Managing BSD Systems with Ansible

Benedict Reuschling

University Politehnica of Bucharest

September 20, 2018

EuroBSDcon 2018

Infrastructure As Code

When the number of machines to manage increases, it is neither efficient nor practical to manually configure each one by hand. Deploying system configuration settings, installing new software, or querying the systems for certain information are just a few of the tasks that can be automated and controlled from a central point. These configuration management systems work by describing how the target system should look like rather than listing individual commands to get to that desired state. It is the job of the configuration management system to compare the desired with the current state of the system and perform the necessary tasks to get there.
The actions to take on the target systems are often described in a domain specific language, so that individual differences between operating systems are abstracted away. This **infrastructure as code** can be shared, reused, and extended to fit the individual requirements of systems and company policies. Administrators can deploy a large number of changes over the network in a short amount of time as parallel jobs to be executed.

Overview

Introduction to Ansible

This chapter will cover Ansible as an example on how to manage multiple machines in a reliable and consistent way. We will look at how Ansible works, what kind of jobs it can do (machine configuration, software deployment, etc.), and how it can be used.
Although there are many other software packages with the same features such as Ansible, it has some distinct features. One of them is the clientless execution on target machines (only SSH is required) and that it is relatively simple to get started. A command-line client is available for ad-hoc commands, while so called *playbooks* allow for more complicated sets of changes to be made to target machines.

Idempotency

An important concept in this area is the so called **idempotency**. It describes the property of certain actions or operations. *An operation is said to be idempotent when the result is the same regardless of whether the operation was executed once or multiple times.* This is important when changing the state of a machine and the target may already have the desired state.
For example, a configuration change might add a user to the system. If it does not exist, it will be added. When that same action is run again and such a user is already present, no action is taken. The result (a user is added) is the same and when that action is run multiple times, it will still not change.
Another example would be adding a line to a configuration file. Some configuration files require that each line is unique and does not appear multiple times. Hence, the system adding that line needs to check whether that line is already present before adding it to the file. If not, it would not be an idempotent operation.
Idempotency appears in many other computer science (and math) fields of study, though the basic principle always stays the same.

Overview

Requirements

Ansible needs to be installed on at least one control machine, which sends the commands to a target system (could be the same machine). This is typically done over the network to a set of hosts. The target machines only have to run the SSH daemon and the control machine must be able to log in via SSH and perform actions (`sudo` privileges).
At the time of this writing, Ansible is using Python version 2.6 or above installed on the control machine and the managed systems. Some modules may have additional requirements listed in the module specific documentation.
The target nodes are typically managed by SSH (**s**ecure **sh**ell), so a running SSH client is needed (there is also a *raw* module that does not require SSH). File transfers are supported via SFTP or SCP. The control machine does not require anything special (no database or daemons running). Ansible can be installed using package managers (apt, pkg, pip, etc).

Setting up the Inventory File

Ansible manages systems (called nodes) from a list of hostnames called the inventory. This is a file which is located by default in `/usr/local/etc/ansible/hosts` on BSD systems. It contains a list of hosts and groups in INI-style format like this:

```
[webservers]
www1.example.com
www2.example.com

[dbservers]
oracle.example.com
postgres.example.com
db2.example.com
```

In this example, there are two groups of hosts: `webservers` and `dbservers`. Each group contains a number of hosts, specified by their hostnames or IP addresses. Systems can be part of more than one group, for example systems that have both a database and a webserver running.

Settings in the Inventory File

Multiple hosts with a numeric or alphanumeric naming scheme can be specified like this:

```
[computenodes]
compute[1:30].mycompany.com

[alphabetsoup]
host-[a:z].mycompany.com
```

This will include hosts named `compute1.mycompany.com`, `compute2.mycompany.com`, ... `compute30.mycompany.com` and `host-a.mycompany.com`, `host-b.mycompany.com`, ... `host-z.mycompany.com`, respectively.
Host-specific variables can be set by listing them after the hostname. For example, the BSDs are using a different path to the ansible executable, so we list it in the inventory file:

```
[mybsdhosts]
mybsdhost1  ansible_python_interpreter=/usr/local/bin/python
```

A complete list of inventory settings can be found at `http://docs.ansible.com/ansible/intro_inventory.html`.

Ansible Configuration File

Ansible can be configured in various ways. It looks for these configuration options in the following order:

- `ANSIBLE_CONFIG` (an environment variable)
- `ansible.cfg` (in the current directory)
- `.ansible.cfg` (in the home directory)
- `/usr/local/etc/ansible/ansible.cfg`

We'll specify a separate user called `ansible` in the file:

```
$ cat ~/.ansible.cfg
[defaults]
inventory = hosts
remote_user = ansible
```

`http://docs.ansible.com/ansible/intro_configuration.html` has a complete list of the available configuration options.

Bootstrapping Python onto the Remote Machine

Ansible uses Python to execute remote commands. This requires Python to be installed on the remote machine, even though we have not made Ansible working yet. This could also pose a problem on devices that do not have SSH running like routers for example. To solve this chicken-and-egg problem, we can use a different method for executing commands. This is called `raw` mode and does not have any abstractions, but rather takes literal commands:

```
ansible mybsdhost1 -m raw -a "pkg install -y python27"
```

Once that command has executed successfully, Ansible is fully set up and can use other modes, modules, and playbooks. To make sure our communication between the control machine and the targets is encrypted, we set up SSH and exchange public keys for passwordless logins.

