Aric Renzo

Containerization with Ansible 2

Implement container management, deployment, and orchestration within the Ansible ecosystem

Packt

Containerization with Ansible 2

Implement container management, deployment, and orchestration within the Ansible ecosystem

Aric Renzo

BIRMINGHAM - MUMBAI

Containerization with Ansible 2

First published: November 2017

Production reference: 1051217

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78829-191-0

www.packtpub.com

Credits

Author Aric Renzo	**Copy Editor** Safis Editing
Reviewer Michael Bright	**Project Coordinator** Judie Jose
Commissioning Editor Vijin Boricha	**Proofreader** Safis Editing
Acquisition Editor Heramb Bhavsar	**Indexer** Tejal Daruwale Soni
Content Development Editor Devika Battike	**Graphics** Tania Dutta
Technical Editor Prachi Sawant	**Production Coordinator** Melwyn Dsa

About the Author

Aric Renzo is a DevOps engineer based in Charlotte, North Carolina, and is a fan of all things geeky and open source. He has experience working on many open source and free software project deployments for clients based on OpenStack, Ansible, Docker, Chef, SaltStack, and Kubernetes. Aric is a member of the Ansible community and teaches courses on basic and advanced Ansible concepts. His past projects include work on data center deployments, network infrastructure automation, MongoDB NoSQL database architecture, and designing highly available OpenStack environments. Aric is a fan of anything to do with DevOps, automation, and making his workflow more efficient.

Aric is a lifelong geek and a graduate of Penn State University in the information sciences and technology program. He is married to Ashley Renzo, an incredibly beautiful and talented science teacher in Gaston County, North Carolina.

Dedicated to the love of my life, Ashley Renzo; without her unending love and encouragement, this book would never have been written. Also to my dearest friends and family who have prayed for me, advised me, and shared their amazing wisdom with me throughout this project. I am so blessed to have these amazing people in my life.

About the Reviewer

Michael Bright, RHCE/RHCSA, is a solution architect working in the HPE EMEA Customer Innovation Center. He has strong experience across cloud and container technologies (Docker, Kubernetes, AWS, GCP, Azure), as well as NFV/SDN. Based in Grenoble, France, he runs a Python user group and is a co-organizer of the Docker and FOSS Meetup groups. He has a keen interest in serverless, container, orchestration, and unikernel technologies, on which he has presented and run training tutorials at several conferences. He has presented many a time on subjects as diverse as NFV, Docker, container orchestration, serverless, unikernels, Jupyter Notebooks, MongoDB, and Tmux. Michael has a wealth of experience across pure research, R&D, and presales consulting roles. The books that he has worked on are *CoreOS in Action*, *Manning* and *Kubernetes in Action*, *Manning*.

www.PacktPub.com

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://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

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

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at https://www.amazon.com/dp/1788291913.

If you'd like to join our team of regular reviewers, you can e-mail us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

Over the last few years, the world of IT has seen radical shifts in the ways in which software applications are developed and deployed. The rise of automation, cloud computing, and virtualization has fundamentally shifted how system administrators, software developers, and organizations as a whole view and manage infrastructure. Just a few years ago, it would seem unthinkable to many in the IT industry to allow mission-critical applications to be run outside the walls of the corporate data center. However, now there are more organizations than ever migrating infrastructure to cloud services such as AWS, Azure, and Google Compute in an effort to save time and cut back on overhead costs related to running physical infrastructure. By abstracting away the hardware, companies can focus on what really matters—the software applications that serve their users.

The next great tidal wave within the IT field formally started in 2013 with the initial release of the Docker container engine. Docker allowed users to easily package software into small, reusable execution environments known as containers, leveraging features in the Linux kernel for use with LXC (Linux Containers). Using Docker, developers can create microservice applications that can be built quickly, are guaranteed to run in any environment, and leverage reusable service artifacts (container images) that can be version controlled. As more and more users adopted containerized workflows, gaps in execution began to appear. While Docker was great at building and running containers, it struggled to be a true end-to-end solution across the entire container life cycle.

The Ansible Container project was developed to bring the power of the Ansible configuration management and automation platform to the world of containers. Ansible Container bridges the container life cycle management gap by allowing container build and deploy pipelines to speak the Ansible language. Using Ansible Container, you can leverage the powerful Ansible configuration management language to not only build containers, but deploy full-scale applications on remote servers and cloud platforms.

This book will serve as a guide to working with the Ansible Container project. It

is my goal that by the end of this book, you will have a firm understanding of how Ansible Container works, and how to leverage its many capabilities to build robust containerized software stacks from development all the way to production.

What this book covers

Chapter 1, *Building Containers with Docker*, introduces the reader to what Docker is, how it works, and the basics of using Dockerfiles and Docker Compose. This chapter lays down the foundational concepts needed to start learning how to use Ansible Container.

Chapter 2, *Working with Ansible Container*, explores the Ansible Container workflow. This chapter gives the reader familiarity with the core Ansible Container concepts such as build, run, and destroy.

Chapter 3, *Your First Ansible Container Project*, gives the user experience in building a simple Ansible Container project by leveraging a community role available on Ansible Galaxy. By the end of this chapter, the reader will be familiar with building projects and pushing container artifacts to container image repositories such as Docker Hub.

Chapter 4, *What's in a Role?*, gives the user an overview of how to write custom container-enabled roles for use with Ansible Container. The overarching goal of this chapter is to write a role that builds a fully functional MariaDB container image from scratch. By the end of this chapter, the user should have basic familiarity with writing Ansible playbooks using proper style and syntax.

Chapter 5, *Containers at Scale with Kubernetes*, gives the reader an overview of the Kubernetes platform and core functionality. In this chapter, the reader will have the opportunity to create a multi-node Kubernetes cluster in the Google Cloud and run containers inside it.

Chapter 6, *Managing Containers with OpenShift*, introduces the reader to Redhat's OpenShift platform. This chapter gives the reader the steps required to deploy a local OpenShift cluster using Minishift and run containerized workloads on it. This chapter also looks at the key differences between Kubernetes and OpenShift, even though the architectures are fundamentally similar.

Chapter 7, *Deploying Your First Project* takes an in-depth look at the final command in the Ansible Container workflow—deploy. Using deploy, the reader

will gain first-hand experience of deploying previously built projects to Kubernetes and OpenShift using the Ansible Container as an end-to-end workflow tool.

Chapter 8, *Building and Deploying a Multi-Container Project*, looks at how Ansible Container can be used to build a project that leverages more than one application container. Critical to a full understanding of this topic is an introduction to container networking and configuring containers to access network resources. This chapter will give the reader an opportunity to build and deploy a multi-container project using Django, Gulp, NGINX, and PostgreSQL containers.

Chapter 9, *Going Further with Ansible Container*, gives the reader an idea of the next steps to take after mastering the entire Ansible Container workflow. Topics explored in this section include integrating Ansible Container with CICD tools, and sharing projects on Ansible Galaxy.

What you need for this book

This book assumes a beginner-to-medium level of experience of working with the Linux operating system, deploying applications, and managing servers. This book walks you through the steps required to bring up a fully-functional lab environment on your local laptop to quickly get up and running using a Virtualbox and Vagrant environment. Prior to starting, it would be helpful to have Virtualbox, Vagrant, and the Git command-line client installed and running on your personal computer. To run this environment with full specifications, the following system requirements must be met or exceeded:

- CPU: 2 cores (Intel Core i5 or equivalent)
- Memory: 8 GB RAM
- Disk space: 80 GB

In this book, you will need the following software list:

- VirtualBox 5.1 or higher
- Vagrant 1.9.1 or higher
- A text editor that edits YAML files (GitHub Atom or VIM preferred)

Internet connectivity is required to install the necessary packages.

Who this book is for

This book is designed to assist those currently working as system administrators, DevOps engineers, or technical architects, (or similar roles) to quickly get up and running with the Ansible Container workflow. It is helpful as well if the reader already has a basic understanding of Docker, Ansible, or other related automation platforms prior to reading the book, although not required. It is my hope that a user can get a firm understanding of these basics while reading the book. The end goal is to help readers gain a solid foundation on how Ansible Container can accelerate building, running, testing, and deploying application containers from development to production environments.

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: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows: --- - name: Create User Account user: name: MyUser state: present - name: Install Vim text editor apt: name: vim state: present

Any command-line input or output is written as follows:
ubuntu@node01:/tmp$ ansible-galaxy init MyRole --container-enabled - MyRole was created successfully

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: "Clicking the Next button moves you to the next screen."

Warnings or important notes appear 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 color images for this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/ContainerizationwithAnsible2_ColorImages.pdf.

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 is 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.

Building Containers with Docker

In recent years, the landscape of the IT industry has dramatically shifted. The rise of highly interactive mobile applications, cloud computing, and streaming media has pushed the limits of the existing IT infrastructure. Users who were once happy with web browsing and email are now taking advantage of the highly interactive services that are available and are continually demanding higher bandwidth, reliability, and more features. In the wake of this shift, IT departments and application developers are continually attempting to find ways to keep up with the increased demand to remain relevant to consumers who depend on their services.

As an application developer, infrastructure support specialist, or DevOps engineer, you have no doubt seen the radical shift in how infrastructure is supported and maintained. Gone are the days when a developer could write an application in isolation, deploy it across an enterprise, and hand over the keys to operations folks who may only have had a basic understanding of how the application functioned. Today, the development and operations paradigms are intrinsically interwoven in what most enterprises are calling **DevOps**. In the DevOps mindset, operations and support staff work directly with application developers in order to write applications, as well as infrastructure as code. Leveraging this new mindset allows services to go live that may scale multiple tiers and spread between hundreds of servers, data centers, and cloud providers. Once an organization adopts a DevOps mindset, this creates a cultural shift between the various departments. A new team mentality usually emerges, in which developers and operations staff feel a new sense of camaraderie. Developers are happy to contribute to code that makes application deployments easier, and operations staff are happy with the increased ease of use, scaling, and repeatability that comes with new DevOps-enabled applications.

Even within the world of DevOps, containerization has been actively growing and expanding across organizations as a newer and better way to deploy and maintain applications. Like anything else in the world of information technology,

we need controlled processes around how containers are built, deployed, and scaled across an organization. Ansible Container provides an abstracted and simple-to-implement methodology for building and running containers at scale. Before we start to learn about Ansible and containerization platforms, we must first examine how applications and services were deployed historically.

Before we get started, let's look at the topics we will address in this chapter:

- **A historical overview of the DevOps and IT infrastructure**:
    - Manual deployments
    - An introduction to automation
    - The virtualization of applications
    - The containerization of applications
    - The orchestrating of containerized applications
- **Building your first Docker container**
    - Setting up a lab environment
    - Starting your first Docker container
    - Building your first Docker container
    - Container life cycle management

DevOps and the shifting IT landscape

Let's take a quick look at the evolution of many IT departments, and the response to this radical shift across the industry. Before we delve into learning about containers, it is important to understand the history of deploying applications and services in order to realize which problems containerization addresses, as well as how infrastructure has changed and evolved over the decades.

Manual deployments of monolithic applications

The manual deployment of large monolithic applications is where most application deployments start out, and the state of most infrastructure in the late 1990's and early to mid-2000's. This approach normally goes something like this:

1. An organization decides they want to create a new service or application.
2. The organization commissions a team of developers to write the new service.
3. New servers and networking equipment are racked and stacked to support the new service.
4. The new service is deployed by the operations and engineering teams, who may have little to no understanding of what the new service actually does.

Usually, this approach to deploying an application is characterized by little to no use of automation tools, basic shell or batch scripts, and large complex overheads to maintain the application or deploy upgrades. Culturally, this approach creates information silos in teams, and individuals become responsible for small portions of a complicated overall picture. If a team member is transferred between departments or leaves the organization, havoc can arise when the people who are then responsible for the service are forced to reverse engineer the original thought processes of those who originally developed the application. Documentation may be vague if it exists at all.

An introduction to automation

The next step in the evolution towards a more flexible, DevOps-oriented architecture is the inclusion of an automation platform that allows operation and support engineers to simplify many aspects of deployment and maintenance tasks within an organization. Automation tools are numerous and varied, depending on the extent to which you wish to automate your applications. Some automation tools work only at an OS-level to ensure that the operating system and applications are running as expected. Other automation tools can use interfaces such as IPMI to remotely power-on bare-metal servers in order to deploy everything from the operating system upward.

Automation tools are based around the configuration management concepts of *current state* and *desired state*. The goal of an automation platform is to evaluate the current state of a server against a programmatic template that defines the servers desired state and only applies actions on the server that are required to bring it into the desired state. For example, an automation platform checking for NGINX in a running state may look at an Ubuntu 16.04 server and see that NGINX is not currently installed.

To bring this server into the desired state, it may run the command `apt-get install nginx` on the backend to bring that server into compliance. When that same automation tool is evaluating a CentOS server, it may determine that NGINX is installed but not running. To bring this server into compliance, it would run `systemctl start nginx` to bring that server into compliance. Notice that it did not attempt to re-install NGINX. To expand our example, if the automation tool was examining a server that had NGINX both installed and running, it would take no action on that server, as it is already in the desired state. The key to a good automation platform is that the tool only executes the steps required to bring that server into the desired state. This concept is known as **idempotency**, and is a hallmark of most automation platforms.

We will now look at a handful of open source automation tools and examine how they work and what makes them unique. Having a firm understanding of automation tools and how they work will help you to understand how Ansible

Container works, and why it is an invaluable tool for container orchestration:

- **Chef**: Chef is a configuration management tool written by Adam Jacobs in 2008 to address specific use cases he was tasked with at the time. Chef code is written in a Ruby-based domain-specific language known as *recipes*. A collection of recipes grouped together for a specific purpose is known as a *cookbook*. Cookbooks are stored on a server, from which clients can periodically download updated recipes using the client software running as a daemon. The *Chef Client* is responsible for evaluating the current state against the desired states described in the cookbooks.
- **Puppet**: Puppet was written in 2005 by Luke Kaines and, similar to Chef, works on a client-server model. Puppet manifests are written in a Ruby DSL and stored on a dedicated server known as the *Puppet Master*. Clients run a daemon known as the *Puppet Agent*, which is responsible for downloading Puppet manifests and executing them locally across the clients.
- **Salt**: Salt is a configuration management tool written by Thomas Hatch in 2011. Similar to Puppet and Chef, Salt works primarily on a *client-server* model in which *states* stored on the *Salt Master* are executed on the minions to bring about the desired state. Salt is notable in that it is one of the fastest and most efficient configuration management platforms, as it employs a message bus architecture (ZeroMQ) between the master and nodes. Levering this message bus, it is quickly able to evaluate these messages and take the corresponding action.
- **Ansible**: Ansible is perhaps one of the more unique automation platforms of the ones we have looked at thus far. Ansible was written in 2012 by Michael DeHaan to provide a minimal, yet powerful configuration management tool. Ansible *playbooks* are simple YAML files that detail the actions and parameters that will be executed on target hosts in a very readable format. By default, Ansible is agentless and leverages a *push* model, in which playbooks are executed from a centralized location (your laptop, or a dedicated host on the network), and evaluated on a target host over SSH. The only requirements to deploy Ansible are that the hosts you are running playbooks against need to be accessible over SSH, and they must have the correct version of Python installed (2.7 at the time of writing). If these requirements are satisfied, Ansible is an incredibly powerful tool that requires very little effort in terms of knowledge and

resources to get started using it. More recently, Ansible launched the Ansible Container project, with the purpose of bringing configuration management paradigms to building and deploying container-based platforms. Ansible is an incredibly flexible and reliable platform for configuration management with a large and healthy open source ecosystem.

So far, we have seen how introducing automation into our infrastructure can help bring us one step closer to realizing the goals of DevOps. With a solid automation platform in place, and the correct workflows to introduce change, we can leverage these tools to truly have control over our infrastructure. While the benefits of automation are great indeed, there are major drawbacks. Incorrectly implemented automation introduces a point of failure into our infrastructure. Before selecting an automation platform, one must consider what will happen in the event that our master server goes down (applicable to tools such as Salt, Chef, and Puppet). Or what will happen if a state, recipe, playbook, or manifest fails to execute on one of your bare metal infrastructure servers. Using configuration management and automation tools is essentially a requirement in today's landscape, and ways to deploy applications which actually simplify and sometimes negate these potential issues are emerging.

Virtualization of applications and infrastructure

With the rise of cloud computing in recent years, the virtualization of applications and infrastructure has for many organizations replaced traditional in-house deployments of applications and services. Currently, it is proving to be more cost-effective for individuals and companies to rent hardware resources from companies such as Amazon, Microsoft, and Google and spin up virtual instances of servers with exactly the hardware profiles required to run their services.

Many configuration management and automation tools today are adding direct API access to these cloud providers to extend the flexibility of your infrastructure. Using Ansible, for example, you can describe exactly the server configuration you require in a playbook, as well as your cloud provider credentials. Executing this playbook will not only spin up your required instances but will also configure them to run your application. What happens if a virtual instance fails? Blow it away and create a new one. With the ushering in of cloud computing, so too comes a new way to look at infrastructure. No longer is a single server or group of servers considered to be *special* and maintained in a specific way. The cloud is introducing DevOps practitioners to the very real concept that infrastructure can be disposable.

Virtualization, however, is not limited to just cloud providers. Many organizations are currently implementing virtualization in-house using platforms such as ESXi, Xen, and KVM. These platforms allow large servers with a lot of storage, RAM, and CPU resources to host multiple virtual machines that use a portion of the host operating system's resources.

Considering the benefits that virtualization and automation bring to the table, there are still many drawbacks to adopting such an architecture. For one, virtualization in all its forms can be quite expensive. The more virtual servers you create in a cloud provider, the more expensive your monthly overhead fee will be, not considering the added cost of large hardware profile virtual

machines. Furthermore, deployments such as these can be quite resourced-intensive. Even with low specifications, spinning up a large number of virtual machines can take large amounts of storage, RAM, and CPU from the hypervisor hardware.

Finally, consideration must also be paid to the maintenance and patching of the virtual machine operating systems, as well as the hypervisor operating system. Even though automation platforms and modern hypervisors allow virtual machines to be quickly spun up and destroyed, patching and updates still must be considered for instances that might be kept for weeks or months. Remember, even though the operating system has been virtualized, it is still prone to security vulnerabilities, patching, and maintenance.

Containerization of applications and infrastructure

Containerization made an entrance on the DevOps scene when Docker was launched in the month of March of 2013. Even though the concepts of containerization predate Docker, for many working in the field, it was their first introduction to the concept of running an application inside a container. Before we go forward, we must first establish what a container is and what it is not.

A container is an isolated process in a Linux system that has control groups and kernel namespaces associated with it. Within a container, there is a very thin operating system layer, which has just enough resources to launch and run other processes. The base operating system layer can be based on any operating system, even a different operating system from the one that is running on the host. When a container is run, the container engine allocates access to the host operating system kernel to run the container in isolation from other processes on the host. From the perspective of the application inside the container, it appears to be the only process on that host, even though that same host could be running multiple versions of that container simultaneously.

The following illustration shows the relationship between the host OS, the container engine, and the containers running on the host:

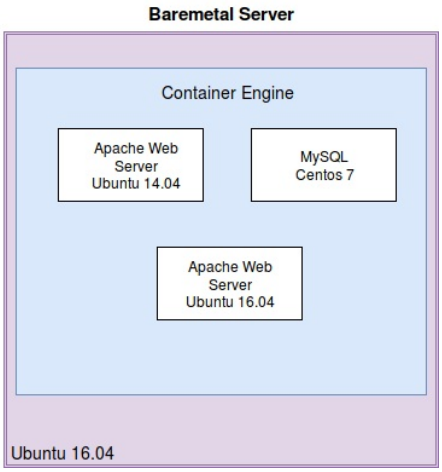


Figure 1: An Ubuntu 16.04 host running multiple containers with different base operating systems

Many beginners at containerization mistake containers for lightweight virtual machines and attempt to fix or modify running containers as you would a VM or a bare metal server that isn't running correctly. Containers are meant to be truly disposable. If a container is not running correctly, they are lightweight enough that one can terminate the existing container and rebuild a new one from scratch in a matter of seconds. If virtual machines and bare metal servers are to be treated as *pets* (cared for, watered, and fed), containers are to be treated as *cattle* (here one minute, deleted and replaced the next minute). I think you get the idea.

This implementation differs significantly from traditional virtualization, in that a container can be built quickly from a container source file and start running on a host OS, similar to any other process or daemon in the Linux kernel. Since containers are isolated and extremely thin, one does not have to be concerned about running any unnecessary processes inside of the container, such as SSH, security tools, or monitoring tools. That container exists for a specific purpose, to run a single application. Container runtime environments, such as Docker, provide the necessary resources so that the container can run successfully and provide an interface to the host's software and hardware resources, such as storage and networking.

By their very nature, containers are designed to be portable. A container using a CentOS base image running the Apache web server can be loaded on a CentOS host, an Ubuntu host, or even a Windows host; they all have the same container runtime environment and run in exactly the same way. The benefits of having this type of modularity are immense. For example, a developer can build a container image for *MyAwesomeApplication 1.0* on his or her laptop, using only a few megabytes of storage and memory, and be confident that the container will run exactly the same in production as it does on their laptop. When it's time to upgrade the *MyAwesomeApplication* to version 2.0, the upgrade path is to simply replace the running container image with the newer container image version, significantly simplifying the upgrade process.

Combining the portability of running containers in a runtime environment such as Docker with automation tools such as Ansible can provide software developers and operations teams with a powerful combination. New software can be deployed faster, run more reliably, and have a lower maintenance overhead. It is this idea that we will explore further in this book.

Orchestrating of containerized applications

Working towards a more flexible, DevOps-oriented infrastructure does not stop with running applications and tools in containers. By their very nature, containers are portable and flexible. As with anything else in the IT industry, the portability and flexibility that containers bring can be built upon to make something even more useful. Kubernetes and Docker Swarm are two container scheduling platforms that make maintaining and deploying containers even easier.

