Exercise 1.1 - Running Ad-hoc commands

For our first exercise, we are going to run some ad-hoc commands to help you get a feel for how Ansible works. Ansible Ad-Hoc commands enable you to perform tasks on remote nodes without having to write a playbook. They are very useful when you simply need to do one or two things quickly and often, to many remote nodes.

> Like many Linux commands, `ansible` allows for long-form options as well as short-form. For example:
>
> ```
> ansible web --module-name ping
> ```
>
> is the same as running
>
> ```
> ansible web -m ping
> ```
>
> We are going to be using the short-form options throughout this workshop

---

Step 1: Let's start with something really basic - pinging a host. The `ping` module makes sure our web hosts are responsive.

```
ansible web -m ping
```

Step 2: Now let's see how we can run a good ol' fashioned Linux command and format the output using the `command`module.

```
ansible web -m command -a "uptime" -o
```

Step 3: Take a look at your web node's configuration. The `setup` module displays ansible facts (and a lot of them) about an endpoint.

```
ansible web -m setup
```

Step 4: Now, let's install Apache using the `yum` module

```
ansible web -m yum -a "name=httpd state=present" -b
```

Step 5: OK, Apache is installed now so let's start it up using the `service` module

```
ansible web -m service -a "name=httpd state=started" -b
```

Step 6: Finally, let's clean up after ourselves. First, stop the httpd service

```
ansible web -m service -a "name=httpd state=stopped" -b
```

Step 7: Next, remove the Apache package

```
ansible web -m yum -a "name=httpd state=absent" -b
```

Exercise 1.2 - Writing Your First playbook

Now that you’ve gotten a sense of how ansible works, we are going to write our first ansible **playbook**. The playbook is where you can take some of those ad-hoc commands you just ran and put them into a repeatable set of **plays** and **tasks**.

A playbook can have multiple plays and a play can have one or multiple tasks. The goal of a **play** is to map a group of hosts. The goal of a **task** is to implement modules against those hosts.

For our first playbook, we are only going to write one play and two tasks.

Section 1 - Creating a Directory Structure and Files for your Playbook

There is a best practice on the preferred directory structures for playbooks. We strongly encourage you to read and understand these practices as you develop your Ansible ninja skills. That said, our playbook today is very basic and creating a complex structure will just confuse things.

Instead, we are going to create a very simple directory structure for our playbook, and add just a couple of files to it.

Step 1: Create a directory called **apache_basic** in your home directory and change directories into it

```
mkdir ~/apache_basic
 cd ~/apache_basic
```

Step 2: Define your inventory. Inventories are crucial to Ansible as they define remote machines on which you wish to run your playbook(s). Use `vi` or `vim` to create a file called `hosts`.`[web]`

```
node-1 ansible_host=<IP Address of your node-1>
node-2 ansible_host=<IP Address of your node-2>
```

Step 3: Use `vi` or `vim` to create a file called `install_apache.yml`

Section 2 - Defining Your Play

Now that you are editing `install_apache.yml`, let's begin by defining the play and then understanding what each line accomplishes

```
---
- hosts: web
  name: Install the apache web service
  become: yes
```

- `---` Defines the beginning of YAML
- `hosts: web` Defines the host group in your inventory on which this play will run against
- `name: Install the apache web service` This describes our play
- `become: yes` Enables user privilege escalation. The default is sudo, but su, pbrun, and several others are also supported.

Section 3: Adding Tasks to Your Play

Now that we've defined your play, let's add some tasks to get some things done. Align (vertically) the **t** in `task` with the **b**`become`.

Yes, it does actually matter. In fact, you should make sure all of your playbook statements are aligned in the way shown here.

If you want to see the entire playbook for reference, skip to the bottom of this exercise.

```
tasks:
 - name: install apache
   yum:
     name: httpd
     state: present

 - name: start httpd
   service:
     name: httpd
     state: started
```

- `tasks:` This denotes that one or more tasks are about to be defined
- `- name:` Each task requires a name which will print to standard output when you run your playbook. Therefore, give your tasks a name that is short, sweet, and to the point

---

```
yum:
  name: httpd
  state: present
```

- These three lines are calling the Ansible module **yum** to install httpd. Click here to see all options for the yum module.

```
service:
  name: httpd
  state: started
```

- The next few lines are using the ansible module **service** to start the httpd service. The service module is the preferred way of controlling services on remote hosts. Click here to learn more about the **service** module.

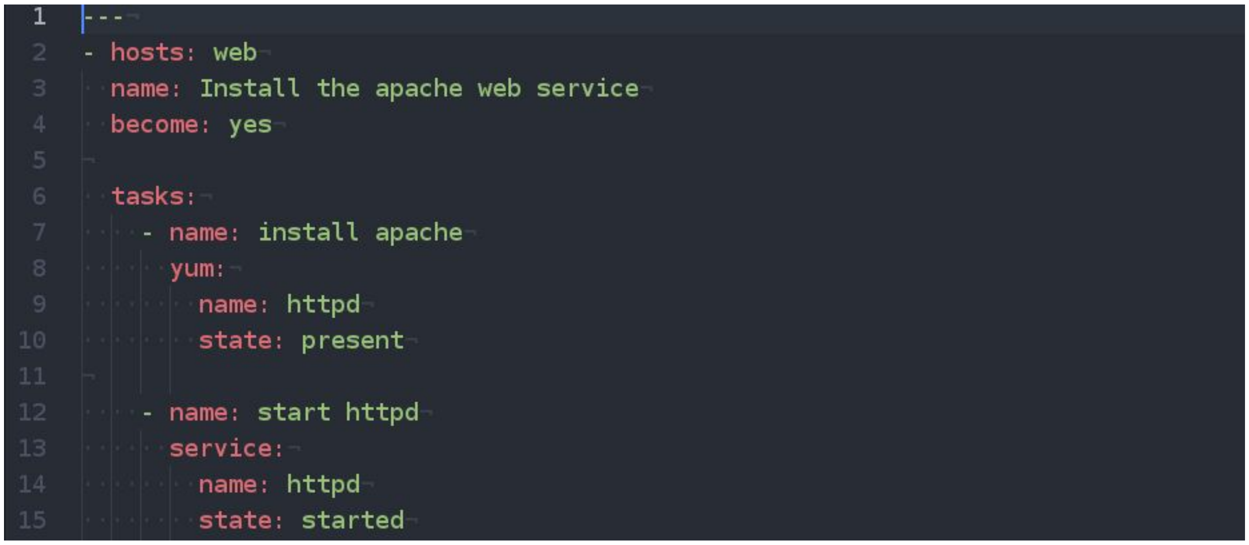
Section 4: Review

Now that you've completed writing your playbook, it would be a shame not to keep it.

Use the `write/quit` method in `vi` or `vim` to save your playbook, i.e. `Esc :wq!`

And that should do it. You should now have a fully written playbook called `install_apache.yml`. You are ready to automate!

> Ansible (well, YAML really) can be a bit particular about formatting especially around indentation/spacing. When you all get back to the office, read up on this YAML Syntax a bit more and it will save you some headaches later. In the meantime, your completed playbook should look like this. Take note of the spacing and alignment.



```yaml
---
- hosts: web
  name: Install the apache web service
  become: yes

  tasks:
    - name: install apache
      yum:
        name: httpd
        state: present

    - name: start httpd
      service:
        name: httpd
        state: started
```

Figure 1: Completed Playbook - w/Spacing

Exercise 1.3 - Running Your Playbook

We are now going to run your brand spankin' new playbook on your two web nodes. To do this, you are going to use the `ansible-playbook` command.

Step 1: From your playbook directory (`~/apache_basic`), run your playbook.

```
ansible-playbook -i ./hosts -k install_apache.yml
```

However, before you go ahead and run that command, lets take a few moments to understand the options.

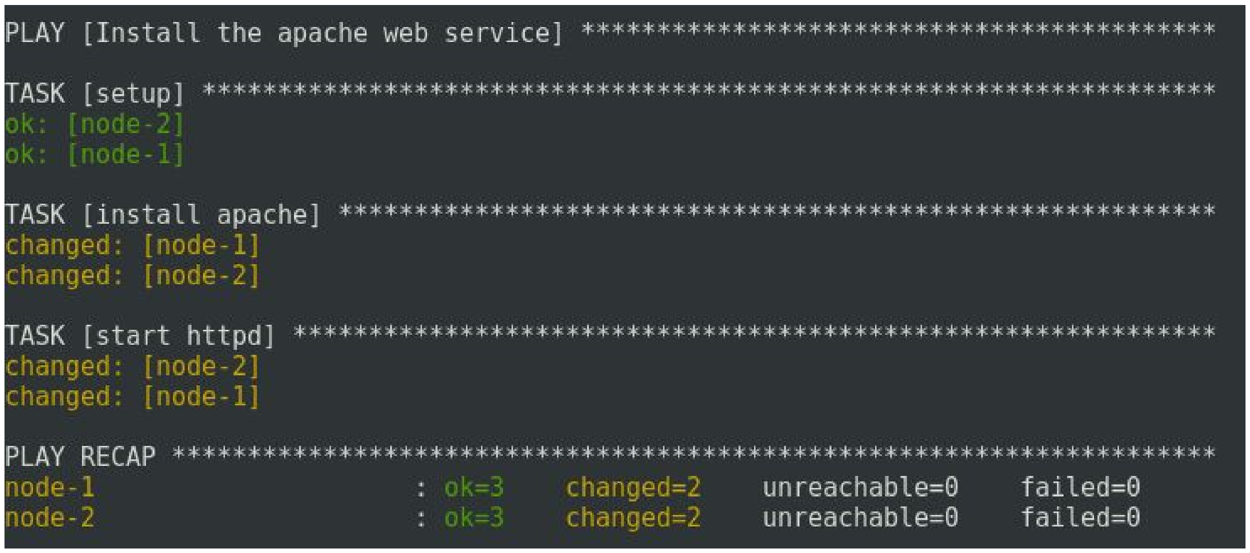
- **-i** This option allows you to specify the inventory file you wish to use.

- **-k** This option prompts you for the password of the user running the playbook.
- **-v** Although not used here, this increases verbosity. Try running your playbook a second time using `-v` or `-vv` to increase the verbosity
- **--syntax-check** If you run into any issues with your playbook running properly; you know, from that copy/pasting that you didn't do because we said "*don't do that*"; you could use this option to help find those issues like so…

```
ansible-playbook -i ./hosts -k install_apache.yml --syntax-check
```

OK, go ahead and run your playbook as specified in **Step 1**

In standard output, you should see something that looks very similar to the following:



```
PLAY [Install the apache web service] ******************************************

TASK [setup] *******************************************************************
ok: [node-2]
ok: [node-1]

TASK [install apache] **********************************************************
changed: [node-1]
changed: [node-2]

TASK [start httpd] *************************************************************
changed: [node-2]
changed: [node-1]

PLAY RECAP *********************************************************************
node-1                     : ok=3    changed=2    unreachable=0    failed=0
node-2                     : ok=3    changed=2    unreachable=0    failed=0
```

Figure 1: apache_basic playbook stdout

Notice that the play and each task is named so that you can see what is being done and to which node it is being done to. You also may notice a task in there that you didn't write; <cough> `setup`

<cough>. This is because the `setup` module runs by default. To turn if off, you can specify gather_facts: false in your play definition like this:

```
---
- hosts: web
  name: Install the apache web service
  become: yes
  gather_facts: false
```

Step 2: Remove Apache

OK, for the next several minutes or as much time as we can afford, we want to to experiment a little. We would like you to reverse what you've done, i.e. stop and uninstall apache on your web nodes. So, go ahead and edit your playbook and then when your finished, rerun it as specified in **Step 1**. For this exercise we aren't going to show you line by line, but we will give you a few hints.

- If your first task in the playbook was to install httpd and the second task was to start the service, which order do you think those tasks should be in now?
- If `started` makes sure a service is started, then what option ensures it is stopped?
- If `present` makes sure a package is installed, then what option ensures it is removed? Er… starts with an **ab**, ends with a **sent**

Feel free to browse the help pages to see a list of all options.

- Ansible yum module
- Ansible service module

Exercise 1.4 - Using Variables, Loops, and Handlers

Previous exercises showed you the basics of Ansible Core. In the next few exercises, we are going to teach some more advanced ansible skills that will add flexibility and power to your playbooks.

Ansible exists to make tasks simple and repeatable. We also know that not all systems are exactly alike and often require some slight change to the way an Ansible playbook is run. Enter variables.

Variables are how we deal with differences between your systems, allowing you to account for a change in port, IP address or directory.

Loops enable us to repeat the same task over and over again. For example, lets say you want to install 10 packages. By using an ansible loop, you can do that in a single task.

Handlers are the way in which we restart services. Did you just deploy a new config file, install a new package? If so, you may need to restart a service for those changes to take effect. We do that with a handler.

For a full understanding of variables, loops, and handlers; check out our Ansible documentation on these subjects.

Ansible Variables

Ansible Loops

Ansible Handlers

Section 1 - Adding variables and a loop to your playbook

To begin, we are going to create a new playbook, but it should look very familiar to the one you created in exercise 1.2

Step 1: Navigate to your home directory create a new project and playbook

```
% cd
% mkdir apache-basic-playbook
% cd apache-basic-playbook
% vim site.yml
```

Step 2: Add a play definition and some variables to your playbook. These include addtional packages your playbook will install on your web servers, plus some web server specific configurations.

```
---
- hosts: web
  name: This is a play within a playbook
  become: yes
  vars:
    httpd_packages:
      - httpd
      - mod_wsgi
    apache_test_message: This is a test message
    apache_max_keep_alive_requests: 115
```

Step 3: Add a new task called **install httpd packages**.

```
tasks:
  - name: install httpd packages
    yum:
      name: "{{ item }}"
      state: present
    with_items: "{{ httpd_packages }}"
    notify: restart apache service
```

What the Helsinki is happening here!?

- `vars:` You've told Ansible the next thing it sees will be a variable name
- `httpd_packages` You are defining a list-type variable called httpd_packages. What follows is a list of those packages
- `{{ item }}` You are telling Ansible that this will expand into a list item like `httpd` and `mod_wsgi`.
- `with_items: "{{ httpd_packages }}` This is your loop which is instructing Ansible to perform this task on every `item` in `httpd_packages`
- `notify: restart apache service` This statement is a `handler`, so we'll come back to it in Section 3.

Section 2 - Deploying files and starting a service

When you need to do pretty much anything with files and directories, use one of the Ansible Files modules. In this case, we'll leverage the `file` and `template` modules.

After that, you will define a task to start the apache service.

Step 1: Create a `templates` directory in your apache-basic-playbook directory and download two files.

Directory: /home/studentX/apache-basic-playbook

```
% mkdir templates
% cd templates
% curl -O http://ansible-workshop.redhatgov.io/workshop-files/httpd.conf.j2
% curl -O http://ansible-workshop.redhatgov.io/workshop-files/index.html.j2
```

Step 2: Add some file tasks and a service task to your playbook

```
- name: create site-enabled directory
  file:
    name: /etc/httpd/conf/sites-enabled
    state: directory

- name: copy httpd.conf
  template:
    src: templates/httpd.conf.j2
    dest: /etc/httpd/conf/httpd.conf
  notify: restart apache service

- name: copy index.html
  template:
    src: templates/index.html.j2
    dest: /var/www/html/index.html

- name: start httpd
  service:
    name: httpd
    state: started
    enabled: yes
```

So… what did I just write?

- `file:` This module is used to create, modify, delete files, directories, and symlinks.
- `template:` This module specifies that a jinja2 template is being used and deployed. `template` is part of the `Files` module family and we encourage you to check out all of the other file-management modules here.
- **jinja-who?** - Not to be confused with 2013's blockbuster "Ninja II - Shadow of a Tear", jinja2 is used in Ansible to transform data inside a template expression, i.e. filters.
- **service** - The Service module starts, stops, restarts, enables, and disables services.

Section 3 - Defining and using Handlers

There are any number of reasons we often need to restart a service/process including the deployment of a configuration file, installing a new package, etc. There are really two parts to this Section; adding a handler to the playbook and calling the handler after the task. We will start with the former.

Step 1: Define a handler

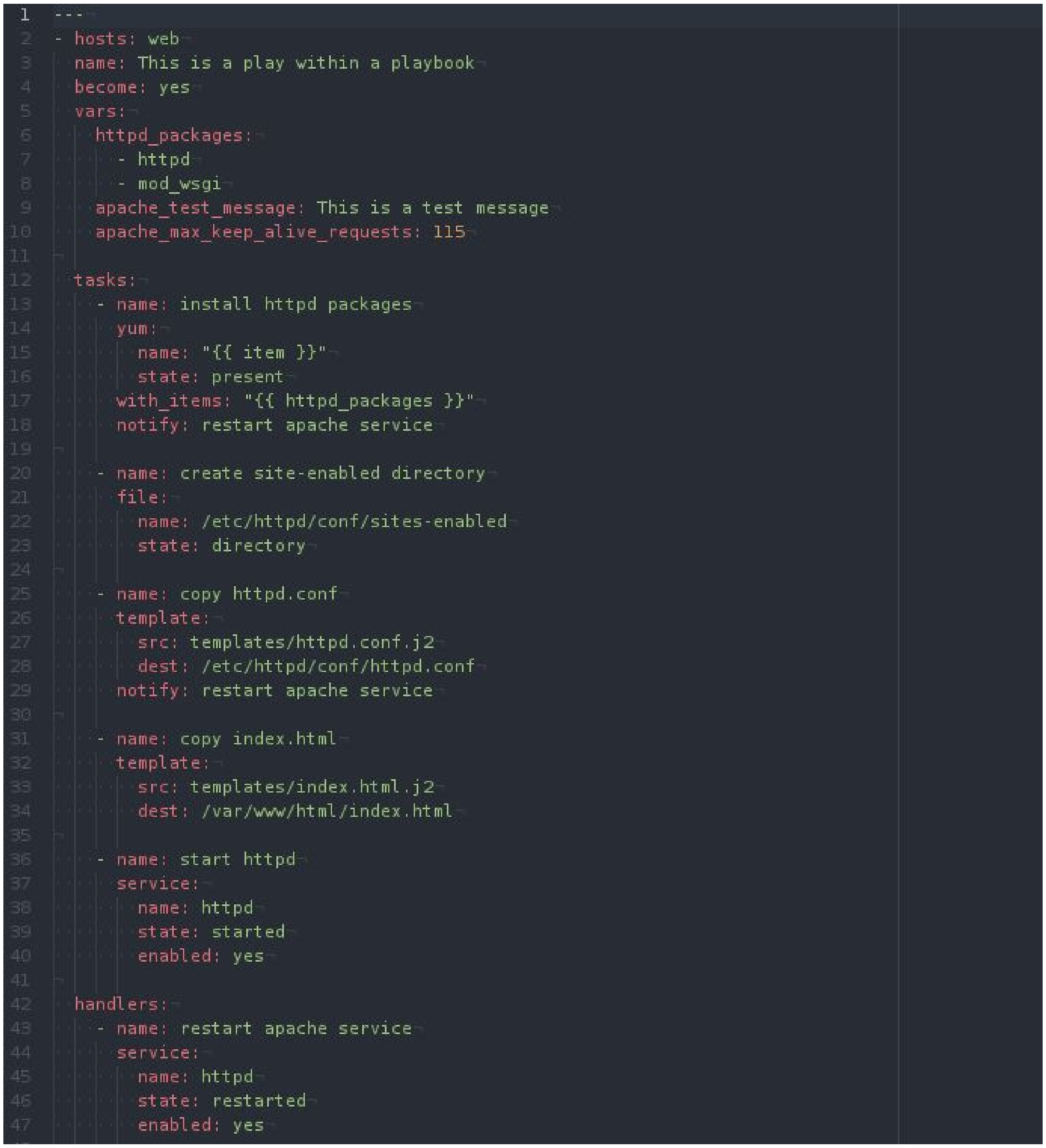
```
handlers:
  - name: restart apache service
    service:
      name: httpd
      state: restarted
      enabled: yes
```

You can’t have a former if you don’t mention the latter

- `handler:` This is telling the **play** that the `tasks:` are over, and now we are defining `handlers:`. Everything below that looks the same as any other task, i.e. you give it a name, a module, and the options for that module. This is the definition of a handler.
- `notify: restart apache service` …and here is your latter. Finally! The `nofify` statement is the invocation of a handler by name. Quite the reveal, we know. You already noticed that you’ve added a `notify` statement to the `copy httpd.conf` task, now you know why.

Section 4: Review

Your new, improved playbook is done! But don’t run it just yet, we’ll do that in our next exercise. For now, let’s take a second look to make sure everything looks the way you intended. If not, now is the time for us to fix it up. The figure below shows line counts and spacing.



```yaml
---
- hosts: web
  name: This is a play within a playbook
  become: yes
  vars:
    httpd_packages:
      - httpd
      - mod_wsgi
    apache_test_message: This is a test message
    apache_max_keep_alive_requests: 115

  tasks:
    - name: install httpd packages
      yum:
        name: "{{ item }}"
        state: present
      with_items: "{{ httpd_packages }}"
      notify: restart apache service

    - name: create site-enabled directory
      file:
        name: /etc/httpd/conf/sites-enabled
        state: directory

    - name: copy httpd.conf
      template:
        src: templates/httpd.conf.j2
        dest: /etc/httpd/conf/httpd.conf
      notify: restart apache service

    - name: copy index.html
      template:
        src: templates/index.html.j2
        dest: /var/www/html/index.html

    - name: start httpd
      service:
        name: httpd
        state: started
        enabled: yes

  handlers:
    - name: restart apache service
      service:
        name: httpd
        state: restarted
        enabled: yes
```

Figure 1: Completed Playbook - w/Spacing

```yaml
- hosts: web
  name: This is a play within a playbook
```

```
become: yes
vars:
  httpd_packages:
    - httpd
    - mod_wsgi
  apache_test_message: This is a test message
  apache_max_keep_alive_requests: 115

tasks:
  - name: httpd packages are present
    yum:
      name: "{{ item }}"
      state: present
    with_items: "{{ httpd_packages }}"
    notify: restart apache service

  - name: site-enabled directory is present
    file:
      name: /etc/httpd/conf/sites-enabled
      state: directory

  - name: latest httpd.conf is present
    template:
      src: templates/httpd.conf.j2
      dest: /etc/httpd/conf/httpd.conf
    notify: restart apache service

  - name: latest index.html is present
    template:
      src: templates/index.html.j2
      dest: /var/www/html/index.html

  - name: httpd is started and enabled
    service:
      name: httpd
      state: started
      enabled: yes

handlers:
  - name: restart apache service
    service:
      name: httpd
      state: restarted
```

Exercise 1.5 - Running the apache-basic-playbook

Congratulations! You just wrote a playbook that incorporates some key Ansible concepts that you use in most if not all of your future playbooks. Before you get too excited though, we should probably make sure it actually runs.

So, lets do that now.

Section 1 - Running your new apache playbook

Step 1: Make sure you are in the right directory and create a host file.

```
% cd ~/apache-basic-playbook
```

> Since you already did the work of creating a host file in Lession 1.0, feel free to just copy `hosts` from your `apache_basic` project. Or, hey… if you like to type, create a new file called `hosts` and put this in it.

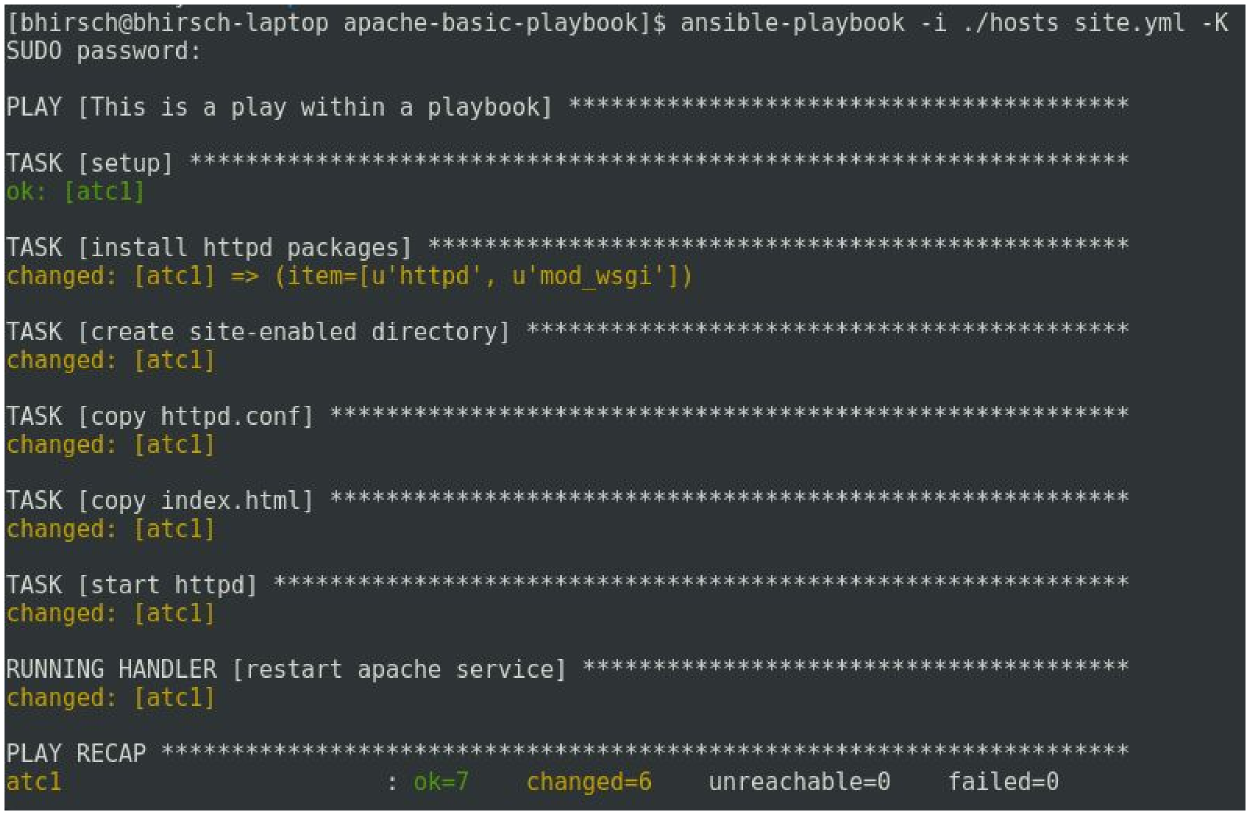
```
[web]
node-1 ansible_host=<IP Address of your node-1>
node-2 ansible_host=<IP Address of your node-2>
```

Step 2: Run your playbook

```
% ansible-playbook -i ./hosts site.yml -k
```

Section 2: Review

If successful, you should see standard output that looks very similar to the following. If not, just let us know. We'll help get things fixed up.



```
[bhirsch@bhirsch-laptop apache-basic-playbook]$ ansible-playbook -i ./hosts site.yml -K
SUDO password:

PLAY [This is a play within a playbook] ****************************************

TASK [setup] *******************************************************************
ok: [atc1]

TASK [install httpd packages] **************************************************
changed: [atc1] => (item=[u'httpd', u'mod_wsgi'])

TASK [create site-enabled directory] *******************************************
changed: [atc1]

TASK [copy httpd.conf] *********************************************************
changed: [atc1]

TASK [copy index.html] *********************************************************
changed: [atc1]

TASK [start httpd] *************************************************************
changed: [atc1]

RUNNING HANDLER [restart apache service] ***************************************
changed: [atc1]

PLAY RECAP *********************************************************************
atc1                       : ok=7    changed=6    unreachable=0    failed=0
```

Figure 1: apache-basic-playbook stdout

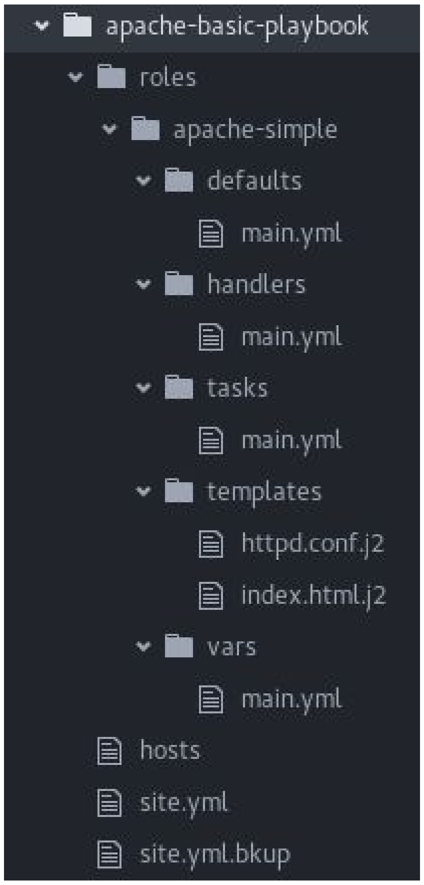
Exercise 1.6 - Roles: Making your playbooks reusable

While it is possible to write a playbook in one file as we've done throughout this workshop, eventually you'll want to reuse files and start to organize things.

Ansible Roles is the way we do this. When you create a role, you deconstruct your playbook into parts and those parts sit in a directory structure. "Wha?? You mean that seemingly useless best practice you mentioned in exercise 1.2?". Yep, that one.

For this exercise, you are going to take the playbook you just wrote and refactor it into a role. In addition, you'll learn to use Ansible Galaxy.

Let's begin with seeing how your apache-basic-playbook will break down into a role.



```
apache-basic-playbook
    roles
        apache-simple
            defaults
                main.yml
            handlers
                main.yml
            tasks
                main.yml
            templates
                httpd.conf.j2
                index.html.j2
            vars
                main.yml
    hosts
    site.yml
    site.yml.bkup
```

Figure 1: apache-basic-playbook role directory structure

Fortunately, you don't have to create all of these directories and files by hand. That's where Ansible Galaxy comes in.

Section 1 - Using Ansible Galaxy to initialize a new role

Ansible Galaxy is a free site for finding, downloading, and sharing roles. It's also pretty handy for creating them which is what we are about to do here.

Step 1: Navigate to your `apache-basic-playbook` project

```
% cd ~/apache-basic-playbook
```

Step 2: Create a directory called `roles` and `cd` into it

```
% mkdir roles
% cd roles
```

Step 3: Use the `ansible-galaxy` command to initialize a new role called `apache-simple`

```
% ansible-galaxy init apache-simple
```

Take a look around the structure you just created. It should look a lot like Figure 1 above. However, we need to complete one more step before moving onto section 2. It is Ansible best practice to clean out role directories and files you won’t be using. For this role, we won’t be using anything from `files`, `tests`.

Step 4: Remove the `files` and `tests` directories

```
% cd ~/apache-basic-playbook/roles/apache-simple/
% rm -rf files tests
```

Section 2: Breaking your `site.yml` playbook into the newly created `apache-simple` role

In this section, we will separate out the major parts of your playbook including `vars:`, `tasks:`, `template:`, and `handlers:`

Step 1: Make a backup copy of `site.yml`, then create a new `site.yml`

```
% mv site.yml site.yml.bkup
% vim site.yml
```

Step 2: Add the play definition and the invocation of a single role

```
---
- hosts: web
  name: This is my role-based playbook
  become: yes
  roles:
    - apache-simple
```

Step 3: Add some default variables to your role in `roles/apache-simple/defaults/main.yml`

```
---
# defaults file for apache-simple
apache_test_message: This is a test message
apache_max_keep_alive_requests: 115
```

Step 4: Add some role-specific variables to your role in `roles/apache-simple/vars/main.yml`

```
---
# vars file for apache-simple
httpd_packages:
  - httpd
  - mod_wsgi
```

Hey, wait just a minute there buster… did you just have us put variables in two seperate places?

Yes… yes we did. Variables can live in quite a few places. Just to name a few:

- vars directory
- defaults directory
- group_vars directory
- In the playbook under the `vars:` section
- In any file which can be specified on the command line using the `--extra_vars` option
- On a boat, in a moat, with a goat *(disclaimer: this is a complete lie)*

Bottom line, you need to read up on variable precedence to understand both where to define variables and which locations take precedence. In this exercise, we are using role defaults to define a couple of variables and these are the most malleable. After that, we defined some variables in `/vars` which have a higher precedence than defaults and can't be overridden as a default variable.

Step 6: Create your role handler in `roles/apache-simple/handlers/main.yml`

```
---
# handlers file for apache-simple
- name: restart apache service
  service:
    name: httpd
    state: restarted
    enabled: yes
```

Step 7: Add tasks to your role in `roles/apache-simple/tasks/main.yml`

```
---
# tasks file for apache-simple
- name: install httpd packages
  yum:
    name: "{{ item }}"
    state: present
  with_items: "{{ httpd_packages }}"
  notify: restart apache service

- name: create site-enabled directory
  file:
    name: /etc/httpd/conf/sites-enabled
    state: directory

- name: copy httpd.conf
  template:
    src: templates/httpd.conf.j2
    dest: /etc/httpd/conf/httpd.conf
  notify: restart apache service

- name: copy index.html
  template:
    src: templates/index.html.j2
    dest: /var/www/html/index.html
```

```
- name: start httpd
  service:
    name: httpd
    state: started
    enabled: yes
```

Step 8: Download a couple of templates into `roles/apache-simple/templates/`. And right after that, let’s clean up from exercise 2.1 by removing the old templates directory.

```
% cd ~/apache-basic-playbook/roles/apache-simple/templates/
% curl -O http://ansible-workshop.redhatgov.io/workshop-files/httpd.conf.j2
% curl -O http://ansible-workshop.redhatgov.io/workshop-files/index.html.j2
% rm -rf ~/apache-basic-playbook/templates/
```

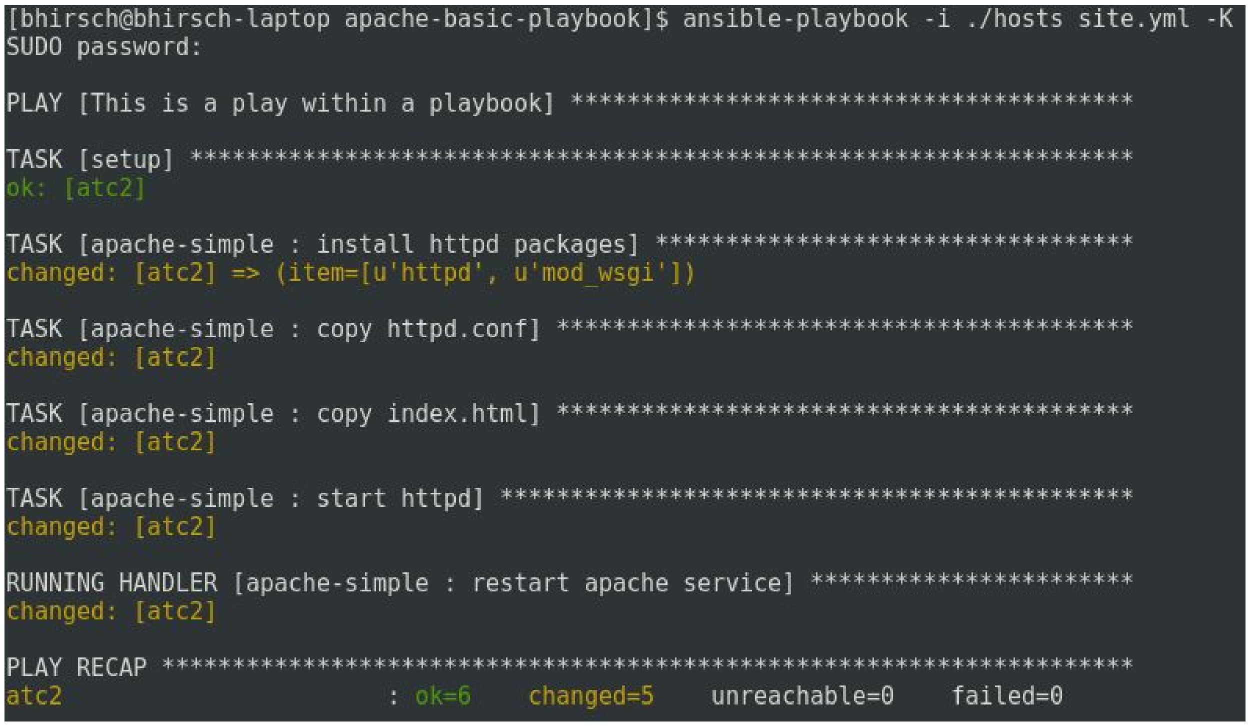
Section 3: Running your new role-based playbook

Now that you’ve successfully separated your original playbook into a role, let’s run it and see how it works.

Step 1: Run the playbook

```
% ansible-playbook -i ./hosts site.yml -k
```

If successful, your standard output should look similar to the figure below.



```
[bhirsch@bhirsch-laptop apache-basic-playbook]$ ansible-playbook -i ./hosts site.yml -K
SUDO password:

PLAY [This is a play within a playbook] ****************************************

TASK [setup] *******************************************************************
ok: [atc2]

TASK [apache-simple : install httpd packages] **********************************
changed: [atc2] => (item=[u'httpd', u'mod_wsgi'])

TASK [apache-simple : copy httpd.conf] *****************************************
changed: [atc2]

TASK [apache-simple : copy index.html] *****************************************
changed: [atc2]

TASK [apache-simple : start httpd] *********************************************
changed: [atc2]

RUNNING HANDLER [apache-simple : restart apache service] ***********************
changed: [atc2]

PLAY RECAP *********************************************************************
atc2                       : ok=6    changed=5    unreachable=0    failed=0
```

Figure 1: ansible-basic-playbook role-based stdout

Section 3: Review

You should now have a completed playbook, `site.yml` with a single role called `apache-simple`. The advantage of structuring your playbook into roles is that you can now add new roles to the playbook using Ansible Galaxy or simply writing your own. In addition, roles simplify changes to variables, tasks, templates, etc.