Assignment 2

Due Feb 19 by 5:45pm **Points** 100

SEIS 665 Assignment 2: Linux & Git

Overview

This week we will focus on becoming familiar with launching a Linux server and working with some basic Linux and Git commands. We will use AWS to launch and host the Linux server. AWS might seem a little confusing at this point. Don't worry, we will gain much more hands-on experience with AWS throughout the course. The goal is to get you comfortable working with the technology and not overwhelm you with all the details.

Requirements

You need to have a personal AWS account and GitHub account for this assignment. You should also read the ***Git Hands-on Guide*** and ***Linux Hands-on Guide*** before beginning this exercise.

A word about grading

One of the key DevOps practices we learn about in this class is the use of automation to increase the speed and repeatability of processes. Automation is utilized during the assignment grading process to review and assess your work.

It's important that you follow the instructions in each assignment and type in required files and resources with the proper names. All names are case sensitive, so a name like "Web1" is not the same as "web1". If you misspell a name, use the wrong case, or put a file in the wrong directory location you will lose points on your assignment. This is the easiest way to lose points, and also the most preventable. You should **always double-check your work** to make sure it accurately reflects the requirements specified in the assignment. You should always carefully review the content of your files before submitting your assignment.

The assignment

Let's get started!

Create GitHub repository

The first step in the assignment is to setup a Git repository on GitHub. We will use a special solution called GitHub Classroom for this course which automates the process of setting up student assignment repositories.

Here are the basic steps:

1. Click on the following link to open Assignment 2 on the GitHub Classroom site: **https://classroom.github.com/a/K4zcVmX- (https://classroom.github.com/a/K4zcVmX-)**
2. Click on the **Accept this assignment** button.
3. 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. You will need to use this address to set up the repository on a Linux server.

Example:

```
https://github.com/UST-SEIS665/hw2-seis665-02-spring2019-<your github id>.git
```

At this point your new repository to ready to use. The repository is currently empty. We will put some content in there soon!

Launch Linux server

The second step in the assignment is to launch a Linux server using AWS EC2. The server should have the following characteristics:

- Amazon Linux 2 AMI 64-bit (usually the first option listed)
- Located in a U.S. region (us-east-1)
- t2.micro instance type
- All default instance settings (storage, vpm, security group, etc.)

I've shown you how to launch EC2 instances in class. You can review it on Canvas.

Once you launch the new server, it may take a few minutes to provision.

Log into server

The next step is to log into the Linux server using a terminal program with a secure shell (SSH) support. You

can use **iTerm2** (http://www.iterm2.com/) on a Mac and GitBash/**PuTTY** (http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html) on a PC.

You will need to have the private server key and the public IP address before attempting to log into the server. The server key is basically your password. If you lose it, you will need to terminate the existing instance and launch a new server. I recommend reusing the same key when launching new servers throughout the class. Note, I make this recommendation to make the learning process easier and not because it is a common security practice.

I've shown you how to use a terminal application to log into the instance using a Windows desktop. Your personal computer or lab computer may be running a different OS version, but the process is still very similar. You can review the videos on the Canvas.

Working with Linux

If you've made it this far, congratulations! You've made it over the toughest hurdle. By the end of this course, I promise you will be able to launch and log into servers in your sleep.

You should be looking at a login screen that looks something like this:

```
Last login: Mon Mar 21 21:17:54 2016 from 174-20-199-194.mpls.qwest.net
```

```
 __|  __|_  )
 _|  (     /   Amazon Linux AMI
___|\___|___|
```

```
https://aws.amazon.com/amazon-linux-ami/2015.09-release-notes/
8 package(s) needed for security, out of 17 available
Run "sudo yum update" to apply all updates.
```

```
ec2-user@ip-172-31-15-26 ~]$
```

Your terminal cursor is sitting at the shell prompt, waiting for you to type in your first command. Remember the shell? It is a really cool program that lets you start other programs and manage services on the Linux system. The rest of this assignment will be spent working with the shell.

Note, when you are asked to type in a command in the steps below, don't type in the dollar-sign (`$`) character. This is just meant to represent the command prompt. The actual commands are represented by the characters to the right of the command prompt.

Let's start by asking the shell for some help. Type in:

```
$ help
```

The shell provides you with a list of commands you can run along with possible command options.

Next, check out one of the pages in the built-in manual:

```
$ man ls
```

A *man page* will appear with information on how to use the `ls` command. This command is used to list the contents of file directories. Either `space` through the contents of the man page or hit `q` to exit.

