The Files Inside

In the next step, we have to specify the media we want to install from. In the days of CDs and DVDs, we'll hardly need installation files spanning media but the possibility is there (if you need to use it, you can later refer to the individual disks using the media `Id` supplied here. `DiskPrompt` can contain any textual description of the various units of installation media that makes it possible for the user to determine which one to insert, Windows Installer will use this description to prompt for it):

```
<Media Id='1' Cabinet='Sample.cab' EmbedCab='yes
' DiskPrompt='CD-ROM #1' />
<Property Id='DiskPrompt' Value="Acme's Foobar
1.0 Installation [1]" />
```

Using the `EmbedCab` attribute, we can decide whether we want the cabinet (archive) of our installation files to become part of the .msi package file itself or to remain separate. Embedding is the usual decision for the final installation package (thus resulting in a single, self-contained file for download or shipment on media). If neither `Cabinet` nor `EmbedCab` is specified, the source files will be left untouched: they can then be copied directly on the distribution media, together with the installer .msi file.

As we have stressed in the introduction, Windows Installer moved from the earlier programmatic approach to a declarative, descriptive one: we describe the hierarchical structure of our source folder structure using hierarchically nested XML structures, and expect the installer to *recreate* this structure during installation on the user machine. Windows Installer requires us to start with an outermost folder, the root destination folder for the whole installation. It has a predefined identifier of `TARGETDIR` and it will be set to our root directory that contains the source cabinet file or the source file tree of

About FireGiant

Now you can install with confidence. FireGiant arms you with accurate support from the creators of WiX.

Because there are no questions we can't answer or problems we can't solve, you can unleash the full functionality of WiX without a doubt.

Get Supported › (/services/support-programs the installation package, which also have a predefined name: `SourceDir`. This provides the basic link between where to install *from* and where to install *to*:

```
<Directory Id='TARGETDIR' Name='SourceDir'>
```

Inside this root folder, we go on with our actual structure. According to established guidelines, installed files go into predetermined locations. For instance, applications should go under `\Program Files\Company\Product`. Shortcuts, desktop icons, user preferences, and so forth all have their own predefined target locations. For our convenience, the installer environment provides predefined names (https://msdn.microsoft.com/en-us/library/aa372057.aspx) for all of them, allowing us to refer to them very easily. This also frees us from localization issues because these folders might very well have different, localized names in non-English Windows versions. In our current example, we'll use three of these names: `ProgramFilesFolder`, `ProgramMenuFolder`, and `DesktopFolder`. Note that these predefined names refer to full paths: even if the Desktop folder is several folders deep in `C:\Users\User\Desktop`, a single `Directory` tag is all it takes to refer to it. With our own nested folders, we have to specify each level separately:

```
<Directory Id='ProgramFilesFolder' Name='PFiles'
>
    <Directory Id='Acme' Name='Acme'>
        <Directory Id='INSTALLDIR' Name='Foobar
1.0'>
```

Note that for each element (and this will be the case throughout the use of WiX) we have to provide an `Id` identifier. Most of the time, these identifiers must be unique because we will cross-reference them all across the WiX source file, so make sure you come up with a naming scheme that makes it easy to follow. In some cases (like `ProgramFilesFolder`) we can use predefined names. In other cases, we use property names (roughly equivalent to a string variable), like `INSTALLDIR`. We will later refer to this property name again.

The notion of *components* and the rules governing their use are probably the most important concepts of the Windows Installer technology and failing to observe these rules is the primary reason for failing setups. Thus, it is very important for us to get a good understanding of components before we can go on with our first sample.

The component is the atomic unit of things to be installed. It consists of resources---files, registry keys, shortcuts, or anything else---that should always be installed as a single unit. Installing a component should never influence other components, removing one should never damage another component or leave any orphaned resource on the target machine. As a consequence, components cannot share files: the same file going to the same location must never be included in more than one component.

It's not at the component level that you have to think about what files make up your product. If the product requires an EXE, three DLLs and a couple of data files, this doesn't mean they have to go into a single component, quite the opposite. Later on, we will decide at a higher level of the hierarchy what components belong together to form a standalone part of your product.

Again, a component should only contain items that belong together so strongly that they always need to be installed or removed together. If this means a single file, then your components will contain a single file each. This is not only normal but exactly what you're expected to do. Don't be afraid, Windows Installer can efficiently handle thousands of components or more, if needed.

