Learning Objectives

Learners will be able to…

- **Create files by copying**
- **Make use of Ansible’s template functionality**
- **Set file permissions**

Creating a File

Ansible and File Management

Most of the configurations in Linux systems are stored as files, and managing files is one of the most basic tasks Ansible performs.

> growthhack
>
> It's best practice to configure the system by defining the **desired state**; by updating a system's configuration files it gains resiliency to reboot issues and can be snapshotted to be used as a golden image.

▼ **Specify Desired State (Don't Program Behavior)**

Defining the desired state has many benefits, among them is setting the system configuration to be fully working even if isolated, allowing the system to reboot without provisioning again.

The practice of defining the desired state is particularly important to note for software developers transitioning into DevOps roles. Developers sometimes take the algorithmic approach of "programming behavior" of systems, configuring Ansible to run on boot to start, stop, and update files. This is considered bad practice; if a conflict arises, such as a process failing or not having enough memory, the configuration will 'die' and the system will end up not being fully configured.

As an example, if we want nginx to be enabled when managing services, best practice would be to enable true so that it starts automatically on system boot. Alternatively, it would be problematic to instead call Ansible to run scripts that will trigger nginx to be true.

With Ansible you're not writing an algorithm to do something step by step - you're simply specifying a state of the system.

The File Module

Ansible can perform many tasks with its large collection of built-in **modules** (the full list can be found here).

definition

Modules in Ansible

A **module** is a pre-written piece of Python code that performs a specific action on a target host. They are used in playbooks and tasks to perform a variety of operations: installing packages, creating users, modifying configuration files, and much more.

Files and directory management are handled by the builtin file module, `ansible.builtin.file`, which handles:
* Creating an empty file
* Managing directories
* Managing permissions
* Managing symlinks in the system
* Managing access or create time of the file
* Deleting files and directories

Next, we'll use the builtin file module to create a file.

The State Parameter: Creating a file

The `state` defines the operation performed on a file or directory.

Let's practice creating a file using the `cases/create-file.yml` playbook already written for you. The task will utilize the `ansible.builtin.file` module and is appropriately named "Create Test File":

```
- hosts: localhost
  tasks:
  - name: Create test file
    ansible.builtin.file:
      path: /home/codio/workspace/test_file
      state: touch
```

The `/home/codio/workspace/test_file` path will place the file in the file tree to the left. Specifying `state: touch` will create a new, empty file. Let's run our playbook using the following command in the terminal:

```
ansible-playbook cases/create-file.yml
```

You should now see the empty `test_file` file in the `workspace directory`.

The State Parameter: Removing a file

If `state` is set to `absent` the resource will be removed.

Change the `state` parameter in our `create-file.yml` playbook so that it specifies `state: absent` and run the playbook again:

```
ansible-playbook cases/create-file.yml
```

Our newly created `test_file` file should now be removed from our workspace.

Manage Directories and Symlinks

The File Module (Continued)

In addition to managing files, the `ansible.builtin.file` module can be used to manage directories by setting `state` to `directory`.

On the left, you'll see the `directory.yml` playbook that we'll be using to create a directory.

```
- hosts: localhost
  tasks:
  - name: Create Test directory
    ansible.builtin.file:
      path: /home/codio/workspace/test/directory
      recurse: true
      state: directory
```

Go ahead and run the following command in the terminal to run the playbook:

```
ansible-playbook cases/directory.yml
```

Opening the `test` folder in the file tree, you should now see the empty `directory`.

> important
>
> It's important to note that to create all parent directories, `recurse: true` should be used; otherwise Ansible will fail if parent folder does not already exist.

Symlinks

Similar to creating files and directories `ansible.builtin.file` can be used to create symlinks by using `state: link`. Here are the contents of a simple playbook `symlink.yml` that creates a symlink:

```
- name: Create Test Link
  ansible.builtin.file:
    src: /home/codio/workspace/test
    dest: /home/codio/workspace/test_link
    state: link
```

Try running it in the terminal (the file path is `cases/symlink.yml`)

Copy a File From Resources

The Copy Module

If you have fully defined file content prepared for a remote host, such as a configuration file that you want to upload to a server, the file can simply be copied from your control node using your Ansible scripts to a managed node using the builtin copy module, `ansible.builtin.copy`.

The copy module has many parameters (click here for the full list); in the example below we will be focusing on the most applicable: the source `src`, destination `dest`, and `content` parameters.

```
- name: Copy index.html to the dest folder
  ansible.builtin.copy:
    src: files/index.html
    dest: /home/codio/workspace/dest/index.html
```

- `src` specifies the path to the file we want to copy
- `dest` describes the remote absolute path for the copied file (in this case, the `dest` folder in our workspace)

---

growthhack

It's good practice to store all files to be copied in the `files` directory. This helps organize your playbooks by keeping these files separate from templates, etc.

The Content Parameter

Instead of copying an entire file from a local source onto managed nodes, it is possible to use the `content` parameter to define a file's contents, within the playbook, to be copied, without creating a separate file.

In our example, we are copying the `index.html` file, which is a small file containing a string with html header syntax. This makes it a great example for use of the `content` parameter instead of the `src` parameter.

```
    - name: Copy index.html to the dest folder
      ansible.builtin.copy:
        content: "<h1> Hello, World</h1>"
        dest: /home/codio/workspace/dest/index.html
```

- `content` specifies the contents of the file created on the remote host.
- `dest` describes the location of where to create the file if it doesn't already exist.

▼ **The Content Parameter: Additional Information**

The `Hello, World` example used above is a good use case of implementing the `content` parameter because the text is brief and contained to a single line.

If you are using the copy module with the content parameter, it's important to carefully format any multi-line strings that may be included. You may need to review Literal and Formal Block Scalar formatting (described in the first module of this course).

However, if multi-line strings are being used, your playbook will quickly become bloated and difficult to read. Therefore we suggest avoiding using the `content` parameter in your playbooks and copy files located in the control node instead.

Copying a File

Let's run our playbook to see the Copy Module in action. Copy either of the code blocks above and add the code to our `copy-file.yml` playbook.

Let's run our playbook using the following command in the terminal:

```
ansible-playbook cases/copy-file.yml
```

▼ **Check your indentation**

Your playbook indentation should appear as it does below, regardless of the code block you used.

```
- hosts: localhost
  tasks:
    - name: Copy index.html to the dest folder
      ansible.builtin.copy:
        content: "<h1> Hello, World</h1>"
        dest: /home/codio/workspace/dest/index.html
```

Open the `dest` folder in the file tree to make sure you successfully copied the `index.html` file.

Create a File From a Template

The Template Module

Unless it's a simple use case, copying files is insufficient, as we need to vary file contents (i.p. address, hostname, version) based on the system a playbook is applied to. Ansible uses the powerful `jinja2` templating engine through the builtin module `ansible.builtin.template`. Similar to the copy module, the `src` and `dest` parameters are used.

```
    - name: Copy index.html to the dest folder
      ansible.builtin.template:
        src: templates/index.html.j2
        dest: /home/codio/workspace/dest/index.html
```

- `src` is the path of the Jinja2 formatted template
- `dest` specifies the remote location where the template will be rendered

---

growthhack

To keep playbooks organized, it's good practice to store all templates in the `templates` directory.

The Validate Parameter

Another important parameter used with the template module is `validate`, which executes a command before the file is copied to the system to check that the generated configuration is correct. Here's an example:

```
- name: Update sshd configuration safely, avoid locking yourself
out
  ansible.builtin.template:
    src: etc/ssh/sshd_config.j2
    dest: /etc/ssh/sshd_config
    owner: root
    group: root
    mode: '0600'
    validate: /usr/sbin/sshd -t -f %s
    backup: yes
```

`Validate` is particularly important for services like Apache, NGIX, databases, and sshd; if something goes wrong with the ssh configuration, you will be unable to connect to that instance. To avoid this we use `validate` to ensure the configuration is still valid, where `%s` will be replaced with the file name Ansible prepares via our template.

The Backup Parameter

The `backup` parameter simply determines if a backup will be created or not, and the remaining parameters used in the example above, which set file ownership, are covered on the following page.

> topic
>
> A full list of parameters and attributes used by the template module can be found here.

Set File Ownership

Parameters: Owner, Group, and Mode

The `ansible.builtin.template`, `ansible.builtin.file`, and `ansible.builtin.copy` modules support setting ownership and file permissions

```
    owner: bin
    group: wheel
    mode: '0644'
```

- `owner` defines name of the user to set as owner of the file
- `group` defines group name to set as owner of the file
- `mode` defines file permissions

If `owner` or `group` is not defined then current user permissions will be used. However, if the current user is `root` then the previous permission and ownership will be used

The Mode Parameter

`mode` defines umask for the files. For these permissions, Ansible parses octal numbers or strings: `0644` format can be used with a leading 0 to indicate an octal number or using quotes to indicate `'644'` as string to let Ansible convert it to a number.

The mode may be specified as a symbolic modes: `u+rwx` or `u=rw,g=r,o=r`.

> The full documentation on permissions can be found here.

Replace/Add/Delete Content in Existing Files

The Replace Module

If files need to be modified, the `ansible.builtin.replace` module can be used.

```
- name: Replace between the expressions
  ansible.builtin.replace:
    path: /file/path
    after: 'start'
    before: 'end'
    regexp: 'foo'
    replace: bar
```

- `before` and `after` (both python regexp fields) can be used to specify the limits of where to search for the text.
- `regexp` is the expression being searched for in the file
- `replace` is the replacement where backreferences can be defined as `\1`. If `replace` is not defined the matches will be removed entirely.

Permissions (`owner`, `group`, and `mode`) and `validate` parameters for `ansible.builtin.replace` are similar to the `ansible.builtin.template` module.

Handlers

Handlers

After a file has changed we sometimes need to restart/reload a service to apply the new configuration. If a single task is being implemented that we know will require a restart, we can add an additional task that is triggered to restart the service upon noticing the change. However, if we are running multiple tasks we likely need to avoid a restart on every update made, or avoid restarting the service if it's unnecessary.

Handlers are used to manage these situations by differentiating between services that need to be restarted, and performing the restart at the appropriate time. Ansible has a built-in `notify` mechanism to execute handlers so that they only run if notified.

For example, let's imagine we are updating sshd and we make several updates to different files. We use validate to ensure the new configurations have not broken the file. As the tasks in our playbook run, they may be written so that they `notify` the handlers that a restart is required. Once the tasks are completed, the handlers run sequentially and restart the sshd if required.

> important
>
> Defined separately from tasks, handlers execute only after all tasks have been completed.

Handlers are executed sequentially, in the order they are defined in the handlers section, not in the order listed in the notify statement. Notifying the same handler multiple times will result in executing the handler only once, regardless of how many tasks notify it. For example, if multiple tasks update a configuration file and notify a handler to restart Apache, Ansible only restarts Apache once to avoid unnecessary restarts.

The Notify Keyword

The `notify` keyword can be applied to a task and accepts a list of handler names that are notified on a task change, as shown in the example below

```
tasks:
  - name: Write the apache config file
      ansible.builtin.copy:
        src: /srv/httpd.j2
        dest: /etc/httpd.conf
      notify:
      - Restart apache

handlers:
  - name: Restart apache
    ansible.builtin.service:
      name: httpd
      state: restarted
```

The above code block is part of a playbook that includes a task to copy an Apache configuration file and a handler to restart the Apache service if the configuration file changes.

The task “Write the apache config file” uses the “copy” module to copy the source file `/srv/httpd.j2` to the destination `/etc/httpd.conf`. The `notify` keyword is used to trigger the handler `Restart apache` when this task completes.

The handler, defined separately in the playbook, uses the `service` module to manage the Apache service. It specifies the name of the service as `httpd` and sets its state to `restarted`, which will restart the service if it is running or start it if it is not.

In summary, if the Apache configuration file is changed during the playbook execution, the “Write the apache config file” task will complete, and the `notify` keyword will trigger the `Restart apache` handler to restart the Apache service with the updated configuration