WiX Cookbook

WiX is a dialect of XML used to make installers for Windows. Its declarative style avoids the complexity and limitations of procedural code, providing you with everything you need to package up an entire application into a single MSI file.

This book gives you a good overview of WiX's capabilities to develop your own installer packages with functionalities beyond those available in Windows Installer. In the recipes of this book, you will see ways in which WiX can cut down on your installation time and help you streamline your deployment processes. You will see how to make customized installer UIs, write custom actions, create shortcuts, and also set your application as the default for a file type.

What this book will do for you...

- Install websites, application pools, and virtual directories in IIS
- Add a wizard to your installer and customize it with your own dialogs
- Set up new user accounts, add security groups, and secure files and folders on installation
- Prevent deployment to unsupported systems by using launch conditions
- Update XML files during installation
- Configure shortcuts to your application
- Include prerequisites with a bootstrapper
- Create custom actions using C#
- Add environment variables to the system

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

Over 60 hands-on recipes packed with tips and tricks to boost your Windows installations

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 “Organizing and Building WiX Projects”
- A synopsis of the book’s content
- More information on WiX Cookbook

About the Author

Nick Ramirez is a software developer living in Columbus, Ohio. As a believer that deployment shouldn't be terrifying, he has become a big fan of technologies such as WiX. His other related interests include build automation, software architecture, and playing Minecraft. Nick lives with his wife and two cats.

I would like to thank the editors at Packt Publishing for making the process of writing so much easier. They are superb at keeping things on track and well organized. A big thanks goes to all those involved in the WiX community for continuing to pour vitality into the project and keeping the discussions going. There are many tireless volunteers who are always there to answer a question or start a discussion. Those developers who contribute source code to the WiX project deserve an extra bit of thanks for the hard work that they do. Because of them, WiX continues to be the best deployment tool in the market.
WiX Cookbook

The race to get software developed and deployed is reaching a dizzying pace. As companies accelerate their target release dates, topics such as continuous deployment, process automation, and DevOps take center stage. The need for a reliable and reproducible method of installing software is clear.

In this regard, it's easy to see where a tool like WiX can shine. Its XML syntax makes it easy to read, edit, and store in source control. It creates a package that can either be installed by nontechnical consumers with the click of a mouse or by a remote system administrator who needs to work from the command line. Probably its greatest strength is that it takes tasks that used to be performed by human beings and makes them automated. Think about installing databases, creating websites, and editing configuration files. These are all jobs that WiX can do at the time of installation. More than that, it can react to its environment, installing one file or another depending on the operating system, pulling down missing resources from the Web, and preventing installation on unsupported machines.

In this book, we'll cover many of these sorts of topics. Each chapter centers on a common theme, but taken together, they'll provide a good general overview of WiX's capabilities. You'll quickly see ways in which WiX can cut down on the time you might spend to manually set up your software now. For those who have only limited experience with WiX, the first few chapters should give you the background you need to be up and running. Those seeking to streamline their deployment processes will find the chapters on Windows user and group creation, file permissions, websites, and databases interesting. It's an exciting but challenging time for installation, and I hope you'll find that the recipes presented here give you the running start you'll need.

What This Book Covers

Chapter 1, Organizing and Building WiX Projects, helps you to build your WiX projects right, whether that means from Visual Studio, the command line, or on a build server with automation. We'll also see how to reference the output of other projects that will be included in the installer and how to separate WiX markup into libraries.

Chapter 2, Installing Files and Directories, covers creating folders and files on the end user's computer. We'll also see how to get heat.exe to generate this sort of markup for us and how to make decisions during installation about which files to create.

Chapter 3, File and Folder Permissions, introduces recipes to set the permissions on installed files and folders. We'll also see how to create file shares and choose a default program to use when opening a custom file type.
Chapter 4, Shortcuts, digs into all things related to shortcuts. We'll see how to put an icon on a shortcut, point a shortcut to a folder, as well as place a shortcut in the most popular places: the Start menu and the desktop. We'll also touch on how to create an advertised shortcut that will give users the ability to install features on demand.

Chapter 5, Editing XML Files during Installation, will come in handy especially when altering XML-based configuration files for your software. We'll add XML elements, remove them, set attributes, and insert inner text as part of our installation.

