Community Experience Distilled

Mastering Ansible

Design, develop, and solve real world automation and orchestration needs by unlocking the automation capabilities of Ansible

Jesse Keating

[PACKT] PUBLISHING open source* community experience distilled

Mastering Ansible

Design, develop, and solve real world automation and orchestration needs by unlocking the automation capabilities of Ansible

Jesse Keating

[PACKT] PUBLISHING
open source
community experience distilled

BIRMINGHAM - MUMBAI

Mastering Ansible

First published: November 2015

Production reference:1191115

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B32PB, UK.

ISBN 978-1-78439-548-3

www.packtpub.com

Credits

Author
Jesse Keating

Reviewers
Ryan Eschinger
Sreenivas Makam
Tim Rupp
Sawant Shah
Patrik Uytterhoeven

Acquisition Editor
Meeta Rajani

Content Development Editor
Zeeyan Pinheiro

Technical Editor
Rohan Uttam Gosavi

Copy Editor
Pranjali Chury

Project Coordinator
Nidhi Joshi

Proofreader
Safis Editing

Indexer
Monica Ajmera Mehta

Graphics
Disha Haria

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

About the Author

Jesse Keating is an accomplished Ansible user, contributor, and presenter. He has been an active member of the Linux and open source communities for over 15 years. He has first-hand experience with a variety of IT activities, software development, and large-scale system administration. He has presented at numerous conferences and meet-ups, and he has written many articles on a variety of topics.

His professional Linux career started with Pogo Linux as a Lead Linux Engineer handling many duties, including building and managing automated installation systems. For 7 years, Jesse served at the Fedora Release Engineer as a senior software engineer at Red Hat. In 2012, he joined Rackspace to help manage Rackspace's public Cloud product, where he introduced the use of Ansible for the large-scale automation and orchestration of OpenStack-powered Clouds. Currently, he is a senior software engineer and the OpenStack release lead at Blue Box, an IBM company, where he continues to utilize Ansible to deploy and manage OpenStack Clouds.

He has worked as technical editor on *Red Hat Fedora and Enterprise Linux 4 Bible*, *A Practical Guide to Fedora and Red Hat Enterprise Linux 4th Edition*, *Python: Create-Modify-Reuse*, and *Practical Virtualization Solutions: Virtualization from the Trenches*. He has also worked as a contributing author on *Linux Toys II*, and *Linux Troubleshooting Bible*. You can find Jesse on Twitter using the handle `@iamjkeating`, and find his musings that require more than 140 characters on his blog at `https://derpops.bike`.

Acknowledgment

I'd like to thank my wife—my partner in crime, my foundation, my everything. She willingly took the load of our family so that I could hide away in a dark corner to write this book. Without her, it would never have been done. She was also there to poke me, not so gently, to put down the distractions at hand and go write! Thank you Jessie, for everything. I'd like to to thank my boys too, Eamon and Finian, for giving up a few (too many) evenings with their daddy while I worked to complete one chapter or another. Eamon, it was great to have you so interested in what it means to write a book. Your curiosity inspires me! Fin, you're the perfect remedy for spending too much time in serious mode. You can always crack me up! Thank you, boys for sharing your father for a bit.

I'd also like to thank all my editors, reviewers, confidants, coworkers past and present, and just about anybody who would listen to my crazy ideas, or read a blurb I put on the page. Your feedback kept me going, kept me correct, and kept my content from being completely adrift.

About the Reviewers

Ryan Eschinger is an independent software consultant with over 15 years of experience in operations and application development. He has a passion for helping businesses build and deploy amazing software. Using tools such as Ansible, he specializes in helping companies automate their infrastructure, establish automated and repeatable deployments, and build virtualized development environments that are consistent with production. He has worked with organizations of all shapes, sizes, and technical stacks. He's seen it all—good and bad—and he loves what he does. You can find him in one of the many neighborhood coffee shops in Brooklyn, NY, or online at `http://ryaneschinger.com/`.

Sreenivas Makam is currently working as a senior engineering manager at Cisco Systems, Bangalore. He has a master's degree in electrical engineering and around 18 years of experience in the networking industry. He has worked on both start-ups and big established companies. His interests include SDN, NFV, Network Automation, DevOps, and Cloud technologies. He also likes to try out and follow open source projects in these areas. You can find him on his blog at `https://sreeninet.wordpress.com/`.

Tim Rupp has been working in various fields of computing for the last 10 years. He has held positions in computer security, software engineering, and most recently, in the fields of Cloud computing and DevOps.

He was first introduced to Ansible while at Rackspace. As part of the Cloud engineering team, he made extensive use of the tool to deploy new capacity for the Rackspace Public Cloud. Since then, he has contributed patches, provided support for, and presented on Ansible topics at local meetups.

He is currently stationed at F5 Networks, where he is involved in Solution development as a senior software engineer. Additionally, he spends time assisting colleagues in various knowledge-sharing situations revolving around OpenStack and Ansible.

I'd like to thank my family for encouraging me to take risks and supporting me along the way. Without their support, I would have never come out of my shell to explore new opportunities. I'd also like to thank my girlfriend for putting up with my angry beaver moments as I balance work with life.

Sawant Shah is a passionate and experienced full-stack application developer with a formal degree in computer science.

Being a software engineer, he has focused on developing web and mobile applications for the last 9 years. From building frontend interfaces and programming application backend as a developer to managing and automating service delivery as a DevOps engineer, he has worked at all stages of an application and project's lifecycle.

He is currently spearheading the web and mobile projects division at the Express Media Group—one of the country's largest media houses. His previous experience includes leading teams and developing solutions at a software house, a BPO, a non-profit organization, and an Internet startup.

He loves to write code and keeps learning new ways to write optimal solutions. He blogs his experiences and opinions regarding programming and technology on his personal website, `http://www.sawantshah.com`, and on Medium, `https://medium.com/@sawant`.You can follow him on Twitter, where he shares learning resources and other useful tech material at `@sawant`.

Patrik Uytterhoeven has over 16 years of experience in IT. Most of this time was spent on HP Unix and Red Hat Linux. In late 2012, he joined Open-Future, a leading open source integrator and the first Zabbix reseller and training partner in Belgium.

When he joined Open-Future, he gained the opportunity to certify himself as a Zabbix Certified trainer. Since then, he has provided training and public demonstrations not only in Belgium but also around the world, in countries such as the Netherlands, Germany, Canada, and Ireland. His next step was to write a book about Zabbix. *Zabbix Cookbook* was born in March 2015 and was published by *Packt Publishing*.

As he also has a deep interest in configuration management, he wrote some Ansible roles for Red Hat 6.x and 7.x to deploy and update Zabbix. These roles, and some others, can be found in the Ansible Galaxy at `https://galaxy.ansible.com/list#/users/1375`.

He is also a technical reviewer of *Learning Ansible* and the upcoming book, *Ansible Configuration Management, Second Edition*, both by Packt Publishing.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and readPackt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at`www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Welcome to Mastering Ansible, your guide to a variety of advanced features and functionality provided by Ansible, which is an automation and orchestration tool. This book will provide you with the knowledge and skills to truly understand how Ansible functions at the fundamental level. This will allow you to master the advanced capabilities required to tackle the complex automation challenges of today and beyond. You will gain knowledge of Ansible workflows, explore use cases for advanced features, troubleshoot unexpected behavior, and extend Ansible through customization.

What this book covers

Chapter 1, *System Architecture and Design of Ansible*, provides a detailed look at the ins and outs of how Ansible goes about performing tasks on behalf of an engineer, how it is designed, and how to work with inventories and variables.

Chapter 2, *Protecting Your Secrets with Ansible*, explores the tools available to encrypt data at rest and prevent secrets from being revealed at runtime.

Chapter 3, *Unlocking the Power of Jinja2 Templates*, states the varied uses of the Jinja2 templating engine within Ansible, and discusses ways to make the most out of its capabilities.

Chapter 4, *Controlling Task Conditions*, describes the changing of default behavior of Ansible to customize task error and change conditions.

Chapter 5, *Composing Reusable Ansible Content with Roles*, describes the approach to move beyond executing loosely organized tasks on hosts to encapsulating clean reusable abstractions to applying the specific functionality of a target set of hosts.

Chapter 6, *Minimizing Downtime with Rolling Deployments*, explores the common deployment and upgrade strategies to showcase relevant Ansible features.

Chapter 7, *Troubleshooting Ansible*, explores the various methods that can be employed to examine, introspect, modify, and debug the operations of Ansible.

Chapter 8, *Extending Ansible*, discovers the various ways in which new capabilities can be added to Ansible via modules, plugins, and inventory sources.

What you need for this book

To follow the examples provided in this book, you will need access to a computer platform capable of running Ansible. Currently, Ansible can be run from any machine with Python 2.6 or 2.7 installed (Windows isn't supported for the control machine). This includes Red Hat, Debian, CentOS, OS X, any of the BSDs, and so on.

This book uses the Ansible 1.9.x series release.

Ansible installation instructions can be found at `http://docs.ansible.com/ansible/intro_installation.html`.

Who this book is for

This book is intended for Ansible developers and operators who have an understanding of the core elements and applications but are now looking to enhance their skills in applying automation using Ansible.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "When `ansible` or `ansible-playbook` is directed at an executable file for an inventory source, Ansible will execute that script with a single argument, `--list`."

A block of code is set as follows:

```
- name: add new node into runtime inventory
  add_host:
    name: newmastery.example.name
    groups: web
    ansible_ssh_host: 192.168.10.30
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "The first is an SSH feature, **ControlPersist**, which provides a mechanism to create persistent sockets when first connecting to a remote host that can be reused in subsequent connections to bypass some of the handshaking required when creating a connection."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
System Architecture and Design of Ansible

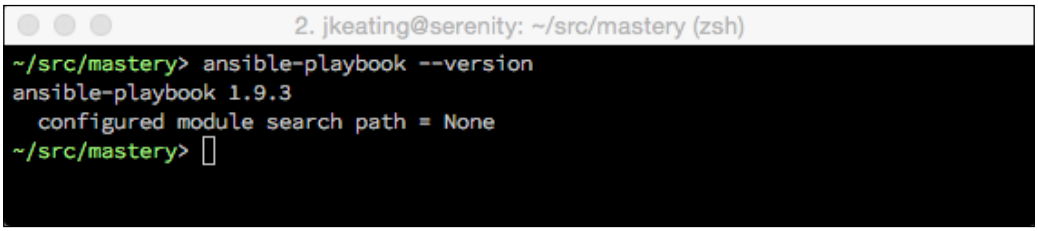
This chapter provides a detailed exploration of the architecture and design of how **Ansible** goes about performing tasks on your behalf. We will cover basic concepts of inventory parsing and how the data is discovered, and then dive into playbook parsing. We will take a walk through module preparation, transportation, and execution. Lastly, we will detail variable types and find out where variables can be located, the scope they can be used for, and how precedence is determined when variables are defined in more than one location. All these things will be covered in order to lay the foundation for mastering Ansible!

In this chapter, we will cover the following topics:

- Ansible version and configuration
- Inventory parsing and data sources
- Playbook parsing
- Module transport and execution
- Variable types and locations
- Variable precedence

Ansible version and configuration

It is assumed that you have Ansible installed on your system. There are many documents out there that cover installing Ansible in a way that is appropriate for the operating system and version that you might be using. This book will assume the use of the Ansible 1.9.x version. To discover the version in use on a system with Ansible already installed, make use of the version argument, that is, either `ansible` or `ansible-playbook`:



Note that `ansible` is the executable for doing ad-hoc one-task executions and `ansible-playbook` is the executable that will process playbooks for orchestrating many tasks.

The configuration for Ansible can exist in a few different locations, where the first file found will be used. The search order changed slightly in version 1.5, with the new order being:

- `ANSIBLE_CFG`: This is an environment variable
- `ansible.cfg`: This is in the current directory
- `ansible.cfg`: This is in the user's home directory
- `/etc/ansible/ansible.cfg`

Some installation methods may include placing a **config** file in one of these locations. Look around to check whether such a file exists and see what settings are in the file to get an idea of how Ansible operation may be affected. This book will assume no settings in the `ansible.cfg` file that would affect the default operation of Ansible.

Inventory parsing and data sources

In Ansible, nothing happens without an inventory. Even ad hoc actions performed on localhost require an inventory, even if that inventory consists just of the localhost. The inventory is the most basic building block of Ansible architecture. When executing `ansible` or `ansible-playbook`, an inventory must be referenced. Inventories are either files or directories that exist on the same system that runs `ansible` or `ansible-playbook`. The location of the inventory can be referenced at runtime with the `-inventory-file` (`-i`) argument, or by defining the path in an Ansible config file.

Inventories can be static or dynamic, or even a combination of both, and Ansible is not limited to a single inventory. The standard practice is to split inventories across logical boundaries, such as *staging* and *production*, allowing an engineer to run a set of plays against their staging environment for validation, and then follow with the same exact plays run against the production inventory set.

Variable data, such as specific details on how to connect to a particular host in your inventory, can be included along with an inventory in a variety of ways as well, and we'll explore the options available to you.

The static inventory

The static inventory is the most basic of all the inventory options. Typically, a static inventory will consist of a single file in the `ini` format. Here is an example of a static inventory file describing a single host, `mastery.example.name`:

```
mastery.example.name
```

That is all there is to it. Simply list the names of the systems in your inventory. Of course, this does not take full advantage of all that an inventory has to offer. If every name were listed like this, all plays would have to reference specific host names, or the special `all` group. This can be quite tedious when developing a playbook that operates across different sets of your infrastructure. At the very least, hosts should be arranged into groups. A design pattern that works well is to arrange your systems into groups based on expected functionality. At first, this may seem difficult if you have an environment where single systems can play many different roles, but that is perfectly fine. Systems in an inventory can exist in more than one group, and groups can even consist of other groups! Additionally, when listing groups and hosts, it's possible to list hosts without a group. These would have to be listed first, before any other group is defined.

Let's build on our previous example and expand our inventory with a few more hosts and some groupings:

```
[web]
mastery.example.name

[dns]
backend.example.name

[database]
backend.example.name

[frontend:children]
web

[backend:children]
dns
database
```

What we have created here is a set of three groups with one system in each, and then two more groups, which logically group all three together. Yes, that's right; you can have groups of groups. The syntax used here is `[groupname:children]`, which indicates to Ansible's inventory parser that this group by the name of `groupname` is nothing more than a grouping of other groups. The children in this case are the names of the other groups. This inventory now allows writing plays against specific hosts, low-level role-specific groups, or high-level logical groupings, or any combination.

By utilizing generic group names, such as `dns` and `database`, Ansible plays can reference these generic groups rather than the explicit hosts within. An engineer can create one inventory file that fills in these groups with hosts from a preproduction staging environment and another inventory file with the production versions of these groupings. The playbook content does not need to change when executing on either staging or production environment because it refers to the generic group names that exist in both inventories. Simply refer to the right inventory to execute it in the desired environment.

Inventory variable data

Inventories provide more than just system names and groupings. Data about the systems can be passed along as well. This can include:

- Host-specific data to use in templates
- Group-specific data to use in task arguments or conditionals
- Behavioral parameters to tune how Ansible interacts with a system

Variables are a powerful construct within Ansible and can be used in a variety of ways, not just the ways described here. Nearly every single thing done in Ansible can include a variable reference. While Ansible can discover data about a system during the setup phase, not all data can be discovered. Defining data with the inventory is how to expand the dataset. Note that variable data can come from many different sources, and one source may override another source. Variable precedence order is covered later in this chapter.

Let's improve upon our existing example inventory and add to it some variable data. We will add some host-specific data as well as group specific data:

```
[web]
mastery.example.name ansible_ssh_host=192.168.10.25

[dns]
backend.example.name

[database]
backend.example.name

[frontend:children]
web

[backend:children]
dns
database

[web:vars]
http_port=88
proxy_timeout=5

[backend:vars]
ansible_ssh_port=314

[all:vars]
ansible_ssh_user=otto
```

In this example, we defined `ansible_ssh_host` for `mastery.example.name` to be the IP address of `192.168.10.25`. An `ansible_ssh_host` is a **behavioral inventory parameter**, which is intended to alter the way Ansible behaves when operating with this host. In this case, the parameter instructs Ansible to connect to the system using the provided IP address rather than performing a DNS lookup on the name `mastery.example.name`. There are a number of other behavioral inventory parameters, which are listed at the end of this section along with their intended use.

Our new inventory data also provides group level variables for the web and backend groups. The web group defines `http_port`, which may be used in an `nginx` configuration file, and `proxy_timeout`, which might be used to determine **HAProxy** behavior. The `backend` group makes use of another behavioral inventory parameter to instruct Ansible to connect to the hosts in this group using port `314` for SSH, rather than the default of `22`.

Finally, a construct is introduced that provides variable data across all the hosts in the inventory by utilizing a built-in *all* group. Variables defined within this group will apply to every host in the inventory. In this particular example, we instruct Ansible to log in as the `otto` user when connecting to the systems. This is also a behavioral change, as the Ansible default behavior is to log in as a user with the same name as the user executing `ansible` or `ansible-playbook` on the control host.

Here is a table of behavior inventory parameters and the behavior they intend to modify:

Inventory parameters	**Behaviour**
`ansible_ssh_host`	This is the name of the host to connect to, if different from the alias you wish to give to it.
`ansible_ssh_port`	This is the SSH port number, if not `22`.
`ansible_ssh_user`	This is the default SSH username to use.
`ansible_ssh_pass`	This is the SSH password to use (this is insecure, we strongly recommend using `--ask-pass` or the SSH keys)
`ansible_sudo_pass`	This is the sudo password to use (this is insecure, we strongly recommend using `--ask-sudo-pass`)
`ansible_sudo_exe`	This is the sudo command path.
`ansible_connection`	This is the connection type of the host. Candidates are local, smart, ssh, or paramiko. The default is paramiko before Ansible 1.2, and smart afterwards, which detects whether the usage of ssh will be feasible based on whether the ssh feature ControlPersist is supported
`ansible_ssh_private_key_file`	This is the private key file used by SSH. This is useful if you use multiple keys and you don't want to use SSH agent

Inventory parameters	Behaviour
`ansible_shell_type`	This is the shell type of the target system. By default, commands are formatted using the `sh-style` syntax. Setting this to `csh` or `fish` will cause commands to be executed on target systems to follow those shell's syntax instead
`ansible_python_interpreter`	This is the target host Python path. This is useful for systems with more than one Python, systems that are not located at `/usr/bin/python` (such as `\*BSD`), or for systems where `/usr/bin/python` is not a 2.X series Python. We do not use the `/usr/bin/env` mechanism as it requires the remote user's path to be set right and also assumes that the Python executable is named Python, where the executable might be named something like `python26`.
`ansible\_\*\_interpreter`	This works for anything such as Ruby or Perl and works just like `ansible_python_interpreter`. This replaces the shebang of modules which run on that host

Dynamic inventories

A static inventory is great and enough for many situations. But there are times when a statically written set of hosts is just too unwieldy to manage. Consider situations where inventory data already exists in a different system, such as **LDAP**, a cloud computing provider, or an in-house **CMDB** (inventory, asset tracking, and data warehousing) system. It would be a waste of time and energy to duplicate that data, and in the modern world of on-demand infrastructure, that data would quickly grow stale or disastrously incorrect.

Another example of when a dynamic inventory source might be desired is when your site grows beyond a single set of playbooks. Multiple playbook repositories can fall into the trap of holding multiple copies of the same inventory data, or complicated processes have to be created to reference a single copy of the data. An external inventory can easily be leveraged to access the common inventory data stored outside of the playbook repository to simplify the setup. Thankfully, Ansible is not limited to static inventory files.

A dynamic inventory source (or plugin) is an executable script that Ansible will call at runtime to discover real-time inventory data. This script may reach out into external data sources and return data, or it can just parse local data that already exists but may not be in the Ansible inventory `ini` format. While it is possible and easy to develop your own dynamic inventory source, which we will cover in a later chapter, Ansible provides a number of example inventory plugins, including but not limited to:

- OpenStack Nova
- Rackspace Public Cloud
- DigitalOcean
- Linode
- Amazon EC2
- Google Compute Engine
- Microsoft Azure
- Docker
- Vagrant

Many of these plugins require some level of configuration, such as user credentials for EC2 or authentication endpoint for OpenStack Nova. Since it is not possible to configure additional arguments for Ansible to pass along to the inventory script, the configuration for the script must either be managed via an `ini` config file read from a known location, or environment variables read from the shell environment used to execute `ansible` or `ansible-playbook`.

When `ansible` or `ansible-playbook` is directed at an executable file for an inventory source, Ansible will execute that script with a single argument, `--list`. This is so that Ansible can get a listing of the entire inventory in order to build up its internal objects to represent the data. Once that data is built up, Ansible will then execute the script with a different argument for every host in the data to discover variable data. The argument used in this execution is `--host <hostname>`, which will return any variable data specific to that host.

In *Chapter 8*, *Extending Ansible*, we will develop our own custom inventory plugin to demonstrate how they operate.

Run-time inventory additions

Just like static inventory files, it is important to remember that Ansible will parse this data once, and only once, per `ansible` or `ansible-playbook` execution. This is a fairly common stumbling point for users of cloud dynamic sources, where frequently a playbook will create a new cloud resource and then attempt to use it as if it were part of the inventory. This will fail, as the resource was not part of the inventory when the playbook launched. All is not lost though! A special module is provided that allows a playbook to temporarily add inventory to the in-memory inventory object, the `add_host` module.

The `add_host` module takes two options, `name` and `groups`. The name should be obvious, it defines the hostname that Ansible will use when connecting to this particular system. The `groups` option is a comma-separated list of groups to add this new system to. Any other option passed to this module will become the host variable data for this host. For example, if we want to add a new system, name it `newmastery.example.name`, add it to the `web` group, and instruct Ansible to connect to it by way of IP address `192.168.10.30`, we will create a task like this:

```
- name: add new node into runtime inventory
  add_host:
    name: newmastery.example.name
    groups: web
    ansible_ssh_host: 192.168.10.30
```

This new host will be available to use, by way of the name provided, or by way of the web group, for the rest of the `ansible-playbook` execution. However, once the execution has completed, this host will not be available unless it has been added to the inventory source itself. Of course, if this were a new cloud resource created, the next `ansible` or `ansible-playbook` execution that sourced inventory from that cloud would pick up the new member.

Inventory limiting

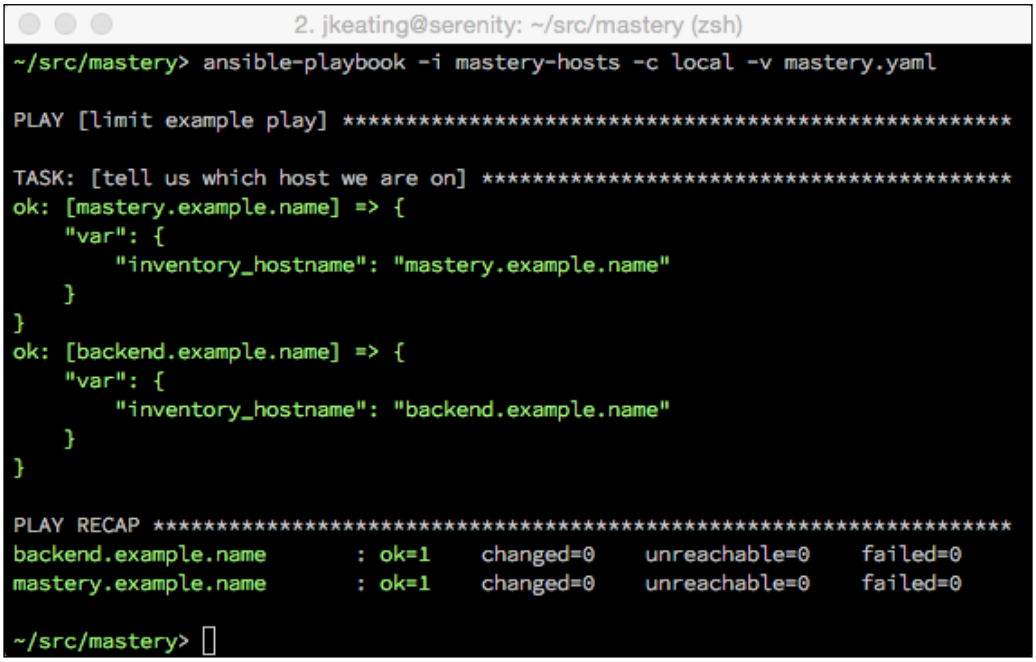
As mentioned earlier, every execution of `ansible` or `ansible-playbook` will parse the entire inventory it has been directed at. This is even true when a limit has been applied. A limit is applied at run time by making use of the `--limit` runtime argument to `ansible` or `ansible-playbook`. This argument accepts a pattern, which is basically a mask to apply to the inventory. The entire inventory is parsed, and at each play the supplied limit mask further limits the host pattern listed for the play.

Let's take our previous inventory example and demonstrate the behavior of Ansible with and without a limit. If you recall, we have the special group `all` that we can use to reference all the hosts within an inventory. Let's assume that our inventory is written out in the current working directory in a file named `mastery-hosts`, and we will construct a playbook to demonstrate the host on which Ansible is operating. Let's write this playbook out as `mastery.yaml`:

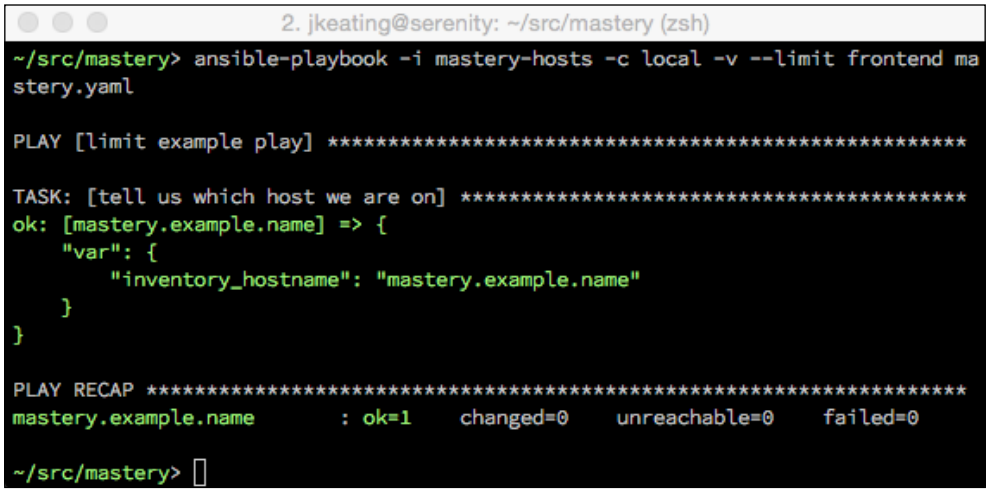
```
---
- name: limit example play
  hosts: all
  gather_facts: false

  tasks:
    - name: tell us which host we are on
      debug:
        var: inventory_hostname
```

The debug module is used to print out text, or values of variables. We'll use this module a lot in this book to simulate actual work being done on a host.

Now, let's execute this simple playbook without supplying a limit. For simplicity's sake, we will instruct Ansible to utilize a local connection method, which will execute locally rather than attempting to SSH to these nonexistent hosts. Let's take a look at the following screenshot:



As we can see, both hosts `backend.example.name` and `mastery.example.name` were operated on. Let's see what happens if we supply a limit, specifically to limit our run to only frontend systems:

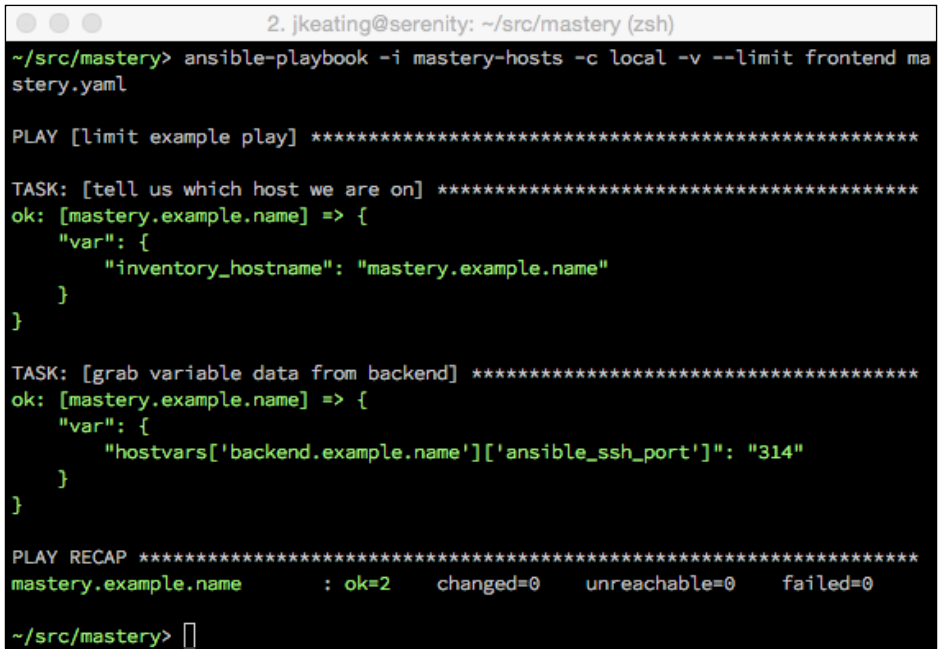


We can see that only `mastery.example.name` was operated on this time. While there are no visual clues that the entire inventory was parsed, if we dive into the Ansible code and examine the inventory object, we will indeed find all the hosts within, and see how the limit is applied every time the object is queried for items.

It is important to remember that regardless of the host's pattern used in a play, or the limit supplied at runtime, Ansible will still parse the entire inventory set during each run. In fact, we can prove this by attempting to access host variable data for a system that would otherwise be masked by our limit. Let's expand our playbook slightly and attempt to access the `ansible_ssh_port` variable from `backend.example.name`:

```
---
- name: limit example play
  hosts: all
  gather_facts: false

  tasks:
    - name: tell us which host we are on
      debug:
        var: inventory_hostname

    - name: grab variable data from backend
      debug:
        var: hostvars['backend.example.name']['ansible_ssh_port']
```

We will still apply our limit, which will restrict our operations to just `mastery.example.name`:



We have successfully accessed the host variable data (by way of group variables) for a system that was otherwise limited out. This is a key skill to understand, as it allows for more advanced scenarios, such as directing a task at a host that is otherwise limited out. Delegation can be used to manipulate a load balancer to put a system into maintenance mode while being upgraded without having to include the load balancer system in your limit mask.

Playbook parsing

The whole purpose of an inventory source is to have systems to manipulate. The manipulation comes from playbooks (or in the case of `ansible` ad hoc execution, simple single task plays). You should already have a base understanding of playbook construction so we won't spend a lot of time covering that, however, we will delve into some specifics of how a playbook is parsed. Specifically, we will cover the following:

- Order of operations
- Relative path assumptions
- Play behavior keys

- Host selection for plays and tasks
- Play and task names

Order of operations

Ansible is designed to be as easy as possible for a human to understand. The developers strive to strike the best balance between human comprehension and machine efficiency. To that end, nearly everything in Ansible can be assumed to be executed in a top to bottom order; that is the operation listed at the top of a file will be accomplished before the operation listed at the bottom of a file. Having said that, there are a few caveats and even a few ways to influence the order of operations.

A playbook has only two main operations it can accomplish. It can either run a play, or it can include another playbook from somewhere on the filesystem. The order in which these are accomplished is simply the order in which they appear in the playbook file, from top to bottom. It is important to note that while the operations are executed in order, the entire playbook, and any included playbooks, is completely parsed before any executions. This means that any included playbook file has to exist at the time of the playbook parsing. They cannot be generated in an earlier operation.

Within a play, there are a few more operations. While a playbook is strictly ordered from top to bottom, a play has a more nuanced order of operations. Here is a list of the possible operations and the order in which they will happen:

- Variable loading
- Fact gathering
- The `pre_tasks` execution
- Handlers notified from the `pre_tasks` execution
- Roles execution
- Tasks execution
- Handlers notified from roles or tasks execution
- The `post_tasks` execution
- Handlers notified from `post_tasks` execution

Here is an example play with most of these operations shown:

```
---
- hosts: localhost
  gather_facts: false

  vars:
```

```
    - a_var: derp

  pre_tasks:
    - name: pretask
      debug: msg="a pre task"
      changed_when: true
      notify: say hi

  roles:
    - role: simple
      derp: newval

  tasks:
    - name: task
      debug: msg="a task"
      changed_when: true
      notify: say hi

  post_tasks:
    - name: posttask
      debug: msg="a post task"
      changed_when: true
      notify: say hi
```

Regardless of the order in which these blocks are listed in a play, this is the order in which they will be processed. Handlers (the tasks that can be triggered by other tasks that result in a change) are a special case. There is a utility module, `meta`, which can be used to trigger handler processing at that point:

```
- meta: flush_handlers
```

This will instruct Ansible to process any pending handlers at that point before continuing on with the next task or next block of actions within a play. Understanding the order and being able to influence the order with `flush_handlers` is another key skill to have when there is a need to orchestrate complicated actions, where things such as service restarts are very sensitive to order. Consider the initial rollout of a service. The play will have tasks that modify `config` files and indicate that the service should be restarted when these files change. The play will also indicate that the service should be running. The first time this play happens, the config file will change and the service will change from not running to running. Then, the handlers will trigger, which will cause the service to restart immediately. This can be disruptive to any consumers of the service. It would be better to flush the handlers before a final task to ensure the service is running. This way, the restart will happen before the initial start, and thus the service will start up once and stay up.

Relative path assumptions

When Ansible parses a playbook, there are certain assumptions that can be made about the relative paths of items referenced by the statements in a playbook. In most cases, paths for things such as variable files to include, task files to include, playbook files to include, files to copy, templates to render, scripts to execute, and so on, are all relative to the directory where the file referencing them lives. Let's explore this with an example playbook and directory listing to show where the things are.

- Directory structure:

```
.
├── a_vars_file.yaml
├── mastery-hosts
├── relative.yaml
└── tasks
    ├── a.yaml
    └── b.yaml
```

- Contents of `_vars_file.yaml`:

```
---
something: "better than nothing"
```

- Contents of `relative.yaml`:

```
---
- name: relative path play
  hosts: localhost
  gather_facts: false

  vars_files:
    - a_vars_file.yaml

  tasks:
    - name: who am I
      debug:
        msg: "I am mastery task"

    - name: var from file
      debug: var=something

    - include: tasks/a.yaml
```

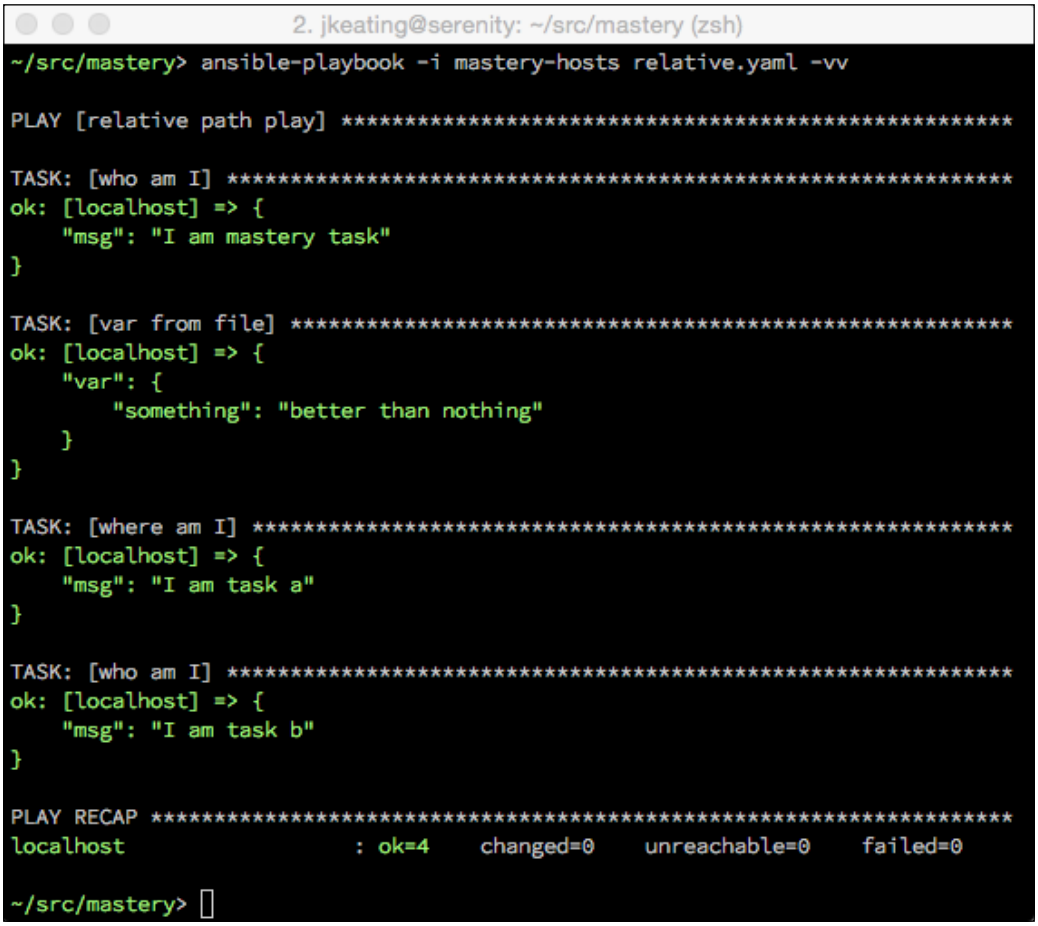
- Contents of `tasks/a.yaml`:

```
---
- name: where am I
  debug:
    msg: "I am task a"

- include: b.yaml
```

- Contents of tasks/b.yaml:

```
---
- name: who am I
  debug:
    msg: "I am task b"
```

Here the execution of the playbook is shown as follows:



We can clearly see the relative reference to paths and how they are relative to the file referencing them. When using roles there are some additional relative path assumptions, however we'll cover that in detail in a later chapter.

Play behavior keys

When Ansible parses a play, there are a few keys it looks for to define various behaviors for a play. These keys are written at the same level as `hosts: key`. Here are the keys that can be used:

- **any_errors_fatal**: This Boolean key is used to instruct Ansible to treat any failure as a fatal error to prevent any further tasks from being attempted. This changes the default where Ansible will continue until all the tasks are complete or all the hosts have failed.
- **connection**: This string key defines which connection system to use for a given play. A common choice to make here is local, which instructs Ansible to do all the operations locally, but with the context of the system from the inventory.
- **gather_facts**: This Boolean key controls whether or not Ansible will perform the fact gathering phase of operation, where a special task will run on a host to discover various facts about the system. Skipping fact gathering, when you are sure that you do not need any of the discovered data, can be a significant time saver in a larger environment.
- **max_fail_percentage**: This number key is similar to `any_errors_fatal`, but is more fine-grained. This allows you to define just what percentage of your hosts can fail before the whole operation is halted.
- **no_log**: This is a Boolean key to control whether or not Ansible will log (to the screen and/or a configured log file) the command given or the results received from a task. This is important if your task or return deal with secrets. This key can also be applied to a task directly.
- **port**: This is a number key to define what port SSH (or an other remote connection plugin) should use to connect unless otherwise configured in the inventory data.
- **remote_user**: This is a string key that defines which user to log in with on the remote system. The default is to connect as the same user that `ansible-playbook` was started with.

- **serial**: This key takes a number and controls how many systems Ansible will execute a task on before moving to the next task in a play. This is a drastic change from the normal order of operation, where a task is executed across every system in a play before moving to the next. This is very useful in rolling update scenarios, which will be detailed in later chapters.
- **sudo**: This is a Boolean key used to configure whether sudo should be used on the remote host to execute tasks. This key can also be defined at a task level. A second key, `sudo_user`, can be used to configure which user to sudo to (instead of root).
- **su**: Much like sudo, this key is used to su instead of sudo. This key also has a companion, `su_user`, to configure which user to su to (instead of root).

Many of these keys will be used in example playbooks through this book.

Host selection for plays and tasks

The first thing most plays define (after a name, of course) is a host pattern for the play. This is the pattern used to select hosts out of the inventory object to run the tasks on. Generally this is straightforward; a host pattern contains one or more blocks indicating a host, group, wildcard pattern, or regex to use for the selection. Blocks are separated by a colon, wildcards are just an asterisk, and regex patterns start with a tilde:

```
hostname:groupname:*.example:~(web|db)\.example\.com
```

Advanced usage can include group index selection or even ranges within a group:

```
Webservers[0]:webservers[2:4]
```

Each block is treated as an inclusion block, that is, all the hosts found in the first pattern are added to all the hosts found in the next pattern, and so on. However, this can be manipulated with control characters to change their behavior. The use of an ampersand allows an inclusion selection (all the hosts that exist in both patterns). The use of an exclamation point allows exclusion selection (all the hosts that exist in the previous patterns that are NOT in the exclusion pattern):

```
Webservers:&dbservers
Webservers:!dbservers
```

Once Ansible parses the patterns, it will then apply restrictions, if any. Restrictions come in the form of limits or failed hosts. This result is stored for the duration of the play, and it is accessible via the `play_hosts` variable. As each task is executed, this data is consulted and an additional restriction may be placed upon it to handle serial operations. As failures are encountered, either failure to connect or a failure in execute tasks, the failed host is placed in a restriction list so that the host will be bypassed in the next task. If, at any time, a host selection routine gets restricted down to zero hosts, the play execution will stop with an error. A caveat here is that if the play is configured to have a `max_fail_precentage` or `any_errors_fatal` parameter, then the playbook execution stops immediately after the task where this condition is met.

Play and task names

While not strictly necessary, it is a good practice to label your plays and tasks with names. These names will show up in the command line output of `ansible-playbook`, and will show up in the log file if `ansible-playbook` is directed to log to a file. Task names also come in handy to direct `ansible-playbook` to start at a specific task and to reference handlers.

There are two main points to consider when naming plays and tasks:

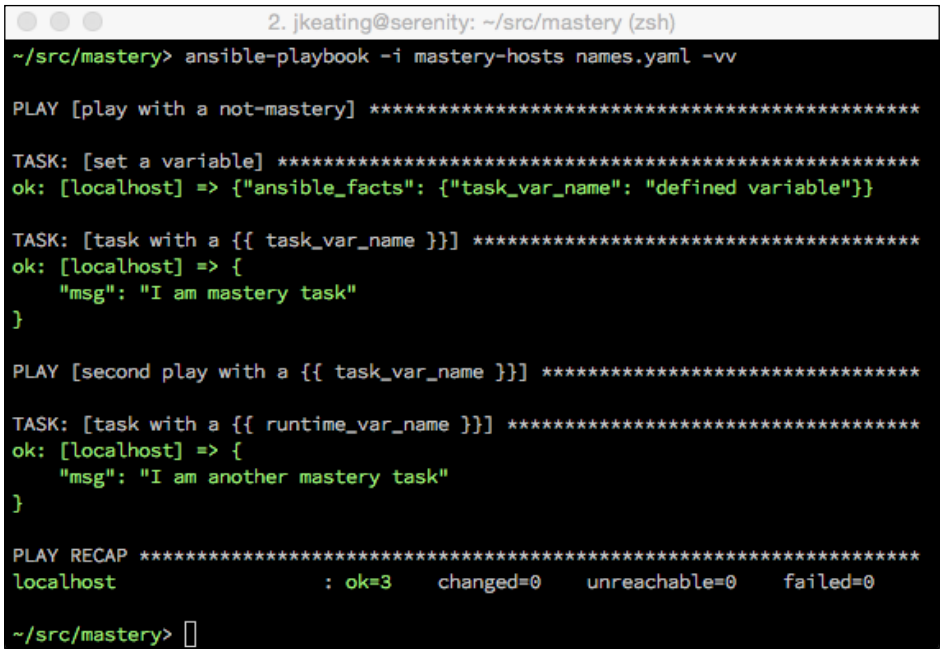
- Names of plays and tasks should be unique
- Beware of what kind of variables can be used in play and task names

Naming plays and tasks uniquely is a best practice in general that will help to quickly identify where a problematic task may reside in your hierarchy of playbooks, roles, task files, handlers, and so on. Uniqueness is more important when notifying a handler or when starting at a specific task. When task names have duplicates, the behavior of Ansible may be nondeterministic or at least not obvious.

With uniqueness as a goal, many playbook authors will look to variables to satisfy this constraint. This strategy may work well but authors need to take care as to the source of the variable data they are referencing. Variable data can come from a variety of locations (which we will cover later in this chapter), and the values assigned to variables can be defined at a variety of times. For the sake of play and task names, it is important to remember that only variables for which the values can be determined at playbook parse time will parse and render correctly. If the data of a referenced variable is discovered via a task or other operation, the variable string will be displayed unparsed in the output. Let's look at an example playbook that utilizes variables for play and task names:

```
---
- name: play with a {{ var_name }}
  hosts: localhost
  gather_facts: false

  vars:
    - var_name: not-mastery

  tasks:
    - name: set a variable
      set_fact:
        task_var_name: "defined variable"

    - name: task with a {{ task_var_name }}
      debug:
        msg: "I am mastery task"

- name: second play with a {{ task_var_name }}
  hosts: localhost
  gather_facts: false

  tasks:
    - name: task with a {{ runtime_var_name }}
      debug:
        msg: "I am another mastery task"
```

At first glance, one might expect at least `var_name` and `task_var_name` to render correctly. We can clearly see `task_var_name` being defined before its use. However, armed with our knowledge that playbooks are parsed in their entirety before execution, we know better:



As we can see, the only variable name that is properly rendered is `var_name`, as it was defined as a static play variable.

Module transport and execution

Once a playbook is parsed and the hosts are determined, Ansible is ready to execute a task. Tasks are made up of a name (optional, but please don't skip it), a module reference, module arguments, and task control keywords. A later chapter will cover task control keywords in detail, so we will only concern ourselves with the module reference and arguments.

Module reference

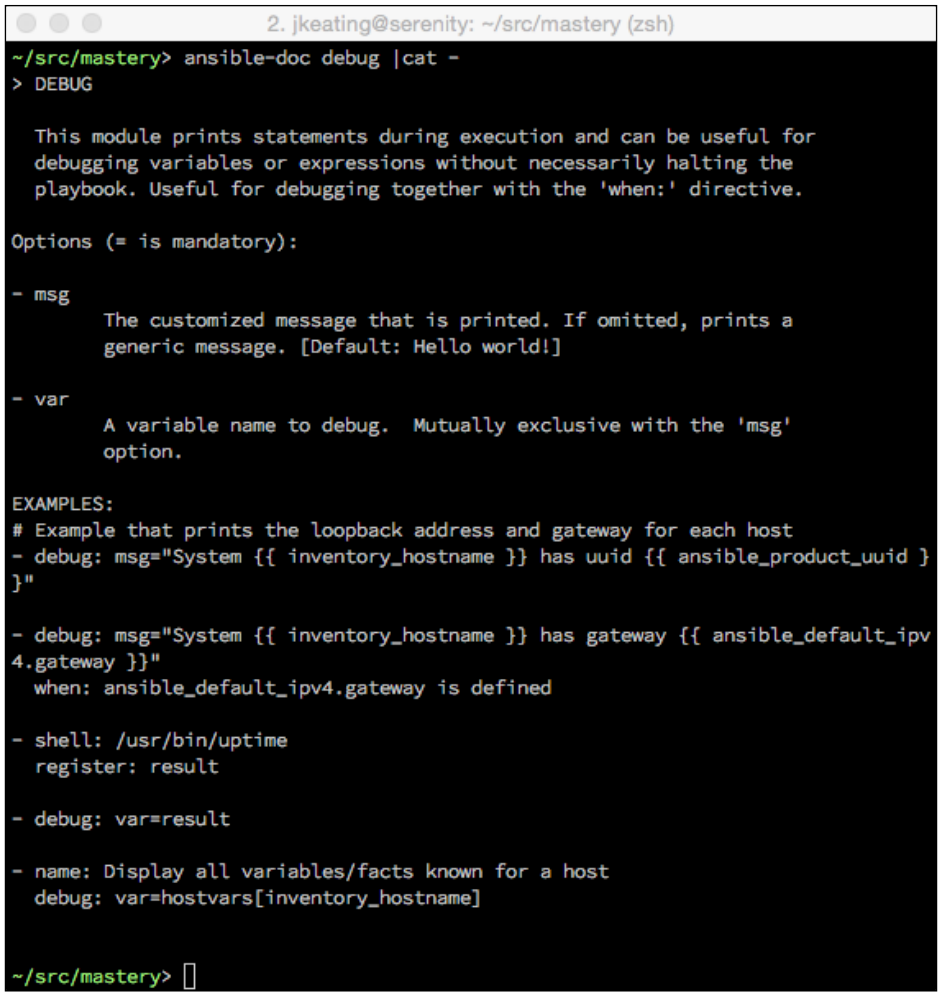
Every task has a module reference. This tells Ansible which bit of work to do. Ansible is designed to easily allow for custom modules to live alongside a playbook. These custom modules can be a wholly new functionality, or they can replace modules shipped with Ansible itself. When Ansible parses a task and discovers the name of the module to use for a task, it looks into a series of locations in order to find the module requested. Where it looks also depends on where the task lives, whether in a role or not.

If a task is in a role, Ansible will first look for the module within a directory tree named `library` within the role the task resides in. If the module is not found there, Ansible looks for a directory named `library` at the same level as the main playbook (the one referenced by the `ansible-playbook` execution). If the module is not found there, Ansible will finally look in the configured library path, which defaults to `/usr/share/ansible/`. This library path can be configured in an Ansible config file, or by way of the `ANSIBLE_LIBRARY` environment variable.

This design, allowing modules to be bundled with roles and playbooks, allows for adding functionality, or quickly repairing problems very easily.

Module arguments

Arguments to a module are not always required; the help output of a module will indicate which models are required and which are not. Module documentation can be accessed with the `ansible-doc` command:



This command was piped into `cat` to prevent shell paging from being used.

Arguments can be templated with **Jinja2**, which will be parsed at module execution time, allowing for data discovered in a previous task to be used in later tasks; this is a very powerful design element.

Arguments can be supplied in a `key = value` format, or in a complex format that is more native to YAML. Here are two examples of arguments being passed to a module showcasing the two formats:

```
- name: add a keypair to nova
  nova_keypair: login_password={{ pass }} login_tenant_name=admin
                name=admin-key

- name: add a keypair to nova
  nova_keypair: login_password: "{{ pass }}" login_tenant_name: admin
                name: admin-key
```

Both formats will lead to the same result in this example; however, the complex format is required if you wish to pass complex arguments into a module. Some modules expect a list object or a hash of data to be passed in; the complex format allows for this. While both formats are acceptable for many tasks, the complex format is the format used for the majority of examples in this book.

Module transport and execution

Once a module is found, Ansible has to execute it in some way. How the module is transported and executed depends on a few factors, however the common process is to locate the module file on the local filesystem and read it into memory, and then add in the arguments passed to the module. Finally, the boilerplate module code from core Ansible is added to complete the file object in memory. What happens next really depends on the connection method and runtime options (such as leaving the module code on the remote system for review).

The default connection method is `smart`, which most often resolves to the `ssh` connection method. With a default configuration, Ansible will open an SSH connection to the remote host, create a temporary directory, and close the connection. Ansible will then open another SSH connection in order to write out the task object from memory (the result of local module file, task module arguments, and Ansible boilerplate code) into a file within the temporary directory that we just created and close the connection.

Finally, Ansible will open a third connection in order to execute the module and delete the temporary directory and all its contents. The module results are captured from `stdout` in the JSON format, which Ansible will parse and handle appropriately. If a task has an `async` control, Ansible will close the third connection before the module is complete, and SSH back in to the host to check the status of the task after a prescribed period until the module is complete or a prescribed timeout has been reached.

Task performance

Doing the math from the above description, that's at least three SSH connections per task, per host. In a small fleet with a small number of tasks, this may not be a concern; however, as the task set grows and the fleet size grows, the time required to create and tear down SSH connections increases. Thankfully, there are a couple ways to mitigate this.

The first is an SSH feature, **ControlPersist**, which provides a mechanism to create persistent sockets when first connecting to a remote host that can be reused in subsequent connections to bypass some of the handshaking required when creating a connection. This can drastically reduce the amount of time Ansible spends on opening new connections. Ansible automatically utilizes this feature if the host platform where Ansible is run from supports it. To check whether your platform supports this feature, check the SSH main page for `ControlPersist`.

The second performance enhancement that can be utilized is an Ansible feature called pipelining. Pipelining is available to SSH-based connection methods and is configured in the Ansible configuration file within the `ssh_connection` section:

```
[ssh_connection]
pipelining=true
```

This setting changes how modules are transported. Instead of opening an SSH connection to create a directory, another to write out the composed module, and a third to execute and clean up, Ansible will instead open an SSH connection and start the Python interpreter on the remote host. Then, over that live connection, Ansible will pipe in the composed module code for execution. This reduces the connections from three to one, which can really add up. By default, pipelining is disabled.

Utilizing the combination of these two performance tweaks can keep your playbooks nice and fast even as you scale your fleet. However, keep in mind that Ansible will only address as many hosts at once as the number of forks Ansible is configured to run. Forks are the number of processes Ansible will split off as a worker to communicate with remote hosts. The default is five forks, which will address up to five hosts at once. Raise this number to address more hosts as your fleet grows by adjusting the `forks=` parameter in an Ansible configuration file, or by using the `--forks` (`-f`) argument with `ansible` or `ansible-playbook`.

Variable types and location

Variables are a key component of the Ansible design. Variables allow for dynamic play content and reusable plays across different sets of inventory. Anything beyond the very basics of Ansible use will utilize variables. Understanding the different variable types and where they can be located, as well as learning how to access external data or prompt users to populate variable data, is the key to mastering Ansible.

Variable types

Before diving into the precedence of variables, we must first understand the various types and subtypes of variables available to Ansible, their location, and where they are valid for use.

The first major variable type is **inventory variables**. These are the variables that Ansible gets by way of the inventory. These can be defined as variables that are specific to `host_vars` to individual hosts or applicable to entire groups as `group_vars`. These variables can be written directly into the inventory file, delivered by the dynamic inventory plugin, or loaded from the `host_vars/<host>` or `group_vars/<group>` directories.

These types of variables might be used to define Ansible behavior when dealing with these hosts, or site-specific data related to the applications that these hosts run. Whether a variable comes from `host_vars` or `group_vars`, it will be assigned to a host's `hostvars`, and it can be accessed from the playbooks and template files. Accessing a host's own variables can be done just by referencing the name, such as `{{ foobar }}`, and accessing another host's variables can be accomplished by accessing `hostvars`. For example, to access the `foobar` variable for `examplehost`: `{{ hostvars['examplehost']['foobar'] }}`. These variables have global scope.

The second major variable type is **role variables**. These are variables specific to a role that are utilized by the role tasks and have scope only within the role that they are defined in, which is to say that they can only be used within the role. These variables are often supplied as a **role default**, and are meant to provide a default value for the variable, but can easily be overridden when applying the role. When roles are referenced, it is possible to supply variable data at the same time, either by overriding role defaults or creating wholly new data. We'll cover roles in-depth in later chapters. These variables apply to all hosts within the role and can be accessed directly, much like a host's own `hostvars`.

The third major variable type is **play variables**. These variables are defined in the control keys of a play, either directly by the `vars` key or sourced from external files via the `vars_files` key. Additionally, the play can interactively prompt the user for variable data using `vars_prompt`. These variables are to be used within the scope of the play and in any tasks or included tasks of the play. The variables apply to all hosts within the play and can be referenced as if they are `hostvars`.

The fourth variable type is **task variables**. Task variables are made from data discovered while executing tasks or in the fact gathering phase of a play. These variables are host-specific and are added to the host's `hostvars` and can be used as such, which also means they have global scope *after* the point at which they were discovered or defined. Variables of this type can be discovered via `gather_facts` and **fact modules** (modules that do not alter state but rather return data), populated from task return data via the `register` task key, or defined directly by a task making use of the `set_fact` or `add_host` modules. Data can also be interactively obtained from the operator using the `prompt` argument to the `pause` module and registering the result:

```
- name: get the operators name
  pause:
    prompt: "Please enter your name"
  register: opname
```

There is one last variable type, the **extra variables**, or `extra-vars` type. These are variables supplied on the command line when executing `ansible-playbook` via `--extra-vars`. Variable data can be supplied as a list of `key=value` pairs, a quoted JSON data, or a reference to a YAML-formatted file with variable data defined within:

```
--extra-vars "foo=bar owner=fred"
--extra-vars '{"services":["nova-api","nova-conductor"]}'
--extra-vars @/path/to/data.yaml
```

Extra variables are considered global variables. They apply to every host and have scope throughout the entire playbook.

Accessing external data

Data for role variables, play variables, and task variables can also come from external sources. Ansible provides a mechanism to access and evaluate data from the **control machine** (the machine running `ansible-playbook`). The mechanism is called a **lookup plugin**, and a number of them come with Ansible. These plugins can be used to lookup or access data by reading files, generate and locally store passwords on the Ansible host for later reuse, evaluate environment variables, pipe data in from executables, access data in the `Redis` or `etcd` systems, render data from template files, query `dnstxt` records, and more. The syntax is as follows:

```
lookup('<plugin_name>', 'plugin_argument')
```

for example, to use the `mastery` value from `etcd` in a debug task:

```
- name: show data from etcd
  debug: msg="{{ lookup('etcd', 'mastery') }}"
```

Lookups are evaluated when the task referencing them is executed, which allows for dynamic data discovery. To reuse a particular lookup in multiple tasks and reevaluate it each time, a playbook variable can be defined with a lookup value. Each time the playbook variable is referenced the lookup will be executed, potentially providing different values over time.

Variable precedence

As you learned in the previous section, there are a few major types of variables that can be defined in a myriad of locations. This leads to a very important question, what happens when the same variable name is used in multiple locations? Ansible has a precedence for loading variable data, and thus it has an order and a definition to decide which variable will "win". Variable value overriding is an advanced usage of Ansible, so it is important to fully understand the semantics before attempting such a scenario.

Precedence order

Ansible defines the precedence order as follows:

1. Extra vars (from command line) always win
2. Connection variables defined in inventory
3. Most everything else

4. Rest of the variables defined in inventory
5. Facts discovered about a system
6. Role defaults

This list is a useful starting point, however things are a bit more nuanced, as we will explore.

Extra-vars

Extra-vars, as supplied on the command line, certainly overrides anything else. Regardless of where else a variable might be defined, even if it's explicitly set in a play with `set_fact`, the value provided on the command line will be the value used.

Connection variables

Next up are **connection variables**, the behavioral variables outlined earlier. These are variables that influence how Ansible will connect to and execute tasks on a system. These are variables like `ansible_ssh_user`, `ansible_ssh_host`, and others as described in the earlier section regarding behavioral inventory parameters. The Ansible documentation states that these come from the inventory, however, they *can* be overridden by tasks such as `set_fact`. A `set_fact` module on a variable such as `ansible_ssh_user` will override the value that came from the inventory source. There is a precedence order within the inventory as well. Host-specific definitions will override group definitions, and child group definitions will override parent of group definitions. This allows for having a value that applies to most things in a group and overrides it on specific hosts that would be different. When a host belongs to multiple groups and each group defines the same variable with different values, the behavior is less defined and strongly discouraged.

Most everything else

The "most everything else" block is a big grouping of sources. These include:

- Command line switches
- Play variables
- Task variables
- Role variables (not defaults)

These sets of variables can override each other as well, with the rule being that the last supplied variable wins. The role variables in this set refer to the variables provided in a role's `vars/main.yaml` file and the variables defined when assigning a role or a role dependency. In this example, we will provide a variable named `role_var` at the time we assign the role:

```
- role: example_role
  role_var: var_value_here
```

An important nuance here is that a definition provided at role assignment time will override the definition within a role's `vars/main.yaml` file. Also remember the last provided rule; if within the role `example_role`, the `role_var` variable is redefined via a task, that definition will win from that point on.

The rest of the inventory variables

The next lower set of variables is the remaining inventory variables. These are variables that can be defined within the inventory data, but do not alter the behavior of Ansible. The rules from connection variables apply here.

Facts discovered about a system

Discovered facts variables are the variables we get when gathering facts. The exact list of variables depends on the platform of the host and the extra software that can be executed to display system information, which might be installed on said host. Outside of role defaults, these are the lowest level of variables and are most likely to be overridden.

Role defaults

Roles can have default variables defined within them. These are reasonable defaults for use within the role and are customization targets for role applications. This makes roles much more reusable, flexible, and tuneable to the environment and conditions in which the role will be applied.

Merging hashes

In the previous section, we focused on the order of precedence in which variables will override each other. The default behavior of Ansible is that any overriding definition for a variable name will completely mask the previous definition of that variable. However, that behavior can be altered for one type of variable, the hash. A hash variable (a "dictionary" in Python terms) is a dataset of keys and values. Values can be of different types for each key, and can even be hashes themselves for complex data structures.

In some advanced scenarios, it is desirable to replace just one bit of a hash or add to an existing hash rather than replacing the hash altogether. To unlock this ability, a configuration change is necessary in an Ansible config file. The config entry is `hash_behavior`, which takes one of **replace**, or **merge**. A setting of merge will instruct Ansible to merge or blend the values of two hashes when presented with an override scenario rather than the default of replace, which will completely replace the old variable data with the new data.

Let's walk through an example of the two behaviors. We will start with a hash loaded with data and simulate a scenario where a different value for the hash is provided as a higher priority variable.

Starting data:

```
hash_var:
  fred:
    home: Seattle
    transport: Bicycle
```

New data loaded via include_vars:

```
hash_var:
  fred:
    transport: Bus
```

With the default behavior, the new value for `hash_var` will be:

```
hash_var:
  fred:
    transport: Bus
```

However, if we enable the merge behavior we would get the following result:

```
hash_var:
  fred:
    home: Seattle
    transport: Bus
```

There are even more nuances and undefined behaviors when using merge, and as such, it is strongly recommended to only use this setting if absolutely needed.

Summary

While the design of Ansible focuses on simplicity and ease of use, the architecture itself is very powerful. In this chapter, we covered key design and architecture concepts of Ansible, such as version and configuration, playbook parsing, module transport and execution, variable types and locations, and variable precedence.

You learned that playbooks contain variables and tasks. Tasks link bits of code called modules with arguments, which can be populated by variable data. These combinations are transported to selected hosts from provided inventory sources. A fundamental understanding of these building blocks is the platform on which you can build a mastery of all things Ansible!

In the next chapter, you will learn how to secure secret data while operating Ansible.

3
Unlocking the Power of Jinja2 Templates

Templating is the lifeblood of Ansible. From configuration file content to variable substitution in tasks, to conditional statements and beyond, templating comes into play with nearly every Ansible facet. The templating engine of Ansible is Jinja2, a modern and designer-friendly templating language for Python. This chapter will cover a few advanced features of Jinja2 templating.

- Control structures
- Data manipulation
- Comparisons

Control structures

In Jinja2, a control structure refers to things in a template that control the flow of the engine parsing the template. These structures include, but are not limited to, conditionals, loops, and macros. Within Jinja2 (assuming the defaults are in use), a control structure will appear inside blocks of `{% … %}`. These opening and closing blocks alert the Jinja2 parser that a control statement is provided instead of a normal string or variable name.

Conditionals

A conditional within a template creates a decision path. The engine will consider the conditional and choose from two or more potential blocks of code. There is always a minimum of two: a path if the conditional is met (evaluated as true), and an implied `else` path of an empty block.

The statement for conditionals is the `if` statement. This statement works much like it does in Python. An `if` statement can be combined with one or more optional `elif` with an optional final `else`, and unlike Python, requires an explicit `endif`. The following example shows a config file template snippet combining both a regular variable replacement and an `if else` structure:

```
setting = {{ setting }}
{% if feature.enabled %}
feature = True
{% else %}
feature = False
{% endif %}
another_setting = {{ another_setting }}
```

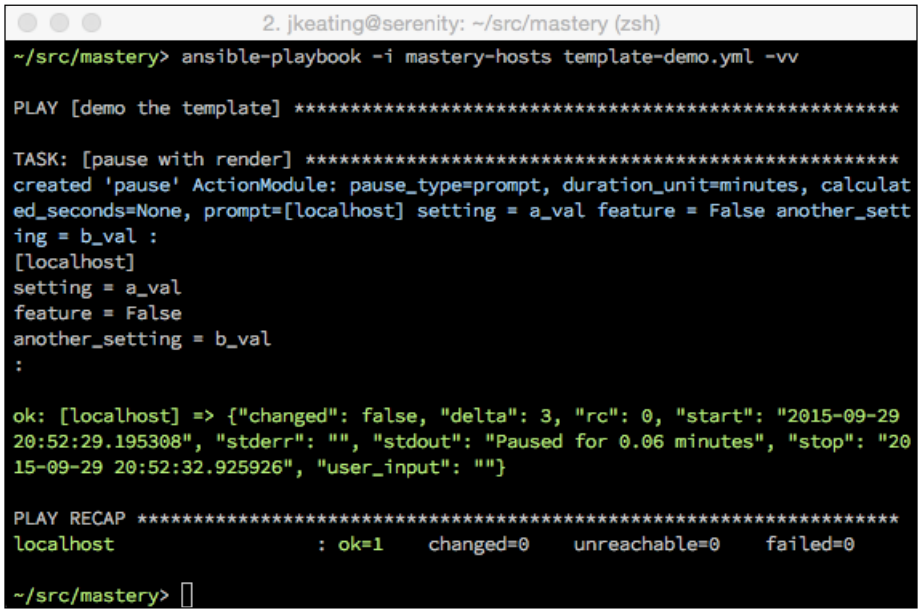
In this example, the variable `feature.enabled` is checked to see if it exists and is not set to `False`. If this is true, then the text `feature = True` is used; otherwise, the text `feature = False` is used. Outside of this control block, the parser does the normal variable substitution for the variables inside the mustache brackets. Multiple paths can be defined by using an `elif` statement, which presents the parser, with another test to perform should the previous tests equate to `false`.

To demonstrate rendering the template, we'll save the example template as `demo.j2`. We'll then make a playbook named `template-demo.yaml` that defines the variables in use and then uses a template lookup as part of a `pause` task to display the rendered template on screen:

```
---
- name: demo the template
  hosts: localhost
  gather_facts: false
  vars:
    setting: a_val
    feature:
      enabled: true
    another_setting: b_val
  tasks:
    - name: pause with render
      pause:
        prompt: "{{ lookup('template', 'demo.j2') }}"
```

Executing this playbook will show the rendered template on screen while it waits for input. We can simply press Enter to complete the playbook:

If we were to change the value of `feature.enabled` to `false`, the output would be slightly different:



Inline conditionals

The `if` statements can be used inside of **inline expressions**. This can be useful in some scenarios where additional newlines are not desired. Let's construct a scenario where we need to define an API as either `cinder` or `cinderv2`:

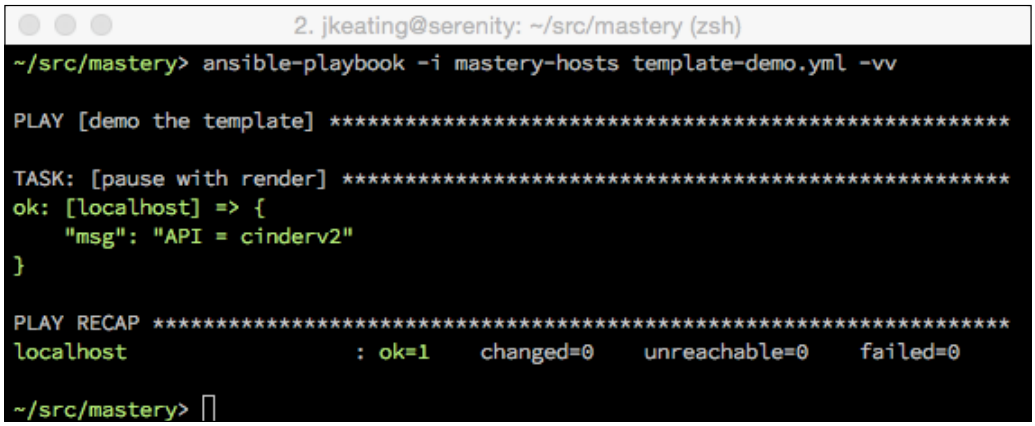
```
API = cinder{{ 'v2' if api.v2 else '' }}
```

This example assumes `api.v2` is defined as Boolean `True` or `False`. Inline `if` expressions follow the syntax of `<do something> if <conditional is true> else <do something else>`. In an inline `if` expression, there is an implied `else`; however, that implied `else` is meant to evaluate as an undefined object, which will normally create an error. We protect against this by defining an explicit `else,` which renders a zero length string.

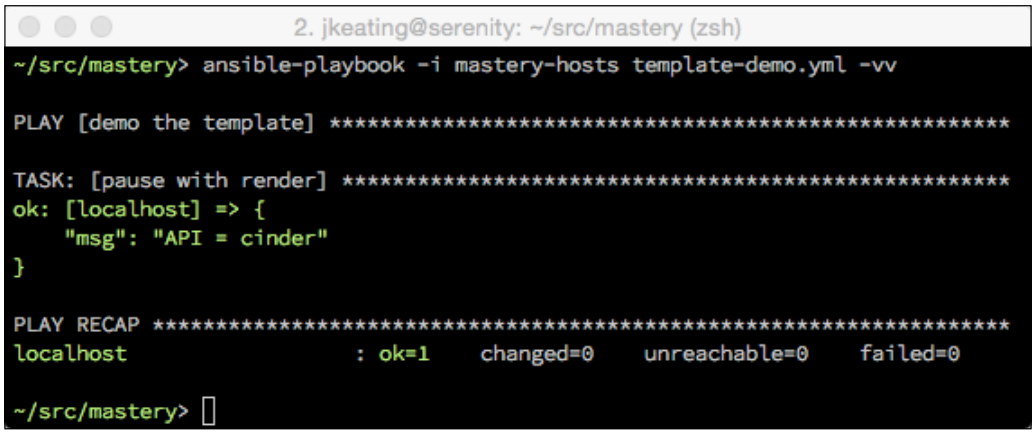
Let's modify our playbook to demonstrate an inline conditional. This time, we'll use the `debug` module to render the simple template:

```
---
- name: demo the template
  hosts: localhost
  gather_facts: false
  vars:
    api:
      v2: true
  tasks:
    - name: pause with render
      debug:
        msg: "API = cinder{{ 'v2' if api.v2 else '' }}"
```

Execution of the playbook will show the template being rendered:



Changing the value of `api.v2` to `false` leads to a different result:



Loops

A loop allows you to create dynamically created sections in template files, and is useful when you know you need to operate on an unknown number of items in the same way. To start a loop control structure, the `for` statement is used. Let's look at a simple way to loop over a list of directories in which a fictional service might find data:

```
# data dirs
{% for dir in data_dirs %}
data_dir = {{ dir }}
{% endfor %}
```

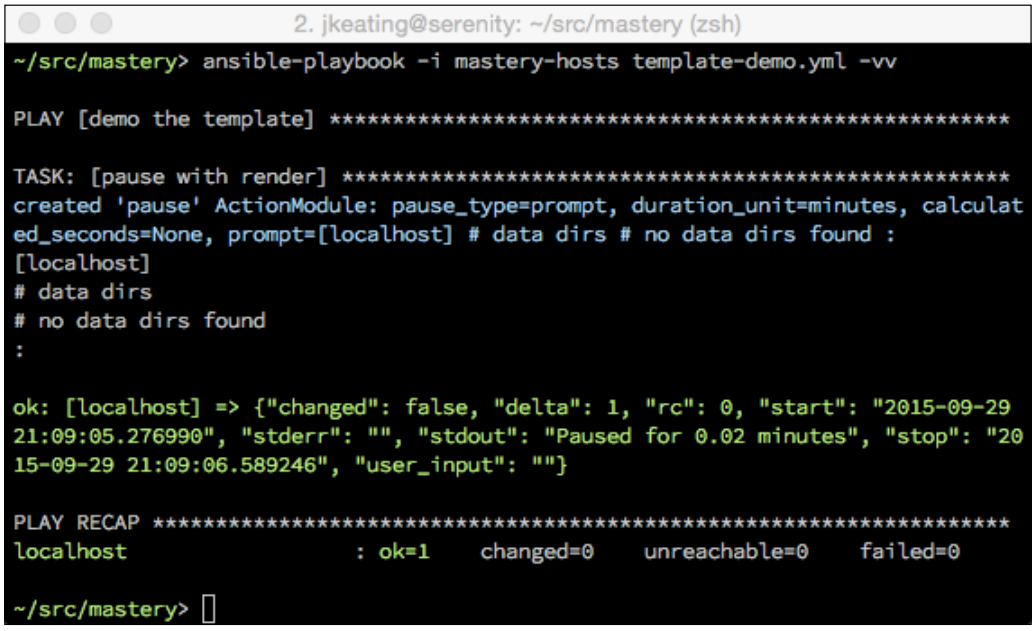
In this example, we will get one `data_dir =` line per item within the `data_dirs` variable, assuming `data_dirs` is a list with at least one item in it. If the variable is not a list (or other iterable type) or is not defined, an error will be generated. If the variable is an iterable type but is empty, then no lines will be generated. Jinja2 allows for the reacting to this scenario and also allows substituting in a line when no items are found in the variable via an `else` statement. In this next example, assume that `data_dirs` is an empty list:

```
# data dirs
{% for dir in data_dirs %}
data_dir = {{ dir }}
{% else %}
# no data dirs found
{% endfor %}
```

We can test this by modifying our playbook and template file again. We'll update `demo.j2` with the above template content and make use of a `prompt` in our playbook again:

```
---
- name: demo the template
  hosts: localhost
  gather_facts: false
  vars:
    data_dirs: []
  tasks:
    - name: pause with render
      pause:
        prompt: "{{ lookup('template', 'demo.j2') }}"
```

Running our playbook will show the following result:



Filtering loop items

Loops can be combined with conditionals as well. Within the loop structure, an `if` statement can be used to check a condition using the current loop item as part of the conditional. Let's extend our example and protect against using (`/`) as a `data_dir`:

```
# data dirs
{% for dir in data_dirs %}
```

```
{% if dir != "/" %}
data_dir = {{ dir }}
{% endif %}
{% else %}
# no data dirs found
{% endfor %}
```

The preceding example successfully filters out any `data_dirs` item that is (`/`) but takes more typing than should be necessary. Jinja2 provides a convenience that allows you to filter loop items easily as part of the `for` statement. Let's repeat the previous example using this convenience:

```
# data dirs
{% for dir in data_dirs if dir != "/" %}
data_dir = {{ dir }}
{% else %}
# no data dirs found
{% endfor %}
```

Not only does this structure require less typing, but it also correctly counts the loops, which we'll learn about in the next section.

Loop indexing

Loop counting is provided *for free*, yielding an index of the current iteration of the loop. As variables, these can be accessed in a few different ways. The following table outlines the ways they can be referenced:

Variable	Description
`loop.index`	The current iteration of the loop (`1` indexed)
`loop.index0`	The current iteration of the loop (`0` indexed)
`loop.revindex`	The number of iterations until the end of the loop (`1` indexed)
`loop.revindex0`	The number of iterations until the end of the loop (`0` indexed)
`loop.first`	Boolean `True` if the first iteration
`loop.last`	Boolean `True` if the last iteration
`loop.length`	The number of items in the sequence

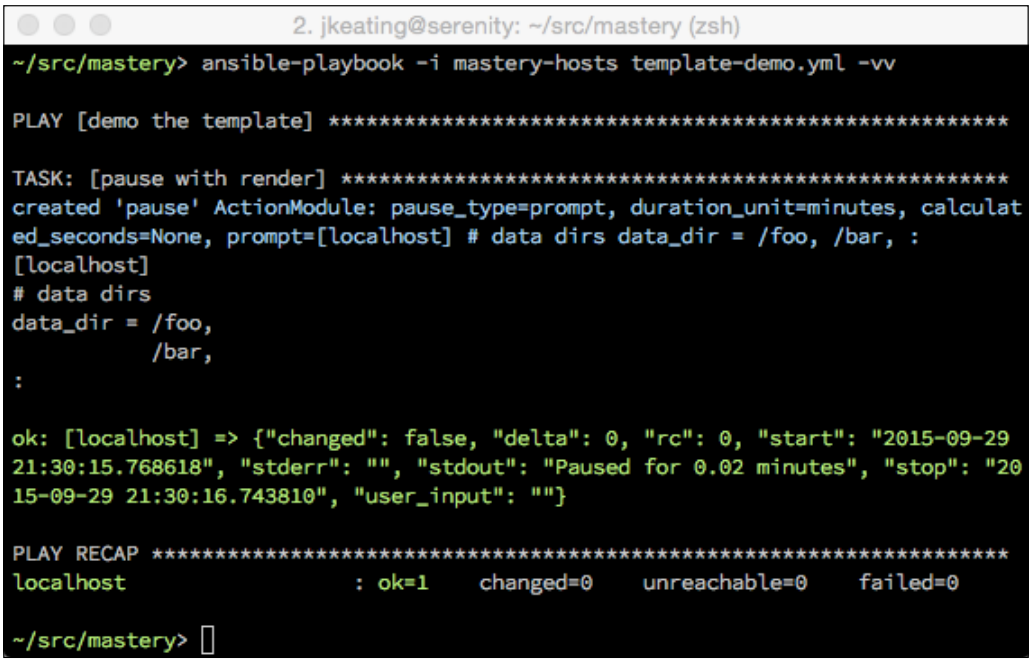
Having information related to the position within the loop can help with logic around what content to render. Considering our previous examples, instead of rendering multiple lines of `data_dir` to express each data directory, we could instead provide a single line with comma-separated values. Without having access to loop iteration data, this would be difficult, but when using this data, it can be fairly easy. For the sake of simplicity, this example assumes a trailing comma after the last item is allowed, and that whitespace (newlines) between items is also allowed:

```
# data dirs
{% for dir in data_dirs if dir != "/" %}
{% if loop.first %}
data_dir = {{ dir }},
{% else %}
           {{ dir }},
{% endif %}
{% else %}
# no data dirs found
{% endfor %}
```

The preceding example made use of the `loop.first` variable in order to determine if it needed to render the `data_dir =` part or if it just needed to render the appropriately spaced padded directory. By using a filter in the `for` statement, we get a correct value for `loop.first`, even if the first item in `data_dirs` is the undesired (`/`). To test this, we'll once again modify `demo.j2` with the updated template and modify `template-demo.yaml` to define some `data_dirs`, including one of `/` that should be filtered out:

```
---
- name: demo the template
  hosts: localhost
  gather_facts: false
  vars:
    data_dirs: ['/', '/foo', '/bar']
  tasks:
    - name: pause with render
      pause:
        prompt: "{{ lookup('template', 'demo.j2') }}"
```

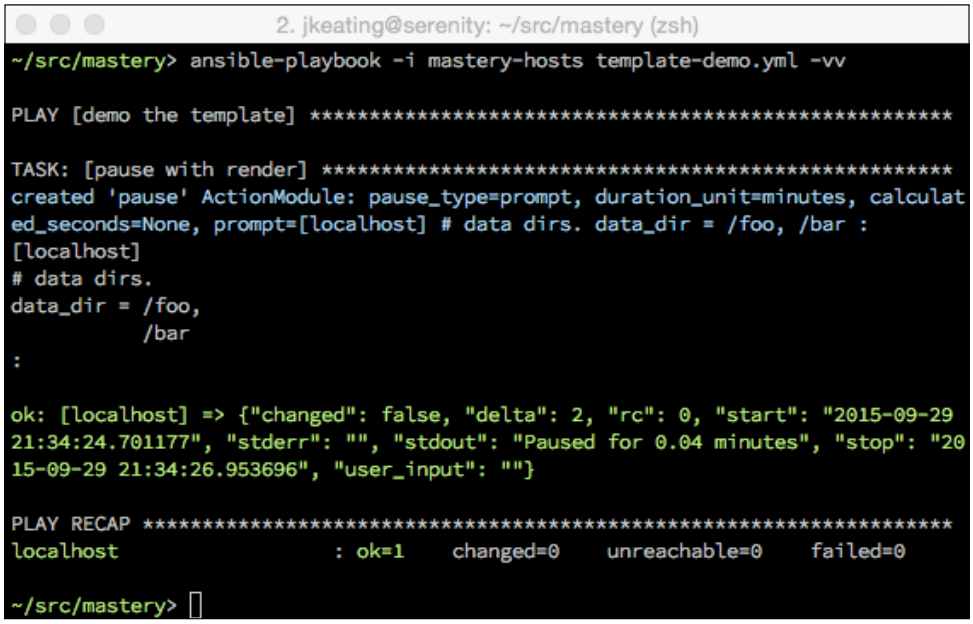
Now, we can execute the playbook and see our rendered content:



If in the preceding example trailing commas were not allowed, we could utilize inline `if` statements to determine if we're done with the loop and render commas correctly, as shown in the following example:

```
# data dirs.
{% for dir in data_dirs if dir != "/" %}
{% if loop.first %}
data_dir = {{ dir }}{{ ',' if not loop.last else '' }}
{% else %}
           {{ dir }}{{ ',' if not loop.last else '' }}
{% endif %}
{% else %}
# no data dirs found
{% endfor %}
```

Using inline `if` statements allows us to construct a template that will only render a comma if there are more items in the loop that passed our initial filter. Once more, we'll update `demo.j2` with the above content and execute the playbook:



Macros

The astute reader will have noticed that in the previous example, we had some repeated code. Repeating code is the enemy of any developer, and thankfully, Jinja2 has a way to help! A macro is like a function in a regular programming language—it's a way to define a reusable idiom. A macro is defined inside a `{% macro … %} … {% endmacro %}` block and has a name and can take zero or more arguments. Code within a macro does not inherit the namespace of the block calling the macro, so all arguments must be explicitly passed in. Macros are called within mustache blocks by name and with zero or more arguments passed in via parentheses. Let's create a simple macro named `comma` to take the place of our repeating code:

```
{% macro comma(loop) %}
{{ ',' if not loop.last else '' }}
{%- endmacro -%}
# data dirs.
{% for dir in data_dirs if dir != "/" %}
{% if loop.first %}
data_dir = {{ dir }}{{ comma(loop) }}
{% else %}
```

```
            {{ dir }}{{ comma(loop) }}
{% endif %}
{% else %}
# no data dirs found
{% endfor %}
```

Calling `comma` and passing it in the loop object allows the macro to examine the loop and decide if a comma should be emitted or not. You may have noticed some special marks on the `endmacro` line. These marks, the (`-`) next to the (`%`), instruct Jinja2 to strip the whitespace before, and right after the block. This allows us to have a newline between the macro and the start of the template for readability without actually rendering that newline when evaluating the template.

Macro variables

Macros have access inside them to any positional or keyword argument passed along when calling the macro. Positional arguments are arguments that are assigned to variables based on the order they are provided, while keyword arguments are unordered and explicitly assign data to variable names. Keyword arguments can also have a default value if they aren't defined when the macro is called. Three additional special variables are available:

- `varargs`
- `kwargs`
- `caller`

The `varargs` variable is a holding place for additional unexpected positional arguments passed along to the macro. These positional argument values will make up the `varargs` list.

The `kwargs` variable is like `varargs`; however, instead of holding extra positional argument values, it will hold a hash of extra keyword arguments and their associated values.

The `caller` variable can be used to call back to a higher level macro that may have called this macro (yes, macros can call other macros).

In addition to these three special variables are a number of variables that expose internal details regarding the macro itself. These are a bit complicated, but we'll walk through their usage one by one. First, let's take a look at a short description of each variable:

- `name`: The name of the macro itself
- `arguments`: A tuple of the names of the arguments the macro accepts

- `defaults`: A tuple of the default values
- `catch_kwargs`: A Boolean that will be defined as true if the macro accesses (and thus accepts) the `kwargs` variable
- `catch_varargs`: A Boolean that will be defined as true if the macro accesses (and thus accepts) the `varargs` variable
- `caller`: A Boolean that will be defined as true if the macro accesses the `caller` variable (and thus may be called from another macro)

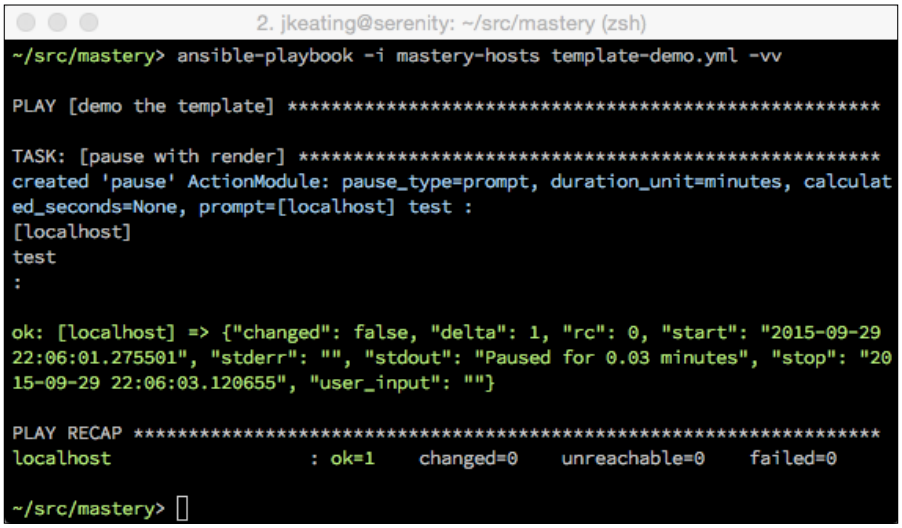
Similar to a class in Python, these variables need to be referenced via the name of the macro itself. Attempting to access these macros without prepending the name will result in undefined variables. Now, let's walk through and demonstrate the usage of each of them.

name

The `name` variable is actually very simple. It just provides a way to access the name of the macro as a variable, perhaps for further manipulation or usage. The following template includes a macro that references the name of the macro in order to render it in the output:

```
{% macro test() %}
{{ test.name }}
{%- endmacro -%}
{{ test() }}
```

If we were to update `demo.j2` with this template and execute the `template-demo.yaml` playbook, the output would be:

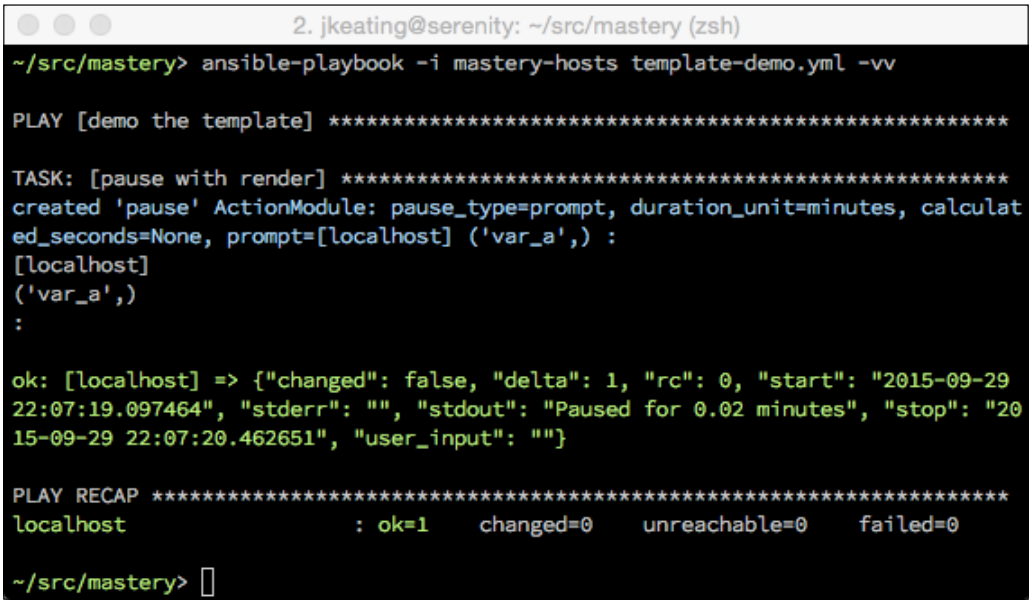


arguments

The `arguments` variable is a tuple of the arguments that the macro accepts. These are the explicitly defined arguments, not the special `kwargs` or `varargs`. Our previous example would have rendered an empty tuple `()`, so lets modify it to get something else:

```
{% macro test(var_a='a string') %}
{{ test.arguments }}
{%- endmacro -%}
{{ test() }}
```

Rendering this template will result in the following:

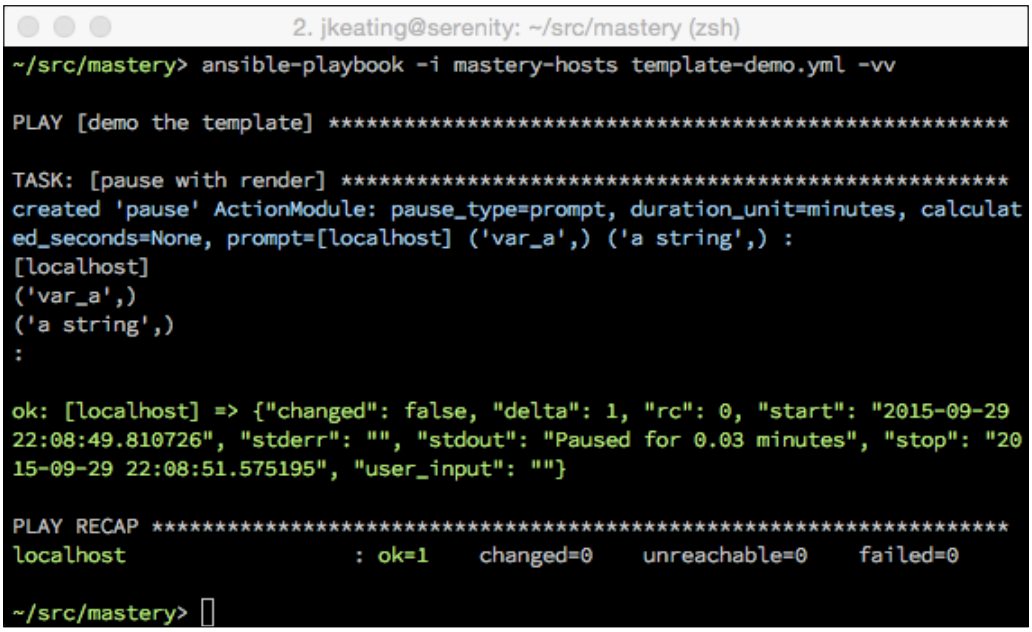


defaults

The `defaults` variable is a tuple of the default values for any keyword arguments that the macro explicitly accepts. Let's change our macro to display the default values as well as the arguments:

```
{% macro test(var_a='a string') %}
{{ test.arguments }}
{{ test.defaults }}
{%- endmacro -%}
{{ test() }}
```

Rendering this version of the template will result in the following:

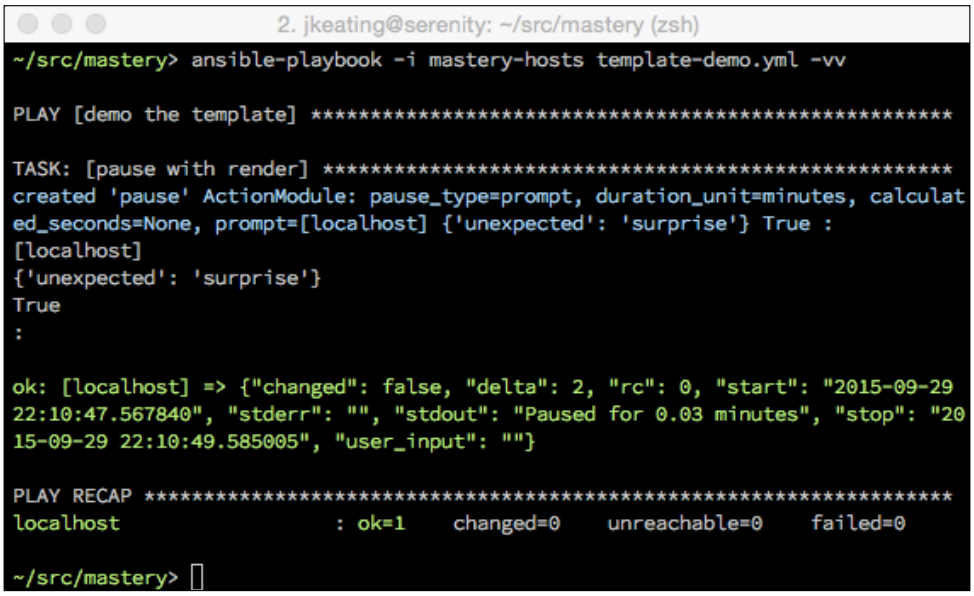


catch_kwargs

This variable is only defined if the macro itself accesses the `kwargs` variable in order to catch any extra keyword arguments that might have been passed along. Without accessing the `kwargs` variable, any extra keyword arguments in a call to the macro will result in an error when rendering the template. Likewise, accessing `catch_kwargs` without also accessing `kwargs` will result in an undefined error. Let's modify our example template again so that we can pass along extra `kwargs`:

```
{% macro test() %}
{{ kwargs }}
{{ test.catch_kwargs }}
{%- endmacro -%}
{{ test(unexpected='surprise') }}
```

The rendered version of this template will be:

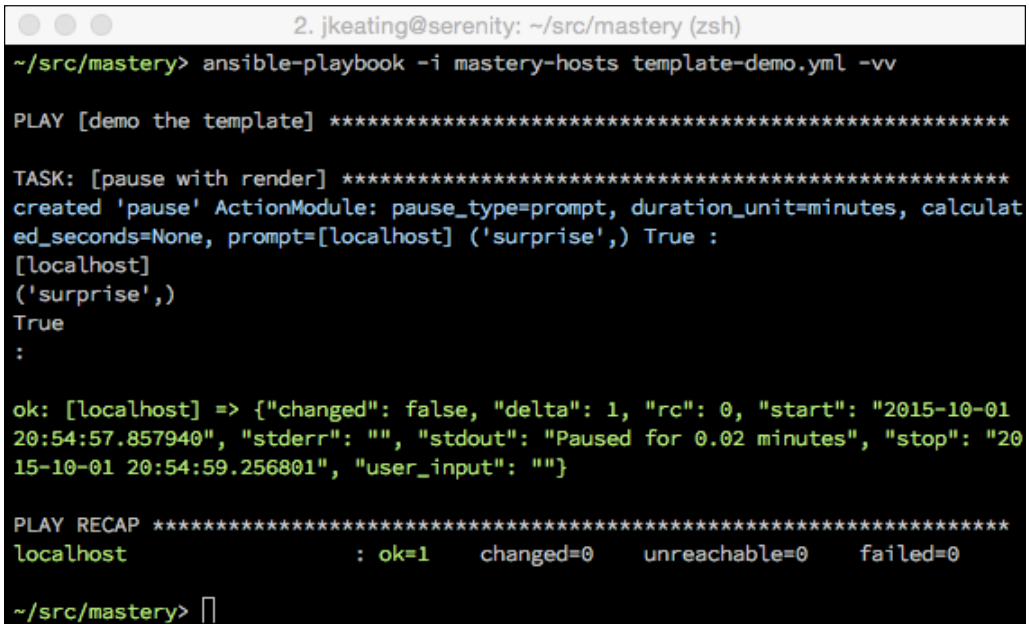


catch_varargs

Much like `catch_kwargs`, this variable exists if the macro accesses the `varargs` variable. Modifying our example once more, we can see this in action:

```
{% macro test() %}
{{ varargs }}
{{ test.catch_varargs }}
{%- endmacro -%}
{{ test('surprise') }}
```

The template's rendered result will be:

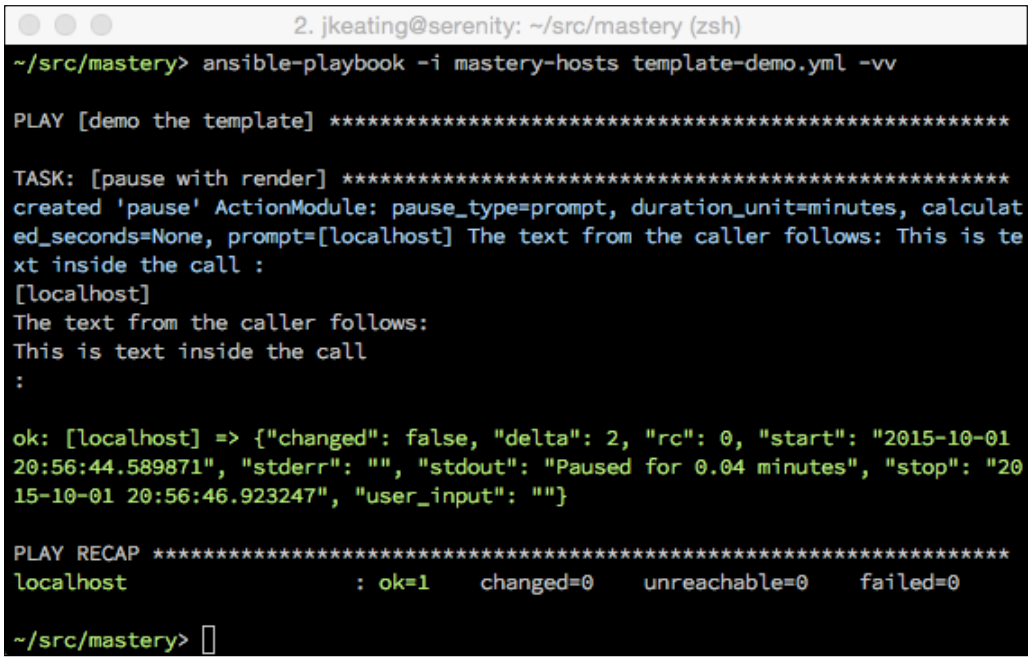


caller

The `caller` variable takes a bit more explaining. A macro can call out to another macro. This can be useful if the same chunk of the template is to be used multiple times, but part of the inside changes more than could easily be passed as a macro parameter. The `Caller` variable isn't exactly a variable, it's more of a reference back to the call in order to get the contents of that calling macro. Let's update our template to demonstrate the usage:

```
{% macro test() %}
The text from the caller follows:
{{ caller() }}
{%- endmacro -%}
{% call test() %}
This is text inside the call
{% endcall %}
```

The rendered result will be:



A call to a macro can still pass arguments to that macro—any combination of arguments or keyword arguments can be passed. If the macro utilizes `varargs` or `kwargs`, then extras of those can be passed along as well. Additionally, a macro can pass arguments back to the caller too! To demonstrate this, let's create a larger example. This time, our example will generate out a file suitable for an Ansible inventory:

```
{% macro test(group, hosts) %}
[{{ group }}]
{% for host in hosts %}
{{ host }} {{ caller(host) }}
{%- endfor %}
{%- endmacro -%}
{% call(host) test('web', ['host1', 'host2', 'host3']) %}
ssh_host_name={{ host }}.example.name ansible_sudo=true
{% endcall %}
{% call(host) test('db', ['db1', 'db2']) %}
ssh_host_name={{ host }}.example.name
{% endcall %}
```

Once rendered, the result will be:

```
2. jkeating@serenity: ~/src/mastery (zsh)
~/src/mastery> ansible-playbook -i mastery-hosts template-demo.yml -vv

PLAY [demo the template] ******************************************************

TASK: [pause with render] *****************************************************
created 'pause' ActionModule: pause_type=prompt, duration_unit=minutes, calculat
ed_seconds=None, prompt=[localhost] [web] host1 ssh_host_name=host1.example.name
 ansible_sudo=true host2 ssh_host_name=host2.example.name ansible_sudo=true host
3 ssh_host_name=host3.example.name ansible_sudo=true [db] db1 ssh_host_name=db1.
example.name db2 ssh_host_name=db2.example.name :
[localhost]
[web]
host1 ssh_host_name=host1.example.name ansible_sudo=true
host2 ssh_host_name=host2.example.name ansible_sudo=true
host3 ssh_host_name=host3.example.name ansible_sudo=true

[db]
db1 ssh_host_name=db1.example.name
db2 ssh_host_name=db2.example.name
:

ok: [localhost] => {"changed": false, "delta": 1, "rc": 0, "start": "2015-10-01
21:05:39.584425", "stderr": "", "stdout": "Paused for 0.03 minutes", "stop": "20
15-10-01 21:05:41.503232", "user_input": ""}

PLAY RECAP ********************************************************************
localhost                  : ok=1    changed=0    unreachable=0    failed=0

~/src/mastery>
```

We called the `test` macro twice, once per each group we wanted to define. Each group had a subtly different set of host variables to apply, and those were defined in the call itself. We saved ourselves some typing by having the macro call back to the caller passing along the host from the current loop.

Control blocks provide programming power inside of templates, allowing template authors to make their templates efficient. The efficiency isn't necessarily in the initial draft of the template. Instead, the efficiency really comes into play when a small change to a repeating value is needed.

Data manipulation

While control structures influence the flow of template processing, another tool exists to modify the contents of a variable. This tool is called a **filter**. Filters are like small functions, or methods, that can be run on the variable. Some filters operate without arguments, some take optional arguments, and some require arguments. Filters can be chained together as well, where the result of one filter action is fed into the next filter and the next. Jinja2 comes with many built-in filters, and Ansible extends these with many custom filters available to you when using Jinja2 within templates, tasks, or any other place Ansible allows templating.

Syntax

A filter is applied to a variable by way of the pipe symbol (`|`), followed by the name of the filter and then any arguments for the filter inside parentheses. There can be a space between the variable name and the pipe symbol, as well as a space between the pipe symbol and the filter name. For example, if we wanted to apply the filter `lower` (which makes all the characters lowercase) to the variable `my_word`, we would use the following syntax:

```
{{ my_word | lower }}
```

Because the lower filter does not take any arguments, it is not necessary to attach an empty parentheses set to it. If we use a different filter, one that requires arguments, we can see how that looks. Let's use the `replace` filter, which allows us to replace all occurrences of a substring with another substring. In this example, we want to replace all occurrences of the substring `no` with `yes` in the variable `answers`:

```
{{ answers | replace('no', 'yes') }}
```

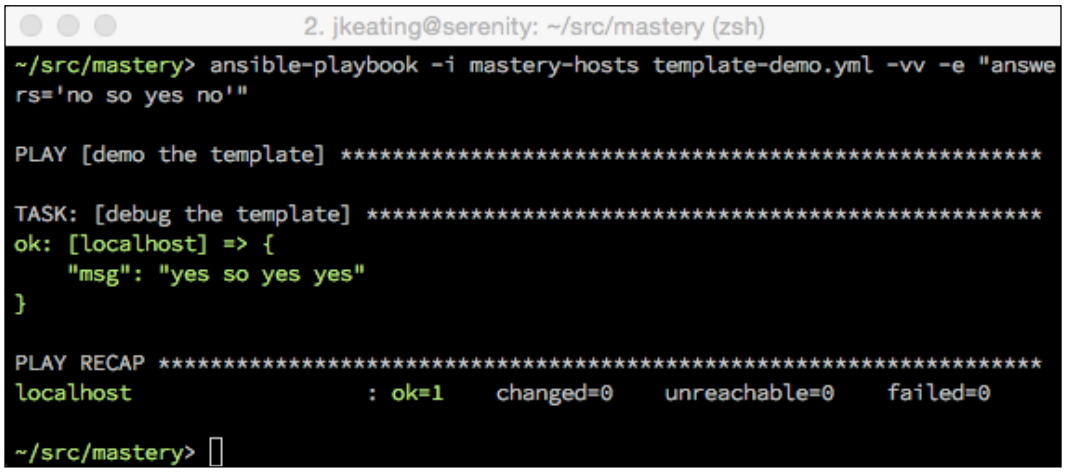
Applying multiple filters is accomplished by simply adding more pipe symbols and more filter names. Let's combine both `replace` and `lower` to demonstrate the syntax:

```
{{ answers | replace('no', 'yes') | lower }}
```

We can easily demonstrate this with a simple play that uses the `debug` command to render the line:

```
---
- name: demo the template
  hosts: localhost
  gather_facts: false
  tasks:
    - name: debug the template
      debug:
        msg: "{{ answers | replace('no', 'yes') | lower }}"
```

Now, we can execute the playbook and provide a value for `answers` at run time:



Useful built-in filters

A full list of the filters built into Jinja2 can be found in the Jinja2 documentation. At the time of writing this book, there are over 45 built-in filters, too many to describe here. Instead, we'll take a look at some of the more commonly used filters.

default

The `default` filter is a way of providing a default value for an otherwise undefined variable, which will prevent Ansible from generating an error. It is shorthand for a complex `if` statement checking if a variable is defined before trying to use it, with an `else` clause to provide a different value. Let's look at two examples that render the same thing. One will use the `if / else` structure, while the other uses the `default` filter:

```
{% if some_variable is defined %}
{{ some_variable }}
{% else %}
default_value
{% endif %}
{{ some_variable | default('default_value') }}
```

The rendered result of each of these examples is the same; however, the example using the `default` filter is much quicker to write and easier to read.

While `default` is very useful, proceed with caution if you are using the same variable in multiple locations. Changing a default value can become a hassle, and it may be more efficient to define the variable with a default at the play or role level.

count

The `count` filter will return the length of a sequence or hash. In fact, `length` is an alias of `count` to accomplish the same thing. This filter can be useful for performing any sort of math around the size of a set of hosts or any other case where the count of some set needs to be known. Let's create an example in which we set a `max_threads` configuration entry to match the count of hosts in the play:

```
max_threads: {{ play_hosts | count }}
```

random

The `random` filter is used to make a random selection from a sequence. Let's use this filter to delegate a task to a random selection from the `db_servers` group:

```
- name: backup the database
  shell: mysqldump -u root nova > /data/nova.backup.sql
  delegate_to: "{{ groups['db_servers'] | random }}"
  run_once: true
```

round

The `round` filter exists to round a number. This can be useful if you need to perform floating-point math and then turn the result into a rounded integer. The `round` filter takes optional arguments to define a precision (default of `0`) and a rounding method. The possible rounding methods are `common` (rounds up or down, the default), `ceil` (always round up), and `floor` (always round down). In this example, we'll round a math result to zero precision:

```
{{ math_result | round | int }}
```

Useful Ansible provided custom filters

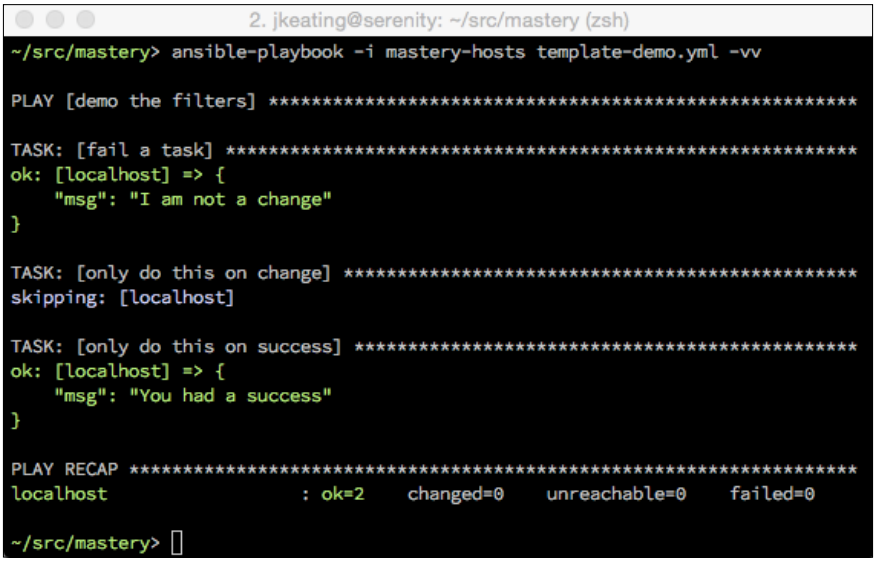
While there are many provided filters with Jinja2, Ansible includes some additional filters that playbook authors may find particularly useful. We'll outline a few of them here.

Filters related to task status

Ansible tracks task data for each task. This data is used to determine if a task has failed, resulted in a change, or was skipped all together. Playbook authors can register the results of a task and then use filters to easily check the task status. These are most often used in conditionals with later tasks. The filters are aptly named `failed`, `success`, `changed`, and `skipped`. They each return a Boolean value. Here is a playbook that demonstrates the use of a couple of these:

```
---
- name: demo the filters
  hosts: localhost
  gather_facts: false
  tasks:
    - name: fail a task
      debug:
        msg: "I am not a change"
      register: derp
    - name: only do this on change
      debug:
        msg: "You had a change"
      when: derp | changed
    - name: only do this on success
      debug:
        msg: "You had a success"
      when: derp | success
```

The output is as shown in the following screenshot:

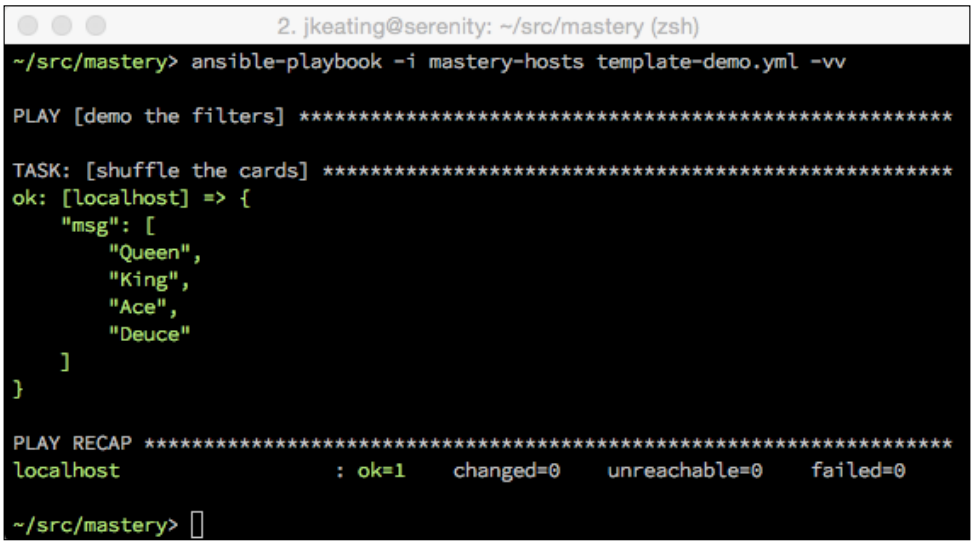


shuffle

Similar to the `random` filter, the `shuffle` filter can be used to produce randomized results. Unlike the `random` filter, which selects one random choice from a list, the `shuffle` filter will shuffle the items in a sequence and return the full sequence back:

```
---
- name: demo the filters
  hosts: localhost
  gather_facts: false
  tasks:
    - name: shuffle the cards
      debug:
        msg: "{{ ['Ace', 'Queen', 'King', 'Deuce'] | shuffle }}"
```

The output is as shown in the following screenshot:



Filters dealing with path names

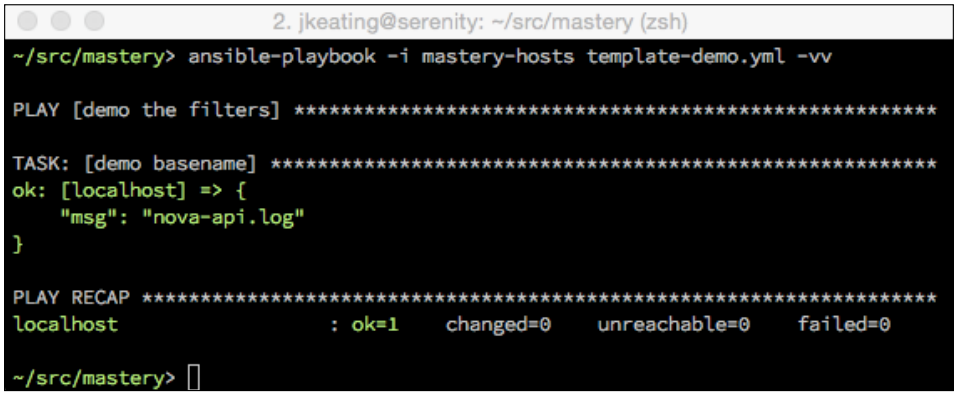
Configuration management and orchestration frequently refers to path names, but often only part of the path is desired. Ansible provides a few filters to help.

basename

To obtain the last part of a file path, use the `basename` filter. For example:

```
---
- name: demo the filters
  hosts: localhost
  gather_facts: false
  tasks:
    - name: demo basename
      debug:
        msg: "{{ '/var/log/nova/nova-api.log' | basename }}"
```

The output is as shown in the following screenshot:



dirname

The inverse of `basename` is `dirname`. Instead of returning the final part of a path, `dirname` will return everything except the final part. Let's change our previous play to use `dirname`, and run it again:

expanduser

Often, paths to various things are supplied with a user shortcut, such as `~/.stackrc`. However some uses may require the full path to the file. Rather than the complicated `command` and `register` call to use the shell to expand the path, the `expanduser` filter provides a way to expand the path to the full definition. In this example, the user name is `jkeating`:

```
---
- name: demo the filters
  hosts: localhost
  gather_facts: false
  tasks:
    - name: demo filter
      debug:
        msg: "{{ '~/.stackrc' | expanduser }}"
```

The output is as shown in the following screenshot:

Base64 encoding

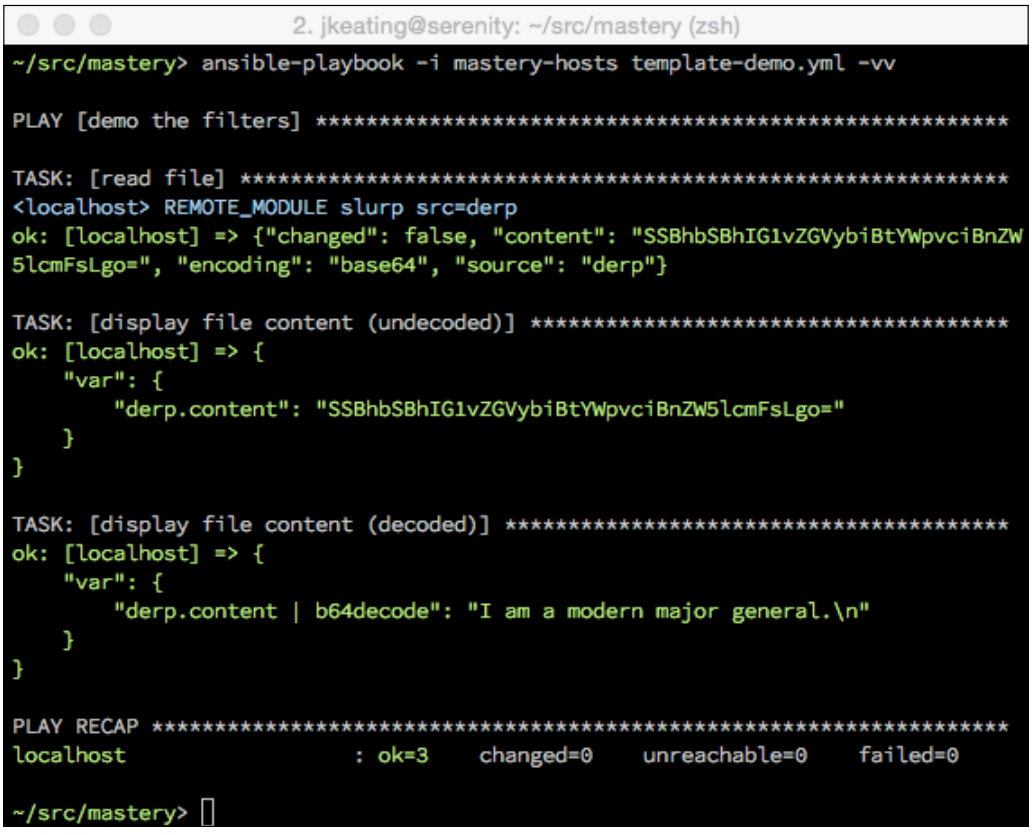
When reading content from remote hosts, like with the `slurp` module (used to read file content from remote hosts into a variable), the content will be `Base64` encoded. To decode such content, Ansible provides a `b64decode` filter. Similarly, when running a task that requires `Base64` encoded input, regular strings can be encoded with the `b64encode` filter.

Let's read content from the file `derp`:

```
---
- name: demo the filters
  hosts: localhost
  gather_facts: false
```

```
tasks:
  - name: read file
    slurp:
      src: derp
    register: derp
  - name: display file content (undecoded)
    debug:
      var: derp.content
  - name: display file content (decoded)
    debug:
      var: derp.content | b64decode
```

The output is as shown in the following screenshot:



Searching for content

It is fairly common in Ansible to search a string for a substring. In particular, the common administrator task of running a command and grepping the output for a particular key piece of data is a reoccurring construct in many playbooks. While it's possible to replicate this with a `shell` task to execute a command and pipe the output into `grep`, and use careful handling of `failed_when` to catch `grep` exit codes, a far better strategy is to use a `command` task, `register` the output, and then utilize Ansible-provided regex filters in later conditionals. Let's look at two examples, one using the `shell`, `pipe`, `grep` method, and another using the `search` filter:

```
- name: check database version
  shell: neutron-manage current |grep juno
  register: neutron_db_ver
  failed_when: false
- name: upgrade db
  command: neutron-manage db_sync
  when: neutron_db_ver|failed
```

The above example works by forcing Ansible to always see the task as successful, but assumes that if the exit code from the shell is non-zero then the string `juno` was not found in the output of the `neutron-manage` command. This construct is functional, but a bit clunky, and could mask real errors from the command. Let's try again using the `search` filter:

```
- name: check database version
  command: neutron-manage current
  register: neutron_db_ver
- name: upgrade db
  command: neutron-manage db_sync
  when: not neutron_db_ver.stdout | search('juno')
```

This version is much cleaner to follow and does not mask errors from the first task.

The `search` filter searches a string and will return `True` if the substring is found anywhere within the input string. If an exact complete match is desired instead, the `match` filter can be used. Full Python regex syntax can be utilized inside the `search` / `match` string.

Omitting undefined arguments

The `omit` variable takes a bit of explaining. Sometimes, when iterating over a hash of data to construct task arguments, it may be necessary to only provide some arguments for some of the items in the hash. Even though Jinja2 supports in-line `if` statements to conditionally render parts of a line, this does not work well in an Ansible task. Traditionally, playbook authors would create multiple tasks, one for each set of potential arguments passed in, and use conditionals to sort the loop members between each task set. A recently added magic variable named `omit` solves this problem when used in conjunction with the `default` filter. The `omit` variable will remove the argument that the variable was used with altogether.

To illustrate how this works, let's consider a scenario where we need to install a set of Python packages with `pip`. Some of the packages have a specific version, while others do not. These packages are in a list of hashes named `pips`. Each hash has a `name` key and potentially a `ver` key. Our first example utilizes two different tasks to complete the installs:

```
- name: install pips with versions
  pip:
    name: "{{ item.name }}"
    version: "{{ item.ver }}"
  with_items: pips
  when: item.ver is defined
- name: install pips without versions
  pip:
    name: "{{ item.name }}"
  with_items: pips
  when: item.ver is undefined
```

This construct works, but the loop is iterated twice and some of the iterations will be skipped in each task. This next example collapses the two tasks into one and utilizes the `omit` variable:

```
- name: install pips
  pip:
    name: "{{ item.name }}"
    version: "{{ item.ver | default(omit) }}"
  with_items: pips
```

This example is shorter, cleaner, and doesn't generate extra skipped tasks.

Python object methods

Jinja2 is a Python-based template engine. Because of this, Python object methods are available within templates. Object methods are methods, or functions, that are directly accessible by the variable object (typically a `string`, `list`, `int`, or `float`). A good way to think about this is if you were writing Python code and could write the variable, then a period, then a method call, you would then have access to do the same in Jinja2. Within Ansible, only methods that return modified content or a Boolean are typically used. Let's explore some common object methods that might be useful in Ansible.

String methods

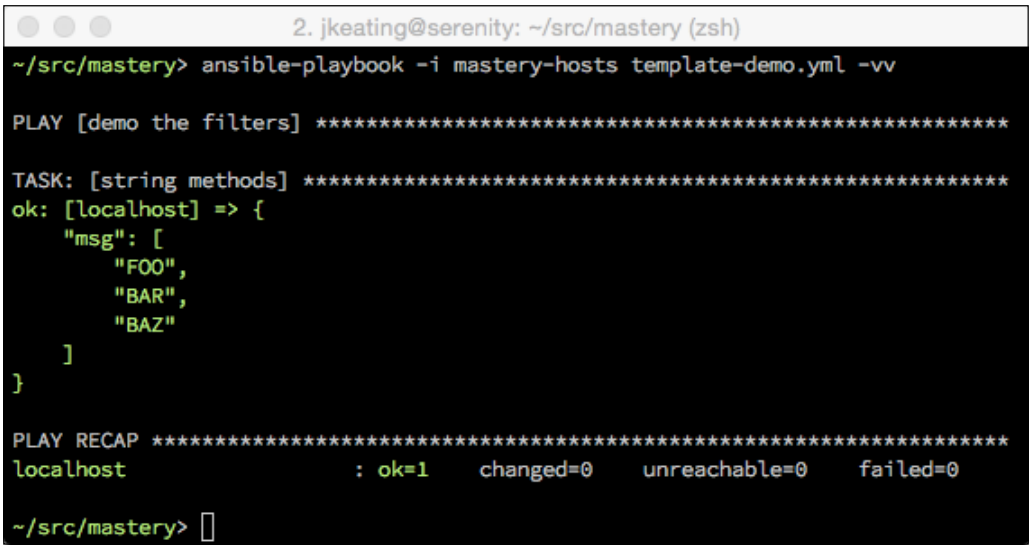
String methods can be used to return new strings or a list of strings modified in some way, or to test the string for various conditions and return a Boolean. Some useful methods are as follows:

- `endswith`: Determines if the string ends with a substring
- `startswith`: Like `endswith`, but from the start
- `split`: Splits the string on characters (default is space) into a list of substrings
- `rsplit`: The same as split, but starts from the end of the string and works backwards
- `splitlines`: Splits the string at newlines into a list of substrings
- `upper`: Returns a copy of the string all in uppercase
- `lower`: Returns a copy of the string all in lowercase
- `capitalize`: Returns a copy of the string with just the first character in uppercase

We can create a simple play that will utilize some of these methods in a single task:

```
---
- name: demo the filters
  hosts: localhost
  gather_facts: false

  tasks:
    - name: string methods
      debug:
        msg: "{{ 'foo bar baz'.upper().split() }}"
```

The output is as shown in the following screenshot:



Because these are object methods, we need to access them with dot notation rather than as a filter via (`|`).

List methods

Only a couple methods do something other than modify the list in-place rather than returning a new list, and they are as follows:

- `index`: Returns the first index position of a provided value
- `count`: Counts the items in the list

int and float methods

Most `int` and `float` methods are not useful for Ansible.

Sometimes, our variables are not exactly in the format we want them in. However, instead of defining more and more variables that slightly modify the same content, we can make use of Jinja2 filters to do the manipulation for us in the various places that require that modification. This allows us to stay efficient with the definition of our data, preventing many duplicate variables and tasks that may have to be changed later.

Comparing values

Comparisons are used in many places with Ansible. Task conditionals are comparisons. Jinja2 control structures often use comparisons. Some filters use comparisons as well. To master Ansible's usage of Jinja2, it is important to understand which comparisons are available.

Comparisons

Like most languages, Jinja2 comes equipped with the standard set of comparison expressions you would expect, which will render a Boolean `true` or `false`.

The expressions in Jinja2 are as follows:

Expression	Definition
`==`	Compares two objects for equality
`!=`	Compares two objects for inequality
`>`	True if the left-hand side is greater than the right-hand side
`<`	True if the left-hand side is less than the right- hand side
`>=`	True if the left-hand side is greater than or equal to the right-hand side
`<=`	True if the left-hand side is less than or equal to the right-hand side

Logic

Logic helps group two or more comparisons together. Each comparison is referred to as an operand:

- `and`: Returns `true` if the left and the right operand are true
- `or`: Returns `true` if the left or the right operand is true
- `not`: Negates an operand
- `()`: Wraps a set of operands together to form a larger operand

Tests

A test in Jinja2 is used to see if a value is something. In fact, the `is` operator is used to initiate a test. Tests are used any place a Boolean result is desired, such as with `if` expressions and task conditionals. There are many built-in tests, but we'll highlight a few of the particularly useful ones.

- `defined`: Returns true if the variable is defined
- `undefined`: The opposite of defined

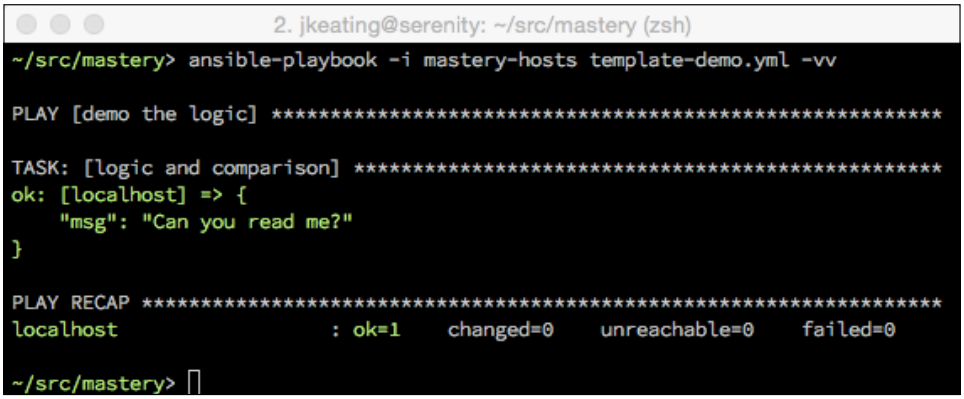
- `none`: Returns true if the variable is defined, but the value is none
- `even`: Returns true if the number is divisible by 2
- `odd`: Returns true if the number is not divisible by 2

To test if a value is not something, simply use `is not`.

We can create a playbook that will demonstrate some of these value comparisons:

```
---
- name: demo the logic
  hosts: localhost
  gather_facts: false
  vars:
    num1: 10
    num3: 10
  tasks:
    - name: logic and comparison
      debug:
        msg: "Can you read me?"
      when: num1 >= num3 and num1 is even and num2 is not defined
```

The output is as shown in the following screenshot:



Summary

Jinja2 is a powerful language that is used by Ansible. Not only is it used to generate file content, but it is also used to make portions of playbooks dynamic. Mastering Jinja2 is vital for creating and maintaining elegant and efficient playbooks and roles.

In the next chapter, we will explore more in depth Ansible's capability to define what constitutes a change or failure for tasks within a play.

Thank you for buying

Mastering Ansible

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

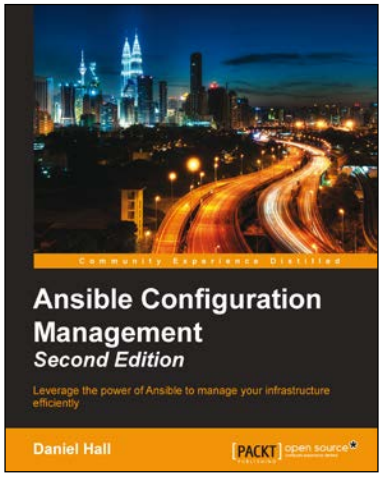
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



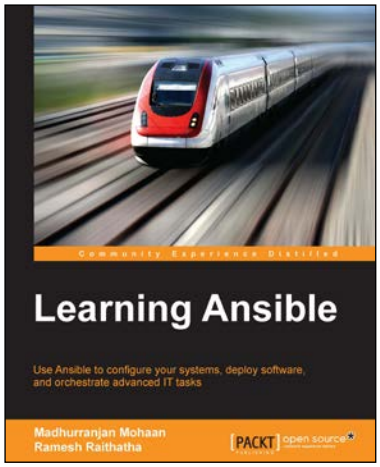
Ansible Configuration Management

Second Edition

ISBN: 978-1-78528-230-0 Paperback: 122 pages

Leverage the power of Ansible to manage your infrastructure efficiently

1. Configure Ansible on your Linux and Windows machines effectively.
2. Extend Ansible to add features such as looping, conditional executions, and task delegations.
3. Explore the capabilities of Ansible from basic to more advanced topics with the help of this step-by-step guide.

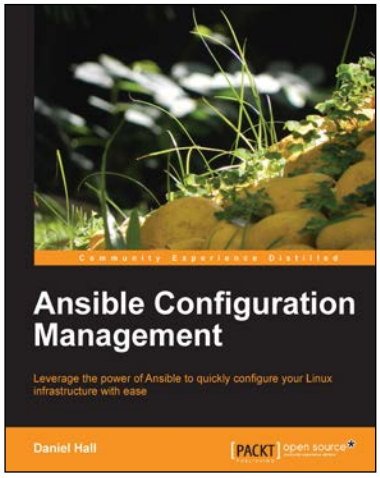


Learning Ansible

ISBN: 978-1-78355-063-0 Paperback: 308 pages

Use Ansible to configure your systems, deploy software, and orchestrate advanced IT tasks

1. Use Ansible to automate your infrastructure effectively, with minimal effort.
2. Customize and consolidate your configuration management tools with the secure and highly-reliable features of Ansible.
3. Unleash the abilities of Ansible and extend the functionality of your mainframe system through the use of powerful, real-world examples.

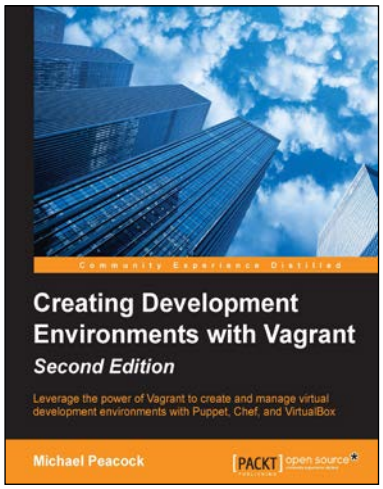


Ansible Configuration Management

ISBN: 978-1-78328-081-0 Paperback: 92 pages

Leverage the power of Ansible to quickly configure your Linux infrastructure with ease

1. Starts with the most simple usage of Ansible and builds on that.
2. Shows how to use Ansible to configure your Linux machines.
3. Teaches how to extend Ansible to add features you need.
4. Explains techniques for using Ansible in large, complex environments.



Creating Development Environments with Vagrant

Second Edition

ISBN: 978-1-78439-702-9 Paperback: 156 pages

Leverage the power of Vagrant to create and manage virtual development environments with Puppet, Chef, and VirtualBox

1. Get your projects up and running quickly and effortlessly by simulating complicated environments that can be easily shared with colleagues.
2. Provision virtual machines using Puppet, Ansible, and Chef.