NuGetter: Automated TFS Build, Package and Deploy Version 1.1

NuGet Packaging and Deploy using TFS Build Workflow

Mark Nichols
June 16, 2011

Contents

Summary:

The NuGet project was designed to provide developers with a standardized mechanism for sharing and installing code, assemblies, etc. The creation of the packages is pretty straightforward to do in a manual way but wouldn't it be nice if it was all automated? And, not just the packaging, the versioning and deploying should be automated as well. This way, you as the developer can create a library, build it, deploy it and **immediately** test it.

Capabilities:

- Includes all phases of the build process: compile, version, pre-package, package, push/deploy and publish
- NuGet Package and deploy features for a simple to an extremely complex library package
- Single or multiple solution builds
- Single or multiple configuration builds
- Manage versioning of the assemblies coordinated or separately from the NuGet package
- Create a package, create and push a package or create a package and push and publish to a NuGet gallery
- Build and have immediate access to the package in a test environment through inherent "Push/Deploy" feature
- Push locations include the NuGet Gallery, a local directory, network share or web site
- Use in any combination of manual, continuous integration or scheduled builds
- Ability to execute PowerShell scripts prior to packaging to organize the files (e.g., lib, tools, content) for the NuGet packaging process (pre-packaging)
- No requirement for NuGet.exe to be installed on the build machine – NuGet.exe can be held in source control and deployed only at the time of the build
- All of the above is managed through the standard TFS Build Workflow process
- Remotely store/manage package information such as version numbers, API keys, and NuSpec manifest files

NuGetter Installation

NuGetter follows TFS build extension standards and is made up of a .NET 4 Workflow custom activity assembly and TFS build template(s). The most flexible approach is to store them both in source control and then tell the build controller where to find the assembly. Your other option is to install the assembly into the Global Assembly Cache (GAC). The downside to the GAC installation is that it is required on all build machines that intend to use the NuGetter build process.

NuGetter Workflow Activity Assembly (Use from Source Control)

To make the TfsBuild.NuGetter.Activities.dll available to the build process you store the assembly in source control and then tell the build controller where to find it. The build controller allows a single custom assembly store so if you have other custom assemblies, they will all have to be stored in the same Team Project location.

If you haven't created a store for custom assemblies, a recommended approach is to create a TFS Team Project to store the custom assemblies and any custom build templates. This way you have a central location for the assemblies and build templates. All projects that need the build capability can access them and you won't have to copy anything into an application's project area. Maintenance is much easier this way.

