Ansible Fundamentals

Writing Ansible Playbooks

Objectives

This module demonstrates how to:

- Create a simple playbook.
- Create and reference variables in playbooks.
- Run conditional tasks.
- Trigger Tasks with Handlers
- Recover from Errors with Blocks
- Deploy Files with Jinja2 Templates
- Process Variables with Jinja2 Filters

Writing Playbooks

- An Ansible Playbook is the main way to automate tasks in Ansible.
- A *playbook* is a YAML-based text file containing a list of one or more plays to run in a specific order.
- A *play* is an ordered list of *tasks* run against specific hosts within an inventory.
- Each task runs a module that performs some simple action on or for the managed host.
- Most tasks are idempotent and can be safely run a second time without problems.
- Playbooks can change lengthy, complex manual administrative tasks into an easily repeatable routine with predictable and successful outcomes.

Creating a Simple Playbook

Red Hat

Formatting an Ansible Playbook

- A playbook is saved using the standard file extension **.yml**.
- Indentation with space character indicates the structure of the data in the file.
- YAML does not place strict requirements on how many spaces are used for the indentation, but there are two basic rules.
  - Data elements at the same level in the hierarchy (such as items in the same list) must have the same indentation.
  - Items that are children of another item must be indented more than their parents.
- Only the space character can be used for indentation. Tab characters are not allowed.

Example: A simple playbook with one play, of one task

1. A name for the play to help document its intended purpose.
2. Hosts or groups on which the play will run, taken from the inventory.
3. This play will perform privilege escalation.
4. The beginning of the list of tasks in the play.
5. Clear descriptive names for each task help document what each task does.
6. Each task uses one module to perform the work, in this case **user**.
7. Argument for the **user** module to specify the user to manage, its UID number, and that it should exist.

```
---
- name: Converted ad hoc command example   1
  hosts: all                                2
  become: yes                               3
  tasks:                                    4
    - name: user exists with UID 4000       5
      user:                                 6
        name: newbie                        7
        uid: 4000                           7
        state: present                      7
```


Red Hat

Running an Ansible Playbook

Use the **ansible-playbook** command to run a playbook:

```
[user@host ansible]$ ansible-playbook site.yml

PLAY [Converted ad hoc command example] ****************************************

TASK [Gathering Facts] *********************************************************
ok: [localhost]

TASK [user exists with UID 4000] ***********************************************
changed: [localhost]

PLAY RECAP *********************************************************************
localhost     : ok=2  changed=1  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
```

Limiting Playbook Execution

- You can also limit the hosts you target on a particular run with the **`--limit`** flag.
- The limit is a host pattern that further limits the hosts for the play.

- For example:
  - A play has **hosts: webservers** in its definition
  - You run the playbook containing the play with the **--limit datacenter2** option
  - datacenter2 and webservers are both groups in the inventory
  - The play will only run on hosts that are in both webservers and datacenter2

```
ansible-playbook site.yml --limit datacenter2
```

Validating a Playbook

- Syntax validation with **`--syntax-check`**.
  - Running **`--syntax-check`** on the playbook will verify that it can be ingested by ansible.
  - The option can either be before or after the playbook.

```
[student@workstation ~]$ ansible-playbook --syntax-check webserver.yml
ERROR! Syntax Error while loading YAML.
  mapping values are not allowed in this context

The error appears to have been in ...output omitted... line 3, column 8, but may
be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

- name:play to setup web server
  hosts: servera.lab.example.com
       ^ here
```

Validating a Playbook

You can use the **-C** option to perform a dry run of the playbook execution. This causes Ansible to report what changes would have occurred if the playbook were executed, but does not make any actual changes to managed hosts.

```
[student@workstation ~]$ ansible-playbook -C webserver.yml

PLAY [play to setup web server] ************************************************

TASK [Gathering Facts] *********************************************************
ok: [servera.lab.example.com]

TASK [latest httpd version installed] ******************************************
changed: [servera.lab.example.com]

PLAY RECAP *********************************************************************
servera.lab.example.com    : ok=2    changed=1    unreachable=0    failed=0
```

Using Variables in Plays

Red Hat

Objectives

- Explain the key places where variables are commonly set.
- Explain the basic rules of variable precedence.
- Create and run a playbook that uses variables.

