

Nathan Tomlin, CSC2510

Ansible Automation Documentation:

SECTION 1: INTRODUCTION

In this specific part of the ansible automation project, we will be discussing the full documentation surrounding the creation of our project. This document exists as a guide to teach those who may be interested in creating something like we have created here. Regardless of what your goal is, this document serves as an all-in-one base template for getting your ansible servers up and running, as well as configuring them to automate processes and tasks on remote servers by using a combination of playbooks, shell scripts, and most importantly logic.

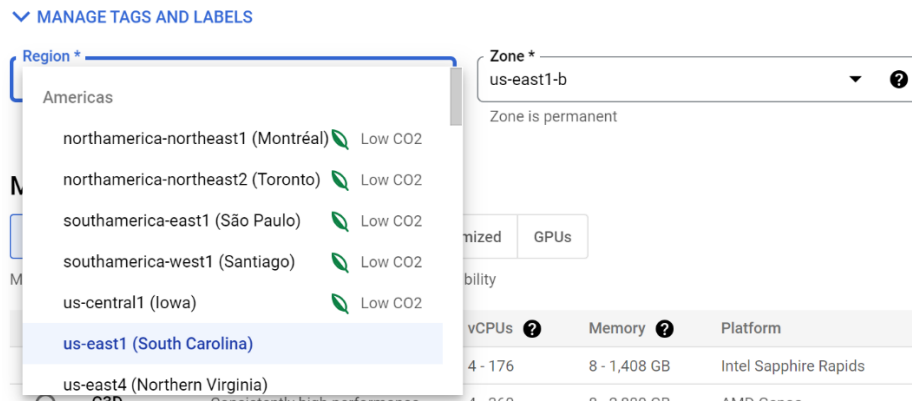
To start, let's begin by discussing the overarching goal of the project. The outcome is simple; we needed to create 3 ansible management servers (Dev, Test, and Prod) to control a combination of 6 web servers (3 of which also function as database servers). More specifically, each ansible server will be responsible for managing 2 servers, one of which is a web server, and the other being a web server that also functions as a database server. Each of these ansible servers are to execute their own playbook which sets up the server environment for the two servers they are responsible for. This includes installing node.js and apache2 on the web servers (one of which is also a database server), and installing MariaDB on only the web / database server. In addition to this, we must also execute a shell script from each of the specific ansible playbooks which clones/pulls from a remote repository onto the remote server. It is also important that the cloned repository is set up with the branch that corresponds to each specific server environment. Finally, we must automate the playbooks we created to run on a fixed schedule, which is once every minute.

When looking at this project as a whole, it seems intimidating and overwhelming, and that's because it is. The catch is that it is only overwhelming if you perceive it as one task to accomplish. Looking at it this way will destroy you. Instead, we need to break down the project into simple subtasks, which we can complete one by one and piece together at the end.

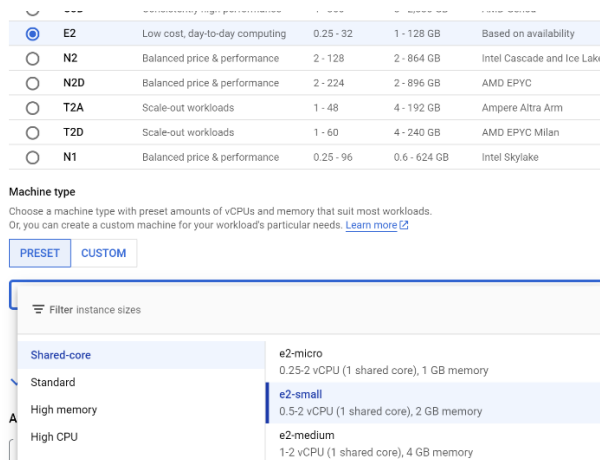
SECTION 2: CREATING THE SERVERS

For this specific project, we are going to be using the google cloud platform (GCP) to create and run each of our servers. To start, we create and configure the ansible management servers. Each ansible management server should be named as such: **environment**ansible, where environment is the name of the specific environment the ansible server will be automating