Once the assembly is added to source control, you need to inform the TFS build controller where to find the assembly. You can do this from the Team Foundation Server Administration Console under Build Configuration. Click "properties" for the build controller.
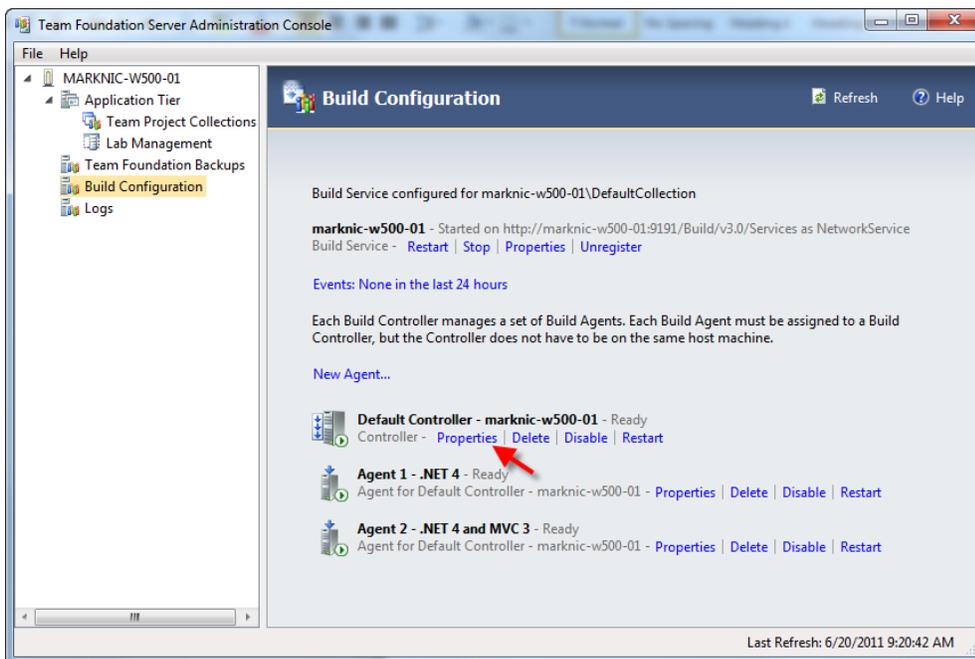
Another way to get to the properties dialog is to right-click "Builds" within a Team Project and select "Manage Build Controllers…" Then, select your build controller and click the "Properties" button.
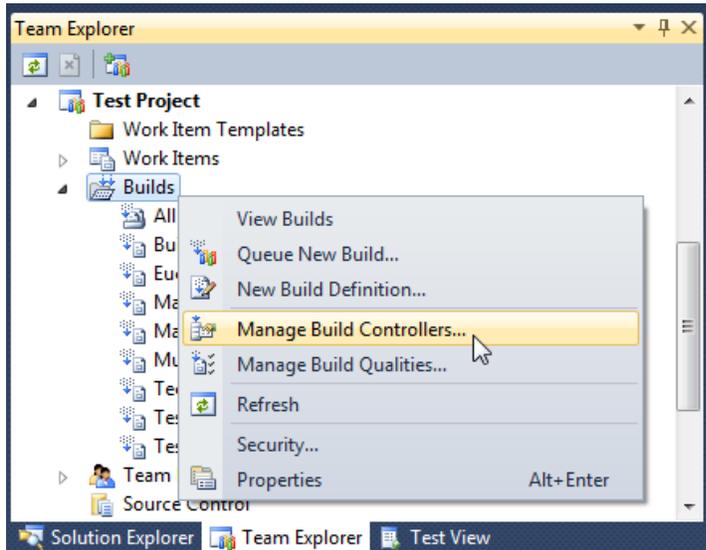
As long as you have the appropriate administrator rights, you can get there either way.
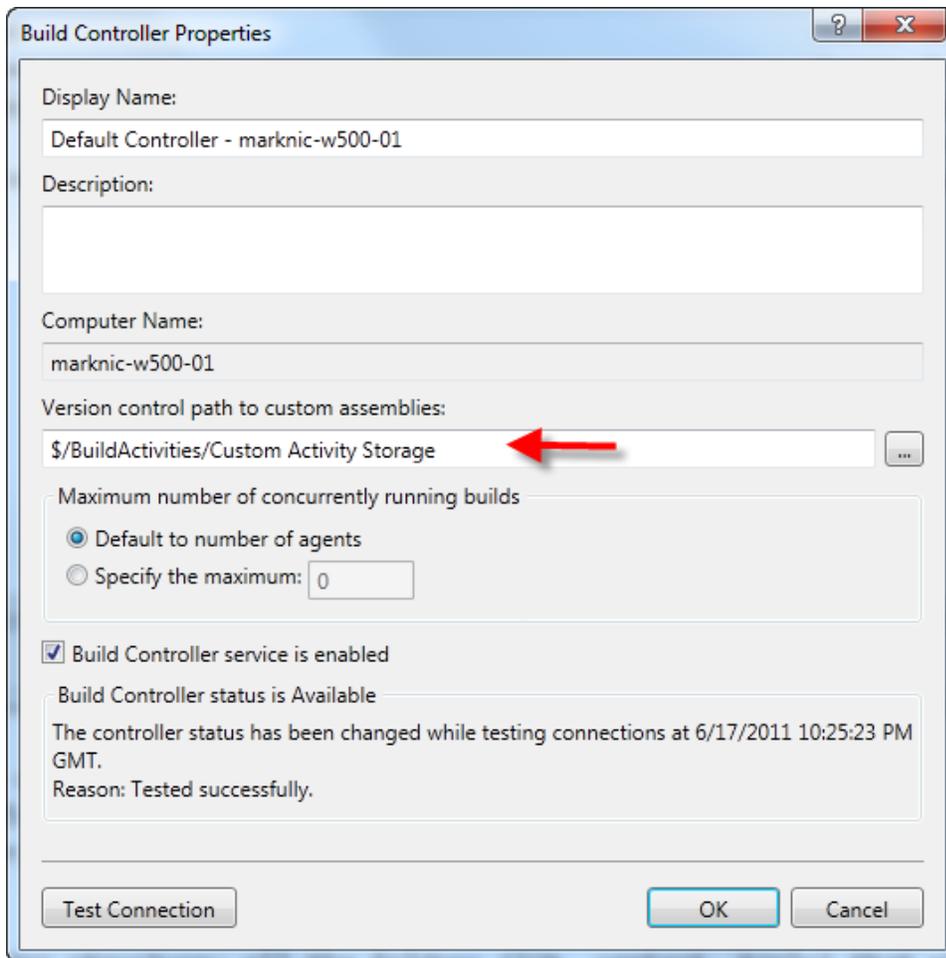
Once the dialog is up, browse to or enter the location in source control where you placed the custom workflow assembly. Click OK and TFS will know where to get the workflow assemblies it needs for the build process. Again, this approach is how you manage the assemblies if they are NOT installed in the GAC. You can use this source control location to house all of your custom workflow assemblies.

