iControlsh - An iControl Based Console BIG-IP Shell

Joe Pruitt, 2010-20-10

I've been toying with the idea of building an iControl shell for a long time. For those who don't know what I'm talking about, a shell is essentially a fully enclosed application that runs within a console on your computer. Windows has the Command Prompt and Unix has it's various shells (sh, bash, ksh, etc) and since most of my working day involves interacting with a shell of some sort, I figured it might be useful to create an interactive BIG-IP shell.

For this example, I picked PowerShell as my language of choice for a couple of reasons: it already is a shell, I have tons of reference samples already, and it's very extensible.

The Methodology

I wanted to design this application in such a way that it could be actively developed on without having to mess with the core engine. That way I could add more supported components and methods down the road in a very easy manner.

PowerShell has the concept of user defined "functions" and that ended up being the basis of my modular design. At the core of the code is a input processor that receives commands from the user and distributes them to the various registered components. It looks something like this:

```
while(1)
{
  $prompt = "$(Get-CurrentPath)> ";
  $i = (Read-Host $prompt).Trim();

  Debug-Message "< $($MyInvocation.MyCommand.Name) '$i' >";
  if ( $i.Length -gt 0 )
  {
    Debug-Message "CommandLine: '$i'...";
    switch -Wildcard ($i.ToLower())
    {
      ""   { break; }
      "c*" { Initialize-Connection $i; }
      "h*" { Show-Help -Full -Topic (Get-Arguments $i); }
      "i"  { Show-Information; }
      "q"  { Exit-Console; }
      "r"  { Invoke-Expression $LoadModStr; }
      "d"  { $script:DEBUG = -not $script:DEBUG; }
      "!*" {
        $cmd = $i.Replace("!", "").Trim();
        Debug-Message "Executing command $cmd";
        Invoke-Expression $cmd;
      }
      ".." { Move-Up; }
      default {
        if ( $script:Connected )
        {
          if ( Is-SubDirectory $i )     { Move-Down $i; }
          elseif ( Is-ValidCommand $i ) { Process-Command $i; }
          else                          { Show-Help; }
        }
        else
        {
          Show-Help;
        }
      }
    }
  }
}
```

You'll see that there are some other functions that I've written in there that do various things.

Get-CurrentPath

I wanted to include some sort of navigation into this shell to give the user some context as to where they were in the current command structure. This is very similar to how the various systems prompts will work with your current directory you are in. I opted for the format of icsh:/module/interface for my navigation format.

```
theboss.dev.net> : ?
            Child Objects
            ----------------
```

```
             gtm
             ltm
             networking
             system

             Commands In this object
             -----------------------
             gtm/wideip/list
             ltm/pool/list
             ltm/pool/members
             ltm/pool/stats
             ltm/poolmember/disable
             ltm/poolmember/enable
             ltm/poolmember/list
             ltm/virtualserver/list
             networking/vlan/list
             system/systeminfo/list
theboss.dev.net> : gtm
theboss.dev.net/gtm> : wideip
theboss.dev.net/gtm/wideip> : ..
theboss.dev.net/gtm> : ..
theboss.dev.net> : ltm
theboss.dev.net/ltm> : pool
theboss.dev.net/ltm/pool> : ..
theboss.dev.net/ltm> : poolmember
theboss.dev.net/ltm/poolmember> : ..
theboss.dev.net/ltm> : ..
theboss.dev.net> : system
theboss.dev.net/system> : systeminfo
```

Initialize-Connection

This function will ensure that the iControl PowerShell subsystem is configured and will initialize the connection to the BIG-IP.

Show-Help

This function presents the global commands as well as any child objects (ie. “ltm” has “pool”, “poolmember”, “virtualserver”, etc") and the available sub commands.

Show-Information

This is a diagnostic function to print out shell version and various internal variable related to the state of the shell.

Exit-Console

This is a wrapper function to ensure that the users connection information is persisted to the registry so that it can be remembered the next time the shell is started.

Move-Up

Navigate up an object in the hierarchy.

Move-Down

Navigate down to a child of the current object.

Process-Command

This is where the fun happens. But, before I do, it will be a good idea to talk about how objects and methods are injected into the shell. There is a child directory where the main script resides called Modules. In there I’ve created a folder hierarchy to match the object hierarchy in the iControl API. The file system looks like this

```
c:\icontrolsh\iControlsh.ps1
c:\icontrolsh\modules\gtm
c:\icontrolsh\modules\gtm\wideip.ps1
c:\icontrolsh\modules\ltm
c:\icontrolsh\modules\ltm\pool.ps1
c:\icontrolsh\modules\ltm\poolmember.ps1
c:\icontrolsh\modules\ltm\virtualserver.ps1
c:\icontrolsh\modules\networking
c:\icontrolsh\modules\networking\vlan.ps1
```

```
c:\icontrolsh\modules\system
c:\icontrolsh\modules\system\systeminfo.ps1
```

