Automation

Ansible Conditionals and Loops

DE HOGESCHOOL MET HET NETWERK

Elfde-Liniestraat 24, 3500 Hasselt, www.pxl.be

Conditionals

Types of Ansible Conditionals

- **When Conditionals**: most common conditionals in Ansible - they specify conditions that must be met for a task to be executed.
- **Failed Conditionals**: specify tasks that should be executed if a previous task fails.
- **Changed Conditionals**: specify tasks that should be executed if a previous task has made changes to the system.
- **Custom Conditionals**: conditionals that you define yourself using Ansible's Jinja2 template language.

```
- name: Do something if myvar is set
  command: /path/to/some/command
  when: myvar == "somevalue"
```

Understanding Ansible Conditionals

- Ansible conditionals are used to make decisions in playbooks based on certain conditions.
- For example, you might want to run a task only if a certain variable is set to a specific value.
- Conditionals are expressed using the **when** keyword.

```
- name: Install Apache if it's not already installed
  ansible.builtin.package:
    name: apache2
    state: present
  when: "'apache2' not in ansible_facts.packages"
```

Ansible Conditionals operators

- Comparison operators (e.g., ==, >, <, !=)
- Logical operators (e.g., **and**, **or**, **not**)
- Regular expression matching (e.g., =~)

```
tasks:
 - name: Shut down CentOS 6 and Debian 7 systems
   ansible.builtin.command: /sbin/shutdown -t now
   when: (ansible_facts['distribution'] == "CentOS" and ansible_facts['distribution_major_version'] == "6") or
         (ansible_facts['distribution'] == "Debian" and ansible_facts['distribution_major_version'] == "7")
```

Implicit logical operator **and**

Specify multiple conditions that all need to be true as a list (== a logical and)

```yaml
tasks:
 - name: Shut down CentOS 6 systems
   ansible.builtin.command: /sbin/shutdown -t now
   when:
     - ansible_facts['distribution'] == "CentOS"
     - ansible_facts['distribution_major_version'] == "6"
```

Testing dictionary membership with `in`

example playbook to install vsftpd (very secure ftp daemon) on ftp servers.

```
---
- name: Install vsftpd on ftpservers
  hosts: all
  become: yes
  tasks:
    - name: Install FTP server when host in ftpserver group
      yum:
        name: vsftpd
        state: latest
      when: inventory_hostname in groups["ftpserver"]
```

Jinja2 templates and filters

- Filters let you transform JSON data into YAML data, split a URL to extract the hostname, get the SHA1 hash of a string, add or multiply integers, etc.
- List of built-in filters in the official Jinja2 template documentation.
- Providing default values
  - `{{ some_variable | default(5) }}`
  - `{{ lookup('env', 'MY_USER') | default('admin', true) }}`
- Casting data types
  - `when: some_string_value | bool`
  - `when: ansible_facts['lsb']['major_release'] | int >= 6`

Using registered variables

A registered variable is global, registered with the keywork **`register`**, and contains the status and the output of the task that created it.

Use **`variable.stdout`** to access the string contents of the variables.

