
become
Ansible

become Ansible

Zero to Production-Ready

Josh Duffney

Version 1.0.4, 2020-10-21

Table of Contents

Preface

It seemed so simple: Just automate everything. Automate all the things and my job will be easy. I'll no longer have to do that trivial and mundane work; perhaps I'll automate so much I'll automate myself out of a job. But how do I deploy and configure infrastructure? It can't be that difficult…

I've spent the last six years automating infrastructure. It started off with monolithic scripts that built out new infrastructure. That was then refactored into individual components, each of which handled a single layer of the process. At the time, I wrote everything myself; it worked great for me. But things started to fall apart as I advocated for my team and organization to adopt it. What I had created was fragile and I became the bottleneck. I soon realized that the maintenance of the project alone was a full-time job.

Then, in June of 2014, I listened to a podcast: Episode 275 of the PowerScripting Podcast with PowerShell MVP Steve Murawski. It was on desired state configuration (DSC) and at the time I thought I had the solution to my problems. DSC was exactly what I was looking for, a way to describe my infrastructure as code! What I didn't realize at the time was that this podcast would change the trajectory of my career. This podcast started my journey into DevOps.

As I learned DSC, I became aware of what Jeffery Snover meant when he said, "DSC is a platform, not a tool." While attempting to implement DSC in two different organizations, I learned that it's the foundation, the engine behind what makes the changes, but it requires tooling and process to be built around it. That tooling and process are known as configuration management.

Like any self-respecting engineer would do, I decided to build my own. Most of what existed on the market came with a price tag or required me to learn a special language. I wasn't willing to go with either option at the time. After several iterations and many, *many* months of development, I finished building a configuration management tool. It was work I'm still proud of today – and versions of it are still in use. Yet I ran into the same constraint as before, time.

The more the tooling got used, the more the feature and bug work piled up. Another issue was documentation. Without documentation, how could I expect others to stop reinventing the wheel and use what was already built? Again, development and maintenance became a full-time job. It wasn't feasible. I had other work to do and I didn't want to work on this project 100% of the time. I had helped create a tire, but this tire couldn't hold much air. It wouldn't get us to where I wanted to be.

OK, so I had helped create a flat tire, but you can drive on a flat tire. At least for a while. It was enough to start moving in the right direction. The good news was that creating a home-grown configuration management tool showed such promise and value that a team was formed to implement configuration management for the entire organization. I was asked if I'd be interested in joining that team. Of course I was interested! But I also had concerns. I was comfortable; for the first time in my career I worked on a team with a manageable toil level and a cushy on-call. Yet, the invitation was too tempting. I had to join.

And that decision is the reason this book exists. Our objective was to increase the stability and reliability of the environments by automating the deployment and configuration of infrastructure. We had no idea how steep the hill we were about to climb was. But the pain, friction, and frustration of staying the same exceeded the pain of changing. So we climbed.

What follows is a distillation and reorganization of the lessons learned while an engineering team brought Ansible into an organization.

Acknowledgements

The Team

- Brett McCarty
- Nate Foreman aka "Abe Froman sausage king of chicago"
- Jeff Robertson
- Brad Lane
- Jai Kang
- Eric Boffeli
- Spencer Flanders

About the book

become Ansible aims to be a road map for engineers, teams, and organizations alike. This book guides you through setting up an Ansible environment, teaching you the fundamentals of Ansible through practical examples. It also demonstrates and discusses the kinds of implementation details and design considerations you'll encounter while onboarding a team or bringing Ansible into an organization.

About the author

Hello, reader. My name is Josh Duffney. I started my career in tech proudly stating I'm not developer. Oddly enough, that's what I do. But instead of building apps I build infrastructure. As you start to define your infrastructure as code the difference between sysadmin and developer fades.

If you'd like to learn more about my story, here are a few blog posts I've written along the way.

Doubling My Salary a PowerShell Story

Becoming a Craftsman, the Journal of a DevOps Engineer

You're an Engineer Be an Engineer

I use Twitter as my journal, follow me @joshduffney

if you're not part of the attention economy, subscribe to my weekly mailbrew.

Who should read this book

This book is for DevOps practitioners who are interested in using Ansible to orchestrate automation or to implement configuration management and infrastructure as code. It does not assume that you have any prior knowledge of DevOps or Ansible. Whether you're a sysadmin or a developer, this book is for you, as Ansible offers an intersection between systems administration and development.

The most important things to have are an open mind and a willingness to learn.

How this book is organized

become Ansible consists of two parts, each of which builds on the knowledge and progress that came before. It is therefore best to read the book cover to cover and then use it as a reference.

Part 1 answers the question "How do I set up an Ansible environment?". It starts off by walking you through building Ansible in a Docker container. Using a container offers several benefits. It:

- provides a consistent development experience for you and your team

- separates the Ansible environment from the infrastructure it aims to manage
- reduces management overhead (less infrastructure to manage)
- provides an immutable Ansible environment.

The book then guides you through how to connect Ansible to AWS or Azure. Once you've connected to a cloud provider, you learn how to deploy a development environment to AWS or Azure with Ansible.

Part 2 answers the question "How do I use Ansible?". You will learn how to issue Ansible commands, develop Ansible playbooks, use variables, create inventory files, use groups, create reusable and sharable roles, how to leverage an external source for a dynamic inventory and how to deploy Ansible using GitHub actions with a release pipeline taking you from zero to production-ready.

