

Revisiting modern .NET for the FoxPro Interop Developer

by Rick Strahl

prepared for *Virtual FoxFest, 2022*

[Session Example Code](#) on GitHub



If you're building modern applications that need to interface with various system or operating system features, you'll likely need external functionality that isn't available natively in FoxPro. Whatever your needs are, you can probably find this functionality in .NET either via built-in .NET system features, or by way of open source or third party libraries. For Windows applications .NET is easily the most comprehensive integration solution as it provides an interop layer to integrate directly from FoxPro.

As FoxPro developers we can take advantage of .NET by using .NET as a proxy to access .NET features or libraries. We can call into .NET code from FoxPro either via out-of-box COM Interop features, or with the [open source wwDotnetBridge Interop library](#). The latter gives you many more features and more control over how you interact with .NET.

While both COM Interop and wwDotnetBridge can directly call .NET code, if your code requires complex logic that has to be accessed in .NET, you'll find that these approaches often are cumbersome as FoxPro's support for many .NET features is limited. wwDotnetBridge gives access to most advanced .NET concepts, but even so it's often much more verbose than native .NET code.

For this reason it's often much easier to create a small .NET component to perform the actual .NET code integration, and then expose the created component to FoxPro. This results in cleaner code, better performance and on the FoxPro side a clean interface that is custom tailored to the integration's needs.

Yes, this approach involves writing .NET code - C# most likely - and creating a small project, adding .NET classes, building and compiling into a .NET assembly (a DLL) which you can then call from FoxPro. If you've never used .NET before this can be intimidating, but the reality is that this component integration approach is actually one of the easiest ways to ease into getting started with .NET, as you deal with a small very business specific piece of code. Building library level code lets you skip over some of the high level decisions like what UI framework to use and how to build your UI, but instead focus on pure business logic.

In short, building integration components is a great way to get your feet wet with .NET and think about how you can use it to extend your applications beyond just FoxPro code.

The timing of this session is also well timed IMHO: These days it's a lot easier to create a .NET project without having to install a massive Visual Studio and other support tools. These days you can install a single .NET SDK and use the Command Line or generic editor tools to create a .NET project, build and compile it. The

process and resources involved are a lot simpler than it used to be in the past. Visual Studio still is easier, but it's no longer a requirement - in fact you can everything you need to build from the command line these days.

In this session I want to show you get started with using .NET via [wwDotnetBridge](#), initially to call some .NET components, and then show how you can create your own .NET components with minimal effort and use them from FoxPro. In the process I'll explain the key concepts on how this process works and for things that you need to watch out for in the Interop process.

So strap in, and let's get started...

Why use .NET with FoxPro

FoxPro is a very old tool and while capable, let's face it there are many new technologies out there that provide rich access to features that FoxPro can't natively access. In fact these days there are many things that aren't even exposed to regular Win32 APIs any more.

Extend FoxPro's Reach

Increasingly new operating system features, and certainly many third party and API integrations are provided for Windows developers in the way of .NET libraries. .NET is not the only way that you can extend FoxPro but it's certainly one of the most common one as it provides deep integrations with many different features.

The reason for this is simple: .NET code is relatively simple to create and there's lots of tooling available today to do it. It also helps (or hurts 😊) that .NET, while completely Open Source, is essentially a Microsoft supported product. For Windows it's easily the most popular platform that's used for integrations of all kinds both for Windows OS features themselves as well as for third parties that provide service or feature APIs to integrate with.

There are 1000's of libraries available and .NET includes a prolific package manager repository call NuGet that makes it easy to add functionality to .NET projects.

Sounds good, but what does that have to do with FoxPro? A lot it turns out, because you as a FoxPro developer can relatively easily integrate with .NET either using direct Interop with .NET features via native COM Interop built right into .NET, or via the Open Source [wwDotnet Bridge Library](#) library which provides much enhanced functionality to access .NET code from FoxPro. But even beyond that, .NET is easy enough to pick up to create components that can provide elaborate functionality that would be very tedious to build via the Interop mechanisms available. While possible, it's often easier to build a small custom *wrapper component* in .NET and instead call it from FoxPro, rather than writing the verbose code with FoxPro and Interop functionality.

Modern .NET is easier than old .NET

For many, .NET has a stodgy image of a big and unwieldy Microsoft product of the past. A lot has changed in recent years (since about 2015) when Microsoft worked through a major overhaul of the .NET platform.

