SEIS 665 Assignment 3: Shell scripting

Overview

Last week we launched a Linux server on AWS and worked with several basic Linux and Git commands. We will expand on last week's learnings in this assignment. Read through the entire assignment first before launching a new server.

Automation is a critical component in modern IT services. Gone are the days when IT professionals would carefully hand-craft IT solutions. Some enterprises may still engage in these labor intensive practices, but more and more organizations recognize the benefits of investing in automation.

Automation also aligns well with DevOps practices such as "infrastructure as code", where infrastructure is created by writing code. You might feel a little nervous about writing code, especially if you don't have a background in software development. Don't worry. The level of code we write when automating infrastructure is generally more basic than the code written for software applications. You just need to get comfortable with a few simple practices.

You will notice that some of the tasks in this assignment are less prescriptive than last week. In other words, instead of telling you exactly how to perform a task, you will be expected to perform the task based on knowledge you gained from the previous assignment. Try to perform a task without looking at previous notes first. If you make a mistake, no worries. We all learn by making mistakes.

Requirements

You need to have a personal AWS account and GitHub account for this assignment. You should have also read the *Git Hands-on Guide* and *Linux Hands-on Guide* by this point in the course.

The assignment

Ready? Let's go!

Launch Linux server

The first step in the assignment is to launch a Linux server using AWS EC2. You should follow the same process as the previous assignment. Refer back to the assignment notes if you need to review the launch process.

The server should have the following characteristics:

- Amazon Linux AMI 64-bit (usually the first option listed)
- Located in a U.S. region (us-east-1 or us-west-1)

- t2.micro instance type
- All default instance settings (storage, vpm, security group, etc.)

Note that when you are launching the server, you can elect to use the same key pair you used last week or you can create a new key pair for this server. Either option is fine. From a practical standpoint, it might make sense to reuse the same key pair each time so that you don't have to keep track of which key pair you used to launch a particular server.

Tip	Taking a break It's recommended that you complete this assignment in one sitting. But if you need to work on it over the course of a couple days, that's no problem. You can always `stop` your EC2 instance when you need to take a break, and then `start` the instance back up when you are ready to work again. The EC2 instances we are creating save our changes even when the instances are shutdown. Just don't `terminate` the instance, because then you will lose all of your work and have to start all over again. AWS does not bill you for instances that are stopped. Well, technically they bill you for the storage costs of the instance, but that amounts to pennies per month.

Log into server

The next step is to log into the Linux server using a terminal program with secure shell (SSH) support. You will need to have the server key and the public IP address before attempting to log into the server.

More Shell Script Fun

Before we dive into the assignment, let's update the software packages on the Linux instance by running a `yum update`:

```
$ sudo yum update -y
```

Remember, it's a good practice to always update the Linux software packages immediately after building the server. Amazon Linux is a *rolling* release, which means that the Linux image is being rebuilt with newer packages several times a week. This means that when you launch an Amazon Linux instance the software is very current.

We will also use Git in this assignment, so install the required software packages:

```
$ sudo yum install git -y
```

Do you remember which directory you are automatically placed into when you initially log into a Linux server? Your home directory. Remember, each user on a Linux system has their own unique home directory. Verify the location:

```
$ pwd
```

You should see that you are currently in the `/home/ec2-user` directory.

Next, we're going to create a sub-directory in our home directory to store this week's assignment files. Let's call it `project1`. Go ahead and create this directory now, and change to the new directory.

Okay, you should be currently in the new directory. Let's create a new Git repository in this directory. Look back at previous notes if you don't remember how to initialize a new Git repository.

List the directory contents to verify that a `.git` directory exists. Remember, the `.git` directory contains the actual Git repository. The directory starts with a dot, so you can't see it if you just use the basic `ls` command. You also need to supply some option flags to the `ls` command to view hidden files and sub-directories.

Create a script file named `script1` and enter the following bash script code into the file:

```
#!/bin/bash

# assign first command line argument
ARG1=${1}

# display first provided argument
echo $ARG1
```

You can use `nano` or whatever text editor you are comfortable with to create the file. Remember, once you save the file you will need to make it executable:

```
$ chmod u+x script1
```

During the remainder of this assignment, when you are asked to create a script file you will need to remember to make the script executable.

