Using PowerShell scripts in deployment

As a convention-oriented deployment tool, Deployment Manager automatically takes care of replacing appSettings and connectionStrings, running XML config transforms, and updating IIS websites. Sometimes however you'll need to do more than this - and that's where PowerShell scripts come in.

PowerShell is a powerful scripting language which can be used to customize the behavior of your package deployments. For useful examples of scripts which can be customized to suit your project, take a look at our PowerShell scripts forum.

Scripts

At the root of your NuGet package, you can add any of the following files:

- *PreDeploy.ps1*
- *Deploy.ps1*
- *PostDeploy.ps1*

After extracting your package, the Agent will detect these scripts and invoke them in a custom PowerShell host on the target machine.

> ⚠ If your package contains multiple *PreDeploy.ps1*, *Deploy.ps1*, or *PostDeploy.ps1* scripts, they will all be run during deployment.

After deployment is complete, any extracted PowerShell scripts will be deleted from the target machine.

Variables

Deployment Manager allows you to define variables to parameterize your deployments. These variables, along with some predefined variables, will be available from within your PowerShell script. For example, a variable named *MyApp.ConnectionString* will be available as either:

- `$MyAppConnectionString`
- `$DeploymentManagerParameters["MyApp.ConnectionString"]`

> ⚠ The first form is not case-sensitive, while the second form is case-sensitive. Also, in the first form, non-alphanumeric characters are removed, while in the second they appear just as they appear in the Deployment Manager web interface.

Output

When invoking a command in your PowerShell scripts, you'll need to pipe output to the Write-Host command. For example, this script will not show any output in the Deployment Manager deployment log:

```
Get-ChildItem
```

While this script will show output:

```
Get-ChildItem | Write-Host
```

The same goes for invoking an external executable. For example:

```
& "MyProgram.exe" "-arg1" "-arg2" | Write-Host
```

Examples

This script uses the Service Control tool (*sc.exe*) to create or update a Windows Service. It assumes some variables have been set in the Deployment Manager web interface:

```
# These variables should be set via the Deployment Manager web interface:
#
# ServiceName - Name of the Windows service
# ServiceExecutable - Path to the .exe containing the service
#
# sc.exe is the Service Control utility in Windows
$service = Get-Service $ServiceName -ErrorAction SilentlyContinue
$fullPath = Resolve-Path $ServiceExecutable
if (! $service)
{
Write-Host "The service will be installed"
New-Service -Name $ServiceName -BinaryPathName $fullPath -StartupType Automatic
}
else
{
Write-Host "The service will be stopped and reconfigured"
Stop-Service $ServiceName -Force
& "sc.exe" config $service.Name binPath= $fullPath start= auto | Write-Host
}
Write-Host "Starting the service"
Start-Service $ServiceName
```

This script uses *InstallUtil.exe* to install custom Event Log sources and Windows Services:

```
$NetFrameworkDirectory = $(System.Runtime.InteropServices.RuntimeEnvironment::GetRuntimeDirectory())
& (Join-Path $NetFrameworkDirectory "InstallUtil.exe") "path\to\my.dll" | Write-Host
```

Security

One of the first commands performed by Agent's custom PowerShell host is to automatically set the execution policy to bypass. This means that PowerShell's inbuilt security measures, such as blocking execution of external scripts, will not apply. This also means there is no need to change the default system-wide execution policy when using Agent for deployment.

Also, keep in mind that scripts are executed in the context of the account that the Agent Windows Service runs as. By default this is Local System, which has extensive local privileges, but usually cannot access file shares, remote SQL databases, or other external resources. If you need wider permissions, you'll need to configure Agent to run under a custom service account.