Chapter No. 1
"Using a Package Manager in Visual Studio"
In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.1 "Using a Package Manager in Visual Studio"

A synopsis of the book’s content

Information on where to buy this book

About the Authors

Damir Arh has been working with the .NET Framework since it was first released when he was still in university. He has a lot of experience in desktop-based applications, be it Windows Forms or WPF. But he has also done some work in ASP.NET, both web forms and MVC. Lately, he has given more attention to mobile platforms, mostly Windows Store apps and Windows Phone. He's a fan of portable class libraries since he first tried them out.

He has been working on large software projects for over 10 years, gaining experience in source control, continuous integration, and good development practices and patterns to cope with their complexities. He likes to share his knowledge with other developers through his blog at www.damirsCorner.com, by answering questions at Stack Overflow, and by speaking at local user group meetings and technical conferences. He has been awarded the Microsoft Most Valuable Professional (MVP) for the second year in a row.

Currently, he works as a Software Architect for Adacta, a company specializing in development of software for insurance companies, based in Ljubljana, Slovenia.

For More Information:
First, I'd like to thank Dejan for deciding to join me in the interesting and challenging project of writing this book. His help made the amount of work required easier to handle. Next, I want to thank the people at Packt Publishing for originally approaching me with the idea for this book, for giving us both the opportunity to write it, and of course for being there throughout the project and helping us getting it completed.

I'd also like to thank the reviewers and the editors for their valuable feedback; they have helped to make this book better.

Last but not least, I'd like to thank my parents and friends for their support and understanding during the evenings and weekends I spent working on this book. Thank you.

Dejan Dakić has been working with the .NET Framework for over five years. He has the most experience with ASP.NET but has also done a lot of work in Windows Forms and WPF. He has been active in mobile development since Windows Mobile, and now he has progressed to Windows RT. Lately, he has been spending time in the Windows Azure environment, focusing on web development, both client- and server-side.

Currently, he works in a team with Damir as a software developer for Adacta—a company specializing in development of software for insurance companies, which is based in Ljubljana, Slovenia.

First, I'd like to thank Damir for inviting me to this interesting project of writing this book. He has been an admirable mentor throughout the whole process.

I'd also like to thank the people at Packt Publishing, reviewers, and the editors for their valuable feedback.

I'd also like to thank my parents, my sister, and friends for their support, comments, and feedback.

Last but not least, I'd like to thank my girlfriend Pia for supporting me and putting up with my late night works on this project. Thank you.

For More Information:
NuGet 2 Essentials

Using third-party libraries in .NET projects has been a nontrivial endeavor for quite a long time. Without a centralized catalog and installation process, developers first needed to discover them using their favorite search engine, then installed them manually or using a proprietary tool, and finally referenced them in such a way that the assemblies would be distributed along with the source code. The complexity of this process and the lack of discoverability were a huge roadblock for the expansion of an open source ecosystem. With only a few exceptions, such as NUnit and log4net, developers mostly used the classes that were shipped with the framework.

This started to change in 2010 when the first version of NuGet was released. At that time it was named NuPack, but it was soon renamed to avoid confusion with another already existing software package. NuGet is a package manager tool, similar to Ruby Gems and Linux package management systems. It enables easy discoverability, installation, and upgrading of third-party libraries used in development projects.

In the three years since its release, NuGet has become one of the most important tools for .NET development. It is included with the latest version of Visual Studio, many Microsoft project templates and libraries take advantage of it, and most importantly, it became a standard approach for distribution of third party-libraries, the open source ones in particular. At the time of writing this book, there were over 14,000 packages available.

This book takes the reader through all aspects of using NuGet, from the basics to the advanced scenarios of creating packages and hosting an internal server. It demonstrates all the concepts on real-world examples, making it easier to recognize the opportunities for using NuGet in everyday development work.

What This Book Covers

Chapter 1, Using a Package Manager in Visual Studio, introduces you to NuGet, demonstrates its benefits, and teaches the basics of using it. It will guide you through the NuGet installation process, show you how to reference your first NuGet package, and demonstrate how dependencies between packages are handled.

Chapter 2, Package Management, covers advanced aspects of using NuGet in Visual Studio. It starts out with two important aspects of NuGet packages: versioning and support for different target platforms. This is followed by a closer look at how NuGet plays along with source control systems. It ends with an introduction to the NuGet PowerShell console.

For More Information:
Chapter 3, *Creating and Publishing a Package*, focuses on the basics of creating a NuGet package for a library. It shows you how to do it manually and how to automatically create it at the end of every build. It also explains package versioning and leads you through the process of publishing your first package in the NuGet Gallery.

Chapter 4, *Adding Features to a NuGet Package*, drills down into the advanced features of NuGet packages. It describes the contents of a package and gives examples for each of the following features: multiplatform support, multi-language support, and debugging support. It also explains which types of files can be included in a package and why you would want to include them.

Chapter 5, *Hosting an Internal NuGet Server*, starts with the reasons to host your own NuGet server and explains how to choose the right setup among the three different options, based on your requirements. It includes detailed installation and configuration instructions for setting up any one of them: a file system repository, a simple NuGet server, and your own NuGet gallery.

For More Information:
Using a Package Manager in Visual Studio

NuGet has become one of the best choices for package management in Microsoft .NET. It simplifies handling of third-party libraries, automatically configures your project by adding references to all the necessary assemblies, and updates source and configuration files.

Before NuGet, there was no real dependency management system for .NET projects that could handle adding, saving, and maintaining libraries to solutions and source control systems.

The official NuGet feed is located on http://www.nuget.org and it is a place for over 15,000 unique packages.

Benefits of using a package manager

As a project grows, the need for additional functionality increases. Some of this functionality can be found in various libraries, so the first issue depends on the decision whether to use external libraries or not. There is a variety of solutions out there and sometimes software companies try to invent their own. We usually end up with one of the two different approaches when doing work on various projects.

The first approach we encountered is simply not using external libraries. Although not having to deal with libraries saves you all the troubles related to third-party dependencies, but you end up reinventing the wheel over and over again.

For More Information:
Using a Package Manager in Visual Studio

The second approach is much better and more common: having all the external assemblies in the root project folder (named as lib, References, or Dependencies) and referencing them in projects where needed. Adding these assemblies—either binaries or source code—in source control can save you the trouble when building a solution or setting up a new development environment.

Once a third-party assembly is added to the repository, it is very likely that the assembly never gets updated to a newer version; we do not receive any notification about the new version being released. Updating to a new library release is complicated. The most common reason for version update is when the newer version solved a specific issue encountered by the development team. Upgrading these libraries can cause headaches, problems with backward compatibility, and even requires migration to a newer version of the .NET Framework.

Software developers often search for these third-party assemblies on various open source code repositories such as GitHub, Codeplex, and similar. Once we find a site for the required library, we will start looking for documentation, date of last build, and last check in; basically, everything that will convince us the library is trustworthy and reliable.

NuGet solves all these problems in a simple and convenient way via a graphical user interface or a PowerShell-based command line named Package Management Console.

In the following sections, we will first demonstrate how we would add a third-party library and then reach the same goal using NuGet. We will also cover the installation of NuGet.

Manually adding third-party library references

In this chapter we will describe the most common way a developer finds a library and adds it to a solution. For the sake of simplicity, let us create a console application that will retrieve and parse an JSON result from a web service, for example, Google query search.

Let us start by creating a new console application:

2. In the **New Project** dialog, choose the **Console Application Visual C#** template.

3. Let us name our solution **CH1_01_GoogleApiSearcher**, and start writing some code as shown:

   ```csharp
class Program
{
    static void Main()
    {
        var request = "https://ajax.googleapis.com/ajax/services/search/web?v=1.0&q=nuget";
        using (var webClient = new WebClient())
        {
            var result = webClient.DownloadString(request);
            Console.WriteLine(result);
        }
    }
}
```

---

For More Information:
In the first line of the `Main` method, we store a HTTP request in a string, and it is used to query Google's Search API for search results for the `nuget` query. In the second line of this method, we are creating an instance of `WebClient`, making the web request, and then storing the result in a new string variable. The last line outputs the raw result to the console, in the form of a large unformatted JSON string.

Great, we have a JSON result. We now need a JSON library to parse our result into a meaningful form. When dealing with unknown libraries, the best choice is to type some relevant keywords into our favorite search engine.

For example, let's write `json for .net` into Google's search engine. The first hit on Google is `Json.NET - Home`, which sounds about right. As we all know, the first hit is not necessarily the best choice when searching for third-party libraries, but in this case we got lucky. Following the first hit, we are forwarded to `http://json.codeplex.com/`. After reading the description, we are sure that this is the appropriate library for our problem. So, we will find a download button and transfer the archived files to the computer. After opening the archived file, we will see two folders: `Bin` and `Source`. We decide that we will not be needing the library's source code; let us just choose the binary assembly and reference it from our project. In the `Bin` folder, we see another subset of folders with platform names as shown in the following screenshot; so again, we have to choose what to use in our code:

For More Information:

Since we are writing our application in .NET Framework 4.5, files in the Net45 folder should be the right ones. But where should we put these files? There are no clear guidelines about how and where external references should be stored. In our root solution folder we should create a folder that will contain our external references. Let's name our newly created folder ExternalReferences and place the assemblies there.

Now we are set to add a reference in our project:

1. Choose Add Reference from the context menu in Solution Explorer on the References node.
2. Use the Browse button to select our assembly file from the file dialog box. The assembly file is located in the ExternalReferences folder in our project's root folder.
3. Click on the OK button to confirm the selection and add the assembly to the project.

Now we are ready to use the Newtonsoft.Json library for JSON manipulations. By changing our source code, we are able to get more useful information from our web service result. Let's make changes to our Main method, so it will look as shown:

```csharp
static void Main(string[] args)
{
    var request = "https://ajax.googleapis.com/ajax/services/search/web?v=1.0&q=nugget";
    string result;
    using (var webClient = new WebClient())
    {
        result = webClient.DownloadString(request);
    }
    // Deserializing JSON string to object
    dynamic data = JsonConvert.DeserializeObject(result);
    // Looping throughout results
    foreach (var item in data.responseData.results)
    {
        Console.WriteLine("{0} {1}", item.titleNoFormatting, item.url);
    }
}
```

**Downloading the example code**

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
Using a Package Manager in Visual Studio

We used a method from our recently-added `JsonConvert.DeserializeObject` library. This code is written in .Net 4.5 and we can use the dynamic type for our result.

The dynamic keyword is also available in .Net 4.0; more about the dynamic keyword is available at http://bit.ly/csdynmc.

In the foreach loop, we are looping through results and displaying the `titleNoFormatting` and `url` properties of results, which are our search results. Running the application outputs the following to the console window:

```
NuGet Gallery | Home http://nuget.org/
NuGet - Wikipedia, the free encyclopedia
  http://en.wikipedia.org/wiki/NuGet
NuGet Package Manager extension - Visual Studio Gallery - Microsoft
  http://visualstudiogallery.msdn.microsoft.com/27077b70-9dad-4c64-adcf-c7cf6bc9970c
```

This library did not require editing of the configuration file, which is usually the most time-consuming step when using third-party libraries. Now, we have added a reference manually and successfully used it in our project. In the next chapter, we will learn how to set up NuGet in a Visual Studio environment; then we will take a look at how to add references using the NuGet Package Management.

**Setting up NuGet in Visual Studio**

This section will cover setting up NuGet in your development environment. If you are using Visual Studio 2012, you can skip it. NuGet client is a part of Visual Studio 2012 and it is fully integrated by default.

**Requirements**

Let's start with the operating system requirements. PowerShell 2.0 or later is required by NuGet Package Manager Console. It is already installed on the following operating systems:

- Windows 8/8.1
- Windows 7
- Windows Server 2008 R2

For More Information:

If you are using an older version of Windows, you have to install PowerShell 2.0 manually. This is required for the following systems:

- Windows XP SP3
- Windows Server 2008
- Windows Vista SP1
- Windows Server 2003 R2

Installation of PowerShell 2.0 is not a subject of this book. The detailed PowerShell 2.0 installation procedure can be found at http://bit.ly/PS20instll.

Installing NuGet
Visual Studio 2010 requires installation of NuGet Package Manager. First let us check if the NuGet extension is already installed in your Visual Studio. If NuGet is already installed, there should be Library Package Manager under the Tools menu.

Installing from the Visual Studio gallery
You can install NuGet from Microsoft's Visual Studio gallery by performing the following steps:

2. After selecting the Visual Studio instances where you wish to install NuGet Package Manager, you need to agree with the terms and click on the Install button.
3. After the installation is complete, a restart of Visual Studio is required. Once Visual Studio restarts, NuGet is ready to use.

Installing from the Extension Manager
We can install NuGet Package Manager from Visual Studio 2010 using the Extension Manager, as follows:

1. Navigating to Tools | Extension Manager brings up an Extension Manager dialog box.
2. Selecting the Online Gallery tab from the left side of the dialog box shows us all the available extensions from the Visual Studio Gallery.

For More Information:
3. Type `nuget` in the search box to find the **NuGet Package Manager** extension.

4. When **NuGet Package Manager** is selected, the **Download** button appears; and after clicking on it, we start the installation process described in the previous section.

### Installing the command-line utility

By using the `NuGet.exe` command-line utility, we can interact with NuGet directly from the command line. The `NuGet.exe` utility is available for download at [http://bit.ly/nugetdl](http://bit.ly/nugetdl). When the downloading is complete, we would recommend adding `NuGet.exe` to the PATH environment variable.


Now it is time to open the command prompt and start using the NuGet command-line utility. Just by typing `nuget` we get a list of available commands.

### Adding references to third-party libraries using NuGet

In this section, we will introduce a way of adding libraries with NuGet. Assuming you have successfully installed NuGet to your environment, you are now ready to start with a new project.

We are going to create a new empty console application, just like we did in the previous demo. In Visual Studio, perform the following steps:

1. Navigate to **File** | **New Project** | **Console Application**.
2. Name the project `CH1_02_GoogleApiSearcherNuget` and then confirm by clicking on the **OK** button.

When the project is created, a `Program.cs` source file with an empty **Main** method is opened in the text editor. In **Solution Explorer** our project's tree view is displayed. Right-clicking on the **References** node brings up the context menu, which contains three different menu items for adding a reference:

- Add Reference...
- Add Service Reference
- Manage NuGet Packages...

For More Information:
In the previous example we added a reference using the first option. Now our goal is to add a library using NuGet. Selecting Manage NuGet Packages... opens the dialog box, where we are managing all the packages related to the selected project. By default, the dialog opens on the Installed packages view, which is empty in our case since we have not added any packages yet. Selecting the Online option from the left-hand side menu lists every available package in the NuGet official package source. There are a lot of packages in this source; using the Search online textbox we can narrow down the results. Typing json into the textbox will automatically trigger a query on the NuGet feed, returning only the packages related to json. The list view in the middle of a dialog contains the search results. When selecting an item in the list view, the Install button appears. On the details pane on the right side, we can see all the important information about the selected package, for example, who is the creator, what is the package version, links to the project sites, library terms, and dependencies.

Let us select Json.NET and then click on the Install button. After a couple of progress bars, a green check icon appears on the package, meaning the package is installed in our project, as shown in the following screenshot:

Now we can re-use the code from the previous sample and compile the solution. Output is the same as that of the previous program, only this time we spent a lot less time for adding the reference.

By adding a NuGet package, we gained two files. The first file is the assembly under the References folder and the other is packages.config in our project's root. Information about our packages is stored in the packages.config file. For each package there is a unique package ID, version, and target framework version. We will see how NuGet uses this information in the next chapter.

Dependencies with NuGet

In this section, I would like to cover how NuGet handles dependencies. Let us continue with our solution from the previous section. If we want to create a portable class library from our project, we will not have the WebClient class available to us.

For More Information:
Using a Package Manager in Visual Studio

The problem is solved using the NuGet package called Microsoft.Net.Http. Perform the following steps:

1. In Solution Explorer, right-click on the References node and then select Manage NuGet Packages....
2. Search for httpclient in the Online tab; when the search is complete, select Microsoft HTTP Client Libraries.

In the package summary window under Description and Tags, we can see that this package has two dependencies: Microsoft.Bcl and Microsoft.Bcl.Build. This means that these two libraries must be installed while installing the package Microsoft HTTP Client Libraries. Let us install this package and accept terms when prompted.

This time we gained three extra lines in our packages.config file: one for our package and two dependency packages that are required for our package.

Now we can use HttpClient instead of WebClient. Instead of the following code:

```csharp
using (var webClient = new WebClient())
{
    result = webClient.DownloadString(request);
}
```

Use the following code:

```csharp
using (var webClient = new HttpClient())
{
    result = webClient.GetStringAsync(request).Result;
}
```

It should produce the same result. GetStringAsync is a task-based asynchronous pattern method. For the sake of simplicity, we are calling the Result property, which triggers synchronous execution. More about this pattern is available at http://bit.ly/taskasync.

For More Information:
Summary
This chapter has introduced NuGet and covered many of its benefits. We have briefly described the manageability problems with third-party libraries without NuGet and how NuGet solves these issues. We have also described the installation process for NuGet and all the tools required for the development environment. We added an external library manually by following a step-by-step tutorial. We concluded with a tutorial where we used NuGet to achieve the same result; we also used NuGet to add a library with dependencies on other libraries. In the next chapter, we will take a dive deep into the NuGet features such as updating referenced third-party libraries, multitarget platform support, and using NuGet with source control.

For More Information:
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.