sake.docs Documentation

Release 0.2.2

Jeff Ogata

November 27, 2015

Contents

Sake is a C# language enabled make system and is used to build the projects that comprise the ASP.NET 5 stack. Sake uses a custom build of the Spark view engine, and additional insight into working with Sake can be gained from reviewing Spark.

Note: I pronounce "Sake" as rhyming with "make". This seems to make the most sense, whether you consider "Sake" to be a blend of "CS make", or "Spark make", and it avoids confusion when discussing it along with psake (PowerShell make), which is pronounced as the Japanese rice wine.

Note: Sake was created by Louis DeJardin. I was not a contributor to the Sake project, and this documentation is based on trial and error, review of the Sake source code and Spark documentation, and looking at Sake's use in the ASP.NET 5 projects.

See also:

Source code for the samples is available on github.

CHAPTER 1

Getting Started

To get started with Sake, create the following two files:

The `build.cmd` file checks for and downloads NuGet if needed, installs the Sake NuGet package if needed, and finally executes Sake specifying `makefile.shade` as the build file.

```
@echo off
cd %~dp0

SETLOCAL
SET NUGET_VERSION=latest
SET CACHED_NUGET=%LocalAppData%\NuGet\nuget.%NUGET_VERSION%.exe

IF EXIST "%CACHED_NUGET%" goto copynuget
echo Downloading latest version of NuGet.exe...

IF NOT EXIST "%LocalAppData%\NuGet" md "%LocalAppData%\NuGet"
@powershell -NoProfile -ExecutionPolicy unrestricted -Command "$ProgressPreference = 'SilentlyContinu

:copynuget
IF EXIST .nuget\nuget.exe goto restore
md .nuget
copy "%CACHED_NUGET%" .nuget\nuget.exe > nul

:restore
IF EXIST packages\Sake goto run
.nuget\NuGet.exe install Sake -ExcludeVersion -Source https://www.nuget.org/api/v2/ -Out packages

:run
packages\Sake\tools\Sake.exe -f makefile.shade
```

`makefile.shade` is a Spark view engine template file that specifies a default build target and writes `Hello world!` to the console.

```
#default
   @{
      Log.Info("Hello world!");
   }
```

Note: Andrew Stanton-Nurse has a Sublime 3 package that adds colorization for `.shade` files: Sublime-Sake

Note: The Spark view engine supports template files using off-side rule formatting where indentation denotes structure, as in Python, Jade, and Haml. These files have a `.shade` file extension to differentate them from `.spark`

template files, which use opening and closing tags for structure.

Run the build:

```
>build.cmd
Attempting to gather dependencies information for package 'Sake.0.2.2' with respect to project 'packa
Attempting to resolve dependencies for package 'Sake.0.2.2' with DependencyBehavior 'Lowest'
Resolving actions to install package 'Sake.0.2.2'
Resolved actions to install package 'Sake.0.2.2'
Adding package 'Sake.0.2.2' to folder 'packages'
Added package 'Sake.0.2.2' to folder 'packages'
Successfully installed 'Sake 0.2.2' to packages
info: Hello world!
```

The build file will restore the Sake nuget package and write out the log message.

Congratulations! You've created your first Sake build.

CHAPTER 2

Working with Sake

2.1 Element Tags and C#

`.shade` files use an offside-rule format, like Python, Jade, or Haml, which means that indentation determines structure. `.shade` templates can contain a mix of element tags and C# code. The concept of element tags in a `.shade` file makes sense when you consider that `.shade` files are templates processed by the Spark view engine into a dynamically generated class. Originally, these classes would have been used to generate a response in a web site (think Razor and `.cshtml` files); Sake repurposes Spark to process `.shade` template files into classes that run a build.

2.1.1 Working with Element Tags

Sake comes with a standard set of element tags, like `exec` and `log`, and you can also create your own *Custom Element Tags*. A `.shade` file will also use element tags from Spark, like `use` and `macro`. See the Spark elements reference for more information.

Note: I have not tried to use all of the Spark elements in a Sake build file, and some may not make sense to use in a build file, or may not work as described in the Spark documentation.

