

Installing VSIX package via WiX installer

Introduction

WiX Toolset is a powerful set of tools used to create Windows installations packages. Unfortunately it's learning curve is quite steep and the documentation is not it's strong point. That's why I decided to show you, step by step, how to create an installer for your VSIX packages. I will not cover all of the WiX features as this is not intended to be an exhaustive WiX guide but I will describe in detail what is necessary to deploy a VSIX package. Also I will add some interesting details that can be useful more in general when creating WiX installers. The end result should be a MSI installation file that will correctly install and uninstall your VSIX package. Let's start.

It's true that VSIX leverages an installer on it's own. Still you should consider this approach if your installation is more complex and/or requires other components to be installed.

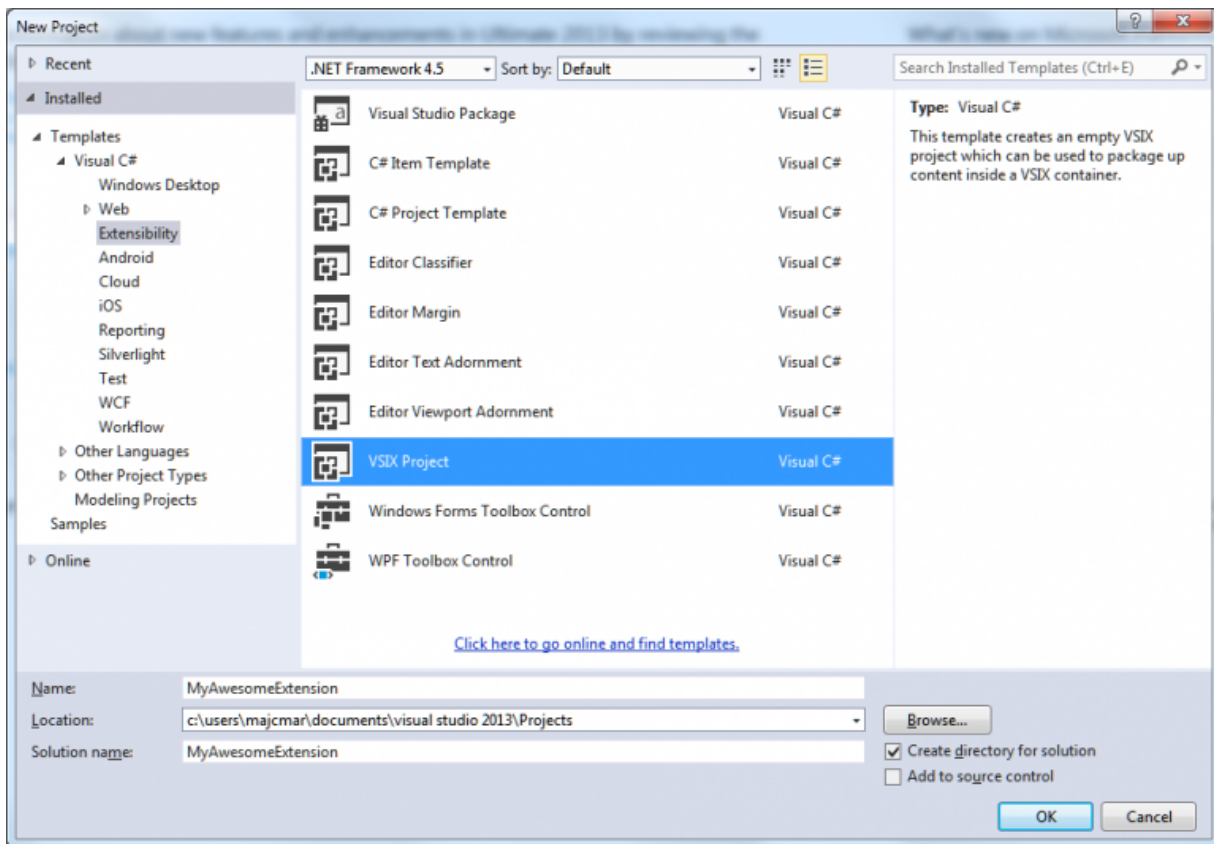
Prerequisites

In order to follow this example you will need any version of Visual Studio starting from 2010 till the most recent 2015. Also any sku except the Express is fine. You will need a Visual Studio SDK and WiX Toolset installed. The examples you will find are made w Visual Studio 2013 and WiX 3.9 R2. You can download them from here ([SDK](#) and [WiX](#)).