In that overhaul the core runtime was completely overhauled and rewritten and the entire .NET eco-system revamped to support among other things:

- cross-platform support on Windows, Mac and Linux
- focus on high performance runtime improvements
(*.NET is now amongst the fastest platforms for Web applications*)
- a completely new and much simpler project system

Revisiting modern .NET for the FoxPro Interop Developer by Rick Strahl

- a single point SDK to build, test, run .NET code
- Command line tooling
- a move away from Visual Studio only for .NET development

All of this has resulted in many big improvements in the .NET runtime in terms of performance and scalability to the point that .NET is now among the top fastest server interfaces and the fastest among 'integrated platforms' (like nodeJs,python etc.).

The improved project system and build tools have allowed moving away from the requirement of Visual Studio, which is one of the reason a lot of people in the past have been turned off by .NET and Microsoft tools in general. It's now possible to use any editor you choose, and use command line tooling to build, test and run your projects, or use the openly available tooling that is integrated in most editor platforms. Since .NET is now cross-platform there are also many tools in Mac or Linux you can use to build and run .NET applications. In fact, I've recently switched to use [Rider from JetBrains](#) for much of my .NET development these days.

In short, it's possible today to do .NET development on your terms. Want a lightweight environment and build and test from the command line? You can do that. Want to use your editor of choice with a minimal .NET tooling integration? Yup, you can do that too. Want to still use Visual Studio and get all the rich functionality it provides beyond the basics? Yeah, Microsoft still services the Visual Studio behemoth too.

That said, big IDEs like Visual Studio or Rider are still the best way to work with .NET for large applications and complex code bases. But if you just need to build a small component for integration with your FoxPro application, you may not need anything more than an editor and a couple of commands from the Command Line to compile your code.

.NET Framework or .NET Core

Earlier I mentioned Microsoft did a major overhaul of .NET and the result of that is what is now known as **.NET Core**. It's a completely separate version of .NET from the old classic **.NET Framework**.

All of the major improvements to the runtimes have been made to .NET Core. The Full Framework is 'done': .NET 4.8 is the last version of .NET Framework according to Microsoft. This version will only get bug fixes and security updates, but no new feature improvements.

So, sounds like you'd want to use .NET Core from here on out right?

FoxPro Applications should prefer the classic .NET Framework

Well, for FoxPro application development I would recommend sticking to .NET Framework for a few reasons:

- The .NET Framework Runtime is installed on Windows
- The framework 'just works' - nothing to install or configure

This means if you create a .NET Framework component and ship it with your application, it will just work. Load the DLL and fire away you're off to the races.

For .NET Core you have to ensure that the correct .NET Core Runtime is installed on the machine. The installs are not small either and you need to ensure you install the right major version at least of the runtime to match the target of your components. This is less an issue for components which can work more easily with mismatched .NET Core versions, but full applications have to match the major runtime version in order to run at minimum.

For this reason I personally prefer creating .NET components for use with FoxPro using .NET Framework. Because these components will just work on Windows it's one less thing I have to worry about when deploying an application.

FoxPro and .NET Interop

So you've decided you need to interop with some .NET Components - how do you do that? There are a couple of options available to the FoxPro developer.

- **.NET COM Interop**

.NET supports a native COM interface wrapper that allows **.NET components to be exposed as COM objects**. Unfortunately the native support is **very limited** and requires explicitly registered and marked components which means either you have to use your own components or the handful of .NET components that explicitly support COM Interop. COM by itself is also limited in .NET type features it can access, due to limitations in the COM type system compared to .NET's rich type system. For example, you can't call static members, you can't easily access collections and dictionaries, and even many simple types like Guids, Decimals or Longs can't be accessed directly.

- **wwDotnetBridge**

wwDotnetBridge is an Open Source FoxPro library that provides a proxy interface into .NET. It also uses COM Interop, but works around many limitations of COM Interop by providing Proxy functionality inside of .NET that allows accessing features that native COM interaction cannot access directly. It requires no explicit registration for components, can access most .NET types directly, and supports common features like access to static members, collections and dictionaries and many other problem types not supported via pure COM.

Of these two using wwDotnetBridge is almost always the better choice, as it removes the COM registration and COM marked requirement of stock COM Interop. That feature alone is enough to use wwDotnetBridge in my view - once you get the .NET object reference behavior is initially identical to COM Interop. But beyond that wwDotnetBridge then provides proxy functionality to work around COM Interop limitations. Personally I **never** use .NET COM Interop from FoxPro - there's no benefit in its use and no downside for using wwDotnetBridge except for the two DLL distribution requirement.