The following `.shade` file illustrates the basics of working with element tags and can be run using the `build.cmd` file from Getting Started. Sake requires at least one target, so we define one here named `#default`. *Targets* are explained in detail later in the documentation.

Interesting things to note:

- Strings are delimited with single or double quotes.
- Element tags that are not indented run before targets.

```
-// example of a single-line comment

-/*
   example of a
   multi-line comment
*/

log warn="This executes first."

#default
   log info='Hello world'

log warn="This also executes before the target."
```

Running the file above produces the following output:

```
warn: This executes first.
warn: This also executes before the target.
info: Hello world
```

2.1.2 Working with C#

C# code can be used as a code block, delimited with `@{` and `}`:

```
@{
   var message = "Hello world!";
   Log.Info(message);
}
```

C# can also appear in element tags, delimited with `${` and `}`:

```
log info="The current date and time is ${DateTime.Now.ToString()}."
```

Note: Version 0.2.2 of Sake targets .NET 4.0, which corresponds to C# 4.

String Delimiters

As with element tags, strings in C# code can be delimited using either single or double quotes. This raises the interesting problem of working with char variables in C#. For example, the following code will generate an exception in a `.shade` file:

```
var tokens = "a,b,c".Split(',');
```

The `','` argument is treated as a string, and an exception will be thrown because `Split` expects a char. To work around this, cast to char:

```
var tokens = "a,b,c".Split((char)',');
```

Namespaces

The `use` element is analogous to the `using` directive in C#. In the example below, `Console` and `Directory` do not need to be fully qualified because the `System` and `System.IO` namespaces are specified by the `use` elements:

```
use namespace="System"
use namespace="System.IO"

#default
   @{
      Console.WriteLine(Directory.GetCurrentDirectory());
   }
```

The following `.shade` file shows the basics of working with C# in Sake, and also how you can work with both C# and tags in the same build file.

```
use namespace="System"

#default

   @{
```

```
    var now = DateTime.Now;

    Console.WriteLine("Hello world using C#!");
}

log info="Hello world using tags!  It is ${now.ToString()}"
```

This produces the following output:

```
>build.cmd
Hello world using C#!
info: Hello world using tags!  It is 11/14/2015 12:24:29 PM
```

2.2 Targets

Sake build steps are organized into targets. Targets are defined in a `.shade` file as an element starting with a `#` and can be set up to be dependent on each other.

An example `.shade` file illustrating the topics presented in this page appears at the end of the page. Be sure to review the `build.cmd` file as it changes slightly to pass a target to Sake.

2.2.1 Default Target

The first target in a `.shade` file is the default target. If Sake is executed without specifying a target, the default target is executed.

2.2.2 Dependencies

To indicate that a target `target-b` depends on `target-a` to run before it, add `.target-a` to the declaration of `target-b`:

```
#target-b .target-a
```

For example:

```
#target-a
   log info='target a'

#target-b .target-a
   log info='target b'

#target-c .target-b
   log info='target c'
```

When run specifying `target-c`, the following output is produced:

```
>build.cmd target-c
info: target a
info: target b
info: target c
```

Dependencies can also be specified from the predecessor by using the `target` attribute:

```
#target-1 target="target-2"
   log info='target 1'

#target-2 target="target-3"
   log info='target 2'

#target-3
   log info='target 3'
```

Running `target-3` executes `target-1` and `target-2` as expected:

```
>build.cmd target-3
info: target 1
info: target 2
info: target 3
```

2.2.3 Multiple Dependencies

To specify multiple dependencies, list them in order in the definition of the target:

```
#target-x
   log info='target x'

#target-y
   log info='target y'

#target-z .target-y .target-x
   log info='target z'
```

Note that `.target-y` appears before `.target-x` in the dependency list, and when `target-z` is run, `target-y` is run before `target-x`:

```
>build.cmd target-z
info: target y
info: target x
info: target z
```

