Deploying WordPress

In the previous chapter, we built a playbook that installs and configures a basic **LAMP stack**. In this chapter, we will be building on top of the techniques we used there to create a playbook that installs a **LEMP stack**, which, as you might recall, replaces Apache with NGINX and then installs WordPress.

Once we finish this chapter, you should be able to do the following:

* Prepare our initial playbook
* Download and install the WordPress CLI
* Install and configure WordPress
* Log in to your WordPress installation

The chapter covers the following topics:

* Preinstallation tasks
* The stack\_install role
* The stack\_config role
* The wordpress role
* Running the WordPress playbook

Before we start, we should quickly cover what WordPress is; you have likely visited a website powered by WordPress at some point in the last 24 hours.

It is an open-source **content management system** (**CMS**) powered by PHP and MySQL and used by around 810 million websites, which is around 43% of all the websites on the internet today, according to the statistics published by Colorlib in August of 2023.

Technical requirements

Like in [*Chapter 4*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/4), *Deploying a LAMP Stack*, we will use the local **Multipass virtual machine** we have been using throughout the title. Again, additional packages will be downloaded when launching the virtual machine and deploying WordPress.

You can find a complete copy of the playbook in the repository accompanying this title at https://github.com/PacktPublishing/Learn-Ansible-Second-Edition/tree/main/Chapter05.

Preinstallation tasks

As mentioned in [*Chapter 4*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/4), *Deploying a LAMP Stack*, a LEMP stack is composed of the following elements:

* **Linux**: In our case, this will be the Ubuntu Multipass virtual machine
* **NGINX**: If you remember, it is pronounced as *engine-x*, which means there is an *E* in *LEMP* and not an *N* (which would also make it impossible to pronounce as an abbreviation)
* **MariaDB**: As we have already seen, this will be the database component
* **PHP**: We will be using PHP 8 again for this

Before we install WordPress, we need to install and configure these components. Also, as this playbook will eventually be executed against publicly available cloud servers, we must consider some best practices around our NGINX configuration.

However, before we start looking at the playbook, let’s start things off by getting the initial structure of the playbook set up:

$ mkdir Chapter05 Chapter05/group\_vars Chapter05/roles

$ touch Chapter05/group\_vars/common.yml Chapter05/hosts Chapter05/site.yml

$ cd Chapter05

CopyExplain

This gives us our basic layout. Next, we must copy the cloud-init.yaml, example\_key, example\_key.pub, and hosts.example files from the previous chapters, so when it is time to run the playbook, we have everything we need to launch the virtual machine using Multipass.

Now that we have the basics configured, we can make a start by writing the playbook to deploy and configure our initial software stack.

The stack\_install role

We are going to start by creating a role called stack\_install using ansible-galaxy role init:

$ ansible-galaxy role init roles/stack\_install

CopyExplain

This will install our initial software stack. Once installed, we hand it over to a second role, which will then configure the software stack before a third role starts the WordPress installation.

So, what packages do we need? WordPress has the following requirements:

* PHP 7.4 or greater
* MySQL 5.7 or greater OR MariaDB 10.4 or greater
* Nginx or Apache with the mod\_rewrite module
* HTTPS support

We know from the previous chapter that the versions of PHP and MariaDB we are installing meet this requirement, leaving just NGINX, which we can download and install from the principal NGINX repository to get the latest and greatest version.

Enabling the NGINX repository

Before we look at the tasks and variables that we will need to enable the mainline NGINX repository, let’s start off the roles/stack\_install/tasks/main.yml file with a task that updates the operating system and the cache of available packages:

- name: "Update apt-cache and upgrade packages"

  ansible.builtin.apt:

    name: "\*"

    state: "latest"

    update\_cache: true

CopyExplain

The remainder of the tasks we will be defining enable the repository before we finally install the packages.

Moving onto the roles/stack\_install/default/main.yml file, we need to set some variables containing information on the repository, which we will add alongside the default Ubuntu ones.

These variables start with one that contains the URL of the signing key for the repository that will be enabled:

repo\_keys\_url:

  - "http://nginx.org/keys/nginx\_signing.key"

CopyExplain

We will then add the following repository URLs:

repo\_packages:

  - "deb http://nginx.org/packages/mainline/ubuntu/ {{ ansible\_distribution\_release }} nginx"

  - "deb-src http://nginx.org/packages/mainline/ubuntu/ {{ ansible\_distribution\_release }} nginx"

CopyExplain

You may have noticed that we are using the ansible\_distribution\_release fact to dynamically run into the URL to put the correct version number of the Ubuntu distribution.

Now, back to the roles/stack\_install/tasks/main.yml file and the two tasks that call these variables – these will look like the following, starting with the addition of the signing key:

- name: "Add the apt keys from a URL"

  ansible.builtin.apt\_key:

    url: "{{ item }}"

    state: "present"

  with\_items: "{{ repo\_keys\_url }}"

CopyExplain

As you can see, we are using with\_items, so, if you need to, you could define more than one URL and add additional signing keys.

This approach is carried forward to the next task, where we are adding more than one repository:

- name: "Install the repo packages"

  ansible.builtin.apt\_repository:

    repo: "{{ item }}"

    state: "present"

    update\_cache: true

  with\_items: "{{ repo\_packages }}"

CopyExplain

The final task in the roles/stack\_install/tasks/main.yml file is the one that installs all of the packages:

- name: "Update cache and install the stack packages"

  ansible.builtin.apt:

    state: "present"

    update\_cache: true

    pkg: "{{ system\_packages + extra\_packages + stack\_packages }}"

CopyExplain

You will notice that rather than defining the packages in a single variable, I have split them into three, and we are combining them by using + when calling the variables.

So, what do these three variables contain, and why don’t we define them as a single variable?