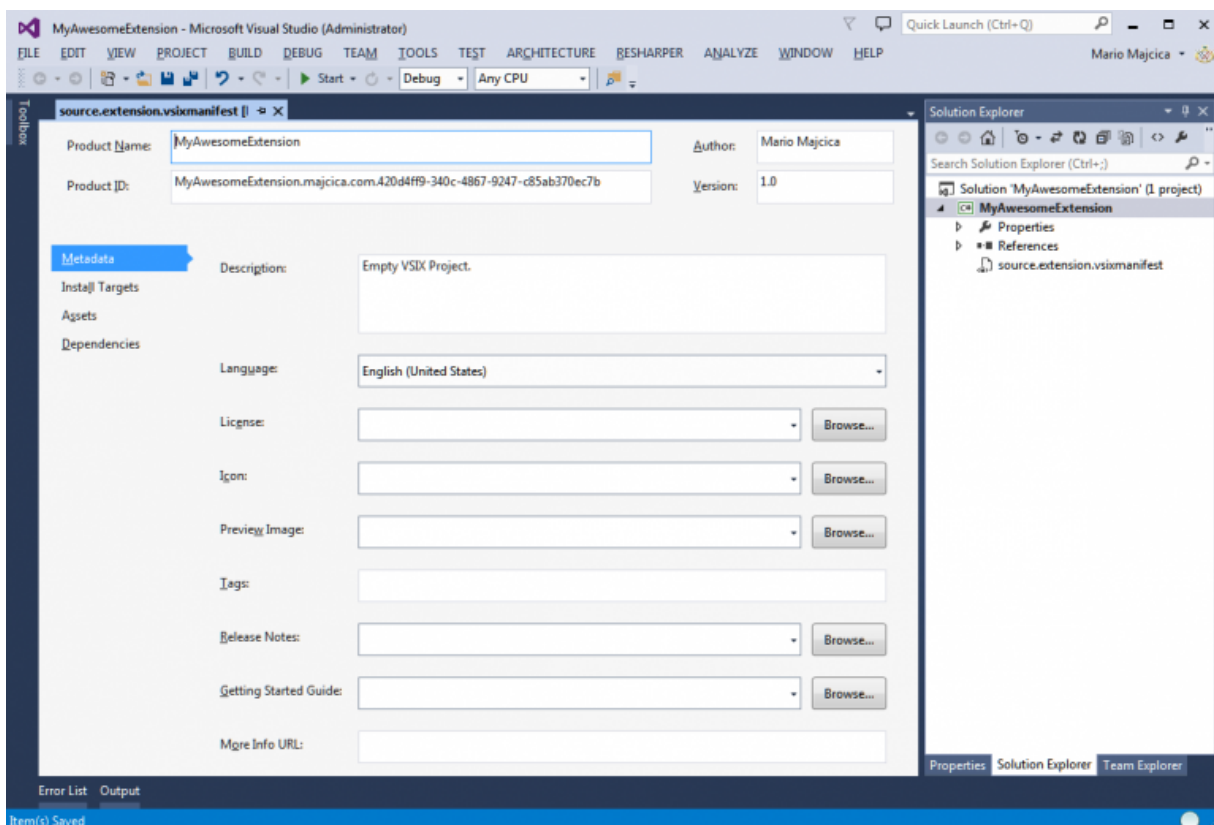
Creating the necessary projects

In order to start our practical example, let's first create the necessary project that we are going to use by our installer later. Create a new project in Visual Studio and choose a "VSIX Project".

I will name it MyAwesomeExtension.