Chapter 6, Custom Actions, focuses on making well-behaved custom actions to extend the WiX functionality. We'll see how to protect the privacy of data sent to custom actions and open console windows without showing them, and rollback failures.

Chapter 7, Installing Wizards, puts a face on our installer by plugging in one of the wizards that comes with the WiX toolset. We'll then see how to customize it with our own images, license, and dialogs.

Chapter 8, Users and Groups, discusses adding users and groups to the target computer. We'll also see how to marry the two by adding users to groups and grant users the log on as a service security setting.

Chapter 9, Handling Prerequisites, includes tactics to only install our software to systems that can support it. We'll examine ways to stop an installation if prerequisites aren't met, install .NET Framework if it's missing, and download resources from the web during installation.

Chapter 10, Installing Websites, explores adding sites to IIS. We'll cover application pools, websites, virtual directories, and web applications. We'll then top it off with a recipe to secure a website with SSL.

Chapter 11, Linking to the Web, connects our installer to online resources. We'll see how to open web pages, display hyperlinks, and install shortcuts to websites.

Chapter 12, Installing SQL Server Databases, is dedicated to recipes about deploying SQL Server databases. If SQL Server isn't already installed on the target computer, we can install it with a bootstrapper. We'll add tables, insert data, and register an ODBC data source.

Chapter 13, Admin Tasks, rounds up a collection of administrative chores, including installing scheduled tasks, adding event sources, registering performance counters, and adding exceptions to the Windows firewall.
Organizing and Building WiX Projects

In this chapter, we will cover the following topics:

- Installing WiX and creating a new project in Visual Studio 2013
- Referencing the output of a .NET console application in a WiX project by using a preprocessor variable
- Separating a portion of WiX markup into its own library
- Compiling a WiX installer on a build machine using MSBuild
- Building a WiX installer from the command line

Introduction

The trouble with any bit of code is handling the logistics of getting it from development to production. How do we work on it in our favorite IDE initially and then allow it to be built elsewhere? Perhaps on a build server that doesn't have access to the IDE?

WiX solves this problem for its own code by allowing it to be built using a variety of workflows. As part of the WiX toolset, we get the compiler and linker needed to create an MSI installer. If we're using Visual Studio then we also get project templates that use these tools on our behalf so that the entire build process is effortless. If we're trying to fit WiX into an automated deployment pipeline, we can either call the compiler and linker from the command line or use ready-made MSBuild tasks.

The recipes in this chapter are designed to get you comfortable when working with a WiX project in Visual Studio and also building it in various ways outside of the IDE. This way, you'll know how to get it from development to production with ease.
Organizing and Building WiX Projects

**Installing WiX and creating a new project in Visual Studio 2013**

It's possible to work with WiX outside of Visual Studio, but within it, you'll benefit from the project templates; IntelliSense and shortcuts to the compiler and linker settings are available on the project's properties. The only downside is that WiX doesn't work with Visual Studio Express. However, its installer will give you the compiler and linker so that you can still get work done even if you're using Notepad to write the markup. SharpDevelop, a free and open source IDE, also supports WiX projects.

Getting WiX up and running starts with downloading and running its installer. This is a one-stop shop to update Visual Studio, getting the compiler and linker as well as other utilities to work with MSI packages. WiX supports Visual Studio 2005 and later, including Visual Studio 2013, which we'll cover here. In this recipe, we will download and install WiX and create our first setup project.

**Getting ready**

To prepare for this recipe, install Visual Studio 2013 and close it before installing WiX.

**How to do it...**

Download and install the WiX toolset to get access to new project templates, IntelliSense, and project properties in Visual Studio. The following steps will guide you:

1. Open a browser, navigate to [http://www.wixtoolset.org](http://www.wixtoolset.org), and follow the link to the downloads page:

   ![WiX Toolset v3.9](http://www.wixtoolset.org/downloads/v3.9.png)

   **Recommended download**

   - WiX v3.9.6095
     - **Rating:** 5 stars (based on 1 rating)
     - **Released:** Oct 31, 2014
     - **Reviewed:** 1 review
     - **Downloads:** 15791
     - **Dev status:** Stable

   **application, 32bit, uploaded Oct 30 - 13887 downloads**
2. Once downloaded, launch the WiX installer and click on **Install**:

3. After completing the installation, open Visual Studio and go to **File | New | Project** | **Windows Installer XML**.

4. Select the **Setup Project** template from the list of available project types. The version of .NET that's displayed has no bearing on the project since it's comprised of XML mark-up and not .NET code. Give the project a name and click on **OK**:
5. The project will initially include a file named `Product.wxs`, which contains the skeleton markup you'll need to create an installer:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
  <Product Id="*"
    Name="My Software"
    Language="1033"
    Version="1.0.0.0"
    Manufacturer="My Company"
    UpgradeCode="889e2707-5235-4d97-b178-cf0cb55d8ab8">
    <Package InstallerVersion="200"
      Compressed="yes"
      InstallScope="perMachine" />
    <MajorUpgrade DowngradeErrorMessage="A newer version of [ProductName] is already installed." />
    <MediaTemplate />
    <Feature Id="ProductFeature"
      Title="MyFirstWixProject" Level="1">
      <ComponentGroupRef Id="ProductComponents" />
    </Feature>
  </Product>
  <Fragment>
    <Directory Id="TARGETDIR" Name="SourceDir">
      <Directory Id="ProgramFilesFolder">
        <Directory Id="INSTALLFOLDER" Name="My Software" />
      </Directory>
    </Directory>
  </Fragment>
  <Fragment>
    <ComponentGroup Id="ProductComponents"
      Directory="INSTALLFOLDER">
      <!-- TODO: Remove the comments around this Component element and the ComponentRef below in order to add resources to this installer. -->
      <!-- <Component Id="ProductComponent"> -->
      <!-- TODO: Insert files, registry keys, and other resources here. -->
      <!-- </Component> -->
    </ComponentGroup>
  </Fragment>
</Wix>
```
How it works...

The WiX team has always worked quickly to keep up with the latest versions of Visual Studio. For example, WiX 3.9 supports Visual Studio 2013. When we launched the installer, it checked which versions of Visual Studio were present and registered its project templates with all that were compatible.

Behind the scenes, WiX introduces a new project type that has a .wixproj file extension. This project file contains MSBuild markup, which points to the WiX compiler and linker. Other IDEs, such as SharpDevelop, can take advantage of these project files to build MSI packages too.

The Product.wxs file contains everything we need to get started with writing WiX markup. The best coding practices for how to structure a WiX file have been defaulted for you. For example, the Directory elements are separated into a Fragment element so that directories are decoupled from the files that will go into them. A ComponentGroup has been set up with a comment guiding you to add Component elements to it. Each version of WiX brings a better Product.wxs file with it.

There's more...

If you were curious about what effect changing the version of the .NET framework listed in the drop-down list at the top of the New Project window would have, the answer, at least for setup projects, is nothing at all. A WiX file contains XML and is compiled with a specialized WiX compiler, so the version of .NET that we select will ultimately be ignored. That's not to say that it doesn't make a difference for any of the other project types. For example, C# Custom Action Project will have a dependency on the version of .NET that's selected. Anyone who uses the installer that in turn uses that custom action will need to have that version of .NET installed.

Referencing the output of a .NET console application in a WiX project by using a preprocessor variable

After setting up our WiX project, the first thing we'll probably want to do is package up the files that we plan to install. Since we're working in Visual Studio, we'll likely want to include the output of other projects such as the .exe file that's created from a console application project. At first, we could try hardcoding the path to the file:

```xml
<Component Id="cmpMyConsoleAppEXE"
  Guid="{882DB6AA-1363-4724-8C43-2950E7ABECD4}"
  >
  <File Source="..\MyConsoleApp\bin\Debug\MyConsoleApp.exe" />
</Component>
```
Although this works, it's a bit brittle and will break if the path to the file changes. Instead, we can use a preprocessor variable to store the path and allow Visual Studio to keep it up-to-date through the power of project references. In this recipe, we'll reference a console application's output and use a preprocessor variable to include that output in our installer.

Getting ready

To prepare for this recipe, create a new WiX setup project and name it ConsoleAppInstaller.

How to do it...

Use a preprocessor variable to get the path to a project's output with the following steps:

1. Add a new C# console application to the same solution as the ConsoleAppInstaller setup project by right-clicking on the solution in Solution Explorer, going to Add | New Project... | Visual C# | Console Application and naming it TestApplication. The name matters as we'll be referencing it later:

![Add New Project](image)
2. Within the setup project, add a reference to TestApplication by right-clicking on the References node in Solution Explorer, choosing Add Reference..., and finding TestApplication under the Projects tab. Click on Add and then on OK:
3. Within the setup project, open `Product.wxs` and replace the `ComponentGroup` markup inside the last fragment with the following code:

```xml
<ComponentGroup Id="ProductComponents"
    Directory="INSTALLFOLDER">
    <Component Id="cmpTestApplicationEXE"
        Guid="{6E2A6370-4784-4CF3-B42B-AA2D29EA5B1B}"
        >
        <File Source="$(var.TestApplication.TargetDir)TestApplication.exe" />
    </Component>
</ComponentGroup>
```

4. Build the project and `TestApplication.exe` will be included in the MSI file. Note that you must set the `EmbedCab` attribute on the `MediaTemplate` element to `yes` to include the CAB file that WiX creates, which is where our `.exe` file is stored, inside the MSI. Also, this example assumes that `TestApplication.exe` is the only file you'd like to include in the installer. Other files, such as DLLs, can be included in the same way though.

**How it works...**

When we referenced the C# console application within the WiX setup project, the preprocessor variable `$(var.[ProjectName].TargetDir)` was made available to us, where `ProjectName` in this case is `TestApplication`. TargetDir points to the output directory of the console application project where our compiled `TestApplication.exe` file can be found.

Other preprocessor variables are also made available. For example, `$(var.[ProjectName].TargetFileName)` gives you the name of the compiled application, which for us would be `TestApplication.exe`. A full list of these variables can be found at http://wixtoolset.org/documentation/manual/v3/votive/votive_project_references.html.

Another benefit of referencing the console application project in this way is that it ensures it is compiled before our setup project is. This way, our installer always includes the most up-to-date version of the application.
The GUID used for the \texttt{Guid} attribute on the \texttt{Component} element in this example can be any GUID, not just the one listed. You can generate a new one in Visual Studio by navigating to \textbf{Tools} | Create GUID. Use \textbf{Registry Format} as the GUID's format. More information can be found at \url{http://wixtoolset.org/documentation/manual/v3/howtos/general/generate_guids.html}.

You can also set the \texttt{Guid} attribute to an asterisk (*) or omit it altogether and WiX will set the GUID for you. You should choose your own if you plan on authoring a patch file for the application in the future or if the contents of \texttt{Component} don’t contain an element that can be marked as a \texttt{KeyPath} element.

---

**Separating a portion of WIX markup into its own library**

As a project grows in complexity and size, we may end up with different teams building different parts of the software in relative isolation. Each team may want to control how their module will be installed or, during development, install only the modules that their code depends upon into their dev environment. To handle these scenarios, we can split our installer into chunks of WiX code called setup libraries.

A setup library can be compiled independently and plugged into the main, monolithic setup project later. We can also include the library in a team-owned setup project that only contains the modules required by the team. In essence, we can mix and match libraries wherever we need them to create installers for different purposes.

You might also want to share some complex installer markup, such as a user interface, with other installers, and a library is the perfect way to do this. Although it’s outside the scope of this book, setup libraries are also used when building custom WiX extensions. In this recipe, we’ll see how to create a setup library and include it in our setup project.

**Getting ready**

To prepare for this recipe, create a setup project and call it \texttt{SetupLibraryInstaller}. 
How to do it...

Add a setup library to the solution and reference it in a setup project. The following steps show how to do this:

1. Add a new setup library to the same solution as the setup project by right-clicking on the solution in Solution Explorer and navigating to Add | New Project... | Windows Installer XML | Setup Library Project. For this example, name the project MySetupLibrary:

![Add New Project dialog](image)

2. After it's created, right-click on the MySetupLibrary project in Solution Explorer and go to Add | New Item... | Text File. Name the text file SampleTextFile.txt and click on Add. Our library will install this single text file.

3. Right-click on the MySetupLibrary project in Solution Explorer again and select Properties. Select the Tool Settings tab and add -bf, which stands for bind files, to the librarian textbox, as shown in the following screenshot:

![Tool Settings dialog](image)
4. Open Library.wxs and replace the existing markup with the following:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
    <Fragment>
        <DirectoryRef Id="INSTALLFOLDER">
            <Directory Id="SampleComponentsDirectory"
                Name="Sample Components" />
        </DirectoryRef>

        <ComponentGroup Id="SampleComponentGroup"
            Directory="SampleComponentsDirectory">
            <Component Id="cmpSampleTextFileTXT"
                Guid="{5382BC02-4484-4C9B-8734-A99D20632EA9}"
                File Source="SampleTextFile.txt" />
        </ComponentGroup>

        <Feature Id="SampleFeature">
            <ComponentGroupRef Id="SampleComponentGroup" />
        </Feature>
    </Fragment>
</Wix>
```
5. In the SetupLibraryInstaller project, add a reference to the setup library by right-clicking on the References node in Solution Explorer and selecting Add Reference.... Click on the Projects tab, highlight MySetupLibrary, click on Add, and then on OK.

