Getting TAEF unittests to work with a C++/WinRT XAML Islands application

- **Author**: Mike Griese @zadjii-msft
- **Created on**: 2019-06-06

So you've built a Win32 application that uses XAML Islands to display it's UI with C++/WinRT. How do you go about adding unittests to this application? I'm going to cover the steps that I took to get the Windows Terminal updated to be able to test not only our C++/WinRT components, but also pure c++ classes that were used in the application, and components that used XAML UI elements.

Prerequisites

Make sure you're using at least the 2.0.190605.7 version of the CppWinRT nuget package. Prior to this version, there are some bugs with C++/WinRT's detection of static lib dependencies. You might be able to get your build working with Visual Studio on earlier versions, but not straight from MsBuild.

Also, if you're going to be running your tests in a CI build of some sort, make sure that your tests are running on a machine running at least Windows 18362. If your CI isn't running that version, then this doesn't matter at all.

Furthermore, you may need an updated TAEF package as well. Our CI uses the TAEF VsTest adapter to allow ADO to run TAEF tests in CI. However, there's a bug in the tests adapter that prevents it from running tests in a UAP context. The `10.38.190605002` TAEF is the most recent release at the time of writing, however, that doesn't have the fix necessary. Fortunately, the TAEF team was kind enough to prototype a fix for us, which is the version `10.38.190610001-uapadmin`, which we're using in this repo until an official release with the fix is available.

Move the C++/WinRT implementation to a static lib

By default, most (newly authored) C++/WinRT components are authored as a dll that can be used to activate your types. However, you might have other classes in that binary that you want to be able to test, which aren't winrt types. If the implementation is sitting in a DLL, it'll be hard to write a TAEF unittest dll that can call the pure c++ types you've defined.

The first thing you're going to need to do is move the implementation of your winrt component from a dll to a static lib. Once you have the static lib, we'll be able to link it into the dll you were previously producing, as well as being able to link it into the dll we'll be using to test the types. Once this is complete, your dll project will exist as little more than some extra packaging for your new lib, as all your code will be built by the lib.

To aid in this description, I'll be referring to the projects that we changed. The dll project we changed to a lib was the `TerminalApp` project. From it, we created a new `TerminalAppLib` project, and changed `TerminalApp` to create a dll by linking the lib `TerminalAppLib` produced.

Create the static lib project

We'll start by creating a new static lib project. The easiest way to do this is by copying your existing dll `vcxproj` file into a new file. Make sure to change the `ProjectGuid` and to add the new project to your `.sln` file. Then, change the `ConfigurationType` to `StaticLibrary`. This Lib should be responsible for building all of your headers, `.cpp` files, `.idl`s for your winrt types, and any `.xaml` files you might have.

You'll likely need to place this new file into a separate directory from the existing dll project, as C++/WinRT uses the project directory as the root of the intermediate build tree. Each directory should only have one `.vcxproj` file in it. For the Terminal project, we created a subdirectory `lib/` underneath `TerminalApp/`, and updated the `Include` paths to properly point at the original files. You could alternatively put all the source in one directory, and have separate `dll/` and `lib/` subdirectories from the source that are solely responsible for building their binary.

At this point, you might face some difficulty including the right winmd references, especially from other C++/WinRT dependencies for this project that exist in your solution. I don't know why, but I had a fair amount of difficulty using a `ProjectReference` from a C++/WinRT StaticLibrary to another C++/WinRT project in my solution. If you're referring to any other projects, you'll need to set up a reference to their built `.winmd`'s manually.

As an example, here's how we've added a reference to the `TerminalSettings` project from our `TerminalAppLib` project:

```
<ItemGroup>
  <!-- Manually add references to each of our dependent winmds. Mark them as
  private=false and CopyLocalSatelliteAssemblies=false, so that we don't
  propagate them upwards (which can make referencing this project result in
  duplicate type definitions)-->

  <Reference Include="Microsoft.Terminal.Settings">
    <HintPath>$(OpenConsoleCommonOutDir)\TerminalSettings\Microsoft.Terminal.Settings.winm
    <IsWinMDFile>true</IsWinMDFile>
    <Private>false</Private>
    <CopyLocalSatelliteAssemblies>false</CopyLocalSatelliteAssemblies>
  </Reference>
</ItemGroup>
```

The `HintPath` may be different depending on your project structure - verify

locally the right path to the `.winmd` file you're looking for.

