Ansible

Raúl Estrada

Octubre 2020

Ansible configuration

One of the features of Ansible is the ability to override the defaults per project. In order to do it that, we just need to create a file called `ansible.cfg` in the root of our project and Ansible will read it and apply the configuration.

There is a big number of parameters that can be configured, and all of them can be found in the official documentation at http://docs.ansible.com/ansible/intro_configuration.html.

As you can see, the documentation for Ansible is quite good, and the majority of the time, it will provide an answer to your problems.

Let's see how the configuration can help us. If you remember from the previous example, we have specified the flag `-i` in order to tell Ansible where our inventory file lives. Ansible has a default for this value, which is `/etc/ansible/hosts`. In our little project, our inventory is in the same folder as our code, and in order to specify it to Ansible, we need to create a configuration file with the following content:

Copy

```
[defaults]
inventory = ./inventory
```

Now, we run our `playbook` again with the following command:

Copy

```
ansible-playbook tasks.yml
```

We did not specify the host list, but Ansible, after reading `ansible.cfg` knows that the inventory file can be located at `./inventory` .

Ansible has a hierarchy of precedence to find the configuration:

- The `ANSIBLE_CONFIG` environment variable
- `ansible.cfg`
- `.ansible.cfg`
- `/etc/ansible/ansible.cfg`

So, if we define an environment variable called `ANSIBLE_CONFIG` pointing to a file, the Ansible configuration will be read from that location and the rest of the options will be ignored. This is particularly helpful in segregating environments: our CI server can define its own configuration in the environment file, whereas developers can have the `ansible.cfg` file checked in into the source control so that is shared across everyone.

There are a few sections that can be specified in `ansible.cfg`. Sections control several aspects of of Ansible, such as connections. Under certain circumstances, we might need to add special parameters for `ssh` to work, and it is as easy as adding the following lines to your `ansible.cfg` file:

Copy

```
[ssh_connection]
ssh_args=<your args here>
```

Ansible variables, remote facts and templates

Variables and templates are an important part of Ansible. They allow us to override values in our configuration (servers and playbooks) so that we can write generic playbooks that can be reused across different configurations with minor tweaks. With templates, we can render configuration files from our host so we could potentially use Ansible to manage the configuration of remote servers with little to no effort. It also can be used to generate and install SSL certificates for different hosts transparently to the user.

Both of them (variables and templates) use a template engine called Jinja2, which allows logic and interpolation to be embedded in our configurations.

In general, there are several ways of defining variables, but we are only going to visit the most common ones (under my criteria), as otherwise, it would take us the size of several chapters to document them properly. If you want to explore further different ways of defining variables, the official documentation provides a fairly comprehensive guide at http://docs.ansible.com/ansible/playbooks_variables.html.

Ansible variables

Variables are the most simple of the potential customizations. With variables, we can define values that are going to be replaced in our playbooks. Let's take a look at the following playbook:

Copy

```
---
- hosts: all
  user: root
  tasks:
  - debug:
    msg: "Hello {{ myName }}! I am {{ inventory_hostname }}"
```

Replace the content of `tasks.yml` with the snippet from earlier. There are two new symbols in our task. Also, our task is new: debug is used to output values from our variables into the terminal while executing the playbook. Let's take a look at the execution (we will use the same configuration as the example from earlier):

Copy

```
ansible-playbook -i inventory tasks.yml
```

It fails:

```
PLAY [all] *********************************************************************

TASK [Gathering Facts] *********************************************************
ok: [35.187.81.127]

TASK [debug] *******************************************************************
fatal: [35.187.81.127]: FAILED! => {"failed": true, "msg": "the field 'args'
has an invalid value, which appears to include a variable that is undefined.
The error was: 'myName' is undefined\n\nThe error appears to have been in
'/code/ansible-variables/tasks.yml': line 5, column 5, but may\nbe elsewhere
in the file depending on the exact syntax problem.\n\nThe offending line
appears to be:\n\n tasks:\n - debug:\n ^ here\n"} to retry,
use: --limit @/Users/dgonzalez/code/ansible-variables/tasks.retry

PLAY RECAP *********************************************************************
35.187.81.127 : ok=1 changed=0 unreachable=0 failed=1
```

The reason for the failure can be seen in in the message: we have a variable defined called `name` that does not have a value associated. Ansible will fail if there is a value that cannot be interpolated, aborting the execution of the task.

There is another interesting piece of information here: Ansible gives you a parameter to retry the playbook only on the hosts that were not successful. If we wanted to retry the playbook only on the failed hosts, we could run the following command:

Copy

```
ansible-playbook -i inventory tasks.yml --limit @/Users/dgonzalez/code/ansible-variables/tasks.retry
```

The new parameter, `tasks.retry` is a file with a list of hosts that are okay to rerun the playlist as they failed before.

Going back to our missing variables, we need to define the variable called `myName` . There are a few ways of doing that; the first is via the command line:

Copy

```
ansible-playbook -i inventory tasks.yml -e myName=David
```

And you can see that the output of the playbook is looking better now:

```
PLAY [all] *********************************************************

TASK [Gathering Facts] *********************************************
ok: [35.187.81.127]

TASK [debug] *******************************************************
ok: [35.187.81.127] => {
 "changed": false,
 "msg": "Hello David! I am 35.187.81.127"
}

PLAY RECAP *********************************************************
35.187.81.127 : ok=2 changed=0 unreachable=0 failed=0
```

As you can see, the variables got interpolated and we can see the message `Hello David! I am 35.187.81.127` .

The second way of defining variables is via inventory, as we have seen earlier:

Copy

```
[nginx-servers]
35.187.81.127 myName=DavidInventory
```

If we modify our inventory to match the preceding snippet, the value of our variable will be `DavidInventory` and we don't need to pass a value in the command line:

Copy

```
ansible-playbook -i inventory tasks.yml
```

This will produce the message `Hello DavidInventory! I am 35.187.81.127` .

The third way to define variables in Ansible is by defining them in the playbook itself. Take a look at the following playbook:

Copy

```
---
- hosts: all
 vars:
 myName: David
 user: root
 tasks:
 - debug:
 msg: "Hello {{ myName }}! I am {{ inventory_hostname }}"
```

As simple as it sounds, once you define the variable in the `vars` section of your playbook, it becomes available; therefore, there is no need to specify the value anywhere else.

The fourth way to define variables is via files. Ansible is designed to be a self-documented component that can be easily understood by someone with not much experience in it. One of the ways in which Ansible facilitates the task of understanding playbooks is the possibility of writing every single configuration piece in a file. Variables are not the exemption, so Ansible will let you define variables in files or playbooks.

Let's start with the files. Create a file called `vars.yml` in the same folder in which you are working (where your playbook and inventory are) with the following content:

Copy

```
myName: DavidFromFile
yourName: ReaderFromFile
```

Now we can run the following command in order to use the variables file:

Copy

```
ansible-playbook -i inventory playbook.yml -e @vars.yml
```

And if you check the output, it would be the same as the one from earlier.

In this case, we have defined a new variable that we are not using (`yourName`), but that is fine. I just wanted to show you that Ansible won't complain if there are free variables, but it will raise an error if there are unbound interpolations.

In this case, we have included `vars.yml` in our playbook via the command line, referencing your local file with `@` in the beginning, but there is another possibility for using variable files in Ansible: including them from within the playbook. Let's take a look at how it is done:

Copy

```
---
- hosts: all
 user: root
 tasks:
 - name: Include vars
 include_vars:
 file: vars.yml
 - debug:
 msg: "Hello {{ myName }}! I am {{ inventory_hostname }}"
```

In this case, we have used the `include_vars` module in our playbook. Now execute the playbook with the following command:

Copy

```
ansible-playbook -i inventory tasks.yml
```

You will get the following output:

```
PLAY [all] *********************************************************

TASK [setup] *******************************************************
ok: [35.187.81.127]

TASK [Include vars] ************************************************
ok: [35.187.81.127]

TASK [debug] *******************************************************
ok: [35.187.81.127] => {
 "msg": "Hello DavidFromFile! I am 35.187.81.127"
}

PLAY RECAP *********************************************************
35.187.81.127 : ok=3 changed=0 unreachable=0 failed=0
```

As you can see, there is an extra task that takes a file and injects the variables in the context.

This module is quite flexible and there are several options to include variable files in our playbook. We have used the most straightforward one, but you can check out other options in the official documentation at http://docs.ansible.com/ansible/include_vars_module.html.

There is another possibility for including a variable file into our playbook, and it is using the `vars_files` directive in our playbook:

Copy

```
---
- hosts: all
 user: root
 vars_files:
 - vars.yml
 tasks:
 - debug:
 msg: "Hello {{ myName }}! I am {{ inventory_hostname }}"
```

This will take the `vars.yml` file and inject all the defined variables into the context, making them available for use.

As you can see, Ansible is quite flexible around the definition of variables.

There is another interesting way of setting up variables in Ansible that helps us further customize our playbooks: `set_fact` . Setting facts allows us to set variables dynamically in our playbooks. `Set_fact` can be used in combination with another interesting instruction called register. Let's look at an example:

Copy

```
---
- hosts: all
 user: root
 tasks:
 - name: list configuration folder
 command: ls /app/config/
 register: contents
 - set_fact:
 is_config_empty: contents.stdout == ""
 - name: check if folder is empty
 debug: msg="config folder is empty"
 when: is_config_empty
 - name: installing configuration
 command: <your command here>
 when: is_config_empty
```

What we are doing here is basically setting a variable to true if the configuration folder of our app is empty (hypothetic configuration folder) so that we can regenerate it only when it is not present. This is done by making use of the instruction when that allows us to execute instructions conditionally. We will come back to it during this chapter.

We have visited the most common ways of defining variables, but there is one question pending: what is the precedence of the different methods for creating variables?

This is something that I have to query myself whenever I am working in a playbook, and the truth is that at the end of the day, you will use only a couple of methods to create variables so that it is not as important as it should be. In my case, I tend to create a file with variables (when not working with roles), and if I want to override a value, I do that on the command line (or environment variable), which is the highest priority in the chain. The complete list of variable precedence can be found at http://docs.ansible.com/ansible/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable.

Ansible remote facts

Remote facts in Ansible are a way to specify configuration on remote hosts either by an explicit configuration file or by a script that returns data about the server. In general, this feature is very useful for operations such as maintenance, setting up flags that specifically mark the host as out of the pool so that our playbooks have no effect in the hosts.

Take a look at the following command (assuming the inventory from the previous example is present in the folder and the VM is running on Google Cloud Platform):

Copy

```
ansible all -m setup -i inventory --user=root
```

This will output an enormous amount of data (JSON-formatted data). This data is all the known facts about the remote host, such as the CPU type, machine ID, network interfaces, kernel version, and so on. They can be used within our playbooks, but they can also be extended to add more data that is controlled by the remote host without any local configuration.

In order to set up custom remote facts, we have several options, but at the end of the day, the custom facts are defined in JSON files by default under `/etc/ansible/facts.d/` . It is also possible to create an executable (a script) under the same folder so that Ansible will execute it and take the output as facts and add them to the facts scope. Take a look at the following file:

Copy

```
{
 "my_name": "David Gonzalez"
 }
```

Put into the remote box (the one used in all the examples from earlier) and create a file in `/etc/ansible/facts.d/example.facts` with the content from earlier.

Once this is done, run the following command:

Copy

```
ansible all -m setup -i inventory --user=root | grep -B 3 -A 3 my_name
```

It almost looks magical, but the output of your command should now include the facts that you created earlier:

Copy

```
},
"ansible_local": {
"example": {
"my_name": "David Gonzalez"
}
},
"ansible_lsb": {
```

Now they can be used in your playbook in the `ansible_local` variable, for example, to access `my_name` :

Copy

```
---
- hosts: all
 user: root
 tasks:
 - name: Gather my_name fact.
 debug: msg="{{ ansible_local.example.my_name }}"
```

As mentioned earlier, Ansible can also gather facts from a script placed in the facts path. This script should have the `x` flag present, which indicates that it can be executed and have the extension `fact`. Let's look at a very interesting trick that I find quite useful. When I try to diagnose a failure in our systems, the first thing I tend to check is the CPU usage. The majority of the time, our systems are highly observable (monitored) so it is easy to check the CPU load, but sometimes, monitoring might not be in place.

First, go to the server that we have been using in the preceding examples and create a file in `/etc/ansible/facts.d/cpuload.fact` with the following content:

Copy

```
#!/bin/bash
CPU_LOAD=`grep 'cpu ' /proc/stat | awk '{usage=($2+$4)*100/($2+$4+$5)} END {print usage "%"}'`
echo { \"cpu_load\": \"$CPU_LOAD\"}
```

This is a simple script that will output JSON with information about the CPU load in your system. Once the file is created, give it execution permissions:

```
chmod u+x /etc/ansible/facts.d/cpuload.fact
```

And we are done. Before disconnecting the SSH session, make sure that the script works as expected by executing it:

Copy

```
/etc/ansible/facts.d/cpuload.fact
```

This should output something like the following:

Copy

```
{ "cpu_load": "0.0509883%"}
```

Now it is time to test our scripted facts. What we are going to do is create a playbook that gets the CPU load and outputs it to the terminal with a debug message. This is the content:

Copy

```
- hosts: all
 user: root
 tasks:
 - name: Get CPU load
 debug: msg="The CPU load for {{ ansible_hostname }} is {{ ansible_local.cpuload.cpu_load }}"
```

Run the preceding playbook:

Copy

```
ansible-playbook -i inventory tasks.yml
```

You should get an output very similar to the following one:

```
PLAY [all] *********************************************************

TASK [Gathering Facts] *********************************************
ok: [35.187.81.127]

TASK [Update sources] **********************************************
ok: [35.187.81.127] => {
 "changed": false,
 "msg": "The CPU load for nginx is 0.0511738%"
}

PLAY RECAP *********************************************************
35.187.81.127 : ok=2 changed=0 unreachable=0 failed=0
```

Now we have a rudimentary tool to check the CPU load on our servers with a simple command, leveraging the host groups to Ansible.

One thing we have not explained is the first task that Ansible outputs in every playbook: gathering facts.

This task gets all those facts that we have been talking about in this section and creates the context for the playbook to run, so in this case, the CPU load that we get is the CPU load gathered at the execution of that task.

Ansible templates

Templates are another powerful tool from Ansible. They allow us to render configuration files, application properties, and anything that can be stored in a human readable file.

Templates rely heavily on variables and a template engine called Jinja2 , which is used by Ansible to render the templates. First, we are going to install `ngnix` on our server with a simple playbook:

Copy

```
---
- hosts: all
 user: root
 tasks:
 - name: Update sources
 apt:
 update_cache: yes
 - name: Upgrade all packages
 apt:
 upgrade: dist
 - name: Install nginx
 apt:
 name: nginx
 state: present
```

As you can see, it is very simple:

- Update the `apt cache`
- Upgrade the system
- Install `nginx`

Now, just run the preceding playbook using the VM created earlier:

```
ansible-playbook -i inventory tasks.yaml
```

And when the playbook is finished, you should have `nginx` running in your remote server. In order to verify it, just open the browser and use the IP of your VM as URL. You should see the `nginx` welcome screen.

Now, we are going to create a template with the `nginx` configuration, where we can add or remove servers with templates in a fairly easy manner. Create a folder called `nginx-servers` in your current directory (where the playbook is) and add a file called `nginx.yml` with the following content:

Copy

```
---
- hosts: all
 user: root
 vars_files:
- vars.yml
 tasks:
 - name: Update sources
 apt:
 update_cache: yes
 - name: Upgrade all packages
 apt:
 upgrade: dist
 - name: Install nginx
 apt:
 name: nginx
 state: present
 - template:
 src: nginx-servers/nginx-one.conf.j2
 dest: /etc/nginx/sites-enabled/default
 owner: root
```

Let's explain the file a bit:

- The system is upgraded using `apt`.
- Using apt as well, `nginx` is installed. Note that Ansible uses a declarative approach to install packages: you state the name of the package and the state that the package should be in after the playbook is executed.
- The playbook renders the configuration for a virtual server in `nginx` from a template called `nginx-one.conf.j2`. We will come back to this in a second.
- The playbook reloads the `nginx` service so that the new configuration takes effect.

We have a few blocks missing in the preceding playbook. The first block is the file called `nginx-one.conf.j2` . This file is a template that is used to render the `nginx` configuration for a virtual host in the server. Let's look at the content of that file:

Copy

```
server {
   listen {{ server_one_port }} default_server;
   index index.html;
}
```

Create a folder called `sites-enabled` and add the `nginx-one.conf.j2` file to it with the preceding content. This file is a standard `nginx` server block but with one particularity: we have a **server_one_port** as a placeholder for the port so that we can control the port where the `nginx` virtual host is exposed. This is very familiar to us: we are using the variables to render the templates.

The second block is the file called `vars.yml ()` with the following content:

Copy

```
server_one_port: 3000
```

This is very simple: it just defines the variables required to render the template from earlier. One thing that you need to be aware when using templates is that all the variables in the context can be accessed in it, from the facts gathered from the remote server to the variables defined everywhere.