So, we have a component consisting of three items, a file and two shortcuts pointing to it. A component has to have its own `Id` identifier as well as its own, unique GUID (the WiX compiler and linker will warn you if you happen to reuse any of these two). This is very important---these GUIDs are the only means for Windows Installer to keep track of the various components. Breaking the component rules will have dire consequences: resources can be left orphaned on the machine during removal, a shared resource might be erroneously removed while another application still needs it, reinstallation of an existing product might fail to restore the functionality, installing the new version of an application can break the previous one.

```
<Component Id='MainExecutable' Guid='YOURGUID-83
F1-4F22-985B-FDB3C8ABD471'>
```

A file is specified by its name. Apart from the actual names, you can decorate the file with several other attributes. `Vital` , when set to *no*, tells the installer that installing this file is not of vital importance. Normally, if installing any file fails for any reason, the installation will be aborted, the user will not be allowed to ignore the problem. Other attributes include `ReadOnly` , `Hidden` , `System` , all making the file to have the appropriate attribute set when installed.

Each component needs a key path. This is the item Windows Installer can later check to see whether the component is actually installed. Although this doesn't seem very important right now when we only learn to install it in the first place, it is important to specify such a key path for every component we use in order to support the uninstallation and repair functionality of the Installer. Besides, the compiler will complain if we don't specify one...

```
<File Id='FoobarEXE' Name='FoobarAppl10.exe' DiskId='1' Source='FoobarAppl10.exe' KeyPath='yes'>
```

Shortcuts also have names and but also provide other important items like working folder and icon specifications. Note the difference between `Directory` (where the shortcut will be placed such as a Start menu or the desktop) and the `WorkingDirectory` (the place the shortcut points to). The second is optional; if omitted, it will default, as expected, to the folder the parent file will be installed into. The `Icon` attribute will allow us to specify the `Id` of an `Icon` tag specified somewhere else in the source rather than the actual filename (even if the .exe extension seems to suggest otherwise, the identifier has to have the same extension as the actual file it will refer to). You can observe that we already reused the `INSTALLDIR` property and, as expected, it will reference to the folder we're installing into, `Program Files\Acme\Foobar 1.0.` Description of other folders might come later in the source code.

Shortcuts can be non-advertised (a simple link pointing to the file in the shortcut's Properties dialog) or advertised (with the link greyed out). This second form lets Windows Installer repair the installation by replacing any missing file the shortcut points to.

```
        <Shortcut Id="startmenuFoobar10" Directo
ry="ProgramMenuDir" Name="Foobar 1.0"
            WorkingDirectory='INSTALLDIR' Icon="
Foobar10.exe" IconIndex="0" Advertise="yes" />
        <Shortcut Id="desktopFoobar10" Directory
="DesktopFolder" Name="Foobar 1.0"
            WorkingDirectory='INSTALLDIR' Icon="
Foobar10.exe" IconIndex="0" Advertise="yes" />
    </File>
</Component>
```

Here come two other components, with their unique `Id` and `Guid`:

```
<Component Id='HelperLibrary' Guid='YOURGUID-6BE
3-460D-A14F-75658D16550B'>
    <File Id='HelperDLL' Name='Helper.dll' DiskI
d='1' Source='Helper.dll' KeyPath='yes' />
</Component>

<Component Id='Manual' Guid='YOURGUID-574D-4A9A-
A266-5B5EC2C022A4'>
    <File Id='Manual' Name='Manual.pdf' DiskId='
1' Source='Manual.pdf' KeyPath='yes'>
        <Shortcut Id='startmenuManual' Directory
='ProgramMenuDir' Name='Instruction Manual' Adve
rtise='yes' />
    </File>
</Component>
```

As you might expect, for an application with hundreds or even thousands of files, this will mean hundreds or thousands of components. Yes, this is normal, this is the expected way to do it. Don't be afraid, there will be no performance problems, the Windows Installer is prepared to handle this all right.

Typing all those hundreds or thousands of components into the WiX source file presents another challenge, of course. The toolset has a small utility (/docs/wix/v3/tutorial/com-expression-syntax-miscellanea/components-of-a-different-color) that can help with this (more about it later) but the real solution is a conceptual change. Stop considering the setup program as a separate application that has to be written in a rush when the main application is already finished. As the WiX source files and the toolset itself can be integrated into your development environment easily, you should keep them in sync all the time. As soon as you start working on a new module or add a new registry reference to your program, modify the corresponding WiX source file at the same time. This way, the setup will be finished together with the application itself and there will be no need to extract all the file and other pieces of information required for the installation later. As the WiX project can be modularized (more about this later), this approach works just as well if you have a large team working on the application rather than a single developer.

