```
vars_prompt:
 - name: username
   prompt: What is your name?

tasks:
- name: Ensure libselinux-python installed
  yum: name=libselinux-python state=present

- name: Ensure Apache installed
  yum: name=httpd state=present
  when: ansible_os_family == "RedHat"

- name: Creates directoy
  file: path=/var/www/html/ansible state=directory

- name: Ensure Apache installed
  yum: name=httpd  state=present
  when: ansible_os_family == "RedHat"

- name: Ensure Apache is running
  service: name=httpd enabled=yes state=started


- name: Deploy configuration File
  template: src=templates/httpd.j2 dest=/etc/httpd/conf/httpd.conf
  notify:
    - Restart Apache


- name: Copy Site Files
  template: src=templates/index.j2 dest={{doc_root}}/index.html mode=0644


- name: Stop IPTABLES Now!!
  service: name=iptables state=stopped
```

```
handlers:
   - name: Restart Apache
     service: name=httpd state=restarted
```

25. Roles

While it's possible to write a playbook in one very large file eventually you'll want to reuse files and start to organize things.

In above example, we have seen in our playbook we have variables, tasks, handlers and templates.

This can slowly grow and will become eventually difficult to read and manage.

Roles is a directory structure where we distribute the content of our main playbook into proper directory structure.

Roles directory structure.

```
├── roles
│   └── apache
│       ├── handlers
│       │   └── main.yml
│       ├── tasks
│       │   └── main.yml
│       ├── templates
│       │   ├── httpd.j2
│       │   └── index.j2
│       └── vars
│           └── main.yml
```

This is what they are all for:

•**files**: This directory contains regular files that need to be transferred to the hosts you are configuring for this role. This may also include script files to run.

•**handlers**: All handlers that were in your playbook previously can now be added into this directory.

•**meta**: This directory can contain files that establish role dependencies. You can list roles that must be applied before the current role can work correctly.

•**templates**: You can place all files that use variables to substitute information during creation in this directory.

•**tasks**: This directory contains all of the tasks that would normally be in a playbook. These can reference files and templates contained in their respective directories without using a path.

•**vars**: Variables for the roles can be specified in this directory and used in your configuration files.

Instead of having all our code in one playbook mashed up, we can distribute it into different directory structure. For example, all the tasks from our playbook will got into roles/apache/tasks/main.yml file, likewise vars go into vars/main.yml.

We can create the roles directory structure with ansible-galaxy command.

```
$ mkdir roles
$ cd roles/
imran@DevOps:.../roles$ ansible-galaxy init apache
- apache was created successfully
imran@DevOps:.../roles$ tree
.
└── apache
    ├── defaults
    │   └── main.yml
    ├── handlers
    │   └── main.yml
    ├── meta
    │   └── main.yml
    ├── README.md
    ├── tasks
    │   └── main.yml
    ├── tests
    │   ├── inventory
    │   └── test.yml
    └── vars
        └── main.yml
$ cd ..
```

Our main playbook will just call the role by its name and will not have any tasks.

```
$ cat webservers.yml

---

- hosts: webservers

  sudo: yes

  gather_facts: no

  roles:

  - apache
```

```
$ cat roles/apache/tasks/main.yml

---

- name: Ensure Apache installed

  yum: name=httpd state=present


- name: Creates directoy

  file: path=/var/www/html/ansible state=directory


- name: Ensure libselinux-python installed

  yum: name=libselinux-python  state=present


- name: Ensure Apache is running

  service: name=httpd enabled=yes state=started



- name: Deploy configuration File

  template: src=templates/httpd.j2 dest=/etc/httpd/conf/httpd.conf

  notify:

    - Restart Apache



- name: Copy Site Files

  template: src=templates/index.j2 dest={{doc_root}}/index.html mode=0644


- name: Stop IPTABLES Now!!

  service: name=iptables state=stopped
```

```
$ cat roles/apache/handlers/main.yml

---

- name: Restart Apache
  service: name=httpd state=restarted
```

```
$ cat roles/apache/vars/main.yml

http_port: 80

doc_dir: /ansible/

doc_root: /var/www/html/ansible/

max_clients: 5

ansible_python_interpreter: python

username: devops
```

```
$ ls roles/apache/templates/

httpd.j2  index.j2
```

It would be a good idea to first write a mashed-up playbook then we can start copying the content of the main playbook to the roles directory structure.

