Error handling in playbooks

When Ansible receives a non-zero return code from a command or a failure from a module, by default it stops executing on that host and continues on other hosts. However, in some circumstances you may want different behavior. Sometimes a non-zero return code indicates success. Sometimes you want a failure on one host to stop execution on all hosts. Ansible provides tools and settings to handle these situations and help you get the behavior, output, and reporting you want.

Ignoring failed commands

By default Ansible stops executing tasks on a host when a task fails on that host. You can use `ignore_errors` to continue on in spite of the failure.

```
- name: Do not count this as a failure
  ansible.builtin.command: /bin/false
  ignore_errors: yes
```

The `ignore_errors` directive only works when the task is able to run and returns a value of 'failed'. It does not make Ansible ignore undefined variable errors, connection failures, execution issues (for example, missing packages), or syntax errors.

Ignoring unreachable host errors

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst][user_guide]playbooks_error_handling.rst`, line 32)

Unknown directive type "versionadded".

```
.. versionadded:: 2.7
```

You can ignore a task failure due to the host instance being 'UNREACHABLE' with the `ignore_unreachable` keyword. Ansible ignores the task errors, but continues to execute future tasks against the unreachable host. For example, at the task level:

```
- name: This executes, fails, and the failure is ignored
  ansible.builtin.command: /bin/true
  ignore_unreachable: yes

- name: This executes, fails, and ends the play for this host
  ansible.builtin.command: /bin/true
```

And at the playbook level:

```
- hosts: all
  ignore_unreachable: yes
  tasks:
  - name: This executes, fails, and the failure is ignored
    ansible.builtin.command: /bin/true

  - name: This executes, fails, and ends the play for this host
    ansible.builtin.command: /bin/true
    ignore_unreachable: no
```

Resetting unreachable hosts

If Ansible cannot connect to a host, it marks that host as 'UNREACHABLE' and removes it from the list of active hosts for the run. You can use *meta: clear_host_errors* to reactivate all hosts, so subsequent tasks can try to reach them again.

Handlers and failure

Ansible runs :ref:`handlers <handlers>` at the end of each play. If a task notifies a handler but another task fails later in the play, by default the handler does *not* run on that host, which may leave the 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 in the same play fails, the configuration file might be changed but the service will not be restarted.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst][user_guide]playbooks_error_handling.rst`, line 72); *backlink*

Unknown interpreted text role "ref".

You can change this behavior with the `--force-handlers` command-line option, by including `force_handlers: True` in a play, or by adding `force_handlers = True` to ansible.cfg. When handlers are forced, Ansible will run all notified handlers on all hosts, even hosts with failed tasks. (Note that certain errors could still prevent the handler from running, such as a host becoming unreachable.)

Defining failure

Ansible lets you define what "failure" means in each task using the `failed_when` conditional. As with all conditionals in Ansible, lists of multiple `failed_when` conditions are joined with an implicit `and`, meaning the task only fails when *all* conditions are met. If you want to trigger a failure when any of the conditions is met, you must define the conditions in a string with an explicit `or` operator.

You may check for failure by searching for a word or phrase in the output of a command

```
- name: Fail task when the command error output prints FAILED
  ansible.builtin.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
  ansible.builtin.raw: diff foo/file1 bar/file2
  register: diff_cmd
  failed_when: diff_cmd.rc == 0 or diff_cmd.rc >= 2
```

You can also combine multiple conditions for failure. This task will fail if both conditions are true:

```
- name: Check if a file exists in temp and fail task if it does
  ansible.builtin.command: ls /tmp/this_should_not_be_here
  register: result
  failed_when:
    - result.rc == 0
    - '"No such" not in result.stdout'
```

If you want the task to fail when only one condition is satisfied, change the `failed_when` definition to

```
failed_when: result.rc == 0 or "No such" not in result.stdout
```

If you have too many conditions to fit neatly into one line, you can split it into a multi-line YAML value with `>`.

```
- name: example of many failed_when conditions with OR
  ansible.builtin.shell: "./myBinary"
  register: ret
  failed_when: >
    ("No such file or directory" in ret.stdout) or
    (ret.stderr != '') or
    (ret.rc == 10)
```

Defining "changed"

Ansible lets you define when a particular task has "changed" a remote node using the `changed_when` conditional. This lets you determine, based on return codes or output, whether a change should be reported in Ansible statistics and whether a handler should be triggered or not. As with all conditionals in Ansible, lists of multiple `changed_when` conditions are joined with an implicit `and`, meaning the task only reports a change when *all* conditions are met. If you want to report a change when any of the conditions is met, you must define the conditions in a string with an explicit `or` operator. For example:

```
tasks:

  - name: Report 'changed' when the return code is not equal to 2
```

```
    ansible.builtin.shell: /usr/bin/billybass --mode="take me to the river"
    register: bass_result
    changed_when: "bass_result.rc != 2"

  - name: This will never report 'changed' status
    ansible.builtin.shell: wall 'beep'
    changed_when: False
```

You can also combine multiple conditions to override "changed" result.

```
- name: Combine multiple conditions to override 'changed' result
  ansible.builtin.command: /bin/fake_command
  register: result
  ignore_errors: True
  changed_when:
    - '"ERROR" in result.stderr'
    - result.rc == 2
```

> **Note**
>
> Just like `when` these two conditionals do not require templating delimiters (`{{ }}`) as they are implied.