Built in COM Interop

.NET includes a COM based interop mechanism that allows for instantiating and accessing of .NET components via COM. The native COM Interop mechanism allows you to instantiate a .NET component via COM and return it as a COM object reference. .NET includes a COM Callable layer that at runtime turns .NET objects into COM accessible objects.

But the built-in COM Interop has a catch: It requires that components are explicitly marked for COM Interop and that the components is registered in the registry as a COM component using a special tool called `regAsm`. `regAsm` creates COM entries in the registry so you can instantiate a .NET Component as a COM object using standard `CREATEOBJECT("ComClass.Class")` syntax. Unfortunately that limits what you can access with this native COM mechanism as almost no native .NET components, or those from third parties are natively exposed as COM objects. Most components lack both the appropriate `[ComVisible]` attributes or the attributes required to allow FoxPro to instantiate these COM components. Effectively this limits the native COM Interop

features to components that you yourself build with the additional burden of requiring that these components have to be registered and re-registered during installation of the application. In short using COM Interop is messy.

wwDotnetBridge

To work around some of these limitations the wwDotnetBridge library provides a number of enhancements that allow any object to be instantiated as well as providing a host of helper features that allow you to work around the limitations of COM only access to .NET components.

- Access most components
- No COM registration required for instantiation
- Support for multiple constructors and constructor parameters
- Helpers to access type that can't be passed over COM
 - any value type, long, decimal
 - collection types and dictionaries
 - anything using .NET Generics
- Type wrappers for incompatible types
 - ComArray for collections
 - ComValue for problem types
- Automatic Type Conversions

Like native COM Interop, wwDotnet Bridge relies on the same COM callable wrapper in .NET and can access all the same features that .NET Interop has access to directly. Where it shines is to work around the limitations of .NET → COM communication. COM is an old protocol and it has very strict type rules of what it can and cannot work with and wwDotnetBridge can work around this.

wwDotnetBridge accomplishes this by acting as a .NET Proxy that optionally can be called to marshal calls into .NET and back through a .NET component that runs inside of .NET. Because the component lives in .NET It can access native .NET functionality and return data in a way that FoxPro can manage. For example, the `Long` .NET datatype isn't supported in COM but wwDotnetBridge can intercept a result value of `Long` and convert it into an `int` for a small value or a `double` which FoxPro does support.

Other problems that wwDotnetBridge solves, are that value types or generic types cannot be accessed through COM because they don't have a fixed reference implementation. wwDotnetBridge works around this by using indirect referencing and calling of members using helper methods like `InvokeMethod()`, `GetProperty()` and `SetProperty()` which use Reflection inside of .NET to make the calls and marshal the result values back to FoxPro.

Finally you can also access static methods and properties with wwDotnetBridge. Static members are instance-less operations - sort of like FoxPro UDFs, but tied to a .NET type. Since static methods don't have an instance there's no way to create a COM instance and call that method - there is no instance. So rather wwDotnetBridge uses Reflection to call the static method or property and marshal the result back to FoxPro.

In a nutshell, wwDotnetBridge provides workarounds for many common .NET scenarios that don't directly work with COM Interop. You can still use direct COM access to any method or property that support it, but for those that don't, wwDotnetBridge has workarounds via its proxy mechanism.

How wwDotnetBridge works

Even though you are still using COM Interop with wwDotnetBridge, it provides additional features to work around the limitations of that model.

The behind the .NET Core moniker is that this update is leaner and more focused on the core runtime, and rather than cramming everything under the sun into this runtime as the full .NET Framework did with all things Windows, it relies on a core framework layer with everything else provided via NuGet packages - components essentially - that are added on top of the Core to provide additional functionality. It's more modular so that the runtime itself is smaller.

This new version also breaks out separate runtimes for the 'core runtime' that every .NET application uses, with other frameworks such as the ASP.NET Runtime or the Windows Desktop Runtime being shipped as separate components that an application can target. So when you're building a Windows application you're not shipping ASP.NET parts and vice versa.