Kubernetes and Docker Swarm can proactively maintain the containers running across the hosts in your cluster, making scaling and upgrading containers very easy. If you want to increase the number of containers running in your cluster, you can simply tell the scheduling API to increase the number of replicas, and the containers will automatically scale in real time across nodes in the cluster.

If you want to upgrade the application version, you can similarly instruct these tools to leverage the new container version, and you can watch the rolling upgrade process happen almost instantly. These tools can even provide networking and DNS services between containers, such that the container network traffic can be abstracted away from the host networking altogether. This is just a taste of what container orchestration and scheduling tools such as Docker Swarm and Kubernetes can do for your containerized infrastructure. However, these will be discussed in much greater detail later in the book.

Building your first docker container

Now that we have covered some introductory information that will serve to bring the reader up to speed on DevOps, configuration management, and containerization, it's time to get our hands dirty and actually build our first Docker container from scratch. This portion of the chapter will walk you through building containers manually and with scripted Dockerfiles. This will provide a foundational knowledge of how the Ansible Container platform works on the backend to automate the building and deployment of container images.

When working with container images, it is important to understand the difference between container *images* and running *instances* of containers. When you build a container using Ansible Container or manually using Dockerfiles, there is a two-part process required to run a container: Building the container image, and running an instance of the container:

- **Building a Container**: The build process involves downloading a base container OS image, and executing the steps outlined in the Dockerfile or Ansible Container playbooks to bring the container into the desired state. The result of the build process is a cached container image that is ready to launch container instances. The `docker pull` command can also be used to download container images from the internet for your local Docker host to cache.
- **Running a Container**: The process of starting a cached container image and running it is known as *running a container*. You can start as many containers you want from a single container image. If you attempt to run a container image that is not already cached on your local Docker host, Docker will attempt to download that container image from the internet.

Instantiating the lab environment

I would encourage you to follow along as we perform these lab exercises. To simplify the process of getting an environment that has the tools covered in this book up-and-running, I have created a Git repository with many example lab scenarios covered throughout this book. We will start off by running through a quick tutorial on how to set up the lab on your local workstation or laptop. To install the lab components, I would suggest using a computer with at least 8 GB of RAM, a virtualization-enabled CPU (Intel Core i5 or equivalent), and a 128 GB or higher hard drive. Linux or macOS are the preferred operating systems for installing the lab, as these tools generally work better on Unix-like operating systems. However, all of these tools also support Windows, but your mileage may vary.

The lab environment will spin up a disposable Ubuntu 16.04 Vagrant VM which comes preloaded with Docker, Ansible Container, and the various tools you will need to successfully become familiar with how Ansible Container works. A text editor geared towards development is also required and will be used to create and edit examples and lab exercises throughout this book. I would suggest using GitHub Atom or Vim, as both editors support syntax highlighting for YAML documents and Dockerfiles. Both GitHub Atom and Vim are available as free and opensource software and are available cross-platform.

Please note, you do not have to install this lab environment in order to learn and understand Ansible Container. It is helpful to follow along and have hands-on experience of working with the technology, but it is not required.

The book should be simple enough to understand without instantiating the lab if you lack the available resources. It should also be noted that you can instantiate your own lab environment on your workstation as well, by installing Ansible, Ansible Container, and Docker. Later in the book, we will cover Kubernetes and OpenShift, so will need those as well for later chapters. These references can be found at the back of the book.

Installing the lab environment:

Below are the steps required to set up the lab environment on your local workstation. Details on installing Vagrant and Virtualbox for your respective platform can be found on the main websites. Try and download similar version numbers to what is listed to ensure maximum compatibility:

1. Download and install Git: `https://git-scm.com/downloads`
2. Download and install VirtualBox (version 5.1): `https://www.virtualbox.org/wiki/Downloads`
3. Download and install Vagrant (version 1.9.1): `https://www.vagrantup.com/docs/installation/`
4. Clone the Ansible Container Lab Git Repository:

```
git clone https://github.com/aric49/ansible_container_lab.git
```

In your Terminal, navigate to the `ansible_container_lab` Git repository and run: `vagrant up` to start the virtual machine: **cd Ansible_Container_Lab**
vagrant up

If Vagrant and VirtualBox are installed and configured correctly, you should start to see the VM launching on your workstation, similar to the following:
user@host:$ vagrant up
Bringing machine 'node01' up with 'virtualbox' provider...
==> node01: Importing base box 'ubuntu/xenial64'...
==> node01: Matching MAC address for NAT networking...
==> node01: Checking if box 'ubuntu/xenial64' is up to date...
==> node01: Setting the name of the VM: AnsibleBook_node01_1496327441174_45550
==> node01: Clearing any previously set network interfaces...
==> node01: Preparing network interfaces based on configuration...
node01: Adapter 1: nat
==> node01: Forwarding ports...
node01: 22 (guest) => 2022 (host) (adapter 1)
==> node01: Running 'pre-boot' VM customizations...
==> node01: Booting VM...

==> node01: Waiting for machine to boot. This may take a few minutes...
node01: SSH address: 127.0.0.1:2022

Once the Vagrant box has successfully booted up, you can execute the command: `vagrant ssh node01` to get access to the VM.

When you are done working in the Vagrant virtual machine, you can use the command: `vagrant destroy -f` to terminate the VM. Destroying the VM should be done when you are finished working with the machine for the day, or when you wish to delete and re-create the VM from scratch, should you need to reset it to the original settings.

Please note: Any work that is not saved in the `/vagrant` directory in the lab VM will be deleted and will be unrecoverable. The `/vagrant` directory is a shared folder between the root of the `lab` directory on your localhost and the Vagrant VM. Save files here if you want to make them available in the future.

Starting your first Docker container

By default, the lab environment begins running with the Docker engine already started and running as a service. If you need to install the Docker engine manually, you can do so on Ubuntu or Debian-based distributions of Linux using: `sudo apt-get install docker.io`. Once Docker is installed and running, you can check the status of running containers by executing `docker ps -a`:
ubuntu@node01:~$ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
ubuntu@node01:~$

We can see in the preceding output that we have column headers, but no actual information. That's because we don't have any container instances running. Let's check how many container images Docker knows about, using the `docker images` command: **ubuntu@node01:~$ docker images**
REPOSITORY TAG IMAGE ID CREATED SIZE

Not much going on there either. That's because we don't have any container images to play around with yet. Let's run our first container, the Docker `hello-world` container, using the `docker run` command:

```
ubuntu@node01:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
78445dd45222: Pull complete
Digest: sha256:c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
Status: Downloaded newer image for hello-world:latest
Hello from Docker!
This message shows that your installation appears to be working correctly.
To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the
executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
to your terminal.
To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash
Share images, automate workflows, and more with a free Docker ID:
https://cloud.docker.com/
For more examples and ideas, visit:
```

https://docs.docker.com/engine/userguide/

The command we executed was: `docker run hello-world`. A lot of things happened when we ran that command. The command `docker run` is the Docker command required to start and run a container within the Docker engine. The container we are running is `hello-world`. If you look through the output, you can see that Docker reports that it is `Unable to find image 'hello-world:latest' locally`. The first step of the Docker run is Docker testing to see if it already has the container image cached locally, so it doesn't have to download and redownload containers that the host is already running. We validated earlier that we currently have no container images in Docker using the `docker images` command, so Docker searched its default registry (Docker Hub) to download the image from the internet. When Docker downloads a container image, it downloads the image one layer at a time and calculates a hash to ensure that the image was pulled correctly and with integrity. You can see from the preceding output that Docker provides the `sha256` digest, so we can be certain that the correct image was downloaded. Since we didn't specify a container version, Docker searched the Docker Hub registry for an image called, `hello-world` and downloaded the latest version. When the container executed, it printed the `Hello From Docker` output, which is the job the container is designed to perform.

You can also use the `docker ps` command without the `-a` flag to show only containers that are currently running, not exited or stopped containers.

Docker containers are built based on layers. Every time you build a Docker image, each command you run to create the image is a layer in the Docker image. When Docker builds or pulls an image, Docker processes each layer individually, ensuring that the entire container image is pulled or built intact. When you begin to build your own Docker images, it is important to remember: the fewer the layers, the smaller the file size, and the more efficient the image will be. Downloading an image with a lot of layers is not ideal for users consuming your service, nor is it convenient for you to quickly upgrade services if your Docker images take a long time to download.

Now that we have downloaded and run our first container image, let's take a look at our list of local Docker images again: **ubuntu@node01:~$ docker images REPOSITORY TAG IMAGE ID CREATED SIZE**

hello-world latest 48b5124b2768 4 months ago 1.84 kB
ubuntu@node01:~$

As you can see, we have the `hello-world` image cached locally. If we reran this container, it would no longer have to pull down the image, unless we specify a higher image version number than what was stored in the local cache. We can now take another look at our `docker ps -a` output: **ubuntu@node01:~$ docker ps -a**
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
b0c4093ab38f hello-world "/hello" 28 minutes ago Exited (0) 28 minutes ago romantic_easley

From the preceding output, you can see that Docker created a new running container with the container ID: `b0c4093ab38f`. It also listed the name of the source image used to spawn this container, the command executed, (in this case: `/hello`), the time it was created, as well as the current status and container name. You can see that this particular container is no longer running, as the status is `Exited (0)`. This particular container is designed in such a way that it performs one single job and quits once that job has finished. The `Exited (0)` status lets the user know that the execution completed successfully. This functions very similarly to a binary executable, such as `cat` or `echo` commands in a Unix-based system. These commands perform a single job and stop once that job has completed. Building this type of container is useful if your purpose is to provide a user with a container that provides an output, such as parsing text, performing calculations, or even executing jobs on the Docker host. As you will see later, you can even pass parameters to the docker `run` command so that we can modify how the applications inside the container run.

Building your first container

Now that we have an understanding of how Docker containers run, as well as how the Docker engine downloads and caches container images, we can start building containers that run services such as web servers and databases. In this lesson, we will build a container from a Dockerfile that will run the Apache web server. We will then expose ports in the Docker engine that will allow us to access the running web service we just instantiated. Let's get started.

Dockerfiles

As we learned previously, Docker containers consist of layers that are essentially stacked on top of each other to form a Docker container image. These layers consist of commands in a plain-text file that the Docker engine will sequentially execute to build a final image. Each line of a Dockerfile represents a layer in the Docker image. The goal of building our Dockerfiles is to keep them as small and concise as possible so that our container images are not larger than necessary. In the `/vagrant` directory of your VM, create a plain-text file called, `Dockerfile`, and open it in the text editor of your choice. We will start with the following lines, which we will explore one by one: **FROM ubuntu:16.04**
RUN apt-get update; apt-get install -y apache2
EXPOSE 80
ENTRYPOINT ["apache2ctl"]
CMD ["-DFOREGROUND"]

Let's take a look at this Dockerfile line-by-line:

- `FROM`: Indicates the base image from which we want our container to be built. In this case, it is the Ubuntu base image, version 16.04. There are multiple base images, and images with applications prebuilt, that you can leverage, available for free on Docker Hub.
- `RUN`: Any commands you want the container to execute during the build process get passed in with the RUN parameter. We are executing `apt-get update` in tandem with `apt-get install`. We are executing both of these commands using the same `RUN` line in order to keep our container layers as small as possible. It is also a good practice to group package management commands in the same `RUN` lines as well. This ensures that `apt-get install` does not get executed without first updating the sources list. It is important to note that, when a Docker image gets rebuilt, it will only execute the lines that have been changed or added.
- `EXPOSE`: The `EXPOSE` line instructs Docker about which ports should be open on the container to accept incoming connections. If a service requires more than one port, they can be listed separately with spaces.
- `ENTRYPOINT`: The `ENTRYPOINT` defines which command you want the container to

run by default when the container launches. In this example, we are starting the `apache2` web server using `apache2ctl`. If you want your container to be persistent, it is important that you run your application in a daemon mode or a background mode that will not immediately throw an `EXIT` signal. Later in the book, we will look at an open source project called, `dumb-init`, which is an init system for running services in containers.

- `CMD`: `CMD` in this example defines the parameters passed into the `ENTRYPOINT` command at runtime. These parameters can be overridden at the time the container is launched by providing additional arguments at the end of your Docker `run` command. All of the commands or arguments you provide in `CMD` are prefixed by `/bin/sh -c`, making it possible to pass in environment variables at runtime. It should also be noted that, depending on how you want the default shell to interpret the application that is being launched inside the container, you can use `ENTRYPOINT` and `CMD` somewhat interchangeably. The online Docker documentation goes into more in-depth details about best practices for using `CMD` versus `ENTRYPOINT`.

Each line within Dockerfile forms a separate layer in the final Docker container image as seen in the following illustration. Usually, developers want to try to make container images as small as possible to minimize disk usage, download, and build time. This is usually accomplished by running multiple commands on the same line in the Dockerfile.

In this example, we are running `apt-get udpate; apt-get install apache2` in order to try and minimize the size of the resulting container image.

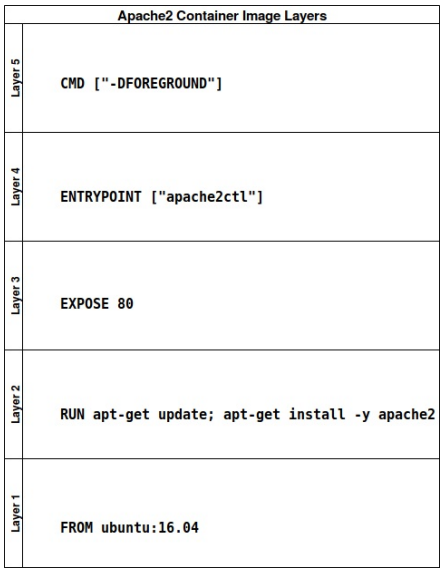


Figure 2: Layers in the Apache2 container image This is by no means an exhaustive list of the commands available for you to use in a Dockerfile. You can export environment variables using `ENV`, copy configuration files and scripts into the container at build time, and even create mount points in the container using the `VOLUME` command. More commands such as these can be found in the official Dockerfile reference guide at https://docs.docker.com/engine/reference/builder/.

Now that we understand what goes into the Dockerfile, let's build in a functional container using the `docker build` command. By default, `docker build` will search in your current directory for a file called `Dockerfile` and will attempt to create a container layer by layer. Execute the following command on your virtual machine: **docker build -t webservercontainer:1.0 .**

It is important to pass in an image build tag using the `-t` flag. In this case, we are tagging the image with the name `webservercontainer` and the version `1.0`. This ensures that you can identify the versions you have built from the `docker image list` output.

If you execute the `docker images` command again, you will see that the newly built image is now stored in the local image cache: **ubuntu@node01:$ docker images**
REPOSITORY TAG IMAGE ID CREATED SIZE
webservercontainer 1.0 3f055adaab20 7 seconds ago 255.1 MB

We can launch new container instances now using `docker run`: **docker run -d --name "ApacheServer1" -p 80:80 webservercontainer:1.0**

This time, we are passing new parameters into `docker run`:

- `-d`: Indicates that we are going to run this container in detached or background mode. Running containers in this mode will not immediately log the user into the container shell upon starting. Rather, the container will start directly in the background.
- `--name`: Gives our container a human-readable name so that we can easily understand what the container's purpose is. If you don't pass in a name flag, Docker will assign a random name to your container.
- `-p`: Allows us to open ports on the host that will be forwarded to the exposed port on the container. In this example, we are forwarding port `80` on the host to port `80` on the container. The syntax for the `-p` flag is always `<HostPort>:<ContainerPort>`.

You can test if this container is running by executing the `curl` command on the VM against localhost on port `80`. If all goes well, you should see the default Ubuntu Apache welcome page: **ubuntu@node01:~$ curl localhost:80**
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
...

This indicates to us that Docker is listening on the localhost on port `80` and forwarding that connection to the container, also listening on port `80`. The great thing about containers is that you can launch multiple instances of the same container, provided they are listening on the different port numbers. In a matter of seconds, you can create a fleet of containers providing various services, and just as quickly wipe them out.

Let's create two more Apache web server containers listening on ports `100` and `200` of the host's networking interfaces. Note that in the following example, I have provided different name parameters as well as different host ports: **docker run -d --name "ApacheServer2" -p 100:80 webservercontainer:1.0**

docker run -d --name "ApacheServer3" -p 200:80 webservercontainer:1.0

If you run the same `curl` command again, this time on port `100` and `200`, you will see the same Ubuntu default web server page. That's boring. Let's give our

containers more personality. We can use the `docker exec` command to log in to running containers and customize them slightly:

```
ubuntu@node01:~$ docker exec -it ApacheServer1 /bin/bash
root@bc951d6ec658:/#
```

The `docker exec` requires the following flags to access a running container:

- `-i`: Run `docker exec` interactively, since we are going to be launching into a Bash shell
- `-t`: Allocate a pseudo-tty, or terminal session
- `ApacheServer1`: The name (or container ID) of the container we want to log into
- `/bin/bash`: The terminal or command we want to launch using the `docker exec` command

Running the `docker exec` command should drop you directly into the Bash shell of the first Apache container. Run the following command to change the `index.html` file in the Docker container. When you've finished, you can exit out of the container's shell session by typing `exit`.

```
root@bc951d6ec658:/# echo "Web Server 1" > /var/www/html/index.html
```

From the Docker host, run the `curl` command again on port `80`. You should see that the page your Apache web server is using has changed:

```
ubuntu@node01:~$ curl localhost:80
Web Server 1
```

Use `docker exec` to log into the other two containers and use `echo` to change the default `index.html` page to something unique to all three web server containers. Your `curl` results should reflect the changes you've made:

```
ubuntu@node01:~$ curl localhost:80
Web Server 1
ubuntu@node01:~$ curl localhost:100
Web Server 2
ubuntu@node01:~$ curl localhost:200
Web Server 3
```

Note: This exercise is for the purposes of demonstrating the `docker exec` command. `docker exec` is not a recommended way to update, fix, or maintain running containers. From a best practices standpoint, you should always rebuild

your Docker containers, incrementing the version tag when changes need to be made. This ensures that changes are always recorded in the Dockerfile so containers can be stood up and torn down as quickly as possible.

You may also have noticed that various Linux operating system tools, text editors, and other utilities are not present in the Docker containers. The goal of containers is to provide the bare-minimal footprint required to run your applications. When building your own Dockerfiles, or later, when we explore Ansible Container environments, think through what is going inside your containers and whether or not your container meets the best practices for designing microservices.

Container life cycle management

Docker gives you the benefit of process isolation using Linux control groups and namespaces. Similar to processes in Unix-like operating systems, these processes can be started, stopped, and restarted to implement changes throughout the lifecycle of the container. Docker gives you direct control of the state of your containers by giving you the options to start, stop, reload, and even view containers logs that might be misbehaving, as needed. Docker gives you the benefit of using either the container's internal ID number or using the container name we assign it when we start using `docker run`. The following is a list of Docker native commands that can be used to manage the lifecycle of a container as you build and iterate through various versions:

- `docker stop <ContainerID or Name>`: Stops the running container and processes within the container.
- `docker start <ContainerID or Name>`: Starts a stopped or exited container.
- `docker reload <ContainerID or Name>`: If the container is running, reload will gracefully stop the container and start the container to bring it back into a running state. If the container is stopped, reload will start the running container.
- `docker logs <ContainerID or Name>`: Displays any logs generated by the container or the application running inside the container leveraging `STDOUT` or `STDERR`. Logs are useful for debugging a misbehaving container without having to `exec` inside the container.

`docker logs` have a `--follow` flag, useful for streaming live log output. This can be accessed using `docker logs --follow <ContainerID or Name>`.

From the preceding example, we can start, stop, reload, or view the logs of any of the Apache web server containers we built earlier, like so:

```
docker stop ApacheServer2
docker start ApacheServer2
docker reload ApacheServer2
docker logs ApacheServer2
```

Similar to this example, you can validate the status of any containers by looking at the output of `docker ps -a`.

For all Docker commands, including `docker run`, `exec`, and `build`, you can see all of the options available for a given command by appending the `--help` flag. For example `docker run --help`.

References

- **Dockerfile reference guide**: https://docs.docker.com/engine/reference/builder/
- **Download Virtualbox**: https://www.virtualbox.org/wiki/Downloads
- **Download Vagrant**: https://www.vagrantup.com/docs/installation/
- **Download Git**: https://git-scm.com/downloads

Summary

In this chapter, we looked at the history of application deployments across IT infrastructure, as well as the history of containers and why they are revolutionizing software development. We also took our first steps in building Docker containers by running containers manually, as well as by building them from scratch through Dockerfiles.

I hope that, if you are new to containerization and Docker, this chapter gave you a good starting point from which you can get hands-on in the world of containerization. Dockerfiles are excellent tools for building containers, as they are lightweight, easily version-controlled, and quite portable. However, they are quite limited in the sense that they are the equivalent of a Bash shell script in the world of DevOps. What happens if you need to tweak configuration files, dynamically configure services based on the states of services, or configure containers based on the environmental conditions they will be deployed into? If you have spent time working on configuration management, you will know that, while shell scripts can do the job, there are much better and easier tools available. Ansible Container is exactly the tool we need in order to apply the power of configuration management to the portability and flexibility that containers bring to our infrastructure. In `Chapter 2`, *Working with Ansible Container*, you will learn about Ansible Container and see first-hand how quickly we can build and deploy containers.

Working with Ansible Container

As we saw in Chapter 1, *Building Containers with Docker*, containerization is changing the way critical IT infrastructure is maintained and deployed. As DevOps methodologies and mindsets evolve across organizations, the lines between development and operations roles are becoming blurred. While tools such as Docker continue to grow and evolve, tools need to be developed to leverage the ever-increasing need to scale and deploy containerized applications.

Ansible is a unique framework for automation, as we saw in Chapter 1, *Building Containers with Docker*, as it relies on an agent-less architecture, bringing servers and virtualized applications into the desired state from a centralized location over the SSH protocol. Compared to the other core automation tools discussed, Ansible brings a different approach from other configuration management tools, such as Chef and Puppet, which rely on agents and centralized servers to store and maintain configuration states.

