Project 10 - Automating a LAMP Web App Installation

Goal:

The goal of this project is to deploy the WordPress web application by using a shell script. It's written in PHP and uses a MariaDB backend, so you'll configure the LAMP stack for this web application.

Instructions:

Create a Virtual Machine

First, start a command line session on your local machine. Next, move into the working folder you created for this course.

```
cd linuxclass
```

Initialize the vagrant project using the usual process of creating a directory, changing into that directory, and running "vagrant init". We'll name this vagrant project "wordpress".

```
mkdir wordpress
cd wordpress
vagrant init jasonc/centos8
```

Configure the Virtual Machine

Edit the Vagrantfile and set the hostname of the virtual machine to "wordpress". Also, assign the IP address of 10.23.45.60 to the machine.

```
config.vm.hostname = "wordpress"
config.vm.network "private_network", ip: "10.23.45.60"
```

Start the Virtual Machine

Now you're ready to start the VM and connect to it.

```
vagrant up
vagrant ssh
```

Create a Script

Open a new file for editing. Let's call it install.sh. We'll test our script as we go. Once we have it the way we want, we'll use it to perform automated installations going forward.

```
nano install.sh
```

As you know, superuser privileges are required to install software, start services, etc. These are things that this script is going to do. Let's put a check at the top of the script to ensure that it is running with superuser privileges. If it isn't, let's force our script to exit because if it continued it would at best just fail and at worst create some unforeseen issues.

By the way, the advantage to this strategy over using sudo commands throughout the script is that it eliminates the additional layer of sudo configuration. If you relied on sudo, you would have to make sure that each one of the commands you intend to run are allowed by sudo. If there are varying sudo configurations per user, then it would be hard to ensure consistent results of the script.

Insert this bit of code into the very top of your script. Remember to always start your scripts with a shebang!

```
#!/bin/bash

if [ $(id -u) -ne 0 ]
then
  echo 'Please run with sudo or as root.'
  exit 1
fi
```

This snippet checks to see if the output of the "`id -u`" command is not equal to 0. The "`id -u`" command returns the UID of the user running the script. If the root user is running the script, then "`id -u`" will return 0 as root always has a UID of 0. If it is run as any other user, then "`id -u`" will return something other than 0. For example, it might return 1000.

If the UID is not equal to 0, then we tell the user how to proceed and then exit the script. Since the script didn't perform its intended function, we return a non-zero exit status.

Advanced Scripting / Styling - Optional - Feel Free to Skip

One of the reasons I love Linux and scripting is that there are usually more than one way to perform the same action. Bash initializes an environment variable named EUID which expands to the effective user ID of the current user. The contents of this variable contain the same result as the "`id -u`" command. This snippet would work equally as well as the previous one.

```
#!/bin/bash

if [ "$EUID" -ne 0 ]
then
  echo 'Please run with sudo or as root.'
  exit 1
fi
```

The `if` statement checks to see if `[ "$EUID" -ne 0 ]` evaluates to true. If it does, then it executes the code in the `if` block. Since `[ "$EUID" -ne 0 ]` evaluates to true or false, we can use it in conjunction with an && (AND) or an || (OR).   We'll use this statement with || which will execute whatever follows the || when the statement is false. Bash lets you surround a block of code within curly brackets and calls it an inline group.

Here is the refined test:

```
[ "$EUID" -eq 0 ] || {
  echo 'Please run with sudo or as root.'
  exit 1
}
```

You can think of this logic as being "either the user is not the root user OR we tell the user how to proceed and then exit the script."

Any of the three above code snippets have the same end result. Use whatever you understand and feels best to you. Again, there are multiple ways to accomplish the same task. Getting the desired result matters most, followed by readability and programmer happiness in my opinion.

