Packaging Options for Rx.NET v.next

This document explores the pros and cons of various potential packaging strategies for the next version of Rx.NET.

The main driving factor behind this is that some projects (e.g. Avalonia) have abandoned Rx.NET because it forces applications targeting a Windows-specific TFM (e.g. net8.0-windows10.0.19041) to have a dependency on the Microsoft.Desktop.App framework, with the effect that tens of megabytes of unnecessary framework DLLs get deployed alongside the app when using self-contained deployment.

Reverting this is non-trivial because most of the obvious ways of fixing this break a greal deal of code that already uses Rx. In many cases, it leaves applications in a position where there's nothing they can do to fix the problem other than abandoning Rx.NET

It is our goal not to have anyone feel that they have to abandon Rx.NET. This document outlines the solutions under consideration, and describes the pros and cons of each approach.

Summary of Issues with Candidate Packaging

This table shows summaries of findings for each of the candidate packaging approaches. These approaches are described in detail in the following sections.

Packaging Option	**Questionable ref assembly tricks**		**Unwanted Microsoft.Windows-Desktop.App**		**Build failures**				**Runtime failures**		
	In legacy facade	**In main package**	**Legacy refs need attention**	**Imposed by System.Reactive**	**Desktop API use requires explicit FX ref where required**	**Rx UI use requires new package ref**	**Multiple Rx versions in scope**	**Other undiagnosed issues**	**Desktop API use requires explicit FX ref where required**	**Rx UI use requires new package ref**	**Other undiagnosed issues**
legacyfacade.1	Yes	No	NA	Yes	Yes	No	Yes	Yes	No	No	No
legacyfacade-refnoui.3	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	No
legacyfacade-refnoui-withfxref.1	Yes	No	NA	Yes	Yes	No	Yes	Yes	No	No	Yes
nofacade-refnoui.5	Yes	Yes	No	No	No	No	No	No	Yes	No	No

Straightforward legacy facade (legacyfacade.1)

This introduces a new main Rx package System.Reactive.Net with no UI framework dependencies. UI-framework-specific code moves into specialised packages such as System.Reactive.For.Wpf. The System.Reactive package becomes a legacy facade to maintain backwards compatibility, offering the same public API as in previous versions, using type forwarders to refer to the new homes of these types.

Issues in brief

We will not use this option, because it has two major weaknesses (one of which is addressed by the legacyfacade-refnoui variation described later):

- System.Reactive imposes a dependency on the Microsoft.Desktop.App framework
- Applications may find that they have two versions of Rx in scope simultaneously

That second problem is easy for applications to fix, but it adds friction. The first is more problematic because there's no good workaround.

Packaging Option	**Questionable ref assembly tricks**		**Unwanted Microsoft.Windows Desktop.App**		**Build failures**				**Runtime failures**		
	In legacy facade	**In main package**	**Legacy refs need attention**	**Imposed by System. Reactive**	**Desktop API use requires explicit FX ref where required**	**Rx UI use requires new package ref**	**Multiple Rx versions in scope**	**Other undiagnosed issues**	**Desktop API use requires explicit FX ref where required**	**Rx UI use requires new package ref**	**Other undiagnosed issues**
legacyfacade .1	Yes	No	NA	Yes	Yes	No	Yes	Yes	No	No	No

System.Reactive continues to provide UI-framework-specific types

The greatest weakness of this packaging option affects applications that have a transitive dependency on System.Reactive. Their only option for avoiding a dependency on Microsoft.Desktop.App (the framework reference that causes self-contained application deployments to grow by 30MB or more) is to set <DisableTransitiveFrameworkReferences>, but this doesn't always work: if the application itself also uses Rx directly, this can create a situation in which code does not compile.

Rx.NET is very widely used, so an application that uses a wide range of NuGet packages may well find that one or more of those happens to use Rx, meaning that the application has a *transitive* dependency on Rx. For many years to come it is likely that some libraries will remain on Rx 6.0, meaning that this is going to be a very common scenario. Since this legacyfacade option does not deal with this scenario well, it is not a viable solution.

Why?