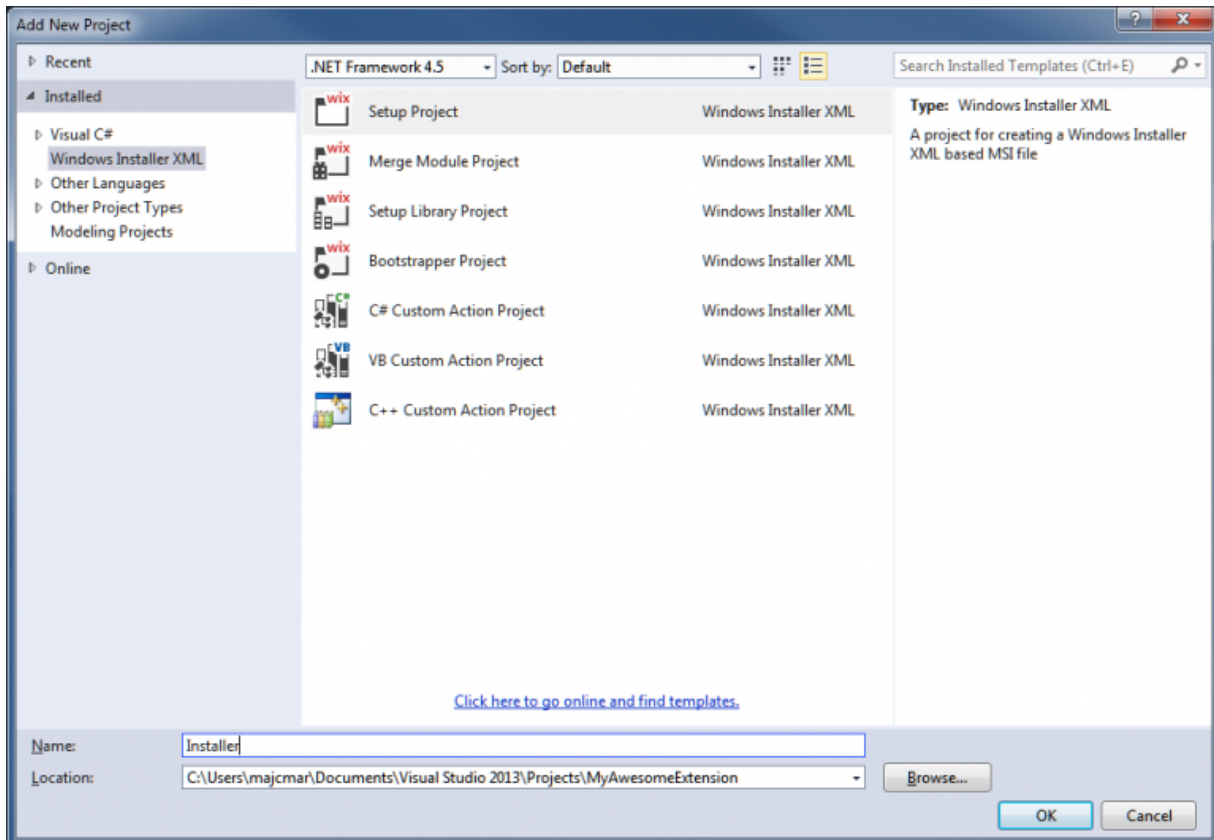


Once the project is created you will be presented by the vsixmanifest designer. You can leave all the defaults. Make sure only that the Product ID is set in the unique way and that, for commodity, it doesn't contain spaces.



This should be it for what concerns our extension. I will not focus on creating Visual Studio extensions and this is just enough to have an extension registered and working. If you hit F5 you should start another instance of Visual Studio, by default in Experimental Mode, and if you check inside Tools > Extensions and Updates menu you should find your newly created extension.

The next thing is adding a WiX Setup Project into your solution. I will simply call it Installer.



This was about it. In the next paragraph I will show you how to install your VSIX package using WiX.

WiX Toolset installer

First thing first. We are going to take advantage of a standard custom action offered by WixVSExtension. In order to do so, we need to reference the WixVSExtension.dll. You can find it inside the WiX installation folder which in my case is C:\Program Files (x86)WiX Toolset v3.9bin. Be aware that the Add reference panel for the WiX setup projects is custom and it slightly differs from the default one. Aside of having the necessary to help us

installing (and uninstalling) VSIX packages, it also contains many other useful properties and custom actions that will ease interacting with Visual Studio. You can find a list of the properties and custom actions [here](#).