Part 1: Setup an Ansible Environment

Ansible Development with Containers

```
Objective
Run Ansible within a Docker Container

Steps
1. Install Docker Desktop
2. Create a Dockerfile
3. Build a Docker Image
4. Run a Docker Container
5. Push a Docker Image to DockerHub
```

In this chapter you will install Docker Desktop and create a Dockerfile. Within the Dockerfile, you will provide the instructions necessary to programmatically install Ansible. After the Dockerfile is written, you will use Docker commands to build an image from the Dockerfile and run a container from that image. By the end of the chapter, you'll have an Ansible environment running in a Docker container.

Install Docker Desktop

Before you can build, create, or run Docker containers, you first have to install the necessary software. Docker Desktop is one of many programs that allow you to work with Docker containers. Docker Desktop is a quick and simple way of getting started with Docker and it's a great solution for local development. It supports Mac and Windows operating systems.

1. Open the Docker Desktop product page
2. Select your operating system (Windows or Mac)
3. Click **Get Docker** and download the executable
4. Run the executable and follow installation prompts

NOTE	The default shell for this book is bash. But, by all means stay in PowerShell. Most of the commands transfer just fine. :)

Create a Dockerfile

A Dockerfile is a text document that contains all the commands used to assemble an image. The commands represent the codified instructions that make up a Docker image.

From

The Dockerfile starts by declaring the base image to use. This is the foundation of the container. Ansible runs on a few different Linux distributions, but for this exercise you'll use CentOS. The first

line is FROM centos:centos7.7.1908. You'll notice that there is more in that than just the distribution name. A version is also included which in Docker terms is a tag. The tag is used to version lock the container. In this example, the base container image is centos and the tag is centos7.7.1908.

Dockerfile

```
FROM centos:centos7.7.1908
```

Run

RUN is used to execute commands that build up the Docker image. Each RUN command builds a layer on top of the base image. Ansible can be installed with PIP, a Python package manager. Before you install PIP, you'll want to update the system and install some additional packages which are necessary to install PIP. The OpenSSH-clients package is also required because Ansible uses SSH to connect to Linux hosts.

Dockerfile

```
FROM centos:centos7.7.1908

RUN yum check-update; \
    yum install -y gcc libffi-devel python3 epel-release; \
    yum install -y openssh-clients; \
    pip3 install --upgrade pip; \
    pip3 install "ansible==2.9.12"
```

TIP

Use a single RUN command

Each command in a Dockerfile creates a layer that adds up to build the image. It's best practice to use as few RUN commands as possible, to prevent layer sprawl.

Pages 7-13 are not included in this preview.

Connect to the Cloud

```
Objective
Connect Ansible to a Cloud Platform

Steps
1. Choose connection method
2. Create environment variables
3. Verify connection to the cloud platform

Options
- Connect to AWS
- [Connect to Azure]
```

Complete the section that relates to the cloud platform of your choice.

Connect to AWS

```
Steps to Connect to AWS
1. Create Environment Variables
2. Get AWS Caller Information
3. Install boto and boto3
```

In this section, you will connect your Ansible container to AWS (Amazon Web Services). Before you begin, you will need an active AWS account, an access key, and a secret access key for an existing IAM user in the AWS account.

Prerequisites

- AWS Account
- Access Key

> **TIP** *Docker command to start Ansible container*
>
> `docker run -it --rm --volume "$(pwd):/ansible" --workdir /ansible ansible`

Obtain Access Key & Secret Access Key

Before you can continue you need to obtain an access key and secret access key for an AWS IAM user. That user must have the AdministratorAccess or SystemAdministrator policy assigned to it for your AWS account. You can use an existing access key or generate a new access key.

Create Environment Variables

Ansible uses the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` to authenticate to AWS. Populate these environment variables with an access key for an existing IAM user in AWS.

Pages 15-22 are not included in this preview.

Deploy an Ansible Dev Environment

```
Objective
Deploy an Ansible development environment to the cloud.

Steps
1. Download Code Samples
2. Deploy Ansible Development Environment
    - Deploy to AWS
    - Deploy to Azure
3. Connect to the Environment
```

In this section, you will deploy an Ansible development environment to either AWS or Azure. The Ansible development environment will consist of two virtual machines, one running a Windows Server OS (operating system) and the other running a Linux OS. These two virtual machines will be configured with Ansible as you progress throughout the book.

Download Code Samples

All code samples for this book are stored in the becomeansible public GitHub repository. You can either download the repository as an archive or clone it with git. The following steps walk you through cloning the repository.

Use yum to install git

```
yum install -y git
```

Install git by using the `yum install` command.

Clone the repository with git

```
git clone https://github.com/Duffney/becomeansible.git
```

Clone the repository using `git clone`. Once the repository has been cloned, the code will be available in a directory named `becomeansible`.

Change directories

```
cd becomeansible/chapter-03
```

Use the `cd` command to change directories to `becomeansible/chapter-03`. The `chapter-03` directory contains all the source code required for this chapter.

NOTE	Another approach is to clone the repository locally and then share the directory with the Docker container using a volume. See Chapter 1 for details of Docker volumes.

Deploy Ansible Development Environment

You have a lot of options when deploying cloud resources. Each cloud provider has its own set of tools. Azure has the Azure Resource Manager. AWS has Cloudformation. There are also several third-party tools, such as Terraform and of course, Ansible. The rest of this chapter will walk you through deploying two virtual machines to either AWS or Azure using Ansible. Follow the section of this chapter that discusses your preferred cloud platform.

Deploy to AWS

```
Deploy an Ansible Development Environment to AWS
1. Create a VPC
2. Deploy a Windows EC2 Instance
3. Deploy a Linux EC2 Instance
```

Your Ansible development environment in AWS will consist of a VPC (virtual private cloud), a subnet, an internet gateway, a route for public traffic into the VPC, and two EC2 (Elastic Compute Cloud) instances. One EC2 instance will use a Windows Server 2019 AMI (Amazon Machine Image) and the other will use an AMI for Red Hat Enterprise Linux 8. Both AMIs used are within the free tier.