The Ansible Container project was launched to address the need to bring critical configuration management techniques to the currently manual process of building and deploying Docker container images with the standard Docker toolchain. Currently, Docker and Docker tools are built with an emphasis on deploying containers to Docker native environments using Swarm and Docker Compose. Ansible Container is a wrapper around many of the standard Docker tools, and provides the functionality to deploy your projects to various cloud providers, Kubernetes, and OpenShift. At the time of writing other container orchestration tools such as Docker Swarm and Apache Mesos are not currently supported. If Dockerfiles are akin to shell scripts during the era of monolithic application deployments, then Ansible Container is a solution for bringing automation and repeatability to the container ecosystem. As Ansible Core uses playbooks and SSH as an interface for bringing about desired states, Ansible Container can use your same playbooks and native container APIs to build and deploy containers.

If you or your organization is already using Ansible roles for customized deployments of applications and services, these same roles can be leveraged to

turn these applications and services into containers, helping to streamline your container build pipeline. When making the leap from bare-metal and virtualized deployments, you can be confident that your customized configurations and settings will be preserved when building your containers.

In this chapter we will learn:

- An introduction to Ansible Container and the microservice architecture
- A quick introduction to Docker Compose
- Ansible Container workflow
- Ansible Container quick start

An introduction to Ansible Container and the microservice architecture

While using Ansible Container has a great number of benefits in reusing existing Ansible artifacts, modules, and playbooks, careful consideration has to be given to any changes required in porting over your existing services. Ansible gives you a large amount of freedom in the way you write playbooks and roles to suite the uniqueness of your organization's architecture and resource constraints. A typical web application, for example, may have three distinct layers of functionality: a web server, which provides your end users with a website; a database for storing data; and a cache, providing the web server with commonly accessed data from the database. Depending on the architecture and any resource constraints, these services might be implemented in any number of ways. You may have your web server, caching layer, and database on three separate and distinct clusters of servers. You could opt to deploy the web server and caching layer on the same cluster, and the database on a secondary cluster. Or all three layers might be deployed on the same bare-metal or virtualized server cluster, with a load balancer providing redundancy as necessary. Your infrastructure is a unique snowflake that Ansible gives you the freedom to write and deploy playbook roles in almost any configuration that fits your needs.

Microservice architecture is a term used to describe the independent and modular breakout of application services to distinct and deployable units. In the world of containers, you want each of your containers to conform to the microservice architecture, creating each service as a separate container that can be deployed and scaled independently of the other services. While it is possible to deploy multiple services in the same container, it is generally a bad idea, as each service adds layers to your containers, creating unnecessary overhead when building and deploying new containers.

In the preceding example, each of the core services (web server, cache, and database) will be a separate microservice you want to isolate and encapsulate

into containers. Having the flexibility to dynamically deploy more cache or database containers on demand creates a huge advantage if your web application goes into production and you realize that the projected traffic is much higher than originally anticipated and database queries are becoming a bottleneck. Having a microservice-oriented design to your containers will allow your infrastructure to be simplified, more easily deployed, and more quickly scaled to meet the needs of demanding users.

The key takes-away when thinking about porting existing Ansible roles into Ansible Container projects is to think through how tightly integrated your roles currently are. Ideally, Ansible roles should be able to be standalone, with little to no reliance on other environmental characteristics. Isn't this starting to sound a lot like the containerized microservices we described before? This is what makes Ansible Container a unique platform among other configuration management tools. Ansible primitives are already designed to fit nicely into a containerized ecosystem. Even if you are not currently using Ansible as your configuration management tool, Ansible Container is still a fantastic tool for building, maintaining, and deploying containers, from development all the way through to production.

A quick introduction to Docker Compose

Docker Compose is one of the Docker workflow tools that allow you to easily build and run multiple containers at once. It is important to have a basic understanding of how Docker Compose works before we start working with Ansible Container since a lot of Ansible Container's core functionality is wrapped around Docker Compose.

In the previous chapter, I illustrated an example in which three Apache web server containers were created to demonstrate running multiple containers simultaneously leveraging the same container base image. With Docker Compose, instead of providing three separate docker `run` commands, one can simply provide a `YAML` definition file that describes the containers you want to run, any docker `run` parameters you want the containers to run with (ports, volumes, and so on), and any links or dependencies you want to create for the containers prior to running them. When Docker Compose is executed, it will automatically try to bring up the containers described in the `YAML` file. If the images are not yet cached locally, it will try to download them from the internet or will build the container images if the Dockerfiles are provided. Let's do a quick exercise to get a feel for how Docker Compose works.

If you are not using the provided Vagrant lab environment, as discussed in `Chapter 1`, *Building Containers with Docker,* you will first need to download Docker Compose using the following command. The steps provided assume you have Docker Engine already installed and running on a Linux or macOS machine. Make sure you install Docker Compose with the same version number as the Docker Engine you already have running to ensure maximum compatibility. Execute the following commands to download the Docker Compose executable and copy it to `/usr/local/bin` with `execute` privileges.

```
sudo curl -L https://github.com/docker/compose/releases/download/1.17.0/docker-compose-

sudo chmod +x /usr/local/bin/docker-compose
```

The most up-to-date installation documentation can be found at `htt`

ps://docs.docker.com/compose/install.

By default, Docker Compose looks for a file in your current working directory called `docker-compose.yml`. I have provided a sample `docker-compose.yml` file as an example. On your workstation, create a directory called `docker-compose` and create a blank `docker-compose.yml` file in that directory. Paste in the following content: version: '2' services: Cache_Server: image: memcached:1.4.36 ports: - 11211:11211 volumes: - .:/var/lib/MyVolume

Let's look at this file line by line:

- `version`: This line indicates which version of the Docker Compose API to use. In this case, we are using version 2. At the time of writing, there is also version 3 of the API, which provides some new features. For our purposes, however, we are content to use version 2. The version parameter usually starts a Docker Compose file and has no indentation.
- `services`: The `services` line starts the section of your `docker-compose.yml` file that lists each service container you are going to create. In this particular Docker Compose file, we are going to create a service called `Cache_Server`, which spins up a single `memcached` container. Each service you specify should be indented two spaces under the `services` declarative. It should also be noted that the service names are user-defined and are used to generate the container name. When creating multi-container Docker Compose files, Docker provides simple DNS resolution between containers, based on the service names. More on this in `Chapter 8`, *Building and Deploying Multi-Container Projects.*
- `image`: `image` is used to specify the container image you want your container to be based on. For this example, we are using the official `memcached` image from Docker Hub, specifying version 1.4.36. We could also have used the latest keyword in place of the version number if we had wanted to always have the latest version of the image.
- `ports`: The `ports` parameter indicate which ports on the host you want to be forwarded to the container. In this case, we will forward port `11211` to the exposed container port `11211`. Similar to `docker run`, ports must be specified in the format `host:container`. This is a `YAML` list, so each port must be indented and prefixed with a hyphen (`-`).
- `volumes`: This parameter specifies any directories or storage volumes on the

Docker host you would like to make accessible to the container. This is useful if there is data in the container you may want to back up, export, or otherwise share with the container. This volume mounting merely serves as an example of the syntax. Similar to the `ports` parameter, `volumes` takes a list in the form of `hostDirectory:containerDirectory`.

To start our container using Docker Compose, you simply execute the command `docker-compose up`. This will, by default, start all of the containers in the Docker Compose file one by one, unless container dependencies are specified. Containers started using `docker-compose` will be started in `attached` mode, meaning that the container process will run, taking over the Terminal you are using. Similar to `docker run`, we can supply the `-d` flag to run the containers in `detached` mode, so we can run some validations in the same Terminal:

```
docker-compose up -d
```

You will observe that, similarly to `docker run`, Docker Compose automatically determines that the container image is not present on the Docker host and successfully downloads the image and corresponding layers from the internet.

```
ubuntu@node01:/vagrant/Docker_Compose/test$ docker-compose up -d
Creating network "test_default" with the default driver
Pulling Cache_Server (memcached:1.4.36)...
1.4.36: Pulling from library/memcached
56c7afbcb0f1: Pull complete
49acdc7c75c9: Pull complete
152590a2a704: Pull complete
4dc7b8165378: Pull complete
4cb74c11bcdd: Pull complete
Digest: sha256:a2dfef5700944ec8bb2d2c0d6f5b2819324b1b91647dc09847ce81e7a91e3fe4n
Status: Downloaded newer image for memcached:1.4.36
Creating test_Cache_Server_1 ...
Creating test_Cache_Server_1 ... done
```

Running `docker ps -a` will reveal that Docker Compose was able to successfully create the running container with the properly exposed ports and volume mounts listed in our `docker-compose.yml` file:

```
ubuntu@node01:/vagrant/Docker_Compose/test$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             ST
cacf58b455f3        memcached:1.4.36    "docker-entrypoint.sh"   7 minutes ago       Up
```

We can use `telnet` to ensure the `memcached` application is functioning and forwarded through the host networking. Using `telnet`, we can store and retrieve data from `Memcached` directly:

```
ubuntu@node01:/vagrant/Docker_Compose/test$ telnet localhost 11211
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
STAT active_slabs 0
STAT total_malloced 0
END
```

Running the `stats slabs` command lets us know that `memcached` has been deployed and is functioning as expected.

Now that we have had a brief introduction to Docker and Docker Compose, we have acquired the basic skills needed to start working with Ansible Container.

Ansible Container workflow

Similar to other orchestration and automation tools, Ansible Container contains a set of utilities that constitute a containerized workflow. Using Ansible Container, you can create, build, run, and deploy containers, from development all the way through to production, using the suite of tools included with Ansible Container out of the box. Ansible Core's *batteries-included* methodology carries over to Ansible Container to provide developers and system administrators with a complete containerized workflow solution. The following is an overview of the primary Ansible Container functions and how they correspond to the typical lifecycle of a containerized application:

- `ansible-container init`: Used to initially start an Ansible Container project. `init` builds and creates the directory scaffolding and base files that are required to start an Ansible Container project.
- `ansible-container build`: Similar to what the name suggests, `build` will parse the primary files in your project and attempt to build the containers described. Ansible Container is able to do this by first creating what is known as a `conductor` container. The `conductor` container is a master container that is created during the build phase of your project and contains a running copy of Ansible. Once the other containers launch, the `conductor` container is responsible for running the Ansible roles and playbooks against them to bring the containers into the desired state.
- `ansible-container run`: `run` works in a very similar way to `docker run` in the respect that, when executed, `run` takes the built containers and attempts to run them in the container engine on the host. By default, the `run` command takes into consideration any development options listed in the `container.yml` file, unless the `--production` flag is passed in at runtime.
- `ansible-container destroy`: Stops any running containers and also removes any built image files. This command is useful when testing an end-to-end deployment from scratch.
- `ansible-container push`: This command pushes the container images you built with Ansible Container to a container registry of your choice, such as

Docker Hub, Quay, or GCR. This command is similar to `docker push`.

- `ansible-container deploy`: `deploy` (formerly `ShipIt`) takes your current project and generates a customized Ansible playbook and role to deploy your container to a cloud service provider. At the time of writing, `deploy` supports only OpenShift and Kubernetes. Running this playbook using the `ansible-playbook` command, will deploy your containers to the specified provider.

As you can see, Ansible Container comes prebuilt with an end-to-end lifecycle management system that allows you to manage containers from development through to production. Ansible Container leverages the powerful and customizable Ansible configuration management system to allow containers to be created and deployed similarly to bare-metal or virtual nodes.

All Ansible Container subcommands can be found by running `ansible-container --help`.

Ansible Container quick-start

This portion of the chapter is going to focus on getting started with Ansible Container, initializing a base project, and recreating the `memcached` example from earlier. If you are not following along with the Vagrant lab provided on GitHub, the first step is to install Ansible Container using the `python-pip` package manager. The following steps will install Ansible Container with support for Docker on a Debian-based distribution of Linux: **sudo apt-get update sudo apt-get install python-pip sudo pip install ansible-container docker**

Ansible Container init

You should now have Ansible Container installed and ready to run in your environment. The first command that's required to start a new Ansible Container project is the `ansible-container init` command. After logging in to your vagrant VM, create an empty directory in the `/vagrant` directory and type: **ubuntu@node01:~$ mkdir /vagrant/demo ubuntu@node01:~$ cd /vagrant/demo ubuntu@node01:/vagrant/demo$ ansible-container init Ansible Container initialized.**

It is important to note that the final lab exercise can be found in the official book GitHub repository, in the directory: *`AnsibleContainer/demo`*.

When Ansible Container has successfully created a new project, it will return the response `Ansible Container initialized`.

As discussed previously, `init` creates the basic directory structure and layout required to start building Ansible Container projects. Navigating to that directory and looking at the directory listing will give you an idea of what an Ansible Container project looks like: demo ├── ansible.cfg ├── ansible-requirements.txt ├── container.yml ├── meta.yml └── requirements.yml

Let's look at these files individually to understand their purpose in an Ansible Container project:

- `ansible.cfg`: The primary configuration file for the Ansible engine. Any settings you want the Ansible `conductor` container to leverage will go in this file. If you're familiar with using Ansible for configuration management tasks, you will already have a basic familiarity with the `ansible.cfg` file. For the most part, you can safely leave this file alone, unless there is a specific way Ansible needs to run during the container build process. More information about Ansible configuration options can be found in the Ansible documentation at `https://docs.ansible.com.`
- `ansible-requirements.txt`: The `ansible-requirements.txt` file is used to specify any Python pip dependencies that your playbooks may need to run successfully.

Ansible Engine is built on a series of modules that perform the tasks described in the playbooks. Any additional Python packages that are required to run the Ansible roles are listed in this file.

- `container.yml`: Describes the state of your containers, including base images, exposed ports, and volume mounts. The syntax for `container.yml` is similar to the Docker Compose format, with a few differences we will look at throughout this book.
- `meta.yml`: The `meta.yml` file includes any metadata about your container project, including the name of the author, version information, software licensing details, and tags. This information makes it easy for other users to find your project should you choose to share it on Ansible Galaxy.
- `requirements.yml`: Defines any Ansible Galaxy roles and version information your container project will use. In this file, you can describe the exact roles and role versions your project requires. Ansible Container will download these roles from Ansible Galaxy prior to building your container project. By specifying your roles in the `requirements.yml` file, you can be sure that your projects consistently use the same roles to build the base container images. It is important to keep in mind the distinction between `ansible-requirements.yml` and `requirements.yml`. `requirements.yml` is used to manage the Ansible roles your project depends on, whereas `ansible-requirements.yml` is used to manage the Python pip packages those roles may require.

Now that we have a feel for what an Ansible Container project looks like, we can dive in and start experimenting with creating a simple Ansible Container project. Remember our Docker Compose project we created earlier? Let's use that as a starting point and port this project to Ansible Container by editing the `container.yml` file. In a text editor, open the `container.yml` file. By default `container.yml` comes with a prepopulated structure, which in many ways resembles a Docker Compose file. Your `container.yml` file should resemble the following. To conserve space, I have removed many of the comments and example data: version: "2" settings: conductor_base: centos:7 services: {} registries: {}

Each of these sections has a particular purpose for structuring your Ansible Container project. It is important to understand what each of these `YAML` definitions is used to describe. The comments that come in the file by default show examples of the various settings each of these sections uses. The following is a list of the key sections of the `container.yml` file and how to use these sections

in your Ansible Container project:

- `version`: The `version` section signifies which version of the Docker Compose API to use. As we discussed before, Ansible Container is a wrapper around many of the Docker Compose services. Here, we can specify which version of the Docker Compose API we want our containers to use.
- `settings`: The `settings` section is used to specify additional integrations or modify any default behaviors of our Ansible Container project. By default, there is one setting enabled.
- `conductor_base`: This indicates which base image we want our project to use. The `conductor` container is responsible for creating a Python environment used for running Ansible playbooks and roles. The `conductor` image will connect to the other containers that it creates, providing access to its own Python environment during the build process. Therefore, it is very important to use the same base container operating system as the container images you plan on building. This will ensure complete compatibility in terms of Python and Ansible. Think of the conductor image as a container that works in a similar way to the Ansible controller node in a standard Ansible implementation. This container will reach out to the other nodes (containers), leveraging the Docker API directly to bring our other containers into the desired state. Once we are done building our containers, the `conductor` container deletes itself by default, unless you instruct Ansible Container to retain the conductor image for debugging purposes. As well as specifying our conductor image, we can also specify other integrations in the settings section, such as Kubernetes credentials or OpenShift endpoints. We will dig deeper into these in later chapters.
- `services`: The `services` section is almost identical to the `services` section in our Docker Compose file. In this section, we will provide our `YAML` definitions, which describe the running state of our containers: which base image we will use, the container name, exposed ports, volumes, and more. Each container described in the services section is a *node* that will be configured by our conductor image running Ansible. By default the `services` section is disabled with two curly braces next to the `YAML` definition: `{}`. Before adding container definitions, delete the curly braces so that Ansible Container can access the child data.
- `registries`: The final section of our `container.yml` file is the `registries` section. It is here that you can specify container registries, from which Ansible

container will pull images. By default, Ansible Container uses Docker Hub, but you may also specify other registries, such as Quay, `gcr.io`, or locally hosted container registries. This section is also used in conjunction with the `ansible-container` push command to push your built containers to the registry service of your choice.

Ansible Container build

The second part of our Ansible Container workflow is the build process. Now that we have our first project initialized, we can explore how the `ansible-container build` function works even though we do not have any services or roles defined. From the `demo` directory, run the `ansible-container build` command. You should see output similar to the following:

```
ubuntu@node01:/vagrant/AnsibleContainer/demo$ ansible-container build
Building Docker Engine context...
Starting Docker build of Ansible Container Conductor image (please be patient)...
Parsing conductor CLI args.
Docker™ daemon integration engine loaded. Build starting.       project=demo
All images successfully built.
Conductor terminated. Cleaning up.      command_rc=0 conductor_id=c4f7806f8afb0910e4f7d
```

Running Ansible Container build for the first time on your local workstation might take a few minutes to complete, as it needs to build the `conductor` container before it can start. Keeping in mind that the conductor container is responsible for connecting to the service containers using the Docker API and executing Ansible playbooks and roles on them. Since this is a basic example of the `ansible-container build` command, there are no Ansible playbooks to run on the containers we are creating. Later in the book we will write our own roles to really explore how the conductor container functions. The below illustration demonstrates the how the conductor container connects to the service containers:

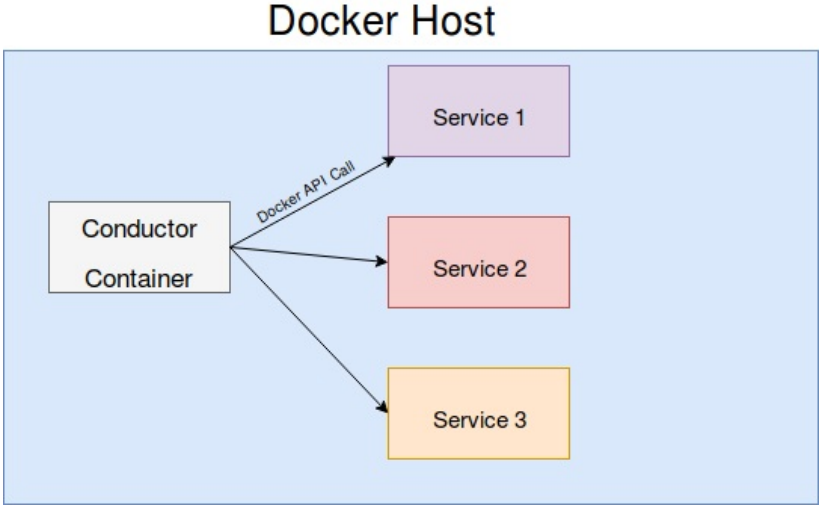


Figure 1: Conductor container bringing the service containers into the desired state However, in this example, Ansible Container will first connects to the Docker API on the localhost to determine the build

context, download the required image dependencies, and execute the build of the `conductor` container. You can see in the preceding output that our `conductor` container was successfully built for our project, `demo`. It also lists the return code, which confirms that our image was successfully built, as well as an internal conductor ID, which Ansible Container generates.

If we execute the command `docker ps -a`, we will see that no containers are currently running or exited. This is expected since we have not yet defined any containers in the `services` section of our `container.yml` file. You may also see that, since we did not pass in any arguments or configuration to instruct Ansible Container to save our `conductor` container, Ansible Container deleted the conductor after it had finished running.

```
ubuntu@node01:demo$ docker ps -a
CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS  PORTS  NAMES
```

However, if we take a look at our `docker images` output, you will find that the `conductor` image we built is cached, as well as the base image used to create it. Note that the conductor image is prefixed with `demo-*`. Ansible Container automatically names container images based on the `project-service` nomenclature. This ensures that, if you are building and running multiple container projects at once, it is easy to tell which containers belong to which projects.

In this case, our project is called, `demo` and the service we are building is `conductor`.

```
ubuntu@node01:demo$ docker images
REPOSITORY  TAG  IMAGE ID  CREATED  SIZE
demo-conductor  latest  a24fbeee16e2  38 seconds ago  574.5 MB
centos  7  3bee3060bfc8  3 weeks ago  192.6 MB
```

We can also build our project by passing in the `--save-conductor-container` flag to keep our `conductor` container after the `ansible-container build` process finishes. This is useful for debugging failed builds by having the ability to view our containers from the context that Ansible is running from. Let's try rebuilding our `demo` project, this time saving the `conductor` container: **ubuntu@node01:demo$ ansible-container build --save-conductor-container Building Docker Engine context... Starting Docker build of Ansible Container Conductor image (please be patient)... Parsing conductor CLI args. Docker™ daemon integration engine loaded. Build starting. project=demo All images successfully built. Conductor terminated. Preserving as requested. command_rc=0 conductor_id=ff84fa95d5908b076ce432d1076533679d945104e506ad5599e41 save_container=True**

This time, you will see the output reflect a slight difference: `Conductor terminated. Preserving as requested`, in addition to the output we observed earlier. This indicates that, while the conductor has stopped due to it having finished its job, the container, `demo_conductor`, remains for us to look at with `docker ps -a`: **ubuntu@node01:/vagrant/AnsibleContainer/demo$ docker ps -a CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES c3c7dc04d251 a24fbeee16e2 "conductor build –pr" 3 minutes ago Exited (0) 3 minutes ago demo_conductor**

With a firm understanding of how the Ansible Container build process works, as well as how Ansible Container builds the conductor image, we can use this knowledge to recreate the Docker Compose project we introduced at the beginning of this chapter. We can use Ansible Container to spin up the `memcached` server container we created before.