Overview

SSH Setup

Ansible communicates securely over SSH with the managed systems. Although this is not the only option, it is the most common one, so we'll cover it here. The SSH daemon (server) must be running on the managed nodes.
The SSH public keys must be exchanged with the target systems so that the control machine can log into them and execute commands without requiring a password. This can be done by a separate user (i.e. `ansible`) that is available on all systems. To protect the key from unauthorized access, it is recommended to set a passphrase for the key. Combined with an SSH agent, the passphrase does not need to be entered each time, but will be handled by the agent when communicating with the remote systems.
The steps are as follows:

1. Create a key pair (public/private) on the local machine
2. Distribute the key to the remote systems
3. Run the SSH agent to cache the key in memory

Generating the SSH Key Pair

To generate a new key for the `ansible` user, use `ssh-keygen`:

```
ansible@host$ ssh-keygen -t ed25519 -f ansible
Generating public/private ed25519 key pair.
Enter passphrase (empty for no passphrase): <enter-your-passphrase>
Enter same passphrase again: <reenter-your-passphrase>
Your identification has been saved in /home/ansible/.ssh/ansible.
Your public key has been saved in /home/ansible/.ssh/ansible.pub.
The key fingerprint is:
SHA256:Hcl0VflRPR12ebzG2F/GoM5zENvFVf9mgI8N6Sl/5Tg ansible@host
The key's randomart image is:
+--[ED25519 256]--+
|           . o..+%|
|          o o  o=B|
|           +. . =B|
|          . .= Bo=|
|         S .+ o=O+|
|           o .ooO+|
|            +..+o=|
|              o.C o|
|                 .o |
+----[SHA256]-----+
```

The passphrase must be sufficiently long and complex. It's less likely for someone to guess it that way. Remember the passphrase!

Distribute the Key to the Remote System

The following files were generated by `ssh-keygen` in `/home/ansible/.ssh/`:

```
ansible@host$ ls -l .ssh
total 8
-rw------- 1 ansible ansible 1,8K Jul 18 10:37 ansible
-rw-r--r-- 1 ansible ansible  395 Jul 18 10:37 ansible.pub
ansible@host:~/.ssh$
```

The public key has the extension `.pub` and can be distributed to remote systems. The file called `id_rsa` represents the private key and *must not* be exposed to others. Do not copy this file or change the permissions!

The public key can be copied to a remote system using either `ssh-copy-id(1)` or if that is not available, by using the following command sequence (which requires entering the passphrase):

```
$ cat ansible.pub | ssh remote-host 'cat >> ~/.ssh/authorized_keys'
```

The remote system must have an `ansible` user and needs permission to log in via `ssh`. Add the line

```
AllowUsers ansible
```

to `/etc/ssh/sshd_config` and restart the SSH server if required.

Load the SSH Key into the SSH Agent

The ssh-agent(1) man page gives the following description about what the program does:

> *ssh-agent is a program to hold private keys used for public key authentication (RSA, DSA, ECDSA, ED25519). The idea is that ssh-agent is started in the beginning of an X-session or a login session, and all other windows or programs are started as clients to the ssh-agent program. Through use of environment variables the agent can be located and automatically used for authentication when logging in to other machines using ssh(1).*

We start the agent together with a new shell so that all programs executed from that shell can access the key:

```
ansible@host$ ssh-agent <myshell>
```

Then, we use `ssh-add` to load the key into the agent (we need to enter the passphrase one last time):

```
ansible@host$ ssh-add
Enter passphrase for /home/ansible/.ssh/ansible: <the-passphrase>
Identity added: /home/ansible/.ssh/ansible (/home/ansible/.ssh/ansible)
```

Testing the Remote SSH Login

To verify that the key was copied successfully to the remote system and loaded into the SSH agent on the local machine, we connect to the remote system with our `ansible` user:

```
ansible@host$ ssh ansible@remote-host
Linux remote-server 2.6.12-10-686 #1 Mon Feb 13 12:18:37 UTC 2006 i686 GNU/Linux
The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.
Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.
Last login: Fri Mar  13 13:41:17 2016
ansible@host$
```

It worked!

We were able to log into the system without the need to enter our passphrase. As long as the shell is running, the key remains stored in memory and is used each time the passphrase is required. That way, the private key is not transferred to the remote system.

Repeat the above steps for each host that should be managed by Ansible. We can now start learning about the way Ansible does that.

Overview

Ansible Management from the Command-Line

Ansible can issue ad-hoc commands from the command-line to remote systems. A typical use case is when a certain command should be executed, but does not need to be saved for later use. The commands are being executed on all the hosts specified on the command line simultaneously. These are the hosts we added to the inventory file in section 1 on page 8.
The syntax is as follows:

```
ansible <host-pattern> [-f forks] [-m module_name] [-a args]
```

The `-f` parameter specifies the level of parallelism, i.e. how many hosts to be contacted in parallel. The default of 5 can be changed to match the number of target systems, as well as available network and system ressources.
Modules specified by `-m` provide the actual functionality that Ansible should perform.
Arguments can be supplied to modules with the `-a` parameter.
Finally, a host pattern specifies on which machines the commands should be executed on.

Ansible Command Example

A simple example to demonstrate Ansible's functionality is using the `ping` module to verify that the target systems are responding:

```
$ ansible all -m ping
```

Here, we want to connect to all hosts listed in our inventory and execute the module called `ping` on them.
The output looks like this (most shells will even give you colors):

```
www1.example.com | SUCCESS => {
     "changed": false,
     "ping": "pong"
}
...
oracle.example.com | UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh.",
    "unreachable": true
}
```

This is a typical Ansible output, telling us whether the remote systems have changed their state somehow or if there were any messages produced when running the command.

Specifying Host Patterns

In the previous example, we used `all` as the host pattern to tell Ansible that we want to run the module on all hosts listed in the `/usr/local/etc/ansible/hosts` file. Another way to specify all hosts is using the Asterisk (`*`) character.
A single host can be provided by giving its name. Multiple hosts are separated by the colon character like this:

```
$ ansible oracle:postgres -m ping
```

Since we grouped our hosts into logical units based on their purpose (database servers, webservers), we can also issue commands to such a group by giving its name:

```
$ ansible webservers -m ping
```

More host patterns for Ansible are documented in `http://docs.ansible.com/ansible/intro_patterns.html`.