Notably, you'll also need to put a `pch.h` and `pch.cpp` in the new lib's directory, and use them instead of the `pch.h` used by the dll. C++/WinRT will be very angry with you if you try to use a `pch.h` in another directory. Since we're putting all the code into the static lib project, take your existing `pch.h` and move it to the lib project's directory and create an empty `pch.h` in the dll project's directory.

Update the dll project

Now that we have a lib that builds all your code, we can go ahead and tear out most of the dead code from the old dll project. Remove all the source files from the dll's `.vcxproj` file, save for the `pch.h` and `pch.cpp` files. You *may* need to leave the headers for any C++/WinRT types you've authored in this project - I'm not totally sure it's necessary.

Now, to link the static lib we've created. For whatever reason, adding a `ProjectReference` to the static lib doesn't work. So, we'll need to manually link the lib from the lib project. You can do that by adding the lib's output dir to your `AdditionalLibraryDirectories`, and adding the lib to your `AdditionalDependencies`, like so:

```
<ItemDefinitionGroup>
  <Link>
    <!-- Manually link with the TerminalAppLib.lib we've built. -->
    <AdditionalLibraryDirectories>$(OpenConsoleCommonOutDir)\TerminalAppLib;%(AdditionalLi

    <AdditionalDependencies>TerminalAppLib.lib;%(AdditionalDependencies)</AdditionalDepend
  </Link>
</ItemDefinitionGroup>
```

We are NOT adding a reference to the static lib project's .winmd here. As of the 2.0.190605.7 CppWinRT nuget package, this is enough for MsBuild and Visual Studio to be able to determine that the static lib's `.winmd` should be included in this package.

At this point, you might have some mdmerge errors, which complain about duplicate types in one of your dependencies. This might especially happen if one of your dependencies (ex `A.dll`) is also a dependency for one of your *other* dependencies (ex `B.dll`). In this example, your final output project `C.dll` depends on both `A.dll` and `B.dll`, and `B.dll` *also* depends on `A.dll`. If you're seeing this, I recommend adding `Private=false` and `CopyLocalSatelliteAssemblies=false` to your dependent dlls. In this example, add similar code to `B.dll`:

```
<ProjectReference Include="$(SolutionDir)src\cascadia\TerminalSettings\TerminalSettings
  <Private>false</Private>
  <CopyLocalSatelliteAssemblies>false</CopyLocalSatelliteAssemblies>
</ProjectReference>
```

where `TerminalSettings` is your `A.dll`, which is included by both `B` and `C`.

We additionally had an `.exe` project that was including our `TerminalApp` project, and all its `.xbf` and `.pri` files. If you have a similar project aggregating all your resources, you might need to update the paths to point to the new static lib project.

At this point, you should be able to rebuild your solution, and everything should be working just the same as before.

Add TAEF Tests

Now that you have a static library project, you can start building your unittest dll. Start by creating a new directory for your unittest code, and creating a `.vcxproj` for a TAEF unittest dll. For the Terminal solution, we use the TAEF nuget package `Microsoft.Taef`.

Referencing your C++/WinRT static lib

This step is the easiest. Add a `ProjectReference` to your static lib project, and your lib will be linked into your unittest dll.

```
<ProjectReference Include="$(SolutionDir)\src\cascadia\TerminalApp\lib\TerminalAppLib.vc
```

Congratulations, you can now instantiate the pure c++ types you've authored in your static lib. But what if you want to test your C++/WinRT types too?

Using your C++/WinRT types

To be able to instantiate your C++/WinRT types in a TAEF unittest, you'll need to rely on a new feature to Windows in version 1903 which enables unpackaged activation of WinRT types. To do this, we'll need to author a SxS manifest that lists each of our types, and include it in the dll, and also activate it manually from TAEF.