The following Ansible modules are used to deploy the resources to AWS using Ansible playbooks.

Table 1. AWS Resources & AWS Ansible Modules

AWS Resource	**Ansible Module**
VPC	ec2_vpc_net
Subnet	ec2_vpc_subnet
Internet Gateway	ec2_vpc_igw
Route Table	ec2_vpc_route_table
Security Group	ec2_group
Key Pair	ec2_key
EC2 Instance	ec2
Elastic IP Address	ec2_eip

Ansible codifies your infrastructure in YAML files called Ansible playbooks. You will use pre-written Ansible playbooks to deploy the Ansible development environment to AWS. Several of the AWS resources depend on other resources. These dependencies mean that you have to run the playbooks in the right order.

The following is an introduction to Ansible playbooks. It will give you a glimpse of what is possible with Ansible.

Pages 25-28 are not included in this preview.

Deploy to Azure

```
Deploy an Ansible Development Environment to Azure
1. Create a Resource Group
2. Deploy a Windows Virtual Machine
3. Deploy a Linux Virtual Machine
```

Your Ansible development environment in Azure will consist of two virtual machines: a Windows Server 2019 virtual machine and a Linux virtual machine running CentOS. Each virtual machine in Azure requires several Azure resources and each of those resources is managed with Ansible using different Ansible modules.

The following table lists all the Azure resources required to deploy a virtual machine to Azure along with the Ansible modules for deploying and configuring those resources with Ansible.

Table 2. Azure Resources & Azure Ansible Modules

Azure Resource	**Ansible Module**
Resource Group	azure_rm_resourcegroup
Virtual Network	azure_rm_virtualnetwork
Subnet	azure_rm_subnet
Public IP Address	azure_rm_publicipaddress
Network Security Group	azure_rm_securitygroup
Network Interface Card	azure_rm_networkinterface
Custom Script Extension	azure_rm_virtualmachineextension
Virtual Machine	azure_rm_virtualmachine

Ansible codifies your infrastructure in YAML files called Ansible playbooks. You will use pre-written Ansible playbooks to deploy the Ansible development environment to Azure. Several of the Azure resources depend on other resources. These dependencies mean that you have to run the playbooks in the right order.

Pages 30-38 are not included in this preview.

Part 2: Become Ansible

Starting at the Source

Ansible changes the way you work. You shift from a world where tasks are executed manually to tasks being automated and codified. Long gone are the days of relying on shell history and file shares to store your code. You need something better. You need source control.

You need source control because you need a place to store and audit your code. You also need a system that allows you to collaborate on that code. If the adoption of Ansible is successful it will no longer be just you or even your team committing changes. It will be the entire organization.

Source control is a broad topic. Entire books are dedicated to it. The good news is that you don't need to understand the cavernous depths of source control. You just need enough information to become competent.

What is Git?

No more undo commands and v1 v2 files. You will no longer have to rely on those methods to track and roll back changes to your code.

When you use Git to store your code it tracks when files are added, changed, renamed, or deleted. Imagine being able to see the difference between every commit. Having a full historical audit of the code base is a powerful thing. Better yet, you have the ability to roll back.

You use Git by either cloning (copying) an existing repository or by initializing a new repository. As you make changes locally, you track changes to files through a series of Git commands. These commands allow you to add, commit, and push those changes to a centralized server so they can be shared with others.

Git is a distributed source control system. It enables you to work locally and independently. But at the same time, it supports collaboration, as you push your changes to the centralized server and pull changes contributed by your teammates to the codebase.

Before you begin

Install Git

Unix/Linux
- http://git-scm.com/download/linux

Mac
- Manual
    http://git-scm.com/download/mac
- Commandline
    brew install git

Windows
- Manual
    https://gitforwindows.org
- Comandline
    choco install git.install

Confirm git was installed by getting the version.

```
git --version
```

Create a GitHub account

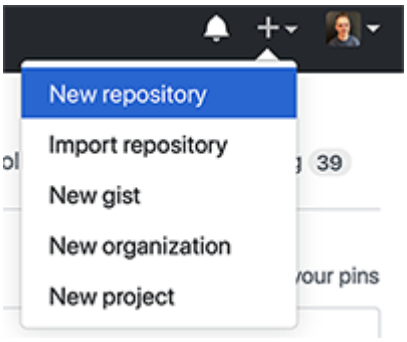
1. Go to https://github.com/join
2. Fill out the **Create your account** page with
  - Username
  - Email address
  - Password
3. Complete the **Verify your account** puzzle
4. Click **Create account**

