TASK CONTROL & HANDLER

Overview	
Goal	Manage task control, handlers, and tags in Ansible playbooks
Objectives	• Construct conditionals and loops in a playbook • Implement handlers in a playbook • Implement tags in a playbook • Resolve errors in a playbook
Sections	• Constructing Flow Control (and Guided Exercise) • Implementing Handlers (and Guided Exercise) • Implementing Tags (and Guided Exercise) • Handling Errors (and Guided Exercise)
Lab	• Implementing Task Control

Constructing Flow Control

Task Iteration with Loops

Ansible supports several different ways to iterate a task over a set of items using a loop. Loops can repeat a task using each item in a list, the contents of each of the files in a list, a generated sequence of numbers, or using more complicated structures.
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 exist, one task can be written that iterates over a list of five users to ensure they all exist.

Simple Loops

A simple loop iterates a task over a list of items. The with_items key is added to the task, and takes as a value the list of items over which the task should be iterated. The loop variable item holds the current value being used for this iteration.
Consider the following snippet that uses the service module twice in order to ensure two network services are running:
Example: -

```
- name: Postfix is running
  service:
    name: postfix
    state: started

- name: Dovecot is running
  service:
    name: dovecot
    state: started
```

These two tasks can be rewritten to use a simple loop, so that only one task is needed to ensure both services are running:

```
- name: Postfix and Dovecot are running
  service:
    name: "{{ item }}"
    state: started
  with_items:
    - postfix
    - dovecot
```

The list used by with_items can be provided by a variable. In the following example, the variable mail_services contains the list of services that need to be running.

```
vars:
  mail_services:
    - postfix
    - dovecot

tasks:
  - name: Postfix and Dovecot are running
    service:
      name: "{{ item }}"
      state: started
    with_items: "{{ mail_services }}"
```

Other Common Loop Directives
The following table shows some additional types of loops supported by Ansible.

Ansible Loops

Loop Keyword	Description
with_file	Takes a list of control node file names. **item** is set to the content of each file in sequence.
with_fileglob	Takes a file name globbing pattern. **item** is set to each file in a directory on the control node that matches that pattern, in sequence, non-recursively.
with_sequence	Generates a sequence of items in increasing numerical order. Can take **start** and **end** arguments which have a decimal, octal, or hexadecimal integer value.
with_random_choice	Takes a list. **item** is set to one of the list items at random.

Ansible when Statement

The when statement is used to run a task conditionally. It takes as a value the condition to test. If the condition is met, the task runs. If the condition is not met, the task is skipped.

One of the simplest conditions which can be tested is whether a Boolean variable is true or false. The when statement in the following example causes the task to run only if run_my_task is true:

```
---
- hosts: all
  vars:
    run_my_task: true

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

References

Loops — Ansible Documentation
http://docs.ansible.com/ansible/playbooks_loops.html
Conditionals — Ansible Documentation
http://docs.ansible.com/ansible/playbooks_conditionals.html
What Makes A Valid Variable Name — Variables — Ansible Documentation
http://docs.ansible.com/ansible/playbooks_variables.html#what-makes-a-validvariable

Guided Exercise: Constructing Flow Control

#cd /home/ansible/playbook/dev-flowcontrol

Create a task file named **configure_database.yml**. This will define the tasks to install the extra packages, create **a my.cnf** on ansible control node and copy to managed hosts /etc/my.cnf and start **mariadb**. The include file can and will use the variables you defined in the **playbook.yml** file and inventory.

Create a my.cnf on ansible control node :-

#cat /home/ansible/playbook/dev-flowcontrol/my.cnf
robo 2.0 pakshi

#cat configure_database.yml

```
[ansible@robo dev-flowcontrol]$ cat configure_database.yml
- yum:
    name: "{{ extra_packages }}"

- copy:
    src: /home/ansible/playbook/dev-flowcontrol/my.cnf
    dest: "{{ configure_database_path }}"
    owner: ansible
    group: ansible
    mode: 0644
    force: yes