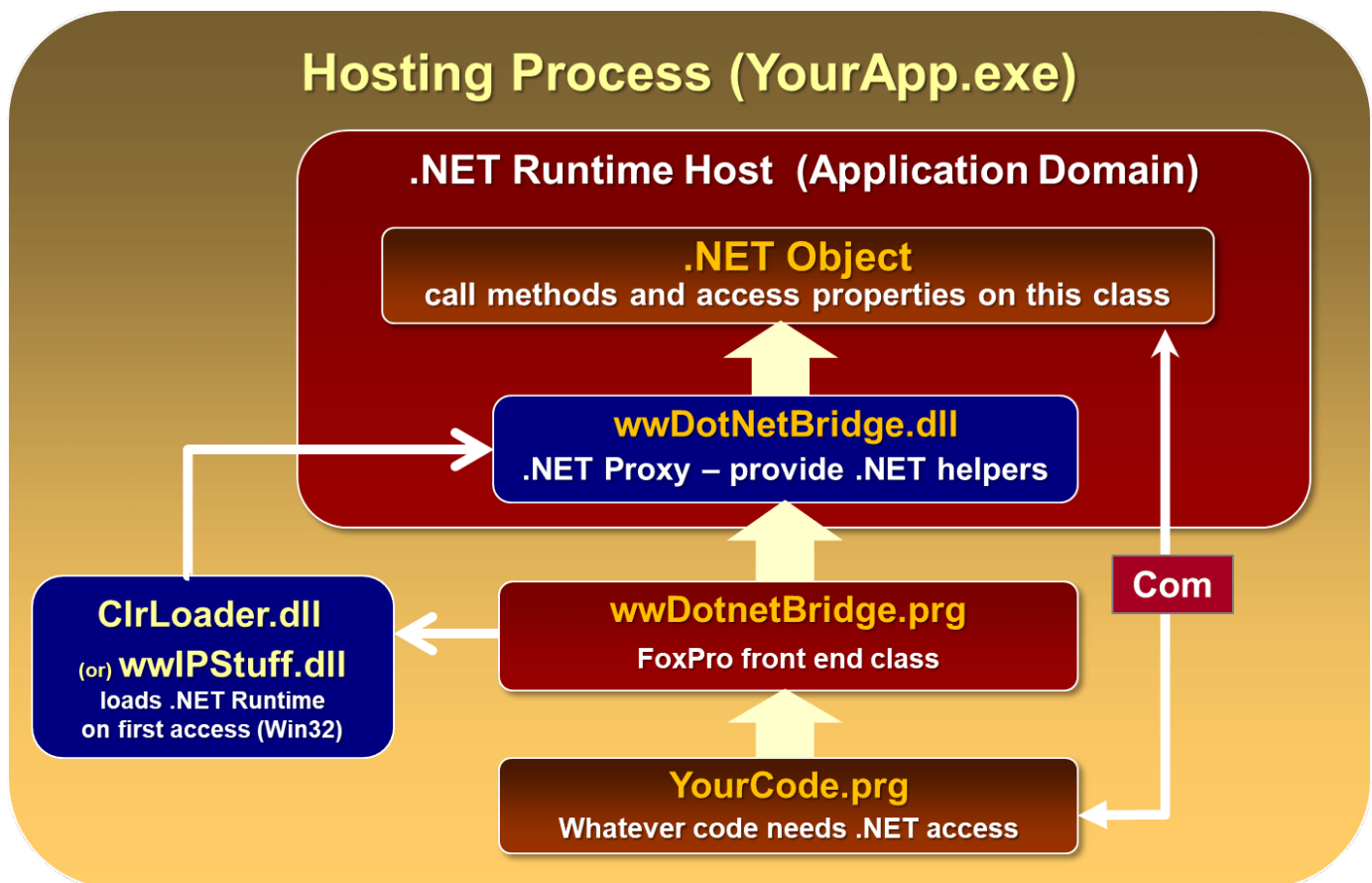
FoxPro Applications are better off sticking

As mentioned wwDotnetBridge works as a proxy that allows instantiation of .NET types, passing them back to FoxPro, and then providing a set of helpers that can proxy individual method calls or member access through .NET for methods or properties that can't be directly accessed via COM Interop.

There are two main features:

- Type invocation
- Proxy functionality

The type invocation is somewhat complex. Here's a diagram that shows the base functionality:



- Your code calls the FoxPro `wwDotnetBridge.prg` class to instantiate a type
- On first load, FoxPro calls a Win32 function in `ClrLoader/wwIPStuff` that loads the .NET Runtime into FoxPro
- As part of that loader the .NET `wwDotnetBridge` component is loaded into .NET
- The `wwDotnetBridge` instance is passed back to FoxPro as a COM object
- The FoxPro `wwDotnetBridge` component receives this .NET proxy instance and hold on to it
- The proxy is now loaded and ready

At this point `wwDotnetBridge` is ready to instantiate new .NET objects.