Creating and using a Git repository

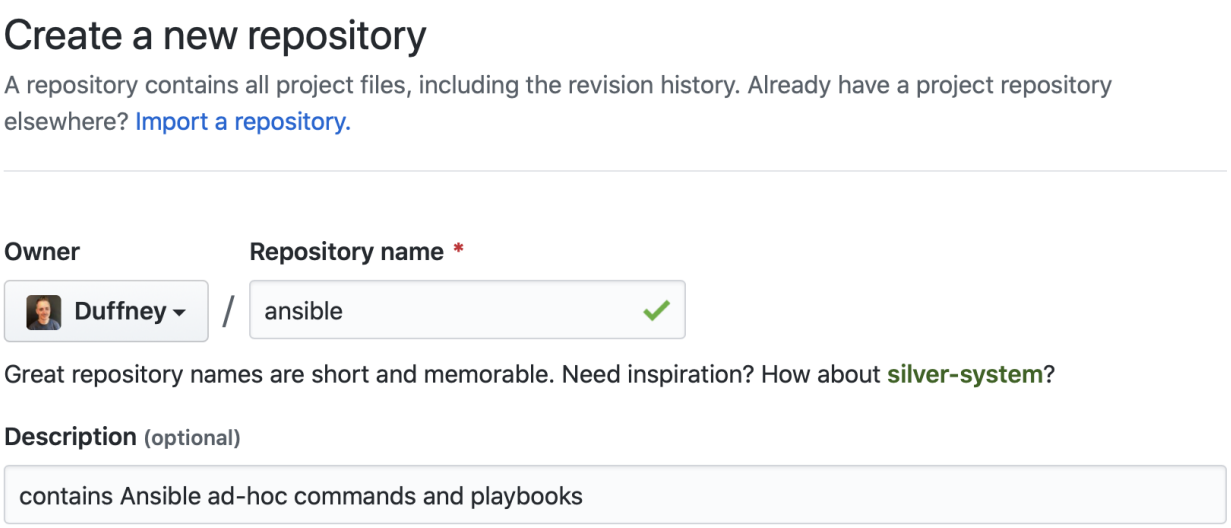
Once logged into your GitHub account, create a repository. Creating the repository in GitHub will initialize the repository and prepare it for use.

Create a GitHub repository.

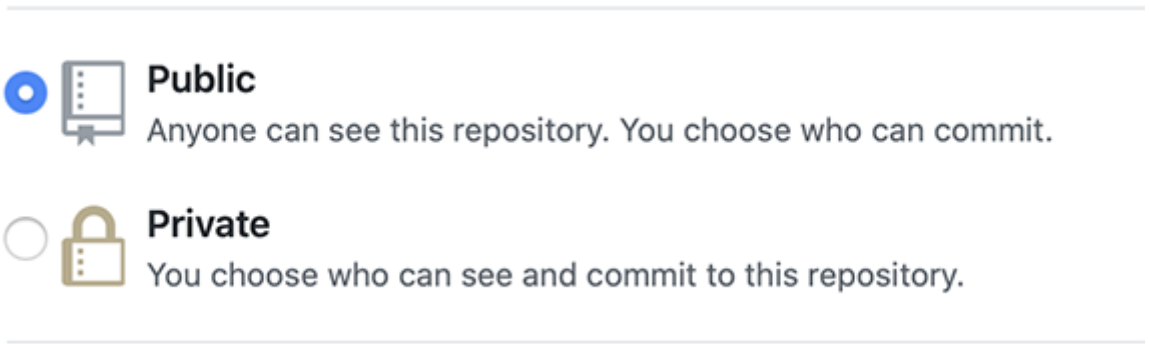
1. In the upper-right corner click the + and then click **New repository**.



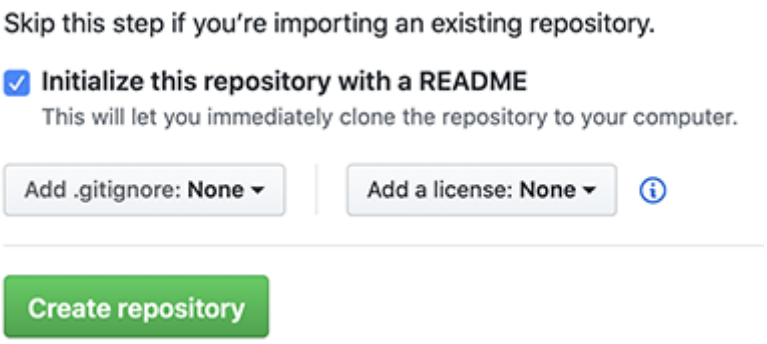
2. Type a name for the repository, and an optional description.



3. Choose to make the repository public or private.



4. Check the box **Initialize this repository with a README**, then click **Create repository**. Optionally, select a license file and gitignore.

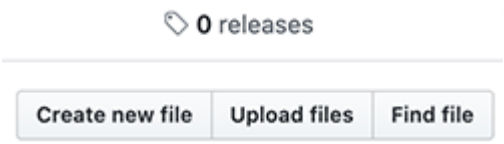


Making changes through Github web interface

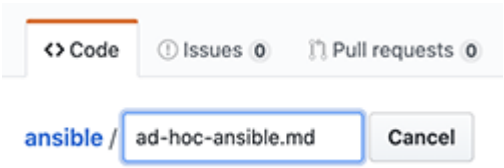
One way to interact with your Git repository is to use GitHub's web interface. When you are using the web interface, changes are made directly on the server. It's the quickest way to make changes to your repository.

Use the Github web interface to create a new file in your repository.

1. Click **Create new file**



2. Name the file `ad-hoc-ansible.md`



3. Click inside the **Edit new file** box
4. Add a Markdown code block with an ansible command

```
```
ansible localhost -m ping
```
```

5. Scroll down to **Commit new file**, enter a commit message. Optionally, add an extended description. Then click **Commit new file**.

Pages 42-49 are not included in this preview.

Using Ad-hoc Ansible Commands

An Ansible ad-hoc command uses the command-line tool `ansible` to automate a single task on one or more nodes. An ad-hoc command looks like this:

```
ansible [pattern] -m [module] -a "[module options]"
```

Ad-hoc commands offer a quick and easy way to execute automation, but they are not built for reusability. Their advantages are they leverage Ansible modules which are idempotent and the knowledge gained by using them directly ports over to the Ansible playbook language.

The Importance of Idempotence

Why not script it? Why not just write your own automation using PowerShell, Bash, or Python? The short answer is idempotence.