After the reference to the project is added we need to include the WixVSExtension namespace in our main wxs file (by default Product.wxs). In the Wix element next to the default namespace add the following `xmlns:VSExtension="http://schemas.microsoft.com/wix/VSExtension"`. The end result should look like this:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
3     xmlns:VSExtension="http://schemas.microsoft.com/wix/VSExtension"
4     ...
```

Now we are going to add another reference to our Wix project and it will point to our VSIX project. This is done for commodity so that we can point Wix to the output of our VSIX project. Once this is done we need to define a component that will leverage the custom action defined in WixVSExtension that will actually install our VSIX. Inside the fragment where an example of ComponentGroup is defined, we are going to remove all of the code inside it, comprehensive of ComponentGroup itself. Define the following:

```
1 <Fragment>
2   <Component Id="C_MyAwesomeExtension" Guid="B93F32FC-4B97-4BA7-BAA0
3     <VSExtension:VsixPackage File="MyAwesomeExtensionFile" PackageId
4     <File Id="MyAwesomeExtensionFile" Name="$ (var.MyAwesomeExtension
5   </Component>
6 </Fragment>
```

Let's analyze what we added.

First element we added is a Component, that we called for convenience `C_MyAwesomeExtension`, we assigned a unique identifier to our component (so that MSI can track this element) and we specified the folder in which this component should be deployed. I will not get into details of what a component is and how it works, I do suppose you already know that and if not you can find more info [here](#).

Inside our component we specified two elements. The simpler one to understand is for sure the File element. We gave an id to the file that we do need in order to deploy our VSIX and named it by using the name of the compiled assembly. In order to retrieve the name I used

a variable that was generated by Wix because of the reference to the project we added, appending to the result the .vsix string. Now, each time you reference a project Wix will create several variables dynamically and they will be at your service. A full list of them you can find at this [link](#). In case we were referencing a project that will output an executable or a library, we could use the TargetFileName variable and omit the .vsix. However as by default Wix doesn't know nothing about VSIX project types and outputs, we need to use TargetName property and concatenate .vsix to it in order to get a file name with the correct extension.

Same thing we did with the source attribute. With it we do specify where this file should be copied from in order to be placed inside the MSI installation file. As for the Name attribute, we are going to use some of auto generated variables to get to the correct path of our compiled VSIX.

VSExtension:VsixPackage element

