APPENDIX A

Package Manifest (.nuspec) Reference

In previous chapters of this book, you've seen NuGet package manifests or `.nuspec` files being created a number of times. When creating packages, NuGet uses a *convention over configuration* approach. These conventions are rather simple by default but relying on them may not be enough to mold the package you wish to create. A NuGet package manifest, also known as a `.nuspec` file, describes the contents and metadata of your package. It is included in each NuGet package, or `.nupkg`, file, and it is the file you'll need to create the actual package. In short, before you can create a package, you'll need to describe it.

This appendix covers all details about available options in the XML of a package manifest. It is largely based on the information available on the official NuGet web site at `http://docs.nuget.org/docs/reference/nuspec-reference`. NuGet documentation (from the Outercurve Foundation) is licensed under Creative Commons license BY 3.0 (`http://creativecommons.org/licenses/by/3.0/`).

■ **Note** Editing `.nuspec` files in Visual Studio or any XML editor can be made easier using the XSD schema of the package manifest. This can be installed into a Visual Studio project using NuGet: `Install-Package NuGet.Manifest.Schema`.

Creating the Metadata Section

The package manifest metadata section is a mandatory section within the package manifest. The metadata section itself also has a set of mandatory elements. These required elements must be specified before you can create a package based on the `.nuspec` manifest. Here is the list of all required metadata fields:

- ID
- Version
- Description
- Authors

The first pieces of metadata your package exposes are the package ID and the package version. This pair of fields will uniquely identify your package on any NuGet feed. The package description and authors are used to search in the NuGet gallery.

Table A-1 lists the possible elements in the package manifest metadata section and their descriptions.

***Table A-1.** Elements in the Package Manifest Metadata Section*

Element	Required?	Description
`id`	Yes	The unique identifier for the package. This is the package name that is shown when packages are listed using the Package Manager Console and used when installing a package using the `Install-Package` command within the Package Manager Console. Package IDs may not contain any spaces or characters that are invalid in an URL.
`version`	Yes	The version of the package in a format like `1.2.3`.
`title`	No	The human-friendly title of the package displayed in the Manage NuGet Packages dialog. If none is specified, the ID is used instead.
`authors`	Yes	A comma-separated list of authors of the package code.
`owners`	No	A comma-separated list of the package creators. This is often the same list as in authors. This is ignored when uploading the package to the NuGet.org Gallery.
`description`	Yes	A long description of the package. This shows up in the right pane of the Add Package Dialog as well as in the Package Manager Console when listing packages using the `Get-Package` command.
`releaseNotes`	No	A description of the changes made in each release of the package. This field only shows up when the `_Updates_` tab is selected and the package is an update to a previously installed package. It is displayed where the description would normally be displayed.
`summary`	No	A short description of the package. If specified, this shows up in the middle pane of the Add Package Dialog. If not specified, a truncated version of the description is used instead.
`language`	No	The locale ID for the package, such as `en-us`.
`projectUrl`	No	A URL for the home page of the package.

Element	Required?	Description
`iconUrl`	No	A URL for the image to use as the icon for the package in the Manage NuGet Packages dialog box. This should be a 32 × 32–pixel `.png` file that has a transparent background.
`licenseUrl`	No	A link to the license that the package is under.
`copyright`	No	Copyright details for the package.
`requireLicenseAcceptance`	No	A Boolean value that specifies whether the client needs to ensure that the package license (described by `licenseUrl`) is accepted before the package is installed.
`dependencies`	No	The list of dependencies for the package.
`references`	No	Names of assemblies under `lib` that are added as project references. If unspecified, all references in `lib` are added as project references. When specifying a reference, only specify the name, not the path inside the package.
`frameworkAssemblies`	No	The list of .NET Framework assembly references that this package requires. These are references to assemblies that exist in the .NET Framework and thus should already be in the GAC for any machine. Specifying framework assembly references ensures these references are added when installing the package.
`tags`	No	A space-delimited list of tags and keywords that describe the package. This information is used to help make sure users can find the package using searches in the Add Package Reference dialog box or filtering in the Package Manager Console window.

Populating Replacement Tokens

NuGet package manifests can be generated based on either a compiled assembly or a project file, for example `MyProject.csproj`. NuGet populates some replacement tokens within the metadata section of a package manifest with the data defined in the project's assembly name, `AssemblyVersionAttribute`, `AssemblyCompanyAttribute`, and `AssemblyDescriptionAttribute`. The `MyProject.nuspec` file adjacent to the `MyProject.csproj` file may contain the replacement tokens listed in Table A-2, which are populated by the values within the project.