Now, let's run the script by supplying an argument:

```
$ ./script1 foo
```

Notice that the response from the script is `foo`. The name of the script is script1 and any terms we enter after the name of the script are called arguments. We only provided one argument in this case, but we could provide a whole list of arguments if we wanted. Arguments provide a method to input data into a script. We can have the script perform different tasks based on the data we input.

In the script code, the `ARG1` variable is assigned to a funny looking value called `${1}`. This value represents the first argument listed after the command. Technically, the command name is also treated like an argument. Its value is represented by `${0}`. How do you think the second argument value is represented? Right, using `${2}`.

Let's save our hard work by committing it to the Git repository.

```
$ git add script1
$ git commit -m "my script1 bash file"
```

Next, make a new script file called `script2` and enter the code:

```
#!/bin/bash

# assign the action
ACTION=${1:-launch}

# display first provided argument
if [ -z "$1" ]
        then
                echo "No argument supplied, default action is: $ACTION"
        else
                echo "Initiating $ACTION."
fi
```

Run the code without supplying an argument:

```
$ ./script2
```

Did you get a permission denied error when trying to run the script? Hint: make sure it is executable.

Now, run the script again with an argument:

```
$ ./script2 update
```

What happened here? A couple things are going on with this script. Note how the `ACTION` variable assignment contains the strange looking `:-` operator. This variable assignment basically says "assign the value of the first argument to the variable ACTION, but if the argument doesn't exist then use `launch` as the default value of the variable".

The if-then-else code structure performs an initial test `[ -z "$1" ]` which checks to see if the first argument is an empty string. This test evaluates as true if the first argument doesn't exist. Note, the spaces in this code are very important. The bash interpreter will not understand the code statement properly if the spaces are left out.

One of the key use-cases for this type of script code is building up a set of arguments that can be used to control the execution of the script. When a user doesn't provide a value for a required argument, sometimes a script needs to use a default value instead.

Go ahead and add this file to the Git repository and make another Git commit. Use an appropriate commit message during your commit.

Enter the `git log` command to verify that your local git repository now contains two commits.

Next, create a new script file called `script3`. The script code is a little longer than previous scripts. Here is the code:

```
#!/bin/bash

# assign variables
ACTION=${1}

function display_help() {

cat << EOF
Usage: ${0} {-h|--help}

OPTIONS:
        -h | --help    Display the command help

Examples:
        Display help:
                $ ${0} -h

EOF
}

case "$ACTION" in
        -h|--help)
                display_help
                ;;
        *)
        echo "Usage ${0} {-h}"
        exit 1
esac
```

Okay, take a break. There are several new concepts introduced in this script file. First, notice the function definition for `display_help()`. We can use functions in script files to encapsulate a set of code which we may want to execute one or more times. You can build complex shell scripts by incrementally adding functions. The function is called during the shell execution by simply referring to the function name `display_help`.

When you execute the script, the bash interpreter skips over all the commands encapsulated in the `display_help()` function definition (everything between the curly brackets). The interpreter won't execute the commands in the function definition until the function is actually called.

After skipping over the function definition, the interpreter reaches the `case` statement. You probably learned in earlier programming classes that a case statement provides an easy way to compare a variable to a set of values. If the variable matches a value then the interpreter executes a specified set of statements. The case statement block begins with the `case` command and ends with the `esac` command (case spelled backwards).

In this case statement, if the `ACTION` variable matches the value `-h` **or** the value `--help` (the or operator is signified by the vertical bar character `|`), the `display_help` function is called. Otherwise, if the ACTION variable doesn't match anything (denoted by the `*` character), a command usage statement is displayed to the user and the script exits. Note that exiting a script with the value of 1 is the appropriate way to signify that the script terminated improperly.

There's one stranger looking thing going on with this script. Look at the `display_help` function code. What's up with this `cat << EOF` stuff? That's just a trick which is used to output multiple text lines to the terminal. We are redirecting the input to the `cat` command and inputing all the lines of text between the two `EOF` tags.

Let's play with this script a little bit. Type:

```
$ ./script3
```

Since you didn't provide any arguments the script helpfully displayed its proper usage.
The `case` statement didn't match the value of the first argument to `-h` or `--help`, so it chose the default match (`*`).

