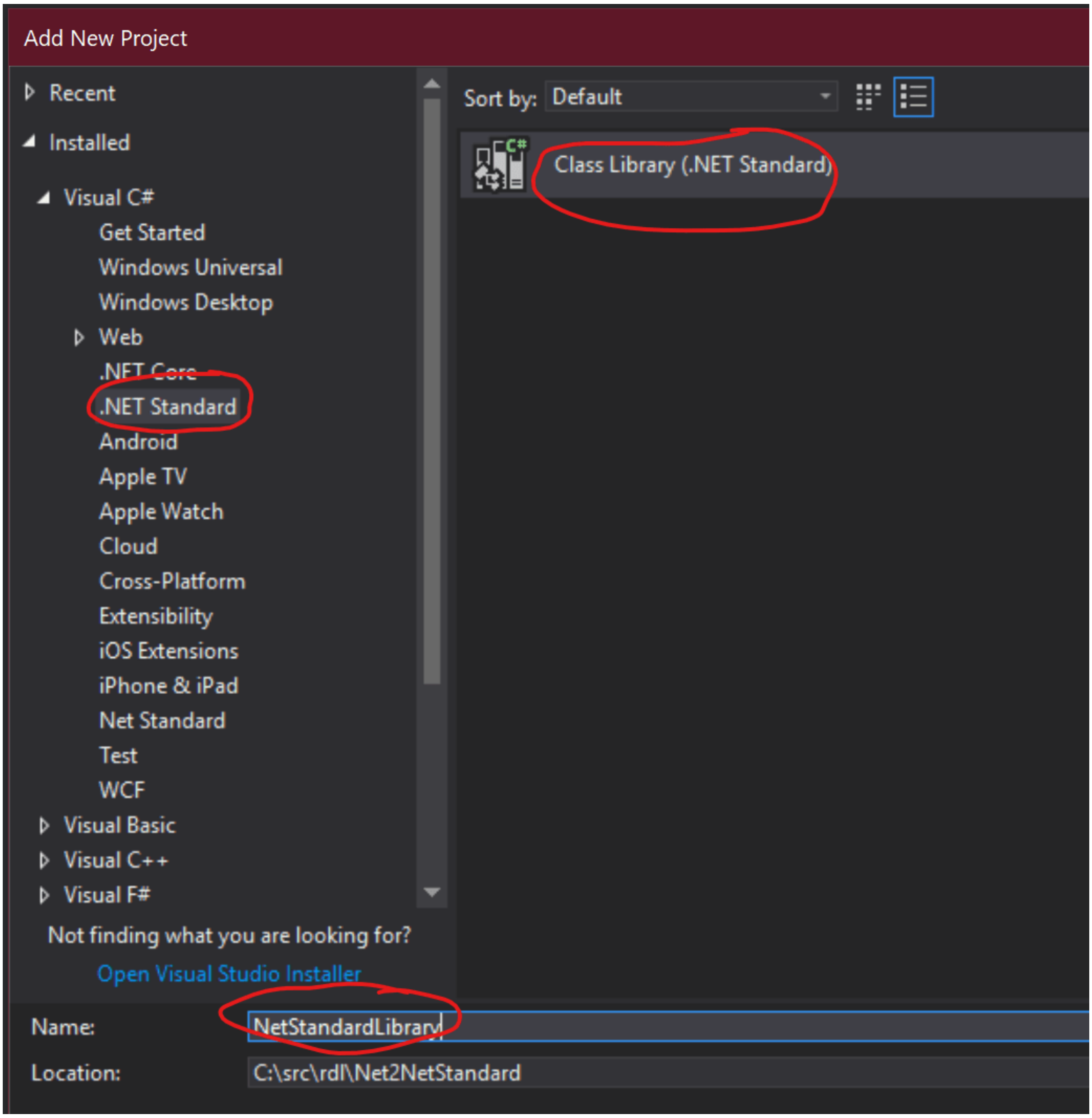
Hopefully your heart has slowed and your palms are now dry ☺

This is the point where you'd do a commit/push/PR of your code to finalize the shift of the files to their new shared project home. All in preparation for the next step where you'll finally get to .NET Standard.