- The FoxPro `wwDotnetBridge` instance is used to create a .NET object
- The proxy instantiates the class in .NET
- The proxy passes back this instance via COM to FoxPro
- FoxPro code receives the COM instance and can now access members

FoxPro now has a COM instance much in the same way as COM instantiation would have yielded with the built-in COM Interop. The big difference is that the .NET component did not need to be registered in any special way: No `RegAsm` or special marker interfaces are required. Any type can just be instantiated.

Once the type is in FoxPro you can now use COM to access methods and properties - as long as the .NET types for parameters, result values and property values are compatible with COM.

If they are not, COM calls will fail with exceptions to the effect that the Interface is not supported (**No such interface**). If that's the case, `loBridge.InvokeMethod()`, `loBridge.GetProperty` and `loBridge.SetProperty()` can be used to make the calls. These intrinsic methods use Reflection in .NET to

pass and retrieve values and pass them back to FoxPro. If a type used is of a known incompatible type, `wwDotnetBridge` fixes up the type either by mapping to a FoxPro compatible type directly, or to a .NET wrapper type. For example, arrays and collections are automatically mapped to a `ComArray` instance which is a wrapper around an array. Most other single incompatible value types are mapped to a `ComValue` instance which wraps the value into a .NET object that stays in .NET with methods that allow retrieval of the value via some translation.

Basic wwDotnet Bridge Examples

All of that is pretty abstract, here's what running `wwDotnetBridge` looks like.

The first example is a simple one that loads a third party assembly to handle Markdown conversion from Markdown text to HTML using the `Markdig` .NET Markdown parser library.

```
*** Load wwDotnetBridge
do wwDotNetBridge

LOCAL loBridge as wwDotNetBridge
loBridge = CreateObject("wwDotNetBridge")

*** Load a third party library - full path or in FoxPro path
loBridge.LoadAssembly("markdig.dll")

lcMarkdown = "Hello **cruel world**."    && some markdown

*** Create an object instance - pipeline parameter for ToHtml
LOCAL loBuilder, loPipeline
loBuilder = loBridge.CreateInstance("Markdig.MarkdownPipelineBuilder")
loPipeline = loBuilder.Build()

*** Invoke a static method
lcHtml = loBridge.InvokeStaticMethod("Markdig.Markdown","ToHtml",lcMarkdown,loPipeline,
null)

? lcHtml
```

This example demonstrates a few of `wwDotnetBridge`'s features:

- Loading of a third party library
- Registrationless instantiation of the builder
- Direct invocation via COM (`loBuilder.Build()`)
- Indirect invocation in this case of a static method

This example is super simple, but that little bit of code provides powerful functionality as you now have the full power of Markdown parsing in your FoxPro application with just a few lines of code.

Let's look at another example that demonstrates more of the intrinsic features of `wwDotnetBridge`:


```

*** Load library
DO wwDotNetBridge

*** Create instance of wwDotnetBridge
LOCAL loBridge as wwDotNetBridge
loBridge = CreateObject("wwDotNetBridge")

*** Create an instance of X509Store
loStore = loBridge.CreateInstance("System.Security.Cryptography.X509Certificates.X509Store")

*** Grab a static Enum value
leReadOnly =
loBridge.GetEnumvalue("System.Security.Cryptography.X509Certificates.OpenFlags","ReadOnly")

*** Use the enum value
loStore.Open(leReadOnly) &&leReadOnly)

*** Alternately you can use the enum value if known
*loStore.Open(0) &&leReadOnly)

*** Collection of Certificates
laCertificates = loStore.Certificates

*** Collections don't work over regular COM Interop
*** so use indirect access
lnCount = loBridge.GetProperty(laCertificates,"Count")

*** Loop through Certificates
FOR lnX = 1 TO lnCount -1
  *** Access collection item indirectly
  LOCAL loCertificate as System.Security.Cryptography.X509Certificates.X509Certificate2
  loCertificate = loBridge.GetProperty(loStore,"Certificates[" + TRANSFORM(lnX) + "]")

  IF !ISNULL(loCertificate)
    ? loCertificate.FriendlyName
    ? loCertificate.SerialNumber
    ? loBridge.GetProperty(loCertificate,"IssuerName.Name")
    ? loCertificate.NotAfter
  ENDIF
ENDFOR

```

The comments in this code describe what's happening with the various wwDotnetBridge functions. Notice that this code **does not do an explicit** `LoadAssembly()` **call** because it uses features that are part of the core .NET runtime so `LoadAssembly()` is not required here.