Type this in:

```
$ ./script3 -h
```

Now the script displays the help information for the command. The `case` statement matched the argument value to `-h` and executed the `display_help` function. The function displayed all the text between the two `EOF` tags on the terminal. Easy!

You can now see how it's possible to build up increasingly complex bash scripts by simply adding more argument options and related functions. That's exactly what we're going to do next. Before we do that, add the `script3` file to the Git repository and make another commit.

During the past two assignments, we've been adding and modifying script files in our Git repository on the same branch — the **master** branch. Typically, you don't want to edit and make changes to the files on the master branch. You should do all of your coding and testing in a separate branch, usually a feature branch. Once you have successfully modified your code then you can merge it back into the master branch. A typical development workflow contains many of these branching and merging activities. Let's start following that practice now.

Start by creating a new branch:

```
$ git checkout -b feature/script3
```

This is a nice shortcut command which creates a new branch called `feature/script3` and immediately checks it out. If you type:

```
$ git branch
```

You will see that the `feature/script3` branch is currently checked out (denoted by the highlighting and asterisk). Git doesn't care about forward-slashes (`/`) in the branch name, and these are commonly used to help categorize the purpose of the branch. In this case, we are going to add more functionality to the `script3` file.

Modify the `script3` file so that it looks like this:

```
#!/bin/bash

# assign variables
ACTION=${1}

function create_file() {

touch "${1}-12345"
}

function display_help() {

cat << EOF
Usage: ${0} {-c|--create|-h|--help} <filename>

OPTIONS:
        -c | --create   Create a new file
        -h | --help     Display the command help

Examples:
        Create a new file:
                $ ${0} -c foo.txt

        Display help:
                $ ${0} -h

EOF
}

case "$ACTION" in
        -h|--help)
                display_help
                ;;
        -c|--create)
                create_file "${2:-server}"
                ;;
        *)
        echo "Usage ${0} {-c|-h}"
        exit 1
esac
```

Next, test out the script by running the command:

```
$ ./script3 -c foo
```

List the contents of the current directory. You should see a new empty file named `foo-12345`. Now, run the command again without providing a filename:

```
$ ./script3 -c
```

You should see a file in the current directory named `server-12345`.

We expanded the bash script by adding a command that allows the user to create a file. Adding the command required a couple basic steps:

- Add the new command (`-c|--create`) to the `case` statement
- Add a new function called `create_file`
- Modify the displayed usage statement
- Modify the `display_help` function to display information on the new option

One additional trick we are using in this script is passing the second argument from the command line into the `create_file` function. This function really works just like a command, so it can accept arguments as well. The filename is passed as the second argument on the command line, but when it is passed to the `create_file` function (`create file "${2:-server}"`) it becomes the **first** argument in the function: `touch "${1}-12345"`.

Remove the `foo-12345` and `server-12345` files from the current directory. If your script is working properly, go ahead and add it to the repository and commit it.

Note, a shortcut to add all the files that have changed in the current working directory to the repository is:

```
$ git commit -a -m "script3 feature update"
```

Okay, let's add a bit more functionality to our script. We can use the script to create new files but it would be nice if the script could also delete files. Your mission is to modify `script3` to add this deletion feature. You should also allow the user to pass an argument that returns the version of the script. Here are the requirements:

- Create a variable called `version` with a value of `1.0.1`
- The `-d` or `--delete` flags should execute the `delete_file` function.
- The `delete_file` function should remove a specified file when provided a name: `<name>-12345`
- If a filename isn't provided as an argument to the script, then use `server` as the default filename
- The `-v` or `--version` flags should output the version of the script (value of the `version` variable) using the `show_version` function..
- The usage statement should display the new deletion flag and version flag options.

- The help text should display useful information about the new deletion and version options.

You can test your script by issuing the following sequence of commands:

```
$ ./script3 -c
$ ./script3 -d
$ ./script3 -c foo
$ ./script3 -d foo
```

If your script is working properly, you should not see a `server-12345` or `foo-12345` file listed in the current directory. If your script isn't working properly, then keep working at it! It's okay (and expected) to make mistakes. Feel free to use Google or Slack to investigate any error messages that you encounter.

Once the script is working properly, commit the updated script to your current branch.

Next, checkout the master branch from the Git repository:

```
$ git checkout master
```

Take a look at your `script3` file again:

```
$ less script3
```

Whoa! What happened to all of our script changes? All of our hard work is gone! Well, no not really. We committed the changes to `script3` on a different branch. When we switched back to the master branch we basically stepped back in time. Git replaced the `script3` file with a version of the file before we created our new branch.

Branching is one of the most powerful features of Git and other version control systems. Branching allows you to experiment with ideas and code without destroying previous versions of files. You can quickly branch your code to test a new idea, and if it doesn't work you don't have to worry about breaking your original code.

Switch back to the feature branch:

```
$ git checkout feature/script3
```

Now, to calm you fears take a look at the script3 file:

```
$ less script3
```

All your modified code is back! The key concept to understand is that the files in your current working directory always represent a single point in time in the Git repository (a commit). Git updates the files in the current directory as you change branches. The lesson here is that you always need to understand **where** you are in the repository, because this directly effects which files and changes you see in the file directory.

Before we push our code up to GitHub, let’s merge our feature branch into the master branch. Think of this like merging our new feature into our production code.

```
$ git checkout master
$ git merge feature/script3
```

Look at the contents of the `script3` file. Your changes from the feature branch have now been merged into the master branch.

Check your work

Here is what the contents of your git repository should look like before final submission:

```
├ script1
├ script2
└ script3
```

Note, all of these script files should be executable.

Push project repo to GitHub

The final step of the assignment is to push your repository up to GitHub. We’ll follow a slightly different process than we followed during the previous assignment. Last time, we first created the Git repository on GitHub and then cloned (copied) the repository to our Linux server. Once we finished working with the repository on our local server, we pushed the changes back up to the GitHub repository.

That process works fine when you are working with a brand new project. But sometimes you want to take an existing project and existing Git repository and connect it to GitHub. That’s what we will do now.

First, click on the following link to create a new repository for this assignment:

https://classroom.github.com/a/XxSqdPS-

GitHub Classroom will provide you with a URL (https) to access the assignment repository. Either copy this address to your clipboard or write it down somewhere.

Example:
`https://github.com/UST-SEIS665/ hw3-seis665-02-spring2019-<Your GitHub ID>.git`

Next, you need to connect your local Git repository on the Linux server to your GitHub repository. The way you do that is by specifying your GitHub repository as the **origin**. A Git repository can be linked to several different Git repositories, called **remotes**. It’s not uncommon for a typical repository to have at least two or three different remotes. In this case, all we care about is the remote called origin. Setup the origin remote by typing:

```
$ git remote add origin <your GitHub repo URL>.git
```
Example:
```
$ git remote add origin https://github.com/UST-SEIS665/ hw3-seis665-02-
spring2019-<Your GitHub ID>.git
```

Now push your local repository to the GitHub repo by typing:

```
$ git push -u origin master
```

The `-u` flag in this command sets the *upstream* server for the repository to the remote defined as origin. Then, the master branch from the repository is pushed to the GitHub account.

Congratulations, your work has now been pushed up to GitHub! Take a look at your repository on GitHub and look specifically at the branches. You will see that only the master branch exists on the GitHub repository. That's because we specifically pushed up the master branch. If you type in `git branch` on your Linux server, you will notice that your local repository has two branches: `master` and `feature/script3`.

It's common for a local repository and upstream repository to have different branches. When you have a team of developers all working against the same GitHub repository, each team member may have various branches that exist on their local workstations but not on the central GitHub repository. Of course, it also makes sense to push these local branches to the GitHub repository at times as well — especially if multiple developers are collaborating on a particular feature and need to share changes.

Terminate server

The last step in the assignment is to terminate your Linux instance. AWS will bill you for every hour the instance is running. The cost is nominal, but there's no need to rack up unnecessary charges. Refer to the previous assignment if you don't remember how to terminate your EC2 instance.

Submitting your assignment

You should have emailed me your GitHub username during the previous assignment. There is no need to email your username again. I will review your published work on GitHub after the homework due date.