Introduction to Ansible Variables

- Ansible supports variables that you can use to store values for reuse throughout an Ansible project.
- This simplifies the creation and maintenance of a project and reduce the number of errors.
- Variables provide a convenient way to manage dynamic values.
- Examples of values that variables might contain:
  - Users to create, modify or delete.
  - Software to install or uninstall.
  - Services to stop, start, or restart.
  - Files to create, modify, or remove.
  - Archives to retrieve from the Internet, or to extract.

Naming Variables

- Variable names must start with a letter, and they can only contain letters, numbers, and underscores.

Invalid variable names	Valid Variable names
web server web-server	web_server
remote.file	remote_file
1st file 1st_file	file_1 file1
remoteserver$1	remote_server_1 remote_server1

Variable Scope

- **Global**
  - The value is set for all hosts.
  - Example: extra variables you set in the job template
- **Host**
  - The value is set for a particular host (or group).
  - Examples: variables set for a host in the inventory or a host_vars directory, gathered facts
- **Play**
  - The value is set for all hosts in the context of the current play.
  - Examples: **vars** directives in a play, **include_vars** tasks, and so on

Defining Variables

- If a variable is defined at more than one level, the level with the highest precedence wins.
- A narrow scope generally takes precedence over a wider scope.
- Variables that you define in an inventory are overridden by variables that you define in the playbook.
- Variables defined in a playbook are overridden by “extra variables” defined on the command line with the **-e** option.

Details on exact variable precedence are available at https://docs.ansible.com/ansible/latest/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable

Managing Variables in Playbooks

- Variables can be defined in multiple ways.
- One common method is to place a variable in a **`vars`** block at the beginning of a play:

```
- hosts: all
  vars:
    user_name: joe
    user_state: present
```

- **It is also possible to define play variables in external files.**
- **Use vars_files at the start of the play to load variables from a list of files into the play:**

```
- hosts: all
  vars_files:
    - vars/users.yml
```

Referencing Variables in Playbooks

- After declaring variables, you can use them in tasks.
- Reference a variable (replace it with its value) by placing the variable name in double braces: **{{ variable_name }}**
- Ansible substitutes the variable with its value when it runs the task.

```
- name: Example play
  hosts: all
  vars:
    user_name: joe

  tasks:
    # This line will read: Creates the user joe
    - name: Creates the user {{ user_name }}
      user:
        # This line will create the user named joe
        name: "{{ user_name }}"
        state: present
```

Referencing Variables

- When you reference one variable as another variable's value, and the curly braces start the value, you must use quotes around the value.
- This prevents Ansible for interpreting the variable reference as starting a YAML dictionary.
- The following message appears if the quotes are missing:

```
The offending line appears to be:

    - systemd:
        name: {{ service }}
              ^ here
We could be wrong, but this one looks like it might be an issue with
missing quotes. Always quote template expression brackets when they
start a value. For instance:

    with_items:
      - {{ foo }}

Should be written as:

    with_items:
      - "{{ foo }}"
```

Host Variables and Group Variables

- *Host variables* apply to a specific host.
- *Group variables* apply to all hosts in a host group or in a group of host groups.
- Host variables take precedence over group variables, but variables defined inside a play take precedence over both.
- Host variables and group variables can be defined
  - In the inventory itself.
  - In **host_vars** and **group_vars** directories in the same directory as the inventory.
  - In **host_vars** and **group_vars** directories in the same directory as the playbook. These are host and group based but have higher precedence than the inventory variables.