Creating the manifest First, you need to create a manifest file that lists each dll your test depends upon, and each of the types in that dll. For example, here's an excerpt from the Terminal's manifest:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <file name="TerminalSettings.dll" hashalg="SHA1">
    <activatableClass name="Microsoft.Terminal.Settings.KeyChord" threadingModel="both" xmln
    <activatableClass name="Microsoft.Terminal.Settings.TerminalSettings" threadingModel="bo
  </file>
  <file name="TerminalApp.dll" hashalg="SHA1">
    <activatableClass name="TerminalApp.App" threadingModel="both" xmlns="urn:schemas-micros
    <activatableClass name="TerminalApp.AppKeyBindings" threadingModel="both" xmlns="urn:sch
    <activatableClass name="TerminalApp.XamlmetaDataProvider" threadingModel="both" xmlns="u
```

```
  </file>
</assembly>
```

Here we have two dlls that we depend upon, `TerminalSettings.dll` and `TerminalApp.dll`. `TerminalSettings` implements two types, `Microsoft.Terminal.Settings.KeyChord` and `Microsoft.Terminal.Settings.TerminalSettings`.

Linking the manifest to the test dll Now that we have a manifest file, we need to embed it in your unittest dll. This is done with the following properties in your `vcxproj` file:

```
<PropertyGroup>
  <GenerateManifest>true</GenerateManifest>
  <EmbedManifest>true</EmbedManifest>
</PropertyGroup>
<ItemGroup>
  <Manifest Include="TerminalApp.Unit.Tests.manifest" />
</ItemGroup>
```

where `TerminalApp.Unit.Tests.manifest` is the name of your manifest file.

Additionally, you'll need to binplace the manifest *adjacent to your test binary*, so TAEF can find it at runtime. I've done this in the following way, though I'm sure there's a better way:

```
<ItemDefinitionGroup>
  <PostBuildEvent>
    <!-- Manually copy the manifest to our outdir, because the test will need
    to find it adjacent to us. -->
    <Command>
      (xcopy /Y &amp;quot;$(OpenConsoleDir)src\cascadia\ut_app\TerminalApp.Unit.Tests.manifest
    </Command>
  </PostBuildEvent>
</ItemDefinitionGroup>
```

Copying your dependencies Additionally, any dlls that implement any types your test is dependent upon will also need to be in the output directory for the test. Manually copy those DLLs to the tests' output directory too. The updated `PostBuildEvent` looks like this:

```
<ItemDefinitionGroup>
  <PostBuildEvent>
    <Command>
      echo OutDir=$(OutDir)
      (xcopy /Y &amp;quot;$(SolutionDir)src\cascadia\ut_app\TerminalApp.Unit.Tests.manifest&amp;qu

      (xcopy /Y &amp;quot;$(OpenConsoleCommonOutDir)\TerminalConnection\TerminalConnection.dll
      (xcopy /Y &amp;quot;$(OpenConsoleCommonOutDir)\TerminalSettings\TerminalSettings.dll&amp;quo
```

```
      (xcopy /Y &quot;$(OpenConsoleCommonOutDir)\TerminalControl\TerminalControl.dll&quot;
    </Command>
  </PostBuildEvent>
</ItemDefinitionGroup>
```

Again, verify the correct paths to your dependent C++/WinRT dlls, as they may be different than the above

Activating the manifest from TAEF Now that the manifest lives adjacent to your test dll, and all your dependent dlls are also adjacent to the unittest dll, there's only one thing left to do. TAEF will not use your dll's manifest by default, so you'll need to add a property to your test class/method to tell TAEF to do so. You can do this with the following:

```
class SettingsTests
{
    // Use a custom manifest to ensure that we can activate winrt types from
    // our test. This property will tell taef to manually use this as the
    // sxs manifest during this test class. It includes all the C++/WinRT
    // types we've defined, so if your test is crashing for an unknown
    // reason, make sure it's included in that file.
    BEGIN_TEST_CLASS(SettingsTests)
        TEST_CLASS_PROPERTY(L"ActivationContext", L"TerminalApp.Unit.Tests.manifest")
    END_TEST_CLASS()

    // Other Test code here
}
```

Now, if you try to add any test methods that instantiate WinRT types you've authored, they'll work. That is of course, so long as they don't use XAML. If you want to use any XAML types, then you'll have to keep reading.

Using Xaml Types (with XAML Islands)

To be able to instantiate XAML types in your unittest, we'll need to make use of the XAML Hosting API (Xaml Islands). This enables you to use XAML APIs from a Win32 context.

Adding XAML Hosting code First and foremost, you'll need to add the following to your test's `precomp.h`:

```
#include <winrt/Windows.system.h>
#include <winrt/Windows.Foundation.Collections.h>
#include <winrt/Windows.UI.Xaml.Hosting.h>
#include <windows.ui.xaml.hosting.desktopwindowxamlsource.h>
```

If you hit a compile warning that refers to `GetCurrentTime`, you'll probably also need the following, after you've `#include`'d `Windows.h`:

```
#ifdef GetCurrentTime
#undef GetCurrentTime
#endif
```

Then, somewhere in your test code, you'll need to start up Xaml Islands. I've done this in my `TEST_CLASS_SETUP`, so that I only create it once, and re-use it for each method.