When you use an imperative language to deploy or configure your infrastructure, you are writing a prescriptive order of operations. This order does not account for all the various scenarios that might be true.

For example, take the task of creating a directory. It is accomplished by running a single command. But what happens when you run the command again? You get an error stating that the directory exists.

That's easy enough to address. Add a try-catch block to account for this scenario. However, what if you want to modify the permissions? Each new situation requires that you reevaluate the automation and add additional logic. What you want is a way to declare the end state and have the automation handle the rest.

This is exactly what Ansible does, it use a declarative language that allows you to define what should be created and figures the rest.

When you use Ansible to create a directory, you declare the configuration and Ansible figures how to make it happen. If the file exists, it returns a status of SUCCESS because the logic exists to handle that scenario. If the permissions need to change, then you update the mode parameter. You shift from writing code for every task and every scenario to declaring every task.

Ansible is able to do this by using automation that is idempotent. When automation is idempotent, it can be repeated an arbitrary number of times and the results will be the same. Idempotency is at the core of how to make your infrastructure repeatable with Ansible.

An Example

The best way to understand the difference between imperative and declarative is to look at an example.

1. Start an Ansible container

```
docker run -it --rm --volume "$(pwd):/ansible" \
--workdir /ansible ansible
```

2. Create the `/ansible` directory with bash.

```
mkdir /ansible
```

IMPORTANT	mkdir: cannot create directory '/ansible': File exists

`mkdir` errored because the directory already existed.

If the command is placed in a script, additional error handling is required to account for the already existing directory.

3. Create the `/ansible` directory with Ansible.

```
ansible localhost -m file -a "path=/ansible state=directory"
```

ansible output

```
localhost | SUCCESS => {
    "changed": false,
    "gid": 0,
    "group": "root",
    "mode": "0755",
    "owner": "root",
    "path": "/ansible",
    "size": 192,
    "state": "directory",
    "uid": 0
}
```

In its simplest form, this is the difference between imperative and declarative. Declarative automation gives you the ability to run something over and over again and expect the same result.

Can additional error handling be added to the bash script in order to account for this scenario? Yes, but that's just more code that you have to write.

When are ad-hoc commands useful

Ad-hoc commands are great for infrequent tasks. For example, if you need to restart a service across several hosts in response to an incident, you can quickly accomplish this with a one-liner.

Use Cases for Ad-hoc Tasks:

- Rebooting servers
- Managing services
- Managing files
- Fact gathering
- Connectivity testing

TIP	*If you have a lot of experience with a scripting language, you are going to be tempted to use the hammer you know well. As often as you can, exercise ad-hoc Ansible commands to improve your competency with Ansible.*

Using ad-hoc Commands

Ad-hoc commands demonstrate the simplicity and capability of Ansible. The concepts and knowledge that you'll learn directly port over to the Ansible playbook language. The `ansible` command requires a: pattern, module name and an arguments list if required by the module. Change to the `becomeansible/chapter-05` directory and start an Ansible container.

Docker command to start Ansible container

```
docker run -it --rm --volume "$(pwd):/ansible" --workdir /ansible ansible
```

Use the host pattern of `localhost` to run the following ad-hoc commands.

Test connectivity

```
ansible localhost -m ping
```

- `localhost` - Host Pattern
- `-m ping` - Specifies the ping module

  The `ping` module tests connectivity. It does not use ICMP and requires Python on the remote-node. It does not require any additional arguments.

Pages 52-57 are not included in this preview.

What is Repeated is a Playbook

Playbooks are the basis of configuration management and orchestrated automation with Ansible. They offer another way to use Ansible and are specifically designed for automating repeatable tasks.

Playbooks are expressed in YAML and have a minimum of amount of syntax. It is not a programming language or script. It is instead a model of a configuration or a process. Because playbooks define a process and/or configuration, they are kept in source control.

Each playbook is composed of one or more plays. A play starts by defining the `hosts` line. The `hosts` line is a list of one or more groups or host patterns, separated by colons.

```
---
- hosts: localhost
```

Optionally, you can define the user to run the play as.

```
---
- hosts: localhost
  remote_user: root
```

The `hosts` line is followed by a `tasks` list. Ansible executes tasks in order, one at a time, against all the machines matched by the pattern in the `hosts` line.

```
---
- hosts: localhost ①
  tasks: ②
  - name: create a directory ③
    file:
      path: /ansible/playbooks
      state: directory
```

① `hosts` line

② `tasks` list

③ `task`

Ansible executes each task on all hosts matched by the host pattern, before moving on. If a task fails to run on a host, it is taken out of the rotation of the entire playbook.

A playbook is not limited to a single play. Multiple plays can be defined within a single playbook. Keep in mind that a play maps tasks to a host or group of hosts.

pseudo playbook multiple plays

```
---
- hosts: webservers ①
  tasks:
  - name: deploy code to webservers
    deployment:
      path: {{ filepath }}
      state: present

- hosts: dbserver ②
  tasks:
  - name: update database schema
    updatedbschema:
      host: {{ dbhost }}
      state: present

- hosts: webservers ③
  tasks:
  - name: check app status page
    deployment:
      statuspathurl: {{ url }}
      state: present
```

① Play One - deploy code to webservers

② Play Two - update database schema

③ Play Three - check app status page

The modules here are used for demonstraton, they are not real Ansible modules.

Pages 59-72 are not included in this preview.

Take Inventory of the Infrastructure

You already have a way of inventorying your infrastructure. It might be mentally by remembering hostnames or by grouping them by a common naming convention. It might be a custom database that stores specific properties about the hosts. Or, it could be a full-blown CMDB (configuration management database) that has entries for all the hosts. Ansible codifies this information by using an inventory.