Using Directories to Set Host and Group Variables

- In the same directory as your playbook, create two directories, **group_vars** and **host_vars**.
  - To define group variables for the **servers** group, you would create a file named **group_vars/servers**.
  - In that file, set variables to values, using YAML syntax. (The example at right sets ansible_user to a string and newfiles to a list of values.)
- To define host variables for a particular host, create a file with a name matching the host in the **host_vars** directory.

```
ansible_user: devops
newfiles:
  - /tmp/a.conf
  - /tmp/b.conf
```

Using Directories to Set Host and Group Variables

```
project
├── group_vars
│   ├── all
│   ├── datacenters
│   ├── datacenters1
│   └── datacenters2
├── host_vars
│   ├── demo1.example.com
│   ├── demo2.example.com
│   ├── demo3.example.com
│   └── demo4.example.com
└── playbook.yml
```

- Set host and group variables in **host_vars** and **group_vars** directories in the same directory as your playbook
- Works just like the host or group variables in your inventory
- They have host scope just like inventory variables
- These “playbook” host and group variables have slightly higher precedence than inventory variables (and override them)
- The files or directories in **host_vars** and **group_vars** have the name of the host or group they apply to
- If you use a directory for the host or group name, it can contain multiple variable files which are all used

Defining Variables in Playbooks

- Below is an example of a play level variable defined in a playbook.
- The dictionary that stores the variable values is listed at the first indentation.
- This makes it easier to update the list of packages, especially if the list is used in several tasks

```
- name: Install packages
  hosts: all
  vars:
    packages:
      - nmap
      - httpd
      - php
      - mod_php
      - mod_ssl

  tasks:
    - name: Install software
      yum:
        name: "{{ packages }}"
        state: present
```

1. Dictionary named vars at the play level.
2. Name of variable.
3. List of values.
4. The packages variable expands to a list of packages for the yum module to install.

Red Hat

Selecting Items from a Dictionary

Arrays are multiple variables that can be browsed. It is possible to return one value from the array of values within a variable.

```
vars:
  users:
    aditya:
      uname: aditya
      fname: Aditya
      lname: Atwal
      home: /home/aditya
      shell: /bin/bash
    carlotta:
      uname: carlotta
      fname: Carlotta
      lname: Spencer
      home: /home/carlotta
      shell: /bin/zsh
```

To return the value of Aditya in this example, we would use the following syntax: **`users['aditya']['fname']`**

To grab Carlotta's home directory reference
`users['carlotta']['home']`

The Register Statement

The **register** statement will capture the output of a command or task. The output is saved into a temporary variable that can be used later in the playbook for either debugging purposes or to achieve something else, such as a particular configuration based on a command's output.

- Return values vary for each module.
- Registered variables are only stored in memory.

Protecting Sensitive Data

Objectives

- Encrypt files containing sensitive data using Ansible Vault
- Run playbooks that reference Ansible Vault-encrypted files
- Update Ansible Vault-encrypted files

Introduction to Ansible Vault

- Ansible Playbooks sometimes need access to sensitive data.
  - Passwords, API keys, other secrets
- These secrets are often passed to Ansible through variables.
- It is a security risk to store these secrets in plain text files.
- Ansible Vault provides a way to encrypt and decrypt files used by playbooks
- The **ansible-vault** command is used to manage these files

Create, View and Edit Encrypted Files

- You will need to provide a password to use or manipulate an Ansible Vault encrypted file
- Create a new encrypted file using the command

  `ansible-vault create filename`
- View an encrypted file using the command

  `ansible-vault view filename`
- Edit an an encrypted file using the command

  `ansible-vault edit filename`

Encrypt an Existing File

- Encrypt an existing file using the command **`ansible-vault encrypt filename`**
- Save the encrypted file to a new name using the option **`--output=new_filename`**
- Decrypt a file with **`ansible-vault decrypt filename`**

Playbooks and Ansible Vault

- Provide the vault password using the `--vault-id` option:
  `ansible-playbook --vault-id @prompt filename`
- The `@prompt` option will prompt the user for the Ansible Vault password.
- If you do not provide a password, Ansible returns an error.