In your text editor, open the `conductor.yml` document we looked at earlier. Delete the curly braces after our `services: {}` declaration, and add the following beneath it, indented two spaces as per the `YAML` syntax: services: AC_Cache_Server: from: memcached:1.4.36 ports: - "11211:11211" volumes: - ".:/var/lib/MyVolume"

You can see that the syntax we are using to specify our service is remarkably similar to the Docker Compose syntax we created earlier. For the purposes of this demonstration, we are going to use the same parameters for `ports` and `volume` that we used with Docker Compose earlier, so that the reader may easily see the slight differences in the syntax. You will note that the `container.yml` syntax and Docker Compose syntax have many similarities, but the primary differences allow Ansible Container to be more flexible with how container services are built and deployed.

Save and close the file. If you execute the `ansible-container build` command again, you should see the following output: **ubuntu@node01:demo$ ansible-container build Building Docker Engine context... Starting Docker build of Ansible Container Conductor image (please be patient)... Parsing conductor CLI args. Docker™ daemon integration engine loaded. Build starting. project=demo Building service... project=demo service=AC_Cache_Server Service had no roles specified. Nothing to do. service=AC_Cache_Server All images successfully built. Conductor terminated. Cleaning up.**

command_rc=0
conductor_id=22126436967e7810aff44c83fb75d2276bb9a66352ddbd44a68d4
save_container=False

After Ansible Container has built our conductor image, we can observe from this output that Ansible Container now recognizes that we have a service called `AC_Cache_Server` enabled and it is attempting to build it. However, we do not have any Ansible roles associated with this service, so it returns the message `Nothing to do`. This would usually be the step in the process during which our playbooks would be executed to build the services we are creating. Since we do not have any roles defined, Ansible Container is going to skip this step and terminate the `conductor` container as usual.

Ansible Container run

Now that we have a service defined, we can use the `ansible-container run` command to start our service. The run command quickly generates a small Ansible playbook that is responsible for starting the containers specified in the `container.yml` file. This playbook leverages the `docker_service` Ansible module for starting, stopping, restarting, and destroying containers. The `docker_service` module is also useful for interfacing with the Docker daemon installed on the host OS to pull and delete images from the Docker image cache. While it's not super important to understand the implementation details behind the module at this point, it is helpful to understand how Ansible Container is working behind the scenes to run containers. Executing the `ansible-container run` command will display the stages of the playbook run, as well as `play recap`, similar to the following output:

```
ubuntu@node01:demo$ ansible-container run
Parsing conductor CLI args.
Engine integration loaded. Preparing run.       engine=Docker™ daemon
WARNING Image memcached:1.4.36 for service AC_Cache_Server not found. An attempt will b

PLAY [localhost] ***************************************************************

TASK [docker_service] **********************************************************
changed: [localhost]

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

All services running.   playbook_rc=0
Conductor terminated. Cleaning up.      command_rc=0 conductor_id=17aaa7aac99ff12427a7f
```

As you can see by reading through the playbook run output, you can easily follow the key highlights of our project as we bring it into a running state:

- Our project cannot find the `memcached` image we specified, so Ansible Container pulls it from the default repository (Docker Hub)
- A single change has been made on our host to bring our container into a running state
- None of our plays failed; one task succeeded (bringing up our container), and this successful task made a change on our host in order to bring up the container
- The conductor service was terminated

Understanding the highlights from the Ansible Container playbook is critical to seeing how Ansible orchestration deploys and maintains our applications. As we discussed previously, the Ansible team works very hard to ensure that Ansible playbook execution is very simple to understand and easy to debug. By displaying all of the steps required to bring up container projects, it is very easy to debug failures and see potential areas for improvement as we move forward into developing more complex projects. The playbook that was just executed is generated on-the-fly when `ansible-container run` is executed, and is located in the `ansible-deployment` directory. Leveraging Ansible Container to run projects takes away much of the complexity of deploying and maintaining projects since all of the deployment complexity is abstracted away. From the perspective of the user, you are concerned with ensuring the containers run and are built properly. Ansible Container becomes an end-to-end lifecycle management tool that enables containers to be built consistently and to run in an expected state every time. As we will see later in the book, having Ansible Container streamline the deployment complexity is especially useful in environments that leverage Kubernetes or OpenShift.

Now that our container run has completed, let's take a look to see what containers are running on our host using the `docker ps -a` command:

ubuntu@node01:/vagrant/AnsibleContainer/demo$ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
c4a7792fb1fb memcached:1.4.36 "docker-entrypoint.sh" 14 seconds ago Up 13 seconds 0.0.0.0:11211->11211/tcp demo_AC_Cache_Server_1

As expected, it is easy to see that our `memcached` container (version 1.4.36) is in a running state. Also, note that the `conductor` container is not running or showing up in our `docker ps` output. Ansible Container only runs the containers defined in the `container.yml` file as the *desired state*, unless you choose to keep the `conductor` container for debugging purposes. The name of the container, as we specified in our `container.yml` file, is `demo_AC_Cache_Server_1`. You may ask yourself why this is the case, as we observed when we created the `container.yml` file that we had specifically named our container `AC_Cache_Server`. One of the great features of Ansible Container is that it understands that, as developers, we might be running and testing multiple versions of our projects at once on the same host or group of hosts. By default, when Ansible Container starts containers, it automatically

appends the name of our project (`demo` in this case) to the front of the name of the running container, and a number indicating the instance ID of the running container.

In this case, since we have one instance of this container running, Ansible Container automatically appended `demo_` and `_1` to the beginning and end of our container name so that it would not conflict if we were testing multiple versions of this container on the same host.

Since we are recreating the exercise we started at the beginning of the chapter on this host, let's run the same `telnet` test using the `stats slabs` command we executed earlier to see if our `memcached` container is running and responding as expected:
ubuntu@node01:/vagrant/AnsibleContainer/demo$ telnet localhost 11211
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
STAT active_slabs 0
STAT total_malloced 0
END

It appears that our containerized service is running and properly accepting requests, listening on the network interfaces of our Docker host. Keep in mind, we specified in our `container.yml` file that our localhost port (`11211`) should be forwarded to the container's listening port (also `11211`).

Let's take a quick peek at the image cache on the Docker host. We can do this by executing the `docker images` command:
ubuntu@node01:/vagrant/AnsibleContainer/demo$ docker images
REPOSITORY TAG IMAGE ID CREATED SIZE demo-conductor latest a24fbeee16e2 48 minutes ago 574.5 MB centos 7 3bee3060bfc8 3 weeks ago 192.6 MB memcached 1.4.36 6c32c12d9101 6 weeks ago 83.88 MB

Based on this output, we can understand more clearly how Ansible Container is working on the backend. In order to bring up our `demo` project, Ansible Container had to leverage three images: `CentOS 7`, `memcached`, and `demo-conductor`. The container image named `demo-conductor` is the conductor image that was created during the build process. To build the conductor image, Ansible Container had to download and cache the `CentOS 7` base image also seen in this output. Finally, `memcached` is the

container that Ansible had to pull from the image repository, as it was specified in the `services` section of our `container.yml` file. The reader may also note that the conductor image is prefaced with the name of our project `demo`, similarly to the running state of our service container in the preceding output. This is again to avoid name conflicts and to have the flexibility to run multiple container projects at once on the same host.

Ansible Container destroy

Once you have finished experimenting with the `demo` project, we can use the `ansible-container destroy` command to stop all running instances of our container and remove all traces of it from our system. `destroy` is useful for cleaning up existing deployments and testing our containers by rebuilding them from scratch. To destroy a container, simply run `ansible-container destroy` in your project directory.

```
ubuntu@node01:/vagrant/AnsibleContainer/demo$ ansible-container destroy Parsing conduct

Engine integration loaded. Preparing to stop+delete all containers and bu

PLAY [localhost] ***************************************************

TASK [docker_service] **********************************************

changed: [localhost]

TASK [docker_image] ************************************************
```

changed: [localhost]

TASK [docker_image] *

changed: [localhost]

PLAY RECAP *

localhost : ok=3 changed=3 unreachable=0 failed=0

All services destroyed. playbook_rc=0

Conductor terminated. Cleaning up. command_rc=0 conductor_id=1d

Similar to the `run` command seen earlier, destroy executes the same playbook that was autogenerated by the `run` process. However, this time, it stops and deletes the containers specified in the `container.yml` file. You may find that the `docker ps -a` output now displays no running containers on our host: Similarly, the `destroy` function has wiped out the `conductor` container image, as well as the service

container images on the Docker host. We can validate this with the `docker images` command: **ubuntu@node01:/vagrant/AnsibleContainer/demo$ docker images REPOSITORY TAG IMAGE ID CREATED SIZE**

<none> <none> e23c420b896a 43 minutes ago 576.3 MB

centos 3bee3060bfc8 4 weeks ago 192.6 MB

Note that the only container left on the system is the base `CentOS` container. This can be manually deleted but, by default, Ansible Container leaves this on the system to speed up the process of destroying and rebuilding projects.

Summary

Over the course of this chapter, we have learned some fundamental concepts about how Ansible Container works, how it leverages the Docker Compose APIs, as well as basic lifecycle management tools built-in to Ansible Container, including `init`, `build`, `run`, and `destroy`. Having a firm grasp and understanding of what these features do and how they work is foundational when it comes to going forward and digging deeper into more complex projects we will create in Ansible Container. Although this example is included in the official Git repository of the book, feel free to recreate and tweak these examples to experiment further with how Ansible Container works. In the next chapter, we will learn how to use Ansible Container with existing roles, leveraging those roles to create reusable container artifacts.

Your First Ansible Container Project

As we learned in `Chapter 2`, *Working with Ansible Container*, Ansible Container is a powerful tool for orchestrating, deploying, and managing containers in a production environment. Using a unique set of versatile tools to initiate, build, run, and deploy Ansible Container enables developers to build containerized applications and deploy them to local environments or cloud hosting providers. Using Ansible Container, we can be sure that containers can be built accurately, will run reliably, and will provide users with a consistent experience, no matter which application or platform the containers are deployed to.

In this chapter, we will focus on building our first Ansible Container project by building an application container, testing it in our local environment, and pushing our container artifact to a container image repository. This will provide the user with a real-world use case for Ansible Container and provide experience with leveraging container-enabled roles. In this chapter, you will learn:

- What are Ansible roles and container-enabled roles?
- Roles in Ansible Galaxy
- Ansible Container NGINX role

What are Ansible roles and container-enabled roles?

Roles in Ansible are a way to organize playbooks into reusable, shareable, and discrete units that are normally broken up by an application. Inside of a role are typically a series of playbooks, configuration file templates, static files, and other metadata that are required to bring the target host (or container) into a desired state. In a typical three-tier application stack, consisting of a web server, database server, and a load balancer, each of these components might be contained in three separate Ansible roles. This provides the benefits of reuse across your infrastructure and a simple way to share playbooks over the internet or with coworkers. For example, if you wrote a load balancer role for one project, and needed to provision another load balancer for an entirely different project, you could simply download the role and assign it to another set of inventory hosts. In Ansible Core, roles are assigned to servers or virtual machines through a parent playbook that describes what the infrastructure looks like and how Ansible should bring that infrastructure into the desired state. The main benefit of roles is that they provide the user with a simple interface to access commonly used playbook tasks and resources so that the user can be certain their infrastructure is configured and running precisely as expected.

In Ansible Container, roles work in a way that is remarkably similar to Ansible Core. In Ansible Container, instead of assigning roles based on infrastructure components, roles are assigned to individual containers, which are then built using the configurations described in Ansible playbooks by the conductor container. One of the major benefits of Ansible Container is that it greatly simplifies the curve to enable containerized resources in your infrastructure. Many Ansible Core roles can be reused to build containers that function very similarly to how your infrastructure runs if you are currently using Ansible Core for configuration management. Unfortunately, since containers and full infrastructure servers are fundamentally different, not all tasks can be directly ported to Ansible Container roles without a little rework. For example, since

containers are much more lightweight than a full-blown operating system, containers usually lack tools and components that come in most operating system releases, such as init systems and resource managers.

To address this disparity, the Ansible Container project has created a different subset of roles, known as *container-enabled roles*. These are roles that are designed with a focus on containers and are usually more minimalistic then regular Ansible roles. These are leveraged to create a final container image with the smallest footprint possible while maximizing functionality and flexibility. Container-enabled roles consist of many of the same constructs that regular Ansible roles do, such as templates, tasks, handlers, and metadata. This makes it easy to get started writing roles for Ansible Container if you are familiar with Ansible syntax and language constructs.

Roles in Ansible Galaxy

Ansible Galaxy, located at `https://galaxy.ansible.com`, is a site created by the Ansible Community to share, download, and encourage the reuse of Ansible roles. From Ansible Galaxy, you can search and download roles for almost any application or platform you wish to automate. If you have experience with Ansible Core, you have undoubtedly used Ansible Galaxy to download, share, and explore roles written and maintained by other Ansible users. If you are new to Ansible, Galaxy makes it easy to find and leverage new roles from your web browser or the Ansible command line. With the release of Ansible Container, you can browse Ansible Galaxy for core roles as well as container-enabled roles. From the main website (`https://galaxy.ansible.com`) you can select BROWSE ROLES | Role Type | Container Enabled to search for roles that fit your particular requirements:

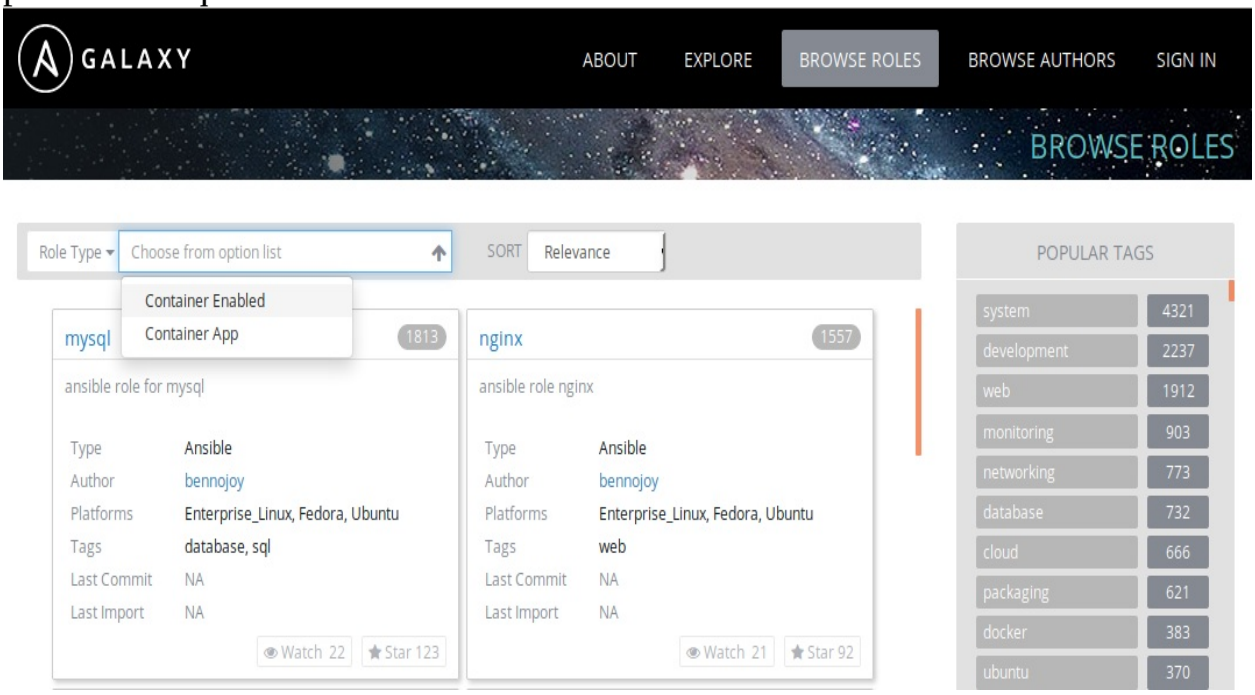


Figure 1: Ansible Galaxy website browsing for container-enabled roles More recently, the Ansible Container community created the concept of *container apps*, which are (sometimes) used to deploy multiple containers that constitute an application stack. We will look into *container apps* later in the book.

Ansible Container NGINX role

Throughout this chapter, we are going to look at how to leverage pre-written Ansible Container roles featured on Ansible Galaxy to quickly get up-and-running using roles to deploy container-based services. One of the major benefits of Ansible Galaxy is that it gives users the ability to leverage the collective knowledge pool of other users who have opted to share their projects in the form of roles. Like many DevOps engineers, you are probably not familiar with how every possible application, framework, or service should be configured for optimal performance. Online repositories such as Ansible Galaxy help to simplify the learning curve of deploying many new applications, since the applications essentially work out-of-the-box with little to no input required from the user. Users who consume roles from Ansible Galaxy also have the option of customizing already-written roles to suit their particular requirements. Throughout this chapter, we will be using the official Ansible Container NGINX role to build and deploy a functional NGINX web server container. The link to the role we are using can be found here: `https://galaxy.ansible.com/ansible/nginx-container/`.

Before we start installing and using the NGINX role, let's review the Ansible Container workflow and how it applies to prewritten roles:

- `ansible-container init`: Used to initialize a new project to use our role with.
- `ansible-container build`: Generates the conductor container that we will use to install the NGINX role. `build` is also used after installing the role to build the container image.
- `ansible-container install`: Leverages the conductor container to download and install our role within the project.
- `ansible-container run`: Runs the project locally to test and verify that the NGINX server is running as intended.
- `ansible-container push`: Pushes the built container image to your Docker Hub repository.

At any time during this chapter, you can review the completed lab exercise from the GitHub repository at: `https://github.com/aric49/ansible_container_lab/tree/master/A`

nsibleContainer/nginx_demo.

Prior to starting work on this lab exercise, it is a good idea to create a free Docker Hub account, which will allow you to upload and share the container you create. Go to `https://hub.docker.com` *to create a free account.*

Starting a new project

By now, you are probably quite familiar with initializing a new Ansible Container project and generating the file and directory structure automatically using the `ansible-container init` command. From a new directory on the Vagrant host, run `ansible-container init` to begin your new project and ensure the required files are automatically generated:

```
ubuntu@node01:$ ansible-container init
Ansible Container initialized.
```

Once you have validated that your new project files and directory scaffolding have been created, we need to run an initial, blank build of our project to create a conductor container. Before Ansible Container can install roles or build more complex projects, a conductor container needs to be present on your workstation so that Ansible Container can modify files locally and download the required dependencies that allow container roles to function properly. Now that we have initialized our project, let's do a blank build of the project in order to create a conductor container:

```
ubuntu@node01:/vagrant/AnsibleContainer/nginx_webserver$ ansible-container build
Building Docker Engine context...
Starting Docker build of Ansible Container Conductor image (please be patient)...
Parsing conductor CLI args.
Docker™ daemon integration engine loaded. Build starting.       project=nginx_webserver
All images successfully built.
Conductor terminated. Cleaning up.      command_rc=0 conductor_id=1e8a3e0164cf617ad121c
```

It is best practice to always use the same base image for your conductor container that you are using to build your project containers with to ensure compatibility. If you opt to use a different base image than the default `centos:7` you may need to modify the `container.yml` file prior to building the project. More on this in later chapters.

Once the project has been built, you should see `All Images Successfully Built` and `command_rc=0` returned, indicating that the Ansible Container conductor container has been successfully built. You may check to ensure the conductor image has been built and resides locally on your host using the `docker images` command.

Newer versions of Ansible Container (1.0+) come with prebuilt conductor images that do not require you to build projects prior to installing roles. However, it is a good idea to build conductor images unique to your projects in order to fully leverage the Ansible Container workflow more effectively.

Installing the NGINX role

Now that we have a new project initialized and a conductor image built, we can use the `ansible-container install` command to install the NGINX role from Ansible Galaxy. The syntax for this command is pretty straightforward: execute `ansible-container install` followed by the username of the user who owns the project, in this case, `ansible`, then a period `.` and the name of the project, `nginx-container`. You should see output similar to the following: **ubuntu@node01:$ ansible-container install ansible.nginx-container Parsing conductor CLI args. - downloading role 'nginx-container', owned by ansible - downloading role from https://github.com/ansible/nginx-container/archive/master.tar.gz - extracting ansible.nginx-container to /tmp/tmpip0YiN/ansible.nginx-container - ansible.nginx-container (master) was installed successfully Conductor terminated. Cleaning up. command_rc=0 conductor_id=a9e6723de6f3a236dd7823dbd999b97a5e1917bcb6794f3b0e9cd save_container=False**

Upon successful completion, you should see the message: **- ansible.nginx-container (master) was installed successfully**

This indicates that the role has been successfully downloaded and installed from Ansible Galaxy and the parent GitHub repository. The `install` command also made some modifications to the `container.yml` and `requirements.yml` files that already exist in your project directory. If you open these files in a text editor, you will find that the role has already been added to these files: `requirements.yml`: - src: ansible.nginx-container

`container.yml`: services: ansible.nginx-container: roles: - ansible.nginx-container

It is important to note that the container role has already added itself to `container.yml` with any pre-populated information the roles author wants us to use the role with. By default, Ansible Container will look inside the role, use the default information provided in the `meta/main.yml` and `meta/container.yml` files of the role, and pass this information into the build process, if it is not overridden in the `container.yml` file. Later in this chapter, we will look at how this works when we

slightly customize how the NGINX role works in our project.