Step 4: Create .NET Standard Project

To recap, you've updated to .NET Framework 4.6.1+, you've moved non-UI code out of the UI to its own class library, and now those code files are in a shared project, while still being compiled by the .NET Framework class library so production is unaffected.

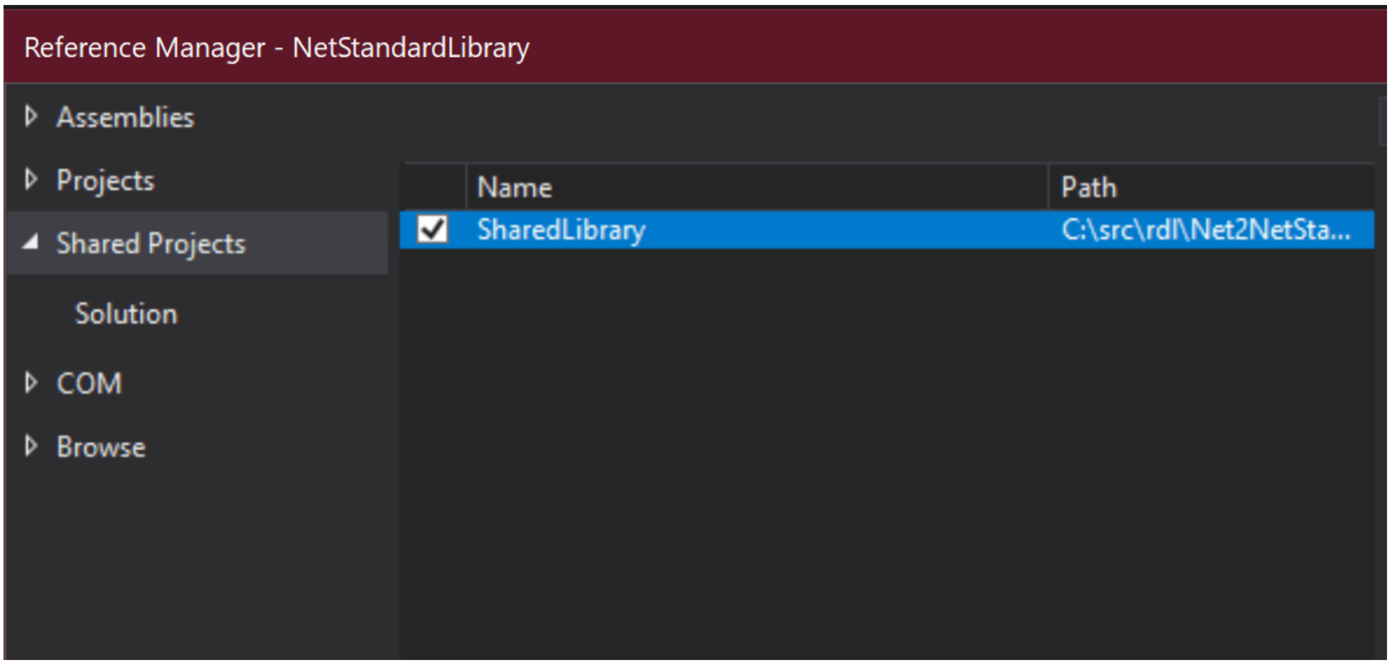
Now you can add a new .NET Standard Class Library project to the solution, the first real step toward the future!



Name: NetStandardLibrary

Location: C:\src\rdl\Net2NetStandard

With that done you can add a reference to the same SharedLibrary project so that *exact same set of code files* will be compiled by this new project as well.



	Name	Path
✓	SharedLibrary	C:\src\rdl\Net2NetSta...

If you try and build the solution or .NET Standard project now you'll find that it won't build. That's because the project is missing some dependencies. However, the original .NET Framework project should keep building fine, production remains unaffected.

Step 5: Mitigate Dependency Conflicts

The new .NET Standard project needs references to Newtonsoft.Json and the Entity Framework, much like the original .NET Framework project. The code makes use of these two packages and won't build without them.