The script does some file system navigation to determine all the unique “modules” in the modules directory and then for each module, it “dot sources” the child .ps1 files. “dot sourcing” is essentially running the script and it allows for any variable or function declarations to be included in the current runspace. The content for the ltm\virtualserver.ps1 script is as follows:

```
# Global Script variables

if ( -not $script:ICONTROLSH )
{
    Write-Host "This script is not intended to run standalone.  Exiting...";
}

#--------------------------------------------------------------------------------
function icsh/ltm/virtualserver/list()
#--------------------------------------------------------------------------------
{
  <#
    .SYNOPSIS
    This function allows you to query the list of the virtual servers on a BIG-IP LTM.

    .DESCRIPTION
    This function allows you to query the list of the virtual servers on a BIG-IP LTM.

    .EXAMPLE
    list

    .NOTES
    NAME: icsh/ltm/virtualserver/list
    AUTHOR: Joe Pruitt, F5

    .LINK
    http://devcentral.f5.com
  #>
  param($Opts);

  Debug-Message "< $($MyInvocation.MyCommand.Name) '$Opts' >";
  $list = (Get-F5.iControl).LocalLBVirtualServer.get_list();
  Write-Host "Virtual Server List";
  Write-Host "-------------------";
  foreach($pool in $list)
  {
    Write-Host "$pool";
  }
}
```

PowerShell has a built-in help system that I’m able to make use of by including metadata within the <# and #> markers. You’ll notice that I included a check for a script variable that’s set in the iControlsh.ps1 script so that this script isn’t accidentally run on it’s own.

After all the scripts are “dot sourced”, the builtin Get-Command cmdlet can be used to list all of the defined iControl shell functions.

```
theboss.dev.net> : !! Get-Command icsh*
CommandType     Name
-----------     ----
Function        icsh/gtm/wideip/list
Function        icsh/ltm/pool/list
Function        icsh/ltm/pool/members
Function        icsh/ltm/pool/stats
Function        icsh/ltm/poolmember/disable
Function        icsh/ltm/poolmember/enable
Function        icsh/ltm/poolmember/list
Function        icsh/ltm/virtualserver/list
Function        icsh/networking/vlan/list
Function        icsh/system/systeminfo/list
```

This get’s us back to the Process-Command function. With a little string manipulation it’s fairly easy to parse the function names for valid child elements as well as invoking the method calls with the PowerShell Invoke-Expression cmdlet.

```
#--------------------------------------------------------------------------------
function Process-Command()
#--------------------------------------------------------------------------------
{
  param($cmd);
  Debug-Message "< $($MyInvocation.MyCommand.Name) >";
  $tokens = $cmd.Split(" ");
  $command = $tokens[0];
```

```
 9:   $args = "";
10:   for($i=1; $i-lt$tokens.Length; $i++)
11:   {
12:     if ( $i -gt 1 ) { $args += " "; }
13:     $args += $tokens[$i];
14:   }
15:   $key = $(Get-CurrentPath) + "/" + $command;
16:   $code = Gen-FunctionName $command;
17:
18:   Debug-Message "Processing command: '$code' '$args'";
19:   if ( $args.Length -gt 0 ) { $code += " -Opts '$args'"; }
20:   Debug-Message "Invoking code <$code>";
21:   Invoke-Expression $code;
22: }
```

The Full Source

The source for this application can be found in the iControl CodeShare under “PsBigipInteractiveShell”.

Video Walkthrough

There’s a lot of code to review in an article like this and I’m going to leave it to you all to view the source in the wiki. In the meantime, you can watch this short video I’ve recorded illustration a sample session within the shell.

BIG-IP Powershell iControl Console Shell

Related Articles on DevCentral

- F5 DevCentral > Hot Topics > iControl
- What Does BIG-IP Stand For?
- F5 DevCentral > Community > Group Details - Microsoft PowerShell ...
- Audio White Paper - Managing BIG-IP Devices With HP and Microsoft ...
- DevCentral Wiki: iControl Wiki Home
- F5 DevCentral > Community > Group Details - iControl Assembly
- Self-Contained BIG-IP Provisioning with iRules and pyControl ...
- Java iControl Objects - LTM Pool Member > DevCentral > F5 ...
- Audio White Paper - F5 iControl
- Introducing PoshBing – The PowerShell library for Microsoft's Bing ...

Technorati Tags: shell, iControl, PowerShell, BIGIP

F5 Networks, Inc. | 401 Elliot Avenue West, Seattle, WA 98119 | 888-882-4447 | f5.com

F5 Networks, Inc.
Corporate Headquarters
info@f5.com

F5 Networks
Asia-Pacific
apacinfo@f5.com

F5 Networks Ltd.
Europe/Middle-East/Africa
emeainfo@f5.com

F5 Networks
Japan K.K.
f5j-info@f5.com