Back to the roles/stack\_install/default/main.yml file, you can see that system\_packages is defined as the following:

system\_packages:

  - "software-properties-common"

  - "python3-pymysql"

  - "acl"

CopyExplain

Following that, the extra\_packages variable contains the following package list:

extra\_packages:

  - "vim"

  - "git"

  - "unzip"

CopyExplain

Finally, we have the list of packages that make up the bulk of our software stack:

stack\_packages:

  - "nginx"

  - "mariadb-server"

  - "mariadb-client"

  - "php-cli"

  - "php-curl"

  - "php-fpm"

  - "php-gd"

  - "php-intl"

  - "php-mbstring"

  - "php-mysql"

  - "php-soap"

  - "php-xml"

  - "php-xmlrpc"

  - "php-zip"

CopyExplain

As we are defining three variables for the packages, it means that we can, if required, overwrite them elsewhere in our playbook.

Let us, for example, assume that we need to install the Amazon Web Services command-line tool on the virtual machine.

This would allow us to push data, such as images, to an Amazon S3 bucket or clear a cache on a CloudFront content delivery network endpoint.

Rather than overriding a long list of packages from a single variable, we could take the extra\_packages variable, add it to group\_vars/common.yml, and append it to the end of the list of packages so that it will now look like the following:

extra\_packages:

  - "vim"

  - "git"

  - "unzip"

  - "awscli"

CopyExplain

As you can see, this is a lot more efficient than repeating all the packages we want to install.

Another advantage of using + to combine everything is that we only need to call a single ansible.builtin.apt task to install everything we need for the following role, which we will dive into now.

The stack\_config role

Now that we have our base software stack installed we need to configure it, let’s start by creating the role by running the following command:

$ ansible-galaxy role init roles/stack\_config

CopyExplain

This gives us the basic file structure needed for the stack\_config role. With that in place, we can now look at configuring the role itself – in this role, we will need to do the following:

* Add a system user for our WordPress installation to run under
* Configure NGINX as per the best practices on the WordPress documentation
* Configure PHP-FPM to run as the WordPress user we created earlier

As we need a user for WordPress to run under, we should make a start there.

WordPress system user

The defaults for the WordPress system user, which should be placed in roles/stackconfig/defaults/main.yml, are as follows:

wordpress\_system:

  user: "wordpress"

  group: "php-fpm"

  comment: "wordpress system user"

  home: "/var/www/wordpress"

  state: "present"

CopyExplain

We refer to this as the system user, as we will create a user in WordPress itself later in the chapter. This user’s details will also be defined in Ansible, so we do not want to get the two different users confused.

The two tasks that use these variables, found in roles/stack\_config/tasks/main.yml, should look like this:

- name: "add the wordpress group"

  ansible.builtin.group:

    name: "{{ wordpress\_system.group }}"

    state: "{{ wordpress\_system.state }}"

CopyExplain

The preceding task ensures that the group is present, and the next task adds an operating system-level user, which is added to the group that has just been created:

- name: "Add the wordpress user"

  ansible.builtin.user:

    name: "{{ wordpress\_system.user }}"

    group: "{{ wordpress\_system.group }}"

    comment: "{{ wordpress\_system.comment }}"

    home: "{{ wordpress\_system.home }}"

    state: "{{ wordpress\_system.state }}"

CopyExplain

As you can see, we are not adding a key to the user this time as we don’t want to log in to the user account to start manipulating files and other actions. This should all be done within WordPress itself or by using Ansible.

NGINX configuration

We are going to be using several template files for our NGINX configuration. The first template is called roles/stack\_config/templates/nginx-nginx.conf.j2, and it will replace the main NGINX configuration deployed by the package installation:

# {{ ansible\_managed }}

user  nginx;

worker\_processes  {{ ansible\_processor\_count }};

error\_log  /var/log/nginx/error.log warn;

pid        /var/run/nginx.pid;

events {

    worker\_connections  1024;

}

http {

    include       /etc/nginx/mime.types;

    default\_type  application/octet-stream;

    log\_format  main  '$remote\_addr - $remote\_user [$time\_local] "$request" '

                      '$status $body\_bytes\_sent "$http\_referer" '

                      '"$http\_user\_agent" "$http\_x\_forwarded\_for"';

    access\_log  /var/log/nginx/access.log  main;

    sendfile        on;

    keepalive\_timeout  65;

    client\_max\_body\_size 20m;

    include /etc/nginx/conf.d/\*.conf;

}

CopyExplain

The file’s content is the same as the file that will be replaced, except that we are updating worker\_processes to use the number of processors detected by Ansible when the setup module runs rather than a hardcoded value.

The task to deploy the configuration file is as you would expect, and it should be placed in roles/stack\_config/tasks/main.yml:

- name: "Copy the nginx.conf to /etc/nginx/"

  ansible.builtin.template:

    src: nginx-nginx.conf.j2

    dest: /etc/nginx/nginx.conf

    mode: "0644"

  notify: "Restart nginx"

CopyExplain

As you can see, we are notifying the restart nginx handler, which is stored in the roles/stack\_config/handlers/main.yml file:

- name: "Restart nginx"

  ansible.builtin.service:

    name: nginx

    state: restarted

    enabled: true

CopyExplain

Next, we have the default site template, roles/stack\_config/templates/nginx-confd-default.conf.j2:

# {{ ansible\_managed }}

upstream {{ php.upstream }} {

        server {{ php.ip }}:{{ php.port }};

}

server {

     listen       80;

     server\_name  {{ ansible\_nodename }};

     root         {{ wordpress\_system.home }};

     index        index.php index.html index.htm;

    include global/restrictions.conf;

    include global/wordpress\_shared.conf;