There's a use of `GetEnumValue()` here which can be used to resolve an Enum value by it's type string representation. Enums are common in .NET and this is one way you can retrieve them.

There are several uses of `GetProperty()` in this code to retrieve properties that aren't directly accessible via COM first is the Count property on the certificate collection which is a custom collection type that COM does not support. The other two uses are shortcuts to access nested properties rather than traversing the object hierarchy and loading the intermediary objects into FoxPro - by using the nesting syntax these objects are directly referenced.

There's a lot more that you can do with wwDotnetBridge, and you can look at additional examples in the [online](#)

[documentation](#) or the [white paper](#).

Creating a Dotnet Component

Using the new .NET SDK tools you can now create .NET components without requiring a big development environment. You can use the command line tools - or a light weight editor like Visual Studio Code - that supports the generic OmniSharp .NET Build tools to build .NET components or applications.

However, if you are already doing .NET development, there are still some advantages to using Visual Studio, chief among them the ability to debug your .NET code interactively when called from your FoxPro code.

But for the core build functionality you can easily use the command line tools or Omnisharp to build your application.

What you need to build a .NET Classlibrary

In order to create a .NET component you need to [install the latest .NET SDK](#) (download the latest **LTS Release**), which includes all the required tools to build, run and debug .NET applications. The SDK also installs the .NET Core runtime along with the ASP.NET and Windows runtimes. Although you don't need that it's there if you choose later to build a full .NET application.

The base **.NET Core Runtime** which we are interested in for building FoxPro integration components includes:

- Class libraries
- Console applications

Distributed applications then require the .NET Core runtime that is of the same major version (ie. 6.0). The current version of the .NET Core runtime as I write this is `6.0.9`. The minor versions update quite frequently but these runtimes are forwards and backwards compatible to the same major version so as long as a v6.0 runtime is installed it works.

Creating a new .NET Project from the Command Line

With the SDK installed you can now create a new .NET Project. Out of the box, the latest .NET Tools do not support `.NET 4.x` projects, but you can still use the tools to create a project.

- Create a folder for your project
- Run `dotnet new classlib <projectName>` (or leave out the name and the folder name is used)
- Change the project's to `<TargetFramework>net472</TargetFramework>`
- Adjust project for C# 7 syntax (.NET Framework only supports up to C# 7)

So create your project in a folder of your choice. I'm going to use a project named `FoxProInterop` based on the following sample folder structure:

```
FoxProDotnet
- FoxPro      (FoxPro samples)
- Bin         (place compiled .NET binaries here)
- Dotnet
- FoxProInterop (Project Files in here)
```

So first start by changing to the folder where you want to create your project under. The create project process

creates a new project under the folder you choose with the `-n` name that you specify.