TFS Build Templates

Build templates can be located just about anywhere in source control but I would recommend creating a folder right next to the one you used for the custom assembly as shown below.
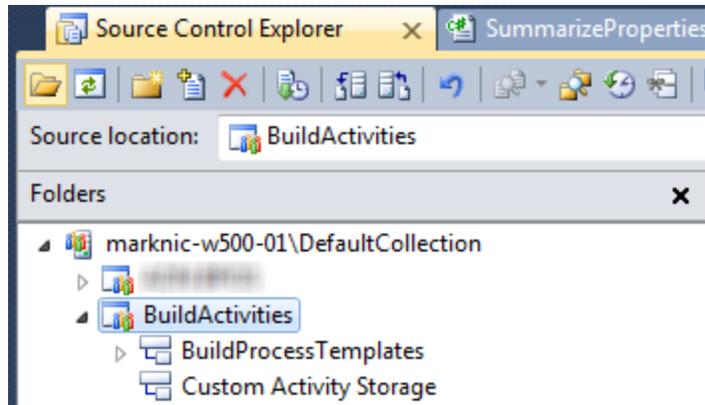
I created a Team Project devoted to managing custom build templates and their required assemblies. This way they are located centrally and all projects that need the custom build processes can easily get to and use them.

The build templates included with NuGetter are derived directly from "DefaultTemplate.xaml" so they contain the same build/compile functionality as DefaultTemplate. The difference is (with one exception) just the addition of packaging and versioning functionality. The exception is **described below** and is necessary to manage the building of multiple solutions.

NuGetterStandardBuildTemplate.xaml

This template will perform all the NuGet packaging actions.

NuGetterVersioningBuildTemplate.xaml

This template does all of the same packaging actions as NuGetterStandardBuildTemplate.xaml but also employs TfsVersioning activities to manage the versioning of the assemblies associated with the build.

NuGetterVersioningBuildTemplate15.xaml

This template is similar to NuGetterVersioningBuildTemplate but takes advantage of the extended versioning properties of version 1.5 of the **TfsVersioning** build extensions. **See the documentation here**.

Modifications to DefaultTemplate.xaml

The standard template for building solutions in TFS has a side effect that needed to be averted. Specifically, when multiple solutions are built, the drop folder is the destination of all solutions. This means that if you are building multiple solutions that generate commonly named assemblies or other files then each is overwritten as each solution is built and copied to the drop folder. The situation of having multiple, commonly named assemblies is a real possibility when creating a NuGet package for multiple .NET frameworks.

To avoid this situation a small modification was made to the build workflow that will create a separate folder in the drop for each solution that is built and it is named after the solution. For example, assume that I am creating a NuGet package that will provide assemblies (and other files) for .NET 3.5 and 4.0. Now, I have created the 2 solutions aptly named MarkNicNuGetLib35.sln and MarkNicNuGetLib40.sln and in my build definition, I have indicated that both solutions should be built when the build definition is triggered. When the solutions are built and you open the drop folder location, you will see the following:
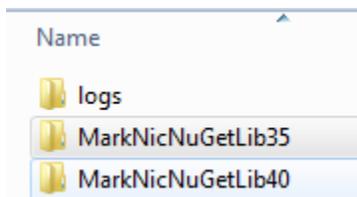
In each folder will be all of the application files for each of the solutions.

This does mean that even if you are only working with a single solution, there will be a folder named after it with all of the application files in it. This is actually a good thing as you will see below in packaging.

Each of the build templates provided in the NuGetter project contains this updated folder output.

User Guide

"Flexibility is the Linchpin of Confusion"

NuGetter has been designed for flexibility and as such it has a lot of options for managing the various parameters and information that it requires to build the NuGet package that you desire. Many parameters have default values but no matter how hard I try, I know I won't be able to create a default setup that will work for a broad audience.

Base concepts for managing build definition parameter data

1. The build definition has been organized into multiple data categories for each of the logical steps that are performed in the build, packaging and deployment steps.
   a. **"Build Versioning"**: This section will appear if you are using the related project TfsVersioning to manage the versioning of your assemblies.
   b. **"NuGetter (A) – Pre-Packaging"**: If you need to manipulate the application files after the compilation step then you can use "pre-packaging" to invoke a PowerShell script and organize the files so that NuGet can easily perform the packaging.
   c. **"NuGetter (B) – Package"**: This section has all the parameters necessary to define how the packaging process will occur including where to find the nuget.exe application, the base path or package source file location, the output location for the package and, if you wish, the version number that should be used when creating the package.
   d. **"NuGetter (C) – Push and Publish"**: Here you can define if and where the package will be deployed ("pushed"). Also, when working with a NuGet gallery that supports the capability, you can define whether or not to publish the package (i.e., make the package public). Also, if necessary, an API key can be provided here to identify the author of the package.
2. There are items in the build definition that are simple value-based parameters and others where the necessary value for the parameter can come from various locations. For example, the "Invoke PowerShell Script" parameter is a simple True or False value to indicate if a script should be invoked. On the other hand, the "PowerShell Script File Path" parameter can either be a relative or source code control path to the script to use in pre-packaging.