The install process also added a reference to the name of the role, `ansible.nginx-container`, to the `requirements.yml` file. This file is used to keep track of the Ansible Galaxy roles and other dependencies that are being used in the project. If you are sharing your project with another developer who wants to build the project locally, the `requirements.yml` file is leveraged by Ansible Container to install all of the dependency roles in one shot. This speeds up the development process quite a bit if you are using multiple container-enabled roles in your project.

Now that we have installed the container-enabled role, let's rerun our build process and build our new container image: **ubuntu@node01:$ ansible-container build Building Docker Engine context... Starting Docker build of Ansible Container Conductor image (please be patient)... Parsing conductor CLI args. Docker™ daemon integration engine loaded. Build starting. project=nginx_webserver Building service… project=nginx_webserver service=ansible.nginx-container PLAY [ansible.nginx-container] ********** TASK [Gathering Facts] ********** ok:[ansible.nginx-container] TASK [ansible.nginx-container : Install epel-release] ********** changed:[ansible.nginx-container] TASK [ansible.nginx-container : Install nginx] ********** changed: [ansible.nginx-container] => (item=[u'nginx', u'rsync']) TASK [ansible.nginx-container : Install dumb init] ********** changed:[ansible.nginx-container] TASK [ansible.nginx-container : Update nginx user] ********** changed:[ansible.nginx-container] TASK [ansible.nginx-container : Put nginx config] ********** changed: [ansible.nginx-container] TASK [ansible.nginx-container : Create directories, if they don't exist] ********** changed: [ansible.nginx-container] => (item=/static) changed: [ansible.nginx-container] => (item=/run/nginx) changed: [ansible.nginx-container] => (item=/var/log/nginx) changed: [ansible.nginx-container] => (item=/var/lib/nginx) TASK [ansible.nginx-container : Clear log files] ********** ok: [ansible.nginx-container] => (item=access.log) ok: [ansible.nginx-container] => (item=error.log) ..... PLAY RECAP**

****
ansible.nginx-container : ok=18 changed=14 unreachable=0 failed=0 Applied role to service role=ansible.nginx-container service=ansible.nginx-container Committed layer as image image=sha256:e4416fbb0ba74f4d39a5b6522466f8c0087582de64298ac63bc43 service=ansible.nginx-container Build complete. service=ansible.nginx-container All images successfully built. Conductor terminated. Cleaning up. command_rc=0 conductor_id=86b437e5e4ebca2d29ef89193be1bd7184b5bc9e8566305dbf470 save_container=False

It looks like, our build output is a bit more interesting than previous examples. You can see that Ansible Container has recognized that our project now has a service called `ansible.nginx-container` and proceeded to run the `ansible.nginx-container` role associated with it in the `container.yml` file. During the build process, the conductor image runs Ansible Core, passing in the playbook tasks located within the role in order to bring the container image into the desired state. Each task that gets executed from the role is displayed in the build output, which allows the developer to see exactly what actions are being executed inside the container. Here are a few key takeaways to keep in mind when examining the Ansible Container build output:

- **Executed tasks**: In Ansible, each task has a unique name associated with it, which helps to make the build output easy for just about anyone to read and understand. Sometimes, logical conditions are not triggered correctly, which can cause some tasks to be skipped. Read through the tasks to make sure those tasks you are expecting to be run are actually run.
- **Changed tasks versus OK tasks**: Since Ansible, at its core, is a configuration management tool, it closely follows the principle of idempotency. In other words, if Ansible sees that a task is not required to be run since the container already has the desired state, Ansible will mark that task as `ok`. When Ansible makes a change, it will mark tasks as `changed`, indicating that Ansible modified something in the base container image. It is important to note that all tasks, regardless of whether they are `skipped`, `changed`, or `ok`, will be counted as `ok` at the end of the build process, indicating that a failure has not occurred during the task execution.
- **PLAY RECAP**: At the end of every Ansible Container build, you will be

presented with a `PLAY RECAP` section highlighting the state of the Ansible Container build. This provides a handy reference to show every task that Ansible Container executed at a quick glance and the status of the tasks: `OK`, `Changed`, `Unreachable`, or `Failed`. Tasks that have failed will cause the build process to stop immediately at the failed task unless otherwise overridden in the role.

Once the build process has completed, Ansible Container commits the changes as a single layer to the base image, creating a brand new container image for your project. Remember, in `Chapter 1`, *Building Containers with Docker*, when we used Dockerfiles to build container images? If you remember, each line in a Dockerfile represents a layer in the container image. Using Dockerfiles to build complex container images can quickly create large and unruly containers that have large file sizes.

Using Ansible Container, we can make as many changes as we want by adding tasks in the role and our final container image is still streamlined by only having one container layer created:

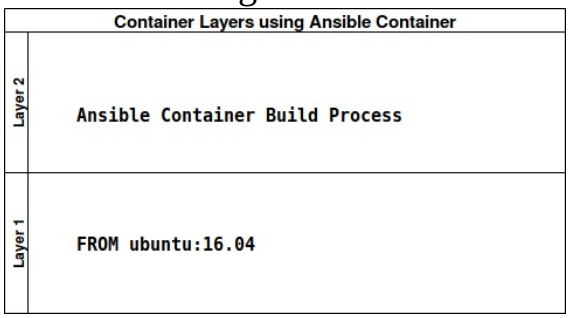


Figure 2: Container image layers in a container image built by Ansible Container However, do keep in mind that you should still strive to keep container images built by Ansible Container as small as possible by only adding the most necessary files, packages, and services. Having the benefits of Ansible Container creating only one layer in the container can quickly be outweighed if having that single layer is 2 GB in size!

Running the NGINX role

Now that our project has been built and the role has been applied without any errors, we can run our container using the `ansible-container run` command. `run` will leverage the local Ansible deployment playbooks, created during the build process, to bring up our container so that we can test it and ensure it is running as expected: **ubuntu@node01:$ ansible-container run Parsing conductor CLI args.**

Engine integration loaded. Preparing run. engine=Docker™ daemon Verifying service image service=ansible.nginx-container

**PLAY [localhost] **

**TASK [docker_service] **

changed: [localhost]

**PLAY RECAP **

localhost : ok=1 changed=1 unreachable=0 failed=0

All services running. playbook_rc=0

Conductor terminated. Cleaning up. command_rc=0 conductor_id=e62dc0e401d3d76bf771c6e8db74fb0970e9d5e57be9ad6642cff9

save_container=False

Based on the provided `PLAY RECAP`, we can easily identify that the task that was executed on our local VM to run the container has made one change in order to bring our container into a running state. The `docker ps -a` output also shows that our container is running: **ubuntu@node01:~$ docker ps -a CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES**

f213412bd485 nginx_webserver.. "/usr/bin/dumb-ini..." 2 minutes ago Up 2 minutes 0.0.0.0:8000->8000/tcp nginxwebserver_ansible..

By default, this container uses the host and container TCP port: `8000` that comes out of the box with the role. Let's use the `curl` utility to see if we can access the NGINX default website on port `8000`: **ubuntu@node01:$ curl localhost:8000**

.....(output truncated)

<title>Test Page for the Nginx HTTP Server on Fedora</title>

Based on the output from `curl`, it looks like we have successfully deployed the NGINX role on our workstation and have a functional NGINX server container running. This is great if you want a web server to run on port `8000` and want it to use only the absolute defaults. Unfortunately, this is probably not ideal for anyone to use. Let's modify our role by overriding a few defaults to see if we can get a container that runs a bit closer to what we might expect to see running in an actual functional environment.

Modifying the NGINX role

Ansible, functions and behaves quite differently from a lot of configuration management platforms such as Chef, Puppet, or Salt. Roles are seen as service abstractions that can be tweaked and modified to function in almost any way the user desires. Ansible provides the concept of variables and variable precedence, which can take input from a number of sources and, in order of precedence, can modify the role so that it will run differently depending on how the role itself is designed. It is important to note that role variable precedence is more common for Ansible Core, in which a user may have playbooks that need to run in development, staging, QA, and production environments, and require different configurations based on the environment they are deployed to.

It is still important to understand how overriding role variables and parameters can be leveraged in Ansible Container in order to build resilient and customized infrastructure artifacts. Ansible roles are designed in such a way that role variables can be overridden without modifying the role itself. Using the concept of variable precedence, Ansible Container will automatically identify role variable values in the `container.yml` file and pass these values into the role, which can be accessed by the playbooks. This allows the user to write code that is portable and repeatable simply by downloading the correct role from Ansible Galaxy and building projects using the correct `container.yml` file that contains all the customizations. Of course, not every part of a role can be overridden in the `container.yml` file, but we will learn in this section how we can make basic modifications and push our customized container images to Docker Hub.

When leveraging a role written by another user on Ansible Galaxy, the first thing a good Ansible Container engineer should do is read through the README file, usually located in the root directory of the role. The README will usually provide a guide on how to run the role in the most basic sense, as well as by providing a list of common variables that can be overridden. Having a firm grasp of the README is key to understanding how the role will function in the overall scheme of more complex projects. You can view the README for the NGINX role here: `https://github.com/ansible/nginx-container/blob/master/README.md`.

As you progress to writing your own Ansible Container roles and container-enabled applications, having an updated and accurate README file will be helpful for other users trying to use your project. Always update your README!

For this exercise, we are going to customize the `container.yml` file so that it will be exposed on the host port `80` instead of the default `8000`, and also pass in a new path for the document root, from which websites will be served. It should also be noted that we have changed the service name from the name of the role to a more commonly understood name: `webserver`. The final `container.yml` file can be found in the GitHub repository for the book in the `AnsibleContainer/nginx_demo` directory.

First, modify the `container.yml` file so that it resembles the following, keeping in mind that we are passing in the overridden variable `STATIC_ROOT` as a child parameter of the role we specified for our service. We determined that `STATIC_ROOT` was a valid variable that can be overridden in the role based on the information the developer provided to us in the role's README file. Essentially, this is telling Ansible Container to use the value the user has provided over the default value, which is hardcoded inside the role:

```
version: '2'
settings:
  conductor_base: centos:7

services:
  webserver:
    roles:
      - role: ansible.nginx-container
        STATIC_ROOT: /MySite
    ports:
      - "80:8000"

registries: {}
```

Upon rebuilding our project, Ansible Container will identify changes in the `container.yml` file. This will prompt Ansible Container to rerun the role, using the updated value for `STATIC_ROOT`. You will notice that, this time, the resulting build process will take less time, and have fewer changed tasks from the first time we executed the build. You should see an output similar to the following, keeping in mind that this example is truncated:

```
ubuntu@node01:$ ansible-container build
Building Docker Engine context...
Starting Docker build of Ansible Container Conductor image (please be patient)...
```

```
Parsing conductor CLI args.
Docker™ daemon integration engine loaded. Build starting.       project=nginx_webserver
Building service...     project=nginx_webserver service=WebServer

PLAY [WebServer] ***************************************************************
```

Running the modified role

Once the build has completed, you can execute the `ansible-container run` command to ensure that our NGINX container is still running as expected:

```
buntu@node01:$ ansible-container run

Parsing conductor CLI args.

Engine integration loaded. Preparing run.       engine=Docker™ daemon

Verifying service image service=WebServer




PLAY [localhost] ***************************************************



TASK [docker_service] **********************************************

changed: [localhost]
```

```
PLAY RECAP *********************************************************

localhost                  : ok=1    changed=1    unreachable=0    failed=0



All services running.   playbook_rc=0

Conductor terminated. Cleaning up.     command_rc=0 conductor_id=b9
```

As you can see from the preceding example, the run process completed as expected, displaying the message `All services running. Conductor Terminated. Cleaning Up` with the relevant zero return codes. This indicates that our container is running as expected. We can validate this in the local Docker environment, using the `docker ps -a` command again. In this example, we can see that port `8000` on the container is mapped to port `80` on the host, indicating that the changes in our `container.yml` file have been accurately built into the new iteration of our project:

```
ubuntu@node01:$ docker ps -a

CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS  PORTS
5d979fc13cad  nginx_webserver-webserver:20170802144124 "/usr/bin/du
```

To ensure our NGINX server is functioning as intended, we can use our trusty `curl` command to make sure we are getting the expected response on the VM localhost port `80`:

```
ubuntu@node01:/vagrant/AnsibleContainer/nginx_webserver$ curl localhost:80
```

.....

<title>Test Page for the Nginx HTTP Server on Fedora</title>

Congratulations! You have successfully built a functioning NGINX server container by leveraging a community role from Ansible Galaxy! We have even customized the role slightly by passing our own parameters into the role to slightly tweak the way the role functions and the resulting container. Unfortunately, the work we put into the container isn't of much use to us running on our local workstation. One of the major benefits of building containers is the ability to upload containers we build to image registries for other users to deploy and use. For this purpose, we will learn about the `ansible-container push` command to push our NGINX image to the free Docker Hub repository we created at the beginning of the chapter for others to use and download.

Pushing the project to Docker Hub

To enable this functionality, we will activate the final portion of the `container.yml` file by removing the curly braces after the `registries` section. Under the registries section, we will create a subsection called docker, that takes two major parameters: URL and namespace. For this example, since we are using the Docker Hub registry, we will provide the public API URL for Docker Hub (at the time of writing) and the username we created at the beginning of the chapter as the namespace parameter. The registries section of your container.yml should resemble the following:

```
registries:
  docker:
    url: https://index.docker.io/v1/
    namespace: username
```

It should also be noted that you can name your registry anything you want in the `container.yml` file. In this example, since we are using Docker Hub, I am using the name: `docker`. If you were using an internal or private registry, you could provide any name that makes sense. For example, `My_Corporate_Registry` might be a good name for an internal image registry hosted by your company. You can even list multiple registries, provided they are each named differently.

It should also be noted here that the `registries` section is a completely optional portion of the `container.yml` file. By default, the `ansible-container push` command will push to Docker Hub if no entries are written in the `registries` section of the `container.yml`. All that is required is for the user to provide a `--username` flag in the `ansible-container push` command.

The following example demonstrates me uploading my project to my personal image registry, supplying my username: `aric49`. Ansible Container will then prompt for your Docker Hub password and push the container image to your free registry, as shown. Ansible Container will automatically name your container based on the service name in your `container.yml` file.

```
ubuntu@node01:$ ansible-container push --username aric49 --tag 1.0
```

```
Enter password for aric49 at Docker Hub:
Parsing conductor CLI args.
Engine integration loaded. Preparing push.      engine=Docker™ daemon
Tagging aric49/nginx_webserver-webserver
Pushing aric49/nginx_webserver-webserver:1.0...
The push refers to a repository [docker.io/aric49/nginx_webserver-webserver]
Preparing
Layer already exists
1.0: digest: sha256:e0d93e16fd1ec9432ab0024653e3781ded3b1ac32ed6386677447637fcd2d3ea si
Conductor terminated. Cleaning up.      command_rc=0 conductor_id=6685c1596e1da31b90b25
```

It is important to always provide the `--tag` flag in the `push` command. This ensures that you can maintain version control over the various iterations of your container images in the future. In this example, we are uploading version 1.0 of our container image. If you make changes to your project in the future, you can upload a version 2.0 tag and the image registry will automatically maintain the older version, 1.0, in case you ever need to roll back or upgrade to another version of your project.

For the purposes of this demonstration, we are not going to use the default push behavior to upload to Docker Hub, instead of uploading our container image to the image registry we specified in the `container.yml` file, which just so happens to also be Docker Hub. We can use the `--push-to` flag to specify the name of the image registry we configured in our project, providing the username and image tagging details as in the preceding example:

```
ansible-container push --username username --push-to docker --tag 1.0
```

Once the container has been uploaded to our image registry of choice, we can execute a manual `docker pull` to download the container from our image registry. By default, `docker pull` requires the user to provide the name of the container image repository, the name of the image, as well as the tagged version you would like to pull. When using Docker Hub, we will use your username as the image repository since we are using our personal Docker Hub account. For example, you can pull my NGINX web server image using the `docker pull` command:

```
ubuntu@node01:~$ docker -D pull aric49/nginx_demo-webserver:1.0
1.0: Pulling from aric49/nginx_demo-webserver
e6e5bfbc38e5: Pull complete
51c9be88e17b: Pull complete
Digest: sha256:e0d93e16fd1ec9432ab0024653e3781ded3b1ac32ed6386677447637fcd2d3ea
Status: Downloaded newer image for aric49/nginx_demo-webserver:1.0
```

Use the `-D` flag to enable debug mode. This allows you to see more

details about how the Docker image is being pulled.

You can see from the preceding output that the image we are pulling is only two layers deep. This is due to the fact that Ansible Container commits all of the playbook runs as a single layer in the container image. This allows the developer to build a rather complex container while minimizing the size of the resulting image. Just remember to keep your playbooks small and efficient, or you will start to lose the benefits of containerized microservice architecture.

Now that our image has been cached locally, we can run the container manually using Docker. Of course, we could always run our project using Ansible Container directly, but the purpose of this example is to demonstrate running our container directly in Docker, which may simulate environments in which you do not, or cannot, install Ansible Container. The only caveat with this approach is that you have to specify the port-forwarding manually since that configuration is a part of our `container.yml` file and is not built intrinsically into the image itself. In this example, we are going to run the container in Docker, giving it the name `Ansible_Nginx` and specifying the container image in the following format: `username/containername:tag`

```
docker run -d -p 80:8000 --name Ansible_Nginx aric49/nginx_demo-webserver:1.0
```

The `docker ps -a` output should show the container running and functional:

```
ubuntu@node01:$ sudo docker ps -a
CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS  PORTS  NAMES
6061f0249930 aric49/nginx_demo-webserver:1.0 "/usr/bin/dumb-init n" 8 seconds ago Up 7
```

You may need to run the Ansible Container `destroy` command prior to manually running the container through Docker, as port `80` may already be used by your running project container.

Summary

In this chapter, you have learned one of the core concepts at the heart of Ansible Container: building container images using roles. By leveraging Ansible roles to create container images, you can be sure the resulting container images are built with the exact configurations that are required for production-grade, reliable, container services. Furthermore, this also ensures that container images are built using close to the exact playbook roles that your infrastructure is already using, allowing container services to be built with the assurance that services currently running in production can be replicated with, generally, little rework effort. Ansible Container provides an excellent shim between bare metal or virtualized application deployments and containerized services. Leveraging Ansible Galaxy, you can even download and share custom container-enabled roles built by yourself or other members of the Ansible Container Community.

However, as already mentioned earlier in the chapter, existing Ansible roles cannot be ported 1:1 directly to container-enabled roles, as containers function quite differently to traditional infrastructures. In the next chapter, we will learn about how to write custom Ansible container-enabled roles, as well as some best practices for porting existing roles over to Ansible Container. Get your text editors ready, we are about to get our hands dirty writing some code!

What's in a Role?

In `Chapter 3`, *Your First Ansible Container Project*, we learned the basics about Ansible Container roles, what they do, and how to download, install, and tweak them from Ansible Galaxy. In this chapter, we will look at writing our own Ansible Container roles that we can use to build custom container images from scratch. You will learn that Ansible provides an easy-to-learn, expressive language for defining desired states, and service configurations. To illustrate how Ansible Container can be used to quickly build services and run containers, over the course of this chapter we will write a role that builds a MariaDB MySQL container that can be run on your local workstation. In this chapter, we will cover:

- Custom roles with Ansible Container
- A brief overview of MariaDB
- Initializing an Ansible Container role
- What's in a container-enabled role?
- Creating the MariaDB project and role
- Writing a container-enabled role
- Customizing a container-enabled role

Custom roles with Ansible Container

One growing theme throughout the course of this book is how much freedom Ansible Container gives you to build and deploy custom container images quickly, efficiently, securely, and reliably. So far, we have looked at using Ansible Container to define and run services from prebuilt community containers, as well as leveraging community-written roles to instantiate, build, and customize our containers. This is an excellent way to get started with Ansible Container and get a head start in familiarizing yourself with the Ansible Container workflow. However, the real power of Ansible Container begins to really show itself when you start writing roles that build custom container images.

If you have experience using Ansible as a configuration management tool, you might be familiar with writing Ansible playbooks and roles already. This will definitely give you a head start with writing containerized roles, but it is not a prerequisite for working through the examples in this chapter. To put everyone on a level playing field, I am going to assume that you have no experience writing Ansible playbooks or roles, and we will essentially start from scratch. For those of you who are Ansible veterans, a lot of this will be a review, but hopefully you may learn something new. For Ansible beginners, I hope this chapter will excite your curiosity, not only to go forward into building more advanced Ansible Container roles but also to go further and explore Ansible Core configuration management concepts as well.

The original motivation behind Ansible was to create a configuration management and orchestration system that is easy for just about anyone to pick up and start working with. Ansible quickly became immensely popular amongst software developers, system administrators, and DevOps engineers as a tool that is not only easy to adopt, but also easy to customize, and even incorporate into existing platforms and configuration management tools. I first started using Ansible because, at the time, I was working on projects that required me to log into numerous bare-metal servers and virtual machines to perform the same set

of commands over and over again. At that time, I was trying to hack my way into making this easier by writing flaky shell scripts that would use SSH to push remote commands to the servers. Over the course of my research into how to make these scripts more resilient, I discovered Ansible, which I immediately adopted and it made my work far easier and more reliable then I had imagined. I believe there are two primary reasons that Ansible is so popular in the IT industry:

- Easy to understand YAML syntax for playbooks and roles. YAML is easy to learn and write, which makes it perfect for Ansible.
- Hundreds, if not thousands, of built-in modules that come with Ansible Core. These modules allow us to do almost anything you can imagine, right out of the box.

Let's look at these two unique aspects of Ansible and understand how we can leverage this ease of use in our own projects.