(either Dev, Test, or Prod). Next, we will choose the server to host our ansible VM, which in this case will be us-east1 (South Carolina) for the region, and us-east1-b for the zone. (Figure below for reference)



Next, we will select our machine configuration as E2, since this is perfect for our needs. Now we will select the machine type as E2 Small. (Figure below for reference)



Finally, we must change the boot disk from Debian to CentOS7, which can be found in the boot disk menu towards the bottom of the page. (Figure on next page for reference)

Confidential Computing is disabled

Container ⓘ

Deploy a container image to this VM

Operating system

CentOS

Version *

CentOS 7

x86_64, x86_64 built on 20231115

Boot disk type *

Standard persistent disk

COMPARE DISK TYPES

Size (GB) *

20

Provision between 20 and 3072 GB

SHOW ADVANCED CONFIGURATION

SELECT **CANCEL**

Boot disk ⓘ

Name

Type

Size

License type ⓘ

Image

CHANGE

Finally, we can click create at the bottom to finish creating our server. We will repeat this process two more times for the remaining two ansible servers, using the exact same setup parameters as we did for this one (with the only difference being the name).

Once we have created ansible servers for Dev, Test, and Prod, we can begin to create the two remaining servers for each environment. The setup for these servers will be similar to the ansible ones, however we will change a couple of settings. Each of these servers should be named as either **enviornmentdb01** OR **enviornmentws01**, where **environment** is the specific ansible sever you are creating (either dev, test, or prod), followed by ws01/db01 depending on whether you are creating the webserver or database server for that environment (for our purposes, the webserver and the database server will function exactly the same, with the only difference being the naming convention and the type of packages we will install - more on that later). Next, we will choose us-central1 (Iowa) for the region, and us-central1-c for the zone. We will keep the same machine type (E2 Small) and instead of changing the boot disk to CentOS7, we will leave it as Debian GNU/Linux 11 (bullseye). Once the servers have been created, they will be automatically started. You can connect to a server by clicking the drop-down arrow next to SSH and selecting open from browser window.

SECTION 3: SETTING UP ANSIBLE SERVERS

Now that we have created all of our servers, it is time to get them set up. We will start by configuring our ansible servers so that we can get it out of the way. The most efficient way to complete this section is to follow all of the steps listed below in their entirety on one singular ansible server, and then come back and repeat every step for the remaining two ansible servers. There are multiple different steps in this section, so pay close attention.

1. Installing Nano & Ansible

Before we do any sort of playbook making or scripting, it is important that we configure each ansible server so that it will work as expected. First, we need to go on each ansible server (the one for dev, the one for prod, & the one for test) and install ansible and nano.

To do so, we will be using two commands. The commands are as follows:

for nano: `$ sudo yum install nano`

for ansible: `$ sudo yum install ansible`

After running each of these two commands, you will be prompted to download them. Type y and then hit enter in order to confirm the download. It is important to not touch your machine during the installation process, as doing so may corrupt the file being installed.

2. Editing the hosts File

Now that we have installed both nano and ansible, our time working on these servers will become much easier. Next we need to configure the host files for each ansible server. We navigate to the host files by using the following commands:

`$ cd /etc/ansible`

`$ sudo nano hosts`

Here we will create our server groups. We also want to add our user and password variables for each server, so that we can log in to our other non-ansible servers as root (more on this later). It is important to remember that we have two webserver, but only one webserver is a database server. We need to group our hosts as follows (as well as variables):

```
GNU nano 2.3.1 File: hosts

# Here's another example of host ranges, this time there are no
# leading 0s:

## db-[99:101]-node.example.com

#Grouped Test Web Hosts
[testwebservers]
10.128.0.9
10.128.0.10

#Grouped Test DB Hosts
[testdbservers]
10.128.0.10

[testwebservers:vars]
ansible_user=root
[ansible_password=Mickey2023!

[testdbservers:vars]
ansible_user=root
ansible_password=Mickey2023!
```

Here, we group the hosts by using `[environmentwebservers]` & `[environmentdbservers]`, where **environment** is the name of the group of servers for that specific ansible server (either dev, test or prod). Next, we place the internal IP address of the remote servers into their targeted group. In this case we have testdb01 and testws01 both inside of the `[testwebservers]` group, and that's because while db01 is technically a database server, it is also a web server. Then, we have db01 in its own group `[testdbservers]`. The reason for this is because we want to install node.js and apache2 on all webservers, but we only want to install MariaDB on the database server. Setting up the hosts this way will make it easier on us when creating our playbooks.