}

CopyExplain

To help identify where the template files will be placed on the target host, I am naming them so that the full path is in the filename. In this case, the filename is nginx-confd-default.conf.j2, and it will be deployed to /etc/nginx/conf.d/default.conf; the task to do this follows:

- name: "Copy the default.conf to /etc/nginx/conf.d/"

  ansible.builtin.template:

    src: nginx-confd-default.conf.j2

    dest: /etc/nginx/conf.d/default.conf

    mode: "0644"

  notify: "Restart nginx"

CopyExplain

The following two files we are deploying are going into a folder that doesn’t exist. So, we first need to create the destination folder. To do this, we need to add the following to roles/stack\_config/tasks/main.yml:

- name: "Create the global directory in /etc/nginx/"

  ansible.builtin.file:

    dest: /etc/nginx/global/

    state: directory

    mode: "0644"

CopyExplain

As we are not making any replacements in the nginx-global-restrictions.conf file, we are using the ansible.builtin.copy module rather than ansible.builtin.template here; the file is stored in roles/stack\_config/files/ and the task that copies it is as follows:

- name: "Copy the restrictions.conf to /etc/nginx/global/"

  ansible.builtin.copy:

    src: nginx-global-restrictions.conf

    dest: /etc/nginx/global/restrictions.conf

    mode: "0644"

  notify: "Restart nginx"

CopyExplain

This file has some sensible defaults in it, such as denying access to files that are included as part of the WordPress installation:

location ~\* /(wp-config.php|readme.html|license.txt|nginx.conf) {

    deny all;

}

CopyExplain

Another import inclusion is adding a configuration to deny access to .php files within /wp-content/ and its sub-folders:

location ~\* ^/wp-content/.\*.(php|phps)$ {

    deny all;

}

CopyExplain

There are several other configurations in the nginx-global-restrictions.conf file; see the repository, which accompanies the book, for the complete configuration, as there are too many snippets for us to go into here.

The same can be said for the next and final block of the NGINX configuration; review the repository for more information on the configuration deployed by the following task:

- name: "Copy the wordpress\_shared.conf to /etc/nginx/global/"

  ansible.builtin.template:

    src: nginx-global-wordpress\_shared.conf.j2

    dest: /etc/nginx/global/wordpress\_shared.conf

    mode: "0644"

  notify: "Restart nginx"

CopyExplain

When we reviewed the default site template, roles/stack\_config/templates/nginx-confd-default.conf.j2, you may have noticed the use of a few variables we haven’t yet defined; they were php.ip and php.port.

As you may have already guessed from the variable labeling, these have to do with the configuration of PHP, so, let us look at configuring the PHP and PHP-FPM part of our deployment.

PHP and PHP-FPM configuration

As we saw in the previous section, there are a few variables defined for PHP in roles/stack\_config/defaults/main.yml, and these are as follows:

php:

  ip: "127.0.0.1"

  port: "9000"

  upstream: "php"

  ini:

    - { regexp: "^;date.timezone =", replace: "date.timezone = Europe/London" }

    - { regexp: "^expose\_php = On", replace: "expose\_php = Off" }

    - {

        regexp: "^upload\_max\_filesize = 2M",

        replace: "upload\_max\_filesize = 20M",

      }

CopyExplain

We then have some variables that define some information on the paths for the various files and service names:

php\_fpm\_path: "/etc/php/8.1/fpm/pool.d/www.conf"

php\_ini\_path: "/etc/php/8.1/fpm/php.ini"

php\_service\_name: "php8.1-fpm"

CopyExplain

The first configuration of the two tasks we will be running deploys the PHP-FPM configuration; this is, what the template, which can be found at roles/stack\_config/templates/php-fpmd-www.conf.j2, looks like:

; {{ ansible\_managed }}

[{{ wordpress\_system.user }}]

user = {{ wordpress\_system.user }}

group = {{ wordpress\_system.group }}

listen = {{ php.ip }}:{{ php.port }}

listen.allowed\_clients = {{ php.ip }}

pm = dynamic

pm.max\_children = 50

pm.start\_servers = 5

pm.min\_spare\_servers = 5

pm.max\_spare\_servers = 35

php\_admin\_value[error\_log] = /var/log/php-fpm/{{ wordpress\_system.user }}-error.log

php\_admin\_flag[log\_errors] = on

php\_value[session.save\_handler] = files

php\_value[session.save\_path]    = /var/lib/php/fpm/session

php\_value[soap.wsdl\_cache\_dir]  = /var/lib/php/fpm/wsdlcache

CopyExplain

As you can see, we have a few replacements in this file. Starting at the top between the square brackets, we are defining the PHP-FPM pool name and using the content of the wordpress\_system.user variable for this.

Next, we have the user and group we want our pool to run under; here, we use wordpress\_system.user and wordpress\_system.group.

Finally, we are setting the IP address and port we want our PHP-FPM pool to listen on by using the php.ip and php.port variables.

The task in roles/stack\_config/tasks/main.yml to deploy the template looks as follows:

- name: "Copy the www.conf to /etc/php-fpm.d/"

  ansible.builtin.template:

    src: php-fpmd-www.conf.j2

    dest: "{{ php\_fpm\_path }}"

    mode: "0644"

  notify: "Restart php-fpm"

CopyExplain

The handler to restart PHP-FPM in roles/stack\_config/handlers/main.yml is very similar to the ones we have already been defining throughout the book:

- name: "Restart php-fpm"

  ansible.builtin.service:

    name: "{{ php\_service\_name }}"

    state: restarted

    enabled: true

CopyExplain

The next task in roles/stack\_config/tasks/main.yml uses the ansible.builtin.lineinfile module:

- name: "Configure php.ini settings"

  ansible.builtin.lineinfile:

    dest: "{{ php\_ini\_path }}"

    regexp: "{{ item.regexp }}"

    line: "{{ item.replace }}"

    backup: "true"

    backrefs: "true"

  with\_items: "{{ php.ini }}"

  notify: "Restart php-fpm"

CopyExplain

We are taking the php.ini file and looping through it by looking for the values defined by the regexp key. Once we find the value, we replace it with the content of the replace key. If there are changes to the file, we are making a backup first, just in case.