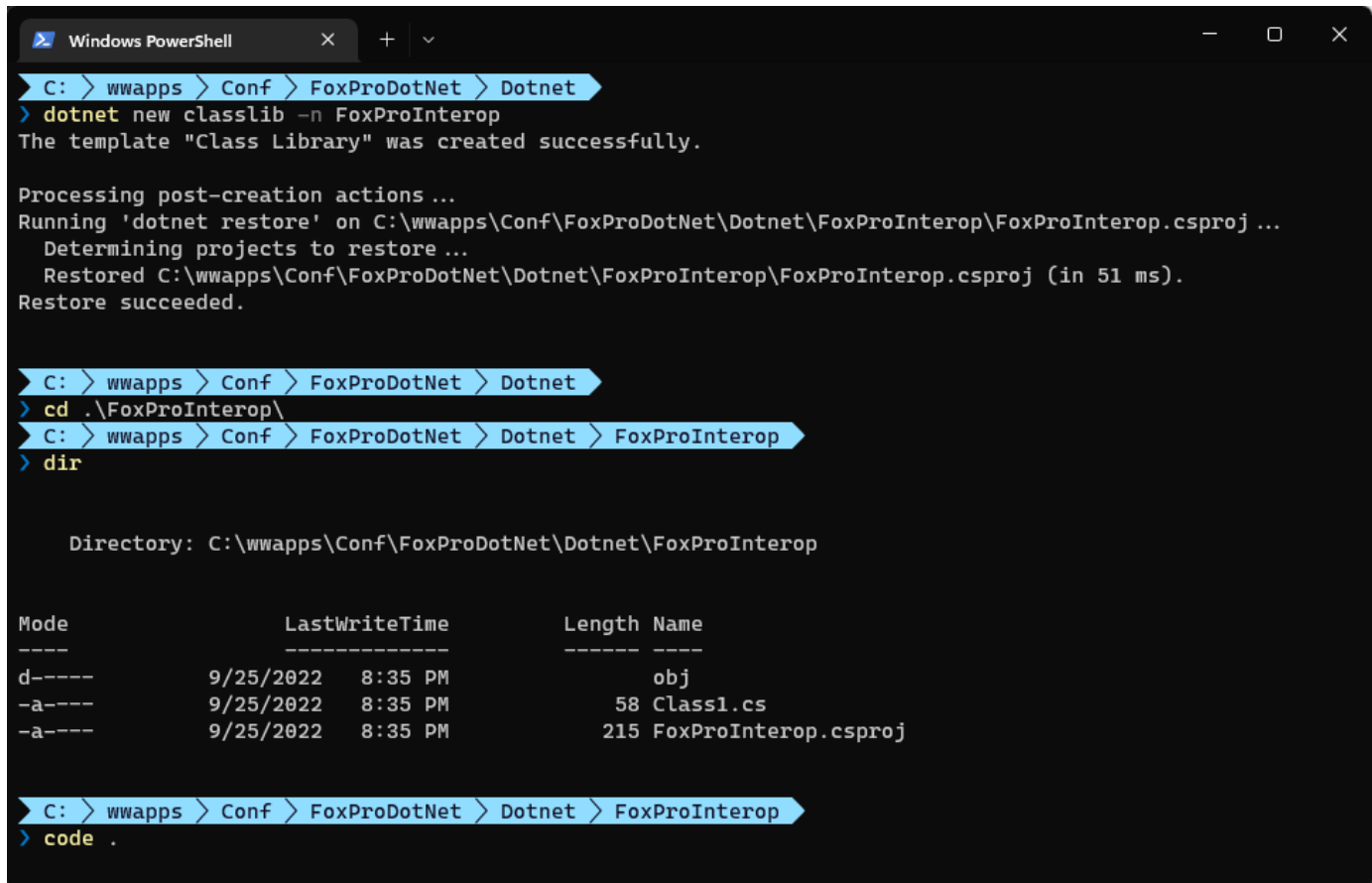
```
# go to folder where you want to create the project (project create below)
cd Dotnet

# Create the project
dotnet new classlib -n FoxProInterop

# Change to the created project folder
cd FoxProInterop

# Start up the editor (or open FoxProInterop.csproj)
code .
```

Here's what that looks like when you run it in Powershell:



```
Windows PowerShell
C: > wwapps > Conf > FoxProDotNet > Dotnet
> dotnet new classlib -n FoxProInterop
The template "Class Library" was created successfully.

Processing post-creation actions ...
Running 'dotnet restore' on C:\wwapps\Conf\FoxProDotNet\Dotnet\FoxProInterop\FoxProInterop.csproj ...
Determining projects to restore ...
Restored C:\wwapps\Conf\FoxProDotNet\Dotnet\FoxProInterop\FoxProInterop.csproj (in 51 ms).
Restore succeeded.

C: > wwapps > Conf > FoxProDotNet > Dotnet
> cd .\FoxProInterop\
C: > wwapps > Conf > FoxProDotNet > Dotnet > FoxProInterop
> dir

Directory: C:\wwapps\Conf\FoxProDotNet\Dotnet\FoxProInterop

Mode                LastWriteTime         Length Name
----                -
d-----          9/25/2022   8:35 PM             obj
-a-----          9/25/2022   8:35 PM             58 Class1.cs
-a-----          9/25/2022   8:35 PM            215 FoxProInterop.csproj

C: > wwapps > Conf > FoxProDotNet > Dotnet > FoxProInterop
> code .
```

Fixing up the Project for .NET 4.72

Next we need to make some changes to the generated project. The project creates a class library which is just a single class file. Using the new SDK project style in .NET code files and many others that are 'processed' as part of a project, don't have to be explicit added to a project, so you can just create a new file and it will be automatically *included in the project* and compiled as part of the library.