General Rules:

- NuGetter assumes that you have an appropriate NuSpec file already created and in source control.
- If a "File Path" is being identified as a parameter in the build definition then it can either be a relative path OR an absolute file path.
- When an absolute file path is an option it can either be a machine-based file path OR a source code control file path. Use of the machine-based path should be avoided because it is very inflexible and assumes the build machine is configured a certain way. Use of the absolute source code control path is much more flexible and is described below.
- Relative paths begin at the "Sources" folder on the build machine so you can assume that where ever you point the Workspace Source Control Folder at in the build, this is where the relative path begins. E.g., if I want to point at a PowerShell script within a folder named "NuGetPackageSupport" (which is a first level subfolder off of the workspace that I have identified for the build) I can just enter the following for the "PowerShell Script File Path": **NuGetPackageSupport/MarkNicNuGetLibPackage.ps1**
- The absolute source code control path will begin with "$/". If you right click on a file and look at its properties, the "Server Name:" is what you would provide in the build parameter for that file. When you reference a file this way the build process will copy it from source control on to the build machine and then will use it from there. For example: if I want to point at a PowerShell script file in source control I might use the following for the "PowerShell Script File Path": **$/Test Project/MarkNicTestLib/NuGetPackageSupport/MarkNicNuGetLibPackage.ps1**
- There are parameters that can be values or they can be relative or absolute file path to a file that contains the value. For example, for the "API Key or File Path" parameter you can enter the actual API Key value in the build definition or you can enter the relative or source code control path to a file that contains the key value. This gives you the flexibility of entering the value or storing it in a file where it can be kept safe and changed when necessary (without needing to change the build definition).

Versioning and the Version "Seed" File

Versioning within the standard NuGet process can be done in one of two ways. The first is to modify/enter the appropriate version number (in a Major.Minor.Build.Reference format) into the NuSpec file. When the packaging process executes, the version number in the NuSpec file is used and the package name will reflect it as will other entries inside of the NuPkg file.

The second way is to include a "-Version" argument followed by a version number. This approach will override the version in the NuSpec file and will version the package appropriately.

As you will see below, there is a "Version or Version Seed File" parameter as part of the build definition. This parameter uses the "-Version" approach and let you dynamically version the package. You can enter a version number directly in the parameter or for more flexibility; you can enter a file path to an XML file containing the version information for the package. That file is called the Version Seed File because it contains the "seeds" or patterns that are to be used in the versioning process.

The XML file format is shown below. It allows the management of version numbers across multiple packages and, by the way, it can also be used to manage the versions of your assemblies (see **TfsVersioning project on Codeplex**).

Seed File Layout/Schema:

```xml
<VersionSeed>
  <!-- ================================================================================================= -->
  <!-- Options:                                                                                          -->
  <!--   Explicit Versions:  1.2.3.4 - Major.Minor.Build.Revision   Example Output Version: 1.2.3.4      -->
  <!--   Partially Explicit Versions:  1.2.J.B - Major.Minor.JulianDate.TfsBuildNo  Example Output: 1.2.11059.5 -->
  <!--   Date Based Versions:  YYYY.M.D.B - Year.Month.Day.TfsBuildNumber   Example Output Version: 2011.2.28.5 -->
  <!--                                                                                                   -->
  <!-- Replacement Pattern Symbols:                                                                      -->
  <!--   YYYY : Full current year                                                                        -->
  <!--     YY : Current year (2 digit)                                                                   -->
  <!--      M : Current month                                                                            -->
  <!--      D : Current day of the month                                                                -->
  <!--      J : Current date in "Julian" format.  Example: 11027 = January 27, 2011, 11278 = October 5, 2011    -->
  <!--      B : TFS Build Number (extracted from the build process)                                      -->
  <!--                                                                                                   -->
  <!-- If Solution name is "Default" then those patterns will be used if an exact name match is not found.    -->
  <!--                                                                                                   -->
  <!-- ================================================================================================= -->
  <Solution name="MarkNicNuGetLib">
    <AssemblyVersionPattern>7.6.5.4</AssemblyVersionPattern>
    <AssemblyFileVersionPattern>7.6.j.b</AssemblyFileVersionPattern>
  </Solution>
  <Solution name="Default">
    <AssemblyVersionPattern>1.0.2.0</AssemblyVersionPattern>
    <AssemblyFileVersionPattern>1.0.j.b</AssemblyFileVersionPattern>
  </Solution>
  <NuGetPackage id="MarkNicNuGetLib">
      <VersionPattern>7.6.5.4</VersionPattern>
  </NuGetPackage>
</VersionSeed>
```

As you can see above, there is versioning support for solutions ("Solution" element) and NuGet packaging ("NuGetPackage" element).

