**Ignoring Failed Commands**

New in version 0.6.

Generally playbooks will stop executing any more steps on a host that has a task fail. Sometimes, though, you want to continue on. To do so, write a task that looks like this:

- name: this will not be counted as a failure

command: /bin/false

ignore\_errors: yes

Note that the above system only governs the return value of failure of the particular task, so if you have an undefined variable used or a syntax error, it will still raise an error that users will need to address. Note that this will not prevent failures on connection or execution issues. This feature only works when the task must be able to run and return a value of ‘failed’.

## Error Handling

Blocks also introduce the ability to handle errors in a way similar to exceptions in most programming languages.

Block error handling example¶

tasks:

- name: Attempt and gracefull roll back demo

block:

- debug: msg='I execute normally'

- command: /bin/false

- debug: msg='I never execute, due to the above task failing'

rescue:

- debug: msg='I caught an error'

- command: /bin/false

- debug: msg='I also never execute :-('

always:

- debug: msg="this always executes"

The tasks in the block would execute normally, if there is any error the rescue section would get executed with whatever you need to do to recover from the previous error. The always section runs no matter what previous error did or did not occur in the block and rescuesections.

Another example is how to run handlers after an error occurred :

Block run handlers in error handling¶

tasks:

- name: Attempt and gracefull roll back demo

block:

- debug: msg='I execute normally'

notify: run me even after an error

- command: /bin/false

rescue:

- name: make sure all handlers run

meta: flush\_handlers

handlers:

- name: run me even after an error

debug: msg='this handler runs even on error'

**Resetting Unreachable Hosts**

New in version 2.2.

Connection failures set hosts as ‘UNREACHABLE’, which will remove them from the list of active hosts for the run. To recover from these issues you can use *meta: clear\_host\_errors* to have all currently flagged hosts reactivated, so subsequent tasks can try to use them again.

**Handlers and Failure**

New in version 1.9.1.

When a task fails on a host, handlers which were previously notified will *not* be run on that host. This can lead to cases where an unrelated failure can leave a host in an unexpected state. For example, a task could update a configuration file and notify a handler to restart some service. If a task later on in the same play fails, the service will not be restarted despite the configuration change.

You can change this behavior with the --force-handlers command-line option, or by including force\_handlers: True in a play, or force\_handlers = True in ansible.cfg. When handlers are forced, they will run when notified even if a task fails on that host. (Note that certain errors could still prevent the handler from running, such as a host becoming unreachable.)

**Controlling What Defines Failure**

New in version 1.4.

Suppose the error code of a command is meaningless and to tell if there is a failure what really matters is the output of the command, for instance if the string “FAILED” is in the output.

Ansible in 1.4 and later provides a way to specify this behavior as follows:

- name: Fail task when the command error output prints FAILED

command: /usr/bin/example-command -x -y -z

register: command\_result

failed\_when: "'FAILED' in command\_result.stderr"

or based on the return code:

- name: Fail task when both files are identical

raw: diff foo/file1 bar/file2

register: diff\_cmd

failed\_when: diff\_cmd.rc == 0 or diff\_cmd.rc >= 2

In previous version of Ansible, this can be still be accomplished as follows:

- name: this command prints FAILED when it fails

command: /usr/bin/example-command -x -y -z

register: command\_result

ignore\_errors: True

- name: fail the play if the previous command did not succeed

fail: msg="the command failed"

when: "'FAILED' in command\_result.stderr"

**Overriding The Changed Result**

New in version 1.3.

When a shell/command or other module runs it will typically report “changed” status based on whether it thinks it affected machine state.

Sometimes you will know, based on the return code or output that it did not make any changes, and wish to override the “changed” result such that it does not appear in report output or does not cause handlers to fire:

tasks:

- shell: /usr/bin/billybass --mode="take me to the river"

register: bass\_result

changed\_when: "bass\_result.rc != 2"

*# this will never report 'changed' status*

- shell: wall 'beep'

changed\_when: False

**Aborting the play**

Sometimes it’s desirable to abort the entire play on failure, not just skip remaining tasks for a host.

The any\_errors\_fatal play option will mark all hosts as failed if any fails, causing an immediate abort:

- hosts: somehosts

any\_errors\_fatal: true

roles:

- myrole