To understand why this problem is fundamentally unavoidable when using the simple legacy facade approach, remember that in this model, the essential role of System.Reactive is to provide backwards compatibility, both at build time and at runtime. This means two things:

- applications *will* end up with a dependency on this package if it uses any libraries that depend on an old version of Rx
- New versions of System.Reactive have to provide the same API as in Rx v6

So although this approach separates the UI-framework-specific code out into new components, the System.Reactive legacy facade is obliged to reference those components to be able to forward types to them. So if you were to look at the dependencies for System.Reactive, 7.0.0-preview-legacyfacade.1.ga1159cd7f3 in the NuGet package viewer in Visual Studio you'd see it list these dependencies:

- .NETFramework,Version=v4.7.2
    - System.Reactive.Net
    - System.Reactive.Net.For.WindowsForms
    - System.Reactive.Net.For.Wpf
    - System.Threading.Tasks.Extensions
- UAP,Version=v10.0.18362
    - System.Reactive.Net
    - System.Reactive.Net.For.WindowsRuntime
    - System.Threading.Tasks.Extensions
    - Microsoft.NETCore.UniversalWindowsPlatform
- net8.0
    - System.Reactive.Net
- net8.0-windows10.0.19041
    - System.Reactive.Net
    - System.Reactive.Net.For.WindowsForms
    - System.Reactive.Net.For.WindowsRuntime
    - System.Reactive.Net.For.Wpf
- .NETStandard,Version=2.0
    - System.Reactive.Net
    - System.Threading.Tasks.Extensions

Since we're considering the 'bloat' problem, the significant target here is net8.0-windows10.0.19041. Since the net6.0-windows10.0.19041 target in v6 offered Windows Forms, WPF, and certain Windows Runtime features, it must continue to do so in its new role as a legacy facade. And to be able to define type forwarders to the newly-repackaged UI-framework-specific types, it must specify dependencies on those new packages. And since the new WPF and Windows Forms packages require the Microsoft.Desktop.App framework, that means that System.Reactive also imposes a dependency on that framework.

DisableTransitiveFrameworkReferences

In some cases applications might be able to work around this problem by adding this to their csproj:

```
<PropertyGroup>
  <DisableTransitiveFrameworkReferences>True</DisableTransitiveFrameworkReferences>
</PropertyGroup>
```

This tells the .NET build system that if any of the packages our application depends on state that they require framework reference (e.g. to Microsoft.Desktop.App) it should ignore that.

In fact, you can do this with Rx 6. There are some situations in which applications using self-contained deployment that find themselves deploying an unwanted copy of the desktop UI frameworks can avoid the problem by adding that setting to the project file.

Unfortunately it doesn't always work. (If it did work, we wouldn't need to rethink Rx's packaging.)

If the application doesn't use Rx directly itself—if its dependency on Rx was purely for the benefit of some libraries it happens to use—this workaround will typically be successful. But if the application was also using Rx itself, this tends to go wrong. Specifically, attempting to use certain extension methods (e.g. ObserveOn) causes compiler errors when you set DisableTransitiveFrameworkReferences.

The basic problem here is that Rx's net8.0-windows10.0.19041 target defines all UI-framework-specific APIs, and in some cases this extends the set of overloads available for certain extension methods. For example, the basic net8.0 and netstandard2.0 targets define just two ObserveOn extension method overloads for IObservable<T>: one taking a SynchronizationContext, and one taking an IScheduler. But the net8.0-windows10.0.19041 target defines some 10 overloads, including ones that accept a Windows Forms Control, or a WPF DispatcherObject. This might not seem like it should be a problem—you might think that as long as you only use one of the two overloads that don't use any UI framework types, everything will be fine. Sadly not. If you write, say, obs.ObserveOn(SynchronizationContext.Current), which should resolve to one of the non-UI-framework-specific overloads, the C# compiler is obliged to *consider* all available overloads when deciding which particular one you meant. So it has to look at all of the UI-framework-specific overloads even though you don't mean to use them. And then, because the project disabled transitive framework references, types such as Windows Forms' Control, or WPF's DispatcherObject are unavailable to the compiler, meaning it can't actually work out whether those overloads are applicable. For all the compiler knows, those types might define implicit conversions that could affect the overload resolution process. And since it can't see those types it can't work out which overload to use, and compilation fails.