Overview

Transferring Files to Remote Systems (Upload)

A common task is to transfer files from the local to remote systems. This includes configuration files, templates, or other data of any kind. Ansible is able to SCP (secure copy) files in parallel to multiple machines. The copy module requires the source and destination as parameters.

```
$ ansible oracle -m copy -a "src=/home/ansible/hosts dest=/tmp/"
oracle | SUCCESS => {
    "changed": true,
    "checksum": "a645d99dd7ac54354df4fb61beaf6e38253e35f7",
    "dest": "/tmp/hosts",
    "gid": 0,
    "group": "wheel",
    "md5sum": "d6d598ab710d6e230e2a8d69fbbc34df",
    "mode": "0644",
    "owner": "ansible",
    "size": 63606,
    "src": "/home/ansible/.ansible/tmp/ansible-tmp-1468020/source",
    "state": "file",
    "uid": 1067
}
```

If the file is not present on the remote system, it will be copied. When the command is run twice, the file is not copied again due to the rule of idempotency described in section 1 on page 5.

Transferring Files from Remote Systems (Download)

We can also retrieve files from remote systems and store them locally in a directory tree, organized by hostname. The `fetch` module works similar to `copy`, but in the other direction.

```
$ ansible oracle -m fetch -a "src=/tmp/afile dest=/tmp/"
oracle | SUCCESS => {
    "changed": true,
    "checksum": "a645d99dd7ac54354fe4fb61beaf6e38253e45f7",
    "dest": "/tmp/oracle/tmp/afile",
    "md5sum": "d6d298ab710d6e1430e1a8d69fbbc76de",
    "remote_checksum": "a645d99dd7ac54254ec4fc61beaf6e38253e45f7",
    "remote_md5sum": null
}
```

After the file transfer has finished, the directory we specified in `dest` will contain directories named after each host we targeted, with a subdirectory `/tmp/` that contains `afile`, according to our `src`.

```
/tmp
└── /oracle
    └── /tmp
        └── afile
```

Overview

Package Management

A common task for configuration management systems is to install, update, delete, and configure application software on the target operating system. Usually, package managers are provided by the OS vendor or by the application or programming language when there is a lot of optional or third-party software available. Ansible provides the same functionality and can sometimes abstract away the complexities of using the package manager directly.
Making changes to packages usually involves administrative permissions, so we'll look at what kind of options Ansible offers here as well.

Basic Package Management Command

Here, we will show how to use the package module that combines many package managers of various Unix distributions to manage applications. Let's say we just installed the operating system on the webserver systems, but have not installed any webserver software yet. We'll use Ansible for that:

```
$ ansible webservers -m package -a "name=nginx state=present"
```

In this example, we install the Nginx[1] webserver. Ansible compares the state of the system, does not detect any installed version of nginx and performs the necessary steps (based on what the package manager tells it to do) to make sure it is present afterwards.
Updating a package is supported as well. We do not have to know about any specific version number and let Ansible figure that out for us.

```
$ ansible webservers -m package -a "name=nginx state=latest"
```

Removing packages is also possible:

```
$ ansible webservers -m package -a "name=nginx state=absent"
```

[1] https://nginx.org

Ansible Command Permissions

Installing packages or making other kinds of administrative changes normally requires `root` privileges. By default, Ansible defaults to running with the privileges of the user that invoked the command. In slide 10, we defined who the remote user is that is executing commands. We can override that by providing the username on the command-line after `-u`:

```
$ ansible webservers -m package -a "name=nginx state=absent" -u root
```

This will ask for the root password and check whether the current user is allowed to switch to the `root` user.

Typically, an unprivileged user like `ansible` is configured to use a passwordless privilege escalation method such as `sudo` to temporarily gain higher privileges. That requires that the `ansible` user is part of the `sudo` group (see `/etc/group`). That way, Ansible can ask for the user's password when needed to verify it against sudo before executing the privileged commands. The two options to provide are `-b` (become) and `-K` (ask for the passphrase to become that user):

```
$ ansible webservers -Kbm package -a "name=nginx state=present"
```

Overview

File Modifications

Ansible can perform actions on files like changing their permissions and ownership. It can also make changes to the file contents such as adding lines at a specific location or replacing certain strings. This is especially helpful with configuration files that are installed as part of a package or the operating system.
This will create a symbolic link in the directory `/var/tmp` to `/tmp`.

```
$ ansible dbservers -Kbm file -a "src=/tmp dest=/var/tmp state=link"
```

The next command creates a directory `/opt/db2` with `rwxrwxr-x` on the host (or group) called `db2`.

```
$ ansible db2 -Kbm file -a "path=/opt/db2 state=directory mode=775"
```

Owner and group, as well as permissions (see `mode` above) can be set directly in most modules. This helps keeping things logically together, without having to run a second or third command to set permissions and/or ownership.

Editing Files

A common problem for system administrators is to edit configuration files, without adding a certain string multiple times. Although this is usually checked by the configuration file parser, it is better to avoid in the first place. This is where idempotency can help maintain order and avoid clutter.
For example, we might be dealing with a configuration file with a similar style as our Ansible inventory.

```
[general]
setting1 = true
setting2 = "a string"

[special]
option1 = 123
option2 = abc
```

Changing the Configuration File

The module `lineinfile` searches for a line in a file based on a regular expression that is either absent or present.

```
$ ansible postgres -m lineinfile -a "dest=/tmp/file.ini
 state=absent regexp=^setting2"
postgres | SUCCESS => {
    "backup": "",
    "changed": true,
    "found": 1,
    "msg": "1 line(s) removed"
}
```

Afterwards, the file looks like this:

```
[general]
setting1 = true

[special]
option1 = 123
option2 = abc
```

Inserting Lines into the Configuration File

When a match is found by lineinfile, a parameter `insertafter` or `insertbefore` determines where a new line should be placed.

```
$ ansible postgres -m lineinfile -a "dest=/tmp/file.ini
 insertafter='^setting1' line='setting2 = false'"
postgres | SUCCESS => {
    "backup": "",
    "changed": true,
    "msg": "line added"
}
```

Now, the file contains these lines:

```
[general]
setting1 = true
setting2 = false

[special]
option1 = 123
option2 = abc
```

Replacing Strings in a file

Suppose we want to replace a string in a file based on a pattern without adding an extra line or deleting it first. For example, we want to exchange the line `option1 = 123` with `option1 = 321` in `file.ini`.

```
...
[special]
option1 = 123
option2 = abc
```

Ansible provides a module called `replace` to do this:

```
$ ansible postgres -m replace -a "dest=/tmp/file.ini
 regexp='^option1 = 123' replace='^option1 = 321'"
```

Afterwards, the file looks like this:

```
...
[special]
option1 = 321
option2 = abc
```

Overview

What are Playbooks?