YAML syntax

YAML is a data serialization format that recursively stands for, *YAML Ain't Markup Language*. You may have worked with other serialization formats in the past, such as XML or JSON. What makes YAML unique is that it is easy to write, and quite possibly the most human-readable data format currently used. Ansible chose to use YAML as the basis for defining its playbook syntax and language due to the fact that, even if you do not come from a programming background, YAML is super-easy to get started with writing, using, and understanding. YAML is unique in the way that it uses a series of colons (`:`), dashes (`-`), and indentations (spaces, not tabs) to define key-value pairs. These key-value pairs can be used to define almost every type of computer science data types, such as integers, Booleans, strings, arrays, and hash tables. Following is an example of a YAML document, illustrating some of these constructs:

```
---

#This is a Comment in a YAML document. Notice the YAML document starts



MyString: "This is a string"



MyArray:
```

```
  - "Item1"
  - "Item2"


MyBoolean: true


MyInteger: 10


MyHashTable:
  KeyOne: "ValueOne"
  KeyTwo: "ValueTwo"
  KeyThree: "ValueThree"
```

The preceding example demonstrates a simple YAML file consisting of the most basic constructs: a string variable, an array (list of items), a Boolean (true/false) variable, an integer variable, and a hash table holding a series of key-value pairs.

This may look quite similar to work we have done previously in this book when modifying the `container.yml` files, as well as the Docker Compose files. These formats are also defined in YAML and consist of many of the same constructs.

A few things that I would like to call your attention to in the preceding example (you should also keep in them mind when you begin writing Ansible playbooks and roles) are:

- All YAML documents begin with three dashes: `---`. This is important because you may have multiple YAML documents defined in the same single file. Documents would then be separated using the three dashes.
- Comments are defined using the hash sign: `#`.
- Strings are surrounded by quotation marks. This separates strings from literals, such as Booleans (true or false words without quotation marks), or integers (numbers without quotation marks). If you surrounded the words *true*/*false*, or a numerical value with quotation marks, they will be interpreted as strings.
- Colons (`:`) are used to separate key-value pairs, which define almost everything.
- Indentation is indicated by two spaces. Tabs are not recognized in the YAML format. When getting started with writing YAML documents, make sure your text editor is configured to place two spaces into your document when you hit the *Tab* key. This makes it easy to quickly and naturally indent text as you type.

I realize that there is much more to YAML syntax than what I have provided in this example. My goal here is to dig a little deeper than in the earlier chapters to help give the reader a deeper understanding of the YAML format going forward. This is by no means a full description of the entire YAML format. If you want to read more about YAML, I would recommend you check out the official YAML specification website: `http://yaml.org`.

Ansible modules

The second part of what makes Ansible so popular and easy to use is the plethora of modules that Ansible can leverage right out of the box, which can do almost anything the user can think of. Think of modules as the building blocks of Ansible, that define what your playbook does. There are Ansible modules that can edit the content of files on remote systems, add or delete users, install service packages, and even interact with APIs for remote applications. Modules themselves are written in Python and get called in a scripted format from YAML playbooks. The playbooks themselves are simply just a series of calls to Ansible modules that perform a specific series of tasks. Let's look at a very simple playbook to understand how this works in practice:

```
---
- name: Create User Account
  user:
    name: MyUser
    state: present

- name: Install Vim text editor
  apt:
    name: vim
    state: present
```

This simple playbook consists of two separate tasks: creating a user account and installing the Vim text editor. Each task in Ansible calls for exactly one module to perform an action. Tasks in Ansible are defined using YAML dashes, followed by the name of the task, the name of the module, and all of the parameters you want to feed into that module indented as mentioned in the following. In our first task, we are creating a user account by calling the `user` module. We are giving the user module two parameters: `name` and `state`. The `name` represents the name of the user we want to create, and the `state` represents how we want the desired state on our remote system or container to look. In this case, we want a user to exist called `MyUser` and the state we want that user to be in is `present`. If this Ansible playbook gets executed and the user called `MyUser` already exists, Ansible will take no action since the system is in the desired state.

The second task in this playbook installs the text editor Vim on our remote system or container. To accomplish this, we are going to use the `apt` module to

install a Debian APT package. If this was a Red Hat or CentOS system, we would similarly use the `yum` or `dnf` module. The name represents the name of the package we want to install, and the state represents the desired state of the server or container. Thus, we would like the Vim Debian package to be installed. As we mentioned earlier, there are hundreds, if not thousands, of Ansible modules that can be leveraged in playbooks and roles. You can find the full list of Ansible modules by category as well as excellent examples of parameters that the modules take, in the Ansible documentation at `http://docs.ansible.com/ansible/latest/modules_by_category.html`.

The `state` parameter also takes the value `absent` to remove a user, package, or almost anything else that could be defined.

One of the major benefits of Ansible Container is that, in writing container configuration using Ansible roles, you have the entire universe of Ansible modules available to you to choose from. Unfortunately, not all Ansible modules work in the context of a container. A traditional example of this is modules that manage the state of running services, such as the `service` module. The `service` module does not run in containers, since application containers typically lack traditional init systems that you would find in a full operating system to start, stop, and restart running services. In a containerized context, this is handled by starting your container with a `CMD` or `entrypoint` statement that directly executes a service binary.

Furthermore, almost any module that manages the orchestration of cloud services or call external APIs will not run in a containerized context. This is pretty straightforward since you would usually not want to orchestrate the state of external services when you are building an independent containerized microservice. Of course, if you are writing an Ansible playbook that deploys a containerized application you previously built using Ansible Container, you can use these orchestration modules to react in certain ways when the container comes online. However, for the purposes of this chapter, we will limit our discussion only to writing roles that build containerized services.

A brief overview of MariaDB

Throughout this chapter, we will be writing an Ansible Role that builds a MariaDB database container. MariaDB is a fork of the MySQL relational database server, which provides numerous customizations and optimizations that are not found in vanilla MySQL. Out of the box, MariaDB supports numerous optimizations, such as replication, query optimization, encryption, performance, and speed improvements over standard MySQL, yet remains fully MySQL-compatible, leveraging a free and open source GPL license. MariaDB was chosen for this example due to its relative simplicity to deploy and the free nature of the application itself. In this chapter, we will build a relatively basic single-node MariaDB installation that does not contain a lot of features and performance tweaks that you would find in a production-ready installation. The purpose of this chapter is not how to build a production-ready MariaDB container, but rather to illustrate the concepts of building a containerized service using Ansible Container. If you want to go further with this example, feel free to tweak this code in any way you see fit. Extra credit to those of you who build production-ready containers!

Initializing an Ansible Container role

As discussed previously, Ansible roles are a self-contained, reusable set of playbooks, templates, variables, and other metadata that defines an application or service. Since Ansible roles are designed to work with Ansible Galaxy, Ansible Galaxy command-line tools have built-in functionality to initialize roles that contain all of the proper directories, default files, and scaffolding designed to create a functioning Ansible role with minimal hassle. This works very similarly to the `ansible container init` command for creating Ansible Container projects.

What's in a container-enabled role?

To create a new container-enabled role in Ansible Container, we are going to use the `ansible-galaxy init` command with the `container-enabled` flag to create the new role directory structure for us. To examine what happens when we use this command, let's initialize a role in the `/tmp` directory on our Vagrant VM and see what Ansible creates for us: **ubuntu@node01:/tmp$ ansible-galaxy init MyRole --container-enabled - MyRole was created successfully**

Upon successful execution of the `init` command, Ansible should return a message indicating that your new role was `created successfully`. If you run the `ls` command, you will find a new directory named after the role we just initialized. Everything that comprises of the role resides in this directory, according to the default directory structure. When you call a role from Ansible, Ansible will look in all of the locations you indicated your roles should live in and will look for a directory with the same name as your role. We will see this in more detail later in this chapter. If you navigate inside this directory, you will find a folder structure similar to the following: MyRole/

├── defaults

│ └── main.yml

├── handlers

│ └── main.yml

├── meta

│ ├── container.yml

│ └── main.yml

├── README.md

├── tasks

│ └── main.yml

├── templates

├── tests

│ ├── ansible.cfg

│ ├── inventory

│ └── test.yml

└── vars

└── main.yml

Let's take a look at what each of these directories and files does:

- `defaults/`: `defaults` is a directory that contains variables specific to your role and has the lowest priority for overriding the values. Any variables that you want to place in your role that you definitely want or require the user to override should go in the `main.yml` file of this directory. This is not to be confused with the `vars/` directory.
- `handlers/`: `handlers` are a concept in Ansible that defines tasks that should be executed in response to notify events sent from other tasks during a role execution. For example, you may have a task that updates a configuration file in your role. If your service needs to be restarted in response to that configuration file update, you could specify a `notify:` the step in your task, as well as the name of your handler. If the parent task executes and resolves a `CHANGED` status, Ansible will look inside of the `handlers/` directory for the task specified by the notify statement and then execute that task. Please note that handlers do not execute unless another playbook task specifically calls that task using the `notify:` statement and results in a changed status. Handlers are not quite as common in container-enabled roles since containers usually aren't dependent on external events and circumstances.

- `meta`: `Meta` is a directory which contains the metadata for Ansible roles. In a container-enabled role, it contains two primary files: `main.yml` which contains general metadata about the role, such as dependencies, Ansible Galaxy data, and conditions upon which the role is contingent. For the purposes of this example, we will not do very much with this file. The second file, `container.yml`, is more important to us. This `container.yml` is specific to container-enabled roles and is critical for specifying the default values that will be injected into the project-level `container.yml` file when we call our role. Here, we can specify the container image, volume information, as well as the default command and `entrypoint` data that we want our container to run with by default. All of this data can be overridden in our parent `container.yml` file if we so choose.
- `tasks`: The `tasks` directory is where we specify the tasks that actually get executed inside our container and build the service. By default, Ansible will execute the `main.yml` file and execute all tasks in the order as specified. Any other task files can go into this directory as well and can be executed using the `include:` statement from our `main.yml` file.
- `templates`: The templates directory stores the configuration file templates we want to use in our role. Since Ansible is Python-based, it uses the Jinja2 templating engine to place configuration templates into the container and update values based on variables identified in the `defaults/` and `vars/` directories. All files in this directory should have the `.j2` file extension, although this is not required.
- `tests`: Any automated testing that you would like a CICD tool to perform would go here. Usually, developers would put any custom Ansible configurations, parameters, or inventories that the CI/CD tool will requires as input in the directories and files that are autogenerated in this directory.
- `vars`: The `vars/` directory is the location in which a developer can specify other variables available to the role here. It is important to note that the `vars/` directory has a lower precedence than the `defaults/` directory, so variables defined here are more difficult to override than the ones specified in `defaults`. Usually, when I write a role, I will make all of my variables available in the `defaults` directory, as I want the user to have full power to override anything they desire. There might be circumstances in which you may not want your variables as easy to access, in which case they could be specified in the `vars/` directory.

Any file in your role named, `main.yml` indicates that the file is a default and will be executed automatically.

Now that we know what makes up a container-enabled role, we can take this knowledge and create a new Ansible Container project that will build our MariaDB MySQL role. To accomplish this, we are going to initialize a new project and create a subdirectory called `roles/`, which will contain the role we will create. When we build our project, Ansible will know to look inside of our `roles/` directory and find all of the roles we have specified and created there. Please note that the following sections of this chapter will get quite code-heavy. To make the process of following along easier, the completed example can be found in the official book GitHub repository under the `AnsibleContainer/mariadb_demo` directory. However, the best way to learn how to write Ansible code is by repetition and practice, which can only be attained by writing the code yourself. It is strongly suggested that, while it may not be practical to copy the code written in this chapter verbatim, one should obtain practice writing Ansible code by using these examples to create your own project or modifying the example in the Git repository. The more code you write, the better and more fluent of an Ansible developer you will become.

Initializing the MariaDB project and role

Now that we have a feel for how a container-enabled role is structured, we can start our MariaDB container by initializing a new Ansible Container project. In a new directory on your Vagrant host, start a new project as usual by using the `ansible-container init` command: **ubuntu@node01:$ ansible-container init Ansible Container initialized.**

Inside of our `project` directory, we can create a directory that will store our roles. In the Ansible Core, the default location for a role is in the `/etc/ansible/roles` or a `roles/` directory relative to the playbook you are executing. It should be noted, however, that roles can be stored in any location provided the Ansible installation has read access to the path. For the purposes of this demonstration, we are going to create our roles path as a `child` directory of our project. Within our `project` directory, create a new directory called `roles` and initialize our Ansible Container role inside of that directory. We will call our role `mariadb_role`:

```
ubuntu@node01:$ mkdir roles/


ubuntu@node01:$ cd roles/


ubuntu@node01:roles$ ansible-galaxy init mariadb_role --container-enab


- mariadb_role was created successfully
```

Now that our role has been created inside of our project, we need to modify our project `container.yml` file so that it knows the path we are sourcing our roles from, as well as to create a service that we will build using our role. The roles location

can be specified using the `roles_path` option as a child parameter of `settings:` in the `container.yml` file. Here, we can specify the paths we want Ansible to search for roles as list items of `roles_path` using the hyphen notation (`-`). We will specify the `roles` directory we just created. Under the `services:` subsection, we can create a new service called `MySQL_database_container`. This will leverage the role `mariadb_role` that we just created. We also want to make sure that we specify the base image we want to use for our service. For this example, the MariaDB container will be based on Ubuntu 16.04, so we want to make sure our `conductor_base` image is the same to ensure compatibility.

container.yml

Following is a sample of the `container.yml` file that provides these settings:

```
version: "2"


settings:


  conductor_base: ubuntu:16.04


  roles_path:


    - ./roles/




services:


services:


  MySQL_database_container:


```

```
  roles:


    - role: mariadb_role


registries: {}
```

At this point, we could build our project, but it would result in an empty container since our role contains no tasks from which we can build a container image. Let's make things interesting by adding tasks as well as updating the role-specific `container.yml` file.

Always remember to use the same conductor base image that your service containers will use. This will ensure maximum compatibility when building your project.

Writing a container-enabled role

As we discussed previously, it is quite difficult to walk the reader through writing code from scratch, due to the fact that file paths can get complicated rather quickly, making it easy to lose your place. In this section, I will show what the modified files look like and draw the reader's attention to the parts of the files that require explanation. Since it is quite easy to get lost, I will direct the reader to follow along in the official book GitHub repository located at the following URL: https://github.com/aric49/ansible_container_lab/tree/master/AnsibleContainer/mariadb_demo.

As developers of a container-enabled role, the most important parts of writing a role is the role-specific `container.yml` file, which specifies the default values the container will run with, as well as the tasks that are used to build the container and put all of the pieces in place. The tasks you build the container with will often determine the parameters in the role-specific `container.yml` file. When writing a role, developers will often tweak and modify the `container.yml` file as they are writing the playbook tasks. When you call a role from the project-specific `container.yml` file, the contents of the role-specific `container.yml` file will be used to build your container. At any point, a developer can override the role-specific `container.yml` file by simply modifying the parameters in the project `container.yml`.

It is important as a role developer to write sane defaults for your role's `container.yml` to enable other users to leverage your role quickly. For our MariaDB demonstration, we will create a simple role-specific `container.yml` file that resembles the following:

from: ubuntu:16.04

ports:

- "3306:3306"

entrypoint: ['/usr/bin/dumb-init']

command: ['/usr/sbin/MySQLd']

The parameters we are defining in the role-specific `container.yml` should immediately jump out at you in the exact same way in which we would define them in the project-specific `container.yml` services section. Here, we are using the Ubuntu 16.04 base image, which is the same as our conductor container. In order to make our MySQL service accessible to outside users, we are going to expose the MySQL ports `3306` on the host to ports `3306` on the container. Finally, we are going to specify a default entry point and command that the container should run when it starts. A common practice among container developers recently is to leverage lightweight init systems to start and manage processes inside of containers. A popular init system for containers is `dumb-init`, written by Yelp in 2013 to provide an easy to install, lightweight init binary for managing processes inside of containers. `dumb-init` essentially starts as PID 1 inside of the container and takes the container service executable as an argument provided to it. The benefit is that, since `dumb-init` is running as PID 1, all kernel signals will first be intercepted by `dumb-init` and forwarded to the container service (`mysqld`). `dumb-init` will also provide reaping services for our child process, should our container be suddenly stopped or restarted ungracefully. Keep in mind, using a container `init` system is not a requirement when building containers, but in some cases, it will help with running, stopping, and restarting containers if the processes don't

exit cleanly. In this example, we will use `dumb-init` as our `entrypoint` for the container, and use the `command:` parameter to specify the `/usr/sbin/MySQLd` command as an argument into it. This will start the `mysqld` process under the supervision of `dumb-init`, which will intercept all of the POSIX signals and forward them over to `mysqld`.

The second most important aspect of a containerized role is actually writing tasks that are executed in our base image to create the project container. All tasks are YAML files in the `tasks` directory of the role. Each task has a name and calls a single Ansible module with parameters to perform a unit of work in the container. Although there are no strict requirements on how to name tasks, or which order to place them in, you do want to keep in mind the flow of the playbook in terms of dependencies on other tasks that may come before or after certain steps. You also want to make sure you name tasks in such a way that any user watching the build process has a fairly good idea what is going on, even if they are not an Ansible developer. Named tasks are what give Ansible the reputation of being *self-documenting*. This means that, as you write code, the code basically documents itself since almost any user reading your code should know immediately what it does based on the naming of the tasks. It should also be noted that, as all services and applications are different, they are all deployed and configured differently as well. With Ansible and containerization, there is no *one-size-fits-all* approach that can adequately capture the best practices for deploying and configuring your application. One of the benefits of Ansible is that Ansible provides the tools needed to automate almost any configuration one can think of to ensure applications get built and deployed reliably. The following is the content of the `tasks/main.yml` file in the MariaDB role we are in the process of writing. Take a moment and read through the playbook as it is; we will go through each of these tasks one-by-one to provide more detail as to how the playbook runs. As I describe

how the playbook works, it would be helpful to glance back at each task as you read the description.

tasks/main.yml

This file is located in `roles/mariadb_role/tasks/main.yml`:

```
---

- name: Install Base Packages

  apt:

    name: "{{ item }}"

    state: present

    update_cache: true

  with_items:

    - "ca-certificates"

    - "apt-utils"
```

```
- name: Install dumb-init for container init system
  get_url:
    url: https://github.com/Yelp/dumb-init/releases/download/v1.2.0/dumb-init_
    dest: /usr/bin/dumb-init
    owner: root
    group: root
    mode: 0775


- name: Create MySQL Group
  group:
    name: mysql
```

```
    state: present



- name: Create MySQL Users

  user:

    name: mysql

    state: present

    groups: mysql

    append: true



- name: Install MySQL Server
```

```
  apt:
    name: mariadb-server
    state: present
    update_cache: true

- name: Change Permissions on directories
  file:
    path: "{{ item }}"
    owner: mysql
    group: mysql
    mode: 0777
    state: directory
```

```
    recurse: true
  with_items:
    - "/etc/mysql/"
    - "/var/lib/mysql/"
    - "/var/run/mysql/"

- name: Remove my.cnf
  file:
    path: /etc/mysql/my.cnf
    state: absent
```

```
- name: Install MySQL Configuration File
  template:
    src: my.cnf.j2
    dest: /etc/mysql/my.cnf
    owner: mysql
    group: mysql


- name: Initialize Database
  include: initialize_database.yml
  when:
    - initialize_database == true
```

Task breakdown (main.yml)

- `Install Base Packages`: Since we are building an Ubuntu 16.04 image, the `Install Base Packages` task calls the `apt` package module to install two packages: `ca-certificates` and `apt-utils`, which are required for the preceding tasks. We want the state of these packages to be present and installed in the `container` and the `apt` database cache to be updated prior to installing the packages as well. We are able to install multiple packages using the `with_items` operator. `with_items` iterates over the list items specified and run each value through the apt module. Using `with_items`, we don't have to create two or more separate tasks to perform the same action repeatedly. In the `name` section of the task, we specify `{{ item }}`, which is a Jinja2 keyword variable that tells Ansible that it is about to iterate over a list.
- `Install dumb-init for the Container Init System`: This task leverages the `get_url` module in order to download a remote file from the internet into the container. In this particular case, we are downloading the `dumb-init` binary and placing it in the container at the destination `/usr/bin/dumb-init`. We are changing the permissions so the file will be owned by root, in the root group, and will be executable. Ansible allows us to perform all of these actions in a single module call.
- `Create MySQL User and Group`: The next two tasks are quite similar. With these tasks, we are starting to lay the groundwork for installing our MariaDB MySQL service. We are calling the `user` and `group` module to create a user called `mysql`, create a group also called `mysql`, and add the user to the group. Notice that, in the group module, we specify `append: true`. This indicates that whatever groups are already assigned to the `mysql` user, we also want to append `mysql`. This is a safe option to add to any group declaration so that we don't accidentally remove users from other groups they may need to belong to.
- `Install the MySQL Server`: This task functions in a very similar way to the first task that we saw in the playbook. However, instead of installing multiple packages, we are calling the apt module to install only one package, the MariaDB server. As usual, we want the package to be installed and present,

as well as having the package cache updated. Arguably, we could have added this package to the list of installed packages in our very first task, and that would definitely work. However, as a matter of development style, I like to provide a logical distinction between steps in my playbooks so that things do not get confusing further on. After all, installing the base packages is usually a separate and distinct step from installing the core service package.