Legacy facade with type hiding (legacyfacade-refnoui.3)

This is almost identical to the approach desribed in the preceding section. It introduces a new main Rx package System.Reactive.Net with no UI framework dependencies. UI-framework-specific code moves into specialised packages such as System.Reactive.For.Wpf. The System.Reactive package becomes a legacy facade to maintain backwards compatibility, offering the same public API at runtime as in previous versions, using type forwarders to refer to the new homes of these types.

There is one critical difference: in this approach, System.Reactive makes the UI-framework-specific types available only at runtime. Whereas in the preceding approach, System.Reactive offered both build-time and runtime compatibility, in this approach it offers only binary compatibility. The API surface area apparently available at build time does not include any UI-framework-specific features. This in turn enables the package not to have to declare dependencies on any of the UI-framework-specific Rx packages, which in turn also enables it not to impose a dependency on the Microsoft.Desktop.App framework.

It achieves this by providing both runtime and reference assemblies (in the lib and ref subfolder of the NuGet package respectively). The runtime assemblies define the same full public API as previous versions of System.Reactive (using type forwarders to these types' new locations). The reference assemblies omit all the UI-framework-specific types.

Issues in brief

This option, has the follwoing significant weaknesses:

- Applications may find that they have two versions of Rx in scope simultaneously
- Desktop applications that upgrade System.Reactive from v6 to v7 that were in fact using the UI framework support will now get either build or runtime errors, and will need somehow to discover that they also need to add a reference to System.Reactive.For.Wpf/WindowsForms to fix this
- It relies on a trick: reference assemblies defining a *different* API than is on offer at runtime, and 'clever' tricks often cause problems

Packaging Option	**Questionable ref assembly tricks**		**Unwanted Microsoft.Windows Desktop.App**		**Build failures**				**Runtime failures**		
	In legacy facade	**In main package**	**Legacy refs need attention**	**Imposed by System.Reactive**	**Desktop API use requires explicit FX ref where required**	**Rx UI use requires new package ref**	**Multiple Rx versions in scope**	**Other undiagnosed issues**	**Desktop API use requires explicit FX ref where required**	**Rx UI use requires new package ref**	**Other undiagnosed issues**
legacyfacade-refnoui.3	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	No

Desktop dependency no longer imposed

The major benefit that this offers over the preceding solution is that applications using self-contained deployment, and that find themselves with a dependency on System.Reactive (which might be unavoidable because they depend on a library that uses Rx) are now able to avoid the 'bloat' problem (deploying an unwanted copy of Microsoft.Desktop.App). (The 'Imposed by System.Reactive column is green for this option.) They can do this by upgrading to the latest version of System.Reactive.

Mixed versions

A confusing downside of this approach is that if an application has a reference (either direct or transitive) to the latest main Rx assembly, System.Reactive.Net, and also has a transitive reference to an old version of System.Reactive, that reference to the new main Rx package won't automatically upgrade the old one. So the application will simultaneously be using:

- System.Reactive.Net,7.0.0
- System.Reactive,6.0.0

This is represented by the red 'Multiple Rx verions in scope' column. The preceding design option also has this problem. It's an unavoidable consequence of turning System.Reactive into a legacy facade and introducing a new main Rx package.

This situation doesn't necessarily have to be a problem. If an applications doesn't care about the bloat problem (which only affects self-contained deployment) and if it doesn't use Rx directly—if Rx is being used only by other libraries—these two versions can happily coexist in the same process.

If the application *does* use Rx directly, the error messages that arise from this situation are somewhat baffling. That said, we might be able to mitigate that with a code analyzer that detects when this has happened, and tells you what to do.

Discoverability

When the application is using self-contained deployment and finds itself with the bloat problem caused by an indirect reference to an old version of Rx, it probably won't be at all obvious how to fix this. The correct action will be to add a reference to System.Reactive,7.0.0 to their application, but since System.Reactive will have been marked as deprecated, developers could very well think that this can't be the right way to proceed.

This is an example of the kind of confusion that meant some veteran Rx developers really didn't want yet another significant change to Rx's packaging.

