Executing PowerShell scripts from C#

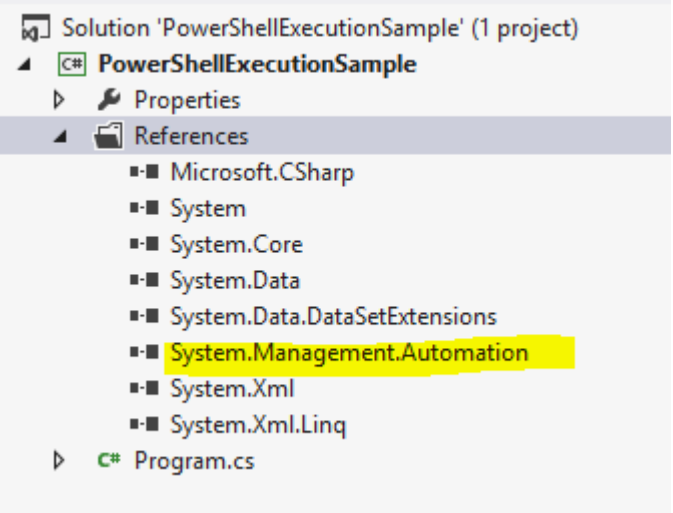
By kexugit

In today's post, I will demonstrate the basics of how to execute PowerShell scripts and code from within a C#/.NET applications. I will walk through how to setup your project prerequisites, populate the pipeline with script code and parameters, perform synchronous and asynchronous execution, capture output, and leverage shared namespaces.

Update 8/7/2014: Here is the downloadable solution file.
Update 11/5/2014: Added a section on execution policy behavior.

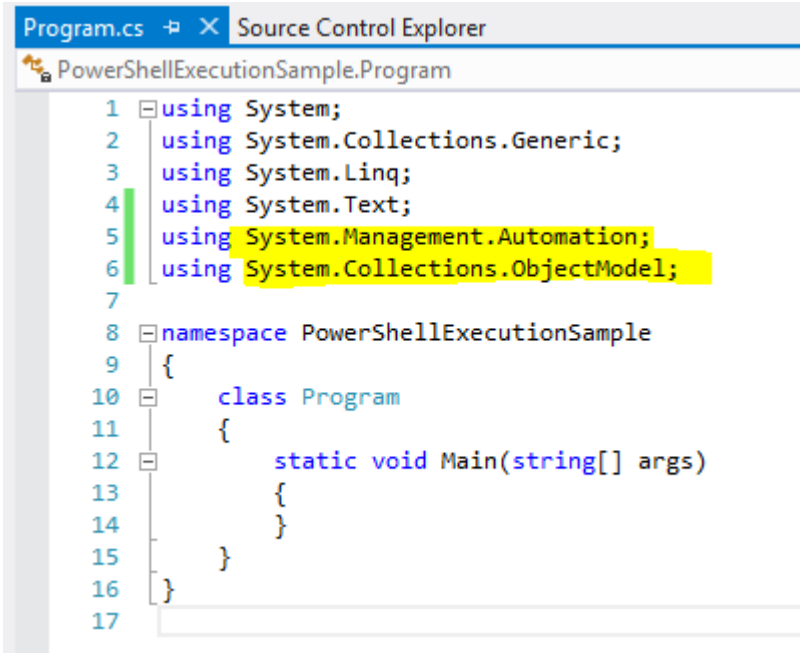
Prerequisites:

First, ensure that PowerShell 2.0 or later is installed on the system you are developing on. The features used below will not be supported on PowerShell 1.0. Next, start by making a new console application (targeting .NET 4.0) in Visual Studio.



In the solution explorer, add a project reference to the ***System.Management.Automation*** assembly ***** . On my machine (PowerShell 3.0), it was located at ***C:\Program Files (x86)\Reference Assemblies\Microsoft\WindowsPowerShell\3.0***. Then, add ***using*** statements for the above referenced assembly, along with ***System.Collections.Objectmodel***, which is used later for execution output.