While the `ansible` command allows ad-hoc commands to the issued to target systems, playbooks allow for more complex and larger number of actions to be done on a host. Similar to scripts, they can define and use variables, execute actions based on them and define a number of tasks to be executed in a specific order. These playbooks are typically installing machines for production use after the operating system is installed and the SSH access has been configured. Ansible playbooks are written in a language called YAML (*y*et *a*nother *m*arkup *l*anguage), which has a minimal yet powerful enough syntax for this use. By learning to write playbooks, we'll also get to know YAML.

Our first Playbook

Each playbook is a text file with the suffix `.yml` and contains at least one or more **plays** in a list. Each play is executed against a defined set of **hosts** and executes one or more **tasks** on it. A task is running a module, similar to what we did with the `ansible` ad-hoc command. A simple playbook is listed below:

```
---
- name: My first playbook
  hosts: dbservers
  tasks:
    - name: test connection
      ping:
...
```

A playbook is executed using `ansible-playbook` with the path to the playbook passed as a parameter:

```
$ ansible-playbook ping.yml
```

Executing our First Playbook

```
PLAY [My first playbook] *******************************************

TASK [setup] *******************************************************
ok: [mysql]
ok: [postgres]
ok: [db2]

TASK [test connection] *********************************************
ok: [postgres]
ok: [mysql]
ok: [db2]

PLAY RECAP *********************************************************
mysql            : ok=2    changed=0    unreachable=0    failed=0
db2              : ok=2    changed=0    unreachable=0    failed=0
postgres         : ok=2    changed=0    unreachable=0    failed=0
```

From the output ordering, we see that tasks are executed in parallel and return any results when available, without waiting for each other. Each time a play runs, it gathers facts about the target hosts (`setup`). It can be turned off with `gather_facts: false` to speed up the play.

Example Playbook to Allow a User to Log in via SSH

```
$ cat ssh-access.yml
- name: "Allow {{user_id}} to log in via SSH"
  gather_facts: false
  hosts: '{{ host }}'
  tasks:
    - name: Adding the user {{user_id}} to the AllowUsers line in sshd_config
      replace:
        backup: no
        dest: /etc/ssh/sshd_config
        regexp: '^(AllowUsers(?!.*\b{{ user_id }}\b).*)$'
        replace: '\1 {{ user_id }}'

    - name: Restarting SSH service
      service:
        name: sshd
        state: restarted
```

Example Playbook to Allow a User to Log in via SSH

```
$ cat ssh-access.yml
- name: "Allow {{user_id}} to log in via SSH"
  gather_facts: false
  hosts: '{{ host }}'
  tasks:
    - name: Adding the user {{user_id}} to the AllowUsers line in sshd_config
      replace:
        backup: no
        dest: /etc/ssh/sshd_config
        regexp: '^(AllowUsers(?!.*\b{{ user_id }}\b).*)$'
        replace: '\1 {{ user_id }}'

    - name: Restarting SSH service
      service:
        name: sshd
        state: restarted
```

Execute with:

```
$ ansible-playbook -Kb ssh-access.yml -e 'host=bsdhost1 user_id=joe'
```

Making Playbooks work more like shell scripts

Playbooks are really nothing more than Python scripts and can be executed the same way as a shell script. As we saw before, a normal invocation looks like this:

```
$ ansible-playbook myplaybook.yml
```

Adding an interpreter line to the beginning of the playbook and setting the executable bit allows us to run it like a shell script:

```
$ head -n 1 myplaybook.yml
#!/usr/local/bin/ansible-playbook
$ chmod +x myplaybook.yml
$ ./myplaybook.yml
```

Overview

YAML

YAML is easy to grasp, but somewhat difficult to master. This is because it depends on the proper indentation and number of spaces to correctly parse and subsequently execute the instructions that are coded in it by the YAML programmer. One benefit of YAML is that it is easy for humans to read and uses less syntactic sugar like JSON or XML.
If you are using vim as your editor, this setting in `.vimrc` will help with the proper YAML formatting:

```
autocmd FileType yaml setlocal ts=2 sts=2 sw=2 expandtab
```

YAML Syntax

YAML files start with three dashes (`---`) on a single line, similar to the `#!/bin/sh` definition at the beginning of shell scripts. At the end of a YAML file, three `...` indicate the end of the script. Ansible will not complain if they are omitted, but generally, it is good style to add them to let other programs parsing them know that this is a proper YAML file.

Comments begin with # and go until the end of the line.

```
--- # this starts the YAML file
 This is a string
 "This is a string in quotes with 'single' quotes"
...
```

There is no need to quote **strings**, except for cases where variables are used, a colon (`:`) is contained, or regular quotes are required. In this case, single quotes can enclose double quotes from a normal string.