2.2.4 Example

`build.cmd`

```
@echo off
cd %~dp0

SETLOCAL
SET NUGET_VERSION=latest
SET CACHED_NUGET=%LocalAppData%\NuGet\nuget.%NUGET_VERSION%.exe

IF EXIST "%CACHED_NUGET%" goto copynuget
echo Downloading latest version of NuGet.exe...

IF NOT EXIST "%LocalAppData%\NuGet" md "%LocalAppData%\NuGet"
@powershell -NoProfile -ExecutionPolicy unrestricted -Command "$ProgressPreference = 'SilentlyContinu

:copynuget
IF EXIST .nuget\nuget.exe goto restore
```

```
md .nuget
copy "%CACHED_NUGET%" .nuget\nuget.exe > nul

:restore
IF EXIST packages\Sake goto run
.nuget\NuGet.exe install Sake -ExcludeVersion -Source https://www.nuget.org/api/v2/ -Out packages

:run
packages\Sake\tools\Sake.exe -f makefile.shade %*
```

makefile.shade

```
use namespace="System"

#default
   log info='default'

#target-c .target-b
   log info='target c'

#target-a
   log info='target a'

#target-b .target-a
   log info='target b'

#target-3
   log info='target 3'

#target-1 target="target-2"
   log info='target 1'

#target-2 target="target-3"
   log info='target 2'

#target-x
   log info='target x'

#target-y
   log info='target y'

#target-z .target-y .target-x
   log info='target z'
```

2.3 Extending Sake

`.shade` files containing functions, classes, or custom element tags can be imported into the primary build file. These files can be placed in a directory, which is then provided to Sake using the `I` argument. For example, the following command would run Sake with import files in a directory named `imports`:

```
Sake.exe -I imports -f makefile.shade %*
```

2.3.1 Custom Functions

Import files can contain C# functions and classes in a functions code block:

```
use namespace="System"
use namespace="System.Collections.Generic"

functions
   @{
      private List<CustomItem> _items = new List<CustomItem>();

      public void AddCustomItem(string name)
      {
         _items.Add(new CustomItem { Name = name });
      }

      public void PrintCustomItems()
      {
         foreach(var item in _items)
         {
            Console.WriteLine(item.Name);
         }
      }

      public class CustomItem
      {
         public string Name { get; set; }
      }
   }
```

These functions can be included in another `.shade` file using the `import` element:

```
use import="CustomFunctions"

#default
   @{
      AddCustomItem('foo');
      AddCustomItem('bar');
      AddCustomItem('baz');
      PrintCustomItems();
   }
```

Running this produces the following output:

```
>build.cmd
foo
bar
baz
```

2.3.2 Custom Element Tags

Import files can also be used to create custom element tags. To create a custom element, name the file with a leading underscore; the remainder of the file name will then be the element name. Within the file, default values for attributes can be specified, and any attribute values not provided with default values must be provided when the element is used.

The following simple example defines a default value of `"Hello"` for the `greeting` attribute. A value will be required for the `name` attribute when the element is used.

```
default greeting='Hello'

@{
```

```
    Log.Info(greeting + " " + name);
}
```

If the sample above is saved as `_echo.shade`, it can be used in a target like so:

```
#echotag
    echo name="Bob"
```

Running the `echotag` target produces the following output:

```
>build.cmd echotag
info: Hello Bob
```

To use a custom element in C# code, you can define a `macro`:

```
macro name='Echo' name='string' greeting='string'
    echo
```

The macro can then be called as you would a C# function:

```
#echomacro
    @{
        Echo("Jack", "Good morning");
    }
```

2.3.3 Examples

The following files include the code samples in this page. The `build.cmd` file calls Sake specifying an import folder:

```
@echo off
cd %~dp0

SETLOCAL
SET NUGET_VERSION=latest
SET CACHED_NUGET=%LocalAppData%\NuGet\nuget.%NUGET_VERSION%.exe

IF EXIST "%CACHED_NUGET%" goto copynuget
echo Downloading latest version of NuGet.exe...

IF NOT EXIST "%LocalAppData%\NuGet" md "%LocalAppData%\NuGet"
@powershell -NoProfile -ExecutionPolicy unrestricted -Command "$ProgressPreference = 'SilentlyContinu

:copynuget
IF EXIST .nuget\nuget.exe goto restore
md .nuget
copy "%CACHED_NUGET%" .nuget\nuget.exe > nul

:restore
IF EXIST packages\Sake goto run
.nuget\NuGet.exe install Sake -ExcludeVersion -Source https://www.nuget.org/api/v2/ -Out packages

:run
packages\Sake\tools\Sake.exe -I imports -f makefile.shade %*
```