Most of the core Linux commands have man pages available. But honestly, some of these man pages are a bit hard to understand. Sometimes your best bet is to search on Google if you are trying to figure out how to use a specific command.

When you initially log into Linux, the system places you in your home directory. Each user on the system has a separate home directory. Let's see where your home directory is located:

```
$ pwd
```

The response should be `/home/ec2-user`. The `pwd` command is handy to remember if you ever forget what file directory you are currently located in. If you recall from the *Linux Hands-on Guide*, this directory is also your current working directory.

Type in:

```
$ cd /
```

The `cd` command let's you change to a new working directory on the server. In this case, we changed to the *root* (`/`) directory. This is the parent of all the other directories on the file system.

Type in:

```
$ ls
```

The `ls` command lists the contents of the current directory. As you can see, root directory contains many other directories. You will become familiar with these directories over time.

The `ls` command provides a very basic directory listing. You need to supply the command with some options if you want to see more detailed information.

Type in:

```
$ ls -la
```

See how this command provides you with much more detailed information about the files and directories? You can use this detailed listing to see the owner, group, and access control list settings for each file or directory. Do you see any files listed? Remember, the first character in the access control list column denotes whether a listed item is a file or a directory.

You probably see a couple files with names like `.autofsck`. How come you didn't see this file when you typed in the `ls` command without any options? (Try to run this command again to convince yourself.) Files names that start with a period are called hidden files. These files won't appear on normal directory listings.

Type in:

```
$ cd /var
```

Then, type in:

```
$ ls
```

You will see a directory listing for the `/var` directory. Next, type in:

```
$ ls ..
```

Huh. This directory listing looks the same as the earlier root directory listing. When you use two periods (`..`) in a directory path that means you are referring to the parent directory of the current directory. Just think of the two dots as meaning the directory *above* the current directory.

Now, type in:

```
$ cd ~
$ pwd
```

Whoa. We're back at our home directory again. The tilde character (`~`) is another one of those handy little directory path shortcuts. It always refers to our personal home directory. Keep in mind that since every user has their own home directory, the tilde shortcut will refer to a unique directory for each logged-in user.

Most students are used to navigating a file system by clicking a mouse in nested graphical folders. When they start using a command-line to navigate a file system, they sometimes get confused and lose track of their current position in the file system. Remember, you can always use the `pwd` command to quickly figure out what directory you are currently working in.

Let's make some changes to the file system. We can easily make our own directories on the file system.

Type:

```
mkdir test
```

Now type:

```
ls
```

Cool, there's our new `test` directory. Let's pretend we don't like that directory name and delete it. Type:

```
rmdir test
```

Now it's gone. How can you be sure? You should know how to check to see if the directory still exists at this point. Go ahead and check.

Let's create another directory. Type in:

```
$ mkdir documents
```

Next, change to the new directory:

```
$ cd documents
```

Did you notice that your command prompt displays the name of the current directory? Something like: `[ec2-user@ip-172-31-15-26 documents]$`. Pretty handy, huh?

Okay, let's create our first file in the `documents` directory. This is just an empty file for training purposes. Type in:

```
$ touch paper.txt
```

Check to see that the new file is in the directory. Now, go back to the previous directory. Remember the double dot shortcut?

```
$ cd ..
```

Okay, we don't like our `documents` directory any more. Let's blow it away. Type in:

```
$ rmdir documents
```

Uh oh. The shell didn't like that command because the directory isn't empty. Let's change back into the documents directory. But this time don't type in the full name of the directory. You can let shell auto-

completion do the typing for you. Type in the first couple characters of the directory name and then hit the tab key:

```
$ cd doc<tab>
```

You should use the `tab` auto-completion feature often. It saves typing and makes working with the Linux file system much much easier. Tab is your friend.

Now, remove the file by typing:

```
$ rm paper.txt
```

Did you try to use the `tab` key instead of typing in the whole file name? Check to make sure the file was deleted from the directory.

Next, create a new file:

```
$ touch file1
```

We like `file1` so much that we want to make a backup copy. Type:

```
$ cp file1 file1-backup
```

Check to make sure the new backup copy was created. We don't really like the name of that new file, so let's rename it. Type:

```
$ mv file1-backup backup
```

Moving a file to the same directory and giving it a new name is basically the same thing as renaming it. We could have moved it to a different directory if we wanted.

Let's list all of the files in the current directory that start with the letter `f`:

```
$ ls f*
```

Using wildcard pattern matching in file commands is really useful if you want the command to impact or filter a group of files. Now, go up one directory to the parent directory (remember the double dot shortcut?)

We tried to remove the documents directory earlier when it had files in it. Obviously that won't work again. However, we can use a more powerful command to destroy the directory and vanquish its contents. Behold, the all powerful remove command:

```
$ rm -fr documents
```

Did you remember to use auto-completion when typing in `documents`? This command and set of options forcibly removes the directory and its contents. It's a dangerous command wielded by the mightiest Linux wizards. Okay, maybe that's a bit of an exaggeration. Just be careful with it.

Check to make sure the `documents` directory is gone before proceeding.

Let's continue. Change to the directory `/var` and make a directory called `test`.

Ugh. Permission denied. We created this darn Linux server and we paid for it. Shouldn't we be able to do anything we want on it? You logged into the system as a user called `ec2-user`. While this user can create and manage files in its home directory, it cannot change files all across the system. At least it can't as a normal user. The `ec2-user` is a member of the *root* group, so it can escalate its privileges to *super-user* status when necessary. Let's try it:

```
$ sudo mkdir test
```

Check to make sure the directory exists now. Using `sudo` we can execute commands as a super-user. We can do anything we want now that we know this powerful new command.

Go ahead and delete the `test` directory. Did you remember to use `sudo` before the `rmdir` command? Check to make sure the directory is gone.

You might be asking yourself the question: why can we list the contents of the `/var` directory but not make changes? That's because all users have read access to the `/var` directory and the `ls` command is a read function. Only the *root* users or those acting as a super-user can write changes to the directory.

Let's go back to our home directory:

```
$ cd ~
```

Editing text files is a really common task on Linux systems because many of the application configuration files are text files. We can create a text file by using a text editor. Type in:

```
$ nano myfile.conf
```

The shell starts up the `nano` text editor and places your terminal cursor in the editing screen. Nano is a simple text-based word processor. Type in a few lines of text. When you're done writing your novel, hit `ctrl-x` and answer `y` to the prompt to save your work. Finally, hit `enter` to save the text to the filename you specified.

Check to see that your file was saved in the directory. You can take a look at the contents of your file by typing:

```
$ cat myfile.conf
```

The `cat` command displays your text file content on the terminal screen. This command works fine for displaying small text files. But if your file is hundreds of lines long, the content will scroll down your terminal screen so fast that you won't be able to easily read it. There's a better way to view larger text files. Type in:

```
$ less myfile.conf
```

The `less` command will page the display of a text file, allowing you to page through the contents of the file using the space bar. Your text file is probably too short to see the paging in action though. Hit `q` to quit out of the `less` text viewer.

Hit the up-arrow key on your keyboard a few times until the commmand `nano myfile.conf` appears next to your command prompt. Cool, huh? The up-arrow key allows you to replay a previously run command. Linux maintains a list of all the commands you have run since you logged into the server. This is called the command history. It's a really useful feature if you have to re-run a complex command again.

Now, hit `ctrl-c`. This cancels whatever command is displayed on the command line.

Type in the following command to create a couple empty files in the directory:

```
$ touch file1 file2 file3
```

Confirm that the files were created. Some commands, like `touch`. allow you to specify multiple files as arguments. You will find that Linux commands have all kinds of ways to make tasks more efficient like this.

Throughout this assignment, we have been running commands and viewing results on the terminal screen. The screen is the standard place for commands to output results. It's known as the standard out (*stdout*). However, it's really useful to output results to the file system sometimes. Type in:

```
$ ls > listing.txt
```

Take a look at the directory listing now. You just created a new file. View the contents of the `listing.txt` file. What do you see? Instead of sending the output from the `ls` command to the screen we sent it to a text file.

Let's try another one. Type:

```
$ cat myfile.conf > listing.txt
```

Take a look at the contents of the `listing.txt` file again. It looks like your `myfile.conf` file now. It's like you made a copy of it. But what happened to the previous content in the `listing.txt` file? When you redirect the output of a command using the right angle-bracket character (`>`), the output overwrites the existing file. Type this command in:

```
$ cat myfile.conf >> listing.txt
```

Now look at the contents of the `listing.txt` file. You should see your original content displayed twice. When you use two angle-bracket characters in the commmand the output appends (or adds to) the file instead of overwriting it.

We redirected the output from a command to a text file. It's also possible to redirect the input to a command. Typically we use a keyboard to provide input, but sometimes it makes more sense to input a file to a command. For example, how many words are in your new `listing.txt` file? Let's find out. Type in:

```
$ wc -w < listing.txt
```

Did you get a number? This command inputs the `listing.txt` file into a word count program called `wc`.

Type in the command:

```
$ ls /usr/bin
```

The terminal screen probably scrolled quickly as filenames flashed by. The `/usr/bin` directory holds quite a few files. It would be nice if we could page through the contents of this directory. Well, we can. We can use a special shell feature called *pipes*. In previous steps, we redirected I/O using the file system. Pipes allow us to redirect I/O between programs. We can redirect the output from one program into another. Type in:

```
$ ls /usr/bin | less
```

Now the directory listing is paged. Hit the `spacebar` to page through the listing. The pipe, represented by a vertical bar character (`|`), takes the output from the `ls` command and redirects it to the `less` command where the resulting output is paged. Pipes are super powerful and used all the time by savvy Linux operators.

Hit the `q` key to quit the paginated directory listing command.

Working with shell scripts

Now things are going to get interesting.

We've been manually typing in commands throughout this exercise. If we were running a set of repetitive tasks, we would want to automate the process as much as possible. The shell makes it really easy to automate tasks using shell scripts. The shell provides many of the same features as a basic procedural programming language. Let's write some code.

Type in this command:

```
$ j=123
$ echo $j
```

We just created a variable named `j` referencing the string `123`. The `echo` command printed out the value of the variable. We had to use a dollar sign (`$`) when referencing the variable in another command.

Next, type in:

```
$ j=1+1
$ echo $j
```

Is that what you expected? The shell just interprets the variable value as a string. It's not going to do any sort of computation.

Typing in shell script commands on the command line is sort of pointless. We want to be able to create scripts that we can run over-and-over. Let's create our first shell script.

Use the `nano` editor to create a file named `myscript`. When the file is open in the editor, type in the following lines of code:

```
#!/bin/bash
echo Hello $1
```

Now quit the editor and save your file. We can run our script by typing:

```
$ ./myscript World
```

Er, what happened? Permission denied. Didn't we create this file? Why can't we run it? We can't run the script file because we haven't set the execute permission on the file. Type in:

```
$ chmod u+x myscript
```

This modifies the file access control list to allow the owner of the file to execute it. Let's try to run the command again. Hit the up-arrow key a couple times until the `./myscript World` command is displayed and hit `enter`.

Hooray! Our first shell script. It's probably a bit underwhelming. No problem, we'll make it a little more complex. The script took a single argument called `World`. Any arguments provided to a shell script are represented as consecutively numbered variables inside the script (`$1`, `$2`, etc). Pretty simple.

You might be wondering why we had to type the `./` characters before the name of our script file. Try to type in the command without them:

```
$ myscript World
```

Command not found. That seems a little weird. Aren't we currently in the directory where the shell script is located? Well, that's just not how the shell works. When you enter a command into the shell, it looks for the command in a predefined set of directories on the server called your *PATH*. Since your script file isn't in your special path, the shell reports it as not found. By typing in the `./` characters before the command name you are basically forcing the shell to look for your script in the current directory instead of the default path.

Create another file called `cleanup` using `nano`. In the file editor window type:

```
#!/bin/bash
# My cleanup script
mkdir archive
mv file* archive
```

Exit the editor window and save the file. Change the permissions on the script file so that you can execute it. Now run the command:

```
$ ./cleanup
```

Take a look at the file directory listing. Notice the `archive` directory? List the contents of that directory. The script automatically created a new directory and moved three files into it. Anything you can do manually at a command prompt can be automated using a shell script.

Let's create one more shell script. Use `nano` to create a script called `namelist`. Here is the content of the script:

```
#!/bin/bash
# for-loop test script
names='Jason John Jane'
for i in $names
do
  echo Hello $i
done
```

Change the permissions on the script file so that you can execute it. Run the command:

```
$ ./namelist
```

The script will loop through a set of names stored in a variable displaying each one. Scripts support several programming constructs like for-loops, do-while loops, and if-then-else. These building blocks allow you to create fairly complex scripts for automating tasks.

Installing packages and services

We're nearing the end of this assignment. But before we finish, let's install some new software packages on our server. The first thing we should do is make sure all the current packages installed on our Linux server are up-to-date. Type in:

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

This is one of those really powerful commands that requires `sudo` access. The system will review the currently installed packages and go out to the Internet and download appropriate updates.

Next, let's install an Apache web server on our system. Type in:

```
$ sudo yum install httpd -y
```

Bam! You probably never knew that installing a web server was so easy. We're not going to actually use the web server in this exercise, but we will in future assignments.

We installed the web server, but is it actually running? Let's check. Type in:

```
$ sudo service httpd status
```

Nope. Let's start it. Type:

```
$ sudo service httpd start
```

We can use the `service` command to control the services running on the system. Let's setup the service so that it automatically starts when the system boots up. Type in:

```
$ sudo chkconfig httpd on
```

Cool. We installed the Apache web server on our system, but what other programs are currently running? We can use the `ps` command to find out. Type in:

```
$ ps -ax
```

Lots of processes are running on our system. We can even look at the overall performance of our system using the `top` command. Let's try that now. Type in:

```
$ top
```

The display might seem a little overwhelming at first. You should see lots of performance information displayed including the cpu usage, free memory, and a list of running tasks.

We're almost across the finish line. Let's make sure all of our valuable work is stored in a git repository. First, we need to install git. Type in the command:

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

Check your work

It's very important to check your work before submitting it for grading. A misspelled, misplaced or missing file will cost you points. This may seem harsh, but the reality is that these sorts of mistakes have consequences in the real world. For example, a server instance could fail to launch properly and impact customers because a single required file is missing.

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

```
├archive
|   ├ file1
|   ├ file2
|   └ file3
├ namelist
└ myfile.conf
```

Saving our work in the git repository

Next, make sure you are still in your home directory (`/home/ec2-user`). We will install the git repository you created at the beginning of this exercise. You will need to modify this command by typing in the GitHub repository URL you copied earlier.

```
$ git clone <your GitHub URL here>.git
```

Example:

```
git clone https://github.com/UST-SEIS665/hw2-seis665-02-spring2019-<your github id>.git
```

The git application will ask you for your GitHub username and password. Note, if you have multi-factor authentication enabled on your GitHub account you will need to provide a personal token instead of your password.

Git will clone (copy) the repository from GitHub to your Linux server. Since the repository is empty the clone happens almost instantly. Check to make sure that a sub-directory called "hw2-seis665-02-spring2019-<username>" exists in the current directory (where <username> is your GitHub account name). Git automatically created this directory as part of the cloning process.

Change to the `hw2-seis665-02-spring2019-<username>` directory and type:

```
$ ls -la
```

Notice the `.git` hidden directory? This is where git actually stores all of the file changes in your repository. Nothing is actually in your repository yet.

Change back to the parent directory (`cd ..`). Next, let's move some of our files into the repository. Type:

```
$ mv archive hw2-seis665-02-spring2019-<username>
$ mv namelist hw2-seis665-02-spring2019-<username>
$ mv myfile.conf hw2-seis665-02-spring2019-<username>
```

Hopefully, you remembered to use the auto-complete function to reduce some of that typing. Change to the `hw2-seis665-02-spring2019-<username>` directory and list the directory contents. Your files are in the working directory, but are not actually stored in the repository because they haven't been committed yet.

Type in:

```
$ git status
```

You should see a list of untracked files. Let's tell git that we want these files tracked. Type in:

```
$ git add *
```

Now type in the `git status` command again. Notice how all the files are now being tracked and are ready to be committed. These files are in the git staging area. We'll commit them to the repository next. Type:

```
$ git commit -m 'assignment 2 files'
```

Next, take a look at the commit log. Type:

```
$ git log
```

You should see your commit listed along with an assigned hash (long string of random-looking characters).

Finally, let's save the repository to our GitHub account. Type in:

```
$ git push origin master
```

The git client will ask you for your GitHub username and password before pushing the repository.

Go back to the GitHub.com website and login if you have been logged out. Click on the repository link for the assignment. Do you see your files listed there? Congratulations, you completed the exercise!

Terminate server

The last step 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.

Here are the steps to terminate your instance:

1. Log into your AWS account and click on the EC2 dashboard.
2. Click the `Instances` menu item.
3. Select your server in the instances table.
4. Click on the `Actions` drop down menu above the instances table.
5. Select the `Instance State` menu option
6. Click on the `Terminate` action.

Your Linux instance will shutdown and disappear in a few minutes. The EC2 dashboard will continue to display the instance on your instance listing for another day or so. However, the state of the instance will be `terminated`.

Submitting your assignment — IMPORTANT!

If you haven't already, please e-mail me your GitHub username in order to receive credit for this assignment. There is no need to email me to tell me that you have committed your work to GitHub or to ask me if your GitHub submission worked. If you can see your work in your GitHub repository, I can see your work.