END of the Advanced Scripting Excursion - Start Again Here If you Skipped the Above

Now we're ready to test the code we've created so far. First, let's make the script executable.

```
chmod 755 install.sh
```

Now, let's run it without root privileges to ensure that we get an error message.

```
./install.sh
```

You can also check the exit code by looking at the value of the special variable $?. If you don't run the script as root, the exit code will be 1.

```
./install.sh
echo $?
```

Now execute it with root privileges and check the exit code. Since the script completed without an issue, it should be 0.

```
sudo ./install.sh
echo $?
```

Now that we have our safety check in place we can keep on going. Let's open the file back up and add some more code.

```
nano install.sh
```

We know that we need to install some software. If we use "dnf install PACKAGE", then dnf will prompt for confirmation. We want this script to run without any user input, so we need to make sure we use the "-y" option to automatically answer "yes" to any of dnf's questions. We can also make the output more script friendly by using the "-q" option which make dnf "quiet".

Let's add the following commands in the script to install Apache, PHP, and the required PHP modules.

```
# Install Apache, PHP, and PHP Modules
dnf -q install -y httpd php php-mysqlnd
```

If you wanted to, you could use multiple dnf commands. Maybe the first command could install the web server and the second one could install PHP and its modules.

Note that I decided to use a comment above the installation command. Comments can be used to state in plain language what the following code is intended to do, or it can be abbreviated documentation as is the case here.

At this point the web server can be started and enabled. Let's add the commands to do that in the script next.

```
# Start and enable the web server
systemctl start httpd
systemctl enable httpd
```

Now we can install MariaDB. Add the following to your script.

```
# Install MariaDB
dnf -q install -y mariadb-server

# Start and enable MariaDB
systemctl start mariadb
systemctl enable mariadb
```

When we installed MariaDB manually, this is when we would run the `mysql_secure_installation` script. However, this script prompts you for information. In just a moment we'll talk about a couple of ways to get around this. For now, we'll take advantage of the MariaDB root user not requiring a password to create the required database and user for our web application.

Let's use the `mysqladmin` command to create the wordpress database. Append the following commands to the script.

```
# Create a wordpress database
mysqladmin create wordpress
```

Next, we need to create a database user that can access that database. When performing manual installations we started the `mysql` (MariaDB) client and typed in commands directly to that client. However, you execute SQL statements right from the command line by using the "-e" option followed by the statement you want to execute.

Add the following to the script in order to create a MariaDB user that has full permissions to the "wordpress" database. This command also sets a password for the MariaDB user. Remember to flush the privileges to make the changes take effect.

```
# Create a user for the wordpress database
mysql -e "GRANT ALL on wordpress.* to wordpress@localhost identified by 'wordpress123';"
mysql -e "FLUSH PRIVILEGES;"
```

The work we've done so far in the script is safe to run multiple times. For example, if MariaDB is already installed, dnf will simply report that it's already installed. Also, if you try to start a service that is already started, nothing happens. If you try to create a database that already exists, you'll get an error but it will not alter the existing database. This is a good time to test our script and make sure we haven't made any typos or logic errors up to this point.

Save your changes and test your work by executing the script with root privileges.

```
sudo ./install.sh
```

Now you should see the web server and database server running.

```
ps -ef | grep httpd
ps -ef | grep mysql
```

You should also be able to connect to MariaDB as the wordpress user and see the wordpress database. (Note: the wordpress user should NOT be able to see the mysql database. Only the root MariaDB user should be able to see and manipulate that database.)

```
mysqlshow -u wordpress -pwordpress123
```

If you get any errors or if the web server and database server didn't start, check your work, make any required changes and execute the script again.

Let's get back to scripting.

```
nano install.sh
```

Now that we are done with the initial MariaDB installation and configuration, let's tackle the problem of securing it. Remember, that the `mysql_secure_installation` command prompts you for information when you execute it. There are a couple of ways around this.

The first way is to manually run the script and make a note of your exact responses. Then, you can use an echo statement to "type" those responses for you by piping in the answers to the `mysql_secure_installation` command.

You can use the "`-e`" option to `echo` to enable backslash escapes. This can allow you to simulate ENTER with "`\n`" which represents the line feed character. For example, if you want so simulate <ENTER><ENTER>rootpassword123<ENTER>rootpassword123<ENTER>, you would use this command: `echo "\n\nrootpassword123\nrootpassword123\n"`. Anywhere you would normally hit the ENTER key use "\n".

