Classic SysAdmin: Writing a Simple Bash Script

This is a classic article written by Joe "Zonker" Brockmeier from the Linux.com [http://linux.com/] archives. For more great SysAdmin tips and techniques check out our free intro to Linux course [https://www.edx.org/course/introduction-to-linux?utm_medium=partner-marketing&utm_source=affiliate&utm_campaign=linuxfoundation&utm_content=blog-lfs101] .

The first step is often the hardest, but don't let that stop you. If you've ever wanted to learn how to write a shell script but didn't know where to start, this is your lucky day.

If this is your first time writing a script, don't worry — shell scripting is not that complicated. That is, you can do some complicated things with shell scripts, but you can get there over time. If you know how to run commands at the command line, you can learn to write simple scripts in just 10 minutes. All you need is a text editor and an idea of what you want to do. Start small and use scripts to automate small tasks. Over

Starting Off

Each script starts with a “shebang” and the path to the shell that you want the script to use, like so:

```
#!/bin/bash
```

The “#!” combo is called a shebang [http://en.wikipedia.org/wiki/Shebang_%28Unix%29] by most Unix geeks. This is used by the shell to decide which interpreter to run the rest of the script, and ignored by the shell that actually runs the script. Confused? Scripts can be written for all kinds of interpreters — bash, tsch, zsh, or other shells, or for Perl, Python, and so on. You could even omit that line if you wanted to run the script by sourcing it at the shell, but let’s save ourselves some trouble and add it to allow scripts to be run non-interactively.

What’s next? You might want to include a comment or two about what the script is for. Preface comments with the hash (#) character:

```
#!/bin/bash
# A simple script
```

Let’s say you want to run an rsync [http://www.linux.com/learn/tutorials

```
#!/bin/bash
# rsync script
rsync -avh --exclude="*.bak" /home/user/Documents/ /media/diskid/user_backup/Docume
```

Save your file, and then make sure that it's set executable. You can do this using the chmod utility, which changes a file's mode. To set it so that a script is executable by you and not the rest of the users on a system, use "chmod 700 scriptname" — this will let you read, write, and execute (run) the script — but only your user. To see the results, run ls -lh scriptname and you'll see something like this:

```
-rwx------ 1 jzb jzb   21 2010-02-01 03:08 echo
```

The first column of rights, rwx, shows that the owner of the file (jzb) has **r**ead, **w**rite, and e**x**ecute permissions. The other columns with a dash show that other users have no rights for that file at all.

Variables

The above script is useful, but it has hard-coded paths. That might not be a problem, but if you want to write longer scripts that reference paths often, you probably want to utilize variables. Here's a quick sample:

```
rsync -avh --exclude="*.bak" $SOURCEDIR $DESTDIR
```

There's not a lot of benefit if you only reference the directories once, but if they're used multiple times, it's much easier to change them in one location than changing them throughout a script.

Taking Input

Non-interactive scripts are useful, but what if you need to give the script new information each time it's run? For instance, what if you want to write a script to modify a file? One thing you can do is take an argument from the command line. So, for instance, when you run "script foo" the script will take the name of the first argument (foo):

```
#!/bin/bash

echo $1
```

Here bash will read the command line and echo (print) the first argument — that is, the first string after the command itself.

You can also use read to accept user input. Let's say you want to prompt a user for input:

That script will wait for the user to type in their name (or any other input, for that matter) and use it as the variable $name. Pretty simple, yeah? Let's put all this together into a script that might be useful. Let's say you want to have a script that will back up a directory you specify on the command line to a remote host:

```
#!/bin/bash

echo -e "What directory would you like to back up?"
read directory

DESTDIR=
 This e-mail address is being protected from spambots. You need JavaScript enabled
 :$directory/

rsync --progress -avze ssh --exclude="*.iso" $directory $DESTDIR
```

That script will read in the input from the command line and substitute it as the destination directory at the target system, as well as the local directory that will be synced. It might look a bit complex as a final script, but each of the bits that you need to know to put it together are pretty simple. A little trial and error and you'll be creating useful scripts of your own.

Of course, this is just scraping the surface of bash scripting. If you're

linux?utm_medium=partner-marketing&utm_source=affiliate&utm_campaign=linuxfoundation&utm_content=blog-lfs101]

APRIL 2, 2022 / BY THE LINUX FOUNDATION

Share this entry