Save the `makefile.shade` file in the same folder as the `build.cmd` file:

```
use import="CustomFunctions"

#default
```

```
   @{
      AddCustomItem('foo');
      AddCustomItem('bar');
      AddCustomItem('baz');
      PrintCustomItems();
   }

#echotag
   echo name="Bob"

#echomacro
   @{
      Echo("Jack", "Good morning");
   }

macro name='Echo' name='string' greeting='string'
   echo
```

Create an `imports` folder within the folder containing the `build.cmd` file and create the following files in it.

`CustomFunctions.shade`:

```
use namespace="System"
use namespace="System.Collections.Generic"

functions
   @{
      private List<CustomItem> _items = new List<CustomItem>();

      public void AddCustomItem(string name)
      {
         _items.Add(new CustomItem { Name = name });
      }

      public void PrintCustomItems()
      {
         foreach(var item in _items)
         {
            Console.WriteLine(item.Name);
         }
      }

      public class CustomItem
      {
         public string Name { get; set; }
      }
   }
```

`_echo.shade`:

```
default greeting='Hello'

@{
   Log.Info(greeting + " " + name);
}
```

CHAPTER 3

Examples

Basic Sake examples will be included here. For comprehensive, real-world examples of Sake build files, see the ASP.NET 5 projects, particularly the Universe project.

3.1 Console Example

This example shows custom functions used to write to the Console in different colors.

`build.cmd`:

```
@echo off
cd %~dp0

SETLOCAL
SET NUGET_VERSION=latest
SET CACHED_NUGET=%LocalAppData%\NuGet\nuget.%NUGET_VERSION%.exe

IF EXIST "%CACHED_NUGET%" goto copynuget
echo Downloading latest version of NuGet.exe...

IF NOT EXIST "%LocalAppData%\NuGet" md "%LocalAppData%\NuGet"
@powershell -NoProfile -ExecutionPolicy unrestricted -Command "$ProgressPreference = 'SilentlyContinu

:copynuget
IF EXIST .nuget\nuget.exe goto restore
md .nuget
copy "%CACHED_NUGET%" .nuget\nuget.exe > nul

:restore
IF EXIST packages\Sake goto run
.nuget\NuGet.exe install Sake -ExcludeVersion -Source https://www.nuget.org/api/v2/ -Out packages

:run
packages\Sake\tools\Sake.exe -I imports -f makefile.shade %*
```

`Console.shade` saved to the `imports` directory:

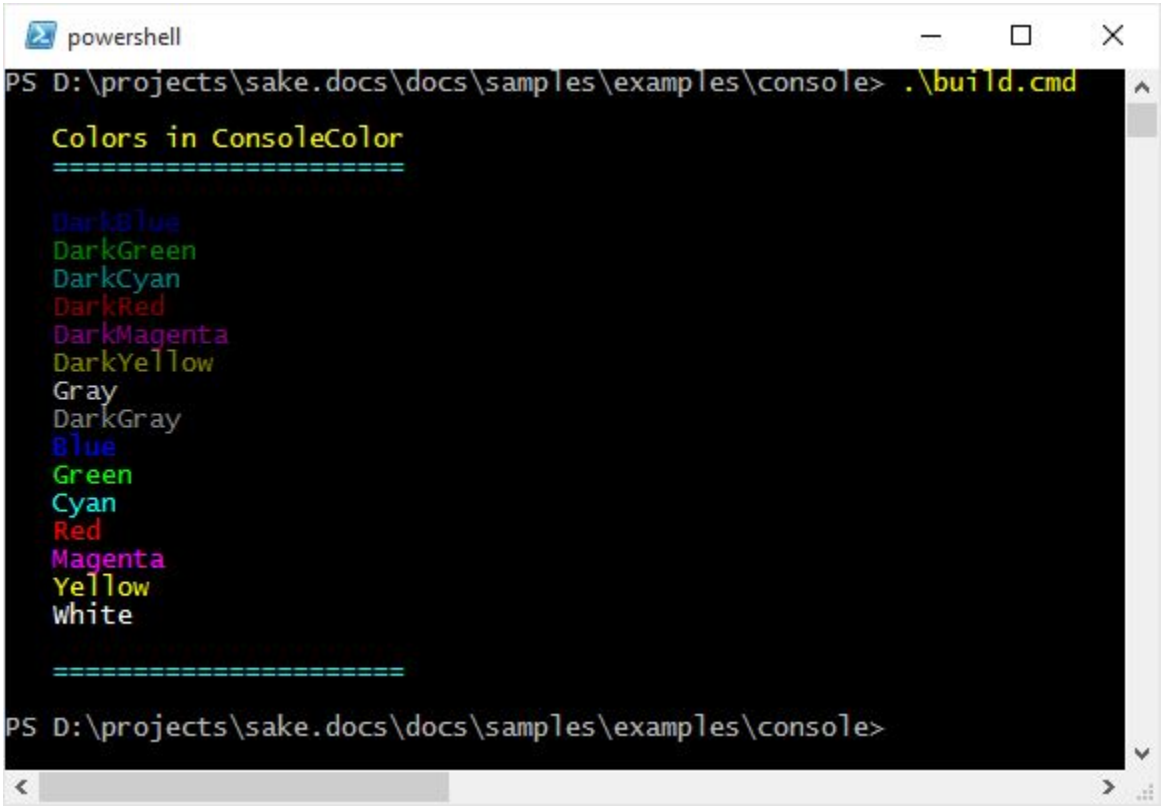
```
use namespace="System"
use namespace="System.IO"

functions
   @{
```

```
    void WriteLine(string text, string colorText)
    {
       ConsoleColor color;

       if (Enum.TryParse<ConsoleColor>(colorText, true, out color))
       {
          WriteLine(text, color);
          return;
       }

       WriteLine(text);
    }

    void WriteLine(string text = null, ConsoleColor? color = null)
    {
       if (text != null && color != null)
       {
          Console.ForegroundColor = color.Value;
       }

       Console.WriteLine(text);

       if (text != null && color != null)
       {
          Console.ResetColor();
       }
    }

    void Write(string text, string colorText)
    {
       ConsoleColor color;

       if (Enum.TryParse<ConsoleColor>(colorText, true, out color))
       {
          Write(text, color);
          return;
       }

       Write(text);
    }

    void Write(string text = null, ConsoleColor? color = null)
    {
       if (text != null && color != null)
       {
          Console.ForegroundColor = color.Value;
       }

       Console.Write(text);

       if (text != null && color != null)
       {
          Console.ResetColor();
       }
    }
 }
```

makefile.shade:

```
use namespace="System.Linq"
use import="Console"

#default
   @{
      WriteLine();
      WriteLine("   Colors in ConsoleColor", "yellow");
      WriteLine("   ======================", "cyan");
      foreach(var color in Enum.GetValues(typeof(ConsoleColor)).Cast<ConsoleColor>())
      {
         Write("   ");
         WriteLine(color.ToString(), color);
      }
      WriteLine();
      WriteLine("   ======================", "cyan");
      WriteLine();
   }
```

Output:



3.2 Help Example

This example shows custom functions and classes used to enumerate the targets in the build and list them in the console. Targets have a `description` attribute, and this example allows for a group to be included in the description, separated from the actual target description using a `|` character. This example makes use of the *Console Example* to output text in color; include `Console.shade` in the `imports` folder if you aren't using the source code from github.