Inventories describe your infrastructure by listing hosts and groups of hosts. Using an inventory gives you the ability to share common variables among groups. Inventories are also parsable with host patterns, giving you the ability to broadly or narrowly target your infrastructure with Ansible automation. In its simplest form, an Ansible inventory is a file.

For example:

Within your infrastructure, you have three web servers and a database server. The Ansible inventory might look like this.

ini inventory file

```
[webservers]
web01
web02
web03

[dbservers]
db01
```

Each host or group can be used to define a variable(s) or used for host pattern matching.

host variable

```
db01 database_name=customerdb
```

Creates a host variable named `database_name` with the value of `customerdb.` This variable is scoped to the host `db01.`

group variable

```
[webserver:vars]
http_port=8080
```

Creates a group variable named `http_port` with the value of `8080,` and it is shared among all three web servers.

host pattern

```
ansible webserver -i hosts -m ping
```

Uses the host pattern `webserver` to target all four web servers with the `ping` command.

Create an Inventory

Converting the ad-hoc ping command to a playbook makes the task more repeatable. However, there are still some points of friction.

One is code duplication. Each playbook contains the same set of variables used to connect to a Linux host. Two is you are still required to pass the hostname or IP address to the `-i` option when running the playbook. Both of these friction points are eliminated by using an inventory.

```
vars:
  ansible_user: ansible
  ansible_password: <Password>
  ansible_ssh_common_args: '-o StrictHostKeyChecking=no'
```

```
ansible-playbook ping.yml -i <Public Ip Address>,
```

Use an Ansible INI file as inventory source

Create an Ansible INI inventory for your Ansible development environment.

1. Create a file named `hosts`, no file extension required
2. Obtain the DNS name of the Linux virtual machine or EC2 instance

   IP address can also be used.

3. Add the DNS name to the `hosts` file

   hosts

   ```
   #azure
   vm-linuxweb001.eastus.cloudapp.azure.com
   #aws
   ec2-50-16-43-189.compute-1.amazonaws.com
   ```

Pages 74-90 are not included in this preview.

Building Reusable Configurations

Ansible's promise is configuration management. It's how you move from ad-hoc and manual configuration to a consistent, reliable, and automated process. In the context of infrastructure automation, there are two types of change that need management: state change and configuration change.

One example of a state change is restarting a service. It has an impact on the system, but it does not modify the system's configuration. Configuration changes are the opposite. They make lasting modifications to the system. In some cases, they have no impact. Configuration management is how you control changes that are being introduced, verified, and if necessary corrected. It is a systems engineering process for establishing and maintaining the consistency of your infrastructure through automation.

Implementing configuration management requires a lot of changes—it isn't simply writing some playbooks and calling it a day. In order to successfully adopt Ansible, you will need to think about much more than writing automation. You'll need to consider your organization's design and the process by which the automation will run.

Configuration changes flow through a system in response to three events: a new machine has been deployed, a configuration modification has been requested, or validation of the current configuration is required. Your configuration management process should address all three of these events with a single workflow. Ansible has several features and design patterns to support this single workflow. Decoupling the Ansible code into roles, dynamic inventory sources, and the ability to link playbooks together in a predetermined sequence are just a few of those.

Beyond the syntax and design patterns, you'll also need a process for managing the Ansible code itself. Interestingly enough, as you start to define your infrastructure as code the difference between infrastructure code and application code diminishes. Because of this, there is a lot of value in releasing your infrastructure code in a release pipeline model, as you would application code.

Finding the right answers is often more about knowing the right questions to ask than it is having a prescribed recipe for success. As you seek the right questions to ask, it's helpful to have an example to reference. When you begin to implement configuration management, remember: it is a system. And a system involves processes, people, and automation.

> You do not rise to the level of your goals. You fall to the level of your systems.
>
> — James Clear

Build a Site

Your environment consists of one IIS and one Nginx web server. As changes flow through your environment you will be required to run one of the two playbooks: `configure_iis_web_server.yml` or `configure_nginx_web_server.yml`.

Following a change through this environment, you would run one of these two playbooks in response to one of those three events mentioned earlier: a new machine being deployed, a configuration change is needed, or the need to validate the current configuration. Before you make the change you have to decide which playbook to run. Making this decision is problematic: it requires manual intervention, and it doesn't take advantage of Ansible's idempotency.

Another option is to combine the two playbooks, running them both any time there is a change. What makes this operation safe is that the tasks within the playbooks are themselves idempotent. Changes will only be made if they are new or if something needs to be corrected, thereby putting the machine back in the desired state. Luckily, there is a better option than copying the tasks into a single monolithic playbook. That better option is to use a site playbook.

Site is a term used in Ansible's documentation to define the holistic configuration of an application, service, or environment. It is a playbook that determines the configuration by listing the sequence of Ansible automation to run. In this scenario you will use a `site.yml` playbook to import the existing playbooks, allowing you to run both at once.

Create a site.yml playbook to configure the web servers.

```
touch site.yml
```

site.yml

```
- name: '[Linux] configure nginx'
  import_playbook: configure_nginx_web_server.yml

- name: '[Windows] configure IIS'
  import_playbook: configure_iis_web_server.yml
```

Ansible offers two methods for creating reusable playbooks, imports, and includes. Import is used when you want to reuse existing playbooks. Import allows you to link playbooks together but also keep them useable independently. Import also preprocesses the statements at the time the playbooks are parsed.

To import playbooks, use `import_playbook`. Naming the import is optional, but encouraged and useful.