6. Open Product.wxs and add a FeatureRef element with an ID of SampleFeature. This includes the feature we added to the Library.wxs file of SetupLibrary in our installer. FeatureRef can go after the existing Feature element as follows:

   `<Feature Id="ProductFeature"
     Title="ConsoleAppInstaller"
     Level="1">
     <ComponentGroupRef Id="ProductComponents" />
     <FeatureRef Id="SampleFeature"/>
   </Feature>`

How it works...

Our setup library contains WiX markup to install a single text file called SampleTextFile.txt. Ordinarily, when you build a library like this, the source files don't get stored within it. Instead, only the WiX markup is compiled without any of the source files it refers to. In that case, we would have had to copy SampleTextFile.txt to the setup project's directory too, so that it can be found at link-time when compiling the installer.

However, because we added the -bf flag, which stands for bind files, to the Librarian settings, the text file was serialized and stored within the library. The -bf flag will handle serializing and storing any type of file including executables, images, and other binary data. Setup libraries are compiled into files with a .wixlib extension.

The markup we added to the library created a component, directory, and feature for the text file. To integrate the new directory with the existing directory structure as defined by our setup project, we chose to reference INSTALLFOLDER with a DirectoryRef element. Just be sure that there's a corresponding Directory element in your setup project that has this name. At link time, the DirectoryRef element in the library is merged with the Directory element in the setup project by matching their IDs. Once we had this, we were able to add a new subdirectory within the INSTALLFOLDER directory called Sample Components. After installation, we can see that the new directory was created and it contains our text file:
To be sure that our library gets compiled before our setup project, we referenced it within the setup project using the References node. Then, to create a link to the library, we included a FeatureRef element in Product.wxs, which had an ID matching the Feature defined in the library. This pulls the Feature with all of its components into the installer.

**There's more...**

The setup libraries might contain more than just components, features, and directories. For example, they might define markup for a user interface using a UI element, which could then be linked to our installer with a UIRef element. Basically, if you can find a corresponding *Ref element, such as DirectoryRef, UIRef, ComponentGroupRef, or FeatureRef, then you'll be able to separate that type of element into a library and use its *Ref element to link it to the setup project.

Even if you can't find a corresponding *Ref element, as long as you have a reference of some kind, such as Property and PropertyRef, the rest of the elements in the library will be carried along with it into the installer. So, at the very least, you could include a single Property in the library and use that as the link between the library elements and the installer.

**Compiling a WiX installer on a build machine using MSBuild**

The WiX Toolset places its compiler and linker in C:\Program Files (x86)\WiX Toolset_v3.9\bin. This is fine when compiling on your own machine but becomes a concern when you'd like to share your project with others or have it compile on a build server. WiX will have to be installed on each computer that builds the project.

Alternatively, we can store the WiX tools in source control, and then whoever needs to build a setup project can get everything they need by cloning the repository. This will also help us keep a handle on which version of WiX we're compiling against on a project-by-project basis.
In this recipe, we'll store the WiX binaries in a fictitious source control directory on the C: drive. We'll then update the .wixproj file of a setup project to use the MSBuild tasks stored there. I will be using a server with the Windows Server 2012 R2 operating system installed on it. You should be able to follow along with other versions of Windows Server.

Getting ready

To prepare for this recipe, perform the following steps:

1. Install the .NET Framework 3.5. It's needed by the WiX build tasks. In Windows Server 2012 R2, it can be installed as a feature within Server Manager:

2. Next, we'll need the MSBuild engine, which is part of Microsoft Build Tools. It can be downloaded from http://www.microsoft.com/en-us/download/details.aspx?id=40760.