Using this first method, here is how to automate the inputs for the `mysql_secure_installation` command.

```
# Secure MariaDB
echo -e "\n\nrootpassword123\nrootpassword123\n\n\n\n\n" | mysql_secure_installation
```

Another approach, is to replicate what the `mysql_secure_installation` command does ourselves in our own script without calling it at all. That command does these things:

- Removes access to any databases that start with the name of "test".
- Deletes the test database.
- Removes anonymous user accounts
- Removes root accounts that are accessible from outside the localhost.
- Sets a password for the root account.

To duplicate this functionality, use the following commands.

```
# Remove the test DB privileges.
mysql -e "DELETE FROM mysql.db WHERE Db='test' OR Db='test\\_%'"

# Drop the test DB.
mysqladmin drop -f test

# Remove anonymous DB users.
mysql -e "DELETE FROM mysql.user WHERE User='';"

# Remove remote root DB account access.
mysql -e "DELETE FROM mysql.user WHERE User='root' AND Host NOT IN
('localhost', '127.0.0.1', '::1');"

# Set a root DB password
mysql -e "UPDATE mysql.user SET Password=PASSWORD('rootpassword123') WHERE User='root';"

# Flush the privileges
mysql -e "FLUSH PRIVILEGES;"
```

Pick one of the two previous methods and append it to your script.

Next, our script needs to download the web application, extract it, and put it into the web server DocumentRoot. We'll use the `mktemp` command with a "`-d`" option to create a temporary directory that we can use to download and extract the web application. When you execute that command a unique directory will be created and the path to that directory will be returned as output. We'll store that output -- the path to that directory -- in a variable, so we can easily use this directory.

Append the following commands to your script.

```
# Download and extract WordPress
TMP_DIR=$(mktemp -d)
cd $TMP_DIR
curl -sOL https://wordpress.org/wordpress-5.5.1.tar.gz
tar zxf wordpress-5.5.1.tar.gz
mv wordpress/* /var/www/html
```

You're probably already familiar with the "-O" (save file) and "-L" (obey redirects) options to curl. We added the "-s" option which puts curl in silent mode. This suppresses the progress meter.

Let's clean up behind ourselves by changing out of the temporary directory we created and then removing it.

```
# Clean up
cd /
rm -rf $TMP_DIR
```

We could actually stop here. As a matter of fact, for some web applications this is where you would need to stop. From here you would complete the installation process by opening up a web browser and running through the setup process, answering questions about what database the web application should use and so on. The majority of setup "wizards" simply take your answers and store them directly in a text file on the web server. Additionally, they may create the initial database layout (schema) and create an administrator user for the web application.

For other web applications, the complete process can be performed from the command line and thus scripted. For some applications you can create a configuration file or populate some values from a template they provide and you're done. Yet others may require you to execute some SQL to create the database schema.

Still, other web applications come with command line tools. There is one such command line tool for WordPress aptly named WP-CLI. We can use that in place of pulling up a web browser and answering questions.

WP-CLI requires the PHP-JSON module, so we'll need to install that in addition to the tool itself. Let's add it to our script.

```
# Install the wp-cli tool
dnf -q install -y php-json
curl -sOL https://raw.github.com/wp-cli/builds/gh-pages/phar/wp-cli.phar
mv wp-cli.phar /usr/local/bin/wp
chmod 755 /usr/local/bin/wp
```

Now we can use WP-CLI to create the WordPress configuration file, `/var/www/html/wp-config.php`. The command is "`wp core config`" followed by options where we'll supply the database name, database user, and database password.

```
# Configure wordpress
cd /var/www/html
/usr/local/bin/wp core config --dbname=wordpress --dbuser=wordpress \
--dbpass=wordpress123
```

Finally, we're ready to perform the actual installation with "`wp core install`". Instead of answering questions in a guided web based installer, we're providing those answers on the command line as options.

```
# Install wordpress
/usr/local/bin/wp core install --url=http://10.23.45.60 \
--title="Blog" --admin_user="admin" --admin_password="admin" \
--admin_email="vagrant@localhost.localdomain"
```

Let's execute our script to install WordPress on our system.

```
sudo ./install.sh
```

You'll see messages stating that the packages we are trying to install have already been installed. That's okay. That's from our earlier test. The database create command will fail because we already created the database. That's fine too. If you see any other error messages, go back and fix the script and run it again.