Multiple Vault Passwords

- You can use the **`--vault-id`** option to set a label on an encrypted file.
- The following example sets the label *vars* on the file and prompts you for the password to use

  ansible-vault encrypt filename --vault-id vars@prompt

- If you have files encrypted with different passwords, you can use this to prompt multiple times:

  ansible-playbook --vault-id vars@prompt --vault-id playbook@prompt site.yml

Change the Password of an Encrypted File

- Change the password of an encrypted file using
  `ansible-vault rekey filename`
- The **rekey** subcommand can be used on multiple data files at once.
- It prompts for the current password, then the new password.

Suppressing Output from a Task

- Sometimes the output of **ansible-playbook** prints a sensitive value
- Suppress the output of a task by adding **no_log: true**
- This can make troubleshooting harder, so use it only where needed

```
- name: Print variable
  debug:
    msg: "{{ secret }}"
```

```
TASK [Print variable] *****************
ok: [localhost] => {
    "msg": "youknowit"
}
```

```
- name: Print variable
  debug:
    msg: "{{ secret }}"
  no_log: true
```

```
TASK [Print variable] *****************
ok: [localhost]
```

Task Iteration with Loops

Objectives

- Demonstrate basic looping functionality to iterate over tasks.

Task Iteration with Loops

- Using loops saves administrators from the need to write multiple tasks that use the same module.
- For example, instead of writing five tasks to ensure five users exists, you can write one task that iterates over a list of five users to ensure they all exist.
- Ansible supports iterating a task over a set of items using the `loop` keyword.
- Loops can be configured to repeat a task using each item in a list.
- The loop variable `item` holds the value used during each iteration.

Example without Loop and with Loop

```
- name: Create users
  hosts: servera.lab.example.com

  tasks:
    - name: Create users
      user:
        name: aditya
        state: present

    - name: Create users
      user:
        name: boris
        state: present

    - name: Create users
      user:
        name: carlotta
        state: present
```

```
- name: Create users
  hosts: servera.lab.example.com
  vars:
    myusers:
      - aditya
      - boris
      - carlotta

  tasks:
    - name: Create users
      user:
        name: "{{ item }}"
        state: present
      loop: "{{ myusers }}"
```

Simple Loops

- Confirmation that the playbook ran successfully:

```
[student@workstation plays]$ ansible-playbook loop.yml -b

PLAY [Create users]
*********************************************************************************************************************************

TASK [Gathering Facts]
*********************************************************************************************************************************
ok: [servera.lab.example.com]

TASK [Create users]
*********************************************************************************************************************************
changed: [servera.lab.example.com] => (item=aditya)
changed: [servera.lab.example.com] => (item=boris)
changed: [servera.lab.example.com] => (item=carlotta)

PLAY RECAP
*********************************************************************************************************************************
servera.lab.example.com                    : ok=2    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Simple Loops

- Confirmation that the accounts were created:

```
[student@servera ~]$ tail /etc/passwd
nginx:x:995:992:Nginx web server:/var/lib/nginx:/sbin/nologin
memcached:x:994:991:Memcached daemon:/run/memcached:/sbin/nologin
rabbitmq:x:993:990:RabbitMQ messaging server:/var/lib/rabbitmq:/sbin/nologin
dockerroot:x:992:989:Docker User:/var/lib/docker:/sbin/nologin
student:x:1000:1000:Student User:/home/student:/bin/bash
devops:x:1001:1001::/home/devops:/bin/bash
aditya:x:1004:1004::/home/aditya:/bin/bash
boris:x:1005:1005::/home/boris:/bin/bash
carlotta:x:1006:1006::/home/carlotta:/bin/bash
```

When are Loops Inefficient?

- The task on the left uses the yum module and a loop to install all the packages.
- This will launch the yum module three times, once for each item in the loop.
- But the yum module can take a list of packages and install them all in one operation. The task on the right is an example of this.
- The task on the right will execute one task, so it will be faster.

```
- name: Install a list of
packages
  yum:
    name: "{{ item }}"
    state: present
  loop:
    - nginx
    - postgresql
    - postgresql-server
```

```
- name: Install a list of
packages
  yum:
    name:
      - nginx
      - postgresql
      - postgresql-server
   state: present
```


Red Hat

Running Conditional Tasks

Objective

- Review the implementation of conditionals and handlers in Ansible tasks.

Running Tasks Conditionally

- Ansible can use conditionals to run or skip tasks when certain conditions are met.
- Variables and facts can both be tested by conditionals.
- Operators such as greater than (>) or less than (<) to compare strings, numerical data, or Boolean values can be used.

Some Uses for Conditional Tasks

- Run a task if a fact reporting the available memory on a managed host is lower than a value.
- Run different tasks to create users on a managed host based on which domain it belongs to
- Skip a task if a certain variable is not set or is set to a specific value
- Use the results of a previous task to determine whether to run the task

Using Conditionals

- The **`when`** statement is used to run a task conditionally.
  - If the condition is met, the task runs. If the condition is not met, the task is skipped.
  - In the following example the task will run only if the value of **`run_my_task`** is true:

```
---
- name: Simple Boolean Task Demo
  hosts: all
  vars:
    run_my_task: true

  tasks:
    - name: httpd package is installed
      yum:
        name: httpd
      when: run_my_task
```

Using Conditionals

- This example is a bit more sophisticated.
- It tests whether the **`my_service`** variable has a value. If it does, the value of **`my_service`** is used as the name of the package to install. If the **`my_service`** variable is not defined the task is skipped without an error.

```
---
- name: Test Variable is Defined Demo
  hosts: all
  vars:
    my_service: httpd

  tasks:
    - name: "{{ my_service }} package is installed"
      yum:
        name: "{{ my_service }}"
      when: my_service is defined
```

Example Conditions

OPERATION	EXAMPLE
Equal (value is a string)	ansible_machine == "x86_64"
Equal (value is numeric)	max_memory == 512
Less than	min_memory < 128
Greater than	min_memory > 256
Less than or equal to	min_memory <= 256
Greater than or equal to	min_memory >= 512
Not equal to	min_memory != 512
Variable exists	min_memory is defined

Example Conditions

OPERATION	EXAMPLE
Variable does not exist	min_memory is not defined
Boolean variable is true. The values of 1, True, or yes evaluate to true.	memory_available
Boolean variable is false. The values of 0, False, or no evaluate to false.	not memory_available
First variable's value is present as a value in second variable's list	ansible_distribution in supported_distros

Constructing Conditions with Ansible Facts

- The **ansible_facts['distribution']** variable is a fact set when the play runs, which identifies the operating system of the current managed host.
- The **supported_os** variable contains a list of operating systems supported by the playbook.
- If the value of **ansible_facts['distribution']** is in the **supported_os** list, the conditional passes and the task runs.

```
- name: Demonstrate "in" in a condition
  hosts: all
  gather_facts: yes
  become: yes
  vars:
    my_service: httpd
    supported_os:
      - RedHat
      - Fedora
  tasks:
    - name: Install "{{ my_service }}"
      yum:
        name: "{{ my_service }}"
        state: present
      when: ansible_facts['distribution'] in supported_os
```


Red Hat

Testing Multiple Conditions

- One when statement can be used to evaluate multiple conditions.
- Conditions are combined with either **`and`** or **`or`** keywords, and group with parentheses.
- Here are some examples:

```
when: ansible_distribution == "RedHat" or ansible_distribution == "Fedora"
```

```
when: ansible_distribution_version == "7.5" and ansible_kernel ==
 "3.10.0-327.el7.x86_64"
```

Testing Multiple Conditions

- The **`when`** keyword also supports using a list to describe a list of conditions.
- When a list is provided all of the conditions are combined using the **`and`** operation.

```
when:
  - ansible_distribution_version == "7.5"
  - ansible_kernel == "3.10.0-327.el7.x86_64"
```

Testing Multiple Conditions

- More complex conditional statements can be expressed by grouping conditions with parentheses.
- This ensures that they are correctly interpreted.

```
when: >
    ( ansible_distribution == "RedHat" and
      ansible_distribution_major_version == "7" )
    or
    ( ansible_distribution == "Fedora" and
    ansible_distribution_major_version == "28" )
```

Combining Loops and Conditional Tasks

- It is possible to combine loops and conditionals.
- In the following example, the **`mariadb-server`** package is installed by the **`yum`** module:
  - If there is a file system mounted on **`/`** with more than 300MB free.
- The **`ansible_mounts`** fact is a list of dictionaries, each one representing facts about one mounted file system.
- The loop iterates over each directory in the list, and the conditional statement is not met unless a dictionary is found representing a mounted file system where both conditions are true.

```
- name: install mariadb-server if enough space on root
  yum:
    name: mariadb-server
    state: latest
  loop: "{{ ansible_mounts }}"
  when: item.mount == "/" and item.size_available > 300000000
```

Combining Loops and Conditional Tasks

- The following playbook restarts the httpd service only if the postfix service is running:

1. Is Postfix running or not?
2. If it is not running and the command fails, do not stop processing it.
3. **register** saves information on the module's result in a variable named **result**.
4. The condition evaluates the output of the previous task. If the exit code of the **`systmectl`** command was 0, then Postfix is active and this task restarts the httpd service.

```
---
- name: Restart HTTPD if Postfix is Running
  hosts: all
  tasks:
    - name: Get Postfix server status
      command: /usr/bin/systemctl is-active postfix (1)
      ignore_errors: yes (2)
      register: result (3)

    - name: Restart Apache HTTPD based on Postfix status
      service:
        name: httpd
        state: restarted
      when: result.rc == 0 (4)
```

Triggering Tasks with Handlers

Objective

- Use handlers to run tasks when another task changes a managed host.

Ansible Handlers

- Ansible modules are designed to be *idempotent*.
- A properly written playbook and it tasks can be run multiple times without changing the managed host.
- However, sometimes when a task does make a change to the system, a further task may need to be run.
- For example, a change to a service's configuration file may then require that the service be reloaded so that the changed configuration takes effect.

Ansible Handlers

- **`Handlers`** are tasks that respond to a notification triggered by other tasks.
- Tasks only notify their handlers when the task changes something on a managed host.
- Each handler has a globally unique name and is triggered at the end of a block of tasks in a playbook.
- If not task notifies the handler by name then the handler will not run.
- If one or more tasks notify the handler, the handler will run exactly once after all other tasks in the play have completed.
- Because handlers are tasks, administrators can use the same modules in handlers that they would use for any other task.
- Normally, handlers are used to reboot hosts and restart services.
- Handlers can be considered as *inactive* tasks that only get triggered when explicitly invoked using a **`notify`** statement.

Ansible Handlers

1. The task that notifies the handler.
2. The notify statement indicates the task needs to trigger a handler.
3. The name of the handler to run.
4. The handlers keyword indicates the start of the list of handler tasks.
5. The name of the handler invoked by tasks.
6. The module to use for the handler.