In fact the generated project file is extremely simple:

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

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

Notice that the project was created for .NET 6.0 - not .NET 4.72 or 4.8 as we would like to. However, that is supported, but unfortunately not through the command line options. To fix this we need to make a change - a simplification really - of the project file:

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

  <PropertyGroup>
    <TargetFramework>net472</TargetFramework>
  </PropertyGroup>

</Project>
```

The `<TargetFramework>` specified the framework that the project compiles to - I want it to compile `net472` which is the framework moniker for .NET Framework 4.72. For 4.8 it's `net48`. .NET Core versions are typically `net6.0` (current) or `net5.0` or `net7.0`.

Why does that matter?

Classic .NET Framework vs .NET Core

.NET Framework

The original, Windows only version of .NET that is now part of the Windows platform is the classic .NET Framework. It ships and is updated with Windows, so it's pre-installed on practically any Windows machine post Vista with version 4.5 or later installed on Windows 8.1 and later. The big advantage of this framework is that it's already installed - no extra installation it require and it just works. The .NET Framework is very rich and includes the vast majority of features that are available to .NET today, but this framework is officially done - meaning Microsoft is no longer adding new features to it. What you see is what you get basically.

This sounds bad, but for many years, most new improvements and features have come in the form of libraries and extensions that extend the framework rather than in the core framework.

- Pre-installed and updated as part of Windows OS
- Includes all Windows features out-of-box
- Tightly integrated with the OS
- Doesn't require any runtimes
- Versions are synchronized to Windows and fully interoperable
- Version 4.8 is the last version
- No longer updated with new features (only patches and security fixes)

.NET Core

Microsoft's latest incarnation of .NET is .NET Core which is a completely re-built version of .NET that is:

- Cross Platform (Windows, Mac, Linux)
- High Performance
- Light weight (smaller footprint and memory usage)
- Optimized for heavy server workloads
- Actively developed
- Requires one or more Runtimes to be installed
- Versions

Choices, Choices

The future *for Microsoft and .NET* is clearly in .NET Core, not .NET Framework. If you're building full .NET server applications there is no reason to build them in classic .NET any longer due to the new service frameworks and massive performance gains in Core. For desktop applications the situation is more varied - you get new language and framework features, all of the performance features but at the downside of having to deal with managing runtime distribution for the .NET Core runtimes and keeping versions up to date.

*For FoxPro the situation is different though, because as an external application calling into .NET you want the process to be as transparent as possible for the host FoxPro application. Installing an explicit runtime can be a big detriment and often may outweigh the benefit the integration provides.

For this reason I would still recommend that we continue to use the classic .NET Framework (`net472` or `net48`) for targeting any components you create for use with Visual FoxPro rather than .NET Core.

If you create your own components there's little reason to use .NET Core - you're not going to see any great performance gains or advantages for new language features in typical library integrations. Instead prefer the universal availability of .NET and not having to worry about installation of a huge runtime or even determining of whether it need to be installed in the first place.

The only reason you should consider using .NET Core from FoxPro is that you need to call a library that is only available for .NET Core. This should be relatively rare today, as most libraries these days either multi-target both .NET Framework and .NET Core or use `.NET Standard` which can be used from classic .NET.

Note that you can use either version using **wwDotnetBridge** - wwDotnetBridge recently added support for creating and accessing .NET Core components, although that too is a bit more complicated as the right runtime has to be located.

Bottom line: If at all possible use full .NET Framework with your .NET Integrations, unless you have to explicitly call a .NET Core only supported library.

Changing the Generated Class

So when the `dotnet new classlib` command created your project, you unfortunately can't target the `net472` target framework - it's not supported. While there's a `-f` target framework switch, classic .NET projects are not supported. If you don't specify a version the latest active version - `net6.0` in this case - is used.

Revisiting modern .NET for the FoxPro Interop Developer by Rick Strahl

We already changed the `<TargetFramework>net472</TargetFramework>` but we also need to fix the generated class, because it uses that latest C# syntax that is not supported in the older full framework versions.

The original C# class generated looks like this:

```
namespace FoxProInterop;
public class Class1
{

}
```

This code uses C# 10 implicit namespaces which removes an indentation level. This is not supported in C# 7.3 which is the latest version that full framework supports.

I'm going to add an explicit namespace and rename the class to `Interop` while I'm at it:

```
namespace FoxProInterop
{
    public class Interop
    {

    }
}
```

I'll also rename the file from `Class1.cs` to `Interop.cs`.

Next I'll add a `HelloWorld` method to the class:

```
using System;

namespace FoxProInterop
{
    public class Interop
    {

        public string HelloWorld(string name) {
            return "Hello World, " + name +
                ". Time is: " + DateTime.Now.ToString("HH:mm:ss");
        }

    }
}
```

Resources

- .NET Decompilers (discover classes/members/names)
 - .NET Reflector (old version (v6) is free)
 - [JetBrains DotPeek \(free\)](#)
 - [Telerik JustDecompile \(free\)](#)