Any of the following **Booleans** can be used:

`true`, `True`, `TRUE`, `yes`, `Yes`, `YES`, `on`, `On`, `ON`, `y`, `Y`
`false`, `False`, `FALSE`, `no`, `No`, `NO`, `off`, `Off`, `OFF`, `n`, `N`

YAML Lists

Lists or sequences are what arrays are in other programming languages like C(++)/Java: a collection of values. They must be delimited with hyphens, a space and must be on the same indentation level:

```
list:
  - item1
  - item2
  - item3
```

An alternative way to list them is:

```
[item1, item2, item3]
```

YAML Dictionaries

Dictionaries or mappings in YAML are typical key-value pairs. They are delimited from each other by a colon (`:`) and a space:

```
person:
  name: John Miller
  date: 31/12/2016
  age: 38
```

Another way of specifying these mappings looks like this:

```
(name: John Miller, date: 31/12/2016, age: 38)
```

Lists and Dictionaries can be mixed, starting with a new indendation level. This is similar to multiple levels of { and } in C/Java.

Wrapping Long YAML Lines

Playbooks that contain a lot of arguments for modules might run over the available line space. To visually order them, the line folding characters > (ignores newlines) and | (pipe, includes newlines) can be used:

```
address: >
    Department of Computer Science,
    University of Applied Sciences Darmstadt
```

More information on YAML's syntax can be found at `http://docs.ansible.com/ansible/YAMLSyntax.html`.

Overview

Variables in Playbooks

Playbooks can become more dynamic by using variables and thus more powerful in what they can do. For example, a playbook can do certain actions based on what values a host variable has, like number of CPUs or available disk space.
Variables can come from different places:

- Declared in the inventory file as host and group variables
- Defined in the playbook
- Defined in a variables file that multiple playbooks can include and use
- From the host (fact gathering)
- Results from module runs
- Lookups (external sources)

A variable is accessed in a playbook using the `{{variable_name}}` syntax. Variable names can contain letters, numbers, and underscores and should begin with a letter.
We will look at each of the above variable definitions.

Inventory Variables for Hosts and Groups

Our inventory file (remember slide 8) contained a list of webservers. Let's say we want to store a variable that defines the default web server port for each of the hosts. Such an inventory file will look like this:

```
[webservers]
www1.example.com wwwport=8080
www2.example.com wwwport=80
```

The following playbook called `wwwvar.yml` will output the variable for the first host:

```
---
- name: This play will output the wwwport inventory variable
  gather_facts: false
  hosts: www1.example.com
  tasks:
    - name: Show the variable value of wwwport
      debug: var=wwwport
...
```

The debug module is useful to echo the values of variables to stdout.

Running the Playbook

When the `wwwvar.yml` playbook is run, the output is:

```
1 PLAY [This play will output the wwwport inventory variable] ******

3 TASK [Show the variable value of wwwport] ************************
4 ok: [www1.example.com] => {
5     "wwwport": 8080
6 }

8 PLAY RECAP ********************************************************
9 www1                     : ok=1    changed=0    unreachable=0    failed=0
```

The above output has in line 5 the correct value for the port that we have set earlier in the inventory file for that host.

Using Group Variables

If we want to set a variable for all the hosts in a group, we need a special section in our inventory file. Suppose we want all webservers to listen on the same port. We add a section like this:

```
[webservers:vars]
wwwport=8000
```

In our playbook, we only need to change the `hosts:` (line 4) to the `webservers` group. The rest of the play remains unchanged.

```
1 ---
2 - name: This play will output the wwwport inventory variables
3   gather_facts: false
4   hosts: webservers
5   tasks:
6     - name: Show the variable value of wwwport
7       debug: var=wwwport
8 ...
```

Running the Playbook

```
PLAY [This play will output the wwwport inventory variables] *****
TASK [Show the variable value of wwwport] ************************
ok: [www1.example.com] => {
    "wwwport": 8000
}

ok: [www2.example.com] => {
    "wwwport": 8000
}

PLAY RECAP ********************************************************
www1.example.com  : ok=1    changed=0    unreachable=0    failed=0
www2.example.com  : ok=1    changed=0    unreachable=0    failed=0
```

Global variables that should be part of all hosts can also be defined in the inventory by using the `all` placeholder:

```
[all:vars]
dns_server=dns1.mycorp.com
```

Defining Variables in Playbooks

Playbooks can define a section called `vars:`, that are available in the whole playbook to use. If we wanted to define our default webserver port in the playbook instead of the inventory, we have to write it like this:

```
1 - name: The play will output the wwwport playbook variable
2   gather_facts: false
3   hosts: www1.example.com
4   vars:
5     wwwport: 8080
6   tasks:
7     - name: Show the playbook variable wwwport
8       debug: var=wwwport
```

Line 4 defines the variable section, the indented line below has the variable we want to declare. The output is similar to the one from the previous slide.

Registering Variables for Later Use

Things don't often go as expected when running playbooks, so we need a way to store variables from hosts to react to them later based on what value they hold. We could also retrieve information from running commands and store them in a playbook variable to use it in one of the next playbook steps.
In this example, we use the `command` module to execute the `id` command on each host for the ansible user. Then, we output the variable `the_id` using `debug` to see how it is structured:

```
1 - name: The play executes the id command and stored the return value
2   gather_facts: false
3   hosts: dbservers
4   tasks:
5     - name: get the id of the ansible user
6       command: id ansible
7       register: the_id
8     - debug: var=the_id
```

Looking at the Return Values

```
PLAY [The play executes the id command and stored the return value]
TASK [get the id of the ansible user] ******************************
TASK [debug] *******************************************************
ok: [postgres] => {
    "the_id": {
        "changed": true,
        "cmd": [
            "id",
            "ansible"
        ],
        "delta": "0:00:00.014088",
        "end": "2016-07-25 09:39:21.460684",
        "rc": 0,
        "start": "2016-07-25 09:39:21.446596",
        "stderr": "",
        "stdout": "uid=50000(ansible) gid=50008(ansible)
  groups=50008(ansible)",
        "stdout_lines": [
            "uid=50000(ansible) gid=50008(ansible)
  groups=50008(ansible)"
        ],
        "warnings": []
    }
}
```