Using the version seed file you can:

- Version the NuPkg package directly: By including a NuGetPackage element (with the "id" attribute set to the same value as the "id" element in the NuSpec file), the packaging process with use the "VersionPattern" to set the version number of the package

```xml
<VersionSeed>
  <NuGetPackage id="MarkNicNuGetLib">
      <VersionPattern>7.6.5.4</VersionPattern>
  </NuGetPackage>
</VersionSeed>
```

- Version the NuPkg package the same as any solution assemblies:
    - Using NuGetter in conjunction with **TfsVersioning**, you can set the Solution and the NuGetPackage versions to the same value.

```xml
<VersionSeed>
  <Solution name="MarkNicNuGetLib">
    <AssemblyVersionPattern>7.6.5.4</AssemblyVersionPattern>
    <AssemblyFileVersionPattern>7.6.j.b</AssemblyFileVersionPattern>
  </Solution>
  <NuGetPackage id="MarkNicNuGetLib">
      <VersionPattern>7.6.5.4</VersionPattern>
  </NuGetPackage>
</VersionSeed>
```

    - Or, even easier, you can remove the NuGetPackage entry. NuGetter will look for the value in NuGetPackage and if it doesn't find it, it will then look for a Solution with the same name. If found, it will then use the version pattern in the "AssemblyVersionPattern" element.

```xml
<VersionSeed>
  <Solution name="MarkNicNuGetLib">
    <AssemblyVersionPattern>7.6.5.4</AssemblyVersionPattern>
    <AssemblyFileVersionPattern>7.6.j.b</AssemblyFileVersionPattern>
  </Solution>
</VersionSeed>
```

Not only can you enter an actual version number, you can use "patterns" and the version number will be generated at the time of the build.

Version Patterns:

Version patterns allow you to dynamically generate version numbers based on numbers that may be different every time the build takes place. For example, you may want the version to indicate the date that it was built or you may want to manually set some of the version and have one number increment each time it is built. The versioning in NuGetter and TfsVersioning both work with version patterns. If you want to use one or more of the version patterns, just use one of the valid pattern symbols and during the build the symbol will be replaced with the desired number.

- If a number is used in any position in the version pattern then that number is passed through unchanged
- Use a symbol pattern and that value will be replaced in the AssemblyInfo file. The symbols are:

- o YYYY: Replaced with the current 4-digit year
- o YY: Replaced with the current 2-digit year
- o M or MM: Replaced with the number for the current month (MM <u>does not</u> give you a leading 0)
- o D or DD: Replaced with the number for the current day (DD <u>does not</u> give you a leading 0)
- o J: Replace with the current date in "Julian" 5-digit format (YYDDD where YY is the year and DDD is the number of the day within the year e.g., 11075 is March 16, 2011 – there are leading 0's for the day)
- o B: Replace with the current build number for the day. Note, using this pattern requires that the "Build Number Format" ends in "$(Rev:.r)". TFS does create the build number format with this "macro" at the end as the default so unless you change it there won't be a problem.

2. Basic	
Automated Tests	Run tests in assemblies matching ***test*.dll using settings
Build Number Format	$(BuildDefinitionName)_$(Date:yyyy.MM.dd)$(Rev:.r)
Clean Workspace	All
Logging Verbosity	Normal
Perform Code Analysis	AsConfigured
Source And Symbol Server Settings	Index Sources

Examples:

"yyyy.mm.dd.b" - If you queued up the 2nd build of the day on April 26, 2011 the version would be: "**2011.4.26.2**"

"1.0.J.B" – Again, if you queued up the 2nd build of the day on April 26, 2011 the version would be: "**1.0.11116.2**" (This is the default for the assembly file version)

Tips/Tricks

The TFS build number increments each time you build. It generally resets every day because the "build number format" contains the date and when the date changes, the "$Rev:.r" portion of the build number resets back to one. If you want to use the build number as an ever increasing number then change the rest of the build number format to a value that doesn't change – just make sure the "$Rev:.r" portion of the number format remains at the end. Now, if you put "b" in the last position of the version pattern, you will get a number that increases by 1 for every build.