build.cmd:

```
@echo off
cd %~dp0

SETLOCAL
SET NUGET_VERSION=latest
SET CACHED_NUGET=%LocalAppData%\NuGet\nuget.%NUGET_VERSION%.exe

IF EXIST "%CACHED_NUGET%" goto copynuget
echo Downloading latest version of NuGet.exe...

IF NOT EXIST "%LocalAppData%\NuGet" md "%LocalAppData%\NuGet"
@powershell -NoProfile -ExecutionPolicy unrestricted -Command "$ProgressPreference = 'SilentlyContinu

:copynuget
IF EXIST .nuget\nuget.exe goto restore
md .nuget
copy "%CACHED_NUGET%" .nuget\nuget.exe > nul

:restore
IF EXIST packages\Sake goto run
.nuget\NuGet.exe install Sake -ExcludeVersion -Source https://www.nuget.org/api/v2/ -Out packages

:run
packages\Sake\tools\Sake.exe -I imports -f makefile.shade %*
```

Help.shade saved to the imports directory:

```
use namespace="System"
use namespace="System.IO"
use namespace="System.Collections"
use namespace="System.Collections.Generic"
use import="Console"

functions
   @{
      void WriteHelp()
      {
         WriteHelpHeader();

         var groups = GetTargetGroups();

         WriteHelpGroups(groups);
         WriteHelpFooter();
      }

      void WriteHelpHeader()
      {
         WriteLine();
         Write("******************************", ConsoleColor.DarkGreen);
         Write(" HELP ", ConsoleColor.Green);
         WriteLine("******************************", ConsoleColor.DarkGreen);
         WriteLine();
         Write("This build script has the following build ");
         Write("targets", ConsoleColor.Green);
         WriteLine(" set up:");
      }
```

```
TargetGroups GetTargetGroups()
{
   var groups = new TargetGroups();

   foreach(var kvp in Targets)
   {
      var target = kvp.Value;
      var tokens = target.Description.Split((char)'|');

      if (tokens.Length == 2)
      {
         groups.Add(tokens[0], target.Name, tokens[1]);
      }
      else
      {
         groups.AddUngrouped(target.Name, target.Description);
      }
   }

   return groups;
}

void WriteHelpGroups(TargetGroups groups)
{
   // write out any ungrouped targets first
   foreach(var target in groups.UngroupedItems)
   {
      WriteLine();
      Write(" ");
      Write(target.Name, ConsoleColor.Green);
      Write(" = ");
      WriteLine(target.Description);
   }

   // write out groups
   foreach(var group in groups)
   {
      WriteLine();
      Write(" ");
      WriteLine(group.Name, ConsoleColor.DarkGreen);

      foreach(var target in group.Targets)
      {
         Write("  > ");
         Write(target.Name, ConsoleColor.Green);
         Write(" = ");
         WriteLine(target.Description);
      }
   }
}

void WriteHelpFooter()
{
   WriteLine();
   WriteLine(" For a complete list of build tasks, view makefile.shade.");
   WriteLine();
   WriteLine("**********************************************************************", ConsoleC
}
```