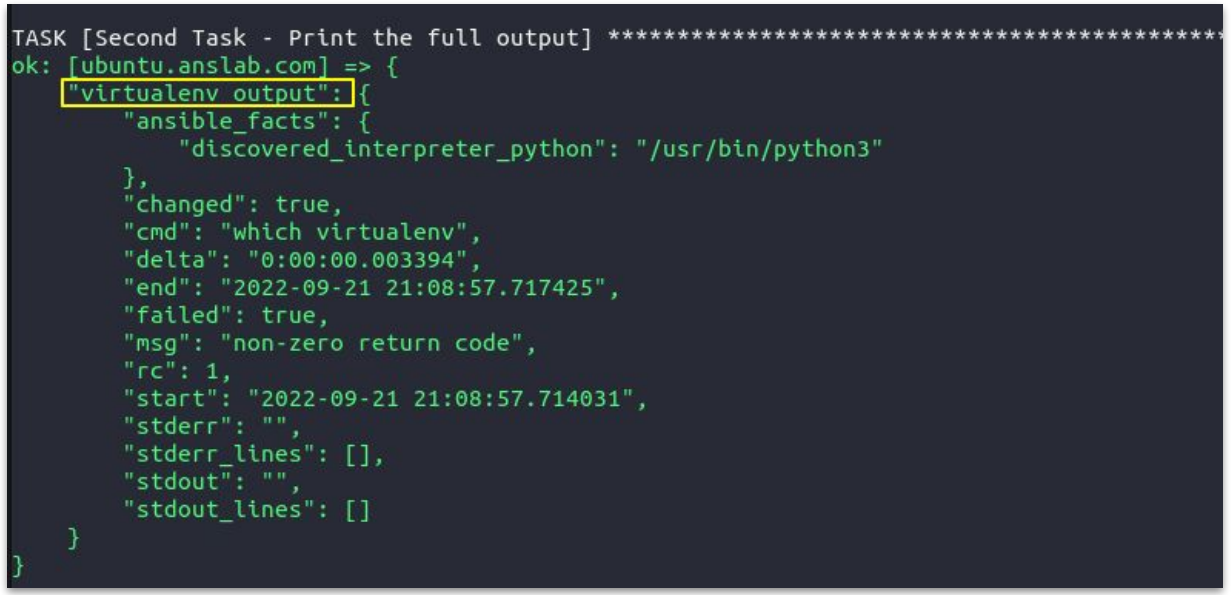
```
tasks:
      - name: Register a variable
        ansible.builtin.shell: cat /etc/motd
        register: motd_contents

      - name: Use the variable in conditional statement
        ansible.builtin.shell: echo "motd contains the word hi"
        when: motd_contents.stdout.find('hi') != -1
```

Using registered variables

```
---
- name: Ansible conditional with a registered variable example
  hosts: web
  remote_user: ubuntu
  become: true
  tasks:
# to list the directory content in '/etc/hosts directory'
      - name: List contents of directory and Store in content1
        ansible.builtin.command: ls /etc/hosts
        register: contents1
# to list the directory content in '/home/ubuntu/hello'
      - name: List contents of directory and Store in content2
        ansible.builtin.command: ls /home/ubuntu/hello
        register: contents2
# display Directory is empty if the directories
# '/etc/hosts' or '/home/ubuntu/hello' are empty
      - name: Check contents for emptiness for content1 or content2
        ansible.builtin.debug:
          msg: "Directory is empty"
        when: contents1.stdout == "" or contents2.stdout == ""
```

Return values of registered variables

- Ansible modules return a data structure that can be registered into a variable, or seen directly when output by the ansible program.
- Each module can optionally document its own unique return values.
- https://docs.ansible.com/ansible/latest/reference_appendices/common_return_values.html



Use the status of registered variables: `failed`, `succeeded`, `skipped`, `changed`

```yaml
tasks:
 - name: Register a variable, ignore errors and continue
   ansible.builtin.command: /bin/false
   register: result
   ignore_errors: true

 - name: Run only if the task that registered the "result" variable fails
   ansible.builtin.command: /bin/something
   when: result is failed

 - name: Run only if the task that registered the "result" variable succeeds
   ansible.builtin.command: /bin/something_else
   when: result is succeeded

 - name: Run only if the task that registered the "result" variable is skipped
   ansible.builtin.command: /bin/still/something_else
   when: result is skipped

 - name: Run only if the task that registered the "result" variable changed something.
   ansible.builtin.command: /bin/still/something_else
   when: result is changed
```

Example of `failed`, `succeeded`

```yaml
---
- hosts: all
  vars:
    - user: angie
  tasks:
    - name: Check if file already exists
      command: ls /home/{{ user }}/myfile
      register: file_exists
      ignore_errors: true

    - name: create file for user
      file:
        path: /home/{{ user }}/myfile
        state: touch
      when: file_exists is failed

    - name: show message if file exists
      debug:
        msg: The user file already exists.
      when: file_exists is succeeded
```

- this playbook checks if the file `/home/angie/myfile` exists on the target host.
- If the file exists, it displays a message.
- If the file does not exist, it creates the file.
- The **`register`** directive is used to capture the output of the command module, and the **`when`** directive is used to conditionally execute the file and debug modules based on the value of the `file_exists` variable.

`defined` and `undefined`

```yaml
tasks:
  - name: Run the command if "foo" is defined
    ansible.builtin.shell: echo "I've got '{{ foo }}' and am not afraid to use it!"
    when: foo is defined

  - name: Fail if "bar" is undefined
    ansible.builtin.fail: msg="Bailing out. This play requires 'bar'"
    when: bar is undefined
```

`failed_when` and `changed_when`

- Ansible **`failed_when`** and **`changed_when`** statements are similar to ansible **`when`** statement. The only difference is that It will mark the task as failed or success[changed], when the condition defined, is met or satisfied.
- The primary purpose of the **`failed_when`** and **`changed_when`** statements are to determine whether the task is actually successful or results in a failure.

Defining failure

- The `failed_when` conditional lets you define what "failure" means in each task.
- 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, define the conditions in a string with an explicit `or` operator.

```
- name: Fail task when the command error output prints Boom!
  ansible.builtin.command: /usr/bin/example-command -x -y -z
  register: command_result
  failed_when: "'Boom !' in command_result.stderr"
```

```
- 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'
```

```
- 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
```

Tip: splitting conditions over multiple lines with >

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

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

Ignoring failure with `ignore_errors`

- 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.
- Only works when the task is able to run and returns a value of 'failed'.
- Does not make Ansible ignore undefined variable errors, connection failures, execution issues (for example, missing packages), or syntax errors.

```
- name: List a non-existent file
  command: ls non-existent.txt
  ignore_errors: true

- name: continue task after failing ls
  debug:
    msg: "Continue task after failure"
```

Ignoring unreachable host errors with `ignore_unreachable`

- 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 ignore a task failure due to the host instance being 'UNREACHABLE' with the **`ignore_unreachable`** keyword.
- Can be defined on playbook as well as on task level.
- With **`ignore_unreachable`** Ansible ignores the task errors, but continues to execute future tasks against the unreachable host.

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

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

Defining changed

- The **changed_when** conditional lets you define when a particular task has "changed" a remote node.
- This lets you determine, based on return codes or output, whether a change should be reported in Ansible statistics and whether a **handler** *(see later)* should be triggered or not.

```
- name: Report 'changed' when the return code is not equal to 2
  ansible.builtin.shell: /usr/bin/mybinary --mode="analyze"
  register: binary_result
  changed_when: "binary_result.rc != 2"

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

Selecting variables files based on `ansible_facts`

```yaml
---
- hosts: webservers
  remote_user: root
  vars_files:
    - "vars/common.yml"
    - [ "vars/{{ ansible_facts['os_family'] }}.yml", "vars/os_defaults.yml" ]
  tasks:
  - name: Make sure apache is started
    ansible.builtin.service:
      name: '{{ apache }}'
      state: started
```

```yaml
---
# for vars/RedHat.yml
apache: httpd
somethingelse: 42
```

```yaml
---
# for vars/Debian.yml
apache: apache2
somethingelse: 13
```

Grouping tasks with `block:`

- **`block`** in an Ansible playbook is used to group multiple tasks together.
- useful for applying the same attributes or error handling to a set of tasks, making the playbook easier to read and maintain.
- E.g., use a **`block`** to group tasks that should be executed together under a specific condition. You can then apply the **`when`** keyword to the entire block, rather than repeating the condition for each individual task.
- In an Ansible playbook, **`rescue`** is used to specify tasks that should be executed when any task within a **`block`** fails, while **`always`** defines tasks that will be executed after the block and rescue sections, regardless of their success or failure

```yaml
tasks:
  - name: Deploy and configure Nginx
    block:
      - name: Copy Nginx configuration
        ansible.builtin.template:
          src: nginx.conf.j2
          dest: "{{ nginx_config_file }}"
          owner: root
          group: root
          mode: 0644

      - name: Enable site configuration
        ansible.builtin.file:
          src: "{{ nginx_config_file }}"
          dest: /etc/nginx/sites-enabled/{{ app_name }}
          state: link

      - name: Reload Nginx
        ansible.builtin.systemd:
          name: nginx
          state: reloaded
    when: "'web_servers' in group_names"

    rescue:
      - name: Remove faulty configuration
        ansible.builtin.file:
          path: "{{ nginx_config_file }}"
          state: absent

      - name: Rollback Nginx to default site
        ansible.builtin.command: ln -sf /etc/nginx/sites-available/default /etc/nginx/sites-enabled/default
        args:
          removes: /etc/nginx/sites-enabled/{{ app_name }}

      - name: Restart Nginx
        ansible.builtin.systemd:
          name: nginx
          state: restarted

      - name: Fail the playbook
        ansible.builtin.fail:
          msg: "Nginx configuration failed. Rolled back to default site."

    always:
      - name: Check Nginx status
        ansible.builtin.systemd:
          name: nginx
          state: started
```

Loops

Ansible Loops

- Implement loops using **`loop`**, **`with_<lookup>`** (`with_list`, `with_items`, `with_dict`, ...), and **`until`**.
- Using **`loop`** is the recommended way vs **`with_<lookup>`**.
- Access each element of the looped-over list with "**`{{ item }}`**"

```yaml
- name: Add several users
  ansible.builtin.user:
    name: "{{ item }}"
    state: present
    groups: "wheel"
  loop:
    - testuser1
    - testuser2
```

Looping over hashes

```yaml
---
- name: Ensure users
  hosts: node1
  become: true

  tasks:
    - name: Ensure three users are present
      user:
        name: "{{ item.username }}"
        state: present
        groups: "{{ item.groups }}"
      loop:
        - { username: 'dev_user', groups: 'ftp' }
        - { username: 'qa_user', groups: 'ftp' }
        - { username: 'prod_user', groups: 'apache' }
```

Loop example

```yaml
- name: Registered variable usage as a loop list
  hosts: all
  tasks:

    - name: Retrieve the list of home directories
      ansible.builtin.command: ls /home
      register: home_dirs

    - name: Add home dirs to the backup spooler
      ansible.builtin.file:
        path: /mnt/bkspool/{{ item }}
        src: /home/{{ item }}
        state: link
      loop: "{{ home_dirs.stdout_lines }}"
      # same as loop: "{{ home_dirs.stdout.split() }}"
```

- **`home_dirs`** is not a list, so difficult for looping.
- However, the Jijnja method **`.stdout_lines`** returns a list of lines. Now we can use the list for iterating over with **`loop`**.

Looping dictionaries

```yaml
users:
  - name: john
    uid: 1001
    gid: 1001
    password: mypassword
  - name: jane
    uid: 1002
    gid: 1002
    password: myotherpassword
```

```yaml
- name: Import users file
  hosts: myserver
  vars_files:
    - /path/to/group_vars/prod/users.yml
  tasks:
    - name: Create users with properties
      user:
        name: "{{ item.name }}"
        uid: "{{ item.uid }}"
        group: "{{ item.gid }}"
        password: "{{ item.password | password_hash('sha512') }}"
        state: present
      loop: "{{ users }}"
```

Combining Loops and Conditionals

- If you combine a when statement with a loop, Ansible processes the condition separately for each item.
- You can execute the task on some items in the loop and skip it on other items.
- You can just use the keyword **`item`** in the **`when:`** clause.

```
tasks:
   - name: run with items greater than 5
     ansible.builtin.command: echo {{ item }}
     loop: [ 0, 2, 4, 6, 8, 10 ]
     when: item > 5
```

Customize loops with `loop_control`:

- **`loop_control`** is used to modify how the loop executes and provides more control over the loop iteration.
- **`loop_control`** is a dictionary containing the following keys:
  - **`label`**: allows you to customize the label used to identify the item being processed in the loop.
  - **`index_var`**: allows you to specify a variable to hold the index of the current iteration.
  - **`loop_var`**: allows you to specify a variable to hold the value of the current iteration.
  - **`extended`**: allows you to specify additional loop options like **`skip_missing`**, **`stop_execution`**, etc.
  - **`flatten`**: allows you to flatten the items in the loop before iterating over them.
  - **`pause`**: allows you to pause the loop at a specified iteration and resume it later.

```yaml
- name: Loop example
  hosts: localhost
  tasks:
    - name: Print numbers with custom label
      debug:
        msg: "The number is {{ item }}"
      loop:
        - 1
        - 2
        - 3
      loop_control:
        label: "Number {{ item }}"
```

```
TASK [Print numbers with custom label] *************
ok: [localhost] => (label=Number 1) => {
    "msg": "The number is 1"
}
ok: [localhost] => (label=Number 2) => {
    "msg": "The number is 2"
}
ok: [localhost] => (label=Number 3) => {
    "msg": "The number is 3"
}
```

Error handling with `until`, `retries` and `delay`

- **`until`**: a condition that must be met for a task or block of tasks to succeed. If the condition is not met, the task will retry until the condition is met or the maximum number of retries is exceeded.
- **`retries`**: maximum number of times to retry a task or block of tasks if they fail.
- **`delay`**: amount of time to wait between retries.

```yaml
---
- name: Retry until a file is available
  hosts: localhost
  tasks:
    - name: Validate if the file is present
      shell: ls -lrt /tmp/myprocess.pid
      register: lsresult
      until: "lsresult is not failed"
      retries: 10
      delay: 10
```

```
PLAY [Retry until a file is available] ************************************

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

TASK [Validate if the file is present] ************************************
FAILED - RETRYING: Validate if the file is present (10 retries left).
FAILED - RETRYING: Validate if the file is present (9 retries left).
FAILED - RETRYING: Validate if the file is present (8 retries left).
FAILED - RETRYING: Validate if the file is present (7 retries left).
FAILED - RETRYING: Validate if the file is present (6 retries left).
FAILED - RETRYING: Validate if the file is present (5 retries left).
FAILED - RETRYING: Validate if the file is present (4 retries left).
FAILED - RETRYING: Validate if the file is present (3 retries left).
FAILED - RETRYING: Validate if the file is present (2 retries left).
FAILED - RETRYING: Validate if the file is present (1 retries left).
fatal: [localhost]: FAILED! => {"attempts": 10, "changed": true, "cmd": "ls -lrt /tmp/mypro
cess.pid", "delta": "0:00:00.007821", "end": "2022-04-07 02:46:49.676415", "msg": "non-zero
 return code", "rc": 1, "start": "2022-04-07 02:46:49.668594", "stderr": "ls: /tmp/myproces
s.pid: No such file or directory", "stderr_lines": ["ls: /tmp/myprocess.pid: No such file o
r directory"], "stdout": "", "stdout_lines": []}

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

Exercises

Web server

Configure a web server on an ubuntu machine using Ansible.

The web server should have the following features:

- Apache web server installed and running
- PHP installed and configured to work with Apache and mysql
- ubuntu packages: apache2, php, libapache2-mod-php, php-mysql, mysql-server, python3-pip
- try 2 more times if package update fails
- A custom index.php file "<?php phpinfo(); ?>" created in the Apache document root
- The web server should be configured to listen on port 8080 instead of the default port 80
- A MySQL server installed and configured with a database and user
    - install pip file pymysql first. then use the mysql_user module.
- The firewall UFW should be configured to allow incoming traffic on port 8080 and port 3306. Use a loop.

end

~~node.js and postgres~~ too hard/messy

Create an Ansible playbook named `playbook.yml`. In this playbook, you will:

- Update the package cache.
- Install Node.js and npm on the Node.js application rhel 9 server.
- Install PostgreSQL on the PostgreSQL database rhel 9 server.
- Deploy the sample Node.js application app.js on the application server.
- Configure the PostgreSQL server with a new database and user for the Node.js application.
- Start and enable the Node.js application and the PostgreSQL service.

Use variables, conditionals, and loops in the Ansible playbook to accomplish the tasks for each server.

```
const http = require('http');
const hostname = '0.0.0.0';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at
http://${hostname}:${port}/`);
});
```

end