And now, the closing tags for the directory elements---one less than what we started with because we're not yet finished. Remaining inside the first, `TARGETDIR` , directory tag, we specify two more full path folders, using predefined names of the Installer: one for our Start Menu shortcuts and another one for our Desktop icons. Only then will the outermost `Directory` tag be closed.

```
        </Directory>
    </Directory>
</Directory>
```

As we need to remove the program folder when the product is uninstalled, we need to create a fourth component as well. The `RemoveFolder` tag will describe our intention; the `On` attribute will determine when the folder will be removed (possible values are *install*, *uninstall*, and *both*). As already mentioned, all components must have their own key path. In this case, this will be an extra `RegistryValue` tag. This tag is beyond the scope of this first lesson, we will return to it later. Putting the `KeyPath` attribute on the component or the folder might work, too, but that would result in a linker warning. So, please, accept this solution for now as something temporarily unexplained for the sake of avoiding any messages from the compiler and linker.

```
    <Directory Id="ProgramMenuFolder" Name="Programs">
        <Directory Id="ProgramMenuDir" Name="Foobar 1.0">
            <Component Id="ProgramMenuDir" Guid="YOURGUID-7E98-44CE-B049-C477CC0A2B00">
                <RemoveFolder Id='ProgramMenuDir' On='uninstall' />
                <RegistryValue Root='HKCU' Key='Software\[Manufacturer]\[ProductName]' Type='string' Value='' KeyPath='yes' />
            </Component>
        </Directory>
    </Directory>

    <Directory Id="DesktopFolder" Name="Desktop" />
</Directory>
```

Note the `Id` identifiers we used to identify these two folders, these are the names we used in our shortcut's `Directory` attribute earlier to make the connection between the location of the shortcut and the actual folder. Last but not least, we tell the installer which *features* we would like to install. Features are separated parts of the application that we offer the user to decide whether to install or not. The details will of course depend on your particular software package but a usual scheme might look like this:

- the basic executables necessary for the functioning of the program

- documentation, help files
- tutorials, sample files
- related utilities

In our first sample, we won't have such features. First, because we could hardly divide the three files we plan to install into various features. Second, to do so, we would also need a user interface that the user can use to turn these features on or off. We'll return to that in the next lesson but for now, we will have one feature (because we have to have at least one). We refer back to the components we would like to install with this feature using their `Id` identifiers:

```
<Feature Id='Complete' Level='1'>
    <ComponentRef Id='MainExecutable' />
    <ComponentRef Id='HelperLibrary' />
    <ComponentRef Id='Manual' />
    <ComponentRef Id='ProgramMenuDir' />
</Feature>
```

We also have to include the icon we want to use in the shortcuts. Note that the `Id` identifier has to carry the same extension as the target file, in this case, *.exe*:

```
<Icon Id="Foobar10.exe" SourceFile="FoobarAppl1
0.exe" />
```

This will store the source file separately in the final installation package (so, if you refer to your main executable, you will end up with two copies). If the size of the file is large enough to cause concern, create a small *.exe* or *.ico* file containing nothing but the icons.

```
<Shortcut Id="desktopFoobar10" Directory="Deskto
pFolder" Name="Foobar 1.0" WorkingDirectory='INS
TALLDIR' Icon="Foobar10.ico" IconIndex="0" />
...
<Icon Id="Foobar10.ico" SourceFile="FoobarAppl1
0.ico" />
```

All there's left to do is to provide the closing tags for the two tags we still have open:

```
    </Product>
</Wix>
```

To summarize: first, we provided the description of our application, both the human readable texts and the required GUIDs. Second, we specified the media we want to install from. Next, we specified the folder structure of our files to be installed. These files, together with their accompanying resources, all went into the appropriate components. And finally, we described the features we would like to install, referring back to the individual components.

‹ The Software Package (/docs/wix/v3/tutorial/getting-started/the-software-package/)
up (/docs/wix/v3/tutorial/getting-started/)
Putting it to Use › (/docs/wix/v3/tutorial/getting-started/putting-it-to-use/)

CONTACT US (/contact-us/) | BLOG (/blog/)

PRODUCTS (/products/)

- HeatWave (/docs/heatwave/)
- HeatWave Build Tools (/products/heatwave-build-tools/)
- FireGiant WiX v3 (/products/fg-wix/)

SERVICES (/services/)

- Support Programs (/services/support-programs/)
- Code Review (/services/code-review/)
- Phone Consultation (/services/phone-consultation/)
- Custom Development (/services/custom-development/)

DOCUMENTATION (/docs/)

- HeatWave (/docs/heatwave/)
- WiX v4 Tutorial (/docs/wix/tutorial/)

- WiX v3 Tutorial (/docs/wix/v3/tutorial/)
- FireGiant WiX v3 (/docs/fg-wix/)

- About Us (/about-firegiant/)
- Why WiX (/why-wix/)
- Leadership Team (/about-firegiant/leadership-team/)
- Privacy Policy (/about-firegiant/privacy-policy/)