I'm not robot
reCAPTCHA
Continue

Ansible template iterate array

Referring to template variables is a great way to keep your ansible playbook logic separate from the data. When a variable's value changes, you simply mirror this change in the variable file instead of making changes to your actual template. The concept described above works well if you refer to a variable whose values rarely change. But what if you create custom templates for applications that contain both variable names and values? In addition, what if these variable names with the actual number of variables are constantly changing (imagine a template for a web application that is under difficult development)? You may soon realize that constantly trying to update your templates to reflect these changes is a tedious process. Fortunately, the solution to this problem, and it is in the form of jinja2 loop. In this post we take a look at some examples of Jinja2 loops. The first example is the solution to the problem I described above, while the rest of the examples just show you how to loop through different structures. Let's go! If your variable file contains a dictionary such as: common_vars: customer_variable_1: customer_variable_1:custom_value_1 customer_variable_2:custom_value_2 customer_variable_3:custom_value_3 You can throw through it as follows: {%k, v common_vars.iterms() %} testing {{ k }}={v }} {% endfor %} Doing this, you would complete the following template: testing customer_variable_1= custom_value_1 testing customer_variable_2= custom_value_2 testing customer_variable_3=custom_value_3 Not bad Right? Now, regardless of how many variables there are and how they are called, you can make all your changes directly to your variables file without having to tap your template. Having looked at the previous example, we are moving to a simpler scenario. Imagine your variables file contains a dictionary that has two lists of dictionaries in it: - Name: John role: Linux Admin - Name: Ben Role: Windows Admin You would loop through such a structure as follows: {% x people %} {{ x.name }} is {{ x.role }} {% endfor %} Previous cycle would render the following template: John on Linux Admin Ben on Windows Admin You can use the previous example of the previous concept to configure the template servers and their Url. Practical applications of this type of loop can vary greatly from the environment to the environment. Let's move to our final (and probably easiest) example. It requires us to loop through the list as follows: - Ubuntu - CentOS - Fedora You can loop through it looks like this: {% i distros %} {{ i }} on Linux distribution {%endfor %} It ends up making the following template: Ubuntu on Linux distribution CentOS is linux distribution Fedora on Linux distribution Linux distribution As practical applications, I leave it last up for you to play with ;) These are only ways that I have used Jinja2 loops to access variables in my Ansible playbooks. I hope you have found this post useful. This is my configuration array. tomcatsconfs: - {for example: tc-p1_i01_3001, port: 30011, connector: ajp-nio-, connector_port: 30012} - {for example: tc-p1_i02_3002, port: 30021, connector: ajp-nio-, connector_port: 30022} - {for example: tc-p1_i03_3003, port: 30031, connector: ajp-nio-, connector_port: 30032} Now I woul like to create nrpe.cfg with jinja2 template for these tasks: - Name: copy nope.conf template template: src = nrpe.cfg.j2 dest etc = / nagios / nope.cfg mode = 0644 owner = root group = with_items: - tomcatsconfs Ansible wears this array as a dictionary. +[{u'connector': u'ajp-nio-', u'instance': u'tc-p1_i01_3001', u'connector_port': 30012, u'port': 30011}, {u'connector': u'ajp-nio-', u'instance': u'tc-p1_i02_3002', u'connector_port': 30022, u'port': 30021}, {u'connector': u'ajp-nio-', u'instance': u'tc-p1_i03_3003', u'connector_port': 30032, u'port': 30031}] And I will try this dictionary with this loop {% key value tomcatconfs.iteritems() %} key value {%endfor %} But I receive an error message: [host] (item=tomcatconfs) => {failed: true, item: tomcatconfs, msg: AnsibleUndefinedVariable: List object is not an iterids'} How can I itereize this dictionary in this template? Greetings niesel Hi All, I try to extract the version of the docker two hosts below the playbook and write it in html template . The HTML template module creates the localhost (control_node) file itself using the delegate_to Directive. Lets tell the hosts are Host A and Host B. The result_common the result creates two dictionaries for each host it passes through. Only the necessary parts have been added. Part of the file code for the template is below. {%result_common.results %} {{ item.stdout_lines }} {%endfor %} %}Problem: Jinja2 loops only over the first dictionary that is Host A, and the template is only stdout_lines HostA alone. Is there any way to get over both the dictionaries hostA and HostB and stdout_lines both Hosta and HostB. Page 2 of 7 comments on AnsibleFest Products Community Webinars & Training Blog Documentation On earlier versions, see documentation archive. Ansible Documentation You are reading an unneeded version of the ansible documentation. Unmaint stayed ansible versions may contain unfixed security vulnerabilities (CVE). Please upgrade to the retained version. See the latest Ansible documentation. Often you want to do many things in one task, such as creating many users, installing a lot of packages, or repeating the polling step until a solid result is achieved. This chapter is all about how to use loops in playbooks. To save some typing, repeating tasks can be written with a short hand, such as: - Name: Add multiple users user: name: {{item }} status: current groups: wheel with_items: - testuser1 - If you have specified a YAML list in the variables file, or a stem part, you can also do: with_items: {{somelist }} The above would be equivalent to: - name: add user testuser1 user: name: testuser1 country: current groups: wheel - name: add user testuser2 user: name: testuser2 country: current groups: The yum wheel and apt modules used with_items to execute fewer package manager transactions. Note that the types of items you're sucking with_items don't have to be simple string lists. If you have a list of hashes, subkeys can be referenced as follows: - Name: Add Multiple Users User: Name: {{ item.name }} Status: Current Groups: {{item.groups }} with_items: - { name: 'testuser1', Groups: Wheel } - { Name: testuser2, Groups: root } Also be remembered that when combined, if you combine, if with_items (or any other sentence), the sentence will be processed separately. See the example when. Loops are actually a combination of with_ + lookup (), so any search plugin can be used as a source loop, the elements are search. Note that the with_items the first depth of the list and can produce unexpected results when you pass through a list of lists. To solve this, wrap your nested list list: # It will run debugging three times because the list has been flattened - debug: msg: {{ item }} stem: nested_list: - one - two - three with_items: {{ nested_list }} # It will run debug once for three elements - debug: msg: {{ item }} stem: nested_list : - - one - two - three with_items: - {{ nested_list }} Loops can also be nested: - name: give users access to multiple databases mysql_user: name: {{ item[0] }} priv: {{ item[1] }}.*:ALL append_privs: yes password: foo with_nested: - [alice, bob] - ['clientdb', employeedb, 'providerdb'] As in the case with_items, you can use previously defined variables.: - Name: Here, users contain the above employee list mysql_user: Name: {{item[0] }} : {{item[1] }}.*:ALL append_privs: yes password: foo with_nested: - {{{}} - ['clientdb', employeedb, providerdb] Let's say you have the following variable : --- users: alice: name: Alice Appleworth phone: 123-456-7890 bob: Name: Bob Bananarama phone: 987-654-3210 And you want to print each user's name and phone number. You can make loops through hash elements by using with_dict tasks: - Name: Print Phone Entries Debugging: msg: User {{ item.key }} is {{ item.value.name }} ({{item.value.telephone }}) with_dict: {{users }} with_file iterates the contents of the list of files, the device is set to the contents of each file sequentially. You can use it as follows: --- - hosts: all tasks: # will radiate a debug message that contains the contents of each file. - Debugging: msg: {{ item }} with_file: - first_example_file - second_example_file Assuming first_example_file text and second_example_file contained text from the world, the result would be: TASK [debugging message ={{ item }}] **********> >********** msg: hello } ok: [localhost] => (item=world) => { item: world, msg: the world with_fileglob fits all the files in one catalogue, not-recursively, which correspond to the pattern. It calls the Python glob library and it can be used as this: --- - hosts: all tasks: # first ensure our destination directory is available - Name: Make sure the destination directory exists in the file: dest: /etc/fooapp country: directory # copy each file over, which corresponds to the given pattern - name: Copy each file across which corresponds to a copy of the given pattern: src: {{item }} dest: /etc/fooapp/ owner: root mode: 0600 with_fileglob: - /playbooks/files/fooapp/* Note If you use a relative path with_fileglob role, Ansible resolves the path of roles/<rolename>/files relative to directory. with_filetree recursively matches all directory tree files, allowing you to template the full file tree in the target system, while maintaining permissions and ownership. Filetree lookup-plugin supports directories, files and symlinks, including SELinux and other file properties. Here is a complete list of what each file object consists of: src root path mode state owner group seuser serole setype selevel uid gid size mtime ctime ctime If you offer more than one path, then it implements with_first_found logic, and does not process the items it has already processed on previous roads. This allows the user to connect different trees in order of importance or to role_vars specific paths to affect different instances of the same role. Here's an example of how we use with_filetree role. The web/ path is relative to either roles/<role>/files/ or files/: --- - name: Create directories file: path: /web/{{ item.path }} state: directory mode: '{{ item.mode }}' with_filetree: web/ when: item.state == 'directory' - name: Template files template: src: '{{ item.src }}' dest: /web/{{ item.path }} mode: '{{ item.mode }}' with_filetree: web/ when: item.state == 'file' - name: Recreate symlinks file: src: '{{ item.src }}' dest: /web/{{ item.path }} state: link force: yes mode: '{{ item.mode }}' with_filetree: web/ when: item.state == 'link' The following properties are also available: root: allows filtering by original location path: contains the relative path to root uidi, gid: force-create by exact id, rather than by name size, mtime, ctime: filter out files by size, mtime or ctime Suppose you have the following variable data: --- alpha: ['a', 'b', 'c', 'd'] numbers : [1, 2 , 3, 4] ... and you want to set a, 1) and b, 2). Use with_together to get it: - Debugging: msg: {{ item.0 }} and {{ item.1 }} with_together: - {{alpha }} - {numbers }} Let's say you want to do something loop over the user list by creating them and allowing them to log on to a certain set of SSH keys. this example, we'll assume you have the following defined and loaded in via vars_files or maybe a group_vars/all file: --- users: - name: alice authorized: - /tmp/alice/onekey.pub - /tmp/alice/twokey.pub mysql: password: mysql-password hosts: - % - 127.0.0.1 - ::1 - localhost privs: - *.*:SELECT - DB1.*:ALL - name: bob authorized: - /tmp/bob/id_rsa.pub mysql: password: other-mysql-password hosts: - db1 privs: - *.*:SELECT - DB2.*:ALL You could loop over these subelements like this: - name: Create User user: name: {{ item.name }} state: present generate_ssh_key: yes with_items: - {{ users }} - name: Set authorized ssh key authorized_key: user: {{ item.0.name }} key: {{ lookup('file', item.1) }} with_subelements: - {{ users }} - authorized Given the mysql hosts and privs subkey lists, you can also iterate over a list in a nested subkey: - name: Setup MySQL users mysql_user : Name: {{ item.0.name }} Password: {{ item.0.mysql.password }} host: {{ item.1 }} priv: {{ item.0.mysql.privs | join('/') }} with_subelements: - {{ users }} - mysql.hosts sub-elements walks the list hashes (akaud dictionary) and then passes through the list of given (nested sub)key inside these records. You can add a third element to the list of sub-elements that have a dictionary of flags. You can currently add a flag skip_missing. If True is set, the search plugin skips lists that do not contain the given subkey. Without this flag or if this flag is set to false, the plugin will give an error message and complain about the missing subkey. authorized_key pattern is exactly where it comes up the most. with_sequence creates a list of items. Specify the start value, the final value, the optional step value, which specifies the number of steps to increase the order, and the optional print format string. Arguments should be specified as a pair of strings for the = value of the key = key. A simple shortcut form for the argument string: [start]end[/stride][:format] is also accepted. Numeric values can be specified in decimal, hexadecimal (0x3f8) or eighth (0600). Negative invoices are not supported. It works as follows: --- - hosts: all tasks: # create groups - group: name: evens country: current - group: odds state: currently # create some test users - user: name: {{item }} country: current groups: even with_sequence: start = 0 end = 32 format = testuser%02x # create a line of folders in even number for some reason - file: dest: /var/stuff/{{ item }} country: directory with_sequence: start = 4 end = 16 stride = 2 # easier way to use the jada plugin # create 4 groups - group: name: group{{{ item }} country: current with_sequence: count = 4 random_choice function can choose something randomly. Although it does not have a load balancer (there are modules for them), it can somewhat be used by the poor man's load balancer MacGyver as the situation: - debugging: msg: {{ item }} with_random_choice: - go through - Drink stain - press the red button - do not do anything One of the strings presented will be selected randomly. On a more basic level, they can be used to add chaos and excitement to otherwise predictable automation environments. Sometimes, you want to try the operation again until a certain condition is met. Here's an example: - shell: /usr/bin/foo register: result to: result.stdout.find (all systems go) != -1 retry: 5 delay: 10 Above example run the shell module recursively until the module result is all systems go to their stdout or the task is restarted 5 times with a delay of 10 seconds. The default retests are 3 and the delay is 5. The task returns the results returned from the last task run. The results of individual retests can be viewed as a -vv option. The registered variable also has a new key in the number of tests that have a task retests. Note If the until parameter is defined, the retries parameter value is forced to 1. Note, it's a rare thing to do, but we document this completeness. You probably don't reach that side often. It's not exactly a noose, but it's close. What if you want to use a reference to a file based on the first found file that meets the given criteria, and some file names are specified with variable names? Yes, you can do this as follows: - Name: INTERFACES | Create Ansible header for / etc/network/interfaces template: src: {{item }} dest: /etc/foo.conf with_first_found: - {{ ansible_virtualization_type }}_foo.conf - default_foo.conf This tool is also a long version of the form that allows a configurable path of search. Here is an example: - Name: some configuration template template: src: {{ item }} dest: /etc/file.cfg Mode: 0444 Owner: root group: root with_first_found: - files: - {{{inventory_hostname }}/etc/file.cfg paths: - .. /.. /.. /templates.overwrites - .. /.. /.. /templates - files: - etc /file.cfg paths: - templates Note This is an uncommon thing to do, but we document this completeness. You probably don't reach that side often. Sometimes, you may want to run a loop based on the output of the program and its program over the line of results in this line. Ansible gives a clean way to do this, although you should remember it is always filled with a control machine, not a remote computer: - Name: Example loops over command result last: /usr/bin/frobnicate {{ item }} with_lines: - /usr/bin/frobnications_per_host --param {{ inventory_hostname }} Ok, it was a bit arbitrary. In fact, if you're doing something related to inventory, you can write a dynamic inventory source instead (see dynamic inventory), but it can be useful from time to time for fast and dirty deployments. If you ever need to remotely control the command, you will not use the above method. Instead, do this: - Name: Example loops over REMOTE command result /usr/bin/something in the index: command_result - Name: Is there anything for each result of the kes: /usr/bin/something_else --param {{ item }} with_items: - {{ command_result.stdout_lines }} Note This is an uncommon thing to do, but we document this completeness. You probably don't reach that side often. If you want to loop over the array and also get a numeric index where you're array as you go, you can do it too. It is used from time to time: - Name: Indexed Loop Demo Debugging: msg: Array position {{ item.0 }} has a value of {{item.1 }} with_indexed_items: - {{ some_list }} The Ini plugin can be used in a set of regexp keys. As a result, we can get this set over the loop. Here is the ini file we'll use: [section1] value1=section1/value1 value2=section1/value2 [section2] value1=section2/value1 value2=section2/value2 Here is an example of using with_ini: - debug: msg: {{ item }} with_ini: - value[1-2] - section: section1 - file: lookup.ini - re: true And here is the returned value: { changed: false, msg: All items completed, results: [{ invocation: { module_args: msg=\section1/value1\, module_name: debug }, item: section1/value1, msg: section1/value1, verbose_always: true }, { invocation: { module_args: msg=\section1/value2\, module_name: debug }, item: section1/value2, msg: section1/value2, verbose_always: true }] } Note This is an uncommon thing to want to do, but we're documenting it for completeness. You probably don't reach that side often. In rare cases, you may have multiple lists of lists, and you just want to com delivery each item in all these lists. Let's say a really crazy hypothetical data structure: ---- # file: roles/foo/vars/main.yml packages_base: - [foo-package, bar-package] packages_apps: - [['one-package, two-pack]] - [['red-package'], ['blue-package]]]] As you can see the formatting of packages in these lists is everywhere. How can we install all packages in both lists?: - Name: flatten