Table A-2. *Package Manifest Metadata Section Replacement Tokens*

Token	Source
`$id$`	The assembly name
`$version$`	The assembly version as specified in the assembly's `AssemblyVersionAttribute`
`$author$`	The company as specified in the `AssemblyCompanyAttribute`
`$description$`	The description as specified in the `AssemblyDescriptionAttribute`

Referencing Dependencies

If your NuGet package depends on other NuGet packages, the listing of these dependencies will be generated based on the `packages.config` file found in your project, if any. External assemblies referenced in your project will also be added to the project manifest based on the files contained in the from the `lib` folder. These conventions are overridden by adding a `<dependencies>` element for NuGet package references or a `<references>` element to add external assembly references.

Specifying Dependencies

The `dependencies` element is a child element of the `metadata` element and contains a set of `dependency` elements. Each dependency element is a reference to another package that this package depends on. When installing a package that contains a list of package dependencies through NuGet, these package dependencies will be downloaded and installed as well. The following is an example list of dependencies:

```
<dependencies>
  <dependency id="RouteMagic" version="1.1.0" />
  <dependency id="RouteDebugger" version="1.0.0" />
</dependencies>
```

NuGet supports using interval notation for specifying version ranges. This way, you can create a package that references another NuGet package based on a specific version, a range of versions, or any latest version of the package. Limiting the version ranges may limit dependency hell in your projects. The NuGet specification was inspired by the Maven Version Range Specification but is not identical to it. The following summarizes how to specify version ranges:

```
1.0 = 1.0 • x
(,1.0] = x • 1.0
(,1.0) = x < 1.0
[1.0] = x == 1.0
(1.0) = invalid
(1.0,) = 1.0 < x
(1.0,2.0) = 1.0 < x < 2.0
[1.0,2.0] = 1.0 • x • 2.0
empty = latest version
```

Specifying Explicit Assembly References

Use the `<references />` element to explicitly specify assemblies that the target project should reference.

For example, if you add the following, only the `xunit.dll` and `xunit.extensions.dll` will be referenced from the appropriate framework or profile subdirectory of the `lib` folder even if there are other assemblies in the folder:

```
<references>
    <reference file="xunit.dll" />
    <reference file="xunit.extensions.dll" />
</references>
```

If this element is omitted, the usual behavior applies, which is to reference every assembly in the `lib` folder.

Specifying Framework Assembly References from the GAC

In some cases, a package may depend on an assembly that's in the .NET Framework. For example, your package may depend on the Managed Extensibility Framework (MEF), which is a .NET framework assembly that should be added as a reference to a project explicitly. When specifying Framework assembly references, NuGet will explicitly add a reference to a framework assembly when installing your NuGet package.

The `<frameworkAssemblies>` element, a child element of the metadata element, allows you to specify a set of `frameworkAssembly` elements pointing to a framework assembly in the Global Assembly Cache (GAC). Note the emphasis on "framework assembly." These assemblies are not included in your package, because they are assumed to be on every machine as part of the .NET Framework.

```
<frameworkAssemblies>
    <frameworkAssembly assemblyName="System.ServiceModel" targetFramework="net40" />
    <frameworkAssembly assemblyName="System.SomethingElse"  />
</frameworkAssemblies>
```

Table A-3 lists all attributes of the `frameworkAssembly` element.

Table A-3. *Attributes for the framleworkAssembly Element*

Attribute	Required?	Description
`assemblyName`	Yes	The fully qualified assembly name.
`targetFramework`	No	If specified, the specific target framework that this reference applies to. For example, if a reference only applies to .NET 4, the value should be `"net40"`. If the reference applies to *all* frameworks, omit this attribute.

Specifying Files to Include in a Package

By convention, you do not have to explicitly specify a list of files in the `.nuspec` file. In some cases, however, it may be useful to explicitly list the files in your project that should be included in the NuGet package. Do note that if you specify any files, the conventions are ignored, and only the files listed in the package manifest are included in the package.

The `files` element is an optional child element of the package element and contains a set of `file` elements. Each `file` element specifies the source and destination of a file to include in the package via the `src` attribute and `target` attribute, respectively.

Table A-4 lists the possible attributes of the `file` element.