```
class TabTests
{
    TEST_CLASS_SETUP(ClassSetup)
    {
        winrt::init_apartment(winrt::apartment_type::single_threaded);
        // Initialize the Xaml Hosting Manager
        _manager = winrt::Windows::UI::Xaml::Hosting::WindowsXamlManager::InitializeForCurre
        _source = winrt::Windows::UI::Xaml::Hosting::DesktopWindowXamlSource{};

        return true;
    }

private:
    winrt::Windows::UI::Xaml::Hosting::WindowsXamlManager _manager{ nullptr };
    winrt::Windows::UI::Xaml::Hosting::DesktopWindowXamlSource _source{ nullptr };
```

Authoring your test's `AppxManifest.xml` This alone however is not enough to get XAML Islands to work. There was a fairly substantial change to the XAML Hosting API around Windows build 18295, so it explicitly requires that you have your executable's manifest set `maxversiontested` to higher than that version. However, because TAEF's `te.exe` is not so manifested, we can't just use our SxS manifest from before to set that version. Instead, you'll need to make TAEF run your test binary in a packaged content, with our own appxmanifest.

To do this, we'll need to author an `Appxmanifest.xml` to use with the test, and associate that manifest with the test.

Here's the AppxManifest we're using:

```
<?xml version="1.0" encoding="utf-8"?>
<Package xmlns:rescap="http://schemas.microsoft.com/appx/manifest/foundation/windows10/restr

  <Identity Name="TerminalApp.Unit.Tests.Package"
          ProcessorArchitecture="neutral"
          Publisher="CN=Microsoft Corporation, O=Microsoft Corporation, L=Redmond, S=Washing
          Version="1.0.0.0"
          ResourceId="en-us" />
  <Properties>
    <DisplayName>TerminalApp.Unit.Tests.Package Host Process</DisplayName>
```

```
    <PublisherDisplayName>Microsoft Corp.</PublisherDisplayName>
    <Logo>taef.png</Logo>
    <Description>TAEF Packaged Cwa FullTrust Application Host Process</Description>
  </Properties>

  <Dependencies>
    <TargetDeviceFamily Name="Windows.Universal" MinVersion="10.0.18362.0" MaxVersionTested=
    <PackageDependency Name="Microsoft.VCLibs.140.00.Debug" MinVersion="14.0.27023.1" Publis
    <PackageDependency Name="Microsoft.VCLibs.140.00.Debug.UWPDesktop" MinVersion="14.0.2702
  </Dependencies>

  <Resources>
    <Resource Language="en-us" />
  </Resources>

  <Applications>
    <Application Id="TE.ProcessHost" Executable="TE.ProcessHost.exe" EntryPoint="Windows.Ful
      <uap:VisualElements DisplayName="TAEF Packaged Cwa FullTrust Application Host Process'
        <uap:SplashScreen Image="taef.png" />
      </uap:VisualElements>
    </Application>
  </Applications>

  <Capabilities>
    <rescap:Capability Name="runFullTrust"/>
  </Capabilities>

  <Extensions>
    <Extension Category="windows.activatableClass.inProcessServer">
      <InProcessServer>
        <Path>TerminalSettings.dll</Path>
        <ActivatableClass ActivatableClassId="Microsoft.Terminal.Settings.TerminalSettings"
        <ActivatableClass ActivatableClassId="Microsoft.Terminal.Settings.KeyChord" Threadin
      </InProcessServer>
    </Extension>
    <!-- More extensions here -->
  </Extensions>
</Package>
```

Change the `Identity.Name` and `Properties.DisplayName` to be more appropriate for your test, as well as other properties if you feel the need. TAEF will deploy the test package and remove it from your machine during testing, so it doesn't terribly matter what these values are.

MAKE SURE that `MaxVersionTested` is higher than `10.0.18295.0`. If it isn't, XAML islands will still prevent you from activating it.

UNDER NO CIRCUMSTANCE should you change the `<Application Id="TE.ProcessHost" Executable="TE.ProcessHost.exe" EntryPoint="Windows.FullTrustApplicatio` line. This is how TAEF activates the TAEF host for your test binary. You might get a warning about `TE.ProcessHost.exe` being deprecated in favor of `TE.ProcessHost.UAP.exe`, but I haven't had success with the UAP version.

Lower in the file, you'll see the `Extensions` block. In here you'll put each of the winrt dependencies that your test needs, much like we did for the previous manifest. Note that the syntax is *not* exactly the same as the SxS manifest.

Copy the AppxManifest to your `$(OutDir)` Again, we'll need to copy this appxmanifest adjacent to the test binary so we can load it from the test. We'll do this similar to how we did the SxS manifest before. The complete `PostBuildEvent` now looks like this:

```
<ItemDefinitionGroup>
  <PostBuildEvent>
    <Command>
      (xcopy /Y &quot;$(SolutionDir)src\cascadia\ut_app\TerminalApp.Unit.Tests.manifest&qu

      (xcopy /Y &quot;$(SolutionDir)src\cascadia\ut_app\TerminalApp.Unit.Tests.AppxManifes

      (xcopy /Y &quot;$(OpenConsoleCommonOutDir)\TerminalConnection\TerminalConnection.dll
      (xcopy /Y &quot;$(OpenConsoleCommonOutDir)\TerminalSettings\TerminalSettings.dll&quo
      (xcopy /Y &quot;$(OpenConsoleCommonOutDir)\TerminalControl\TerminalControl.dll&quot;
    </Command>
  </PostBuildEvent>
</ItemDefinitionGroup>
```

The new line here is the line referencing `TerminalApp.Unit.Tests.AppxManifest.xml`. You can only have one `PostBuildEvent` per project, so don't go re-defining it for each additional step - MsBuild will only use the last one. Again, this is probably not the best way of copying these files over, but it works.

Use the AppxManifest in the test code Now that we have the AppxManifest being binplaced next to our test, we can finally reference it in the test. Instead of using the `ActivationContext` from before, we'll use two new properties to tell TAEF to run this test as a package, and to use our manifest as the AppxManifest for the package.

```
BEGIN_TEST_CLASS(TabTests)
    TEST_CLASS_PROPERTY(L"RunAs", L"UAP")
    TEST_CLASS_PROPERTY(L"UAP:AppXManifest", L"TerminalApp.Unit.Tests.AppxManifest.xml")
END_TEST_CLASS()
```

The complete Xaml Hosting test now looks like this:

```
class TabTests
```

```
{
    BEGIN_TEST_CLASS(TabTests)
        TEST_CLASS_PROPERTY(L"RunAs", L"UAP")
        TEST_CLASS_PROPERTY(L"UAP:AppXManifest", L"TerminalApp.Unit.Tests.AppxManifest.x
    END_TEST_CLASS()

    TEST_METHOD(TryCreateXamlObjects);

    TEST_CLASS_SETUP(ClassSetup)
    {
        winrt::init_apartment(winrt::apartment_type::single_threaded);
        // Initialize the Xaml Hosting Manager
        _manager = winrt::Windows::UI::Xaml::Hosting::WindowsXamlManager::InitializeFor(
        _source = winrt::Windows::UI::Xaml::Hosting::DesktopWindowXamlSource{};

        return true;
    }

private:
    winrt::Windows::UI::Xaml::Hosting::WindowsXamlManager _manager{ nullptr };
    winrt::Windows::UI::Xaml::Hosting::DesktopWindowXamlSource _source{ nullptr };
};
void TabTests::TryCreateXamlObjects(){ ... }
```

Congratulations, you can now use XAML types from your unittest.

Using types from `Microsoft.UI.Xaml`

Let's say you're extra crazy and you're using the `Microsoft.UI.Xaml` nuget package. If you've followed all the steps above exactly, you're probably already fine! You've already put the types in your appxmanifest (there are a lot of them). You should be able to call the `Microsoft.UI.Xaml` types without any problems.

This is because of a few key lines we already put in the appxmanifest:

```
<Dependencies>
  <TargetDeviceFamily Name="Windows.Universal" MinVersion="10.0.18362.0" MaxVersionTested=
  <PackageDependency Name="Microsoft.VCLibs.140.00.Debug" MinVersion="14.0.27023.1" Publis
  <PackageDependency Name="Microsoft.VCLibs.140.00.Debug.UWPDesktop" MinVersion="14.0.270
</Dependencies>
```

Without these `PackageDependency` entries for the VCLibs, Microsoft.UI.Xaml.dll will not be able to load.