```
tasks:
  - name: copy demo.example.conf configuration template 1
    template:
      src: /var/lib/templates/demo.example.conf.template
      dest: /etc/httpd/conf.d/demo.example.conf
    notify: 2
      - restart apache 3

handlers: 4
  - name: restart apache 5
    service: 6
      name: httpd
      state: restarted
```

Ansible Handlers

- In the previous example, the restart apache handler triggers when notified by the template task that a change happened.
- A task may call more than one handler in its notify section.
- Ansible treats the notify statement as an array and iterates over the handler names:

```
tasks:
  - name: copy demo.example.conf configuration template
    template:
      src: /var/lib/templates/demo.example.conf.template
      dest: /etc/httpd/conf.d/demo.example.conf
    notify:
      - restart mysql
      - restart apache

handlers:
  - name: restart mysql
    service:
      name: mariadb
      state: restarted

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

Describing the Benefits of Using Handlers

There are some important things to remember about using handlers:

- Handlers always run in the order specified by the handlers section of the play. They do not run in the order in which they are listed by notify statements in a task, or in the order in which tasks notify them.
- Handlers normally run after all other **tasks** in the play complete. A handler called by a task in the tasks part of the playbook will not run until *all* **tasks** under tasks have been processed.
- Handler names exist in a per-play namespace. If two handlers are incorrectly given the same name, only one will run.
- Even if more than on task notifies a handler, the handler only runs once. If no tasks notify it, a handler will not run.
- If a task that includes a **`notify`** statement does not report a **`changed`** result (for example, a package is already installed and the task reports **`ok`**), the handler is not notified. The handler is skipped unless another task notifies it. Ansible notifies handlers only if the task reports the **`changed`** status.

Recovering from Errors with Blocks

Objectives

- Use blocks to group tasks in a play and to recover from errors in the block.

Ansible Blocks

- In playbooks, *blocks* are clauses that logically group tasks as a unit.
- They can be used to control how tasks are executed.
- For example, a **block** can have a **`when`** conditional applied to the entire block.
- This means that all the tasks in the **block** will only run if the conditional is met.
- At right, the block groups two tasks that run only when **ansible_distribution** is **RedHat**

```
- name: block example
  hosts: all
  tasks:
    - name: installing and configuring Yum versionlock plugin
      block:
      - name: package needed by yum
        yum:
          name: yum-plugin-versionlock
          state: present
      - name: lock version of tzdata
        lineinfile:
          dest: /etc/yum/pluginconf.d/versionlock.list
          line: tzdata-2016j-1
          state: present
      when: ansible_distribution == "RedHat"
```


Red Hat

Block and Rescue: Recovering from Failure

- Blocks can also be used to back out of a change if tasks in the block fail.
- A **block** can have a set of tasks grouped in a **rescue** statement.
- The tasks in the **rescue** statement only run if a task in the **block** fails.
- Normally, the tasks in the rescue statement will recover the managed host from the failure. This is useful if the block exists because multiple tasks are needed to accomplish something.

Block, Rescue, and Always

- **block** also supports a third section, **always**
- After the **block** runs, and the **rescue** if there was a failure in **block**, the tasks in **always** run
- These tasks always run if the block runs at all, but they are subject to the conditional on the block
- To summarize:
  - **`block`**: Defines the main tasks to run.
  - **`rescue`**: Defines the tasks to run if the tasks defined in the block clause fail.
  - **`always`**: Defines the tasks that will always run independently of the success or failure of tasks defined in the block and rescue clauses.

Example of Block/Rescue/Always

```
tasks:
  - name: Upgrade DB
    block:
      - name: upgrade the database
        shell:
          cmd: /usr/local/lib/upgrade-database
    rescue:
      - name: revert the database upgrade
        shell:
          cmd: /usr/local/lib/revert-database
    always:
      - name: always restart the database
        service:
          name: mariadb
          state: restarted
```

- At left is an example of a **block** statement that has a **rescue** block and an **always** block
- The example has one task in each section, but there could be many tasks in each section
- If a task defined in the **`block`** clause fails, tasks defined in the **`rescue`** and **`always`** clauses are executed.
- If all tasks defined in the **block** clause succeed, then tasks in the **always** clause are executed
- If there were a **`when`** condition on the **`block`** clause, it would also apply to its **`rescue`** and **`always`** clauses.