**Demo - Creating a NuGet Package**

**Overview**

To create a package, you first create the code for the package. If you intend to include assemblies or executables in the package, you compile the code. You then organize files that need to be installed in target project into a folder structure that determines what NuGet does with the files when it installs the package. Finally, you create a *.nuspec* file that acts as the manifest for your package. (You might also create transformation files that can update the *app.config*, *web.config*, and project source-code files during package installation.)

After package files are in the proper folders and the *.nuspec* file has been created, you run the [*nuget.exe*](http://nuget.codeplex.com/releases/57303/download/197743) command-line utility and pass it the name of the *.nuspec* file. The following example shows a *nuget.exe* command that creates a NuGet package:

nuget pack xunit.nuspec

The pack command is followed by the name of the *.nuspec* file, an optional parameter for the base folder that contains the files to include in the package, and an optional parameter for the folder to create the package in. When you execute this command, a file named *xunit.1.6.1.nupkg* is created in the *xunit-nuget* folder, and the console displays the following message:

Successfully created package 'c:\xunit-nuget\xunit.1.6.1.nupkg

The *nuget.exe* command-line utility is a wrapper for the **PackageBuilder** API used to create packages. Therefore, as an alternative you can use the API directly to create packages.

**Package Conventions**

There are two types of conventions that apply when creating packages. The conventions listed in this page are **enforced conventions**. These are conventions that are required to follow when building packages. There are also **community (or optional) conventions**, which are conventions that have been formed by the community which make it easier for others to understand what your package is all about and make use of it immediately. We’ve just started [**documenting those package conventions**](http://nuget.codeplex.com/wikipage?title=Package%20Conventions)and will continue to update them as new ones appear.

**Package Folder Structure**

By default, all of the files and folders that are in the specified base folder are included recursively unless the *.nuspec* file includes a list of files to explicitly include. If no base folder is specified, the default is the folder that contains the *.nuspec* file.

A package consists of three types of files: tools, assemblies, and content. The type (or function) of a file is determined by where in the   
package folder structure it is located. This in turn determines what NuGet does with the file during package installation. (File types are not specified in the *.nuspec* file.)

* Tools – The *tools* folder of a package is for programs that you want to be accessible from the command line. After the folder is copied to the target project, it is added to the $env:Path (PATH) environment variable.
* Assemblies (*.dll* files) in the *lib* folder become assembly references when the package is installed.
* Content – Files within the *content* folder are copied to the application root when the package is installed.

**Supporting Multiple .NET Framework Versions and Profiles**

Many libraries target a specific version of the .NET Framework. For example, you might have one version of your library that’s specific to Silverlight, and another version of the same library that takes advantage of .NET Framework 4 features. You do not need to create separate packages for each of these versions. NuGet supports putting multiple versions of the same library in a single package keeping them in separate folders within the package.

When NuGet installs an assembly from a package, it checks the target .NET Framework version of the project you are adding the package to. NuGet then selects the correct version of the assembly in the package by selecting the correct subfolder within the *lib* folder. To enable NuGet to do this, you use the following naming convention to indicate which assemblies go with which framework versions:

lib\{framework name}{version}

The following example shows a folder structure that supports four versions of a library:

\lib

\.NetFramework 1.1

\MyAssembly.dll

\.NetFramework 2.0

\MyAssembly.dll

\.NetFramework 4.0

\MyAssembly.dll

\Silverlight 4.0

\MyAssembly.dll

NuGet attempts to parse the folder name into a [FrameworkName](http://msdn.microsoft.com/en-us/library/dd414023.aspx) object. Names are case insensitive, and you can use abbreviations for both framework name and version number. If you omit the framework name, the .NET Framework is assumed. For example, the following folder structure is equivalent to the previous one:

\lib

\Net 1.1

\MyAssembly.dll

\2

\MyAssembly.dll

\.Net 4

\MyAssembly.dll

\sl4

\MyAssembly.dll

Assemblies that have no associated framework name or version are stored directly in the *lib* folder.

When NuGet installs a package that has multiple assembly versions, it tries to match the framework name of the assembly with the target framework of the project. If a match is not found, NuGet copies the assembly that's for the highest version that is less than or equal to the project's target framework. For example, if you install a package that has the *lib* folder structure shown in the previous example in a project that targets the .NET Framework 3.5, the assembly in the *2* folder (for .NET Framework 2.0) is selected.

**Grouping Assemblies by Framework Version**

NuGet copies assemblies from only a single library folder. For example, suppose a package has the following folder structure:

\lib

\Net20

\MyAssembly.dll (v1.0)

\MyAssembly.Core.dll (v1.0)

\Net40

\MyAssembly.dll (v2.0)

When the package is installed in a project that targets the .NET Framework 4, *MyAssembly.dll (v2.0)* is the only assembly installed. *MyAssembly.Core.dll (v1.0)* is not installed. (One reason why NuGet behaves this way is that *MyAssembly.Core* might have been merged into version 2.0 of *MyAssembly*.) In this example, if you want *MyAssembly.Core.dll* to be installed in a project that targets the .NET Framework 4, you must include it in the *Net40* folder as well as in the *Net20* folder.

The rule about copying assemblies from only one folder also applies to the root *lib* folder. Suppose a package has the following folder structure:

\lib

\MyAssembly.dll (v1.0)

\MyAssembly.Core.dll (v1.0)

\Net40

\MyAssembly.dll (v2.0)

In projects that target the .NET Framework 2.0 and the .NET Framework 3.5, NuGet copies both *MyAssembly.dll* and *MyAssembly.Core.dll*. But as was true of the previous example, in projects that target the .NET Framework 4, only *MyAssembly.dll* from the *Net40* folder will be copied. As in the previous example, if you want *MyAssembly.Core.dll* to be installed in a project that targets the .NET Framework 4, you must include it in the *Net40* folder.

**The .nuspec Manifest File**

After files are arranged in a package folder structure, you create the *.nuspec* XML file that provides metadata for the package. The following example shows a simple *.nuspec* file:

<?xml version="1.0" encoding="utf-8"?>

<package>

<metadata>

<id>sample</id>

<version>1.2.3</version>

<authors>Kim Abercrombie, Franck Halmaert</authors>

<description>Sample is an example package that exists only to show a sample .nuspec file.</description>

<language>en-US</language>

<licenseUrl>http://sample.codeplex.com/license</licenseUrl>  
 <projectUrl>http://sample.codeplex.com/</projectUrl>

</metadata>

</package>

For information about the format of the *.nuspec* file, see [Nuspec Format](http://nuget.codeplex.com/documentation?title=Nuspec%20Format).

**Configuration File and Source File Transformations**

In general, when you create a package, the files that you include in the package are not modified in any way and are simply copied to the appropriate location in the target solution. However, in some cases you might want a file to be modified or transformed during installation. NuGet supports two types of transformations during installation of a package:

* You can specify changes to make in the project’s *app.config* or *web.config* file by including a file named *web.config.transform* or *app.config.transform*. For example, you might want to add an item to the httpModules collection in the configuration file.
* If your package copies source code files to the project, you can append *.pp* to a source code file name and include in the file variables that are replaced with values appropriate to the project during installation. For example, if you put namespace $rootnamespace$ in a source-code file and append *.pp* to the name, the placeholder becomes namespace TargetProject when the code is installed in a target project whose root namespace is TargetProject. In the target project, the file no longer has the *.pp* extension.

For more information, see [Configuration-File and Source-Code Transformations](http://nuget.codeplex.com/wikipage?title=Config%20File%20and%20Source%20Code%20Transformations).

**Automatically Running PowerShell Scripts During Package Installation and Removal**

A package can include PowerShell scripts that automatically run when the package is installed or removed. NuGet automatically runs scripts based on their file names using the following conventions:

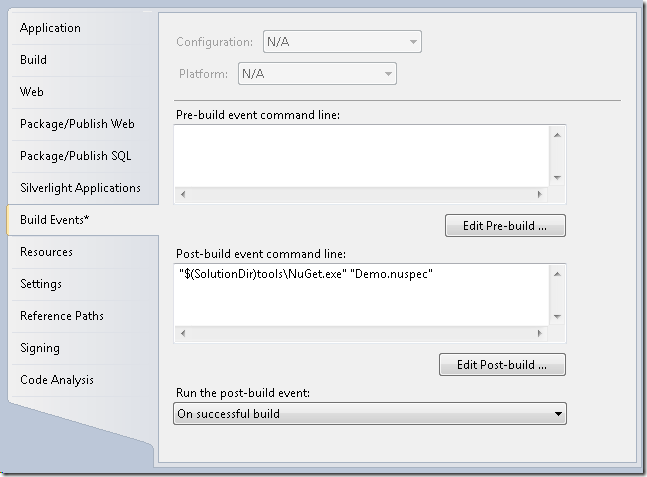
* *Init.ps1* runs the first time a package is installed in a solution. If the same package is installed into additional projects in the solution, the script is not run during those installations. The script also runs every time the solution is opened. For example, if you install a package, close Visual Studio, and then start Visual Studio and open the solution, the *Init.ps1* script runs again.
* *Install.ps1* runs when a package is installed in a project. If the same package is installed in multiple projects in a solution, the script runs each time the package is installed. If a package is not installed into a project (such as the MvcScaffold package), the script runs when the package is installed into the solution. **Note:** The package must have content/dll that will be added to the project for *Install.ps1* to run. Just having something in the tools folder will not kick this off.
* *Uninstall.ps1* runs every time a package is uninstalled.
* These files should be located in the tools directory of your package.
* At the top of your file, add this line: *param($installPath, $toolsPath, $package, $project)*
  + *$installPath* is the path to your package install
  + *$toolsPath* is the path to the tools directory under the package
  + *$package* is your package
  + *$project* is the project you are installing the application to. **Note:** This will be null in *Init.ps1*. It doesn't have a reference to a particular project because it runs at the solution level.
  + *$project.Object* is the equivalent of <http://msdn.microsoft.com/en-us/library/ms170626.aspx>.
* When you are testing *$project* in the console while creating your scripts, you can set it to *$project = Get-Project*

**Building a Package During the Visual Studio Build Process**

It is planned that in future releases tasks will be available for MsBuild, NAnt, and other tools for integrating the creation of NuGet packages into a build process. Until then, one way to integrate package creation is to add the *nuget.exe* file to a known directory in your solution and then invoke the executable file using your build script, such as an NAnt, Psake, or MsBuild script.

While the recommended approach is to integrate package build into your build or continuous-integration process, for demonstration purposes the following example shows how you can add a post-build event that makes the appropriate call to *nuget.exe* to create a package every time you compile.

To add a post-build event, in **Solution Explorer** right-click your project and then click **Properties**. Click the **Build Events** tab and add a **Post-build event command line** entry that points to *nuget.exe* within a *tools* folder in the root of your solution.

[](http://download.codeplex.com/Project/Download/FileDownload.aspx?ProjectName=nuget&DownloadId=187333)

Now every time you build your project, a NuGet package is included in the output.