I didn't pick these two dependencies by accident. Newtonsoft.Json has a NuGet package that supports .NET Standard. Entity Framework does not. These two dependencies exemplify likely scenarios you'll encounter with real code. The possible scenarios are that your existing dependencies:

1. Do not have .NET Standard support, and there's no alternative
2. Already have .NET Standard support with the current version
3. Already have .NET Standard support *if you upgrade to the latest version*
4. Do not have .NET Standard support, but a new equivalent exists

Scenario 1

Scenario 1 is a worst-case scenario that may be a roadblock to forward movement. If you have a dependency on a DLL or NuGet package that has no .NET Standard support, and there's no modern equivalent to the functionality, then you'll almost certainly have to wait until such support does exist or write it yourself.

Scenarios 2 and 3

If you are in scenario 2, where the existing version of your dependency already has .NET Standard support, then reference the same version in your .NET Standard project as in your exisitng projects and your code should continue to compile and work as-is. This is the simplest scenario.

The dependency may fit into scenario 3, where a newer version of the package supports .NET Standard, but not the version you are currently using. This is quite common with Newtonsoft.Json, where the most commonly used version is quite old, but the more recent versions support .NET Standard.

In this case you may be able to upgrade your production projects to the latest version and use the same version for both .NET Framework and .NET Standard. This incurs some risk to production, because you are upgrading a dependency, but it is often the best solution.

In the case that you can't upgrade the version used by production, you'll need to leave the old package version reference in your .NET Framework project and use a newer version in the .NET Standard project. In this case however, you may have to deal with behavior or API differences between package versions and you should treat this as scenario 4.

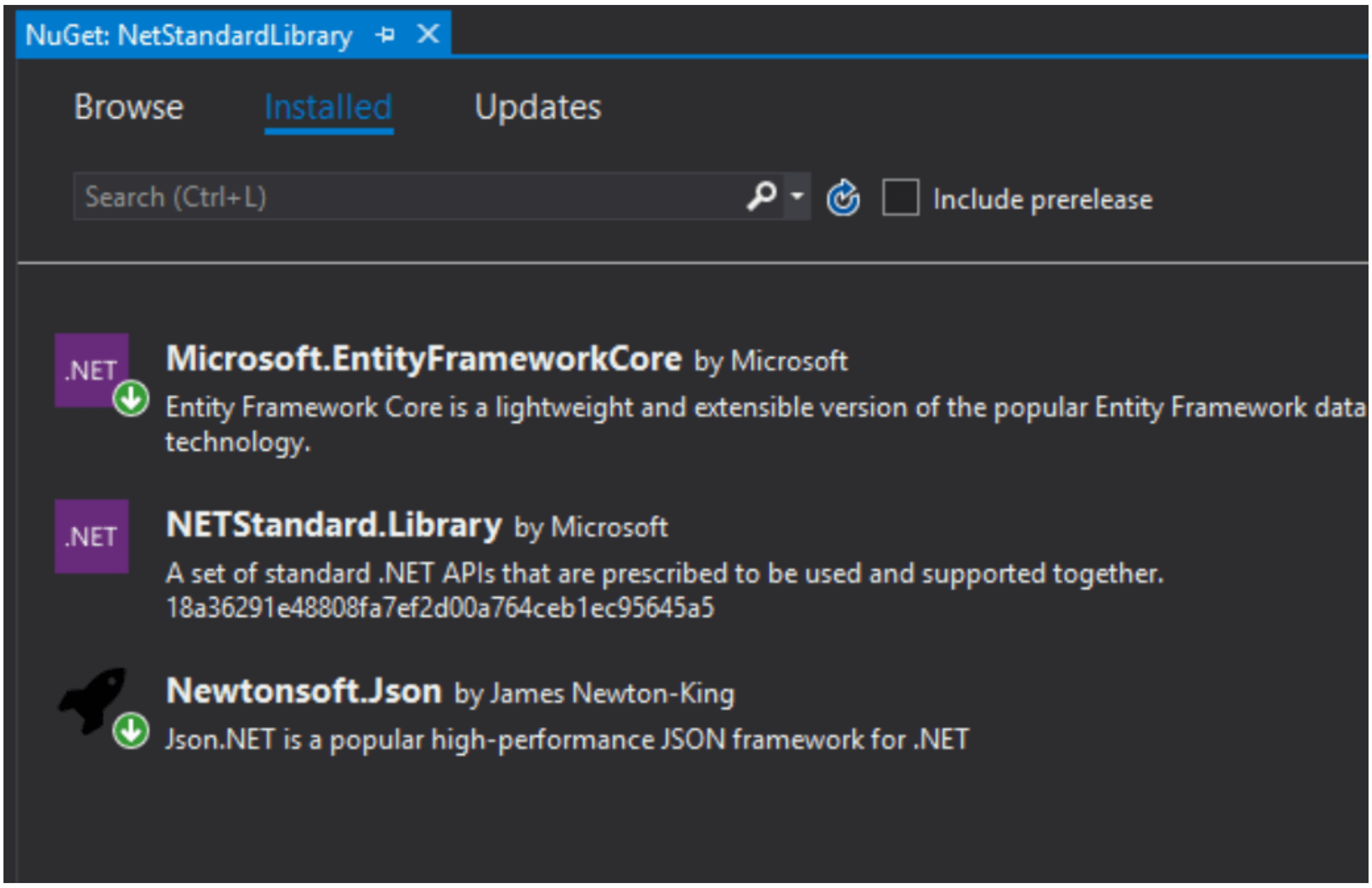
Scenario 4

Entity Framework is an example of scenario 4. Microsoft chose not to carry the existing (legacy?) Entity Framework forward. Instead they implemented something new called *Entity Framework Core*. This new equivalent offers the same *conceptual* functionality, but with a new implementation and API, so it is absolutely not code-compatible with the old Entity Framework in use in production.

I'll discuss two solutions to scenario 4: compiler directives and upgrading production.

Scenario 4: Compiler Directives

In the .NET Standard project, add references to the latest Newtonsoft.Json and EntityFrameworkCore packages from NuGet.



Microsoft.EntityFrameworkCore by Microsoft
Entity Framework Core is a lightweight and extensible version of the popular Entity Framework data technology.

NETStandard.Library by Microsoft
A set of standard .NET APIs that are prescribed to be used and supported together. 18a36291e48808fa7ef2d00a764ceb1ec95645a5

Newtonsoft.Json by James Newton-King
Json.NET is a popular high-performance JSON framework for .NET

You'll find that the project still won't build, because the existing code uses the old Entity Framework API. It is an scenario 4 dependency.

But you shouldn't get any errors compiling the code using Newtonsoft.Json, because it is a scenario 2 dependency.

The offending Entity Framework code is in the `PersonFactory` class:

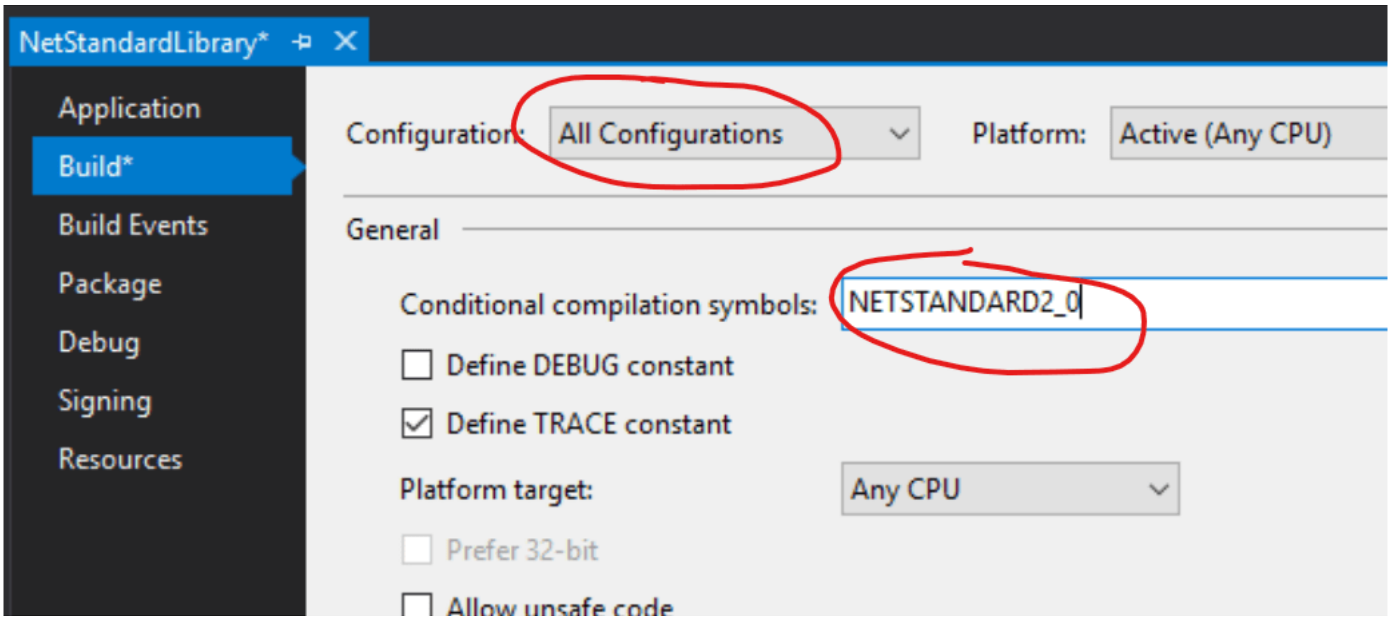
```
using System.Data.Entity;

namespace FullNetLibrary
{
  public class PersonFactory
  {
    public void GetPerson()
    {
      using (var db = new DbContext(""))
      {
      }
    }
  }
}
```

There are two problems in this trivial case. First, the namespaces are different, so the `using` statement is invalid. Second, the API for interacting with entity contexts has changed, so the `new DbContext` statement is invalid. In a more realistic scenario more parts of the API would be invalid as well.

The goal is to minimize changes and risk to production code, while enabling the .NET Standard code to move forward. Remember that this *exact same code file* is being compiled for two different targets: once for .NET Framework, and once for .NET Standard (where it fails).

The solution is to use *compiler directives* so the code file can include code that is only compiled for one target or the other. The first step is to define a constant in the .NET Standard project's Build tab.



Configuration: All Configurations
Platform: Active (Any CPU)
Conditional compilation symbols: NETSTANDARD2_0
Define DEBUG constant: ☐
Define TRACE constant: ☑
Platform target: Any CPU

You can name the constant whatever you'd like, but `NETSTANDARD2_0` is a defacto standard.

Then in your code file you can use this constant in a compiler directive. For example:

```
#if NETSTANDARD2_0
using Microsoft.EntityFrameworkCore;
#else
using System.Data.Entity;
#endif
```

What happens here is that when the .NET Framework project builds there's no `NETSTANDARD2_0` constant defined, so the compiler only uses the `using System.Data.Entity;` code. Conversely, when the .NET Standard project builds the constant is defined, so the compiler only uses the `using Microsoft.EntityFrameworkCore;` code.

At this point you may ask whether this won't get extremely messy to have these `#if` statements scattered throughout your code. And that is a valid concern. There are three scenarios to consider within a code file:

1. No code differences exist between the .NET Framework and .NET Core targets
2. Very few code differences exist between the targets
3. Many code differences exist between the targets

In scenario 1 you don't need compiler directives, so there's no issue. And that'll happen quite often with business logic, where the use of external dependencies is often very low.

Scenario 2 is a judgment call. What qualifies as "few"? My recommendation is that if 80% of the code is common and 20% is different, then you should use `#if` statements on a line-by-line or focused block-by-block scenario. This will result in a code file having numerous compiler directives, but *most* of the code will remain common across both targets.

Scenario 3 is where so much code is different that if you start scattering compiler directives through the code it would become unreadable. Again, my recommendation is that if more than 20% of your code will be different you should consider scenario 3. In this case you should duplicate the code within the file, essentially creating a different set of code for each platform. For example:

```
#if NETSTANDARD2_0
using Microsoft.EntityFrameworkCore;

namespace FullNetLibrary
{
  public class PersonContext : DbContext
  {
    public DbSet<Person> Persons { get; set; }
  }

  public class PersonFactory
  {
    public void GetPerson()
    {
      using (var db = new PersonContext())
      {

      }
    }
  }
}
#else
using System.Data.Entity;

namespace FullNetLibrary
{
  public class PersonFactory
  {
    public void GetPerson()
    {
      using (var db = new DbContext(""))
      {
      }
    }
  }
}
#endif
```

Notice that there's no code that's compiled for both targets. Instead the `#if` statement is used to create an implementation for .NET Standard, and another implementation for .NET Framework.

In a sense this is the lowest risk solution, because the .NET Framework production code *is entirely unchanged*. However, it is also the least maintainable solution, because the entire class has been duplicated, so future changes must be made to both sets of code.

Option 3: Upgrading Production Code

There's another alternative to using compiler directives, and that is to upgrade your production code to use the new dependency. This solution is only available in the case that the new NuGet package not only supports .NET Standard, but also supports .NET Framework. EntityFrameworkCore is an example of this, where you can use the new EntityFrameworkCore package from .NET Framework code.

Obviously this solution brings risk, because you are rewriting your *existing production code* to use the new library. That'll require good unit and acceptance testing of your production code to make sure nothing is broken by the changes.

On the upside, this solution helps keep the common codebase clean and unified. In the Net2NetStandard example, the `PersonFactory` code can end up looking like this:

```
using Microsoft.EntityFrameworkCore;

namespace FullNetLibrary
{
  public class PersonContext : DbContext
  {
    public DbSet<Person> Persons { get; set; }
  }

  public class PersonFactory
  {
    public void GetPerson()
    {
      using (var db = new PersonContext())
      {

      }
    }
  }
}
```

Same code for both the .NET Framework and .NET Standard targets. *But only if* the old Entity Framework reference in the production .NET Framework project is replaced with the new EntityFrameworkCore reference.

This often comes dangerously close to a "big bang" solution, and incurs real risk to the existing software. But there's also a very real upside in terms of maintaining a common codebase for development, testing, and maintenance over time.

Step 6: Mitigate Code Conflicts

The final issue you may encounter is pure code conflicts between your .NET Framework code and what can be done in .NET Standard. This is very uncommon, because .NET Standard describes so much of the functionality normally used by .NET code. However, if you are using some fancy bit of reflection or other "non-mainstream" parts of .NET you could find that your code won't compile for .NET Standard.

Solving this is really the same as Option 3 when dealing with dependency differences: use compiler directives. Or rewrite your "non-mainstream" production code to use techniques that are supported by .NET Standard.

Step 7: Have a Glass of Bourbon

or your beverage of choice

Not that you are done at this point, but you are on the path. In some ways finding the path and getting onto the path is the hardest part. The rest of the work might take months or years, but at least your code is in a structure where it is *possible* to migrate forward, while still maintaining the legacy deployment.

Yes there's some risk and additional unit testing (and acceptance testing) required as you make changes to the legacy code, which also changes the future code. That's a net benefit though, because at least you don't have to write those changes *twice* every time thanks to having a unified codebase.

There's a bit more risk (and therefore testing) required when making changes to the unified codebase for future code, because those changes will usually also impact the legacy app. But you have some control over that impact via compiler directives, and in many cases your business stakeholders will see this also as an advantage because they'll get some new features/capabilities in the existing legacy app even as you build them for the future state.

The point is that you've done the heavy lifting to establish a way forward that is at least achievable. So take a little time and have a small celebration. You deserve it!

2019年1月11日 下午 12:06:07 (Central Standard Time, UTC-06:00) # Disclaimer