Also, we are using backrefs to ensure that if there is no matching regex in the file, then it will be left unchanged; if we didn’t use them, the restart php-fpm handler would be called every time the playbook runs, and we do not want PHP-FPM to be restarted if there is no reason for it be.

Starting NGINX and PHP-FPM

Now that we have NGINX and PHP-FPM installed and configured, we need to start the two services rather than wait until the end of the playbook run.

If we don’t do this now, our upcoming role to install WordPress will fail. The first of the two tasks in roles/stackconfig/tasks/main.yml looks like the following:

- name: "Start php-fpm"

  ansible.builtin.service:

    name: "{{ php\_service\_name }}"

    state: "started"

CopyExplain

The second task looks pretty much the same:

- name: "Start nginx"

  ansible.builtin.service:

    name: "nginx"

    state: "started"

CopyExplain

If you look at the two tasks, they are the same as the two handlers we have already defined.

However, if you look closer, you will notice that while we are using the ansible.builtin.service module, we are only setting the state setting to started rather than restarted, and we are missing the configuration for enabled, which sets the service to start on boot.

Another thing you may have noticed is the use of the php\_service\_name variable; to explain why we are using this, you will need to wait until [*Chapter 6*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/6), *Targeting Multiple Distributions*.

The final component of our software stack that we need to configure is MariaDB, so let us review that before we move on to the WordPress installation and configuration.

MariaDB configuration

The MariaDB configuration will closely match its configuration in [*Chapter 4*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/4), *Deploying a LAMP Stack*, minus a few steps, so I will not go into too much detail here.

The default variables for this part of the role in roles/stack\_config/defaults/main.yml are as follows:

mariadb:

  bind: "127.0.0.1"

  server\_config: "/etc/my.cnf.d/mariadb-server.cnf"

  username: "root"

  password: "Pa55W0rd123"

  hosts:

    - "127.0.0.1"

    - "::1"

    - "{{ ansible\_nodename }}"

    - "localhost"

CopyExplain

As you can see, we are now using a nested variable and have removed the host wildcard, which we had previously defined as % in [*Chapter 4*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/4), *Deploying a LAMP Stack*.

Our first task is to start MariaDB so that we can interact with it:

- name: "Start mariadb"

  ansible.builtin.service:

    name: "mariadb"

    state: "started"

    enabled: true

CopyExplain

Check for the presence of the ~/.my.cnf file:

- name: "Check to see if the ~/.my.cnf file exists"

  ansible.builtin.stat:

    path: "~/.my.cnf"

  register: mycnf

CopyExplain

Set a password:

- name: "Change mysql root password if we need to"

  community.mysql.mysql\_user:

    name: "{{ mariadb.username }}"

    host: "{{ item }}"

    password: "{{ mariadb.password }}"

    check\_implicit\_admin: "true"

    priv: "\*.\*:ALL,GRANT"

    login\_user: "{{ mariadb.username }}"

    login\_unix\_socket: /var/run/mysqld/mysqld.sock

  with\_items: "{{ mariadb.hosts }}"

  when: not mycnf.stat.exists

CopyExplain

Create the ~/my.cnf file:

- name: "Set up .my.cnf file"

  ansible.builtin.template:

    src: "my.cnf.j2"

    dest: "~/.my.cnf"

    mode: "0644"

CopyExplain

Then, remove the anonymous user:

- name: "Delete anonymous MySQL user"

  community.mysql.mysql\_user:

    user: ""

    host: "{{ item }}"

    state: "absent"

  with\_items: "{{ mariadb.hosts }}"

CopyExplain

Now, we have come to our final task, which is to remove the test database:

- name: "Remove the MySQL test database"

  community.mysql.mysql\_db:

    db: "test"

    state: "absent"

CopyExplain

Now, with everything we need to install and run WordPress configured, we can start on WordPress itself.

The wordpress role

Now that we have completed the roles that prepare our target virtual machine, we can proceed with the actual WordPress installation; this will be split into a few different parts, starting with downloading wp-cli and setting up the database.

Before we progress, we should create the role:

$ ansible-galaxy role init roles/wordpress

CopyExplain

Now that we have the empty role files, we can start populating the tasks and variables in the files.

Some facts

Before installing WordPress, we must set some facts using the ansible.builtin.set\_fact module. The following task, the first in the roles/wordpress/tasks/main.yml file, sets two variables using the information gathered when Ansible first connects to the hosts:

- name: "Set a fact for the wordpress domain"

  ansible.builtin.set\_fact:

    wordpress\_domain: "{{ ansible\_ssh\_host }}"

    os\_family: "{{ ansible\_distribution }} {{ ansible\_distribution\_version }}"

CopyExplain

We will use these two variables when we install WordPress using the WordPress CLI, which we will be downloading and installing next.

WordPress CLI installation

WordPress CLI (wp-cli) is a command-line tool used to administer your WordPress installation; we will be using it throughout the role, so, the first thing our role should do is download it. To do this, we need to download the following variables in roles/wordpress/defaults/main.yml:

wp\_cli:

  download: "https://raw.githubusercontent.com/wp-cli/builds/gh-pages/phar/wp-cli.phar"

  path: "/usr/local/bin/wp"

CopyExplain

Moving back to the roles/wordpress/tasks/main.yml file, we use these two variables in the following task, which downloads wp-cli and places it on our host:

- name: "Download wp-cli"

  ansible.builtin.get\_url:

    url: "{{ wp\_cli.download }}"

    dest: "{{ wp\_cli.path }}"

    mode: "0755"

CopyExplain

Now, we have wp-cli on our host with the correct execute permissions.