Once we have everything in place (the two files from earlier, the playbook from earlier, and the inventory from the previous example), we can run the playbook as usual and verify that everything works as expected:

Copy

```
ansible-playbook -i inventory nginx.yml
```

If everything worked as expected, you should have a fully functional `nginx` server (serving the default page) in your VM in Google Cloud Platform on the port `3000` .

Flow control

In Ansible, it is possible to use flow control statements such as loops or conditionals using variables as input. This can be used to repeat tasks on a certain dataset and avoid executing some tasks if a few conditions are not met: we might want to use different commands depending on the underlying system of our server.

We have already seen an example of conditionals using the `when` clause in our previous examples, but let's explain it a bit more:

Copy

```
---
- hosts: all
 user: root
 tasks:
 - command: /bin/false
 register: result
 ignore_errors: True

 - debug: msg="fail!"
 when: result|failed

 - debug: msg="success!"
 when: result|succeeded
```

The preceding code is very easy to read: a command is executed (ignoring the potential errors so our playbook continues), and it registers a variable called result. Then, we have two debug tasks:

- The first one will only be executed only if the `/bin/false` command fails
- The second one will be executed only if the `/bin/false` command succeeds

In this playbook, we are using two new tricks:

- `ignore_errors` : With this clause, if the task fails, the playbook will continue executing the following tasks. This is very helpful if we want to test for assumptions in the system, for example, if some files are present or a certain network interface is configured.
- `Pipe symbol (|)` : This symbol is called `pipe` . It is a Jinja2 expression used to filter values. In this case, we are using the failed and succeeded filters to return true or false depending on the outcome of the command. There are many filters that can be used on Jinja2 to work in a similar way as Unix pipes transforming the data that goes through them.

Another type of control flow structure are loops. Let's look at how loops work:

Copy

```
---
- hosts: all
 user: root
 vars:
 names:
 - David
 - Ester
 - Elena
 tasks:
 - name: Greetings
 debug: msg="Greetings {{ item }}! live long and prosper."
 with_items: "{{ names }}"
```

Here, we are using something new that we did not see at the time of explaining variables: they can have a structure such as lists and dictionaries. In this case, we are defining a list with a few names and outputting a message for each of them. Now it is time to run the playbook. Save the preceding content in a file called `loops.yml` and execute the following command:

Copy

```
ansible-playbook -i inventory loops.yml
```

We will assume that the inventory is the same as the one used in the preceding examples. After finishing, you should see something similar to the following output in your Terminal:

```
PLAY [all] *********************************************

TASK [Gathering Facts] *********************************
ok: [35.187.81.127]

TASK [Greetings] ***************************************
ok: [35.187.81.127] => (item=David) => {
 "item": "David",
 "msg": "Greetings David! Live long and prosper."
}
ok: [35.187.81.127] => (item=Ester) => {
 "item": "Ester",
 "msg": "Greetings Ester! Live long and prosper."
}
ok: [35.187.81.127] => (item=Elena) => {
 "item": "Elena",
 "msg": "Greetings Elena! Live long and prosper."
}
PLAY RECAP *********************************************
35.187.81.127 : ok=2 changed=0 unreachable=0 failed=0
```

It is also possible to define a list using the compact version of the declaration. Take a look at the following statement:

Copy

```
names:
- David
- Ester
- Elena
```

This can be redefined as follows:

Copy

```
names: ['David', 'Ester', 'Elena']
```

And it is totally equivalent.

It is also possible to define dictionaries in Ansible and use them as variables. They can also be used as iterable elements, which enable us to give structure to our data:

Copy

```
---
- hosts: all
 user: root
 vars:
 namesAge:
 - name: David
 age: 33
 - name: Ester
 age: 31
 - name: Elena
 age: 1
 tasks:
 - name: Presentations
 debug: msg="My name is {{ item.name }} and I am {{ item.age }} years old."
 with_items: "{{ namesAge }}"
```

If you are familiar with software development, the preceding snippet will make perfect sense to you: a list of structured data (an array of objects) that holds information to be accessed by the key.

In the rest of the book, we will be using more advanced features of flow control structures in Ansible, and we will explain them as we go, but if you want to learn more about it, the following links might be useful for you:

- Conditionals (http://docs.ansible.com/ansible/playbooks_conditionals.html)
- Loops (http://docs.ansible.com/ansible/playbooks_loops.html)
- Jinja2 Templating (http://docs.ansible.com/ansible/playbooks_templating.html)