FIREGIANT DOCS

WiX v5 for WiX v4 users

WiX v5 is **highly** compatible with WiX v4. WiX v5 continues in the traditions of WiX v4 and is available as both a .NET tool and an MSBuild SDK. The WiX v5 language uses the same XML namespace as WiX v4 and — with a couple of exceptions — is backward compatible with the WiX v4 language. That means that you don't need to translate your WiX v4 projects to use WiX v5.

WiX v5 language changes

Virtual and overridable symbols

WiX has supported letting setup developers override the defaults provided by WiX or its extensions for things like when custom actions are scheduled. (Technically, overridability was available for everyone everywhere but the canonical example is overriding scheduling for custom actions in WiX extensions, so let's go with that.) It worked by letting you specify that something was `Overridable="yes"` so that *your* version took precedence over the overridable one. For example, here's how the `CloseApplications` custom action is defined in `WixToolset.Util.wixext` in WiX v4:

```xml
<InstallExecuteSequence>
    <Custom
        Action="$(var.Prefix)CloseApplications$(var.Suffix)"
        Before="InstallFiles"
        Overridable="yes"
        Condition="VersionNT &gt; 400" />
</InstallExecuteSequence>
```

To reschedule the custom action, you'd use the following:

```xml
<InstallExecuteSequence>
    <Custom
        Action="Wix4CloseApplications_$(sys.BUILDARCHSHORT)"
        After="InstallInitialize" />
</InstallExecuteSequence>
```

WiX v5 introduces the concept of `virtual` and `override` access modifiers for symbol identifiers, which are very similar to the same keywords you find in languages like C# and C++:

- `virtual` declares that the identifier can be overridden.
- `override` declares that the identifier overrides the corresponding `virtual` identifier.

So now, WiX extensions define custom action scheduling with the `virtual` access modifier:

```
<InstallExecuteSequence>
    <Custom
        Action="virtual $(var.Prefix)CloseApplications$(var.Suffix)"
        Before="InstallFiles"
        Condition="VersionNT &gt; 400" />
</InstallExecuteSequence>
```

And to reschedule it, use the `override` access modifier to override the scheduling provided by the `virtual` symbol:

```
<InstallExecuteSequence>
    <Custom
        Action="override Wix4CloseApplications_$(sys.BUILDARCHSHORT)"
        After="InstallInitialize" />
</InstallExecuteSequence>
```

For more information, see the *Virtual Symbols* WIP and the associated pull request.

New WiX v5 language features

File harvesting

An all-too-common question on GitHub discussions and Stack Overflow is of the form "how do I install all the files in a directory?" In WiX v4 and prior, the answer was typically "use Heat and some arcane XSLT." In WiX v5, use `Files`.

`Files` takes wildcarded paths to include *and exclude* files, traverses the specified directories, and generates components and files for each file.

Combined with other features in this list, you can now build a package with potentially thousands of files with some impressively compact WiX authoring:

```
<Wix xmlns="http://wixtoolset.org/schemas/v4/wxs">
    <Package Name="MyProduct" Version="1.0.0.0" Manufacturer="Example Corporation"
UpgradeCode="B0B15C00-1DC4-0374-A1D1-E902240936D5">
        <Files Include="path\to\files\**" />
    </Package>
</Wix>
```

With exclusions, you can exclude files that require special handling, like services:

```
<Wix xmlns="http://wixtoolset.org/schemas/v4/wxs">
    <Package Name="MyService" ...>
        <Files Include="!(bindpath.bin)**">
            <!--
            Don't harvest the service because it needs manual authoring (below).
            -->
            <Exclude Files="!(bindpath.bin)foo.exe" />
        </Files>

        <!--
        This file is a service and therefore needs lovingly hand-crafted authoring.
        -->
        <Component>
            <File Source="!(bindpath.bin)foo.exe" />
            <ServiceInstall ... />
            <ServiceControl ... />
        <Component>
    </Package>
</Wix>
```

For more information, see the `Files` WIP, the associated pull request, and the `Files` element schema documentation.

Naked files

For a little while now, WiX has supported simplified authoring for the simple scenario of one file in a component:

```
<Component>
    <File Source="foo.exe" />
</Component>
```

But add a few dozen of them and you start to wonder about the need for the mostly-empty `Component` elements. Wonder no longer. WiX v5 adds support for so-called "naked" files, which are files without the XML overhead of enclosing `Component` elements. Wherever `Component` elements can appear, so can `File` elements. In the compiler, WiX conjures appropriate components for each file. Simple authoring is now simpler.

```
<ComponentGroup Id="Files" Directory="MyFolder" Subdirectory="bin">
  <File Source="foo.exe" />
  <File Source="bar.dll" />
  <File Source="baz.db" />
</ComponentGroup>
```

For more information, see the *Naked File* WIP and the associated pull request.