- service:
    name: "{{ db_service }}"
    state: started
    enabled: true
[ansible@robo dev-flowcontrol]$
```

In the same directory, create the **playbook.yml** playbook. Define a list variable, **db_users**, that consists of a list of two users, **db_admin** and **db_user**. Add a **configure_database_path** variable set to the file **/etc/my.cnf**.
Create a task that uses a loop to create the users only if the managed host belongs to the databases host group. The file should read as follows:

 # cat playbook.yml

```
[ansible@robo dev-flowcontrol]$ cat playbook.yml
---
- hosts: dev
  become: yes
  vars:
    db_package: mariadb-server
    db_service: mariadb
    db_users:
        - db_admin
        - db_user
    configure_database_path: /etc/my.cnf

  tasks:
  - name: Create the mariaDB users
    user:
      name: "{{ item }}"
    with_items: "{{ db_users }}"
    when: inventory_hostname in groups['dev']

  - name: Install the database server
    yum:
      name: "{{ db_package }}"
    when: db_package is defined

  - name: configure the database software
    include: configure_database.yml
    vars:
      extra_packages:
        - mariadb-bench
        - mariadb-libs
        - mariadb-test
    when: configure_database_path is defined
[ansible@robo dev-flowcontrol]$
```

#ansible-playbook --syntax-check playbook.yml

playbook: playbook.yml

ansible-playbook playbook.yml

```
[ansible@robo dev-flowcontrol]$ ansible-playbook playbook.yml
SUDO password:

PLAY [dev] *********************************************************************

TASK [Gathering Facts] *********************************************************
ok: [robo2.0]

TASK [Create the mariaDB users] ************************************************
changed: [robo2.0] => (item=db_admin)
changed: [robo2.0] => (item=db_user)

TASK [Install the database server] *********************************************
changed: [robo2.0]

TASK [yum] *********************************************************************
changed: [robo2.0]

TASK [copy] ********************************************************************
changed: [robo2.0]

TASK [service] *****************************************************************
fatal: [robo2.0]: FAILED! => {"changed": false, "msg": "Unable to start service mariadb
ntrol process exited with error code. See \"systemctl status mariadb.service\" and \"jo
        to retry, use: --limit @/home/ansible/playbook/dev-flowcontrol/playbook.retry

PLAY RECAP *********************************************************************
robo2.0                    : ok=5    changed=4    unreachable=0    failed=1
```

Note: - See above output, maria db won't start on client machine and it will show as failed. Kindly ignore.

To Verify:-

#ansible dev -a 'yum list installed mariadb-bench mariadb-libs mariadb-test' -b

```
robo2.0 | CHANGED | rc=0 >>
Loaded plugins: fastestmirror
Installed Packages
mariadb-bench.x86_64                1:5.5.60-1.el7_5                @base
mariadb-libs.x86_64                 1:5.5.60-1.el7_5                @anaconda
mariadb-test.x86_64                 1:5.5.60-1.el7_5                @base
```

ansible dev -a 'grep robo /etc/my.cnf' -b

```
robo2.0 | CHANGED | rc=0 >>
robo 2.0 pakshi
```

ansible dev -a 'id db_user'

```
robo2.0 | CHANGED | rc=0 >>
uid=1002(db_user) gid=1002(db_user) groups=1002(db_user)
```

Note: - please cleanup once it done.

#ansible dev -a 'yum remove mariadb-bench mariadb-libs mariadb-test -y' -b

Ansible Handlers

Ansible modules are designed to be idempotent. This means that in a properly written playbook, the playbook and its tasks can be run multiple times without changing the managed host, unless they need to make a change in order to get the managed host to the desired state.

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.

Handlers are tasks that respond to a notification triggered by other tasks. Each handler has a globally-unique name and is triggered at the end of a block of tasks in a playbook. If no task notifies the handler by name, it will not run. If one or more tasks notify the handler, it 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 for any other task. Normally, handlers are used to reboot hosts and restart services.

Handlers can be seen as inactive tasks that only get triggered when explicitly invoked using a notify statement.

Using Handlers

As discussed in the Ansible documentation, there are some important things to remember about using handlers:

- Handlers are always run in the order in which the handlers section is written in the play, not in the order in which they are listed by the notify statement on a particular task.
- Handlers 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 of the tasks under tasks: have been processed.
- Handler names live in a global namespace. If two handlers are incorrectly given the same name, only one will run.
- Handlers defined inside an include cannot be notified.
- Even if more than one task notifies a handler, the handler will only run once. If no tasks notify it, a handler will not run.
- If a task that includes a notify does not execute (for example, a package is already installed), the handler will not be notified. The handler will be skipped unless another task notifies it. Ansible notifies handlers only if the task acquires the CHANGED status.

Important

Handlers are meant to perform an action upon the execution of a task; they should not be used as a replacement for tasks.

References

Intro to Playbooks — Ansible Documentation
http://docs.ansible.com/ansible/playbooks_intro.html

Guided Exercise: Implementing Handlers

#cd /home/ansible/playbook/dev-handlers

In that directory, use a text editor to create the **configure_db.yml** playbook file. This file will install a database server and create some users; when the database server is installed, the playbook restarts the service.

#cat configure_db.yml

```
---
- name: installing mariadb server
  hosts: dev
  become: yes
  vars:
    db_packages:
      - mariadb-server
      - MySQL-python
    db_service: mariadb
    src_file: "/home/ansible/playbook/dev-handlers/my.cnf"
    dst_file: /etc/my.cnf
  tasks:
    - name: Install {{ db_packages }} package
      yum:
        name: "{{ item }}"
        state: latest
      with_items: "{{ db_packages }}"
      notify:
        - start_service
    - name: Download and Install {{ dst_file }}
      copy:
        src: "{{ src_file }}"
        dest: "{{ dst_file }}"
        owner: ansible
        group: ansible
        force: yes
      notify:
        - restart_service
        - set_password

  handlers:
    - name: start_service
      service:
        name: "{{ db_service }}"
        state: started

    - name: restart_service
      service:
        name: "{{ db_service }}"
        state: restarted
    - name: set_password
      mysql_user:
        name: root
        password: redhat
```

#ansible-playbook --syntax-check configure_db.yml
playbook: configure_db.yml

#ansible-playbook configure_db.yml

```
[ansible@robo dev-handlers]$ ansible-playbook configure_db.yml
SUDO password:

PLAY [installing mariadb server] ***********************************************************************************************************

TASK [Gathering Facts] *********************************************************************************************************************
ok: [robo2.0]

TASK [Install [u'mariadb-server', u'MySQL-python'] package] ********************************************************************************
changed: [robo2.0] => (item=[u'mariadb-server', u'MySQL-python'])

TASK [Download and Install /etc/my.cnf] ****************************************************************************************************
changed: [robo2.0]

RUNNING HANDLER [start_service] ************************************************************************************************************
fatal: [robo2.0]: FAILED! => {"changed": false, "msg": "Unable to start service mariadb: Job for mariadb.service failed because the
ntrol process exited with error code. See \"systemctl status mariadb.service\" and \"journalctl -xe\" for details.\n"}

RUNNING HANDLER [restart_service] **********************************************************************************************************

RUNNING HANDLER [set_password] *************************************************************************************************************

NO MORE HOSTS LEFT *************************************************************************************************************************
        to retry, use: --limit @/home/ansible/playbook/dev-handlers/configure_db.retry

PLAY RECAP *********************************************************************************************************************************
robo2.0                    : ok=3    changed=2    unreachable=0    failed=1

[ansible@robo dev-handlers]$
```

Note: - See above output, maria db won't start on client machine and it will show as failed. Kindly ignore the msg.

Tagging Ansible Resources

Sometimes it is useful to be able to run subsets of the tasks in a playbook. Tags can be applied to specific resources as a text label in order to allow this. Tagging a resource only requires that the tags keyword be used, followed by a list of tags to apply. When plays are tagged, the --tags option can be used with ansible-playbook to filter the playbook to only execute specific tagged plays. Tags are available for the following resources:

Important

When roles or include statements are tagged, the tag is not a way to exclude some of the tagged tasks the included files contain. Tags in this context are a way to apply a global tag to all tasks.

Special Tags

Ansible has a special tag that can be assigned in a playbook: always. This tag causes the task to always be executed even if the **--skip-tags** option is used, unless explicitly skipped with --skip-tags always.

There are three special tags that can be used from the command-line with the --tags option:

1. The **tagged** keyword is used to run any tagged resource.
2. The **untagged** keyword does the opposite of the tagged keyword by excluding all tagged resources from the play.
3. The **all** keyword allows administrators to include all tasks in the play. This is the default behavior of the command line.

References

Tags — Ansible Documentation
http://docs.ansible.com/ansible/playbooks_tags.html
Task And Handler Organization For A Role — Best Practices — Ansible Documentation
http://docs.ansible.com/ansible/playbooks_best_practices.html#task-and-handlerorganization-for-a-role

Guided Exercise: Implementing Tags

#cd /home/ansible/playbook/dev-tags

The following steps will edit the same **configure_mail.yml** task file.
In the project directory, create the **configure_mail.yml** task file. The task file contains instructions to install the required packages for the mail server, as well as instructions to retrieve the configuration files for the mail server.

 # cat configure_mail.yml

```
---
- name: Install Postfix
  yum:
    name: postfix
    state: latest
  tags:
    - server
  notify:
    - start_postfix

- name: Install dovecot
  yum:
    name: dovecot
    state: latest
  tags:
    - client
  notify:
    - start_dovecot

- name: copy the main.cf configuration
  copy:
    src: /home/ansible/playbook/dev-tags/main.cf
    dest: /etc/postfix/main.cf
  tags:
    - server
  notify:
    - restart_postfix
```

Create a playbook file named **playbook.yml**. Define the playbook for all hosts. The playbook should read as follows:

cat playbook.yml

```
---
- hosts: dev
  become: yes
  tasks:
   - name: Include configure mail
     include:
       configure_mail.yml
     when: inventory_hostname in groups['dev']

  handlers:
    - name: start_postfix
      service:
        name: postfix
        state: started

    - name: start_dovecot
      service:
        name: dovecot
        state: started

    - name: restart_postfix
      service:
        name: postfix
        state: restarted
```

#ansible-playbook --syntax-check playbook.yml
playbook: playbook.yml

ansible-playbook playbook.yml

```
[ansible@robo dev-tags]$ ansible-playbook playbook.yml
SUDO password:

PLAY [dev] *********************************************************************************************************************************

TASK [Gathering Facts] *********************************************************************************************************************
ok: [robo2.0]

TASK [Install Postfix] *********************************************************************************************************************
changed: [robo2.0]

TASK [Install dovecot] *********************************************************************************************************************
changed: [robo2.0]

TASK [copy the main.cf configuration] ******************************************************************************************************
ok: [robo2.0]

RUNNING HANDLER [start_postfix] ************************************************************************************************************
fatal: [robo2.0]: FAILED! => {"changed": false, "msg": "Unable to start service postfix: Job for postfix.service failed because
ntrol process exited with error code. See \"systemctl status postfix.service\" and \"journalctl -xe\" for details.\n"}

RUNNING HANDLER [start_dovecot] ************************************************************************************************************

NO MORE HOSTS LEFT *************************************************************************************************************************
        to retry, use: --limit @/home/ansible/playbook/dev-tags/playbook.retry

PLAY RECAP *********************************************************************************************************************************
robo2.0                    : ok=4    changed=2    unreachable=0    failed=1
```

Note:- Ignore above error, because playbook is success but due to invalid configuration file, its throwing an error.
:- Please cleanup once playbook executed successfully.

IMPLEMENTING TASK CONTROL

Handling Errors

Errors in Plays: -

Ansible evaluates the return code of each task to determine whether the task succeeded or failed. Normally, when a task fails Ansible immediately aborts the rest of the play on that host, skipping all subsequent tasks.
However, sometimes you may want to have play execution continue even if a task fails. For example, you might expect that a particular task could fail, and a you might want to recover by running some other task conditionally. There are several Ansible features that can be used to manage task errors.

Ignoring Task Failure: -

By default, if a task fails, the play is aborted. However, this behavior can be overridden by ignoring failed tasks. To do so, the **ignore_errors** keyword needs to be used in a task.
The following snippet shows how to use **ignore_errors** on a task to continue playbook execution on the host even if the task fails. For example, if the package does not exist the yum module will fail but having **ignore_errors** set to yes will allow execution to continue.

Forcing Execution of Handlers after Task Failure:-

Normally when a task fails and the play aborts on that host, any handlers which had been notified by earlier tasks in the play will not run. If you set the **force_handlers:** yes directive on the play, then notified handlers will be called even if the play aborted because a later task failed.
The following snippet shows how to use the **force_handlers** keyword in a play to forcefully execute the handler even if a task fails:

Example: -

```
---
- hosts: all
  force_handlers: yes
  tasks:
    - name: a task which always notifies its handler
      command: /bin/true
      notify: restart the database

    - name: a task which fails because the package doesn't exist
      yum:
        name: notapkg
        state: latest

  handlers:
    - name: restart the database
      service:
        name: mariadb
        state: restarted
```

Note: -

Remember that handlers are notified when a task reports a "changed" result but are not notified when it reports an "ok" or "failed" result.

Specifying Task Failure Conditions:-

You can use the **failed_when** directive on a task to specify which conditions indicate that the task has failed. This is often used with "run command" modules that may successfully execute a command, but the command's output or exit code may indicate a failure.

For example, you can run a script that outputs an error message and use that message to define the failed state for the task. The following snippet shows how the **failed_when** keyword can be used in a task:

```
tasks:
  - shell: /usr/local/bin/create_users.sh
    register: command_result
    failed_when: "'Password missing' in command_result.stdout"
```

Specifying When a Task Reports "Changed" Results:-

When a task makes a change to a managed host, it reports the changed state and notifies handlers. When a task does not need to make a change, it reports ok and does not notify handlers.

The **changed_when** directive can be used to control when a task reports that it has changed. For example, the shell module in the next example is being used to get a Kerberos credential which will be used by subsequent tasks. It normally would always report "changed" when it runs. To suppress that change, **changed_when**: false is set so that it only reports "ok" or "failed".

```
- name: get Kerberos credentials as "admin"
  shell: echo "{{ krb_admin_pass }}" | kinit -f admin
  changed_when: false
```

Ansible Blocks and Error Handling:-

In playbooks, blocks are clauses that logically group tasks, and can be used to control how tasks are executed. For example, a task block can have a when directive to apply a conditional to multiple tasks:

```
- name: block example
  hosts: all
  tasks:
    - 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"
```

Blocks also allow for error handling in combination with the rescue and always statements. If any task in a block fails, tasks in its rescue block were executed in order to recover. After the tasks in the block and possibly the rescue run, then tasks in its always block run. To summarize:

- **block:** Defines the main tasks to run.
- **rescue:** Defines the tasks that will be run if the tasks defined in the block clause fails.
- **always:** Defines the tasks that will always run independently of the success or failure of tasks defined in the block and rescue clauses.

References

Error Handling in Playbooks — Ansible Documentation
http://docs.ansible.com/ansible/playbooks_error_handling.html
Error Handling — Blocks — Ansible Documentation
http://docs.ansible.com/ansible/playbooks_blocks.html#error-handling

Guided Exercise: Handling Errors

Create the **playbook.yml** playbook, which contains a play with two tasks. The first task is written with a deliberate error that will cause it to fail.
Open the playbook in a text editor. Define three variables: **web_package** with a value of http, **db_package** with a value of **mariadb-server** and **db_service** with a value of mariadb. The variables will be used to install the required packages and start the server.

Update the first task to ignore any errors by adding the **ignore_errors** keyword. The tasks should read as follows:
#cd /home/ansible/playbook/dev-failures

#cat ignore_error_playbook.yml

```
---
- hosts: dev
  become: yes
  vars:
    web_package: http
    db_package: mariadb-server
    db_service: mariadb
  tasks:
    - name: Install {{ web_package }} package
      yum:
        name: "{{ web_package }}"
        state: latest
      ignore_errors: yes

    - name: Install {{ db_package }} package
      yum:
        name: "{{ db_package }}"
        state: latest
```

#ansible-playbook --syntax-check ignore_error_playbook.yml
#ansible-playbook ignore_error_playbook.yml

```
[ansible@robo dev-failures]$ ansible-playbook ignore_error_playbook.yml
SUDO password:

PLAY [dev] *********************************************************************************************************************************

TASK [Gathering Facts] *********************************************************************************************************************
ok: [robo2.0]

TASK [Install http package] ****************************************************************************************************************
fatal: [robo2.0]: FAILED! => {"changed": false, "msg": "No package matching 'http' found available, installed or updated", "rc": 126
"results": ["No package matching 'http' found available, installed or updated"]}
...ignoring

TASK [Install mariadb-server package] ******************************************************************************************************
changed: [robo2.0]

PLAY RECAP *********************************************************************************************************************************
robo2.0                    : ok=3    changed=1    unreachable=0    failed=0

[ansible@robo dev-failures]$
```

Even though the first task failed, Ansible executed the second one.

In this step, we'll set up a **block** directive, so you can experiment with how they work.

cat block_rescue_playbook.yml

```
---
- hosts: dev
  become: yes
  vars:
    web_package: http
    db_package: mariadb-server
    db_service: mariadb
  tasks:
    - block:
        - name: Install {{ web_package }} package
          yum:
            name: "{{ web_package }}"
            state: latest
      rescue:
        - name: Install {{ db_package }} package
          yum:
            name: "{{ db_package }}"
            state: latest
      always:
        - name: Start {{ db_service }} service
          service:
            name: "{{ db_service }}"
            state: started
```

ansible-playbook --syntax-check block_rescue_playbook.yml
ansible-playbook block_rescue_playbook.yml

```
[ansible@robo dev-failures]$ ansible-playbook block_rescue_playbook.yml
SUDO password:

PLAY [dev] *********************************************************************************************************************************************

TASK [Gathering Facts] *********************************************************************************************************************************
ok: [robo2.0]

TASK [Install http package] ****************************************************************************************************************************
fatal: [robo2.0]: FAILED! => {"changed": false, "msg": "No package matching 'http' found available, installed or updated", "rc": 126,
"results": ["No package matching 'http' found available, installed or updated"]}

TASK [Install mariadb-server package] ******************************************************************************************************************

changed: [robo2.0]

TASK [Start mariadb service] ***************************************************************************************************************************
changed: [robo2.0]

PLAY RECAP *********************************************************************************************************************************************
robo2.0                    : ok=3    changed=2    unreachable=0    failed=1
```

Edit the playbook, correcting the value of the web_package variable to read **httpd**.
That will cause the task in the block to succeed the next time we run the playbook.

```
---
- hosts: dev
  become: yes
  vars:
    web_package: httpd
    db_package: mariadb-server
    db_service: mariadb
```

ansible-playbook block_rescue_playbook.yml

```
[ansible@robo dev-failures]$ ansible-playbook block_rescue_playbook.yml
SUDO password:

PLAY [dev] *********************************************************************

TASK [Gathering Facts] *********************************************************
ok: [robo2.0]

TASK [Install httpd package] ***************************************************
changed: [robo2.0]

TASK [Start mariadb service] ***************************************************
ok: [robo2.0]

PLAY RECAP *********************************************************************
robo2.0                    : ok=3    changed=1    unreachable=0    failed=0
```

Edit the playbook to add two tasks to the start of the play, preceding the block. The first task will use the command module to run the date command and register the result in the **command_result** variable. The second task will use the debug module to print the standard output of the first task's command.