See :ref:`controlling_what_defines_failure` for more conditional syntax examples.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst][user_guide]playbooks_error_handling.rst`, line 175); *backlink***
>
> Unknown interpreted text role "ref".

Ensuring success for command and shell

The :ref:`command <command_module>` and :ref:`shell <shell_module>` modules care about return codes, so if you have a command whose successful exit code is not zero, you can do this:

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst][user_guide]playbooks_error_handling.rst`, line 180); *backlink***
>
> Unknown interpreted text role "ref".

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst][user_guide]playbooks_error_handling.rst`, line 180); *backlink***
>
> Unknown interpreted text role "ref".

```
tasks:
  - name: Run this command and ignore the result
    ansible.builtin.shell: /usr/bin/somecommand || /bin/true
```

Aborting a play on all hosts

Sometimes you want a failure on a single host, or failures on a certain percentage of hosts, to abort the entire play on all hosts. You can stop play execution after the first failure happens with `any_errors_fatal`. For finer-grained control, you can use `max_fail_percentage` to abort the run after a given percentage of hosts has failed.

Aborting on the first error: any_errors_fatal

If you set `any_errors_fatal` and a task returns an error, Ansible finishes the fatal task on all hosts in the current batch, then stops executing the play on all hosts. Subsequent tasks and plays are not executed. You can recover from fatal errors by adding a :ref:`rescue section <block_error_handling>` to the block. You can set `any_errors_fatal` at the play or block level.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst][user_guide]playbooks_error_handling.rst`, line 197); *backlink***
>
> Unknown interpreted text role "ref".

```
- hosts: somehosts
  any_errors_fatal: true
  roles:
    - myrole

- hosts: somehosts
  tasks:
    - block:
        - include_tasks: mytasks.yml
      any_errors_fatal: true
```

You can use this feature when all tasks must be 100% successful to continue playbook execution. For example, if you run a service on machines in multiple data centers with load balancers to pass traffic from users to the service, you want all load balancers to be disabled before you stop the service for maintenance. To ensure that any failure in the task that disables the load balancers will stop all other tasks:

```
---
- hosts: load_balancers_dc_a
  any_errors_fatal: true

  tasks:
    - name: Shut down datacenter 'A'
      ansible.builtin.command: /usr/bin/disable-dc

- hosts: frontends_dc_a

  tasks:
    - name: Stop service
      ansible.builtin.command: /usr/bin/stop-software

    - name: Update software
      ansible.builtin.command: /usr/bin/upgrade-software

- hosts: load_balancers_dc_a

  tasks:
    - name: Start datacenter 'A'
      ansible.builtin.command: /usr/bin/enable-dc
```

In this example Ansible starts the software upgrade on the front ends only if all of the load balancers are successfully disabled.

Setting a maximum failure percentage

By default, Ansible continues to execute tasks as long as there are hosts that have not yet failed. In some situations, such as when executing a rolling update, you may want to abort the play when a certain threshold of failures has been reached. To achieve this, you can set a maximum failure percentage on a play:

```
---
- hosts: webservers
  max_fail_percentage: 30
  serial: 10
```

The `max_fail_percentage` setting applies to each batch when you use it with :ref:`serial <rolling_update_batch_size>`. In the example above, if more than 3 of the 10 servers in the first (or any) batch of servers failed, the rest of the play would be aborted.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst][user_guide]playbooks_error_handling.rst`, line 255); *backlink***
>
> Unknown interpreted text role "ref".

> **Note**
>
> The percentage set must be exceeded, not equaled. For example, if serial were set to 4 and you wanted the task to abort the play when 2 of the systems failed, set the max_fail_percentage at 49 rather than 50.

Controlling errors in blocks

You can also use blocks to define responses to task errors. This approach is similar to exception handling in many programming languages. See :ref:`block_error_handling` for details and examples.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst]`

[user_guide]playbooks_error_handling.rst, line 264); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst][user_guide]playbooks_error_handling.rst, line 266)

Unknown directive type "seealso".

```
.. seealso::

   :ref:`playbooks_intro`
       An introduction to playbooks
   :ref:`playbooks_best_practices`
       Tips and tricks for playbooks
   :ref:`playbooks_conditionals`
       Conditional statements in playbooks
   :ref:`playbooks_variables`
       All about variables
   `User Mailing List <https://groups.google.com/group/ansible-devel>`_
       Have a question?  Stop by the google group!
   :ref:`communication_irc`
       How to join Ansible chat channels
```