Let's focus on the main character of this article, the VSExtension:VsixPackage element. I will describe all of the attributes I used and also the one I haven't and describe the behavior that they do produce.

File

File, as the name suggest, requires to be set to the file name that we used in our MSI tables and points to our VSIX installer. Simply said it needs to be the same as the Id attribute we used for the File element that we discussed earlier. In this way the custom action will know where is the VSIX file that we are planning to install.

Packageld

This is a very important value. Earlier I mentioned the Product ID, when we were creating our VSIX extension. Well that's it, the unique product ID that is set inside vsixmanifest file in your extension. Make sure that it does not contain any spaces and if so add quote marks and escape them properly.

Target

You can specify one of the following values: integratedShell, professional, premium,

ultimate, vbExpress, vcExpress, vcsExpress, vwdExpress. It will indicate which edition (SKU) of Visual Studio is required in order to register this extension.

TargetVersion

Specifies the version of Visual Studio for which to register the extension. Should contain a version number as at example 11.0 in case of Visual Studio 2012 or 14.0 in case of Visual Studio 2015.

Vital

Indicates if the failure of the installation of the VSIX extension will cause a rollback of the whole installation.

Permanent

Often underestimated and forgotten parameter. This is the reason of many complains about the extension not uninstalling on msi uninstall. And that's all about it. If set to no, your extension will be uninstalled once you try to uninstall your product. If set to yes, your extension will not be removed in case the user uninstalls the product.

VsixInstallerPathProperty

As this custom action relies on VsixInstaller.exe to trigger the installation of the VSIX, it needs to know where the VsixInstaller.exe is located in order to use it. It is recommended not to set this attribute and let it be populated by the custom action itself. By default, the latest VsixInstaller.exe on the machine will be used to install the VSIX package.

Believe it or not this is sufficient to let the Wix install your VSIX extension. In our case we do target the Visual Studio 2013 and accepting any edition from professional above.

Behind the scene

So what's the magic behind this custom action? The simplest way to verify what is happening is to compile our project and launch the installation. Before we are able to this there are two minor things we need to take care of. Inside our Feature element we need to change the ComponentGroupRef to ComponentRef and point the id to our C_MyAwesomeExtension component. Also to make the output a single file, we will tell Wix

to integrate the cabinet inside the msi itself. To do so under the MediaTemplate element we are going to add the EmbedCab attribute and set it's value to yes.

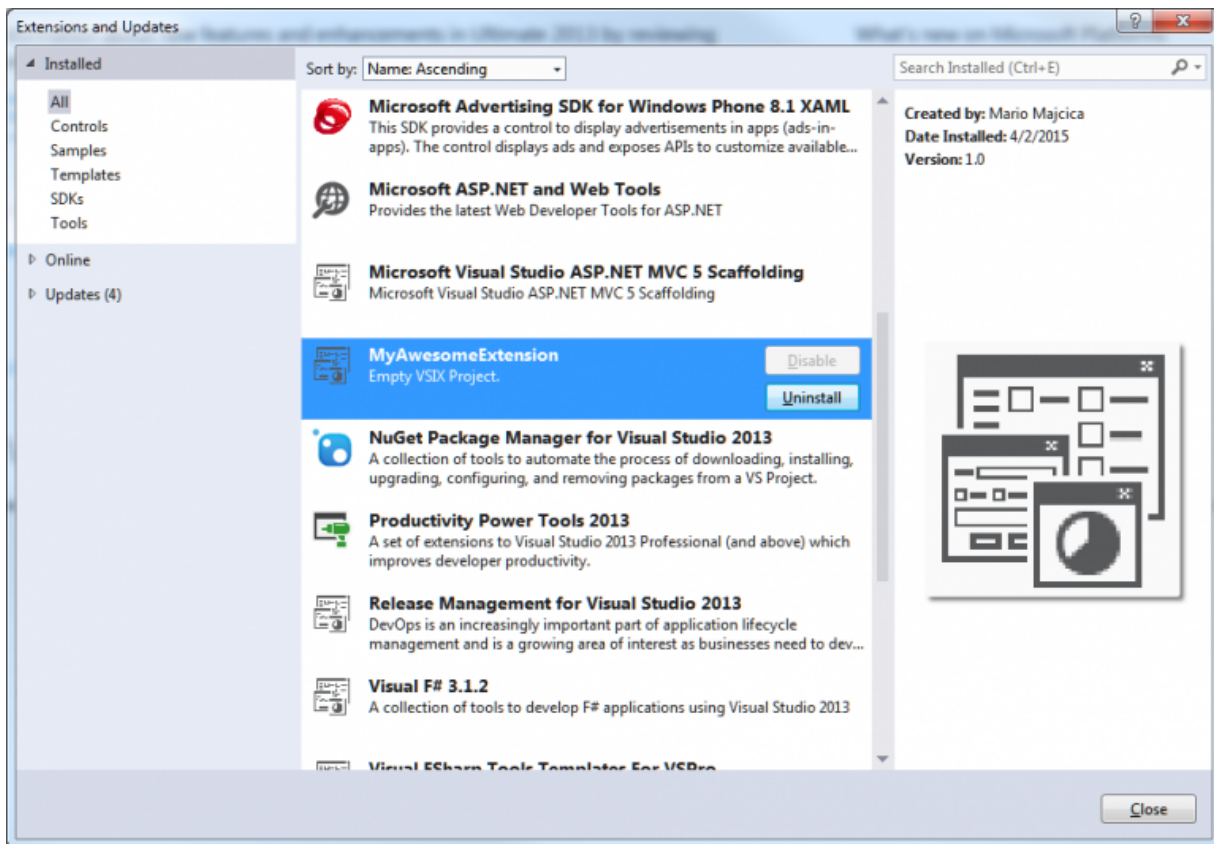
```
1 <MediaTemplate EmbedCab="yes" />
2