Table A-4. *Attributes for the file Element*

Attribute	Required?	Description
`src`	Yes	The location of the file or files to include. The path is relative to the `.nuspec` file unless an absolute path is specified. The wildcard character (*) is allowed. Using a double wildcard character (**) implies a recursive directory search.
`target`	Yes	This is a relative path to the directory within the package where the source files will be placed.
`exclude`	No	The file or files to exclude. This is usually combined with a wildcard value in the `src` attribute. The `exclude` attribute can contain a semicolon-delimited list of files or a file pattern. Using a double wildcard character (**) implies a recursive exclude pattern.

File Element Examples

This section describes some example usages of the `file` element to give you a better understanding of how it's used in NuGet. These use cases have been copied from the NuGet documentation at `docs.nuget.org` mostly, but we've added a few as well.

Single Assembly

Copy a single assembly in the same folder as the `.nuspec` file into the package's `lib` folder:

```
<file src="foo.dll" target="lib" />
```

The source contains `foo.dll`.

The packaged result is `lib\foo.dll`.

Single Assembly with a Deep Path

Copy a single assembly into the package's `lib\net40` folder so that it only applies to projects targeting the .NET 4 framework:

```
<file src="assemblies\net40\foo.dll" target="lib\net40" />
```

The source contains `foo.dll`.

The packaged result is `lib\net40\foo.dll`.

Set of DLLs

Copy a set of assemblies within the `bin\release` folder into the package's `lib` folder:

```
<file src="bin\release\*.dll" target="lib" />
```

The source contains one of these:

- `bin\releases\MyLib.dll`
- `bin\releases\CoolLib.dll`

The packaged result is one of these:

- `lib\MyLib.dll`
- `lib\CoolLib.dll`

Set of DLLs, Excluding a Specific DLL

Copy a set of assemblies within the bin\release folder into the package's lib folder, omitting one assembly:

```
<file src="bin\release\*.dll" target="lib" exclude="bin\release\CoolLib.dll" />
```

The source contains one of these:

- `bin\releases\MyLib.dll`
- `bin\releases\CoolLib.dll`

The packaged result is `lib\MyLib.dll`.

DLLs for Different Frameworks

Copy a set of assemblies compiled for various versions of the .NET Framework:

```
<file src="lib\**" target="lib" />
```

Note that the double wildcard character implies a recursive search in the source for matching files. The source contains one of these:

- `lib\net40\foo.dll`
- `lib\net20\foo.dll`

The packaged result is one of these:

- `lib\net40\foo.dll`
- `lib\net20\foo.dll`

Content Files

Copy content files from a source folder into a specific folder in a NuGet package:

```
<file src="css\mobile\*.css" target="content\css\mobile" />
```

The source contains one of these:

- `css\mobile\style1.css`
- `css\mobile\style2.css`

The packaged result is one of these:

- `content\css\mobile\style1.css`
- `content\css\mobile\style2.css`

Content Files with a Directory Structure

Recursively copy content files from a source folder into a NuGet package:

```
<file src="css\**\*.css" target="content\css" />
```

The source contains one of these:

- `css\mobile\style.css`
- `css\mobile\wp7\style.css`
- `css\browser\style.css`

The packaged result is one of these:

- `content\css\mobile\style.css`
- `content\css\mobile\wp7\style.css`
- `content\css\browser\style.css`

Content Files with a Deep Path

Copy content files from a source folder into a folder in a NuGet package which has a different folder hierarchy than the source files:

```
<file src="css\cool\style.css" target="Content" />
```

The source contains `css\cool\style.css`.
The packaged result is `content\style.css`.

Content Files Copied to a Folder with “Dot” in Its Name

Copy content files from a source folder into a folder in a NuGet package which has a “dot” in its name:

```
<file src="images\Neatpic.png" target="Content\images\foo.bar" />
```

Note that, because the `target` extension doesn't match the `src` extension, NuGet treats it as a directory.
The source contains `images\Neatpic.png`.
The packaged result is `content\images\foo.bar\Neatpick.png`.

Content File with a Deep Path and a Deep Target

Copy a content file from a source folder into a NuGet package can be done using either of the following two lines:

```
<file src="css\cool\style.css" target="Content\css\cool" />
<file src="css\cool\style.css" target="Content\css\cool\style.css" />
```

Because the file extensions of the source and target match, the `target` is assumed to be the file name, not a directory name.
The source contains `css\cool\style.css`.
The packaged result is `content\css\cool\style.css`.

Content File Copy and Rename

Copy a content file from a source folder into a NuGet package to a different file name in the NuGet package:

```
<file src="ie\css\style.css" target="Content\css\ie.css" />
```

The source contains `ie\css\style.css`.
The packaged result is `content\css\ie.css`.

Excluding Files from the Package Manifest

The `<file>` element within a `.nuspec` file can be used to include a specific file or a set of files using a wildcard character. When doing so, there's no way to exclude a specific subset of the included files. For example, suppose you want all text files within a directory except a specific one:

```
<files>
    <file src="docs\*.txt" target="content\docs" exclude="docs\admin.txt" />
</files>
```

Use semicolons to specify multiple files:

```
<files>
    <file src="*.txt" target="content\docs" exclude="admin.txt;log.txt" />
</files>
```

Use a wildcard character to exclude a set of files, such as all backup files:

```
<files>
    <file src="tools\*.*" target="tools" exclude="tools\*.bak" />
</files>
```

Or use a double wildcard character to exclude a set of files recursively across directories:

```
<files>
    <file src="tools\**\*.*" target="tools" exclude="**\*.log" />
</files>
```

■ **Note** By convention, NuGet ignores a series of files automatically. Every file starting with a dot (.) is ignored by default. The reason for this is that NuGet packages should not contain files and folders created by some source control clients such as TortoiseSVN or Git. If these files are required to be shipped in a NuGet package, make sure to explicitly add them using the `file` element.

APPENDIX B

NuGet Command Line Reference

In previous chapters of this book, you've seen the NuGet command line being used. While we've covered most commands, this appendix lists all available commands from the NuGet command line. Parts of this appendix are based on the information available on the official NuGet web site at `http://docs.nuget.org/docs/reference/command-line-reference`. NuGet documentation (from the Outercurve Foundation) is licensed under Creative Commons license BY 3.0 (`http://creativecommons.org/licenses/by/3.0/`).

Deleting Packages

After publishing a package to a NuGet feed, the NuGet `delete` command enables a user to delete a package from the server. Note that some NuGet feeds, like the official NuGet gallery at NuGet.org, may not allow package deletion. The reason for this is threefold:

- Other packages may depend on that package. Those packages might not necessarily be on the same feed.
- It ensures that folks who are not committing packages (package restore) will not have broken builds.
- It helps ensure that important community owned packages are not mass deleted.

Here's an example:

```
nuget delete <package Id> <package version> [API Key] [options]
```

Available options are shown in Table B-1.

Table B-1. *Available Options for the NuGet delete Command*

Option	Description
`Source`	Specify the server URL.
`NoPrompt`	Do not prompt when deleting.
`Help`	Help information for the command.

Using the help Command

This command displays general help information and help information about other commands:

```
nuget help [command]
```

For example, retrieving help for the NuGet `push` command may be done by issuing the following command:

```
nuget help push
```

Available options are shown in Table B-2.

***Table B-2.** Available Options for the NuGet help Command*

Option	Description
`All`	Print detailed information for all available commands.
`Markdown`	Print detailed help in markdown format.
`Help`	Help information for the command.

Installing Packages

The following command installs a package using the specified sources:

```
nuget install packageId|pathToPackagesConfig [options]
```

If no sources are specified, all sources defined in `%AppData%\NuGet\NuGet.config` are used. If `NuGet.config` specifies no sources, the default NuGet feed is used.

Note that when no package name is specified to the `install` command and instead `packages.config` is referenced, the `install` command will install all packages listed in `packages.config`. This may be very useful to restore a set of dependencies in a project without having to explicitly install all packages manually.

Available options are shown in Table B-3.

***Table B-3.** Available Options for the NuGet install Command*

Option	Description
`Source`	Specify the server URL.
`OutputDirectory`	Specify the directory in which packages will be installed.
`Version`	The version of the package to install.
`ExcludeVersion`	If set, the destination folder will contain only the package name, not the version number.

Option	Description
`PreRelease`	Allow prerelease packages to be installed. This flag is not required when restoring packages from packages.config.
`NoCache`	If set, the NuGet package cache will not be used for installing the specified package.
`Help`	Help information for the command.

Listing Packages

The following command displays a list of packages from a given source:

```
nuget list [search terms] [options]
```

If no sources are specified, all sources defined in `%AppData%\NuGet\NuGet.config` are used. If `NuGet.config` specifies no sources, the default NuGet feed is used.

Available options are shown in Table B-4.

Table B-4. *Available Options for the NuGet list Command*

Option	Description
`Source`	A list of package sources in which to search.
`Verbose`	Display detailed information for each package.
`AllVersions`	List all versions of a package. By default, only the last version is displayed.
`PreRelease`	Allow prerelease packages to be listed.
`Help`	Help information for the command.

Using the pack Command to Create Packages

Create a NuGet package based on the specified `.nuspec` file, a project file, or an assembly name using the pack command:

```
nuget pack <nuspec | project | assembly> [options]
```

Available options are shown in Table B-5.

Table B-5. *Available Options for the NuGet pack Command*

Option	Description
`OutputDirectory`	Specify the directory for the created NuGet package file. If none is specified, use the current directory.
`BasePath`	The base path of the files defined in the `.nuspec` file.
`Verbose`	Show verbose output for package building.
`Version`	Override the version number from the `.nuspec` file.
`Exclude`	Specify one or more wildcard patterns to exclude when creating a package.
`Symbols`	Determine if a package containing sources and symbols should be created. When specified with a `.nuspec` file, create a regular NuGet package file and the corresponding symbols package.
`Tool`	Determine if the output files of the project should be in the `tools` folder.
`Build`	Determine if the project should be built before building the package.
`NoDefaultExcludes`	Prevent default exclusion of NuGet package files and files and folders starting with a dot (e.g., `.svn`).
`NoPackageAnalysis`	Specify if the command should not run package analysis after building the package.
`Properties`	Provide the ability to specify a semicolon-delimited list of properties when creating a package.
`Help`	Help information for the command.

Publishing Packages

The following command publishes a package that was uploaded to the server but not added to the feed:

```
nuget publish <package id> <package version> [API key] [options]
```

For example, some NuGet feeds like the official NuGet gallery at NuGet.org support listing and unlisting a package. The `publish` command can list a package that has previously been unlisted.

Available options are shown in Table B-6.

Table B-6. Available Options for the NuGet publish Command

Option	Description
`Source`	The package source to publish to. By default, NuGet assumes you are publishing to a `v2` feed and appends `/api/v2/package` to the specified package source URL. To overcome this or to publish to a `v1` feed, you have to explicitly specify the source URL
`Help`	Help information for the command.

Pushing a Package

The following command pushes a package to the server:

```
nuget push <package path> [API key] [options]
```

Depending on the NuGet feed you are pushing to, the package will immediately be published as well. Optionally, this behavior can be disabled by specifying the `-CreateOnly` option.

Available options are shown in Table B-7.

Table B-7. Available Options for the NuGet push Command

Option	Description
`CreateOnly`	Specify if the package should be created and uploaded to the server but not published to the server. False by default.
`Source`	The package source to publish to. By default, NuGet assumes you are pushing to a `v2` feed and appends `/api/v2/package` to the specified package source URL. To overcome this or to push to a `v1` feed, you have to explicitly specify the source URL
`ApiKey`	The API key for the server.
`Help`	Help information for the command.

Using the setApiKey Command

This command saves an API key for a given server URL:

```
nuget setapikey <API key> [options]
```

When no URL is provided, the API key for the official NuGet gallery is saved. API keys are stored in `%AppData%\NuGet\NuGet.config`.

Available options are shown in Table B-8.

***Table B-8.** Available Options for the NuGet setApiKey Command*

Option	Description
`Source`	The package source to store the API key for.
`Help`	Help information for the command.

Using the sources Command

This command provides the ability to manage list of sources located in `%AppData%\NuGet\NuGet.config`:

```
nuget sources <List|Add|Remove|Enable|Disable> -Name [name] -Source [source]
```

Available options are shown in Table B-9.

***Table B-9.** Available Options for the NuGet sources Command*

Option	Description
`Name`	Name of the source.
`Source`	Path to the package source.
`Help`	Help information for the command.

Using the spec Command

Generate a nuspec for a new package with the following command:

```
nuget spec [package id]
```

If this command is run in the same folder as a project file (`.csproj`, `.vbproj`, or `.fsproj`), it will create a package manifest that uses replacement tokens, such as `$id$`, `$version$`, `$description$`, and `$authors$`.

Available options are shown in Table B-10.

***Table B-10.** Available Options for the NuGet spec Command*

Option	Description
`AssemblyPath`	Assembly to use for metadata.
`Force`	Overwrite a `.nuspec` file if it exists.
`Help`	Help information for the command.

Updating Packages

Update packages to latest available versions using the following command, which also updates `NuGet.exe` itself.

```
nuget update <packages.config|solution>
```

Available options are shown in Table B-11.

Table B-11. *Available Options for the NuGet update Command*

Option	Description
`Source`	A list of package sources to search for updates.
`Id`	Package IDs to update.
`RepositoryPath`	Path to the local packages folder (location where packages are installed).
`Safe`	Looks for updates with the highest version available within the same major and minor version as the installed package.
`Self`	Update the running `NuGet.exe` file to the newest version available from the server.
`Verbose`	Show verbose output while updating.
`Prerelease`	Allow updating to prerelease versions. This flag is not required when updating prerelease packages that are already installed.
`Help`	Help information for the command.

APPENDIX C

NuGet Package Manager Console PowerShell Reference

Throughout this book, you've seen the NuGet Package Manager Console making use of PowerShell. Although those chapters already covered the most important NuGet commands, this appendix lists all available NuGet commands exposed in the NuGet Package Manager Console.

This appendix builds on top of the contents of this book and will provide references to other parts of this book as well. However, this appendix is not meant to be a full PowerShell reference: there are plenty of good books on that topic.

Parts of this appendix are based on the information available on the official NuGet web site at `http://docs.nuget.org/docs/reference/package-manager-console-powershell-reference`. The NuGet documentation (from the Outercurve Foundation) is licensed under Creative Commons license BY 3.0 (`http://creativecommons.org/licenses/by/3.0/`).

Support for Common Parameters

Note that the PowerShell commands listed in this appendix all support the following set of common PowerShell cmdlet parameters, indicated by the `[<CommonParameters>]` option:

- `Verbose`
- `Debug`
- `ErrorAction`
- `ErrorVariable`
- `WarningAction`
- `WarningVariable`
- `OutBuffer`
- `OutVariable`

For more detailed information, you can simply type `Get-Help About_CommonParameters` in the NuGet Package Manager Console.

Adding Binding Redirects

This command adds binding redirects to the `app.config` or `web.config` file:

```
Add-BindingRedirect [-ProjectName] <string>
```

It has not been explicitly mentioned earlier in this book because, as of NuGet v1.2, this command is run automatically when needed during package installation. The available parameter for this command is shown in Table C-1.

Table C-1. *Available Option for the NuGet Add-BindingRedirect Command*

Option	Description	Required
`ProjectName`	Specify the project to analyze and add binding redirects to.	*True*

Getting a Set of Packages

This command gets the set of packages available from the package source:

```
Get-Package -Source <string> [-ListAvailable] [-Updates] [-ProjectName] [-Recent] [-Filter↩
 <string>] [-First <int>] [-Skip <int>] [-AllVersions] [-IncludePrerelease]
```

This command defaults to showing only the list of installed packages. Use the `-ListAvailable` flag to list packages available from the package source. Available options for this command are shown in Table C-2.

Table C-2. *Available Options for the NuGet Get-Package Command*

Option	Description	Required
`Source`	Specify the URL or directory path for the package source containing the package to install. When set to a local file system path, `Source` can be either absolute or relative to the current directory. If omitted, NuGet looks in the currently selected package source to find the corresponding package URL.	No
`ListAvailable`	Get packages available from the online package source.	*False*
`ProjectName`	Specify the project to get installed packages from. If omitted, the command will return installed projects for the entire solution.	*False*
`Recent`	Get the list of recently installed packages.	*False*
`Updates`	Get packages that have an update available from the package source.	*False*
`Filter`	Specify a filter string used to narrow down the list of packages returned. The filter is searched for in the package ID, the description, and tags.	*False*
`First`	Specifies the number of packages to return from the beginning of the list.	*False*

Option	Description	Required
`Skip`	Skip (do not return) the specified number of packages, counting from the beginning of the list.	*False*
`AllVersions`	Display all available versions of a package. The latest version of each package is listed by default.	*False*
`IncludePrerelease`	Indicate whether to include prerelease packages in the returned results.	*False*

Using this command, you can easily loop through all installed packages on the currently targeted project. The following command prints all installed package IDs for the current project into the Package Manager Console window:

```
Get-Package | % { Write-Host “$_.Id is installed” }
```

Getting Project Information

This command gets the specified project. If none is specified, it returns the default project.

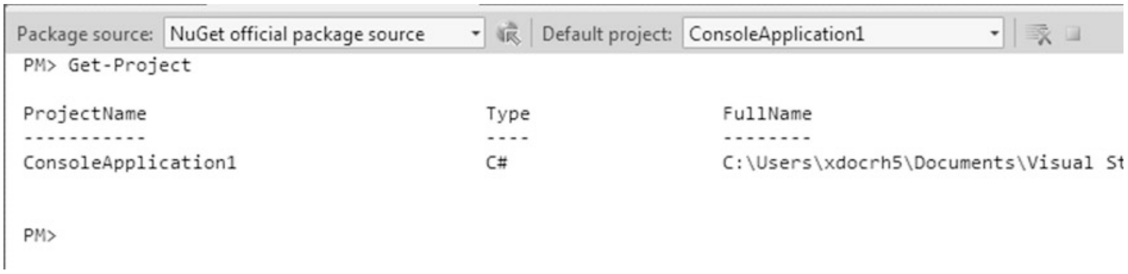
```
Get-Project [[-Name] <string>] [-All] [<CommonParameters>]
```

Available parameters for this command are shown in Table C-3.

Table C-3. *Available Options for the NuGet Get-Project Command*

Option	Description	Required
`Name`	Specify the project to return. If omitted, the default project selected in the Package Manager Console is returned.	*False*
`All`	Return every project in the solution.	*False*

An example of calling `Get-Project` is shown in Figure C-1; it illustrates the output including the `ProjectName`, `Type`, and `FullName` properties of the object.


Package source: NuGet official package source
Default project: ConsoleApplication1

```
PM> Get-Project

ProjectName                    Type          FullName
-----------                    ----          --------
ConsoleApplication1            C#            C:\Users\xdocrh5\Documents\Visual St


PM>
```

Figure C-1. *Example output of the Get-Project cmdlet in NuGet Package Manager Console*

This command opens up quite a few interesting opportunities to further automate the way we work with NuGet, and by extension, Visual Studio.

An example use case could be to install a package in multiple target projects at once, based on some selection criteria. For instance, let's say we want to install the NUnit package into all of our testing projects, which are named using the convention `*Tests.csproj`. This is not something you can easily achieve using the user interface dialogs of the NuGet Visual Studio Extension. Smart usage of the Get-Project command, however, allows you to achieve this very easily, as shown in Listing C-1.

Listing C-1. *Installing a Package in Selected Target Projects Using the Get-Project Cmdlet*

```
Get-Project -All | Where { $_.Name.EndsWith("Tests.csproj") } | Install-Package NUnit
```

When combining the Get-Project command with the Get-Package command, you could also list all installed packages for each project. Listing C-2 shows you how you can do this using a single line in PowerShell.

Listing C-2. *Listing All Installed Packages for All Projects in the Current Visual Studio Solution*

```
Get-Project -All | % { Write-Host $_.ProjectName; Get-Package -ProjectName ↩
$_.ProjectName | % { Write-Host "  $_.Id is installed" } }
```

Installing Packages

This command installs a NuGet package and, by default, also its dependencies into the target project:

```
Install-Package [-Id] <string> [-IgnoreDependencies] [-ProjectName <string>] [[-Version]↩
 <string>] [[-Source] <string>] [-IncludePrerelease] [<CommonParameters>]
```

Available parameters for this command are shown in Table C-4.

***Table C-4.** Available Parameters for the NuGet Install-Package Command*

Option	Description	Required
`Id`	Specify the package ID of the package to install.	*True*
`IgnoreDependencies`	Install only this package and not its dependencies.	*False*
`ProjectName`	Specify the project to install the package into. If omitted, the default project is chosen.	*False*
`Version`	Specify the version of the package to install. If omitted, defaults to the latest version.	*False*
`Source`	Specify the URL or directory path for the package source containing the package to install. When set to a local file system path, `Source` can be either absolute or relative to the current directory. If omitted, NuGet looks in the currently selected package source to find the corresponding package URL.	*False*
`IncludePrerelease`	Indicate whether this command will consider prerelease packages. If omitted, only stable packages are considered.	*False*

Creating Packages

This command creates a new package when supplied with a `.nuspec` package specification file:

```
New-Package [[-ProjectName] <string>] [-SpecFileName] <string> [-TargetFile] <string>↩
 [-NoClobber] [<CommonParameters>]
```

Available parameters for this command are shown in Table C-5.

***Table C-5.** Available Parameters for the NuGet New-Package Command*

Option	Description	Required
`ProjectName`	Specify the project containing the `.nuspec` file to use when creating the package. If omitted, the current project selected in the console is used.	*False*
`SpecFileName`	Specify the `.nuspec` file used to create the package. If omitted, the `.nuspec` file within the current project is used if there is only one such file.	*True*
`TargetFile`	Specify the full name of the output NuGet package file.	*True*
`NoClobber`	If specified, the target file is not overwritten.	*False*

To create a new NuGet package based on the currently selected project in the NuGet Package Manager Console, you simply call `New-Package`. This will work if you have a single `.nuspec` file—yes, only one!—within this project.

If you want to specify the NuGet manifest file and project to be used, you could provide those values as follows:

```
New-Package -Project MyProjectName -SpecFileName MyPackage.nuspec
```

Open Package Pages

This command will open the browser pointing to `ProjectUrl`, `LicenseUrl`, or `ReportAbuseUrl` of the specified package.

```
Open-PackagePage -Id <string> [-Version] [-Source] [-License] [-ReportAbuse] [-PassThru]↩
 [<CommonParameters>]
```

Available parameters for this command are shown in Table C-6.

Table C-6. *Available Parameters for the NuGet Open-PackagePage Command*

Option	Description	Required
`Id`	Specify the ID of the NuGet package to search for.	*False*
`Version`	Specify the version of the package to search for. If omitted, defaults to the latest version.	*False*
`Source`	Specify the source of the repository to search for package. If omitted, defaults to the selected source in the package source drop-down control.	*False*
`License`	Indicate that the cmdlet should open the `LicenseUrl` of the specified package. If neither `LicenseUrl` nor `ReportAbuseUrl` is set, the cmdlet will open the `ProjectUrl` by default.	*False*
`ReportAbuse`	Indicate that the cmdlet should open the `ReportAbuseUrl` of the specified package. If neither `LicenseUrl` nor `ReportAbuseUrl` is set, the cmdlet will open the `ProjectUrl` by default.	*False*
`PassThru`	If specified, the cmdlet will return the value of the requested URL.	*False*

Uninstalling Packages

This command uninstalls a NuGet package. If other packages depend on this package, the command will fail unless the `–Force` option is specified.

```
Uninstall-Package [-Id] <string> [-RemoveDependencies] [-Force] [-Version <string>]↩
 [-ProjectName <string>] [<CommonParameters>]
```

Available parameters for this command are shown in Table C-7.

Table C-7. *Available Parameters for the NuGet Uninstall-Package Command*

Option	Description	Required
`Id`	Specify the package ID of the package to uninstall.	*True*
`RemoveDependencies`	Uninstall the package and its *unused* dependencies.	*False*
`Force`	Force uninstalling this package and its dependencies, whether they are used by other packages or not.	*False*
`Version`	The version of the package to uninstall. If omitted, defaults to the latest version.	*False*
`ProjectName`	Indicate whether to include prereleases when searching for updates. If omitted, only stable packages are considered.	*False*

Updating Packages

This command updates a package and its dependencies to a newer version:

```
Update-Package [-Id] <string> [-IgnoreDependencies] [-ProjectName <string>] [-Version↩
 <string>] [-Safe] [-Source <string>] [-IncludePrerelease] [<CommonParameters>]
```

Available parameters for this command are shown in Table C-8.

Table C-8. *Available Parameters for the NuGet Update-Package Command*

Option	Description	Required
`Id`	Specify the package ID of the package to update. If omitted, every package is updated.	*True*
`IgnoreDependencies`	Update all of the package's dependencies to the latest version.	*False*
`ProjectName`	Specify the project containing the package to update. If omitted, the package is updated in every project with the package installed.	*False*
`Version`	Specify the version that the package will be upgraded to. If omitted, defaults to the latest version.	*False*
`Safe`	Constrain upgrades to newer versions with the same major and minor version components. For example, if version 1.0.0 of a package is installed, and versions 1.0.1, 1.0.2, and 1.1 are available in the feed, the `-Safe` flag updates the package to 1.0.2.	*False*

Option	Description	Required
`Source`	Specify the URL or directory path for the package source containing the package to update. When set to a local file system path, `Source` can be either absolute or relative to the current directory. If omitted, NuGet looks in the currently selected package source to find the corresponding package URL.	*False*
`IncludePrerelease`	Indicate whether to include prereleases when searching for updates. If omitted, only stable packages are considered.	*False*

Index