Build Definition Parameters

NuGetter (A) – Pre-Packaging

Parameter	Description
Invoke PowerShell Script	True/False – Tells the build process to invoke the PowerShell script identified in the "PowerShell Script File Path" parameter. Default: **False**
PowerShell Script File Path	File path to the PowerShell script to run prior to packaging with NuGet. This script option is designed to provide you with the capability to organize project files for packaging. Default: **No File Path**

NuGetter (B) – Package

Parameter	Description
Additional NuGet Command Line Options	Enter any NuGet options desired that are NOT part of the existing NuGetter parameters. For example, "-Exclude" or "-NoDefaultExcludes". The text entered in this parameter is appended, as is, to the command line sent to NuGet. Default: **No Options**
Base Path	The name of the folder containing the files to be used by NuGet to do the packaging. This is the "BasePath" parameter as defined by the NuGet command line interface. If you use a PowerShell prepackaging script, this is where you should place/organize the files for packaging. Default: "**NuGetPrePackage**"
NuGet.Exe File Path	No value in this parameter indicates that the NuGet.exe application exists in the application "Path" on the build machine. A value here should indicate the file path where the NuGet.exe application file can be found during the build. The path can be relative or absolute and more information can be found in the General Rules. Default: **No file path provided**
NuSpec File Path	The file path for the application's "NuSpec" file. This build assumes that a NuSpec file physically exists before the build begins. A value is required in this parameter before the NuGet packaging will execute. Default: **No file path provided**
Output Directory	This is where NuGet.exe will place the "NuPkg" file after performing the packaging process. This is the same as the "OutputDirectory" parameter in the NuGet.exe application. Default: "**NuGetPackage**"

Version or Version Seed File Path	A version value here will indicate to the process that the value should be used to override the version value in the NuSpec file. A file path value here indicates that the version should be extracted from a "Version Seed File". For more information see the Version Seed File description in this document. Note: If you are using the TfsVersioning extensions, the same version seed file can be used here. Default: **No value provided**

NuGetter (C) - Push and Publish

Parameter	Description
API Key or File Path	API Key value for the NuGet Gallery or File Path to a file containing the API Key. There is no required file format except that it needs to be a text file. Other text can exist in the file and as long as the API key exists in the file, it should be found and extracted. Default: **No value provided**
Create Only – Do Not Publish	True/False – This is the same as the CreateOnly NuGet command line parameter. It indicates if the NuPkg file should be published as well as pushed to the gallery. A value of "True" means that the package should only be copied (not published) to the gallery. Default: **True**
Invoke Push Switch	True/False – Indicates if the push step should be attempted. Default: **False**
Source (Push Destination)	Location to "push" the newly created NuPkg file. This can be: • A URL (such as the NuGet Gallery) • Network Share (UNC Address) • Local drive location This parameter is similar to the NuGet "Source" command line parameter but NuGetter will also copy to non-URL locations. This is especially handy when deploying to test server locations for package verification before deploying to production. Default: **No value provided**

Build Versioning

If you use a build template that contains a section named "Build Versioning" then the TfsVersioning build extensions are also included. This build extension allows you to manage the versions of .NET solution assemblies. For detailed instructions please refer to the **documentation** in the **TfsVersioning** site.

"Pre-Packaging" and the Use of PowerShell

Pre-Packaging is a term used to describe the process or steps taken to organize the project files in such a way so that the packaging process can happen or is made easier to execute. Using a PowerShell script also lets you test the file organization inside or out of the build process.

Because of the way that projects and solutions are compiled, the files may be (and are probably) organized differently than what NuGet needs to generate a package. Some of this can be managed in the NuSpec file but it may not be flexible enough for your situation. Or, you may not want to keep changing the "Files" section for your project.

NuGetter has the ability to call a PowerShell script to create folders, move files and generally organize things so that it is easy to see what you are packaging (in the drop folder) and easy for NuGet to grab what you want and perform the packaging.

To make things even easier, NuGetter provides build-time folder location information so that you can create a script that adapts to the current build and the current build agent machine and drop location. Three folder locations are provided via PowerShell variables:

- $tfsDropFolder: Path to the folder where TFS copies your application
- $tfsSourcesFolder: Path to all of the source files used to generate the application. This is the workspace files. There may be cases where you have files in source control that are not part of the solution that need to be placed in the NuGet package. With this variable, you can access and copy the files where you need them.
- $tfsBinariesFolder: Path to the binaries folder on the build machine
- $tfsNuGetPrePackageFolder: This is the value of the relative path entered into the build definition. With this value, you always know where the files should be organized. This value is later used by the NuGet application as the "Base Path" or source for the package process.

These variables are automatically passed into the invocation of your PowerShell script so you can use them at any point.

Sample Project Approaches

I will go through the steps for creating a build for the following types of projects. Hopefully, it will help indicate how to best use NuGetter for your particular situation.