- `Change permissions on directories`: This task is one of the more complex tasks in the playbook. In this task, we have a handful of directory paths that need to have their permissions changed so that the MySQL service can write data to them. The `file` module allows us to create, delete, or modify any file present in our container. Similarly to our first task, we are going to call the file module on our `{{ item }}` keyword variable so that each list item specified in our `with_items` will have the same permissions and attributes applied to it. If the paths specified do not exist, Ansible will create them with the state `directory` and apply the appropriate permissions to them. We are also providing the `recurse: true` option so that the permissions will apply to all subdirectories from those locations specified.
- `Remove my.cnf`: `my.cnf` is the primary configuration file that is used by MySQL to configure how the database service operates. When MariaDB is first installed, it creates `my.cnf` as a symlink, which leads to another configuration file it uses instead. We don't want this behavior; hence, we are going to delete the default `my.cnf` file using the file module and set the state value to `absent`. We will use our own `my.cnf` file instead.
- `Install the MySQL Configuration File`: Now that the default `my.cnf` symlink has been removed, we can call the template module to place a new `my.cnf` file in its place. The `templates` module works by leveraging the local `templates` directory and looking for a file that matches the name of the source file we are specifying, `my.cnf.j2`. Templates use the Jinja2 templating language to put the new configuration in place and replace any variables sourced from the role. The location for the new configuration file will be `/etc/MySQL/my.cnf` and will have the appropriate permissions applied to it.
- `Initialize the Database`: The final task in this playbook is known as an `include` task. Include statements, logically enough, include other playbook YAML files for execution. Usually, include statements are a great way to break down your playbooks into logically grouped blocks of similar tasks. In this scenario, we want to include the playbook `initialize_database.yml`, based on

the logical condition that the variable `initialize_database` is set to true. In other programming languages constructs such as *if*, *else...if*, and *else* exist, to indicate logical evaluations. Ansible handles this using the keyword `when` to list the conditions for *when* an action will occur. In this case, *when* the variable `initialize_database` is true, the playbook `initialize_database.yml` will be executed. If the variable is set to false, it will skip those tasks.

Now that we have a good understanding of what the tasks inside of the `main.yml` playbook are running, let's take a look at the tasks inside of the `initialize_database.yml` playbook to see what will happen if the `initialize_database` variable evaluates to true:

tasks/initialize_database.yml

This file is located in `roles/mariadb_role/tasks/initialize_database.yml`.

```
---


- name: Temporarily Start MariaDB Server shell: MySQLd --user=MySQL &


- name: Create Initial Accounts shell: MySQL -e "CREATE USER '{{ default




- name: Grant Privileges to New Account shell: MySQL -e "GRANT ALL ON




- name: Create Default Databases shell: MySQL -e "CREATE DATABASE {{

  with_items:

    - "{{ databases }}"
```

```
- name: Flush Privileges

  shell: MySQL -e "FLUSH PRIVILEGES;"
```

Task breakdown (initialize_database.yml)

- `Temporarily Start MariaDB Server`: By default, when MariaDB is first installed, there are no databases created and no users have access to the database. In some cases, we may want to spin up a vanilla MariaDB server and have an external user or tool create the default databases and access credentials. However, there may also be an equal number of circumstances in which we might need to create database instances that come with built-in databases and user credentials. In order to create these defaults, we will first need to start the MySQL server so that it can be accessed from the command line. To start the server temporarily, we will call the `shell` module, which evaluates shell commands in a very similar manner, as if you were typing them on a Bash prompt. We will run the command `mysqld`, specifying the user to run as `mysql`, and force the server to run in the background using the ampersand indicator (`&`). The MySQL server will continue to run at this point until the build has completed and the container has been shut down.
- `Create Initial Accounts`: The create initial accounts step similarly calls the shell module in order to leverage the MySQL command-line client. The `-e` flag allows us to pass in executable SQL commands, which will be evaluated by the server. We will use this particular command to create a default username and password that we can use to log in to the database. The default credentials will be sourced from our variables, hence the double curly braces.
- `Grant Privileges to New Account`: Using the `shell` module again, we can call the MySQL client to grant privileges on the new account we created previously. In this example, we will grant all privileges, connecting from any network interface to have access to this MySQL server.
- `Create Default Databases`: Using our iteration or looping operator `with_items`, we can pass in a list of databases we want the MySQL client to create. In our `defaults/main.yml` file, we have specified the databases variable as an array or list of items. Ansible will identify the fact that our databases variable is

actually a list of strings and iterate over that. The result is that any number of databases we specify as a list item of the databases variable will be iterated over and created in our MySQL container.

- `Flush Privileges`: One final call to the shell module will allow us to execute the SQL command, `FLUSH PRIVILEGES`, which allows the new user accounts to take effect in the database. After this command executes, the container build will have finished, signaling Ansible Container to shut down the intermediate container and commit the final changes to the container we just finished building.

Now that we have had a look inside the `tasks` directory and learned about how the role executes tasks, let's look inside of the `templates/` directory to learn about the templated configuration files we are generating and passing into the container. You will observe that, in the `roles` templates directory, there is one file: `my.cnf.j2`. This is the template for `my.cnf` file that we want Ansible to compile and pass into the container during the build process. It is a best practice to always name your Ansible template file after the destination filename with the `.j2` extension. This indicates the file is a Jinja2 template and contains variables and Jinja2 logic for Ansible to evaluate.

Jinja2 is a powerful templating language that can do some pretty neat stuff in your projects. Although not strictly required, having a working understanding of Jinja2 can help you a lot in your Ansible development. You can read more about the Jinja2 language at the official website: *http://jinja.pocoo.org/.*

The following is the content of the `my.cnf.j2` file in the `templates` directory:

templates/my.cnf.j2

This file is located in `roles/mariadb_role/templates/my.cnf.j2`: # Ansible Container Generated MariaDB Config File [client]

port = 3306

socket = /var/lib/MySQL/MySQL.sock

The MariaDB server

[MySQLd]

user = MySQL

port = 3306

socket = /var/lib/MySQL/MySQL.sock datadir = /var/lib/MySQL

bind-address = 0.0.0.0

skip-external-locking

key_buffer_size = {{ key_buffer_size }}

max_allowed_packet = {{ max_allowed_packet }}

table_open_cache = {{ table_open_cache }}

sort_buffer_size = {{ sort_buffer_size }}

Notice that, in the first line of the file, we are spelling out to the user in a comment block that the file is an *Ansible Container-Generated MariaDB Config File*. If you have experience of connecting into remote servers to troubleshoot problems, you will know how handy it is to know exactly where the files come

from, where the values are populated from, and which configuration management tool is responsible for putting those files there. While not strictly required, and surely as a matter of taste, I like to place such banners on files that Ansible touches. This way, someone, later on, will know exactly how this container came to be in this state.

The next thing you will notice is that the last four lines of the configuration file have values set to double curly braces with the name of the configuration file key in between them. As we discussed earlier, the double curly braces indicate Jinja2 variable parameters. When Ansible evaluates this template prior to installing it in its destination inside the container, Ansible will parse the file for all Jinja2 blocks and execute the instructions it reads to bring the template into the desired state. This could mean populating the values of variables, evaluating logical conditions, or even sourcing environment information that the template requires. In this case, Ansible will see the double curly braces and replace them with the values defined for those variables. By changing or overriding the variables, Ansible makes it quite easy to change the way containers and applications function. Also, notice that the names of the variables match the configuration option they are modifying. Variable names are purely up to the developer, and as such, the developer may choose what they want the names to be. However, it is usually a best practice to use descriptive variable names so that it is very clear to the user what settings they are overriding or modifying.

Reading through these role files, it is probably very clear to you that variables have a lot to do with how Ansible runs, how templates are populated, and even how tasks are executed and controlled. Let's now take a look at how variables are defined in roles and how we can leverage variables to make roles more flexible and enable their reuse. As stated before, role variables can be stored in two places: the `defaults/` directory or the `vars/` directory. As the developer, you can choose which location (or both locations) you want to store your variables in. The only difference is the variable precedence in which the variables are evaluated in. The variables stored in `defaults/` are the easiest to override. Variables stored in `vars/` have a slightly lower precedence and thus are more difficult to override. In this example, I have opted to store all variables in the `defaults/` directory in the `main.yml` file. Let's see what that file looks like: ---

defaults file for MySQL_role initialize_database: true default_user: "root"

default_password: "password"

databases:

- "TestDB1"

- "TestDB2"

- "TestDB3"

#MySQL Basic Tuning for my.cnf key_buffer_size: "16K"

max_allowed_packet: "1M"

table_open_cache: 4

sort_buffer_size: "64K"

Here, you can see that these are all the variables we have seen before, referenced in the role tasks as well as the templated file for `my.cnf`. Variable YAML files are essentially just static YAML files that use exactly the same YAML constructs we explored in the beginning of the chapter. For example, this file is, by default, initializing the database by setting the `initialize_database` variable to the Boolean value of true. We also can see that the default credentials that will be created in the database are set to the strings `root` and `password`, as well as a list of test databases that will get created during the initialize database tasks. Finally, towards the bottom, we have a grouping of variables that define the values that will be incorporated into the template. If we build the role as-is, without providing any variable overrides, we will get a container built with exactly these specifications. However, this book would not be complete without exploring exactly how we can go about customizing the role we just wrote!

Building the container-enabled role

Before we begin customizing our role, let's first build the role and demonstrate the default functionality using the default variables we specified. Let's go ahead and return to our Ansible Container workflow and execute `ansible-container build`, followed by the `ansible-container run` commands from the `root` directory of our project:

```
ubuntu@node01:$ ansible-container build
Building Docker Engine context...
Starting Docker build of Ansible Container Conductor image (please be patient)...
Parsing conductor CLI args.
Docker™ daemon integration engine loaded. Build starting.       project=mariadb_demo
Building service...     project=mariadb_demo service=MySQL_database_container

PLAY [MySQL_database_container] ************************************************

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

TASK [mariadb_role : Install Base Packages] ************************************
changed: [MySQL_database_container] => (item=[u'ca-certificates', u'apt-utils'])

TASK [mariadb_role : Install dumb-init for Container Init System] **************
changed: [MySQL_database_container]

TASK [mariadb_role : Create MySQL Group] ***************************************
changed: [MySQL_database_container]

TASK [mariadb_role : Create MySQL Users] ***************************************
changed: [MySQL_database_container]

TASK [mariadb_role : Install MySQL Server] *************************************
changed: [MySQL_database_container]

TRUNCATED
```

You may note from the build output that Ansible is taking the list items we provided in the task using the `with_items` iteration operator and exactly building our image, bringing it into the desired state based on the variables we have provided in our role, which for now, are the default variables.

Let's run our project and attempt to access the MySQL services:

```
ubuntu@node01:$ ansible-container run
Parsing conductor CLI args.
```

```
Engine integration loaded. Preparing run.       engine=Docker™ daemon
Verifying service image service=MySQL_database_container

PLAY [localhost] ***************************************************************

TASK [docker_service] **********************************************************
changed: [localhost]

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

All services running.   playbook_rc=0
Conductor terminated. Cleaning up.      command_rc=0 conductor_id=e33fb25670f1bc040a6ed
```

Executing `docker ps -a` will show that our container is running with port `3306` exposed on the host:

```
ubuntu@node01:~$ docker ps -a
CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS  PORTS  NAMES
7cab59c33cfa  mariadb_demo-MySQL_database_container  "/usr/bin/dumb-ini…"  About a minu
```

To test to ensure everything is working, we can download and install the `mariadb-client` package, or any MySQL client of your choosing:

```
ubuntu@node01:~$ sudo apt-get install -y mariadb-client
```

Once the MariaDB client has been installed, you can use the following command to connect to the MariaDB container exposed on the localhost of the Vagrant VM. If you're unfamiliar with the MySQL client, remember that all flags passed into the client do not have spaces after them. It looks a little strange, but it should drop you into a MySQL console:

```
ubuntu@node01:~$ MySQL -h127.0.0.1 -uroot -ppassword
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 2
Server version: 10.0.31-MariaDB-0ubuntu0.16.04.2 Ubuntu 16.04

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

Let's run the `show databases;` command to see if the test databases we have

specified in our default variables are being created:

```
MariaDB [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| TestDB1            |
| TestDB2            |
| TestDB3            |
| information_schema |
| MySQL              |
| performance_schema |
+--------------------+
6 rows in set (0.00 sec)

MariaDB [(none)]>
```

It appears as though everything was created properly and is working as expected. When you are done working in this session, you can exit from the MySQL CLI using the `exit` command. Use `ansible-container destroy` to reset your environment. Let's make things interesting by customizing our role and sourcing external variable values.

Customizing the container-enabled role

As we saw in the previous chapter, it is very easy to abstract away from changes in Ansible Container projects by adding variables directly to the project `container.yml` file and rebuilding the project. This provides the added convenience of having all of our configuration changes in a single location and functioning effectively as a single point of truth. This might be sufficient for some use cases, but what about circumstances in which one would need to provide containers configured differently to support multiple environments or locations, such as the development, testing, QA, and production environments? You could simply update the `container.yml` file and build separate images for these scenarios. However, Ansible Container provides us with a better way to handle this by providing the ability to source external variable files. A part of the `ansible-container` parent command is the `--var-files` flag, which provides the option to source an external YAML file for variable definitions. This provides us with an abstraction that allows separate builds to run and exist in parallel using different configuration options. This also allows us to customize our role using separate variable files for almost any circumstance that can be version-controlled along with our project.

To enable this functionality, let's create a directory in the root of our project (the same level as the project-specific `container.yml`) called `variable_files`. Inside of this directory, we will create three separate files: `dev.yml`, `test.yml`, and `prod.yml` with slightly different configuration options. The following are examples of these three files. Do enjoy my Star Trek references!

Before we begin, it would be a good idea to perform an `ansible-container destroy` action before rebuilding the containers using different variables. This way, you can see exactly what is being changed during the build process.

In the development, the primary user of our database will be Yeoman Rand. She will be primarily concerned with Starfleet data:

Development Defaults for MySQL role initialize_database: true default_user: "yeoman_rand"

default_password: "starfleet"

databases:

- "starfleet_data"

In `System Test`, Mr. Spock will be the primary user of our database. He has slightly more interest in data related to Planet Vulcan, ship ordinances, shuttlecraft, as well as federation data.

variable_files/test.yml

This is located in `<project_root>/variable_files/test.yml`:

```
---
# System Test Defaults for MySQL role
initialize_database: true
default_user: "MrSpock"
default_password: "theBridge"
databases:
  - "planet_vulcan"
  - "federation_data"
  - "shuttle_crafts"
  - "ship_ordinances"
```

variable_files/prod.yml

In production, Captain Kirk is going to need to store vastly different data than the other crew members. We will need to enhance our MySQL configuration from a bit to support the added overhead of storing the Captain's Logs, Enterprise data, as well as federation mandates. This file is located in: `<project_root>/variable_files/prod.yml`: --- # defaults file for MySQL_role initialize_database: true default_user: "captainkirk" default_password: "ussenterprise" databases: - "CaptainsLog" - "USS_Enterprise" - "InterstellarColonies" - "FederationMandates" #MySQL Basic Tuning for my.cnf key_buffer_size: "128K" max_allowed_packet: "20M" table_open_cache: 12 sort_buffer_size: "128K"

You may also notice that not all variables are being overridden in every example shown here. In cases where variables are not being overridden by the sourced files, Ansible will take the values present in `defaults/main.yml` in the role. It is important that your role defaults provide values for all variables, as variables referenced without values will break the build process.

Variable files can be named anything you want. Since we are sourcing these files during the build process, and they are not something that Ansible Container will automatically discover, the naming convention is entirely up to you.

We can build containers based on any of these variables by executing our `ansible-container build` command and adding the `--vars-files` flag as a parameter of the `ansible-container` command. Remember, we always run `build` commands in the same directory as the project-specific `container.yml` file: **ansible-container --vars-files variable_files/dev.yml build**

During the build, you should notice that many of the tasks look slightly different based on the variables we are providing. For example, when sourcing development variables, you will see that only one database gets created: `starfleet_data`. This is an indication that the new variables have been sourced and are populated correctly in the build process. Let's perform an `ansible-container run`

of the new version of our container and try to log in with the same credentials as before:

```
ubuntu@node01:$ ansible-container run
Parsing conductor CLI args.
Engine integration loaded. Preparing run.       engine=Docker™ daemon
Verifying service image service=MySQL_database_container

PLAY [Deploy mariadb_demo] *****************************************************

TASK [docker_service] **********************************************************
changed: [localhost]

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

All services running.   playbook_rc=0
Conductor terminated. Cleaning up.      command_rc=0 conductor_id=bf5221a710736d238b099
```

Now, to log in using the MariaDB client:

```
ubuntu@node01:$ MySQL -h127.0.0.1 -uroot -ppassword
ERROR 1698 (28000): Access denied for user 'root'@'172.18.0.1'
```

It is very clear to see that the default credentials we have in the role defaults are no longer working. Let's try again using the credentials we specified in our development variable file for the Yeoman Rand user:

```
ubuntu@node01:$ MySQL -h127.0.0.1 -uyeoman_rand -pstarfleet
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 3
Server version: 10.0.31-MariaDB-0ubuntu0.16.04.2 Ubuntu 16.04

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

It looks like our new container is working using the sourced variable files for development. Let's run the `show databases;` command to make sure the database was properly created and exists: **MariaDB [(none)]> show databases; '+--------------------+ | Database | +--------------------+ | information_schema | | MySQL | | performance_schema | | starfleet_data | +--------------------+ 4 rows in set (0.04 sec) MariaDB [(none)]>**

As you can see, the database `starfleet_data` exists alongside the default MariaDB databases such as `information_schema`, `MySQL`, and `performance_schema`. It appears that the container was built properly and is ready for deployment in our development

environment (for the purposes of this example). We can now push the image to a container registry of our choosing. For this example, I will add Docker Hub to the `registries` section of our project-specific `container.yml` file, specifying the namespace as my Docker Hub username (remember to remove the curly braces after the start of the registries stanza). Once that file is saved, let's tag the image as `dev` and `push` it up to our Docker Hub repository so that we have a build image artifact that we can use to deploy our application:

container.yml

The project-specific `container.yml` file is located in the `root` directory of your project:

```
registries:

  docker:

    url: https://index.docker.io/v1/

    namespace: username
```

Push the image using the `--push-to` flag:

```
ubuntu@node01:$ ansible-container push --push-to docker --tag dev

Parsing conductor CLI args.

Engine integration loaded. Preparing push.      engine=Docker™ daemon

Tagging aric49/mariadb_demo-MySQL_database_container

Pushing aric49/mariadb_demo-MySQL_database_container:dev...
```

The push refers to a repository [docker.io/aric49/mariadb_demo-MySQL

Preparing

Waiting

Pushing

Pushed

Pushing

Pushed

Pushing

Pushed

dev: digest: sha256:98d288cfa09acc3f06578532cd6ccd78af0eb65b84ba3b0

Conductor terminated. Cleaning up. command_rc=0 conductor_id=32

It's not completely necessary to configure Docker Hub as the image registry in the `container.yml` *file, as Ansible Container will default to using Docker Hub. However, I like to make sure I don't accidentally*

push images to the wrong registries, so it is best practice to always provide the image repository in the `container.yml` *file and always push using the* `--push-to` *flag command to specify the correct repository.*

We can do the same build process for our `test.yml` configuration as well as our `prod.yml` configurations and push those up to the Docker Hub repository (remembering to do a `destroy` between builds). Notice that, while uploading a different version of the image, Docker will automatically identify layers of the image that is identical to the previously uploaded version. In this case, Docker will help you to save bandwidth and resources by not pushing layers that are identical, but only the layers that have changed, as shown in the following. Note the `Layer already exists` lines:

```
ubuntu@node01:$ ansible-container push --push-to docker --tag test

Parsing conductor CLI args.

Engine integration loaded. Preparing push.      engine=Docker™ daemon

Tagging aric49/mariadb_demo-MySQL_database_container

Pushing aric49/mariadb_demo-MySQL_database_container:test...

The push refers to a repository [docker.io/aric49/mariadb_demo-MySQL

Preparing

Waiting
```

Layer already exists

Pushing

Layer already exists

Pushing

Layer already exists

Pushing

Pushed

test: digest: sha256:1f9604585e50efe360a927f0a5d6614bb960b109ad8060

Conductor terminated. Cleaning up. command_rc=0 conductor_id=ef

We should now have three different container image artifacts available to download. These images are available to download and deploy in our imaginary development lab, system test lab, as well as our production environment. These container images are guaranteed to run in these environments in the exact same way as they do in our local workstations. At this point, we can do a final exercise and run all three of these containers on different ports in order to simulate these containers running in different environments with different configurations. To quickly demonstrate this, we will use the native `docker run` command to specify

our tagged image and the ports we want the container service to use; we also specify that our service should run in the background using the `-d` flag. Notice that each instance of our container that we are creating uses the `dev`, `test`, and `prod` tags as well as our user repository address. In my case, it is `aric49`:

```
docker run -d --name MySQL_Dev -p 3308:3306 aric49/mariadb_demo-MySQL_database_containe

docker run -d --name MySQL_Test -p 3309:3306 aric49/mariadb_demo-M

docker run -d --name MySQL_Prod -p 33010:3306 aric49/mariadb_demo
```

Testing the container functionality is exactly the same process as before. We can use the MariaDB client to log into an instance of our containers. This time, however, we will need to specify which port our service is listening on since all three instances cannot listen on the default port `3306` on the host networking side. If we wanted to log into our production container, we could specify the credentials for Captain Kirk using port `33010` and the `ussenterprise` password:

```
ubuntu@node01:$ MySQL -h127.0.0.1 -ucaptainkirk -pussenterprise -P33010

Welcome to the MariaDB monitor.  Commands end with ; or \g.

Your MariaDB connection id is 2

Server version: 10.0.31-MariaDB-0ubuntu0.16.04.2 Ubuntu 16.04
```

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>

Leveraging the same Ansible Container project and container-enabled role, we were able to use the Ansible Container default primitives in order to build containers with a variety of configurations that are available for use under different circumstances and use cases. This approach enables us to be certain that the build process will remain exactly the same throughout future build iterations, but we will have the flexibility to supply new configuration values into our role without modifying the code we wrote previously. Using container tagging, a snapshot of container configurations can be captured and shared with other users. We now have a tremendously useful and repeatable pipeline to ensure that future versions of our application containers have traceability back to the source roles used to generate them. Even if the container images are accidentally deleted from our image registry, we can easily build and rebuild our containers at any time, since all configuration in our containers is declared as code using the Ansible playbook language. If you have worked in an IT-related DevOps or a systems administrator position for very long, you will understand how valuable it is to have this level of insight into your infrastructure.