Legacy facade with type hiding but desktop framework reference (legacyfacaderefnouiwithfxref)

This variation mainly exists to illustrate the kind of problem where fixing one issue causes new ones that are, for a wide range of scenarios, much worse than the problem being fixed. (That's quite a common problem with the various ways in which people have suggested Rx's long-standing packaging problems might be fixed.)

This is essentially the same as the preceding approach (legacyfacaderefnoui) but it fixes one column: the 'Rx UI use requires new package ref' column under 'Build failures' is green here:

Packaging Option	**Questionable ref assembly tricks**		**Unwanted Microsoft.WindowsDesktop.App**		**Build failures**				**Runtime failures**		
	In legacy facade	**In main package**	**Legacy refs need attention**	**Imposed by System. Reactive**	**Desktop API use requires explicit FX ref where required**	**Rx UI use requires new package ref**	**Multiple Rx versions in scope**	**Other undiagnosed issues**	**Desktop API use requires explicit FX ref where required**	**Rx UI use requires new package ref**	**Other undiagnosed issues**
legacyfacade-refnoui-withfxref.1	Yes	No	NA	Yes	Yes	No	Yes	Yes	No	No	Yes

Scenario addressed here is applications that were in the situation that Rx 6's UI framework support packaging was specifically designed for. If the reason your app targets net8.0-windows10.0.19041 is that you're building a WPF application, then Rx 6's behaviour works great: it makes all the WPF-specific types available to you without additional package references. And it even gives you a dependency on the Microsoft.Desktop.App framework without you having to ask for it.

Of course that last part is, for many potential Rx users, exactly the problem they want us to fix. But for someone writing a WPF application, this is benefit, not a problem.

This design variation illustrates what happens if we take the preceding design option, but try to retain the existing behaviour in this one particular way: the package continues to supply an implicit reference to the Microsoft.Desktop.App package.

(Actually the other reason we have this is that when you're trying to do the preceding packaging, it's quite easy to end up with this one by accident...)

Anyway, the red cell in the 'Imposed by System.Reactive' makes this a non-starter, so we won't be doing this.

System.Reactive remains as main Rx package, with type hiding (nofacade-refnoui.5)

Unlike all the other packing options shown so far, this one does not relegate System.Reactive to being a legacy facade: in this model System.Reactive continues to be the main Rx package. This has some significant advantages. It also causes two problems, one potentially quite serious.

In this model, we use the same type hiding trick as in the legacyfacade-refnoui approach: although System.Reactive provides the full legacy API for runtime backwards compatibility, it includes reference assemblies that omit these, so the API visible at build-time does not offer any UI-framework-specific features. Applications or libraries building against System.Reactive v7.0 or later must use the new System.Reactive.For.* packages to get the UI-framework-specific types they require.

In the summary table, this option seems to present mostly green cells compared to the prevalance of yellow and red for the other options. However, it has one particularly serious issue.

Issues in brief

This design has the following problems:

- Just like legacyfacade-refnoui, this relies on a trick that could cause problems, but this time that trick is in the main Rx package, and not just a compatibility facade
- With this design option we are doomed to continue to provide a uap10.0.18362 target in the main Rx assembly for the foreseeable future (likely at least for a decade), something that the .NET SDK has never supported, and never will

Full disclosure: that second one is a major cause of pain for ongoing Rx.NET development, so it biases anyone who has to work on Rx (e.g., me) against this particular solution. However, as far as I know, this second point doesn't cause any problems for people using Rx. If this were the only issue with this problem, I would reluctantly resign myself to having to continue to battle with the problems caused by the present of old-school UWP code in the main project. It's horrible, but if this is the way to provide the best experience for developers using Rx, then I just have to cope with the pain.

(And no the support that net9.0 added for UWP does not help us in any way at all in the near term. On the contrary, it just adds yet another configuration we have to consider in our test matrix. It does not change the fact that if we simply drop the uap10.0.18362 target completely from System.Reactive, then anyone who still has to maintain applications built for that target would find their build was broken in a way that was impossible to fix without removing Rx, which might mean they'd have to stop using other components that happen to depend on Rx. The good thing about net9.0's addition of UWP is that it does point to a possible future in which nobody has any good reason to remain on the uap10.0.18362 target. A plausible path to deprecation and eventual removal of our uap target does exist, but that's several years into the future, so it doesn't help us today.)