Run the `site.yml` playbook

```
ansible-playbook site.yml -i hosts --ask-vault-pass
```

Pages 93-106 are not included in this preview.

Dynamic Inventories for Scaling

Infrastructure is rarely static. Existing infrastructure is scaled out or up. New infrastructure is deployed to support new applications or services. And sometimes new infrastructure just shows up. You don't always know what it's for, but you manage it anyhow.

Each of those events requires modifications to the Ansible inventory. Adding hosts to the inventory requires you to create a new entry in the `hosts` file and to assign it to the appropriate groups. While doable, this can be cumbersome. Ansible solves the problem of inventory management with dynamic inventories.

A dynamic inventory uses an external source to generate the Ansible inventory. That external source can be your cloud provider, an IP address management (IPAM) and data center infrastructure management (DCIM) tool such as Netbox, or even a custom script that generates JSON that Ansible can parse as an inventory.

Ansible provides a few ways for you to use dynamic inventories, but the recommended approach is to use an inventory plugin. For a full list, check out Ansible's plugin list. There are several inventory plugins available; however, this book only covers two: `aws_ec2` for AWS and `azure_rm` for Azure.

At a high level, both plugins query the cloud provider for a list of instances or virtual machines. The results are then used to populate the hosts portion of an inventory. Group creation and membership are also dynamically created. The `aws_ec2` plugin uses a `groups` or `keyed_groups` parameter, while the `azure_rm` plugin uses `conditional_groups` and/or `keyed_groups`. Each of these plugins, regardless of the parameter name, use hostvars to determine group membership.

```
Options
- AWS - Create a Dynamic Inventory
- [Azure - Create a Dynamic Inventory]
```

AWS - Create a Dynamic Inventory

Ansible has a built-in inventory plugin for AWS called `aws_ec2`. The plugin queries AWS for ec2 instance details and constructs an Ansible inventory from that information. The inventory's hosts are populated by the ec2 instances that are returned. Group and group memberships are determined by the host variables that are assigned to each host.

Create a file named `hosts_aws_ec2.yml` and define the `aws_ec2` settings.

hosts_aws_ec2.yml

```
plugin: aws_ec2
regions:
  - us-east-1
filters:
  tag:app: ansible
```

The file starts by defining the inventory plugin, `aws_ec2`. Next, is the regions parameter that lists all the regions for the inventory. All ec2 instances within the listed regions will then be added to the inventory.

The filter `tag:app: ansible` limits the inventory to only ec2 instances that match the tag specified. Without this, it would include all ec2 instances in the `us-east-1` region.

Confirm the inventory works by running the `ansible-inventory` command.

```
ansible-inventory -i hosts_aws_ec2.yml --graph
```

ansible-inventory output

```
@all:
  |--@aws_ec2:
  |  |--ec2-3-231-170-254.compute-1.amazonaws.com
  |  |--ec2-54-89-136-132.compute-1.amazonaws.com
  |--@ungrouped:
```

Viewing the output, you'll notice that the `all` group contains two nested groups: `aws_ec2` and `ungrouped`. Both of the ec2 instances are listed under the `aws_ec2` group.

Run the `site.yml` playbook using the dynamic inventory

```
ansible-playbook site.yml -i hosts_azure_rm.yml --ask-vault-pass
```

Pages 109-118 are not included in this preview.

Build a CI \ CD pipeline for Ansible with Github Actions

Ansible lifts a huge operational burden. So much so, in fact, that the friction of build and release go unnoticed. In contrast to manual configuration, entering a few keystrokes to run Ansible is a godsend. With frequency, however, that novelty diminishes. And your next constraint becomes velocity.

To achieve a higher velocity, human interaction has to be removed. However, velocity isn't just about speed. It does little good to go fast, if the system is unreliable or unsafe. So, automated testing also becomes part of the process. Your answer is well-defined by a release pipeline model. The release pipeline has four stages: source, build, testing, and release.

You already have the first stage in place. Storing your Ansible code in a git repository checks that box. Using source control provides a single path for changes. It answers the following questions. Who changed the environment? What did they change? When did the change occur?

The next stage is build. Build runs scripts when changes are checked in. It's within this stage that you define tasks that help catch problems at the earliest stage, as well as set up notifications for problems in the pipeline.

Testing is another step in the build process. Build and testing are both a part of continuous integration (CI). Broadly speaking, there are three types of tests: lint, unit, and integration.

Linting verifies syntax and enforces coding standards. Unit tests validate that the code's logic is sound, while integration tests validate expectations by running against real components.

The release is the final stage. It takes the run button away. The release is responsible for deploying code to production. And all the other environments you have to maintain. I hope it's not too many.

Start simple. Start small. It takes time to mature this process.

> **NOTE** Read more about the Release Pipeline Model here.

Release Pipeline for Ansible

You will be using Github Actions to automate the build, test, and release stages of the release pipeline. The idea is to have all the stages run after a commit or pull request is made to the master branch. What makes up each stage?

Build

Ansible isn't a compiled language, but it does have environment requirements. Because the code will be running on a hosted agent provided by Github, Ansible won't be installed. You will create an action that uses a Docker container to setup the build environment.

Test

You will start testing by using a linter. Ansible has a command-line utility called `ansible-lint` that is

used for analyzing your code and flagging issues based on rules. After linting, other types of tests could be added to this stage. However, to start linting will be the only step in the test stage.

Release

The work that you did in Chapter 8 has set you up for an easy release stage. All you need to do is run the `site.yml` playbook, and Ansible will take care of the rest. However, there are a few differences when running Ansible in the release pipeline. You will figure out how to populate environment variables that connect Ansible to your cloud provider. In addition, you will learn how to pass the vault password into the `ansible-playbook` command.