Before we start to use wp-cli, we have one more bit of preparation work to do: create the database and user, which we will use with our WordPress installation.

Creating the WordPress database

The next part of the role creates the database our WordPress installation will use; as per the other tasks in this chapter, it uses a nested variable, which can be found in roles/wordpress/defaults/main.yml:

wp\_database:

  name: "wordpress"

  username: "wordpress"

  password: "W04DPr3S5"

CopyExplain

The tasks in roles/wordpress/tasks/main.yml to create the database are as follows:

- name: "Create the wordpress database"

  community.mysql.mysql\_db:

    db: "{{ wp\_database.name }}"

    state: "present"

CopyExplain

Now that the database has been created, we can add the user:

- name: "Create the user for the wordpress database"

  community.mysql.mysql\_user:

    name: "{{ wp\_database.username }}"

    password: "{{ wp\_database.password }}"

    priv: "{{ wp\_database.name }}.\*:ALL"

    state: "present"

  with\_items: "{{ mariadb.hosts }}"

CopyExplain

Notice how we are using the mariadb.hosts variable from the previous role. Now that we have the database created, we can start downloading and installing WordPress.

Downloading, configuring, and installing WordPress

Now that we have everything in place to install WordPress, we can make a start, first by setting some default variables in roles/wordpress/defaults/main.yml:

wordpress:

  domain: "http://{{ wordpress\_domain }}/"

  title: "WordPress installed by Ansible on {{ os\_family }}"

  username: "ansible"

  password: "password"

  email: "test@example.com"

  plugins:

    - "jetpack"

    - "wp-super-cache"

    - "wordpress-seo"

    - "wordfence"

    - "nginx-helper"

CopyExplain

Now that we have our variables, we can start our download if we need to. To find out whether we need to download WordPress, we should check for the presence of an existing WordPress installation. The task to do this in roles/wordpress/tasks/main.yml looks like the following:

- name: "Are the wordpress files already there?"

  ansible.builtin.stat:

    path: "{{ wordpress\_system.home }}/index.php"

  register: wp\_installed

CopyExplain

As you can see, the first task uses the ansible.builtin.stat module to check for an index.php file in our system user’s home directory, which in our case is also the webroot.

If this is the first time that the playbook is being run against the host, then we will need to download WordPress:

- name: "Download wordpresss"

  ansible.builtin.command: "{{ wp\_cli.path }} core download"

  args:

    chdir: "{{ wordpress\_system.home }}"

  become\_user: "{{ wordpress\_system.user }}"

  become: true

  when: not wp\_installed.stat.exists

CopyExplain

This task uses the ansible.builtin.shell module to issue the following command:

$ su wordpress -

$ cd /var/www/wordpress

$ /usr/local/bin/wp core download

CopyExplain

There are a few arguments we should work through before moving on to the next task, which are the following:

* args and chdir: You can pass additional arguments to the ansible.builtin.shell module using args. Here, we are passing chdir, which instructs Ansible to change to the directory we specify before running the shell command we provide.
* become\_user: This is the user we want to run the command as. The command will run as the root user if we do not use this flag.
* become: This instructs Ansible to execute the task as the defined user.

The next task in the playbook sets the correct permissions on the user’s home directory:

- name: "Set the correct permissions on the homedir"

  ansible.builtin.file:

    path: "{{ wordpress\_system.home }}"

    mode: "0755"

  when: not wp\_installed.stat.exists

CopyExplain

Now that WordPress is downloaded, we can start the installation. First, we need to check whether this has already been done:

- name: "Is wordpress already configured?"

  ansible.builtin.stat:

    path: "{{ wordpress\_system.home }}/wp-config.php"

  register: wp\_configured

CopyExplain

If there is no wp-config.php file, then the following task will be executed:

- name: "Sort the basic wordpress configuration"

  ansible.builtin.command: "{{ wp\_cli.path }} core config --dbhost={{ mariadb.bind }} --dbname={{ wp\_database.name }} --dbuser={{ wp\_database.username }} --dbpass={{ wp\_database.password }}"

  args:

    chdir: "{{ wordpress\_system.home }}"

  become\_user: "{{ wordpress\_system.user }}"

  become: true

  when: not wp\_configured.stat.exists

CopyExplain

This is like you logging in and running the following:

$ su wordpress -

$ cd /var/www/wordpress

$ /usr/local/bin/wp core config \

--dbhost=127.0.0.1\

--dbname=wordpress\

--dbuser=wordpress \

--dbpass=W04DPr3S5

CopyExplain

As you can see, we are using Ansible to execute commands as if we had a local terminal open.

Now that we have our wp-config.php file created, with the database credentials in place, we can install WordPress.

First, we need to check whether WordPress has already been installed:

- name: "Do we need to install wordpress?"

  ansible.builtin.command: "{{ wp\_cli.path }} core is-installed"

  args:

    chdir: "{{ wordpress\_system.home }}"

  become\_user: "{{ wordpress\_system.user }}"

  become: true

  ignore\_errors: true

  register: wp\_installed

CopyExplain

As you can see from the presence of the ignore\_errors option, if WordPress is not installed, this command will give us an error. We are then using this to our advantage when registering the results, as you can see from the following task:

- name: "Install wordpress if needed"

  ansible.builtin.command: "{{ wp\_cli.path }} core install --url='{{ wordpress.domain }}' --title='{{ wordpress.title }}' --admin\_user={{ wordpress.username }} --admin\_password={{ wordpress.password }} --admin\_email={{ wordpress.email }}"

  args:

    chdir: "{{ wordpress\_system.home }}"

  become\_user: "{{ wordpress\_system.user }}"

  become: true

  when: wp\_installed.rc == 1

CopyExplain

This task is only executed if the previous task returns an error, which is what happens if WordPress is not installed.

Now that our primary WordPress site is installed, we can continue installing the plugins.

WordPress plugins installation

The final part of our WordPress installation is to download and install the plugins we defined in the wordpress.plugins variable.

As per previous tasks, we will build a little logic into the tasks. First, we run the following task to see whether all the plugins are already installed:

- name: "Do we need to install the plugins?"

  ansible.builtin.command: "{{ wp\_cli.path }} plugin is-installed {{ item }}"

  args:

    chdir: "{{ wordpress\_system.home }}"

  become\_user: "{{ wordpress\_system.user }}"

  become: true

  with\_items: "{{ wordpress.plugins }}"

  ignore\_errors: true

  register: wp\_plugin\_installed

CopyExplain

If the plugins are not installed, this task should fail, so we have ignore\_errors in there.

As you can see, we are registering the results of the entire task, because, if you remember, we are installing several plugins, such as wp\_plugin\_installed.

The next two tasks take the results of wp\_plugin\_installed and use the ansible.builtin.set\_fact module to set a fact depending on the results:

- name: "Set a fact if we don't need to install the plugins"

  ansible.builtin.set\_fact:

    wp\_plugin\_installed\_skip: true

  when: wp\_plugin\_installed.failed is undefined

CopyExplain

The preceding task is set if we don’t need to install any of the plugins, and the following one is used if we need to install at least one of the plugins:

- name: "Set a fact if we need to install the plugins"

  ansible.builtin.set\_fact:

    wp\_plugin\_installed\_skip: false

  when: wp\_plugin\_installed.failed is defined

CopyExplain

As you can see, we are setting wp\_plugin\_installed\_skip to be true or false: if the fact is set to false, then the next task will loop through installing the plugins:

- name: "Install the plugins if we need to or ignore if not"

  ansible.builtin.command: "{{ wp\_cli.path }} plugin install {{ item }} --activate"

  args:

    chdir: "{{ wordpress\_system.home }}"

  become\_user: "{{ wordpress\_system.user }}"

  become: true

  with\_items: "{{ wordpress.plugins }}"

  when: not wp\_plugin\_installed\_skip

CopyExplain

Now that we have the plugins’ tasks defined, we can have a go at running our playbook.

Running the WordPress playbook

To run the playbook and install WordPress, we need to finish walking through the files; site.yml should look as follows:

---

- name: "Install and configure WordPress and supporting software"

  hosts: "ansible\_hosts"

  gather\_facts: true

  become: true

  become\_method: "ansible.builtin.sudo"

  vars\_files:

    - "group\_vars/common.yml"

  roles:

    - "stack\_install"

    - "stack\_config"

    - "wordpress"

CopyExplain

With that out of the way, we can run the playbook.

Note

The Chapter05 folder in the GitHub repository accompanying this title contains the example hosts file and keys for launching a local virtual machine using Multipass. If you are following along, please refer to the instructions in [*Chapter 1*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/1), *Installing and Running Ansible*; these detail how to launch the virtual machine and prepare your hosts file.

As we know, to run the playbook, we need to issue the following command once our Multipass virtual machine is up and running:

$ ansible-playbook -i hosts site.yml

CopyExplain

Let’s cover some of the highlights rather than go through the whole output here, starting with adding the NGINX repository:

TASK [roles/stack\_install : add the apt keys from a URL] \*\*

changed: [ansiblevm] => (item=http://nginx.org/keys/nginx\_signing.key)

TASK [roles/stack\_install : install the repo packages] \*\*\*\*

changed: [ansiblevm] => (item=deb http://nginx.org/packages/mainline/ubuntu/ jammy nginx)

changed: [ansiblevm] => (item=deb-src http://nginx.org/packages/mainline/ubuntu/ jammy nginx)

CopyExplain

As you can see, the name of the Ubuntu release is added – in the example, this is jammy.

When making changes to the php.ini file, only two of the three changes we defined need to be applied, as expose\_php is already set to Off:

TASK [roles/stack\_config : configure php.ini] \*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [ansiblevm] => (item={'regexp': '^;date.timezone =', 'replace': 'date.timezone = Europe/London'})

ok: [ansiblevm] => (item={'regexp': '^expose\_php = On', 'replace': 'expose\_php = Off'})

changed: [ansiblevm] => (item={'regexp': '^upload\_max\_filesize = 2M', 'replace': 'upload\_max\_filesize = 20M'})

CopyExplain

Remember that we set the ignore\_errors flag for some of the checks when it came to installing and configuring WordPress; this is why:

TASK [roles/wordpress : do we need to install wordpress?] \*

fatal: [ansiblevm]: FAILED! => {"changed": true, "cmd": "/usr/local/bin/wp core is-installed", "delta": "0:00:00.142910", "end": "2023-09-17 12:28:16.500304", "msg": "non-zero return code", "rc": 1, "start": "2023-09-17 12:28:16.357394", "stderr": "PHP Warning:  Undefined array key \"HTTP\_HOST\" in /var/www/wordpress/wp-includes/functions.php on line 6135\nWarning: Undefined array key \"HTTP\_HOST\" in /var/www/wordpress/wp-includes/functions.php on line 6135\nPHP Warning:  Undefined array key \"HTTP\_HOST\" in /var/www/wordpress/wp-includes/functions.php on line 6135\nWarning: Undefined array key \"HTTP\_HOST\" in /var/www/wordpress/wp-includes/functions.php on line 6135", "stderr\_lines": ["PHP Warning:  Undefined array key \"HTTP\_HOST\" in /var/www/wordpress/wp-includes/functions.php on line 6135", "Warning: Undefined array key \"HTTP\_HOST\" in /var/www/wordpress/wp-includes/functions.php on line 6135", "PHP Warning:  Undefined array key \"HTTP\_HOST\" in /var/www/wordpress/wp-includes/functions.php on line 6135", "Warning: Undefined array key \"HTTP\_HOST\" in /var/www/wordpress/wp-includes/functions.php on line 6135"], "stdout": "", "stdout\_lines": []}

...ignoring

TASK [roles/wordpress : install wordpress if needed] \*\*\*\*\*\*

changed: [ansiblevm]

CopyExplain

As you can see, an error was ignored, and the task to install WordPress was triggered. The same thing happened for the plugins:

TASK [roles/wordpress : set a fact if we don't need to install the plugins] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

skipping: [ansiblevm]

TASK [roles/wordpress : set a fact if we need to install the plugins] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [ansiblevm]

CopyExplain

On first execution, the recap looked something like the following:

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ansiblevm                  : ok=39   changed=28   unreachable=0    failed=0    skipped=1    rescued=0    ignored=2

CopyExplain

Rerunning the playbook immediately after shows how the logic we have added throughout the task execution kicks in, which results in a lot of the later tasks being skipped entirely:

TASK [roles/wordpress : are the wordpress files already there?] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [ansiblevm]

TASK [roles/wordpress : download wordpresss] \*\*\*\*\*\*\*\*\*\*\*\*\*\*

skipping: [ansiblevm]

CopyExplain

Note that, this time, the check for the plugins doesn’t result in an error:

TASK [roles/wordpress : do we need to install the plugins?]

changed: [ansiblevm] => (item=jetpack)

changed: [ansiblevm] => (item=wp-super-cache)

changed: [ansiblevm] => (item=wordpress-seo)

changed: [ansiblevm] => (item=wordfence)

changed: [ansiblevm] => (item=nginx-helper)

TASK [roles/wordpress : set a fact if we don't need to install the plugins] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [ansiblevm]

TASK [roles/wordpress : set a fact if we need to install the plugins] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

skipping: [ansiblevm]

TASK [roles/wordpress : install the plugins if we need to or ignore if not] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

skipping: [ansiblevm] => (item=jetpack)

skipping: [ansiblevm] => (item=wp-super-cache)

skipping: [ansiblevm] => (item=wordpress-seo)

skipping: [ansiblevm] => (item=wordfence)

skipping: [ansiblevm] => (item=nginx-helper)

CopyExplain

Now that WordPress is installed, we should be able to access it in a browser by going to the host you have defined in your hosts file, in my case, http://192.168.64.26.nip.io/; yours will be different.

You will see the default WordPress site:

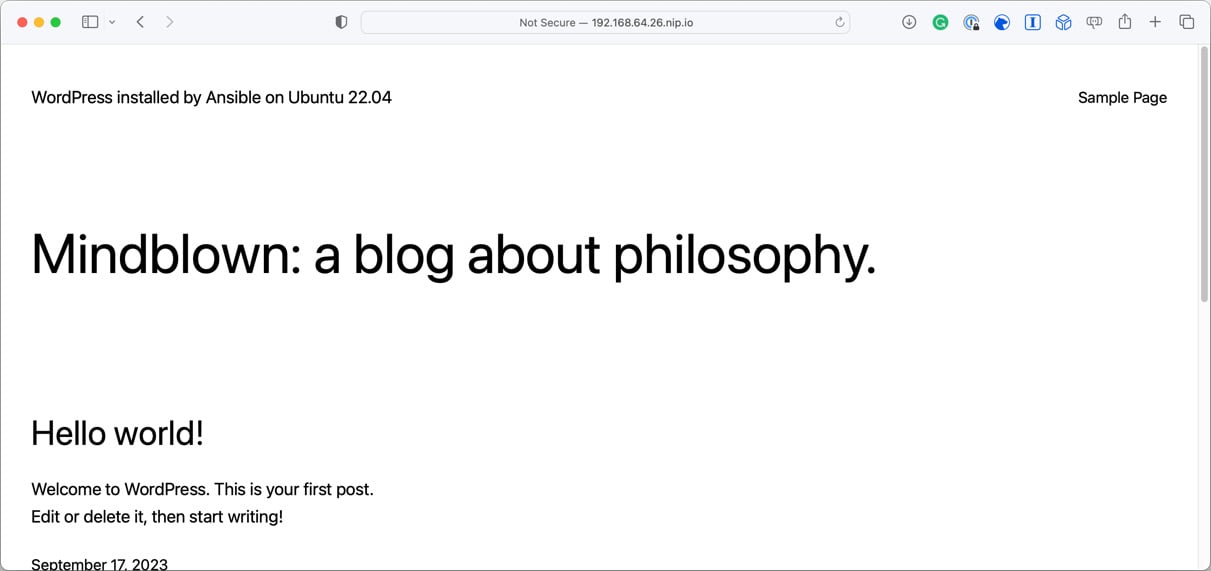


Figure 5.1 – Our freshly installed WordPress website

As you can see, the site’s description in the top left reads **WordPress installed by Ansible on Ubuntu 22.04**, which is what we set when installing WordPress.

Also, if you go to the WordPress admin area by appending /wp-admin/ to the end of your URL, for example, http://192.168.64.26.nip.io/wp-admin/, you should be able to log in to WordPress using the username and password we defined:

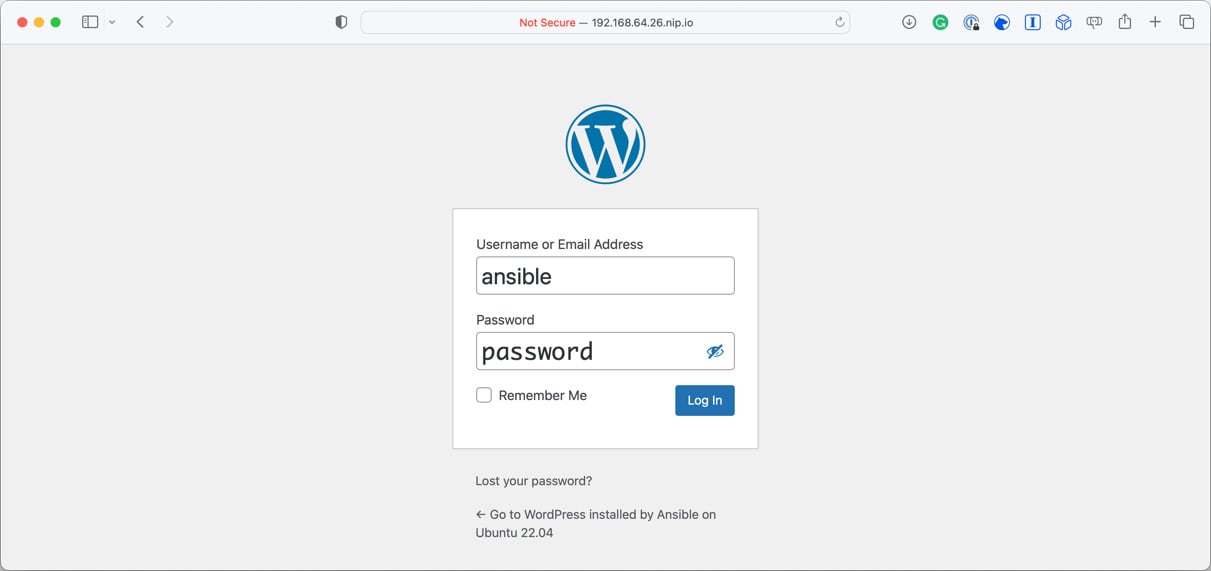


Figure 5.2 – The WordPress admin login page

Once logged in, you should see a few messages about the plugins we installed during the playbook run needing to be configured:

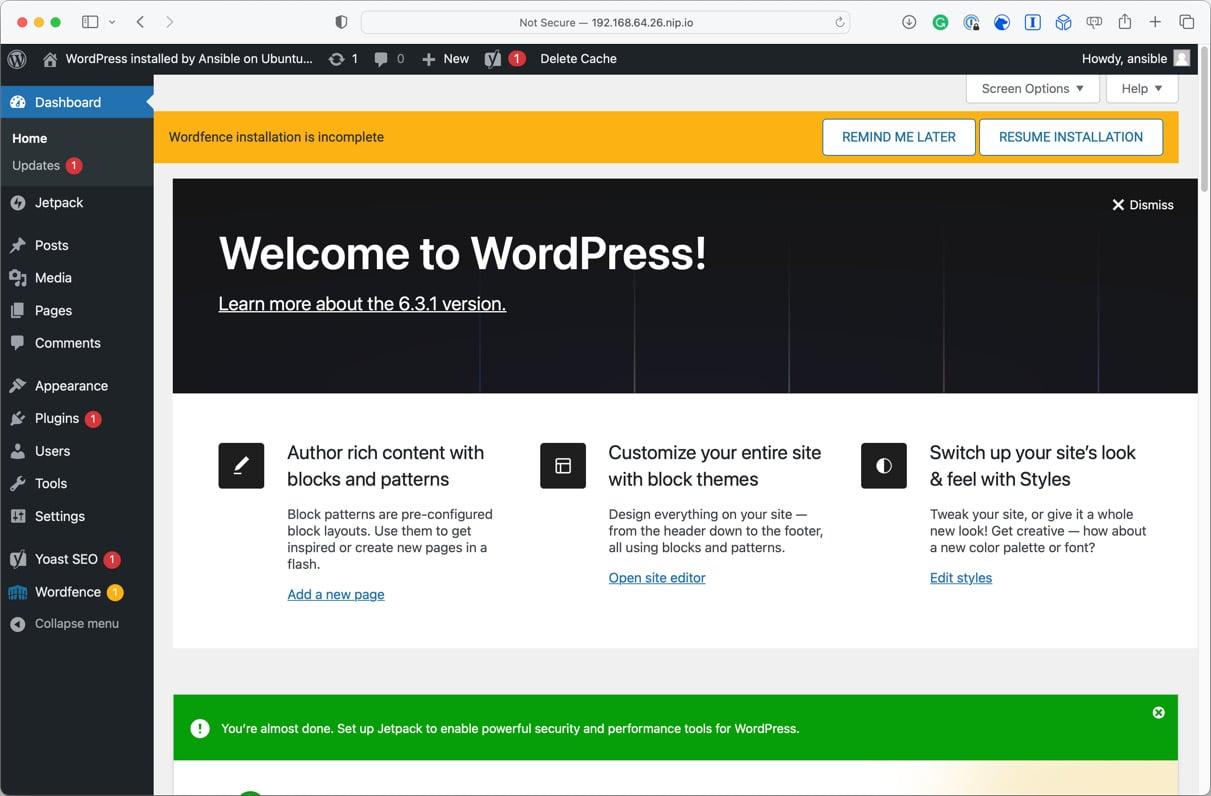


Figure 5.3 – Prompts when first logging into WordPress

Feel free to play with the WordPress installation and even, if you are so inclined, try and break it – if you needed to, you could delete and relaunch the Multipass virtual machine and quickly rerun the playbook to reinstall WordPress.

Summary

In this chapter, we have reused many of the same principles we covered in the previous chapter and moved on to deploying a complete application. What is good about this is that the process is both repeatable and just a single command.

So far, we have been targeting an Ubuntu virtual machine. If we ran our playbook against a Red-Hat-based virtual machine, the playbook would give an error as commands and paths are different.

The next chapter will target multiple operating systems using the same playbook.