Using the Return Value's Variables

We can access individual members of the `the_id` array using the variable we defined (`the_id`) and the dot operator followed by the member name. In this example, we use the returned value in the text of the new `name:` section below our original playbook content to see the value.

```
1 - name: The play executes the id command and stores the return value
2   gather_facts: false
3   hosts: dbservers
4   tasks:
5     - name: get the id of the ansible user
6       command: id ansible
7       register: the_id
8     - debug: var=the_id
9       name: The command returned {{ the_id.stdout_lines }}
```

The relevant section of the output looks like this:

```
 TASK [The command returned [u'uid=50000(ansible) gid=50008(ansible)
groups=50008(ansible)']] ***
```

Sharing Variables among Playbooks with an External File

As playbooks grow, re-use of variables becomes more important. Also, big playbooks can have a long list of variables and make it difficult to concentrate on the tasks. To solve that problem, Ansible playbooks can include variables from a external file. By creating a `vars` subdirectory and putting a `vars.yml` file in there, we can move all variables from the playbook into that file. To include that file, we change our ansible playbook to look like this:

```
- hosts: dbservers
  vars_files:
    - ./vars/vars.yml
...
```

Access to the external variables is the same as if they were playbook-local variables. Multiple files can be included this way. Ansible does not care if the variables are used in the playbook or not. They are available like any other. The content of the `vars.yml` file looks like this:

```
variable: value
```

It's simple to use that file, since we can just move all playbook-local variables into that file.

What about Passwords and other Sensible Information?

Playbooks use passwords to configure hosts and services, which are sensible information that should not be stored in a clear text file. Ansible solves this by using Vault. The vault will store all information in encrypted form, so that they can be transmitted and shared safely. Without the passphrase, no one can decrypt the file. When running the playbook, the passphrase must be provided so that the playbook can access the information stored in the vault.
We'll create a new vault using the `ansible-vault` command and store it next to our `vars.yml` file:

```
$ ansible-vault create ./vars/vault.yml
```

You'll be asked to set a passphrase that you should not forget as it is used to decrypt the file later. The content of the vault is the same as our `vars.yml` file, just with passwords. Good practice is to use a `vault_` prefix for variable names which helps to determine them from regular variables and where they are located. After saving and exiting the file, the contents will be encrypted.

Accessing the Vault Content

To view the unencrypted content of the vault, the `show` command can be used (the passphrase is needed):

```
$ ansible-vault view ./vars/vault.yml
```

To make changes to the variables stored in the vault, `edit` can be used.

```
$ ansible-vault edit ./vars/vault.yml
```

To use the vault variables in a playbook, include it first:

```
- hosts: dbservers
  vars_files:
    - ./vars/vars.yml
    - ./vars/vault.yml
```

Then, you need to provide the `--vault-id @prompt` parameter to your playbook execution:

```
./myplaybook.yml --vault-id @prompt
Vault password (default):
```

Vaults can have different names/ids to distinguish them from each other. Once the password is entered, the playbook will run as normal and use the protected variables from the vault.

Gathering Facts from the Host as Variables

Ansible inserts an implicit task into each playbook that begins to gather various facts from the targer host. This can be suppressed (and has been so far) using the line `gather_facts: false` in the playbook.
All variables from a single host can be accessed using the `setup` module:

```
$ ansible hostname -m setup
```

Typical facts include:

Network information: IPv4/v6 addressses, gateway, DNS, interface, etc.

Operating System: Distribution release, versions, environment variables

Hardware information: CPU, RAM, disk space, devices, available swap

Date and time: day, month, year (in various formats), weekday, time

Ansible information: Ansible user, version, nodename, package manager

Even more information is available when the remote system has `facter` and/or `ohai` installed.

Filtering Gathered Facts

The full list of facts is often too much information since we are often interested in a single variable only. The `filter` option allows to limit the results to a certain variable:

```
$ ansible postgres -m setup -a "filter=ansible_distribution*"
postgres | SUCCESS => {
    "ansible_facts": {
        "ansible_distribution": "FreeBSD",
        "ansible_distribution_major_version": "11",
        "ansible_distribution_release": "11.1-RELEASE-p1",
        "ansible_distribution_version": "11.1"
    },
    "changed": false
}
```

Without the asterisk (*), sub-keys are omitted and only `ansible_distribution` is returned.

```
$ ansible postgres -m setup -a "filter=ansible_distribution"
postgres | SUCCESS => {
    "ansible_facts": {
        "ansible_distribution": "FreeBSD"
    },
    "changed": false
}
```

Variables from the Command Line

Playbook variables can be overridden on the command line in case the variables in the playbook should not be used for the current run.

```
- name: Echo the message from the command line
  hosts: www1.example.com
  vars:
    message: "empty message"
  tasks:
    - name: echo the message
      debug: msg="{{ message }}"
```

The -e option can override the message variable in the playbook for the current invocation:

```
$ ansible-playbook message.yml -e "message=Hello"
ok: [www1.example.com] => {
    "msg": "Hello"
}
```

When spaces are part of the variable value, single quotes need to be used:

```
$ ansible-playbook message.yml -e 'message=Hello world!'
```

Lookups

Various external sources can provide variables to playbooks. Ansible provides so called lookups to query external sources like files (text and JSON), environment variables, Redis, DNS, Jinja2 templates, MongoDB databases, etc. This list is not exhaustive and many more plugins are available to query a certain source to provide Ansible with variables and values.
The syntax is as follows:

```
lookup(lookup-type, variable-to-lookup or command-to-execute)
```

For example, to lookup the `HOME` environment variable, we would use:

```
lookup('env','HOME')
```

A complete list of lookups with examples is available from `http://docs.ansible.com/ansible/latest/playbooks_lookups.html`.