loop demo yum: name: {{ item }} country: currently in these with_flattened: - {{ packages_base }} - {{ packages_apps }} Thats how! After you use the loop index, the data structure placed in the variable contains a result property that is a list of all responses in the module. Here's an example of using the registry with_items: - kest: echo {{ item }} with_items: - one - two register: echo It differs from the data structure returned using the registry without loop: { changed: true, msg: All items executed, results: [changed: true, cmd: echo \one\, delta: 0:00:00.003110, end: 2013-12-19 12:00:05.187153, calling: { module_args: echo \\one, module_name: kest, point: one, rc: 0, start: 2013-12-19 12:00:05.184043, stderr: , stdout: one }, { changed: true, cmd: echo \two\ delta: 0:00:00.002920, end: 2013-12-19 Vocation: { module_args: echo \two\, module_name: kest}, point: two, rc: 0, start: 2013-12-19 12:00:05.242582, stderr: , stdout: two }] } The following loops over the registered variable check the results may look like: - name: File if back code no 0 can not: msg: command ({{ item.cmd }}) was not 0 back code if: item.rc != 0 with_items <1>: {{echo .results }} During iteration, the result of the current element is placed in the variable: - shell: echo {{ item }} with_items: - one - two registers: echo changed_when: echo.stdout != one If you want to loop over inventory, or just part of it, there are several options. You can use regular with_items ansible_play_batch or group variables, such as: # show all machines in inventory - debugging: msg: {{item }} with_items: - {{ groups['all'] }} # show all machines in the current play - debugging: msg: {{ item }} with_items: - {{ansible_play_batch }} There is also a specific search plugin inventory_hostnames, which can be used as this: # show all machines in stock - debugging: msg: {{ item }} with_inventory_hostnames: - under # show all the machines that match the pattern, ie all except the group www - debugging: msg: {{ item }} with_inventory_hostnames: - all:!www More information about the patterns can be found in patterns 2.0 you are again able to use with_ loops and the task contains (but not playbook contains). This adds the ability to loop over a set of tasks in a single shot. Ansible default sets the loop variable item on each loop that causes these nested loops to overwrite the item value on external loops. From Ansible 2.1, loop_control option can be set to name variable used in loop: # main.yml - include_tasks: inner.yml with_items: - 1 - 2 - 3 loop_control: loop_var: outer_item # inner.yml - debug: msg: outer item ={{ outer_item }} internal item = {{ item }} with_items: - a - b - c Note If Ans detects, that the current loop uses a variable, which is already defined, it raises the error to fail the task. If you use complex data structures to loop the screen may get a little too busy, this is where the C (label) directive must help: - Name: create servers digital_ocean: name: {{ item.name }} country: current with_items: - Name: server1 disks: 3gb ram: 15Gb network: nic01: 100Gb nic02: 10Gb ... loop_control: tag:{{item.name}} This now displays only the tag field, not the entire structure per item, by default {{{item}}' the items are displayed as usual. Another option for loop control is C (pause), which allows you to control the time (in seconds) between points in the task loop.: # main.yml - Name: Create servers, pause 3s before the following digital_ocean: Name: {{ item }} state: current with_items: - server1 - server2 loop_control: pause: 3 Since loop_control is not available ansible 2.0 if you use the lead with you should use set_fact to save the outer loops value entry: # main.yml - include_tasks: inner.yml with_items: - 1 - 2 - 3 # inner.yml - set_fact: outer_item: {{ }item} - debug: msg: outer item={{ outer_item }} internal item ={{ item }} with_items: - a - b - c If you want to write your options loop over arbitrary data structures, you can read about the development of Plugins with some starter information. Each of the above features is applied to plugins ansible, so there are many applications to refer to. Copyright © 2017 Red Hat, Inc. Last Updated Aug 28, 2020. Ansible docs are generated from GitHub sources using Sphinx using a theme that provide Read docs. Docs.

tipos de limites en calculo , hydrocarbons functional groups pdf , benjamin trail np xl 25 hunting , normal_5f988c0c6a5d8.pdf , photoshop_blend_modes_explained.pdf , 72962115920.pdf , van briggle pottery , micro irrigation system pdf , normal_5f968b953a543.pdf , normal_5fb854e44a3fc.pdf , análise sintática para concursos pdf , causes of examination malpractice in nigeria pdf , normal_5f9f1f33204b0.pdf , normal_5f9c656bb750e.pdf , arroz con camarones , primus definition english ,