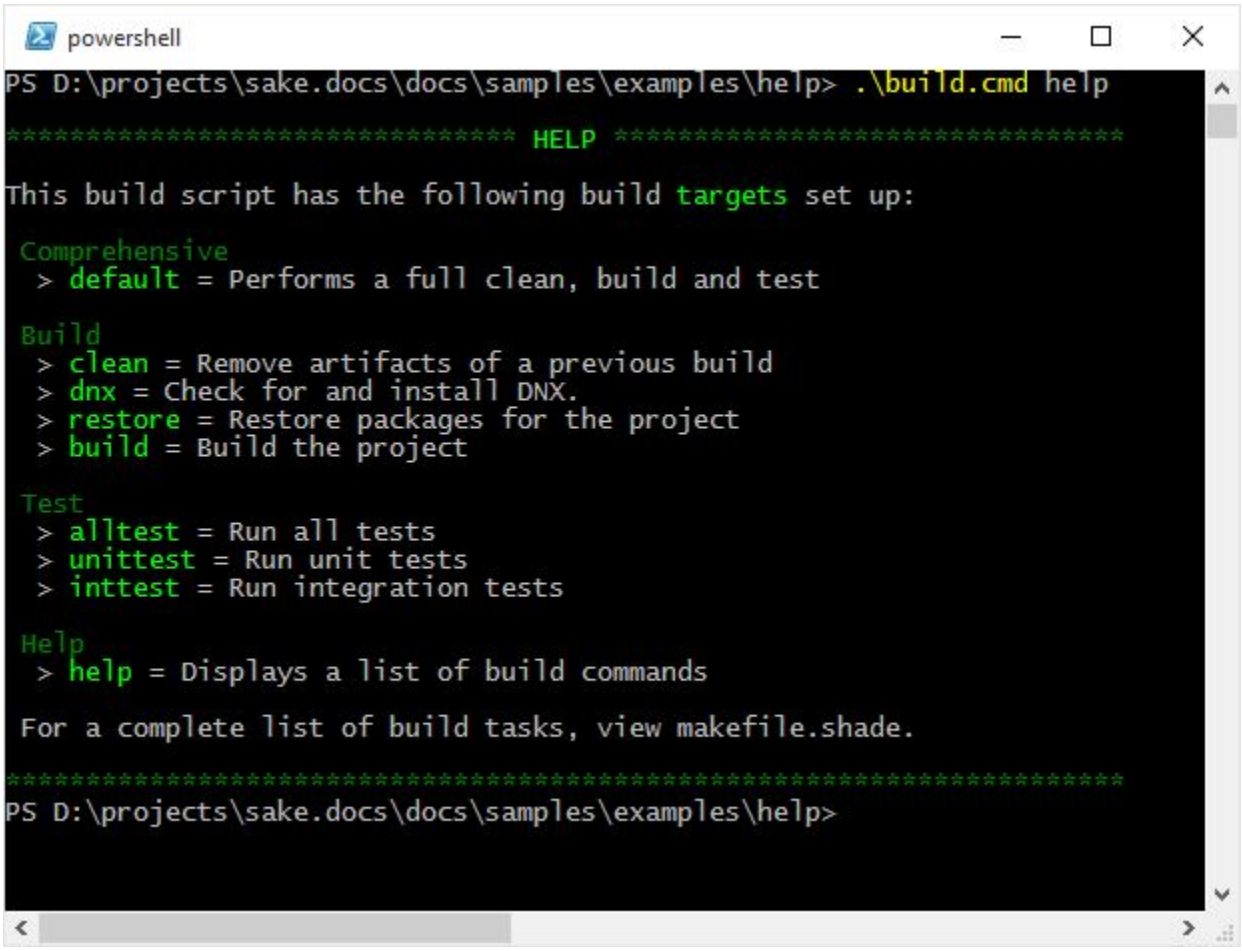
```
public class TargetItem
{
   public TargetItem(string name, string description)
   {
      Name = name;
      Description = description;
   }

   public string Name { get; private set; }

   public string Description { get; private set; }
}

public class TargetGroup
{
   private readonly List<TargetItem> _targets;

   public TargetGroup(string name)
   {
      _targets = new List<TargetItem>();
      Name = name;
   }

   public string Name { get; private set; }

   public List<TargetItem> Targets { get { return _targets; } }

   public TargetItem Add(string name, string description)
   {
      var item = new TargetItem(name, description);
      _targets.Add(item);
      return item;
   }
}

public class TargetGroups : IEnumerable<TargetGroup>
{
   private readonly Dictionary<string, TargetGroup> _groups;
   private readonly List<TargetItem> _ungrouped;

   public TargetGroups()
   {
      _groups = new Dictionary<string, TargetGroup>();
      _ungrouped = new List<TargetItem>();
   }

   public List<TargetItem> UngroupedItems
   {
      get { return _ungrouped; }
   }

   public TargetGroup Add(string groupName, string itemName, string itemDescription)
   {
      var group = _groups.ContainsKey(groupName) ? _groups[groupName] : null;

      if (group == null)
      {
         group = new TargetGroup(groupName);
```

```
                _groups.Add(group.Name, group);
            }

            group.Add(itemName, itemDescription);

            return group;
        }

        public TargetItem AddUngrouped(string itemName, string itemDescription)
        {
                var item = new TargetItem(itemName, itemDescription);
            _ungrouped.Add(item);
            return item;
        }

        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
        {
            return this.GetEnumerator();
        }

        public IEnumerator<TargetGroup> GetEnumerator()
        {
            foreach(var kvp in _groups)
            {
               yield return kvp.Value;
            }
        }
    }
}
```

makefile.shade:

```
use import="Console"
use import="Help"

#default description="Comprehensive|Performs a full clean, build and test"

#clean description="Build|Remove artifacts of a previous build"

#dnx description="Build|Check for and install DNX."

#restore description="Build|Restore packages for the project"

#build description="Build|Build the project"

#alltest description="Test|Run all tests"

#unittest description="Test|Run unit tests"

#inttest description="Test|Run integration tests"

#help description="Help|Displays a list of build commands"
@{
    WriteHelp();
}
```