Below the grouped hosts we see the variables assigned to each host group. These variables are essential as they allow us to remote into the target servers later on. Configure them in this way: `[environmentwebservers:vars]` & `[environmentdbservers:vars]`, where **environment** is the name of the group of servers for that specific ansible server (either dev, test or prod). Then add in the variables as shown above. It is also **ESSENTIAL** that all of the entries in the host file are:

1. Set up so that they have only **ONE** white space to the left of the opening bracket [
2. Set up so that there is only **ONE** space underneath each group of host and host vars.
3. Set up so that the IP addresses inside of the host group and the variables inside of the vars group are **DIRECTLY** under the opening brackets [
4. Save the changes you made to the hosts file by pressing ctrl + x, followed by y to save changes, and then enter to save.

3. Editing the ansible.cfg File

Now that we have grouped our hosts and created our host variables, it is time to make our way over to the ansible.cfg file, where we will be uncommenting the following line:

```
# host_key_checking=False.
```

The line above can be found towards the middle of the `ansible.cfg` file, which we can access by the following command (assuming we are still inside `/etc/ansible/` like in the previous step):

```
$ sudo nano ansible.cfg
```

Initially the line will be preceded by a comment, however we want to remove that comment so that the end result looks like this:

```
# This variable is set to True by default for backwards compatibility. It
# will be changed to a default of 'False' in a future release.
# ansible_facts.
# inject_facts_as_vars = True

# additional paths to search for roles in, colon separated
#roles_path = /etc/ansible/roles

# uncomment this to disable SSH key host checking
host_key_checking = False

# change the default callback, you can only have one 'stdout' type enabled at a time.
#stdout_callback = skippy

## Ansible ships with some plugins that require whitelisting,
## this is done to avoid running all of a type by default.
## These setting lists those that you want enabled for your system.
## Custom plugins should not need this unless plugin author specifies it.
```

Save the changes (same commands as previous step) and then exit out of the file.

4. Installing Git

As a precaution, we will go ahead and install git on the ansible server, using the following command:

```
$ sudo yum install git
```

When prompted to download, type `y` and then hit enter.

5. Repeat All 4 Steps On the Remaining 2 Ansible Servers – Now that we have finished with all of the steps in this section, go ahead and repeat all of the steps listed on the other 2 ansible servers. When finished, **RESTART ALL ANSIBLE VM's**.

SECTION 4: SETTING UP THE REMAINING SERVERS

After setting up all of the ansible servers, it is time to configure the remaining 6 servers. Unfortunately, this section will have us go just as in depth as we did in the previous one, mainly

because there are still some steps we must take to ensure that the remaining 6 non-ansible servers all get along with their respective ansible management server. Once again, the most efficient way to complete this section is to follow all of the steps listed below in their entirety on one singular non-ansible server, and then come back and repeat every step for the remaining 5 non-ansible servers.

1. Installing Git

The first step we need to take in this section is to install git on our non-ansible server. This will allow us to access git source control and commands like clone and pull later on when we run our shell script. Git can be installed by the following command on the target servers (**Running Debian**).

```
$ sudo apt install git
```

When prompted to download, type y and then hit enter.

2. Changing Log-In Password

In order for our ansible servers to remote into our non-ansible servers, we must ensure that our password matches the password we set in the host file of our ansible machine. To do this use the following command:

```
$ sudo passwd
```

You will be prompted to enter a password. Keep in mind that your password will not appear on the screen (that does not mean the command is frozen). Type your password as you normally would, and then hit enter to save, making sure that it matches the password you set in the ansible server. It will then prompt you to re-enter the password. Once done, hit enter again.

3. – Editing the ssh config File

After setting up the password for our server, we now want to head into the ssh_config file to make some changes. Head in to the config file using the command:

```
$ sudo nano /etc/ssh/sshd_config
```

Once inside, we are looking for two specific lines. The first is **PermitRootLogin no** and the second is **PasswordAuthentication no**. We want to find each of the two lines and change the values from no to **yes**. Once finished, the file should look like this:

```
#LoginGraceTime 2m
PermitRootLogin yes
#StrictModes yes
#MaxAuthTries 6
#MaxSessions 10

#PubkeyAuthentication yes

# Expect .ssh/authorized_keys2 to be disregarded by default in future.
#AuthorizedKeysFile .ssh/authorized_keys .ssh/authorized_keys2

#AuthorizedPrincipalsFile none

#AuthorizedKeysCommand none
#AuthorizedKeysCommandUser nobody

# For this to work you will also need host keys in /etc/ssh/ssh_known_hosts
#HostbasedAuthentication no
# Change to yes if you don't trust ~/.ssh/known_hosts for
# HostbasedAuthentication
#IgnoreUserKnownHosts no
# Don't read the user's ~/.rhosts and ~/.shosts files
#IgnoreRhosts yes

# To disable tunneled clear text passwords, change to no here!
PasswordAuthentication yes
#PermitEmptyPasswords no
```

Once this task is complete, is essential to restart **ALL non-ansible VM's**.

5. Generating SSH Keys

The final step in this section is going to cover how to generate SSH keys for all of the non-ansible servers, as this is essential to us being able to access our remote repositories later. The first step in this process is to generate an SSH key by using the following command inside of your **home directory** (this makes the ssh filepath easy to locate later):

```
$ ssh-keygen -t rsa -b 4096 -f .ssh/name -C username@tntech.edu
```

Where **username** is the username of your google cloud platform account and name is whatever you decide to name your SSH key.

For this project I have chosen to name all my SSH keys the same for the sake of simplicity.

Once you hit enter, you will be prompted to set a password for your SSH keys. For this project I chose to leave mine blank, but in general you should always set a strong password for this. After you enter your password, you should see something like this:


```
ntomlin42@testws01:~$ ssh-keygen -t rsa -b 4096 -f .ssh/gcpgithub -C ntomlin42@tntech.edu
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in .ssh/gcpgithub
Your public key has been saved in .ssh/gcpgithub.pub
The key fingerprint is:
SHA256:x184XG5ZeRXL+pBFY7Rm+fNcUExCzeTe2cNDYMNZkIs ntomlin42@tntech.edu
The key's randomart image is:
+---[RSA 4096]-----+
|
| . * O & * |
| . + * = X |
| o = X o |
| . . E + O + = |
| S + = + * * |
| . . + o . * |
| . . . o |
|
+-----[SHA256]-----+
ntomlin42@testws01:~$
```

6. Adding SSH Keys to GitHub

Now we need to add the SSH key we just generated to our GitHub account. To do this, start by printing your SSH key by using the following command (do not share this key with **ANYONE**):

```
$ cat .ssh/name.pub
```

Where **name** is the name of your SSH key. Once entered, you should see something similar to this:

```
The key fingerprint is:
SHA256:n7q25WGB00xCg0bYACnyGQvfFSBnf8lQIHVu7G7409I ntomlin42@tntech.edu
The key's randomart image is:
+---[RSA 4096]-----+
|.oo+*o=+o|
|= o=+. * .|
|o+ =o.+ B|
| =... = .|
| . S .|
| * o o|
| . * o *|
| o+*B.|
| . = .|
+-----[SHA256]-----+
ntomlin42@devws01:~$ cat .ssh/gcpgithub.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDAwltNYiS/jlnk2RlqDrMO8BNXeKxtwOhV8BCsQ4qNFHGHY7EYMPZjI+OdIXjOuX9Geiy88c7g
GAYtCewUTrZvq5mgAgB6C3oDWUxUevKr3h5Yw4R7Nkx+l+18eCO2khmvghTnSlb4Nnyr43hkq6n/olj+payYG+o60krAnUhBG1PGrLJ/gfwvGZWB
EnFY/Hds4QEx5R4H16u6o+TWiH5AFFnsCXi3NCPyWiWHgmVgb+0yJcYvWwqAZFABFqzqXOQPtcbbqNrwCSz91bUoX/drWpjnc061QPKCSLhhQik/
Y6v3UkwJDB1wOUWdFz7K/2P6+J5PGLq+ixI/3UbyinyMImogwPzjK5YiQM5tXK80jNLGKssQvXtuhRCWHUwSSB4AlU6oRSJrxp098J9qc3H+5MG1
Cq1s4FONjPspihdCVep4ONsV68JkpZePb18b6M9TPpJOn3gn2WzUMJjck7bmHRg5AhsEAtHgRQAz7/HY7ttsHAn5ZuW7O+92VRCo/MMspacEU+k7
2QXQUUzV1WcAlGfTRnNi2gmafiOVv1qvsIg7Rsey5V8AOprAygCsEeslXqFC0PEGPeilBGucleEcBKFxaKygn1T6ErtMWHtMIduUKZehuYlY6OXB
TKK4eMstItOT06tv7lVduG9lQU7pNNLUGYEKmhHE2L0buIffVQ== ntomlin42@tntech.edu
```

Copy everything from ssh-rsa all the way to your email address. Now we need to take this over to GitHub and add it to our account's SSH keys. To do this, sign into github and navigate to the profile icon in the top right. Once there, choose settings from the drop-down menu, and then navigate to the SSH keys tab.

Add new SSH Key

Title

Key type

Authentication Key

Key

Begins with 'ssh-rsa', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', 'ecdsa-sha2-nistp521', 'ssh-ed25519', 'sk-ecdsa-sha2-nistp256@openssh.com', or 'sk-ssh-ed25519@openssh.com'

Add SSH key

From here, name the title of the key the same way you named your server during setup. Then, paste the SSH key we copied into the Key box. Hit add SSH key and it will be saved to your profile.

5. Repeat All 4 Steps on the Remaining 5 Non-Ansible Servers – Now that we have finished with all of the steps in this section, go ahead and repeat all of the steps listed on the other 5 non-ansible servers.

SECTION 6: CLONING & PULLING REMOTE REPOSITORY FROM SHELL SCRIPT

After getting all of the servers set up and ready to go, we can now begin the process of creating our playbooks. The first part in creating our playbook is to go ahead and create the shell script that our playbook will be calling to create the repository and continually pull from it on both our web and database servers. To get started, we will create a new directory in the home directory of our ansible servers called `ansible_playbooks`. We will create all of our necessary files inside of this directory for simplicity. The shell script should be named `cloneorrepo.sh`, because that's exactly what it does. Take a look at the next page.

```
#!/usr/bin/env bash

branch=$1

time_stamp=$(date '+%Y-%m-%d %H:%M:%S')

# Check if the provided environment is valid
case "$branch" in
    "Dev" | "Prod" | "Test")
        # Log the chosen environment to webrepolog.txt
        ;;
    *)
        # Exit with an error message
        echo "Error: Invalid environment. Please choose Dev, Prod, or Test." >>/home/ntomlin42/brancherror.txt
        exit 0
        ;;
esac

repo_url="git@github.com:ttu-bburchfield/swollenhippofinal.git"
target_dir="/home/ntomlin42/swollenhippoweb"

eval "$(ssh-agent -s)"
ssh-add /home/ntomlin42/.ssh/gcpgithub

# Check if the target directory exists
if [ -d "$target_dir" ]; then
    # Directory exists, pull the latest changes
    echo "Updating existing repository..."
    cd "$target_dir"
    git pull origin main
    echo "Successfully pulled repository on $time_stamp. Chosen branch: $branch" >> webrepolog.txt
else
    # Directory doesn't exist, create it and clone the repository
    echo "Cloning repository for the first time..."
    mkdir "$target_dir"
    cd "$target_dir"
    ssh-keyscan -H github.com >> ~/.ssh/known_hosts
    git clone "$repo_url" "$target_dir"
    git checkout -b "$branch"
    echo "Cloned successfully on $time_stamp! Chosen branch: $branch" >> webrepolog.txt
fi

echo "Update complete."
```

Inside of the shell script we have multiple moving parts that piece together to give us the ability to either clone a repository or pull a repository. The way it works is simple. It takes in a parameter (that will be input from running the playbook later) for either the Dev, Test, or Prod environments (assigned to the variable named `$branch`). The case statement is going to check that input to see if it matches exactly. If it does not, then we will create a log file in the target server with a message to let the user know that their input did not match any of the available environments, and then the script will stop running. However, if the input does match one of the available environments, then we move on to the next part of the script.

The next part of the script defines our target repository by setting it equal to the variable `repo_url`. It also defines our target directory where we want to store our local copy of the repository as `target_dir`. The `eval command` as well as the `ssh-add` are necessary to ensure that we will be able to use SSH to clone the repositories. You can see that in our `ssh-add` command, we specified the entire filepath to our SSH key that we created earlier. This is necessary, or else the script will not run as intended.

The final part of the script is where the magic happens. We use an if statement to check and see if our target directory exists, because if it does not then that is how we know if the repository has been cloned previously. If the repository has been cloned previously, then script will instead pull from the repository by using the following commands:

`cd "$target_dir"` - switch to the directory where the local repository is located at.

`git pull origin main` – pull the latest changes from the remote repository.

The script then adds a line of text to the log file named `webrepo.txt` to let the user know the repo was pulled successfully, and also includes the date and time of when the repo was pulled, as well as the environment the user is in. However, if the repository has not been cloned previously, then the script will automatically know because the directory of the local repository directory does not exist. This **ensures** that the script will only clone the repository **one time**, which is necessary for our script to function correctly. When the repository is cloned for the first time, the script utilizes the following commands:

`mkdir "$target_dir"` – creates the target directory to store the cloned repository.

`cd "$target_dir"` – changes directory to the newly created target directory.

`ssh-keyscan -H github.com >> ~/.ssh/known_hosts` – adds GitHub to lists of known hosts.

`git clone "$repo_url" "$target_dir"` – clones the repository URL to the target directory.

`git checkout -b "$branch"` – creates and sets the branch to match the environment.

The script will then add a line of text to the `webrepo.txt` file letting the user know the time and date of when the repository was cloned, as well as which environment they are in.

Once finished with the shell script, go ahead and repeat this section in the other two ansible VM's.

SECTION 6: THE ANSIBLE PLAYBOOK

Now that we have completed the shell script that we need to run in our playbook, we can go ahead and create our ansible playbook. The first part of creating the playbook is to create it inside of the `ansible_playbook` directory which we made in the last section. Once inside, we want to create the following file: `environmentsetup.yml`, where **environment** is the name of our ansible server (dev, test, or prod). Once inside the playbook, edit the playbook include the following:

```

--
- name: installapacheandnode
  hosts: devwebservers
  tasks:
    - name: ensure apache is installed
      apt:
        name: apache2
        state: latest

    - name: ensure nodejs is installed
      apt:
        name: nodejs
        state: latest

- name: installmariadb
  hosts: devdbservers
  tasks:
    - name: ensure mariadb is installed
      apt:
        name: mariadb-server
        state: latest

- name: cloneRepoOrPullRepo
  hosts: devwebservers
  tasks:
    - name: Copy the shell script to the server
      copy:
        src: /home/ntomlin42/ansible_playbooks/cloneorrepo.sh
        dest: /home/ntomlin42
        mode: '0755' # Ensure the script is executable

    - name: Execute the shell script
      command: bash /home/ntomlin42/cloneorrepo.sh "{{serverselect}}"
      become: true
      become_user: ntomlin42

```

Here we are defining our tasks, as well as what they do and who we will apply them to. You can see that we have a task to install apache2 and nodeJS on our web servers for the dev environment, as well as a task to install MariaDB on our database server. Notice that we have also used the `state: latest` command, which will install the package if it is not installed, and then continue to update the packages to the latest version upon each consecutive call of the playbook. This eliminates the need for another ansible playbook to keep the packages up to date and increases the efficiency of our ansible management server. We also have a task to run our shell script that we created in the earlier section (two specifically).

The first part of our task copies the script that we created to pull or clone the repository onto the servers which we listed in the host tab (in this case it will copy the script to both the devws01 & the devdb01 server). It then defines the file path to where the script should be copied to, and it also gives the target host permission to execute the script by using the command `mode: '0755'`.

The second part of the script deals with executing the script that was just copied on to the remote server. It first defines the filepath to follow in order to execute the script we just copied, and also takes in a parameter to pass into to the script upon execution (`serverselect`). This is where dev, prod, or test would be entered (more on this later, this is handled in the cronjob section). It then uses the `become_user: ntomlin42` command to run the script as the

normal user of the remote server (though the username should be replaced for the one you set up your servers with).

When finished with the playbook, copy and paste it to the other two remaining ansible servers, making sure to change both the **file name** as well as the names of the **target hosts** to match that of the ansible server you are running it on. **Note that you can test the script now to see if it works by calling the playbook directly from the terminal. This is recommended before you proceed any further.** To do this, use the command:

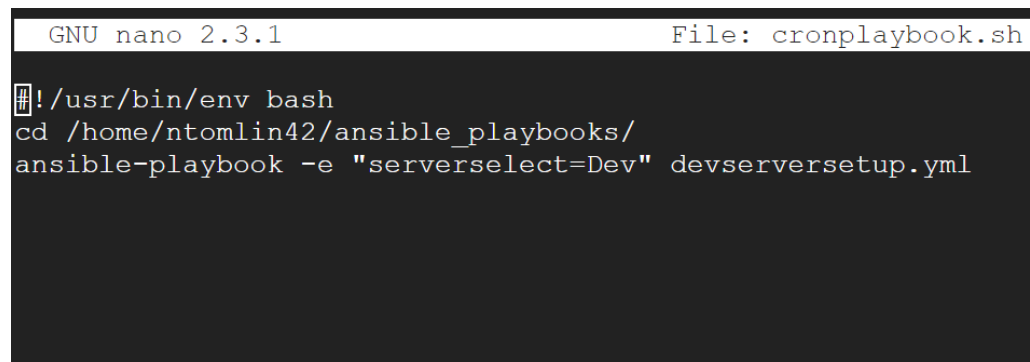
```
Ansible-playbook -e "serverselect=environment" environmentserversetup.yml
```

Where **environment** is the specific ansible server you are on (Dev, Test, or Prod).

If there are no errors when running the playbook, then you should be good to go. However, to make sure everything is working correctly, navigate over to the ws01/db01 server for the particular environment you are on and cat the webrepolog.txt file (found in the swollenhippo directory). If working correctly, you should see the output message we appended to the file when we made the cloneorrepo shellscript.

SECTION 7: THE CRONJOB

After finishing the playbooks for all 3 ansible servers, the last task we need to do is to create cronjob(s) to run our ansible playbook(s) on a scheduled timer of once per minute. To do this, we will first need to create a shell script that will execute our playbook.



```
GNU nano 2.3.1 File: cronplaybook.sh

#!/usr/bin/env bash
cd /home/ntomlin42/ansible_playbooks/
ansible-playbook -e "serverselect=Dev" devserversetup.yml
```

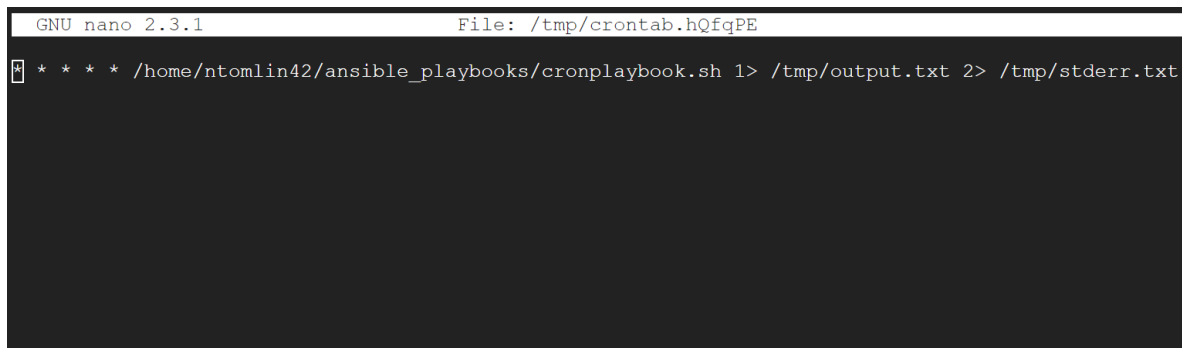
The shell script should be created with the name **cronplaybook.sh**. You'll notice that the shell script has 2 lines (3 including the shebang), one of which is the cd command followed by the directory in which your playbook is located. Make sure to include this, because if left out cron will have trouble finding the playbook. The following line is how we would normally call our command in the terminal but notice the **-e** as well as the **"serverselect=Dev"**. Notice that this is the variable we had inside of the ansible playbook to be passed through when executing the shell script. You will want this **serverselect** variable to be equal to whatever server environment

you are executing this script on (**Dev**, **Test**, or **Prod**). You will also want to change the playbook to correspond to the correct playbook name for the specific ansible server you are on as well.

Next, we need to add this to the crontab in order for it to automatically run our script for us. We can do this by first opening the crontab by using the following command:

```
$ sudo EDITOR=nano crontab -e
```

This will bring us inside of the crontab editor, where we can place the directory of the files we want cron to run automatically.

A screenshot of a terminal window showing the nano text editor. The title bar at the top reads "GNU nano 2.3.1" and "File: /tmp/crontab.hQfqPE". The editor content shows a single line of a cron job: "* * * * /home/ntomlin42/ansible_playbooks/cronplaybook.sh 1> /tmp/output.txt 2> /tmp/stderr.txt". The line is highlighted in blue. The rest of the editor is empty.

Once inside of the crontab, you can place the path to your file as shown above. Make sure to change the filepath to the one that corresponds to your ansible server. Here you can see that the file path is also preceded by 5 asterisks (*), which tells cron how often you want it to execute your file (we want it to be ran every minute, so we just use * * * * *). Note that everything past the cronplaybook.sh is not required but can be helpful as when added it will write any errors to the stderr.txt file located in the tmp directory. This is very helpful for debugging, because the crontab log will not usually show error messages. When finished, make sure to copy these steps to the remaining 2 ansible servers, making sure to change the value of the **serverselect** variable as well as the **playbook name**.

SECTION 8: OUTCOME

After completing all necessary sections as listed above, you should now have 3 fully functional automated ansible servers which will have set up each of their respective web/database servers. Each sub-server should also have a cloned repository initialized with the corresponding branch. In addition to this, each sub-server should be updated by the minute, which includes pulling the most recent changes from the remote repository, as well as checking to make sure all of the latest packages are installed. The outcome should look something like this:

```
ntomlin42@devdb01:~/swollenhippweb$ cat webrepolog.txt
Cloned successfully on 2023-12-03 06:16:39! Chosen branch: Dev
Successfully pulled repository on 2023-12-03 06:17:10. Chosen branch: Dev
Successfully pulled repository on 2023-12-03 06:27:27. Chosen branch: Dev
Successfully pulled repository on 2023-12-03 06:43:47. Chosen branch: Dev
Successfully pulled repository on 2023-12-03 06:48:44. Chosen branch: Dev
Successfully pulled repository on 2023-12-03 06:52:24. Chosen branch: Dev
Successfully pulled repository on 2023-12-03 06:59:16. Chosen branch: Dev
Successfully pulled repository on 2023-12-03 07:11:38. Chosen branch: Dev
Successfully pulled repository on 2023-12-03 18:26:09. Chosen branch: Dev
Successfully pulled repository on 2023-12-03 18:33:20. Chosen branch: Dev
Successfully pulled repository on 2023-12-04 01:49:50. Chosen branch: Dev
Successfully pulled repository on 2023-12-04 01:50:10. Chosen branch: Dev
Successfully pulled repository on 2023-12-04 01:51:10. Chosen branch: Dev
Successfully pulled repository on 2023-12-04 01:52:11. Chosen branch: Dev
Successfully pulled repository on 2023-12-04 01:53:10. Chosen branch: Dev
```