Reading usernames from a file

In the following playbook, usernames are read from a file `users.txt` and echoed onto the screen using `with_lines`. The file contents can then be used later in the script to add or manipulate the user accounts on the system.

```
#!/usr/local/bin/ansible-playbook
- name: "Reading file contents from {{ file }}"
  gather_facts: false
  hosts: postgres
  tasks:
    - name: "Lines from {{ file }}"
      debug: msg="{{ item }} came from the file {{ file }}".
      with_lines: "cat {{ file|quote }}"
```

To sanitize inputs from the file we append `|quote` to the variable `file`. The content of the `users.txt` file looks like this:

```
$ cat users.txt
jack
jill
```

Results from Running the File Lookup Playbook

```
$ ./lookup.yml -e 'host=postgres file=users.txt'
PLAY [Reading file contents from users.txt] ********************************
ok: [postgres] => (item=jack) => {
    "changed": false,
    "item": "jack",
    "msg": "\"jack came from the file users.txt\"."
}
ok: [postgres] => (item=jill) => {
    "changed": false,
    "item": "jill",
    "msg": "\"jill came from the file users.txt\"."
}

TASK [Lines from users.txt] ************************************************

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

Assigning Lookup Values to Playbook Variables

Instead of using the debug module to echo the files content, we can assign it to a local playbook variable for later use.

```
vars:
  user_file: "{{ lookup('file', 'users.txt') }}"

tasks:
  - user:
      name: "{{ item }}"
    with_lines: "{{ user_file|quote }}"
```

Overview

Handlers

So far, our playbooks have been running through the tasks from the beginning to the end. Sometimes, it makes sense to run a task only when there has been a change. The concept of idempotency that Ansible follows already checks for things that should be (and have not yet been) done, but there is no dependency between playbook tasks yet. There are often cases when the successful execution of a task should trigger another task to run. If there are no changes being made (because they have already been applied), there is no need to run certain other tasks that follow it.

A typical example are services that should only be restarted when there were actual changes. Why restart a service (in production) if nothing has changed? Such a change could be modifications of a configuration file (Apache, SSH) or the presence of new files in a directory (document root, fileserver).

For these cases, handlers are available that only run when a task returns the state `changed`, as opposed to `ok` when changes have already been applied in a previous run. Handlers can react to these changes, allowing for additional tasks to run when changes occur and do not trigger extra tasks if no changes were made.

Note: Handlers run *after* all the tasks of a playbook were executed.

Extending the SSH Playbook to use handlers

```
1  #!/usr/local/bin/ansible-playbook
2  - name: "Enable SSH access on {{host}} for user {{user_id}}"
3    hosts: '{{host}}'
4    tasks:
5      - name: "Adding user {{user_id}} to the AllowUsers line in sshd_config"
6        replace:
7          backup: no
8          dest: /etc/ssh/sshd_config
9          regexp: '^(AllowUsers(?!.*\b{{ user_id }}\b).*)$'
10         replace: '\1 {{user_id}}'
11         validate: 'sshd -T -f %s'
12       notify: "Restart SSH"

14   handlers:
15       - name: "Restarting SSH"
16         service: name=ssh state=restarted
17         listen: "Restart SSH"
```

The task defines a `notify` part in line 12 that calls a handler with the same description or name in line 17. When the state of a task is changed, a handler can `listen` for these changes and run the tasks defined in the handler. Otherwise, the handler is not executed.

Running the Playbook with the Handler

```
PLAY [Enable SSH access on postgres for user foo] ***************************

TASK [Adding user foo to the AllowUsers line in sshd_config] ****************
changed: [postgres]

RUNNING HANDLER [Restarting SSH] ********************************************

TASK [Restarting SSH] *******************************************************
changed: [postgres]

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

Running the playbook a second time, the handler is not called since the task is in the ok state.

```
PLAY [Enable SSH access on postgres for user foo] ***************************

TASK [Adding the user foo to the AllowUsers line in sshd_config] *************
ok: [postgres]

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

Conditional Execution

Conditional execution is helpful when tasks should only run if certain conditions are met. The conditions are typically defined by variable values. Instead of using if-statements, playbooks use `when` to define a condition that should be checked upon execution of a task.
For example, the following playbook should only execute the task to restart the sshd service for operating systems that are FreeBSD (detected by `gather_facts`). The task looks like this:

```
#!/usr/local/bin/ansible-playbook
- name: Conditionally restart SSH when FreeBSD OS is detected
  gather_facts: true
  tasks:
    - name: "Restart SSH on FreeBSD"
      service:
        name: sshd
        state: restarted
      when: ansible_os_family == "FreeBSD"
```

There is no need to use {{}} around variable names in `when` statements. Ansible is smart enough to know that these are variables and evaluates them accordingly.
More about conditional executions with the `when` directive can be found here:
`http://docs.ansible.com/ansible/latest/playbooks_conditionals.html`

Overview

Loops in Playbooks

Loops can help a great deal when a certain action should be repeated multiple times. Who wants to create 100 users from the command line manually when we can solve this problem with a short loop statement?
To start with a simple example, consider adding two users, which is already good to automate to not repeat yourself. Here, we create two users `userA` and `userB` based on the list we provide.

```
- name: add two users
  user: name={{ item }} state=present groups=wheel
  with_items:
    - userA
    - userB
```

Loops over a sequence

In this example, we want to create 100 users (`user1`, `user2`, ..., `user100`) without listing them all in the `with_items` list one by one (tedious to type). To do that, we can make use of the `with_sequence` construct, which acts like a `for` loop in languages like C and Java.