```
- name: Check local time
  command: date
  register: command_result
- name: Print local time
  debug:
    var: command_result["stdout"]
```

#ansible-playbook block_rescue_playbook.yml

```
[ansible@robo dev-failures]$ ansible-playbook block_rescue_playbook.yml
SUDO password:

PLAY [dev] *********************************************************************

TASK [Gathering Facts] *********************************************************
ok: [robo2.0]

TASK [Check local time] ********************************************************
changed: [robo2.0]

TASK [Print local time] ********************************************************
ok: [robo2.0] => {
    "command_result[\"stdout\"]": "Wed Apr 17 21:15:31 IST 2019"
}

TASK [Install httpd package] ***************************************************
ok: [robo2.0]

TASK [Start mariadb service] ***************************************************
ok: [robo2.0]

PLAY RECAP *********************************************************************
robo2.0                    : ok=5    changed=1    unreachable=0    failed=0
```

That **command** task shouldn't report "changed" every time it runs because it's not changing the managed host. Since you know that the task will never change a managed host, add the line **changed_when**: false to the task to suppress the change.

```
- name: Check local time
  command: date
  register: command_result
  changed_when: false
```

ansible-playbook block_rescue_playbook.yml

```
[ansible@robo dev-failures]$ ansible-playbook block_rescue_playbook.yml
SUDO password:

PLAY [dev] *********************************************************************

TASK [Gathering Facts] *********************************************************
ok: [robo2.0]

TASK [Check local time] ********************************************************
ok: [robo2.0]

TASK [Print local time] ********************************************************
ok: [robo2.0] => {
    "command_result[\"stdout\"]": "Wed Apr 17 21:21:23 IST 2019"
}

TASK [Install httpd package] ***************************************************
ok: [robo2.0]

TASK [Start mariadb service] ***************************************************
ok: [robo2.0]

PLAY RECAP *********************************************************************
robo2.0                    : ok=5    changed=0    unreachable=0    failed=0
```

As a final exercise, edit the playbook to explore how the **failed_when** directive interacts with tasks.
Edit the "Install {{ web_package }} package" task so that it reports as having failed when **web_package** has the value **httpd**. Since this is the case, the task will report failure when we run the play.

```
- name: Install {{ web_package }} package
  yum:
    name: "{{ web_package }}"
    state: latest
  failed_when: web_package == "httpd"
```

ansible-playbook block_rescue_playbook.yml

```
[ansible@robo dev-failures]$ ansible-playbook block_rescue_playbook.yml
SUDO password:

PLAY [dev] *********************************************************************

TASK [Gathering Facts] *********************************************************
ok: [robo2.0]

TASK [Check local time] ********************************************************
ok: [robo2.0]

TASK [Print local time] ********************************************************
ok: [robo2.0] => {
    "command_result[\"stdout\"]": "Wed Apr 17 21:31:07 IST 2019"
}

TASK [Install httpd package] ***************************************************
fatal: [robo2.0]: FAILED! => {"changed": false, "failed_when_result": true, "msg": "", "rc": 0, "results": ["All packages providing ht
tpd are up to date", ""]}

TASK [Install mariadb-server package] ******************************************
ok: [robo2.0]

TASK [Start mariadb service] ***************************************************
ok: [robo2.0]

PLAY RECAP *********************************************************************
robo2.0                    : ok=5    changed=0    unreachable=0    failed=1
```

Look carefully at the output. The "Install httpd package" task reports that it failed, but it actually ran and made sure the package is installed first! The failed_when directivechanges the status the task reports after the task runs, it does not change the behaviour of the task itself.However, the failure reported might change the behavior of the rest of the play.Since that task was in a block and reported that it failed, the "Install mariadb-server package" task in the block's rescue section was run.