- Simple Project – Single Solution/single assembly
- Complex Project – Multiple Solution/Multiple Framework

Simple Project

Situation:

- A simple, single solution, single assembly project
- Name of the solution: "MyWebFixifier"
- NuGet.exe exists in source code control at "$/BuildActivities/NuGet Exe/NuGet.exe"
- Test "Package Source" location is: "\\localhost\Local NuGet"
- The NuSpec file location is: "$/NuGet Projects/ MyWebFixifier /NuGetPackageSupport/ MyWebFixifier.nuspec"
- The package should be versioned '3.4.5.6"

TFS Build Process Template Settings:

5. NuGetter (A) - PrePackaging	
Invoke PowerShell Script	False
PowerShell Script File Path	
6. NuGetter (B) - Package	
Additional NuGet Command Line Options	
Base Path	**MyWebFixifier**
NuGet.Exe File Path	
NuSpec File Path	**NuGetPackageSupport\MyWebFixifier.nuspec**
Output Directory	NuGetPackage
Version or Version Seed File Path	**3.4.5.6**
7. NuGetter (C) - Push and Publish	
Api Key or File Path	
Create Only - Do Not Publish	True
Invoke Push Switch	**True**
Source (Push Destination)	**\\localhost\Local NuGet**

Explanation:

- The simplicity of this project does not require the use of a PowerShell script to organize the files so nothing needs to be changed in (A) PrePackaging.

- The base path was changed to the name of the solution: MyWebFixifier
- The NuSpec file exists within the workspace and as such will be copied to the "Sources" build folder so a relative path was used. You could also use the absolute source code control path of "$/NuGet Projects/ MyWebFixifier /NuGetPackageSupport/ MyWebFixifier.nuspec". Either will work.
- The Output Directory was left to the default "NuGetPackage"
- The version number "3.4.5.6" was entered directly rather than using a seed file. Although, a seed file could have been used.
- The destination of the deployment was "\\localhost\Local NuGet" (a local test location for NuGet packages)
- Since we want to push the newly created package, the "Invoke Push Switch" was set to "True"
- The local test server location does not require an API Key or Publishing

Complex Project

Situation:

- NuGet project that will target .NET 3.5 and .NET 4.0 projects with separate assemblies
- Each assembly should be created through a separate solution
- The assemblies need to be versioned (during the build) separately from the NuGet package
- The NuGet package should include an incrementing number as the last value in the version
- Versioning should be managed through a file so that different builds can version similarly
- In addition to the assemblies, the package will contain a class and will also install a reference
- The NuGet.exe application will be stored in source code control
- All of the support files (other than NuGet.exe) will be part of the workspace
- Need to deploy a successful package to a test server (network share)

TFS Build Process Template Settings:

Build process template:

NuGetterVersioningBuildTemplate.xaml ← A. NuGetter build template

Build process parameters:

B. 2 Solutions being built

◢ **1. Required**	
▷ Items to Build	**Build 2 projects with default platforms and configurations**
◢ **2. Basic**	
▷ Automated Tests	Run tests
Build Number Format	**MarkNicTestLib Incrementing Build$(Rev:.r)**
Clean Workspace	All
Logging Verbosity	Normal
Perform Code Analysis	AsConfigured
▷ Source And Symbol Server Settings	Index Sources
▷ **3. Advanced**	
◢ **4. Build Versioning**	
Assembly File Version Pattern	**2.3.J.B**
Assembly Version Pattern	**2.3.4.5**
AssemblyInfo File Pattern	AssemblyInfo.*
Build Number Prefix	0
Force Create Version	True
Perform Check-in of the AssemblyInfo Files	True
Use Version Seed File	**True**
Version Seed File Path	TfsVersion\VersionSeed.xml
◢ **5. NuGetter (A) - PrePackaging**	
Invoke PowerShell Script	**True**
PowerShell Script File Path	**NuGetPackageSupport\MarkNicNuGetLibPackage.ps1**
◢ **6. NuGetter (B) - Package**	
Additional NuGet Command Line Options	
Base Path	NuGetPrePackage
NuGet.Exe File Path	**$/BuildActivities/NuGet/NuGet.exe**
NuSpec File Path	**NuGetPackageSupport\MarkNicNuGetLib.nuspec**
Output Directory	NuGetPackage
Version or Version Seed File Path	**TfsVersion\VersionSeed.xml**
◢ **7. NuGetter (C) - Push and Publish**	
Api Key or File Path	
Create Only - Do Not Publish	True
Invoke Push Switch	**True**
Source (Push Destination)	**\\marknic-w500-01\Local NuGet**

C. Ever increasing build number

D. Assemblies versioned using a remote "seed" file

E. PowerShell "PrePackaging"

F. NuGet.exe NOT on build machine

G. Package versioned remotely

H. Auto-deploy to test server

Explanation:

- The NuGetterVersioningBuildTemplate is used to perform the build. **(A)**
- In the "Items to Build" two solutions "MarkNicNuGetLib35.sln" and "MarkNicNuGetLib40.sln" are identified to be built. The modified build template will place the assemblies in folders named after the solutions. **(B)**
- TfsVersioning is used to automate the versioning of the assemblies using a seed file **(D)**
- Since the package will support two frameworks, the files need to be organized in the package. To facilitate this, the PowerShell script "MarkNicNuGetLibPackage.ps1" (shown below) will be called before the packaging. **(E)** The files will be organized as shown here:

```
NuGetPrePackage \ content \ models \ *.cs.pp
                \ lib \ net35 \ *.dll
                \ lib \ net40 \ *.dll
                \ tools \ *.ps1
```

- Don't have admin access to the build server. Because of this, NuGet.exe must be retrieved from source control at the time of build. **(F)**
- A version seed file "VersionSeed.xml" contains the versioning for the assemblies and the NuGet package. **(G)**

```xml
<VersionSeed>
  <Solution name="MarkNicNuGetLib40">
    <AssemblyVersionPattern>1.3.2.0</AssemblyVersionPattern>
    <AssemblyFileVersionPattern>1.3.j.b</AssemblyFileVersionPattern>
  </Solution>
  <Solution name="MarkNicNuGetLib35">
    <AssemblyVersionPattern>1.3.2.0</AssemblyVersionPattern>
    <AssemblyFileVersionPattern>1.3.j.b</AssemblyFileVersionPattern>
  </Solution>
  <NuGetPackage id="MarkNicNuGetLib">
    <VersionPattern>7.6.5.b</VersionPattern>
  </NuGetPackage>
</VersionSeed>
```

- The version number for the package needs to include an ever-increasing value (in the last version position). This way there will always be a new version number for each build. If you look at the pattern being used for the package: `<VersionPattern>7.6.5.b</VersionPattern>` this will grab the "build number" from TFS provide through the "Build Number Format". That format was modified so that the build number will not reset every day. **(C)**
- Once a successful build and package occurs, the newly created package will be deployed to an internal network share location for immediate testing. **(H)**

PowerShell Script Used in "PrePackaging"

The script below was used in the example code. It creates a folder structure and then copies the appropriate files into place so that the NuGet packaging can occur without informing the NuSpec (manifest) file where all of the files are.

```powershell
# Calculate where the files will be copied for the NuGet Packaging process
if ([IO.Path]::IsPathRooted($nuGetPackageSourceFolder))
{
	$nugetPrePackageFolder = $tfsNuGetPrePackageFolder
}
else
{
	$nugetPrePackageFolder = Join-Path $tfsDropFolder $tfsNuGetPrePackageFolder
}

# Create some variables that will be used to create the package structure
$libFolder = "lib"
$contentFolder = "content"
$toolsFolder = "tools"

$net40 = "net40"
$net35 = "net35"

# Function to create a subfolder with some error checking and validation
Function Create-FrameworkFolder
{
	Param([string]$rootPath = $(throw "$rootPath required."), [string]$subFolder)

	if ([System.String]::IsNullOrEmpty($subFolder))
	{
		$folderToCreate = $rootPath
	}
	else
	{
		$folderToCreate = Join-Path $rootPath $subFolder
	}

    if (![IO.Directory]::Exists($folderToCreate))
	{
		New-Item $folderToCreate -ItemType directory
	}
}

# Structure to Create:
# NuGetPrePackage
#               \ content
#                        \ models
#               \ lib
#                        \ net35
#                        \ net40
#               \ tools

Create-FrameworkFolder -rootPath $nugetPrePackageFolder
Create-FrameworkFolder -rootPath $nugetPrePackageFolder -subFolder $contentFolder
Create-FrameworkFolder -rootPath $nugetPrePackageFolder -subFolder $libFolder
Create-FrameworkFolder -rootPath $nugetPrePackageFolder -subFolder $toolsFolder
```

```powershell
$prePackageLibFolder = Join-Path $nugetPrePackageFolder $libFolder
Create-FrameworkFolder -rootPath $prePackageLibFolder -subFolder $net35
Create-FrameworkFolder -rootPath $prePackageLibFolder -subFolder $net40


# Identify the source location(s) for the files that were built as part of the
# TFS Build Process
$net35Folder = Join-Path $tfsDropFolder "MarkNicNuGetLib35"
$net40Folder = Join-Path $tfsDropFolder "MarkNicNuGetLib40"

# Copy all the files into position so NuGet can do the packaging
$dest = Join-Path $prePackageLibFolder $net35
Copy-Item "$net35Folder\*.dll" -Destination $dest

$dest = Join-Path $prePackageLibFolder $net40
Copy-Item "$net40Folder\*.dll" -Destination $dest

$dest = Join-Path $nugetPrePackageFolder $toolsFolder
Copy-Item "$net40Folder\tools\*.ps1" -Destination $dest

$dest = Join-Path $nugetPrePackageFolder $contentFolder
Copy-Item "$net40Folder\content\models" -Destination $dest -recurse
```