3. After installing MSBuild, add its installation directory to the computer's PATH environment variable. Get there by right-clicking on This PC in file explorer and then going to Properties | Advanced system settings | Environment Variables.... Scroll through the list of system variables until you find the one labeled Path. Highlight it, click on Edit..., and then add the path to the MSBuild directory into the Variable value field, preceded by a semicolon. Then, click on OK:
How to do it…

Download the WiX binaries and update your setup project to use the included MSBuild tasks:

1. Open a browser, navigate to http://www.wixtoolset.org, and follow the link to the downloads page. Download wix39-binaries.zip:

2. Make sure that the ZIP file is unblocked by right-clicking on it, choosing Properties, clicking on Unblock (if you don’t see it, just continue to the next step), and then on OK.

3. Extract the contents of the ZIP file to C:\SourceControl\WiX39. Perform this step on both the server and on your own development computer so that our WIX projects can be built in both places using the MSBuild tasks from this folder (note that in a real-world scenario, our source control system would be responsible for copying the binaries to each computer):
4. We will build a simple setup project to confirm that we've got everything on the server configured correctly. Create a setup project on your development machine and call it BuildMachineInstaller.

5. Open the BuildMachineInstaller.wixproj file and add the WixToolPath, WixTargetsPath, and WixTasksPath properties as shown, making sure that the value of WixToolPath ends in a backslash:

```xml
<PropertyGroup>
  <Configuration Condition=" $(Configuration) == 'Debug' ">Debug</Configuration>
  <Platform Condition=" $(Platform) == 'x86' ">x86</Platform>
  <ProductVersion>3.9</ProductVersion>
  <ProjectGuid>f80ca9fc-8e42-406e-92f9-06e484e94d67</ProjectGuid>
  <SchemaVersion>2.0</SchemaVersion>
  <OutputName>BuildMachineInstaller</OutputName>
  <OutputType>Package</OutputType>
  <WixToolPath>C:\SourceControl\WiX39</WixToolPath>
  <WixTargetsPath>$(WixToolPath)wix.targets</WixTargetsPath>
  <WixTasksPath>$(WixToolPath)WixTasks.dll</WixTasksPath>
  <WixTargetsPath Condition=" $(WixTargetsPath) == '' AND $(MSBuildExtensionsPath32) != '' ">
    $(MSBuildExtensionsPath32)\Microsoft\WiX\v3.x\Wix.targets</WixTargetsPath>
  </WixTargetsPath>
</PropertyGroup>
```

6. Copy the BuildMachineInstaller solution folder and all of its subfolders to C:\SourceControl on the build server.

7. Open a command prompt via Run | cmd, execute the following commands to change the directory to the BuildMachineInstaller folder and compile the solution using MSBuild:

```bash
cd C:\SourceControl\BuildMachineInstaller
msbuild BuildMachineInstaller.sln
```

**How it works...**

We started with a blank slate of a freshly installed Windows Server 2012 R2 operating system. Therefore, we had to install all the required software including .NET Framework 3.5 and Microsoft Build Tools 2013. The latter gives us the MSBuild engine, whose path we included in the computer's PATH environment variable.
Next, we downloaded the WiX binaries and copied them to C:\SourceControl. With a source control system, these files could be shared among all computers that need to compile our setup projects. We also had to update our project’s .wixproj file so that it knew where to find these WiX binaries. This is accomplished by adding three MSBuild properties: WixToolPath, WixTargetsPath, and WixTasksPath. The first property sets the path to the WiX binaries, the second to the wix.targets file, and the third to WixTasks.dll. With all of this setup out of the way, we opened a command prompt, navigated to the folder where our solution file was on the build server, and compiled it using MSBuild.

Building a WiX installer from the command line

WiX has excellent integration with Visual Studio, but that shouldn’t stop you from using it in other IDEs. We ought to be able to create an installer using only Notepad and the WiX compiler and linker if we wanted to. Luckily, WiX gives us the freedom to do this. In this recipe, we’ll write a simple .wxs file and compile it into an MSI package using Candle, which is the WiX compiler, and Light, which is the WiX linker.