Default major upgrades

Welcome to the first of three "provide reasonable defaults so setup developers don't need to specify boring stuff over and over" features.

Authoring major upgrades has been straightforward since 2010 but, like with naked files, it sometimes feels silly to have to author unchanging content. But as I said back then:

> Downgrades are blocked by default, which requires you to specify a message for the launch condition message.

However, part of the impetus behind `virtual` and `override` access modifiers was the idea that WiX could now include a *WiX Standard Library*, which coincidentally could include a set of default localization strings. That lets us address the need for the "downgrade blocked" message — well, at least for speakers of US English.

So now in WiX v5, if your package doesn't have a major upgrade (via `MajorUpgrade` or old-school `Upgrade` elements), WiX will give you one for free. It uses a US English string, so if you need another language, override the localization string with a localization file:

```
<?xml version="1.0" encoding="utf-8"?>
<WixLocalization xmlns="http://wixtoolset.org/schemas/v4/wxl" Culture="en-US">
    <String Id="WixDowngradePreventedMessage" Value="[ProductName] does not support
downgrading." />
</WixLocalization>
```

For more information, see the *Default major upgrade behavior and localized error message* WIP and the associated pull request.

Default installation folder

Welcome to the second "provide reasonable defaults so setup developers don't need to specify boring stuff over and over" feature.

WiX v4 eliminated a lot of the verbosity required to author directories in a WiX project — eliminating `TARGETDIR`, adding `StandardDirectory` elements and the awesome `Subdirectory` attribute. WiX v5 takes advantage of that and adds one more: If you reference a directory with an id of `INSTALLFOLDER` but don't define one, WiX gives you one. That default `INSTALLFOLDER` is the equivalent of the following WiX authoring:

```
<StandardDirectory Id="ProgramFiles6432Folder">
    <Directory
        Id="INSTALLFOLDER"
        Name="!(bind.Property.Manufacturer) !(bind.Property.ProductName)"
    />
</StandardDirectory>
```

The directory gets its name from the `Package/@Manufacturer` and `Package/@Name` attribute values.

And don't worry — if you already have such an `INSTALLFOLDER` or never reference a directory with that id, WiX respects your beliefs and won't try to force its own `INSTALLFOLDER` on you or your package.

For more information, see the *Default root directory* WIP and the associated pull request.

Default feature

Welcome to the final “provide reasonable defaults so setup developers don’t need to specify boring stuff over and over” feature.

Fancy, complicated feature trees are passé but MSI requires at least one feature. So if you don’t need multiple features, let’s let WiX create one for you and automatically assign components to it.

Meet the default feature feature.

Again, if you have a set of `Feature` elements, WiX lets them be. This feature kicks in *only* if you haven’t authored any features in your package.

For more information, see the *Default feature* WIP and the associated pull request.

Burn

- `ArpEntry` supports the `AdditionalUninstallArguments` attribute to add arguments to the uninstall command line and `UseUninstallString` to tell Burn to use the `UninstallString` value instead of the default `QuietUninstallString`. Thanks to @nirbar for the pull request.
- Bootstrapper applications are now separate processes rather than hosted by the Burn engine, to increase reliability and security. Being out-of-process also increases compatibility, as Burn no longer needs special support to host .NET or any other language/runtime, for that matter. Want to write a BA in COBOL? You do you. See more information about out-of-proc BAs.

Extension changes

- WixToolset.DifxApp.wixext was deprecated in WiX v4 and was removed in WiX v5. (Microsoft deprecated the underlying DifxApp several years ago.)
- WixToolset.Firewall.wixext now supports the capabilities of the modern Windows Firewall. See the documentation for all the new goodness. Thanks to @chrisbednarski for all the work.
- WixToolset.Netfx.wixext’s DotNetCompatibilityCheck now sets the specified property to 13 when the requested platform is not compatible with the platform the installer is running on. Thanks to @apacker1 for the pull request.

- WixToolset.Util.wixext now has the following `_NODOMAIN` properties from WixQueryOsWellKnownSID: `WIX_ACCOUNT_ADMINISTRATORS_NODOMAIN`, `WIX_ACCOUNT_GUESTS_NODOMAIN`, `WIX_ACCOUNT_LOCALSERVICE_NODOMAIN`, `WIX_ACCOUNT_LOCALSYSTEM_NODOMAIN`, `WIX_ACCOUNT_NETWORKSERVICE_NODOMAIN`, and `WIX_ACCOUNT_USERS_NODOMAIN`. Thanks to @mistoll for the pull request.

See Out-of-process bootstrapper applications for additional extension changes related to building bundles.