3 <Feature Id="ProductFeature" Title="Installer" Level="1">
4   <ComponentRef Id="C_MyAwesomeExtension" />
5 </Feature>
```

We can compile now our project and position our self in the build output directory. A file called Installer.msi should be there.

In order to see what is happening we can't just double click our newly created msi. We need to launch the installation from the command prompt and request the verbose logging to be activated. To request the log file being emitted we need to launch the following command: `msiexec /i Installer.msi /l*v install.log`. This will tell the msi installer to install our application and output a verbose log inside the file named install.log.

If you execute this command, you will briefly see installation starting and that's it. This is because we haven't chosen any UI for this installation. It's not in our scope showing how to create/choose and UI for the installer.

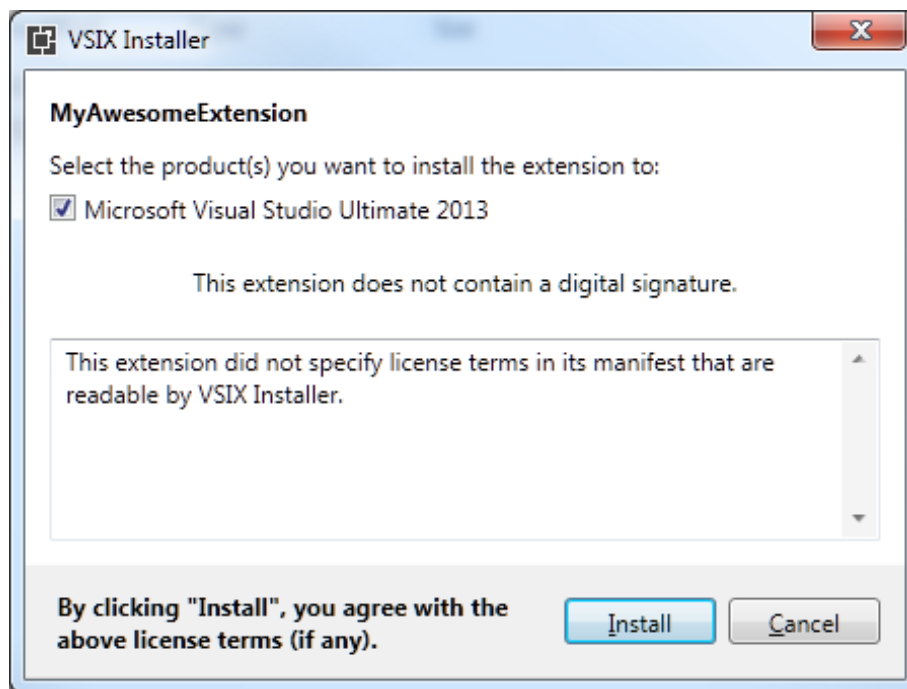
Let's check our default install folder at `C:\Program Files (x86)Installer`. Bingo! Our extension is there. Now open your Visual Studio 2013 instance and check the Extensions and updates window. Again, MyAwesomeExtension is there!



Open the newly created log file to see what was done in order to achieve this. Open the install.log file and search for “/skuName:Pro /skuVersion:12.0”. The line you landed on is the command that was executed by our VSExtension:VsixPackage custom action. The full command is following:

```
C:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\IDE\VSIXInstaller.exe /q  
/skuName:Pro /skuVersion:12.0 "C:\Program Files  
(x86)\Installer\MyAwesomeExtension.vsix" /admin
```

What happened here? Well nothing special. When you double click your vsix file, the vsix extension is associated with VSIXInstaller.exe and it will show you the following window:



Same thing with our custom action, it calls directly VSIXInstaller.exe and triggers the quite mode (so no UI is shown) and passes the necessary parameters in order for VSIXInstaller.exe to perform the installation. Do you recognize the other parameters? That's right, they do match what we have specified with the attributes on our VSExtension:VsixPackage element. Except the last /admin parameter. It implies that the extension will be installed to the admin extension location. If you prefer it to be installed on per user basis, you will need to create a perUser installation, thus, set the Package InstallScope to perUser.

You can get through the log file and probably you will find plenty of interesting information, like the search for the VSIXInstaller.exe and the values that are set in order to accommodate it, etc.

The same thing is done for what concerns the uninstall. We can launch the uninstall with the following command: `msiexec /x Installer.msi /l*v unistall.log`. As you can see it is similar to the install except that instead of /i we used a /x parameter. Once you execute it you will be prompted for the confirmation: