Danny Tuppeny

Formatting PowerShell Objects using the Razor Engine

30 Apr 2011

.NET ASP.NET Open Source PowerShell

I've been meaning to look into PowerShell for a while - it's been on an ever-growing list of things I'd like to learn more about. Recently I got around to spending some time with it and decided an interesting way to learn a little would be to create a module that allowed you to format objects using the Razor Engine.

Edit: When writing this article, I forgot to mention you need to make Powershell run .NET 4 to be able to use Razor. I did this by creating powershell.exe.config and powershell_ise.exe.config files as explained here.

The goal of the module is that you would be able to pipe some objects inand pass some template text, like this:

```
PS> Get-Command | Format-Razor "Command: @Model.Name, @for (int i = 0; i < 5;
i++) { <y>@i</y> }"
```

Ok, so this sample isn't particularly useful, but you should get the idea. The string passed in could be read from a file, and the output could be written to a file. This will allow you to format objects using Razor, which I think it is a pretty cool templating language.

When I opened PowerShell ISE to start coding, I actually expected this to be only a line or two of code, but it turned out to be a little more complicated. Because Razor compiles to code, and you'd likely only want to do this once regardless of how many times the template would be used, it's not as simple as just passing an object and a template into Razor. First we had to create the Razor Engine:

```
# Create an instance of the Razor engine for C#
$language = New-Object System.Web.Razor.CSharpRazorCodeLanguage
$host = New-Object System.Web.Razor.RazorEngineHost($language)
```

```
# Set some default properties for the Razor-generated class
$host.DefaultBaseClass = "TemplateBase" # This is our base class (created
below)
$host.DefaultNamespace = "RazorOutput"
$host.DefaultClassName = "Template"

# Add any default namespaces that will be useful to use in the templates
$host.NamespaceImports.Add("System") | Out-Null

New-Object System.Web.Razor.RazorTemplateEngine($host)
```

This sets up the Razor Engine with some fairly basic settings, including the name and namespace of the generated class, and also the base class. The base class is required to have a few things for Razor to be able to use it:

- A virtual Execute() method
- A Write(object) method, called when Razor outputs a variable/expression
- A WiteLiteral(object) method, called when Razor outputs literal content from the template

Because it's not trivial (or certainly, I couldn't find a way) to create a class inside PowerShell, I cheated a little bit. To avoid including a binary .NET assembly, I compiled the base class from a string on the fly:

```
# HACK: To avoid shipping a DLL, we're going to just compile our TemplateBase
class here
$baseClass = @"
using System.IO;
using System.Text;

public abstract class TemplateBase
{
    public StringBuilder Buffer { get; set; }
    public StringWriter Writer { get; set; }
    public dynamic $modelName { get; set; }

    public TemplateBase()
    {
        this.Buffer = new StringBuilder();
        this.Writer = new StringWriter(this.Buffer);
    }

    public abstract void Execute();

    public virtual void Write(object value)
    {
        WriteLiteral(value);
    }

    public virtual void WriteLiteral(object value)
    {
```

```
        Buffer.Append(value);
    }
}
"@

# Set up the compiler params, including any references required for the
compilation
$codeProvider = New-Object Microsoft.CSharp.CSharpCodeProvider
$assemblies = @(

[System.Reflection.Assembly]::LoadWithPartialName("System.Core").CodeBase.Replac
 ""),

[System.Reflection.Assembly]::LoadWithPartialName("Microsoft.CSharp").CodeBase.R
 "")
)
$compilerParams = New-Object
System.CodeDom.Compiler.CompilerParameters(,$assemblies)

# Compile the template base class
$templateBaseResults =
$codeProvider.CompileAssemblyFromSource($compilerParams, $baseClass);

# Add the (just-generated) template base assembly to the compile parameters
$assemblies = $assemblies +
$templateBaseResults.CompiledAssembly.CodeBase.Replace("file:///", "")
$compilerParams = New-Object
System.CodeDom.Compiler.CompilerParameters(,$assemblies)

# Compile the Razor-generated code
$codeProvider.CompileAssemblyFromDom($compilerParams,
$razorResult.GeneratedCode)
```

This code first compiles the base class into an assembly, and then calls Razor to generate its own code, which is then compiled with a reference to the first assembly, which contains the base template class.

Finally, we create an instance of the Razor-generated class which we'll finally use for the transformation:

```
# Grab the assembly that contains the Razor-generated classes
$assembly = $results.CompiledAssembly

# Create an instance of our Razor-generated class (this name is hard-coded
above)
$type = $assembly.GetType("RazorOutput.Template")
[System.Activator]::CreateInstance($type)
```

To perform a transformation, all we need to do is set the Model property on the Template, and then call the Execute method:

```
# Set the model to the current object, using the $modelName param so the user
can customise it
$template.$modelName = $model

# Execute the code, which writes the output to our buffer
$template.Execute()

# "Return" the output for this item
$template.Buffer.ToString()
```

And that's it, now we can format our objects using a Razor template, as shown earlier:

```
PS> Get-Command | Format-Razor "Command: @Model.Name, @for (int i = 0; i < 5;
i++) { <y>@i</y> }"
```