Note: *You will need to install the Windows SDK in order to obtain the* ***System.Management.Automation*** *assembly file.*
Note*: You do not have to distribute the* ***System.Management.Automation*** *assembly file with your application binaries - it is located in the GAC on machines with Windows PowerShell installed and should resolve automatically.*



Prepare the Pipeline for execution:

The first step is to create a new instance of the ***PowerShell*** class. The static method ***PowerShell.Create()*** creates an empty PowerShell pipeline for us to use for execution. The PowerShell class implements ***IDisposable***, so remember to wrap it up in a using block.

```
    using (PowerShell PowerShellInstance = PowerShell.Create())
{
}
```

Next, we can add scripts or commands to execute. Call the ***AddScript()*** and ***AddCommand()*** methods to add this content to the execution pipeline. If your script has parameters, adding them is easy with the ***AddParameter()*** method. ***AddParameter()*** accepts a string parameter name, and object for the parameter value. Feel free to pass in full object instances/types if you want. The scripts and pipeline handle it just fine. No need to limit yourself with only passing in string values.

The string contents supplied to ***AddScript()*** can come from anywhere. Good choices may be text loaded from files on disk or embedded resources, or user input. For the purposes of this tutorial, I'm just hard-coding some example script text.

```
    using (PowerShell PowerShellInstance = PowerShell.Create())
{
    // use "AddScript" to add the contents of a script file to the end of the execution pipeline.
    // use "AddCommand" to add individual commands/cmdlets to the end of the execution pipeline.
    PowerShellInstance.AddScript("param($param1) $d = get-date; $s = 'test string value'; " +
            "$d; $s; $param1; get-service");

    // use "AddParameter" to add a single parameter to the last command/script on the pipeline.
```

```
        PowerShellInstance.AddParameter("param1", "parameter 1 value!");
    }
```

Script/Command Execution:

So far, we have a PowerShell pipeline populated with script code and parameters. There are two ways we can call PowerShell to execute it: synchronously and asynchronously.

Synchronous execution:

For synchronous execution, we call ***PowerShell.Invoke()*** . The caller waits until the script or commands have finished executing completely before returning from the call to ***Invoke()*** . If you don't care about the items in the output stream, the simplest execution looks like this:

```
      // invoke execution on the pipeline (ignore output)
    PowerShellInstance.Invoke();
```

For situations where you don't need to see or monitor the output or results of execution, this may be acceptable. But, in other scenarios you probably need to peek at the results or perform additional processing with the output that comes back from PowerShell. Let's see what that code looks like:

```
      // invoke execution on the pipeline (collecting output)
    Collection<PSObject> PSOutput = PowerShellInstance.Invoke();

    // loop through each output object item
    foreach (PSObject outputItem in PSOutput)
    {
        // if null object was dumped to the pipeline during the script then a null
        // object may be present here. check for null to prevent potential NRE.
        if (outputItem != null)
        {
            //TODO: do something with the output item
            // outputItem.BaseObject
        }
    }
```

The return object from ***Invoke()*** is a collection of ***PSObject*** instances that were written to the output stream during execution. ***PSObject*** is a wrapper class that adds PowerShell specific functionality around whatever the base object is. Inside the ***PSObject*** is a member called ***BaseObject***, which contains an object reference to the base type you are working with. If nothing was written to the output stream, then the collection of ***PSObjects*** will be empty.

Besides the standard output stream, there are also dedicated streams for warnings, errors, debug, progress, and verbose logging. If any cmdlets leverage those streams, or you call them directly (***write-error***, ***write-debug***, etc.), then those items will appear in the streams collections.

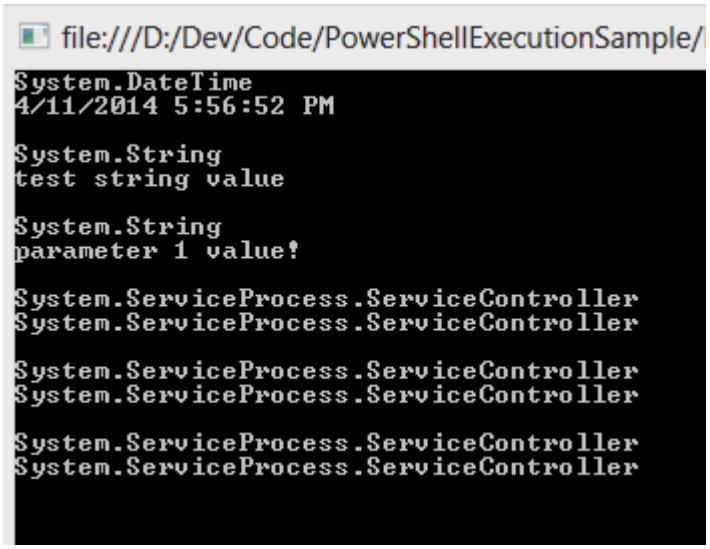
After invoking the script, you can check each of these stream collections to see if items were written to them.

```
  // invoke execution on the pipeline (collecting output)
Collection<PSObject> PSOutput = PowerShellInstance.Invoke();

// check the other output streams (for example, the error stream)
if (PowerShellInstance.Streams.Error.Count > 0)
{
    // error records were written to the error stream.
    // do something with the items found.
}
```

In my sample script supplied above, I am writing a few manually-created objects to the output stream and calling the ***get-service*** cmdlet, which also writes its output to the stream since I didn't save the output in a variable.

As a test, I write the type name and ***ToString()*** on all of the base objects that came back from the sample script. As you can see, accessing the base object allows you to view or manipulate the object directly as needed. The base types that come back from execution are usually standard .NET types you may already be familiar with, just wrapped up in a ***PSObject***.

```
  // loop through each output object item
foreach (PSObject outputItem in PSOutput)
{
    // if null object was dumped to the pipeline during the script then a null
    // object may be present here. check for null to prevent potential NRE.
    if (outputItem != null)
    {
        //TODO: do something with the output item
        Console.WriteLine(outputItem.BaseObject.GetType().FullName);
        Console.WriteLine(outputItem.BaseObject.ToString() + "\n");
    }
}
```




```
Asynchronous execution:
```

For asynchronous execution, we call ***PowerShell.BeginInvoke()*** . ***BeginInvoke()*** immediately returns to the caller and the script execution begins in the background so you can perform other work. Unlike the synchronous ***Invoke()*** method, the return type of ***BeginInvoke()*** is not a collection of ***PSObject*** instances from the output stream. The output isn't ready yet. The caller is given an ***IAsyncResult*** object to monitor the status of the execution pipeline. Let's start with a really simple example:

```
  using (PowerShell PowerShellInstance = PowerShell.Create())
{
    // this script has a sleep in it to simulate a long running script
    PowerShellInstance.AddScript("start-sleep -s 7; get-service");

    // begin invoke execution on the pipeline
    IAsyncResult result = PowerShellInstance.BeginInvoke();

    // do something else until execution has completed.
    // this could be sleep/wait, or perhaps some other work
    while (result.IsCompleted == false)
    {
        Console.WriteLine("Waiting for pipeline to finish...");
        Thread.Sleep(1000);

        // might want to place a timeout here...
    }

    Console.WriteLine("Finished!");
}
```

Note: *If you wrap the PowerShell instance in a* ***using*** *block like the sample above and do not wait for execution to complete, the pipeline will close itself, and will abort script execution, when it reaches the closing brace of the* ***using*** *block. You can avoid this by waiting for completion of the pipeline, or by removing the* ***using*** *block and manually calling* ***Dispose()*** *on the instance at a later time.*

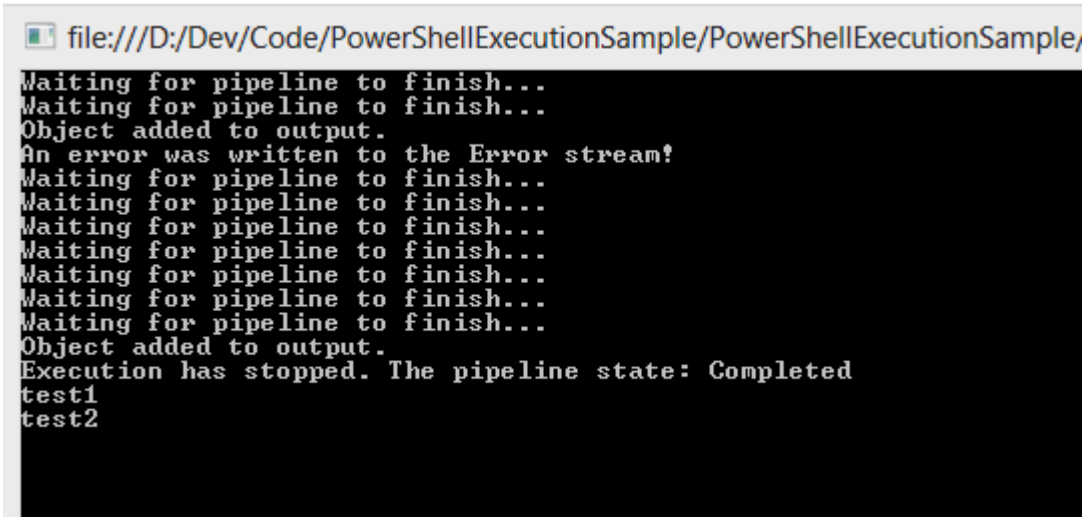
In this first asynchronous example, we ignored the output stream. Again, this is only useful if you don't care about your script results. Fortunately, the ***BeginInvoke()*** method has a few overloads that allow us to extend the functionality. Let's improve the example by adding output collection and event handling for data hitting the pipeline.

First, we will create a new instance of ***PSDataCollection<PSObject>*** . This collection is a thread-safe buffer that will store output stream objects as they hit the pipeline. Next, we will subscribe to the ***DataAdded*** event on this collection. This event will fire every time an object is written to the output stream. To use this new output buffer, pass it as a parameter to ***BeginInvoke()*** .

I added some other functionality and comments to the next code block. First, notice that you can check the state of the pipeline to see its status. The ***State*** will equal ***Completed*** when your script is done. If ***State*** is ***Failed***, it is likely caused by an unhandled exception that occurred in the script, also known as a terminating error. Second, if your scripts utilize ***write-error***, ***write-debug***, ***write-progress***, etc. (all thread-safe collections), you can review these during or after execution to check for items logged there. I have subscribed to the ***DataAdded*** event on the ***Error*** stream as well to be notified in real time.

```
  /// <summary>
/// Sample execution scenario 2: Asynchronous
/// </summary>
/// <remarks>
/// Executes a PowerShell script asynchronously with script output and event handling.
/// </remarks>
public void ExecuteAsynchronously()
{
    using (PowerShell PowerShellInstance = PowerShell.Create())
    {
        // this script has a sleep in it to simulate a long running script
        PowerShellInstance.AddScript("$s1 = 'test1'; $s2 = 'test2'; $s1; write-error 'some error';start-sle

        // prepare a new collection to store output stream objects
        PSDataCollection<PSObject> outputCollection = new PSDataCollection<PSObject>();
        outputCollection.DataAdded += outputCollection_DataAdded;

        // the streams (Error, Debug, Progress, etc) are available on the PowerShell instance.
        // we can review them during or after execution.
        // we can also be notified when a new item is written to the stream (like this):
        PowerShellInstance.Streams.Error.DataAdded += Error_DataAdded;

        // begin invoke execution on the pipeline
        // use this overload to specify an output stream buffer
```

```
            IAsyncResult result = PowerShellInstance.BeginInvoke<PSObject, PSObject>(null, outputCollection);

            // do something else until execution has completed.
            // this could be sleep/wait, or perhaps some other work
            while (result.IsCompleted == false)
            {
                Console.WriteLine("Waiting for pipeline to finish...");
                Thread.Sleep(1000);

                // might want to place a timeout here...
            }

            Console.WriteLine("Execution has stopped. The pipeline state: " + PowerShellInstance.InvocationState

            foreach (PSObject outputItem in outputCollection)
            {
                //TODO: handle/process the output items if required
                Console.WriteLine(outputItem.BaseObject.ToString());
            }
        }
    }

    /// <summary>
    /// Event handler for when data is added to the output stream.
    /// </summary>
    /// <param name="sender">Contains the complete PSDataCollection of all output items.</param>
    /// <param name="e">Contains the index ID of the added collection item and the ID of the PowerShell instance
    void outputCollection_DataAdded(object sender, DataAddedEventArgs e)
    {
        // do something when an object is written to the output stream
        Console.WriteLine("Object added to output.");
    }

    /// <summary>
    /// Event handler for when Data is added to the Error stream.
    /// </summary>
    /// <param name="sender">Contains the complete PSDataCollection of all error output items.</param>
    /// <param name="e">Contains the index ID of the added collection item and the ID of the PowerShell instance
    void Error_DataAdded(object sender, DataAddedEventArgs e)
    {
        // do something when an error is written to the error stream
        Console.WriteLine("An error was written to the Error stream!");
    }
```



As you can see above, the complete asynchronous model allows for some pretty interesting scenarios with long-running scripts. We can listen for events like data being added to the output stream. We can monitor the progress stream to see how much longer a task may take and provide feedback to the UI. We can listen for new errors being written to the error stream and react accordingly, instead of waiting until all execution has completed.

Execution Policy:

When you invoke cmdlets through the PowerShell class in C#, the execution policy behavior is subject to the policy restrictions of the machine. This behavior is the same is if you opened up a PowerShell prompt on the machine. You can issue commands just fine, but invoking another script might get blocked if your execution policy is undefined or restricted in some way. An easy way to get around this is to set the execution policy for the scope of the application process. Simply run ***Set-ExecutionPolicy*** and specify the scope to be ***Process***. This should allow you to invoke secondary scripts without altering the machine/user policies. For more information about policies and their order of precedence, see here.

Shared Namespace:

To wrap up our discussion, I'd like to share a potentially useful feature that can expand some of the functionality for your scripts. When you use the PowerShell class to invoke a script, that script has access to the classes inside the caller's namespace, if they were declared public.

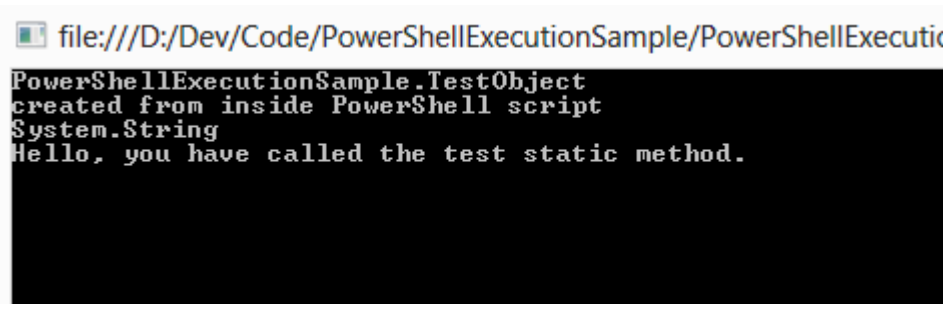
In the example code below, we make a public class with a single public property, and a public static class with a single public method. Both of these items exist in the same namespace as the PowerShell executor code. The script we execute creates a new instance of the class, sets the property, and writes it to the pipeline. The static class and method are then called directly from inside the script, saving the results to a string which is then written to the pipeline as well.

```
  namespace PowerShellExecutionSample
{
    /// <summary>
    /// Test class object to instantiate from inside PowerShell script.
    /// </summary>
```

```
public class TestObject
{
    /// <summary>
    /// Gets or sets the Name property
    /// </summary>
    public string Name { get; set; }
}

/// <summary>
/// Test static class to invoke from inside PowerShell script.
/// </summary>
public static class TestStaticClass
{
    /// <summary>
    /// Sample static method to call from insider PowerShell script.
    /// </summary>
    /// <returns>String message</returns>
    public static string TestStaticMethod()
    {
        return "Hello, you have called the test static method.";
    }
}

/// <summary>
/// Provides PowerShell script execution examples
/// </summary>
class PowerShellExecutor
{
    /// <summary>
    /// Sample execution scenario 3: Namespace test
    /// </summary>
    /// <remarks>
    /// Executes a PowerShell script synchronously and utilizes classes in the callers namespace.
    /// </remarks>
    public void ExecuteSynchronouslyNamespaceTest()
    {
        using (PowerShell PowerShellInstance = PowerShell.Create())
        {
```

```
            // add a script that creates a new instance of an object from the caller's namespace
            PowerShellInstance.AddScript("$t = new-object PowerShellExecutionSample.TestObject;" +
                                         "$t.Name = 'created from inside PowerShell script'; $t;" +
                                         "$message = [PowerShellExecutionSample.TestStaticClass]::TestStatic

            // invoke execution on the pipeline (collecting output)
            Collection<PSObject> PSOutput = PowerShellInstance.Invoke();
```

```
                // loop through each output object item
                foreach (PSObject outputItem in PSOutput)
                {
                    if (outputItem != null)
                    {
                        Console.WriteLine(outputItem.BaseObject.GetType().FullName);

                        if (outputItem.BaseObject is TestObject)
                        {
                            TestObject testObj = outputItem.BaseObject as TestObject;
                            Console.WriteLine(testObj.Name);
                        }
                        else
                        {
                            Console.WriteLine(outputItem.BaseObject.ToString());
                        }
                    }
                }
            }
        }
    }
}
```



As we can see from the console output, the public static class and non-static class were both available from inside the script for use. The ability to leverage public members from the caller's namespace allows you to create some interesting scenarios calling C# code directly from your PowerShell scripts.

- **Anonymous**
  June 20, 2014
  Hello.Nice article.I would like to ask you,what is the difference between runspace and powershell?When do i use using (var runspace = RunspaceFactory.CreateRunspace()) and when powershell(the way you do it?).I have memory leak issues and still searching for a solution. Thanks!

- **Anonymous**
  June 21, 2014
  @tasos: The runspace factory and the PowerShell class provide similar functionality. Runspace factory was

introduced early (powershell v1). The PowerShell instance class stuff was introduced in v2 I believe. The v2 method is just a little bit easier to use.

- **Anonymous**
  June 22, 2014
  The comment has been removed

- **Anonymous**
  June 24, 2014
  I don't know why you would be having memory leaks. If you are still having problems I would recommend asking for support on the MSDN forums, they should be able to help you over there.
  social.msdn.microsoft.com/.../home

- **Anonymous**
  July 08, 2014
  Excellent and well written article. Love the examples. Thank you!

- **Anonymous**
  August 07, 2014
  And of course you have a downloadable test project to go along with the article...

- **Anonymous**
  August 07, 2014
  @MSTech - just updated the top of the article and posted a link to the solution.

- **Anonymous**
  August 26, 2014
  Outstanding article ... especially appreciated the namespace example.

- **Anonymous**
  September 10, 2014
  Hi Keith, Thanks for the wonderful article. I tried to import and run the code you have supplied above. I am getting the following error message "Could not load file or assembly 'System.Management.Automation, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35' or one of its dependencies. Strong name validation failed. (Exception from HRESULT: 0x8013141A)" could you please help me resolving this. I am able to see the System.Management.Automation.dll in the same location as you mentioned in your code. Thanks in Advance.

- **Anonymous**
  September 10, 2014
  @Dan, I haven't run into that before. Are you running it from the downloadable solution link above? Or did you copy and paste the code snippets into a brand new project of your own? If its a brand new project, what .net framework version did you target?

- **Anonymous**
  September 10, 2014

@Keith I am running the downloadable solution link provided above. I am getting the above error when I am running t.ExecuteSynchronously(); statement.

- **Anonymous**
  September 11, 2014
  @Dan - one thing I would try would be to remove the reference to the assembly, and then re-add it back to the project. Alternatively, create a new project in Visual Studio, add the assembly reference, and try some of the sample code in that new project. See if that helps. If you are still having issues after that I would recommend heading over to the MSDN forums for support.

- **Anonymous**
  October 08, 2014
  Is there any way to specify the user that runs the power shell instance? I'd like to user a different use to that which runs the application.

- **Anonymous**
  October 16, 2014
  Hello, thanks for the article. We're trying to automate some of our PowerShell CmdLets in C#. In PowerShell Cmd we are getting an PSObject which has one more object nested. E.g: $a = Get-XYZ $a has some properties say prop1, prop2 etc. $a.prop1 also has some properties say prop1-1, prop1-2 Please help me in getting values of prop1-1, prop1-2 via C#. we were able to access prop1 and prop2 as PSMemberInfo.

- **Anonymous**
  October 28, 2014
  The comment has been removed

- **Anonymous**
  October 29, 2014
  @Greg - by default the PowerShell instance will be run under the user context of the hosting process. I am not aware of any additional parameters or configuration to provide to the PowerShell class that would allow it to run as a different user. The solution to this problem really depends on the design and purpose of your application. Without more info its hard to say.

- **Anonymous**
  October 29, 2014
  @Shanmuk - If you know the object type, can you try casting the PSObject (or PSObject.BaseObject) into your known class and access the properties that way?

- **Anonymous**
  October 29, 2014
  @Carlos - Unfortunately I don't have any experience with MVC applications, but I have solved a similar problem in WCF web services using caching. What you could do is store a collection of fully initialized PowerShell instances in the web cache. Then when a user makes a request that requires PS, it just grabs one from the cache instead of initializing a brand new one.

- **Anonymous**
  November 19, 2014
  I have installed Powershell 5.0 and i am calling powershell scripts from C# then the first script is executing fine but second script is blocked by first. I got error. Even I tried to set executionpolicy Unrestricted -scope process. Still no luck. My current execution policy is as follows: Scope ExecutionPolicy ----- ---------- MachinePolicy Undefined UserPolicy Undefined Process Unrestricted CurrentUser RemoteSigned LocalMachine Unrestricted
  Please suggest me how to resolve this error.

- **Anonymous**
  November 20, 2014
  @Pushpa, was the 2nd script downloaded from the internet, and potentially blocked (at the file)? Please provide the error message.

- **Anonymous**
  November 22, 2014
  I use the the following code and it sends the output to text box only on completion of entire script. Can we use the same code when executing the power shell script via the aspx page ?,
  PSshell.Commands.AddScript(PSInput); IAsyncResult async = PSshell.BeginInvoke(); Response.Write(async.ToString()); ResultBox_ps.Text += async.ToString() + "rn"; PSshell.EndInvoke(async); foreach (PSObject result in PSshell.EndInvoke(async)) { ResultBox_ps.Text += result.ToString() + "rn"; }

- **Anonymous**
  December 18, 2014
  I had the problem with memory leak either. Then I used RunspaceFactory.CreateOutOfProcessRunspace and memory leak gone.

- **Anonymous**
  February 22, 2015
  thanks for the information but I want to build a cmdlet using the c# which does the work of making a connection. if you know can you please help. thanks in advance

- **Anonymous**
  March 02, 2015
  The comment has been removed

- **Anonymous**
  March 12, 2015
  @dan and for anyone else facing this issue. "Thanks for the wonderful article. I tried to import and run the code you have supplied above. I am getting the following error message "Could not load file or assembly

'System.Management.Automation, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35' or one of its dependencies. Strong name validation failed. (Exception from HRESULT: 0x8013141A)" could you please help me resolving this. I am able to see the System.Management.Automation.dll in the same location as you mentioned in your code." After hours of research, this is what I had to do.

1. update my .csproj file to say <Reference Include="System.Management.Automation, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=MSIL"> <SpecificVersion>False</SpecificVersion> <HintPath>C:Program Files (x86)Reference AssembliesMicrosoftWindowsPowerShell3.0System.Management.Automation.dll</HintPath> </Reference>
2. Target a lower version of .Net. I targeted 4.0 just like in the sample solution. Before I was targeting 4.5, which was wrong. After this, it started working. Good luck and hope you guys find this helpful. www.qtptutorial.net

- **Anonymous**
  April 08, 2015
  for strong name issue here is the solution stackoverflow.com/.../error-loading-system-management-automation-assembly

- **Anonymous**
  April 17, 2015
  Thanks for this article! Really appreciated it. If anyone else had trouble figuring out how to get error output within the Error_DataAdded method see below void Error_DataAdded(object sender, DataAddedEventArgs e) { var records = (PSDataCollection<ErrorRecord>)sender; // do something when an error is written to the error stream Console.WriteLine(records[e.Index].ToString()); }

- **Anonymous**
  May 02, 2015
  Thanks so much for the article :) Can I ask you something, why Am I getting this result? http://1drv.ms/1dDZJOq Thanks in advance, Cheers

- **Anonymous**
  May 07, 2015
  Great Article The only one on the net concernig startig PS async from c#. Could you maybe add an example with Output in a mulitiline TextBox of ListBox in windows form application. I am struggling with it. The First powershell script command result is shown in LIstBox and after that application is freezing... Thanks in advance and regards

- **Anonymous**
  May 14, 2015
  How are you guys finding these memory leaks? I've used the above methods and love it - everything appears to be going great, but I'd like to make sure?

- **Anonymous**
  May 19, 2015
  Hi Keith The link to download appears broken. Could you please update? Many thanks!

- **Anonymous**
  May 20, 2015
  @AMO Just tested the link its working for me.

- **Anonymous**
  July 23, 2015
  Hello Keith, Can you please hep me with how to get the list of Properties associated an PSObject while iterating the PSObject collection returned by below function: string exCmdletResult = ""; private Collection<PSObject> CmdletResults(string PSScriptText) { exCmdletResult = ""; Collection<PSObject> PSOutput = null; PowerShell ps = PowerShell.Create(); ps.AddScript(PSScriptText); try { PSOutput = ps.Invoke(); } catch (Exception ex) { exCmdletResult = "Error: " + ex.Message; } finally { ps.Dispose(); } return PSOutput; } I am calling the same through this button click even on a windows form: private void btnExecute_Click(object sender, EventArgs e) { string cmd_Script = txtCommand.Text; string commandOutput = ""; Collection<PSObject> results = CmdletResults(cmd_Script); if (exCmdletResult.Contains("Error: ")) { txtOutput.Text = exCmdletResult; } else { foreach (PSObject result in results) { // How to get the list of properties when you don't know the command } } txtOutput.Text = commandOutput; } Thanks and regards, Savindra

- **Anonymous**
  July 26, 2015
  The comment has been removed

- **Anonymous**
  July 27, 2015
  can i pass the output from the powershell script into a datatable or a list object? i want 2 columns fullname and name in a way i can check it against a database to see if that file already exists

- **Anonymous**
  August 18, 2015
  I just started doing this. An unhandled exception of type 'System.Management.Automation.PSInvalidOperationException' occurred in System.Management.Automation.dll I think its because Powershell wont allow its access from outside. What should I do ?

- **Anonymous**
  August 31, 2015
  The comment has been removed

- **Anonymous**
  October 21, 2015
  hehe private string spaceCharacter = "` "; and the parameters are for powershell.exe , not the script

- **Anonymous**
  November 05, 2015
  I have everything from your article working but.I can't get the "get-service" to return anything but "System.ServiceProcess.ServiceController". I tried putting the result in a $sc variable and then displaying it. Still nothing. What am I missing? I notice it has the same problem if I substitute "Get-ChildItem" for "Get-Service". Thanks so much for a very helpful article.

- **Anonymous**
  February 22, 2016
  Thank You! Great article!