Linting Ansible Code

Ansible has a command-line tool for linting Ansible playbooks. Linter tools analyze source code and flag programming errors, bugs, stylistic errors, and suspicious constructs.

Install ansible-lint

```
pip3 install ansible-lint
```

Run `ansible-lint` against `configure_iis_web_server.yml`

```
ansible-lint configure_iis_web_server.yml
```

ansible-lint output

```
[201] Trailing whitespace ①
configure_iis_web_server.yml:9 ②
```

① Rule number and name

② Playbook and line number of flagged code

Whitespace is a minor offense. It doesn't impact the execution of the playbook. It's just best practice.

However, if you commit this and the release pipeline runs, it will fail the build. You have to either correct the error or ignore the rule. Delete the trailing whitespace(s), if they exist. And run `ansible-lint` again.

Delete the trailing whitespace(s), if they exists. And run the `ansible-lint` again.

Ignoring Lint Rules

Some rules are easier to follow than others. Deleting whitespace(s) is an easy correction. Other rules, however, might require development hours that you can't immediately sink into the code base. In these instances, you can choose to ignore the rule.

lint the `configure_nginx_web_server.yml` playbook

```
ansible-lint configure_nginx_web_server.yml
```

ansible-lint output

```
[403] Package installs should not use latest
configure_nginx_web_server.yml:8
Task/Handler: install nginx
```

Ansible lint flags the usage of using the latest. It suggests to not use the latest, but instead use a specific version. This would lock the version of Nginx. Version locking it would prevent unintended upgrades, and it is good work to be done. However, sometimes you just need to "ship" it.

Use the `-x` parameter to ignore rule 403.

```
ansible-lint configure_nginx_web_server.yml -x 403
```

No flags are thrown by `ansible-lint` now that rule 403 is being ignored.

Ignoring rules using the `-x` parameter is problematic when you're trying to automate it in a pipeline. Instead of updating commandline arguments, you can create a lint configuration.

Create a lint configuration at the root of your Ansible repository.

```
touch .ansible-lint
```

Add rule 403 to the skip list

ansible-lint

```
skip_list:
  - '403'
```

Run ansible lint without `-x`

```
ansible-lint configure_nginx_web_server.yml
```

Ignoring rules entirely has some drawbacks. It ignores all offenses of the rule, not just the ones that you don't care about. Another option is to add comments in the playbook to ignore specific tasks.

Pages 122-137 are not included in this preview.

Appendix A: Ansible Directory Layout

```
When to Create a Repository

- Per Team
    Examples: SRE, Infrastructure, Network
- Per Application and or Platform
    Examples: Redis, Navigation, Database

Reason being, they have different workflows.
```

Basic

```
|-- dev.yml
|-- production.yml

|-- group_vars/
|   |-- all.yml
|   |-- linux.yml
|   |-- windows.yml
|-- host_vars/
|   |-- linuxweb01.yml
|   |-- winweb01.yml

|-- ping.yml
|-- win_ping.yml
|-- configure_nginx_web_server.yml
|-- configure_iis_web_server.yml

|-- roles/
|   |-- chocolatey/
|   |-- ngnix/
|   |-- iis/
```

Alternate

```
|-- inventories/
|   |
|   |-- dev/
|   |   |-- hosts
|   |   |-- group_vars/
|   |   |   |-- all.yml
|   |   |   |-- linux.yml
|   |   |   |-- windows.yml
|   |   |-- host_vars/
|   |   |   |-- linuxweb01.yml
|   |   |   |-- winweb01.yml
|   |
|   |-- production/
|   |   |-- hosts
|   |   |-- group_vars/
|   |   |   |-- all.yml
|   |   |   |-- linux.yml
|   |   |   |-- windows.yml
|   |   |-- host_vars/
|   |   |   |-- linuxweb01.yml
|   |   |   |-- winweb01.yml

|-- ping.yml
|-- win_ping.yml
|-- configure_nginx_web_server.yml
|-- configure_iis_web_server.yml

|-- roles/
|   |-- chocolatey/
|   |-- ngnix/
|   |-- iis/
```

Multistage

```
|-- environments/
|   |
|   |-- 000_cross_env_vars ①
|   |
|   |-- dev/
|   |   |-- hosts
|   |   |-- group_vars/
|   |   |   |-- all/
|   |   |   |  |-- 000_cross_env_vars -> ../../../000_cross_env_vars
|   |   |   |  |-- env_specific ②
|   |   |   |-- linux.yml
|   |   |   |-- windows.yml
|   |   |-- host_vars/
|   |   |   |-- linuxweb01.yml
|   |   |   |-- winweb01.yml
|   |
|   |-- production/
|   |   |-- hosts
|   |   |-- group_vars/
|   |   |   |-- all/
|   |   |   |  |-- 000_cross_env_vars -> ../../../000_cross_env_vars
|   |   |   |  |-- env_specific
|   |   |   |-- linux.yml
|   |   |   |-- windows.yml
|   |   |-- host_vars/
|   |   |   |-- linuxweb01.yml
|   |   |   |-- winweb01.yml

|-- ping.yml
|-- win_ping.yml
|-- configure_nginx_web_server.yml
|-- configure_iis_web_server.yml

|-- roles/
|   |-- chocolatey/
|   |-- ngnix/
|   |-- iis/
```

① symbolic link, shares variable across all environments

② env specific variables

Both `000_cross_env_vars` and `env_specific` are provided to the hosts via the `all` group.

Recommended Reading

- How to Manage Multistage Environments with Ansible
- Ansible and Ansible Tower: best practices from the field
- Ansible Best Practices - Content Organization