Output:



3.3 MSBuild Example

The `_build` element that comes with Sake is written to use MSBuild 4.0. If your source code uses features of C# 5 or 6, this may not work. For example, when building an application that uses string interpolation, which was added in C# 6, the build fails indicating that `$` is an unexpected character.

This example shows a custom element, based on `_build`, which uses MSBuild 14.0 (VS 2015, C# 6) or MSBuild 12.0 (VS 2012/13, C# 5).

The solution that this example builds can be found along with the other files on github. Note in the example `makefile.shade`, the output directory is relative the `.csproj` files.

`build.cmd`:

```
@echo off
cd %~dp0

SETLOCAL
SET NUGET_VERSION=latest
SET CACHED_NUGET=%LocalAppData%\NuGet\nuget.%NUGET_VERSION%.exe

IF EXIST "%CACHED_NUGET%" goto copynuget
echo Downloading latest version of NuGet.exe...
```

```
IF NOT EXIST "%LocalAppData%\NuGet" md "%LocalAppData%\NuGet"
@powershell -NoProfile -ExecutionPolicy unrestricted -Command "$ProgressPreference = 'SilentlyContinu

:copynuget
IF EXIST .nuget\nuget.exe goto restore
md .nuget
copy "%CACHED_NUGET%" .nuget\nuget.exe > nul

:restore
IF EXIST packages\Sake goto run
.nuget\NuGet.exe install Sake -ExcludeVersion -Source https://www.nuget.org/api/v2/ -Out packages

:run
packages\Sake\tools\Sake.exe -I imports -f makefile.shade %*
```

_msbuild.shade saved to the `imports` directory:

```
@{/*

build
    Executes msbuild to compile your project or solution

projectFile=''
    Required. Path to the project or solution file to build.

configuration='Release'
    Determines which configuration to use when building.

outputDir=''
    Directs all compiler outputs into the target path. Note:  this will be relative to the project fi

extra=''
    Additional commandline parameters for msbuild

*/}

default configuration='Release'
default outputDir=''
default extra=''

use namespace="System"
use namespace="System.IO"
use namespace="System.Reflection"

var buildProgram=''

@{
   Assembly buildUtilities = null;
   string toolsVersion = null;

   try
   {
      buildUtilities = Assembly.Load("Microsoft.Build.Utilities.Core, Version=14.0.0.0, Culture=neutr
      toolsVersion = "14.0";
   }
   catch
   {
      buildUtilities = Assembly.Load("Microsoft.Build.Utilities.v12.0, Version=12.0.0.0, Culture=neut
```

```
      toolsVersion = "12.0";
   }

   var helper = buildUtilities.GetType("Microsoft.Build.Utilities.ToolLocationHelper");
   var method = helper.GetMethod("GetPathToBuildTools", new Type[] { typeof(string) } );
   var path = method.Invoke(helper, new object[] { toolsVersion }).ToString();

   buildProgram = Path.Combine(path, "msbuild.exe");
}

var OutDirProperty=''
set OutDirProperty='OutDir=${outputDir}${Path.DirectorySeparatorChar};' if='!string.IsNullOrWhiteSpac

exec program="${buildProgram}" commandline='${projectFile} "/p:${OutDirProperty}Configuration=${confi
```

makefile.shade:

```
#default .build

#clean
   msbuild projectFile="src/SakeMsBuild.sln" outputDir="../../output" extra="/t:Clean"

#build .clean
   msbuild projectFile="src/SakeMsBuild.sln" outputDir="../../output" extra="/t:Rebuild /m"
```

Output: