Learning Objectives

Learners will be able to...

- **Explain reusing artifacts**
- **Construct an artifact**
- **Distinguish between import_ and include_**

Code Reusability

Code Reuse

A major advantage of Infrastructure as Code is the ability to reuse code. Likewise, one of Ansible's key features is the ability to reuse code through the use of modules and roles.

definition

Introducing Roles

You'll remember from previous assignments:
- **Modules** in Ansible are pre-written python scripts that can be used to perform specific tasks, such as creating a user, installing a package, or starting/stopping services.
- **Plays** are a set of tasks that are executed on one or more hosts in a specific order.

Roles are a way of organizing and grouping related tasks and variables into reusable units. A role can be thought of as a collection of tasks, templates, files, and variables that can be applied to multiple plays. Roles can be included in playbooks and can be shared across projects.

Plays are the main components of playbooks, and roles can be included in plays or playbooks. Modules are the building blocks used in tasks within plays and roles to perform specific actions on remote hosts. Consider an analogy: a module is like a brick, and a role is a collection of those bricks. The organization of these bricks (how they are "layered") is your play.

By using modules and roles, developers can write code once and reuse it multiple times, reducing the amount of time and effort required to manage and maintain IT infrastructure. Additionally, this approach encourages a more organized and structured way of managing complex playbooks.

In this assignment we will focus on roles as the preferred way to reuse your existing code and organize complex tasks. But before we focus our learning on roles, let's briefly cover the `import_tasks` module.

Importing Tasks

The Import Tasks Module

In addition to writing tasks directly into our playbooks or including them in roles, we can import a list of tasks, specified in a file, using `ansible.builtin.import_tasks`. It's a simple way of inserting content into your playbook and is useful in keeping commonly used tasks organized and accessible.

Consider the following example:

```
- hosts: all
  tasks:
    - name: Import my tasks
      ansible.builtin.import_tasks:
        file: mytasks.yml
```

In this example, the `file` parameter is importing a specific file, `mytasks.yml`; the tasks contained in the file will be added to the current playbook for subsequent execution

`ansible.builtin.import_tasks` simply executes the tasks from the imported file as they would've been written in the playbook itself. Additionally, the same variables from the main playbook will be available for the included file.

▼
Advantages of grouping tasks in separate files

Breaking up tasks into separate files has distinct advantages; not only can it help organize a burgeoning playbook, it allows tasks to be reused across multiple playbooks based on different use cases. Additionally, should a file be used across multiple playbooks and its contained tasks need to be updated, only that file needs to be updated, negating the need to manually update each playbook.

Take for example a file containing a company's customer database, which may have its own tasks that keep it up to date. Should those tasks require updates, only that file needs to be updated to implement the changes across all of your playbooks.

Roles Folder Structure

Role Directories

Roles enable the automatic loading of related Ansible artifacts including your playbooks' requisite vars, files, tasks, templates, and handlers based on a standardized directory structure. Roles are the most powerful way to streamline your playbooks and make them more modular and reusable.

Each Ansible role resides within its corresponding directory, and the defined directory structure is comprised of seven* standard directories.

> topic
>
> *In fact, there are eight standard directories; the eighth has been excluded because it is limited to advanced usage.

Here is the roles directory structure:

```
roles/
    common/               # this hierarchy represents a "role"
        tasks/            #
            main.yml      #  <-- tasks file can include smaller
files if warranted
        handlers/         #
            main.yml      #  <-- handlers file
        templates/        #  <-- files for use with the template
resource
            ntp.conf.j2   #  <-- templates end in .j2
        files/            #
            config.conf   #  <-- files for use with the copy
resource
        vars/             #
            main.yml      #  <-- variables associated with this
role
        defaults/         #
            main.yml      #  <-- default lower priority
variables for this role
        meta/             #
            main.yml      #  <-- role dependencies

    webserver/            # webserevr role
    monitoring/           # monitoring role
    database/             # database role

webserver.yml             # webserver playbook
database.yml              # database playbook
```

▼ **Description of each file contained in the `common` role:**

- **`tasks/main.yml` - contains the main list of tasks to be executed. Typically, you will write series of tasks in separate files, and include them here.**
- **`handlers/main.yml` - defines handlers, which may be used within or outside this role, to apply configuration changes when needed.**
- **`templates/ntp.conf.j2` - holds the templates that the role deploys, similar to the *files* directory. However, the jinja2 templates can be rendered and copied to the target hosts during playbook execution.**
- **`files/config.conf` - holds the files that the role deploys to remote hosts.**
- **`vars/main.yml` - defines variables used internally for the role (other than those defined in the *defaults* directory)**
- **`defaults/main.yml` - a configuration file where default values are defined for variables in this role. These variables have the lowest priority, making them easily overridden by any other variable**

(including inventory variables).

- `meta/main.yml` - contains metadata for the role, including role dependencies, author information, and supported platforms for optional Galaxy metadata.

In the above example, we have four roles:
- `common` is a role applied to all our instances
- The `webserver` role is assigned to webserver instances
- The `monitoring` role is monitoring functionality used in all of our instances
- The `database` role is associated with database-specific instances

There are also two playbooks:
- `webserver.yml` will include the roles `common`, `webserver`, and `monitoring`
- `database.yml` will include `common`, `database`, and `monitoring`

> **definition**
>
> By default, Ansible will search for relevant content included in a `main.yml` file (including `main.yaml` and `main`) in each directory within a role.

Here is an example of the `database.yml` playbook, assuming the roles are defined as above in separate directories:

```
database.yml
---
- name: Install and configure database server
  hosts: databases
  become: true
  roles:
    - common
    - database
    - monitoring
```

In the example playbook above, you'll notice a few important things included:
- The playbook is intended to configure the `hosts: databases` group
- `become: true` is used because we need root privileges in order to configure the remote hosts
- For `roles`, we want to include our roles directory for `database`, but also the role directories `common` because we want it to apply to all hosts, and `monitoring` because we want it to apply to the `databases` hosts as well.

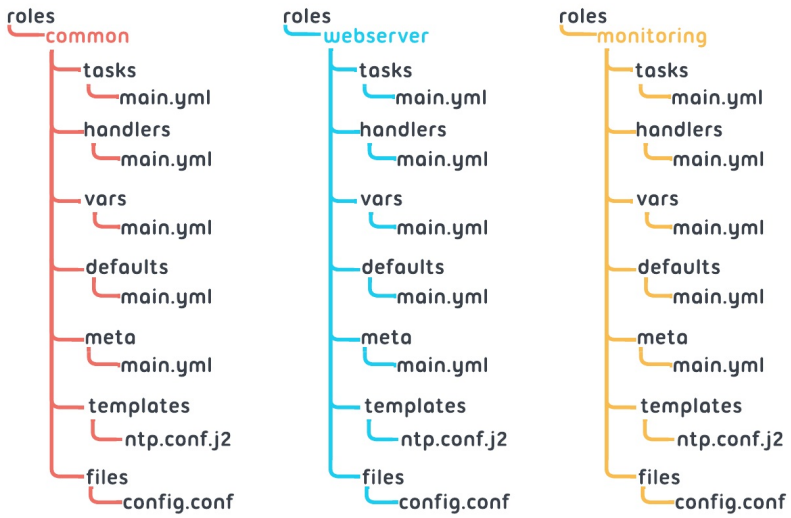
Three Ways of Using Roles

1. Roles At Play Level:

The “original” way roles are used in a play, the `roles` option will automatically import and parse files while your playbook is being parsed.

```
webserver.yml
---
- name: Install and configure webserver
  hosts: webservers
  roles:
    - common
    - webserver
    - monitoring
```

Below is the file structure for `common`, `webserver`, and `monitoring` roles.



When you use the roles option at the play level, for each role (`common`, `webserver`, and `monitoring` in the above example), Ansible will add each existing `main.yml`, Jinja2, and config file to the play.

To make variables available to the tasks within a play, you can pass them to roles by using `vars`, as shown in the following example:

```
---
- hosts: webservers
  roles:
    - common
      vars:
        hostname: "{{ hostname }}"
    - webserver
    - monitoring
      vars:
        hostname: "{{ hostname }}"
```

2. Including Roles: Dynamic Reuse

`included role` allows you to reuse roles dynamically by loading and executing them as a task anywhere in the play's `tasks` section.

This is distinctly different than running roles in the `roles` section, which are automatically executed prior to any other tasks. Should the need arise, `include_role` allows roles to be run after other tasks. Additionally, `include_role` allows a role to be included miltiple with different variables, rather than only being run once.

> growthhack
>
> ### Include Role Use Case
>
> `include_role` is often used in conjunction with "helper functions" because they tend to be used several times with different parameters.

Below is an example of how to include a role:

```
webserver.yml
---
- hosts: webservers
  tasks:
    - name: Include common role
      ansible.builtin.include_role:
        name: common
      vars:
        hostname: "{{ hostname }}"
    - name: Include common role
      ansible.builtin.include_role:
        name: common
      vars:
        hostname: "{{ otherhost }}"
    - name: Include webserver role
      ansible.builtin.include_role:
        name: webserver
```

The play shown above includes several roles; each will be executed at the start of each task. The same role behavior can be adjusted to pass other keywords, such as `vars` or tags to run select tasks. As can be seen in the example, the `common` role is included twice, passing two different `vars`. Finally, `when` can also be used to include roles conditionally.

3. Importing Roles: Static Reuse

Similarly, `import_role` also executes a role within a playbook. However it is a static import, meaning that the role will be executed at the time of the import rather than at the time of the task.

```
webserver.yml
---
- hosts: webservers
  tasks:
    - name: Include common role
      ansible.builtin.import_role:
        name: common
      vars:
        hostname: "{{ hostname }}"
    - name: Include webserver role
      ansible.builtin.import_role:
        name: webserver
```

In the play shown above, the `common` and `webserver` roles will be executed at the start of the play.

Roles Arguments and Validation

Enabling Role Argument Validation

It is possible to define arguments and default values, and enable role argument validation to check supplied parameters against those specifications. This is done by a task created to run before the role's execution to determine whether or not it will run.

> topic
>
> For Ansible version 2.1.1 and above, the `meta/argument_specs.yml` is the directory/file used for the defined specifications.

Let's look at an example role's structure and files used to install java:

```
roles/
  java/
    defaults/
      main.yml # file with our arguments default values
    tasks/
      main.yml # tasks to install Java
```

- `defaults/main.yml` specifies the default argument value:

```
java_version: 13
```

- `tasks/main.yml`

```
- name: Ensure zulu apt repository key is present
  ansible.builtin.apt_key:
    id=0xB1998361219BD9C9
    keyserver=hkp://keyserver.ubuntu.com:80
    state=present

- name: Ensure the zulu apt repository is present
  ansible.builtin.apt_repository:
    repo='deb http://repos.azulsystems.com/ubuntu stable main'
    update_cache=yes
    state=present

- name: Install Java
  ansible.builtin.apt: name=zulu-{{ java_version }} state=latest

- name: Select java version
  ansible.builtin.alternatives:
    name: java
    path: /usr/lib/jvm/zulu-{{ java_version }}-amd64/bin/java
```

In the example above, we can specify any version of Java we want to install from a third party repo.

To install a version specified in the defaults:

```
- include_role:
    name: Install Java
```

... or if you need a specific version:

```
- include_role:
    name: Install Java
  vars:
    java_version: 17
```

It's important to note that there are no validation rules or any requirements by default (unlike Terraform, for example), which can lead to errors.

The example below defines the available arguments that can be passed to the role, in this case the `java_version` argument.

```
ansible.builtin.validate_argument_specs:
  # roles/java/tasks/main.yml entry point
  main:
    short_description: The main entry point for the java role.
    options:
      java_version:
        type: "int"
        required: false
        default: 13
        description: "The integer value, defaulting to 13."
```

- `type` specified that it must be an int
- `required: false` specifies that the argument can be omitted
- `default: 13` specified that the default value is 13 if no other argument is provided
- `description` provides context to improve readability

Dependencies

##

Role Dependencies

Should you have a role in your Ansible playbook that requires another role to be run prior, `dependencies` will automatically run the prerequisite role first. Essentially, role dependencies ensure predictable conditions for the remote host.

> warning
>
> Contrary to the name's implication, role dependencies are actually prerequisites rather than true dependencies.

When your playbook is parsed and all listed roles are loaded, Ansible will run any roles listed under `dependencies` first, followed by the remaining roles. This process is simple by design; it increases role modularity, and consequently, its reusability.

> info
>
> Role dependencies are included statically, meaning they are only executed once. This is true even if multiple roles have the same role dependencies listed.

The `meta/main.yml` file is where dependencies are specified. If a role has prerequisite role dependencies, those roles (and their parameters, if applicable) are listed as shown in the example below:

```
# roles/java/meta/main.yml
---
dependencies:
  - role: common
    vars:
      parameter: 5
```

> Dependencies can also be installed by using the `ansible-galaxy` command, which downloads and install roles from Ansible-Galaxy. This will be covered in the next assignment!