Packaging Option	**Questionable ref assembly tricks**		**Unwanted Microsoft.WindowsDesktop.App**		**Build failures**				**Runtime failures**		
	In legacy facade	**In main package**	**Legacy refs need attention**	**Imposed by System. Reactive**	**Desktop API use requires explicit FX ref where required**	**Rx UI use requires new package ref**	**Multiple Rx versions in scope**	**Other undiagnosed issues**	**Desktop API use requires explicit FX ref where required**	**Rx UI use requires new package ref**	**Other undiagnosed issues**
nofacade-refnoui.5	Yes	Yes	No	No	No	No	No	No	Yes	No	No

The consequences of 'clever' tricks

As far as it's currently possible to tell, this design approach really only has one major shortcoming: it relies on the non-standard trick of making types available at runtime but hiding them in the reference assemblies.

This is arguably an abuse of the reference assembly feature. It's not really what they are for. Reference assemblies are supposed to have the same public API surface area as the runtime assemblies. The tooling Microsoft provides for generating reference assemblies makes that assumption, and we've had to make the Rx.NET build perform some unnatural acts to make this work.

This worries me. The history of addressing Rx packaging problems with clever tricks does not look good. The initial attempt to solve the plug-in problem performed a slightly off-piste trick, and it caused so many problems in scenarios outside the ones it was trying to fix (and those newly-problematic scenarios were much more common than the plug-in problems) that Rx eventually reversed that design decision. The problems we're still wrestling with today around unwanted desktop framework references were a result of some clever tricks that enabled the Great Unification. This worked well at the time, but fairly soon afterwards, developments in the .NET ecosystem meant these choices have gone on to cause problems in the long run.

For that reason, I am very reluctant to introduce a new 'clever' trick into the main Rx package.

For that reason I don't like this nofacade-refnoui approach. Even if nobody today can think of any reason that this will necessarily cause problems, the history of Rx teaches us that this is no guarantee.

I therefore have a bias towards as much normality as possible. (This is also a reason to want to remove all uap10.0.18362 code from the main Rx package.) And for this reason, I prefer the legacyfacade-refnoui approach. Yes, that uses the exact same 'type hiding with reference assemblies' trick but the crucial difference is that it does this only with the legacy compatibilty package, a package we ultimately would like everyone to stop using. The new main package, System.Reactive.Net gets to be normal in this world, something that seems not to be possible if System.Reactive continues to be the main Rx package.

Note that the 'Rx UI use requires new package ref' column under 'Runtime failures' is green. That's different from the legacyfacade-refnoui, where if you really were actually using UI-framework-specific features indirectly through a transitive reference, you'll get a runtime error because those the type forwarders refer to System.Reactive.For.* assemblies, but the System.Reactive NuGet package does not declare dependencies on the packages that contain those assemblies. (It can't, because if it did, the) need to add a package reference

Desktop API use requires explicit FX ref where required

We're able to avoid runtime failures because this isn't purely type forwarders. That does mean there are, once again, two instances of certain types.

Possibilities not yet prototyped

The nofacade-refnoui option uses a technique to avoid runtime failures when you indirectly (via a transitive reference) use UI-framework-specific Rx features but you've not explicitly referenced the relevant packages. It does this by baking in its own copies of the relevant types specifically for use in runtime compatibility scenarios.

In principle we could do the same even when System.Reactive is nominally a facade. An assembly does not need to be purely a facade consisting of nothing by type forwarders: you can have a mixture of forwarders and actual code. So we could make a nofacade-refnoui-hasuitypesatruntime variation, in which System.Reactive contains its own copies of things like DispatcherScheduler and so on. These would be used at runtime by any packages that used these particular features and were compiled against Rx 6. Code compiled against Rx 7 and later would be obliged to use the System.Reactive.For.* packages because these runtime compatibility types would continue not to be visible in the build-time public API of System.Reactive.