As the complexity grow we can then manage with the roles directory structure.

Executing Playbook.

We execute the main playbook which in turn will read the roles directory structure and execute all the tasks & handlers for us.

```
$ ansible-playbook  webservers.yml
```

Overriding variables of roles.

We have few variables defined in our apache roles in roles/apache/**vars**/main.yml file.

When we execute our playbook it uses all these defined variables but we can override these variables without changing the content of roles/apache/vars/main.yml file.

```
---
```

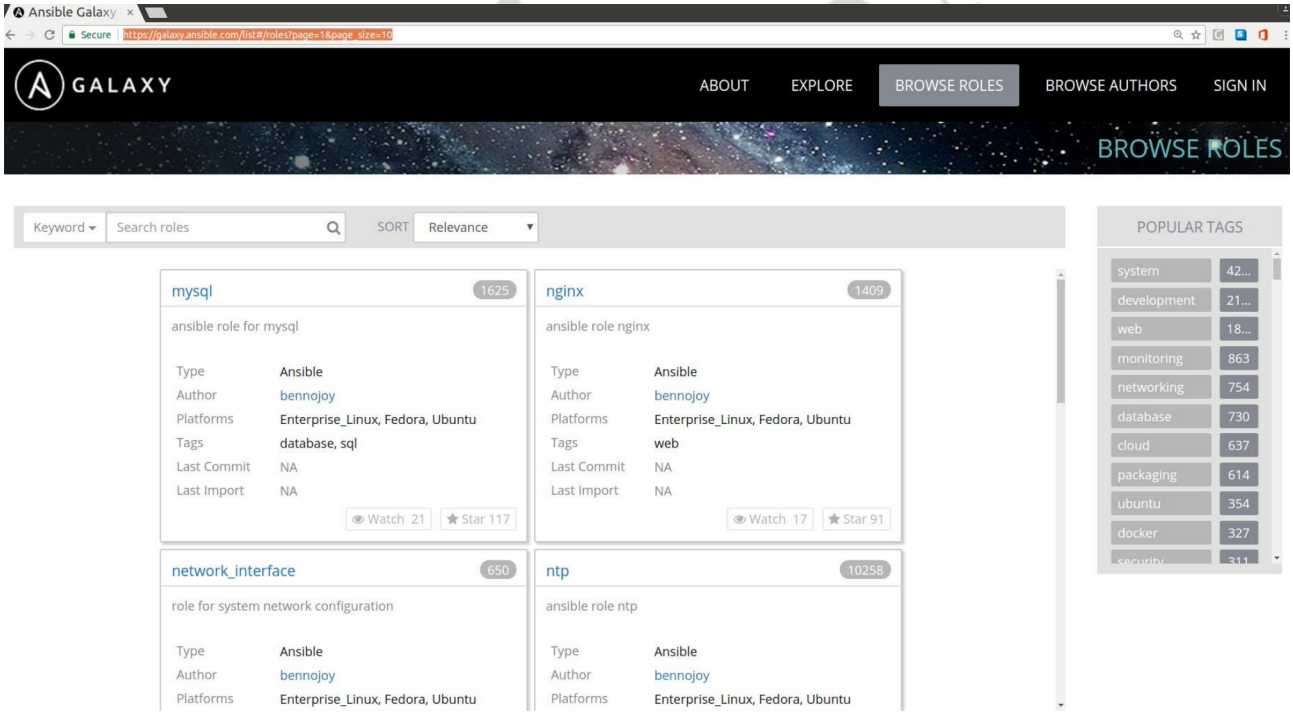
```
- hosts: webservers
  sudo: yes
  gather_facts: no
  roles:
  - {role:apache, http_port:8090, max_clients:250}
```

As shown above we are passing a dictionary now {} instead of the role name. Dictionary has key=value pairs for role name and variables that we want to override.

26. Ansible Galaxy

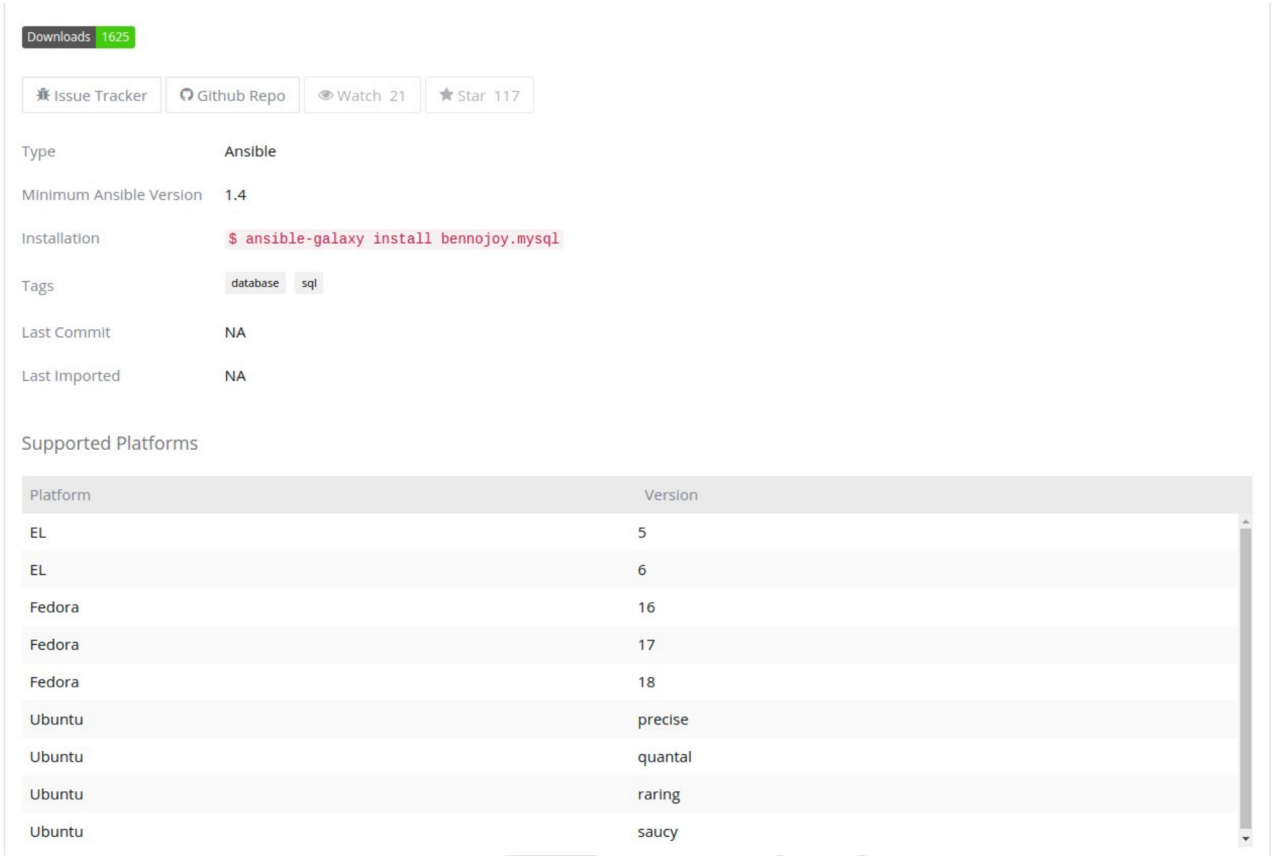
So far, we have got the idea of what ansible roles are and how to create them. There are so many predefined roles in Ansible Galaxy website which we can download and use freely. For example, if we want to setup mysql service, we can create mysql role from scratch or we can use some existing role from ansible galaxy.

https://galaxy.ansible.com/list#/roles?page=1&page_size=10



We can search and find a relevant role for our work. Role gets reviewed and gets star marks.

Click on the role to read about it in detail. Check the supported platforms and ansible version which it supports.



README section talks about how to use this role with examples.

Download the ansible galaxy role.

Role by default gets downloaded into /etc/ansible/roles directory. Once its downloaded you can start using this role from your playbook. Refer to the example given in the README section of the role.

```
$ sudo ansible-galaxy install bennojoy.mysql
- downloading role 'mysql', owned by bennojoy
- downloading role from https://github.com/bennojoy/mysql/archive/master.tar.gz
- extracting bennojoy.mysql to /etc/ansible/roles/bennojoy.mysql
- bennojoy.mysql (master) was installed successfully
```

27. Ansible Vault
Managing secrets with Ansible vault.

The vault feature can encrypt any structured data file used by Ansible. This can include "group_vars/" or "host_vars/" inventory variables, variables loaded by "include_vars" or "vars_files", or variable files passed on the ansible-playbook command line with "-e @file.yml" or "-e @file.json". Role variables and defaults are also included!

We can store our passwords/secrets encrypted in the ansible vault.

We will create vault in group_vars/all directory.

```
mkdir -p group_vars/all
cd group_vars/all
export EDITOR=vim
ansible-vault create vault
```

Give a vault password and put variables that you want to encrypt, below mentioned is the example of storing tomcat password.

```
---
vault_tomcat_pass : <Enter password for tomcat here>
```

Refer to vault_tomcat_pass in group_vars/all/vars file

```
cd group_vars/all/
vi vars
---
tomcatuser: tomcat
tomcatpass: "{{vault_tomcat_pass}}"
```

If you execute your playbook now where you are using tomcatpass, you should get some error like below. For example, I am using {{tomcatpass}} variable in my tomcat.yml playbook.

```
# ansible-playbook tomcat.yml

ERROR! Decryption failed
ERROR! A vault password must be specified
```

You can give --ask-vault-pass option while executing playbook which will ask you the vault password.

```
# ansible-playbook tomcat.yml --ask-vault-pass
```

You can also use a file where you specify vault password.

Lets say “vaultpass” is our vault password.

```
# echo "vaultpass" > ~/.vault_pass.txt
# chmod 0600 ~/.vault_pass.txt
```

Sample Execution.

```
imran@DevOps:.../ans$ tree
.
├── group_vars
│   └── all
│       ├── vars
│       └── vault
└── test.yml

2 directories, 3 files


imran@DevOps:.../ans$ cat group_vars/all/vars
---
tomcatpassword: "{{tomcatpass}}"


imran@DevOps:.../ans$ cat test.yml
---
- hosts: localhost
tasks:
- name: Print tomcat pass
debug: msg="{{tomcatpassword}}"

imran@DevOps:.../ans$ ansible-playbook test.yml --ask-vault-pass
Vault password:
[WARNING]: provided hosts list is empty, only localhost is available

PLAY ***************************************************************************

TASK [setup] *******************************************************************
ok: [localhost]

TASK [Print tomcat pass] *******************************************************
ok: [localhost] => {
"msg": "tomcat"
}

PLAY RECAP *********************************************************************
localhost : ok=2 changed=0 unreachable=0 failed=0
```

Editing Vault

To edit an encrypted file in place, use the *ansible-vault edit* command. This command will decrypt the file to a temporary file and allow you to edit the file, saving it back when done and removing the temporary file:

```
ansible-vault edit vault
```

28. Looping in ansible

To save some typing, repeated tasks can be written in short-hand like so:

with_items is a ansible “for loop” which runs on a list of items. with_items while executing return variable named “item” which hold the value of element in the list

```
- name: add several users
  user:
    name: "{{ item }}"
    state: present
    groups: "wheel"
  with_items:
     - testuser1
     - testuser2
```

If you have defined a YAML list in a variables file, or the ‘vars’ section, you can also do:

with_items: "{{ somelist }}"

The above would be the equivalent of:

```
- name: add user testuser1
  user:
    name: "testuser1"
    state: present
    groups: "wheel"
- name: add user testuser2
  user:
    name: "testuser2"
    state: present
    groups: "wheel"
```

The yum and apt modules use with_items to execute fewer package manager transactions.

```
# tasks file for mysql

- name: Install Mysql package

  yum:

'   name: "{{ item }}"

    state: installed

  with_items:

   - mysql-server

   - MySQL-python

   - libselinux-python

   - libsemanage-python
```

29. Configure Apache Using Ansible

Introduction

Apache is one of the most popular web servers currently used on the Internet. It is easy to set up and configure on Linux distributions like Ubuntu and Debian, as it comes in the package repositories and includes a default configuration that works out of the box.

Prerequisites

We will install Ansible on a Ubuntu server and use it to configure Apache on a second server.

For this tutorial, you will need:

Two Ubuntu servers: one master server with Ansible and one secondary which will run Apache configured through Ansible

Ansible installed on the master server.

SSH key exchange to authorize master to login to secondary server. Refer Bash scripting chapter for SSH key exchange.

Manually set up a local hosts file on your local machine (using your secondary server's IP address), in order to set up and use the Virtual Hosts that will be configured.

Configuring Ansible

Create a new directory.

```
$ mkdir ansible-repo
```

Move into the new directory.

```
$ cd ~/ansible-repo/
```

Create a new file called ansible.cfg and open it for editing.

```
$ vi ansible.cfg
```

Within that file, we want to add in the hostfile configuration option with the value of hosts, within the[defaults] group. Copy the following into the ansible.cfg file, then save and close it.

```
[defaults]

inventory = hosts
```

Create a hosts file(inventory) and open it for editing.

```
$ vi hosts
```

Copy the following into the hosts file.

```
[apache]

apache_server_ip ansible_ssh_user=username
```

This specifies a host group called apache which contains one host. Replace apache_server_ip with the secondary server's hostname or IP address, and username with your SSH username. Now Ansible should be able to connect to your server.

Test the connectivity with ping module.

```
$ ansible apache -m ping
```

The output should look like this:

```
192.168.1.51 | success >> {

    "changed": false,
```

```
    "ping": "pong"
}
```

Another Ansible module that is useful for testing is the command module. It runs custom commands on the host and returns the results. To run the command using echo, a Unix command that echoes a string to the terminal, enter the following command.

```
$ ansible apache -m command -a "/bin/echo hello devops"
```

The output should look like this:

```
192.168.1.51 | success | rc=0 >>


hello devops
```

Installing Apache

We will write the task to install the Apache web server in the playbook.

To install apache via Ansible, we use Ansible's apt module. The apt module contains many options for specialised apt-get functionality. The options we are interested in are:

- name: The name of the package to be installed, either a single package name or a list of packages.
- state: Accepts either latest, absent, or present. Latest ensures the latest version is installed, present simply checks it is installed, and absent removes it if it is installed.
- update_cache: Updates the cache (via apt-get update) if enabled, to ensure it is up to date.

Now let's create our apache.yml playbook with the apt module. Open up the apache.yml file for editing.

```
$ vi apache.yml
```

Copy the following text into it.

```
---
- hosts: apache
  become: yes
  tasks:
    - name: install apache2
```

```
    apt: name=apache2 update_cache=yes state=latest
```

The apt line installs the apache2 package (name=apache2) and ensures we have updated the cache (update_cache=yes).

Now run the playbook.

```
$' ansible-playbook apache.yml --ask-sudo-pass
```

The --ask-sudo-pass flag will prompt you for the sudo password on the secondary server. This is necessary because the installation requires root privileges; the other commands we've run so far did not.

The output should look like this.

```
PLAY [apache] *****************************************************************

GATHERING FACTS ***************************************************************
ok: [192.168.1.51]

TASK: [install apache2] *******************************************************
changed: [192.168.1.51]

PLAY RECAP ********************************************************************
192.168.1.51           : ok=2    changed=1    unreachable=0    failed=0
```

If you visit your secondary server's hostname or IP address in your browser, you should now get a Apache2 Ubuntu Default Page to greet you. This means you have a working Apache installation on your server, and you haven't manually connected to it to run a command yet.

Configuring Apache Modules

Now that Apache is installed, we need to enable a module to be used by Apache.

Let us make sure that the mod_rewrite module is enabled for Apache. Via SSH, this can be done easily by using a2enmod and restarting Apache. However, we can also do it very easily with Ansible using the apache2_module module and a task handler to restart apache2.

The apache2_module module takes two options:

- name -- The name of the module to enable, such as rewrite.
- state -- Either present or absent, depending on if the module needs to be enabled or disabled.

Open apache.yml for editing.

```
$ vi apache.yml
```

Update the file to include this task. The file should now look like this:

```
---
- hosts: apache
  sudo: yes
  tasks:
    - name: install apache2
      apt: name=apache2 update_cache=yes state=latest


    - name: enabled mod_rewrite
                apache2_module: name=rewrite state=present
```

We need to restart apache2 after the module is enabled. One option is to add in a task to restart apache2, but we don't want that to run every time we apply our playbook. To get around this, we need to use a task handler. The way handlers work is that a task can be told to notify a handler when it has changed, and the handler only runs when the task has been changed.

To do this we need to add the notify option into the apache2_module task, and then we can use the service module to restart apache2 in a handler.

```
---
- hosts: apache
  sudo: yes
  tasks:
    - name: install apache2
      apt: name=apache2 update_cache=yes state=latest


    - name: enabled mod_rewrite
      apache2_module: name=rewrite state=present
      notify:
        - restart apache2


  handlers:
    - name: restart apache2
      service: name=apache2 state=restarted
```

Now, rerun the playbook.

```
$ ansible-playbook apache.yml --ask-sudo-pass
```

The output should look like

```
PLAY [apache] *****************************************************************

GATHERING FACTS ***************************************************************
ok: [192.168.1.51]

TASK: [install apache2] *******************************************************
ok: [192.168.1.51]

TASK: [enabled mod_rewrite] ***************************************************
changed: [192.168.1.51]

NOTIFIED: [restart apache2] ***************************************************
changed: [192.168.1.51]

PLAY RECAP ********************************************************************
192.168.1.51               : ok=4    changed=2    unreachable=0    failed=0
```

It looks good so far. Now, run the command again and there should be no changes, and the restart task won't be listed.

Configuring Apache Options

Now that we have a working Apache installation, with our required modules turned on, we need to configure Apache.

By default, Apache listens on port 80 for all HTTP traffic. For the sake of the tutorial, let us assume that we want Apache to listen on port 9090 instead. With the default Apache configuration on Ubuntu x64, there are two files that need to be updated:

```
/etc/apache2/ports.conf
    Listen 80


/etc/apache2/sites-available/000-default.conf
    <VirtualHost *:80>
```

To do this, we can use the lineinfile module. It allows you to perform all sorts of changes to an existing file on the host. For this example, we will use the following options:

dest -- The file to be updated as part of the command.
regexp -- Regular Expression to be used to match an existing line to be replaced.
line -- The line to be inserted into the file, either replacing the regexp line or as a new line on the end.
state -- Either present or absent.

What we need to do to update the port from 80 to 9090 is look for the existing lines which define port 80, and change them to define port 9090.

Open the apache.yml file for editing.

```
$ vi apache.yml
```

```
---
- hosts: apache
  sudo: yes
  tasks:
    - name: install apache2
      apt: name=apache2 update_cache=yes state=latest

    - name: enabled mod_rewrite
      apache2_module: name=rewrite state=present
      notify:
        - restart apache2

    - name: apache2 listen on port 9090
      lineinfile: dest=/etc/apache2/ports.conf regexp="^Listen 80" line="Listen 9090" state=present
      notify:
        - restart apache2

    - name: apache2 virtualhost on port 9090
      lineinfile: dest=/etc/apache2/sites-available/000-default.conf regexp="^<VirtualHost \*:80>" line="<VirtualHost *:9090>" state=present
      notify:
        - restart apache2

  handlers:
```

```
    - name: restart apache2
      service: name=apache2 state=restarted
```

It is important to notice that we also need to restart apache2 as part of this process, and that we can re-use the same handler but the handler will only be triggered once despite multiple changed tasks.

Now run the playbook.

```
$ ansible-playbook apache.yml --ask-sudo-pass
```

Once Ansible has finished, you should be able to visit your host in your browser and it will respond on port 9090, rather than port 80. In most web browsers, this can be easily achieved by adding: port onto the end of the URL: http://192.168.1.51:9090/.

Configuring Virtual Hosts

In this section, we will use the template module to configure a new virtual host on your server.

Create Virtual Host Configuration

The first step is to create a new virtual host configuration. We'll create the virtual host configuration file on the master server and upload it to the secondary server using Ansible.

Here's an example of a basic virtual host configuration which we can use as a starting point for our own configuration. Notice that both the port number and the domain name, highlighted below, are hardcoded into the configuration.

```
<VirtualHost *:9090>
    ServerAdmin webmaster@keplar.com
    ServerName keplar.com
    ServerAlias www.keplar.com
    DocumentRoot /var/www/keplar.com
    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

Create a new file called virtualhost.conf.

```
$ vi virtualhost.conf
```

Paste the following into virtualhost.conf. Because we are using templates, it is a good idea to change the hard-coded values above to variables, to make them easy to change in the future.

```
<VirtualHost *:{{ http_port }}>
    ServerAdmin webmaster@{{ domain }}
    ServerName {{ domain }}
    ServerAlias www.{{ domain }}
    DocumentRoot /var/www/{{ domain }}
    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

Use Template Variables

Next, we need to update our playbook to push out the template and use the variables.

The first step is to add in a section into the playbook for variables. We need to put in both variables used in the template above, and we will change the port back to 80 in the process.

```
---
- hosts: apache
  sudo: yes
  vars:
    http_port: 80
    domain: keplar.com
  tasks:
    - name: install apache2
...
```

Variables can be used in tasks and templates, so we can update our existing lineinfile modules to use the specified http_port, rather than the hard coded 9090 we specified before. The variable needs to be added into the line, and the regexp option needs to be updated so it's not looking for a specific port. The changes will look like this:

```
lineinfile: dest=/etc/apache2/ports.conf regexp="^Listen " line="Listen {{
http_port }}" state=present

lineinfile: dest=/etc/apache2/sites-available/000-default.conf
regexp="^<VirtualHost \*:" line="<VirtualHost *:{{ http_port }}>"
```

Add Template Module

The next step is to add in the template module to push the configuration file onto the host. We will use these options to make it happen:

32. dest -- The destination file path to save the updated template on the host(s), i.e./etc/apache2/sites-available/{{ domain }}.conf.
33. src -- The source template file, i.e. virtualhost.conf.

Applying these to your playbook will result in a task that looks like this:

```
- name: create virtual host file

  template: src=virtualhost.conf dest=/etc/apache2/sites-available/{{ domain
}}.conf
```

Enable the Virtual Host

What we need to do now is enable the virtual host within Apache. This can be done in two ways: by running the sudo a2ensite keplar.com command or manually symlinking the config file into /etc/apache2/sites-enabled/. The former option is safer, as it allows Apache to control the process. For this, the command module comes in use again.

The usage is quite simple, as we discovered above:

```
- name: a2ensite {{ domain }}

  command: a2ensite {{ domain }}

  notify:

  - restart apache2
```

Prevent Extra Work

Finally, the command module needs to know when it should and shouldn't run, so the module is not run unnecessarily if the playbook is run multiple times. In our case, it only needs to be run if the .conf file hasn't been created on the host yet.

This is done using the creates option, which allows you to tell the module what file is being created during the module execution. If the file exists, the module won't run. Because Apache creates a symlink when sites are enabled, checking for that solves the problem.

The changes will look like this:

```
- name: a2ensite {{ domain }}

  command: a2ensite {{ domain }}

  args:

    creates: /etc/apache2/sites-enabled/{{ domain }}.conf

  notify:

  - restart apache2
```