```
- user: name={{ item }} state=present groups=wheel
  with_sequence: start=1 end=100 format=user%02x
```

The `format=` definition specifies what kind of numerical value should be used (decimal, hexadecimal (0x3f8), or octal (0775)).
We can also define a different increment with the `stride` option. We use this to create only even-numbered users:

```
- user: name={{ item }} state=present groups=wheel
  with_sequence: start=0 end=100 stride=2 format=user%02x
```

Nested Loops

So far, we only created users that got added to the same group `wheel`. If we want to define which user should be added to which group, we have two options. We can define the group together with the user as subkeys:

```
- name: Add several users and add them to their group
  user: name={{ item.name }} state=present groups={{ item.groups }}
  with_items:
    - { name: 'userA', groups: 'wheel' }
    - { name: 'userB', groups: 'operator' }
```

We can access the name we gave to the key-value pair by adding it to the end of the `item` keyword, separated by a dot (`item.groups`).

Nested Loops

The second way we can solve this is to use a nested loop. This is especially useful when the user should be added to multiple groups:

```
- name: Add several users and add them to multiple groups
  user: name={{ item[0] }} state=present groups={{ item[1] }}
  with_nested:
    - [ 'userA', 'userB' ]
    - [ 'wheel', 'operator', 'www' ]
```

This represents a two-dimensional array and to access an element from either list, we provide the number in brackets. Thus, `item[0]` represents `userA` first and after all nested elements (`item[1]` = `wheel, operator, www`) were processed, we do the same with `userB`.

Overview

A Complete Example - Deploying a Webserver

We'll sum up what we've learned so far in a scenario for deploying a webserver on a target system. The following steps are typically necessary to deploy a webserver in production:

1. Install the webserver application binaries
2. Configure the webserver (document root, ports to listen on, etc.)
3. Copy webpages or web applications to the document root directory
4. Start the service for the webserver

We will create a playbook that will cover all these steps, so that we will have a fully functional webserver. The use of variables in our playbook will be based on what we've covered so far. Of course, a webserver does usually require a secure environment to run in, SSL certificates, a database for application data storage, and many other things to run in production. However, we'll omit most of these to keep the example focused enough to not run out of proportions.

Installing the Webserver Application Binaries

We will be using Nginx as the webserver for our little project. Our document root is located under `/var/www` and has subdirectories for each webpage hosted on the server. We will run the webserver under the `www` user and group, which may or may not be installed as part of the webserver installation. The web application is split into several HTML files for now, so we don't need any fancy web application software like PHP, Python, or Ruby on Rails.
Already, we can define the following variables in our playbook:

```
vars:
  server: nginx
  user: www
  group: {{ user }}
  docroot: /var/www
  project: demo
  projectdir: /home/{{ project }}/web
  projectfiles:
    - index.html
    - impressum.html
    - about.html
```

Writing the Playbook, Part I

Using our variables, we start by writing the tasks for installing the webserver on a Ubuntu system using the apt package manager. We also create a user and group (`www`), ensure the document root directory is created, and set permissions for that user on it:

```
---
- name: The webserver playbook
  hosts: www3.example.com
  vars:
    ...
  tasks:
    - name: Install {{ server }} from packages
      apt: pkg={{ server }} state=present
```

Writing the Playbook, Part II

In this step, we configure the webserver. This can be solved using templates, where variables from the playbook are replaced with the actual values. These are the webserver IP address, the port to listen on and the document root directory. To keep this example easy, we will use the following file as a template for `nginx.conf`:

Nginx Configuration Template

```
user  nobody;
worker_processes  1;

#error_log  logs/error.log;
#pid   /run/nginx.pid;

events {
    worker_connections  1024;
}

http {
    include /usr/local/etc/nginx/mime.types;

    server {
        listen       80;
        server_name  localhost;
        location / {
            root   /var/www/;
            index  index.html index.htm;
        }
        include /usr/local/etc/nginx/sites-enabled/*;
    }
}
```

Creating the template

```
user  {{ user }};
worker_processes  1;

#error_log  logs/error.log;
#pid   /run/nginx.pid;

events {
    worker_connections  1024;
}

http {
    include /usr/local/etc/nginx/mime.types;

    server {
        listen       80;
        server_name  localhost;
        location / {
            root   {{ docroot }};
            index  index.html index.htm;
        }
        include /usr/local/etc/nginx/sites-enabled/*;
    }
}
```

Deploying the Template to the Target Machine

Ansible has a `template` module that can deploy Jinja2 templates to a target machine, replacing the inline variables with the values defined in the playbook.
We store the template as `nginx.conf.j2` as a Jinja template on our deployment machine. The playbook line for it looks like that:

```
- name: "Deploy nginx.conf template"
  template: src=/deployment/nginx.conf.j2 \
    dest=/usr/local/etc/nginx.conf \
    owner={{ user }} group={{ group }} validate='nginx -tc %s'
```

The module requires the `src` and `dest` to be specified in order to work, the rest is optional. We set the ownership (owner and group) and run a command to validate the resulting `nginx.conf` before using it with the `-t` parameter to `nginx`. The `%s` contains the path to the file to validate. That way, we make sure to never deploy a new configuration that nginx won't accept.

Writing the Playbook, Part III

Now that we have nginx installed and provided a working configuration template filled with the variable values from the playbook, it is time to create the document root directory and copy the HTML files to the target host. To achieve this, we use the file and copy modules.

```
- name: Create the document root directory
  file: path={{ docroot }} state=directory mode=0755
    owner={{ user }} group={{ group }}
```

The above instructs Ansible to create a folder with the proper permissions and owner in the `/var/www` directory as defined in our playbook variables.

Copying the files to the document root directory

The copy module will transfer the files to the directory we just created. Since we have multiple HTML files, we will use a list in our task specification like this:

```
- name: copy files to the document root
  file: src='{{ projectdir }}/{{ projectfiles }}' dest={{ docroot }}
    owner={{ user }} group={{ group }}
```

The copy modules requires a src and destination directory to work with and can optionally set owner and permissions.

Writing the Playbook, Part IV

We can start the web server now to serve the files we just copied. To do that, we use the Ansible service module.

```
- name: "Restarting nginx web server"
  service: name=nginx state=restarted
```

We can now use a browser to look at the files served by nginx.

Further Information

Ansible Documentation
BSD Support
http://docs.ansible.com/ansible/intro_bsd.html

Ansible Documentation
Introduction to Ad-Hoc Commands
http://docs.ansible.com/ansible/intro_adhoc.html

Ansible Documentation
Module Index
http://docs.ansible.com/ansible/modules_by_category.html

Knp University Ansible Online Course
Hosts & the Inventory File
https://knpuniversity.com/screencast/ansible/hosts-inventory