Getting ready

To prepare for this recipe, perform the following steps:

1. Using a text editor such as Notepad, create a file called Product.wxs and add the following markup to it:

   ```xml
   <?xml version="1.0" encoding="UTF-8"?>
   <Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
   <Product Id="*" Name="My Software" Language="1033" Manufacturer="My Company" Version="1.0.0.0" UpgradeCode="8c7d85db-b0d1-4a9a-85ea-130836aeef67">
   <Package InstallerVersion="200" Compressed="yes" InstallScope="perMachine" />
   <MajorUpgrade DowngradeErrorMessage="A newer version of [ProductName] is already installed." />
   <MediaTemplate EmbedCab="yes" />
   <Feature Id="ProductFeature" Title="The main feature" Level="1">
   <ComponentGroupRef Id="ProductComponents" />
   ```
Organizing and Building WiX Projects

```wix
</Feature>
</Product>

<Fragment>
  <Directory Id="TARGETDIR" Name="SourceDir">
    <Directory Id="ProgramFilesFolder">
      <Directory Id="INSTALLFOLDER" Name="My Software" />
    </Directory>
  </Directory>
</Fragment>

<Fragment>
  <ComponentGroup Id="ProductComponents" Directory="INSTALLFOLDER">
    <Component Id="cmpMyTextFileTXT" Guid="{A4540658-09B6-46DA-8880-0B1962E06642}"
      >
      <File Source="MyTextFile.txt" />
    </Component>
  </ComponentGroup>
</Fragment>

2. This installs a text file called MyTextFile.txt. So, add a text file with this name to the same directory as Product.wxs. We will compile the two files from the command line to create an installer.

How to do it...

Open a command prompt and use candle.exe and light.exe to compile and link our WiX source file:

1. Open a command prompt by navigating to Run | cmd.
2. Change the directory to where the Product.wxs and MyTextFile.txt files are using the following command line:
   `cd C:\MyProject`
3. Use Candle to compile the .wxs file into a .wixobj file and then place it in an output folder called obj. Be sure to surround the path to Candle, `%WIX%\bin\candle`, with quotes since it will contain spaces when it is expanded:
   `"%WIX%\bin\candle" *.wxs -o obj`
4. Use Light to link the text file and the .wixobj file together to form an MSI:
   `"%WIX%\bin\light" obj*.wixobj -o bin\CommandLineInstaller.msi`
How it works...

When we installed the WiX toolset, it gave us the WiX compiler, which is candle.exe, and linker, which is light.exe. These are the only tools we need to create an MSI from our WiX source file, Product.wxs. From the command line, we navigated to the directory where our source file was and then used Candle and Light to compile and link the file to create an MSI installer.

The first argument we passed to Candle was *.wxs. This selects all the .wxs files in the current directory and includes them in the compilation. Next, the -o argument tells Candle where to send the output of the compilation step. In this case, we sent it to a directory called obj. Note that the directory name ends in a backslash so that Candle knows that it's a directory. If it didn't exist before, it will be created.

The output of the Candle command was a file called Product.wixobj. This was an intermediate file that was picked up by light.exe in the next step. The first argument we passed to Light was the location of the .wixobj files: obj/*.wixobj. By using an asterisk, we select all the .wixobj files in the obj directory. The -o argument tells Light where to create the MSI file and what to name it. In this case, we create a file called CommandLineInstaller.msi.

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CommandLineInstaller.msi</td>
<td>11/20/2014 11:05 ...</td>
<td>Windows Installer ...</td>
</tr>
<tr>
<td>CommandLineInstaller.wixpdb</td>
<td>11/20/2014 11:05 ...</td>
<td>WIXPDB File</td>
</tr>
</tbody>
</table>

Another file called CommandLineInstaller.wixpdb was also created. This can be used when building patch files. You can learn more by reading Peter Marcu's blog post WIX: Introducing the WixPdb at http://petermarcu.blogspot.com/2008/02/wix-introducing-wixpdb.html.

There are a number of arguments that can be passed to Candle and Light that you might want to get to know. Passing the -? flag to either will give you a list of all the available options:

*%WIX%bin\candle* -?

*%WIX%bin\light* -?

We used the %WIX% system environment variable to resolve the path to the WiX bin directory, where candle.exe and light.exe are present. This variable is added when you install the WiX toolset and resolves to C:\Program Files (x86)\WiX Toolset v3.9. It will not be present if you are using the WiX binaries directly without installing the WiX toolset.
Where to buy this book

You can buy WiX Cookbook from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.