Now WordPress is available here: http://10.23.45.60. You can even login with the username and password we supplied early here: http://10.23.45.60/wp-login.php.

By default, Vagrant shares your project directory (the directory with the Vagrantfile) to /vagrant. Let's save a copy of our script to our local system by copying it into /vagrant.

```
cp install.sh /vagrant
```

Let's log out and make sure our script made it to our local workstation.

```
exit
```

If you're on a Mac, you can use the "`ls`" command to list files. If you're on Windows, use the "`dir`" command.

```
# Mac:
ls

# Windows:
dir
```

If you see the `install.sh` file, it was copied to your local machine successfully. If the file is not present, you can use this method to scp (secure copy) the file to your local system.

```
vagrant plugin install vagrant-scp
vagrant scp :install.sh .
```

Now the script should be on your local system.

Automating the Installation with Vagrant

Provisioners in Vagrant allow you to automatically install software, alter configurations, and more on the machine as part of the vagrant up process. We'll use the shell provisioner which uploads the script you specify into the virtual machine and executes it as root.

Add this line to the Vagrant file.

```
config.vm.provision "shell", path: "install.sh"
```

Here is the full Vagrant file without comments:

```
Vagrant.configure(2) do |config|
  config.vm.box = "jasonc/centos8"
  config.vm.hostname = "wordpress"
  config.vm.network "private_network", ip: "10.23.45.60"
  config.vm.provision "shell", path: "install.sh"
end
```

Let's destroy the virtual machine, removing all traces of our WordPress installation.

```
vagrant destroy
```

Now we when bring up the virtual machine, Vagrant will upload the install.sh script and execute it in the virtual machine. If all goes well, we'll have a working installation of WordPress all by running one command.

```
vagrant up
```

Now WordPress is available here: http://10.23.45.60.

Note that the provisioning only gets run the first time the virtual machine is brought up. If you want to force a run of the provisioner, run "`vagrant provision`". You can also run "`vagrant reload --provision`" to halt the vm, start it again, and run the provisioner.

Here is the final `install.sh` script:

```
#!/bin/bash

if [ $(id -u) -ne 0 ]
then
  echo 'Please run with sudo or as root.'
  exit 1
fi

# Install Apache, PHP, and PHP Modules
dnf -q install -y httpd php php-mysqlnd

# Start and enable the web server
systemctl start httpd
systemctl enable httpd

# Install MariaDB
dnf -q install -y mariadb-server

# Start and enable MariaDB
systemctl start mariadb
systemctl enable mariadb

# Create a wordpress database
mysqladmin create wordpress

# Create a user for the wordpress database
mysql -e "GRANT ALL on wordpress.* to wordpress@localhost identified by 'wordpress123';"
mysql -e "FLUSH PRIVILEGES;"

# Secure MariaDB
echo -e "\n\nrootpassword123\nrootpassword123\n\n\n\n\n" \
| mysql_secure_installation
```

```
# Download and extract WordPress
TMP_DIR=$(mktemp -d)
cd $TMP_DIR
curl -sOL https://wordpress.org/wordpress-5.5.1.tar.gz
tar zxf wordpress-5.5.1.tar.gz
mv wordpress/* /var/www/html

# Install the wp-cli tool
dnf -q install -y php-json
curl -sOL https://raw.github.com/wp-cli/builds/gh-pages/phar/wp-cli.phar
mv wp-cli.phar /usr/local/bin/wp
chmod 755 /usr/local/bin/wp

# Clean up
cd /
rm -rf $TMP_DIR

# Configure wordpress
cd /var/www/html
/usr/local/bin/wp core config --dbname=wordpress --dbuser=wordpress \
--dbpass=wordpress123

# Install wordpress
/usr/local/bin/wp core install --url=http://10.23.45.60 \
--title="Blog" --admin_user="admin" --admin_password="admin" \
--admin_email="vagrant@localhost.localdomain"
```