References

- **Official YAML Standard Guide**: http://yaml.org
- **Ansible Module Index**: http://docs.ansible.com/ansible/latest/modules_by_category.html
- **Ansible Playbook Specification**: http://docs.ansible.com/ansible/latest/playbooks.html

Summary

Over the course of this chapter, we looked at how roles not only enable reuse in Ansible Container but are actually the bread and butter of what makes Ansible Container a powerful tool for building and managing containers.

We first looked at how we can use the Ansible Galaxy command-line tools to create the shell for a container-enabled role, all necessary directories, and default YAML files from which we can build our role. From there, we wrote a custom role that builds a MariaDB container using a sane set of default configuration options. Finally, we developed an abstraction layer on top of our role by passing in custom variable configuration options, from which we can customize our container project without modifying any code.

I hope this chapter has effectively demonstrated the raw power available to you using the Ansible Container project. Up until this point, I think it is easy to make the assumption that it is easier to build container images using Dockerfiles and not worrying about the added overhead of Ansible Container. I hope you will now understand that the benefits of using Ansible Container far outweigh the slight layer of additional complexity required. Using Ansible Container, you can create a powerful pipeline for building, running, testing, and pushing container images. By leveraging the easy-to-understand Ansible playbook syntax language, we have a basis from which we can start building a modern, agile, containerized infrastructure that gives us the ability to deploy changes quickly and truly begin to embrace the promise of genuinely modular infrastructure.

Now that we understand how to build and deploy truly custom containers we can start to look at Kubernetes, an open source framework for automating the deployment, orchestration, and management of containers at scale.

Containers at Scale with Kubernetes

Kubernetes is by far one of the most popular open source projects to take the IT world by storm. It seems like almost everywhere you go, every blog you read, or news article you encounter, tells the tale of how Kubernetes has revolutionized the way DevOps and IT infrastructure are handled. With good reason, Kubernetes has truly taken a firm grasp of the IT landscape and introduced new concepts and ways of looking at infrastructure like no other platform before it. You might be in the camp of IT professionals who have heard of Kubernetes, but you have no idea what it is or how it can really benefit your infrastructure. Or, you could be where most of us are today, in the process of containerizing applications and workloads but don't want to dabble in the additional complexity and murky water of Kubernetes just yet. Finally, you could be one of those lucky DevOps engineers or IT administrators who have successfully adopted containers and Kubernetes and is able to really reap the reliability and flexibility that Kubernetes provides.

The purpose of this chapter is to provide an overview of what Kubernetes is, how it works (at a high level), and how to deploy your containerized applications to Kubernetes clusters using Ansible Container. Before we dive in too deep, you might ask, what is Kubernetes exactly? Kubernetes is a platform developed by Google for deploying, managing, configuring, and orchestrating containers at both small and large scales. Kubernetes was started as an internal project at Google, known as **Borg**, for managing the automatic deployment and scaling of containers across the vast Google infrastructure footprint. Based on some real-world lessons learned with Borg, Google released Kubernetes as an open source project so that other users and organizations could leverage the same flexibility to deploy containers at scale. Using Kubernetes, one can easily run containerized applications across multiple clustered nodes, automatically maintaining the desired number of replicas, service endpoints, and loadbalancing across the cluster.

Throughout this book, we have looked closely at how we can use the Ansible

Container platform to quickly and reliably build container images using Ansible Playbooks and run those containers on our local workstation. Since we now understand quite well how to build version control and configuration management inside of our containers, the next step is using configuration management to declare how our applications should run outside of our container. This is the gap that Kubernetes fills. And, yes, it is just as awesome as it sounds. Ready? Let’s get started.

Throughout this chapter, we will cover:

- A brief overview of Kubernetes
- Getting started using Google Cloud Engine
- Deploying an application in Kubernetes using kubectl
- Writing a Kubernetes manifest
- Deploying and updating containers in Kubernetes

A brief overview of Kubernetes

Admittedly, when one thinks of Kubernetes, one might immediately think of the complexity and multifaceted hierarchy of concepts associated with Kubernetes and be quick to think that this chapter will be over the reader's head in terms of how to understand and apply these concepts. Most users who have unsuccessfully attempted to venture into Kubernetes in the past may still feel the scars and be wary about moving forward with Kubernetes. Container automation using Kubernetes can quickly get quite complicated, but the rewards for learning and using Kubernetes are vast. Before we go forward, I must stress to the reader that Kubernetes is quite a complex platform. Attempting to explain in detail every feature and function of Kubernetes would take an entire book, if not longer. In fact, there has been a multitude of books written on container orchestration using Kubernetes that I would direct the reader's attention to should you would want to dig deeper into your understanding of these concepts. The point of this chapter is to introduce the reader to a basic understanding of what Kubernetes is, the primary functionality, and how the reader can quickly get started using it to optimize the deployment of containers. There is a lot more to be said about Kubernetes than the scope of this book has the time to go into, so if the reader wants to learn more about Kubernetes, I would strongly suggest checking out the documentation on the Kubernetes website at `https://kubernetes.io/docs`.

Throughout the book so far, we have looked at using Ansible Container to build Docker containers that run on our local workstation or a remote server that has Docker installed and running on it. Docker provides us with a usable container runtime environment that has the functionality to start containers, expose ports, mount system volumes, and provide basic network connectivity using a bridged interface and IP Network Address Translation, or NAT. Docker does a very good job at running containers but does not provide the user with very much functionality beyond that. What happens when your container crashes? What do you do when you need to scale out your application to more nodes, racks, or data centers? What happens when a container on one host needs to access resources in a container running on a separate host? This is the exact type of use case that tools such as Kubernetes address. Think of Kubernetes essentially as a service

that uses a scheduler and API to proactively monitor the current state of containers running in Docker (or another container runtime) and continuously attempts to drive it towards the desired state specified by the operator. For example, say you have a 4-node Kubernetes cluster running 3 instances of your application container. If the operator (you) instructs the Kubernetes API that you want the fourth instance of your application container running, Kubernetes will identify that you currently have three running instances and immediately schedule a fourth container to be created. Using a bin-packing algorithm, Kubernetes intrinsically understands that containers should be scheduled to run on separate hosts to provide high availability and make the most use of cluster resources. In the example above, the fourth container scheduled will most likely be scheduled to run on the fourth cluster host, provided no outstanding configuration has been put into place that would prevent new container workloads from running on that host. Furthermore, if one of the hosts in our cluster goes down, Kubernetes is intelligent enough to recognize the disparity and reschedule those workloads to run on different hosts until the downed node has been restored.

In addition to the flexible configuration management capabilities Kubernetes provides, it is also known for its unique ability to provide resilient networking resources to containers such as service discovery, DNS resolution, and load balancing across containers. In other words, Kubernetes has the innate ability to provide internal DNS resolution based on the services running in the cluster. When new pods are added to the service, Kubernetes will automatically see the new containers and update the DNS endpoints so that the new containers can be served by calling the internal service domain name within the cluster. This ensures that other containers can talk directly to other containerized services by calling internal domain names and cluster IP addresses within the Kubernetes overlay network.

Kubernetes incorporates many new concepts that might be somewhat foreign if you come from a background of working with static container deployments. Throughout this chapter, these concepts will be referred to as we learn more about Kubernetes, so it is important to have a grasp of what these terms mean as we go forward:

- **Pod**: a pod represents one or more application containers running in the Kubernetes cluster. By default, a pod definition names at least one container

that the user wishes to run in the cluster, including any additional environment variables, command or entrypoint configuration the user wants the pod to run with. If the pod includes more than one container definition, all containers running in the pod share the pod network and storage resources. For example, you could run a pod that consisted of a web server container as well a caching server. From the perspective of the pod, the web server might run on the localhost port 80, and the cache would likewise run on localhost port `11211`. From the perspective of Kubernetes, the pod itself would have a single IP address internal to the cluster the services would be exposed on, but in reality, would consist of two entirely separate containers.

- **Deployment**: A deployment is an object in Kubernetes that defines pods which will be running in the cluster. Deployments consist of a variety of parameters, such as the name of the container image, volume mounts, and the number of replicas to run. In order to delete pods from a Kubernetes cluster, the deployment must be deleted. If you simply attempt to delete pods, you will see that Kubernetes attempts to recreate those pods. This happens due to the fact that the deployment object is informing the cluster that those pods should be running, and the controller manager (more on this later) will attempt to bring the cluster back into the desired state.
- **Labels**: Labels are key-value pairs that can be assigned to almost any object in Kubernetes. Labels can be assigned to objects to organize subsets of resources in the cluster. For example, if you have a cluster that runs multiple deployments of the same pods, they can be labeled differently to indicate different uses. Labels can even be leveraged to by the scheduler to determine where and when pods should be running in across the cluster.
- **Service**: A service defines a logical subgrouping of pods (usually by a label selector) and the methods by which they should be accessed by other resources in the cluster. For example, you could create a service that exposes a set of pods to the outside world. A selector such as a label could be used to determine which pods should be exposed. Later, if pods are added or removed from the cluster, Kubernetes will automatically scale the service, provided the new pods are running with the same selector.

To make this functionality transparent, Kubernetes provides multiple services running in the cluster that work in conjunction to ensure that the cluster and applications are continuously in the desired state. Collectively, these services are known as the Kubernetes Control Plane. The control plane is what allows the

function, manage running containers, and maintain the state of nodes and resources across the cluster. Let's take a quick look at those now:

- **KubeCTL**: `kubectl`, (pronounced kube-control), is the command-line tool for interacting with Kubernetes. `Kubectl` gives you direct access to the Kubernetes API to schedule new deployments, interact with Pods, expose deployments, and more. The `kubectl` tool requires a set of credentials in order to access the Kubernetes API.
- **Kubernetes API Server**: The Kubernetes API server is responsible for accepting input from the operator, either from the `kubectl` command-line tool or by direct access to the API itself. The Kubernetes API is responsible for coordinating information to the rest of the cluster to execute the desired state. It should also be noted that the Kubernetes API server depends on the ETCD service to store and retrieve information about the cluster nodes and services running in the cluster.
- **Kubernetes Scheduler**: The Kubernetes scheduler is responsible for scheduling new workloads across the cluster nodes. Core to this responsibility is monitoring the cluster to ensure that available resources are present in the cluster to run pods, as well as ensuring that servers are available and reachable.
- **Kubernetes Controller Manager**: The controller manager is primarily concerned with desired state compliance across the cluster. The controller manager service interacts with the ETCD service and watches for new jobs and requests coming in through the API server. When a new request is received and stored in ETCD, the controller manager kicks off a new job in tandem with the scheduler to ensure that the cluster is in the desired state defined by the operator. The controller manager accomplishes this by using control loops to continuously monitor the state of the cluster and immediately correct any discrepancies it sees between the current and desired state. When you delete a Kubernetes pod and a new one automatically gets created, you have the controller manager to thank for that.
- **ETCD**: ETCD is a distributed key-value store created by CoreOS, which is used to store configuration information across the Kubernetes cluster. As stated previously, ETCD is primarily written to by the Kubernetes API server.
- **Container Networking Interface**: The Container Network Interface

project, or CNI, attempts to bring additional network functionality then what comes out of the box with Kubernetes. CNI provides interfaces and plugin support to allow various network plugins to be deployed within Kubernetes clusters. This allows Kubernetes to provide overlay network connectivity to containers distributed across hosts so that containers do not have to rely on the relatively limited networking space provided on the Kubernetes hosts. Common third-party plugins that implement the CNI standard are Flannel, Weave, and Calico.

- **Kubelet**: Kubelet is a service which runs on every host in the Kubernetes cluster. The Kubelet's primary responsibility is to leverage the underlying container runtime (Docker or rkt) to create and manage pods on cluster nodes according to the instructions received by the API, the scheduler, and controller manager. The Kubelet service does not manage containers or pods running on the host that were not created by Kubernetes. Think of the Kubelet as the translation layer between Docker and Kubernetes.

Now that we have an understanding of the Kubernetes platform and how it works, we can start using Kubernetes to run some of the containers we built earlier in this book.

Getting started with the Google Cloud platform

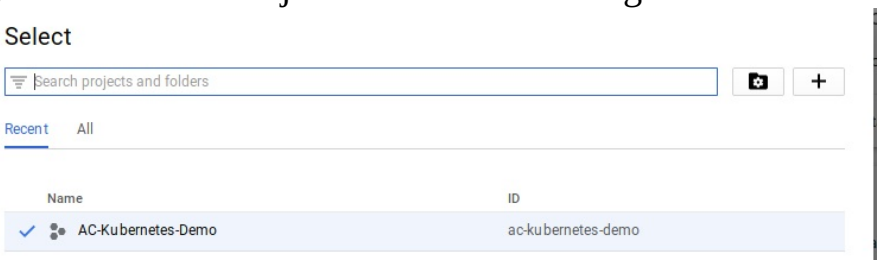
Throughout the many chapters in this book, we have worked primarily in a single-node Vagrant lab that comes preloaded with most of the tools and utilities you need to get started using Docker and Ansible Container to initialize, build, run, and destroy containers through the various examples and lab exercises. Unfortunately, due to the complexity of Kubernetes, it is very difficult to run a Kubernetes environment within the Vagrant lab we have used so far. There are methods, but they would require more computing power and explanation that extends beyond the scope of this book. As a solution, I would suggest that the reader signs up for a free-tier account on Google Cloud Platform to quickly spin up a three-node Kubernetes cluster in only a few minutes, which can be used using `kubectl` command-line agent from the single-node Vagrant lab. At the time of writing, Google is offering a free $300.00 credit for signing up for a free-tier Google Cloud account. Once the $300.00 allowance has expired, Google will not charge you for further use without explicit authorization. In and of itself, this is more than enough to run our simple cluster and cover many of the major Kubernetes concepts.

If you are unable to sign up for a Google Cloud Platform account, you can spin up a local Kubernetes node on your workstation absolutely free of charge using the Minikube project. Configuring Minikube to work on your laptop with proper reachability for `kubectl` commands to work is fairly straightforward if you are using the Virtualbox hypervisor. If you are interested, you can find more information on Minikube at `https://github.com/kubernetes/minikube`.

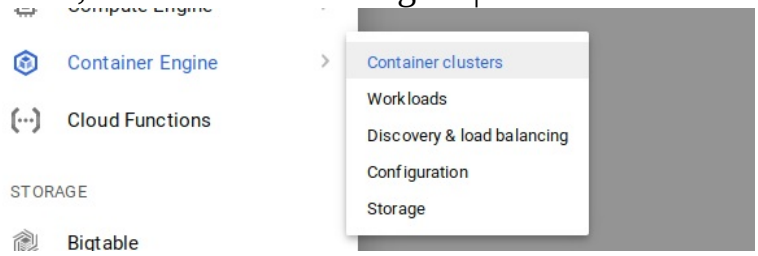
Before we can proceed with creating our Google Cloud Kubernetes cluster, we need to first sign up for an account at `https://cloud.google.com/free/`.

Once you have created a free account, it will prompt you to create a new project. You can name your project anything you like, as Google Cloud will assign a unique identifier to it within the console. I named mine `AC-Kubernetes-Demo`. If the signup process does not prompt you to create a new project, from the main

console you can select: Projects and click the + sign button to create a new project:



Once a project has been created, we can create a Kubernetes cluster using the Google Container Engine. From the main console window, on the left-hand side menu, select Container Engine | Container Clusters from the submenu:



For the purposes of this example, and also to make the most of the free allowance provided to use the Google Container Engine, we will create a three-node container cluster using the minimum specifications. To do this, from the Container clusters dashboard, select the button Create Cluster. This will drop you into a form that will allow you to select your cluster specifications. I created mine to the following specifications:

- Name: Cluster-1
- Cluster Version: 1.6.9
- 1 vCPU per Node (3 total vCPUs)
- Container Optimized OS
- Disabled Automatic Upgrades

Once the cluster has been created, you should see a cluster that resembles the following screenshot:

The most recent versions of the Google Cloud interface may have changed since the time of writing. You may have to set up your

Kubernetes cluster using a slightly different set of steps, or customization options. The default settings should be sufficient to create a cluster that isn't so expensive that it quickly burns through your $300.00 allowance. Remember, the more resources you allocate to your cluster, the faster you will use up your credit!

Once our cluster has been fully deployed, we can connect to it from our Vagrant development lab. To do this, we need to first initialize the `kubectl` tool using the `Gcloud` interface. By default, these packages are not installed in the Vagrant lab to save on time and complexity when provisioning the VM. To enable this functionality, we need to modify the Vagrantfile, located in the `root` directory of the official Git repository for this book. Towards the bottom of the Vagrant file, you will see a section titled: `#Un-Comment this section to Install the Google Cloud SDK`. Un-commenting this section should result in the following changes to the Vagrantfile: **##Un-Comment this to Install the Google Cloud SDK:**
export CLOUD_SDK_REPO="cloud-sdk-$(lsb_release -c -s)"
echo "deb http://packages.cloud.google.com/apt $CLOUD_SDK_REPO main" | sudo tee -a /etc/apt/sources.list.d/google-cloud-sdk.list
curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
apt-get update && sudo apt-get install -y google-cloud-sdk
apt-get install kubectl
SHELL
end

After making these changes, save the file, and launch the lab VM using the `vagrant up` command. If the lab VM is already running, you can use the `vagrant provision` command to re-provision the running VM, or simply destroy and re-create the VM as follows: **user@localhost:~/$ vagrant destroy -f**
==> node01: Forcing shutdown of VM...
==> node01: Destroying VM and associated drives...

user@localhost:~/$ vagrant up
Bringing machine 'node01' up with 'virtualbox' provider...
==> node01: Checking if box 'ubuntu/xenial64' is up to date..

Once the Vagrant lab VM has the `Google Cloud SDK` and `kubectl` installed, Execute the command `gcloud init` and, when prompted to log in, enter `Y` to confirm and

continue logging in.

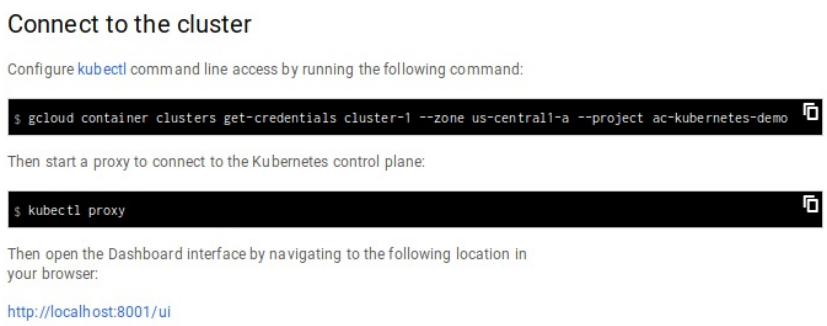
```
ubuntu@node01:~$ gcloud init
Welcome! This command will take you through the configuration of gcloud.

Your current configuration has been set to: [default]

You can skip diagnostics next time by using the following flag:
  gcloud init --skip-diagnostics

Network diagnostic detects and fixes local network connection issues.
Checking network connection...done.
Reachability Check passed.
Network diagnostic (1/1 checks) passed.

You must log in to continue. Would you like to log in (Y/n)? Y
```

The `Gcloud` CLI tool should then return a hyperlink that will allow you to authorize your Vagrant lab with your Google Cloud account. Once you have granted permission to use your Google Cloud account, your web browser should return a code you can enter on the command line to complete the authorization process.

The CLI wizard should then prompt you to select a project. The project you just created should be displayed with a list of options. Select the project we just created: **Pick cloud project to use: [1] ac-kubernetes-demo [2] api-project-815655054520 [3] Create a new project Please enter numeric choice or text value (must exactly match list item): 1**

It will then prompt you if you wish to configure the Google Compute Engine. This is not a necessary step, but if you opt to perform it, you will be presented with a list of geographic regions to select from. Select the one closest to you. Finally, your Google Cloud account should be connected to your Vagrant lab.

Now, we can set up connectivity to our Kubernetes cluster using the `kubectl` tool. This can be accomplished by selecting the Connect button on the Container Engine dashboard, next to our cluster. A screen should pop up displaying details on how to connect to our cluster from our initialized Vagrant lab:



Copy and paste that command into your Vagrant environment: **ubuntu@node01:~$ gcloud container clusters get-credentials cluster-1 --zone us-central1-a --project ac-kubernetes-demo WARNING: Accessing a Container Engine cluster requires the kubernetes commandline client [kubectl]. To install, run $ gcloud components install kubectl Fetching cluster endpoint and auth data. kubeconfig entry generated for cluster-1.**

This should cache the default Kubernetes credentials required for access to our cluster from the `kubectl` command-line tool. `kubectl` will already be installed in the Vagrant lab VM due to the changes made to the Vagrantfile earlier in the chapter.

Since `kubectl` is already installed, we can validate the connectivity to your Kubernetes cluster by executing `kubectl cluster-info` to view details about our running cluster. I censored the IP details for my cluster environment. However, your output will show all the relevant addresses for the core services: **ubuntu@node01:~$ kubectl cluster-info Kubernetes master is running at https://IPADDRESS GLBCDefaultBackend is running at https://IPADDRESS/api/v1/namespaces/kube-system/services/default-http-backend/proxy Heapster is running at https://IPADDRESS/api/v1/namespaces/kube-system/services/heapster/proxy KubeDNS is running at https://IPADDRESS/api/v1/namespaces/kube-system/services/kube-dns/proxy kubernetes-dashboard is running at https://IPADDRESS/api/v1/namespaces/kube-system/services/kubernetes-dashboard/proxy**

You can also run `kubectl get nodes` to see an output of the nodes the cluster consists of: **ubuntu@node01:~$ kubectl get nodes NAME STATUS AGE VERSION gke-cluster-1-default-pool-ca63b897-7pwx Ready 2d v1.6.9 gke-**

cluster-1-default-pool-ca63b897-d9cf Ready 2d v1.6.9 gke-cluster-1-default-pool-ca63b897-fnnt Ready 2d v1.6.9

Deploying an application in Kubernetes using kubectl

KubeCTL or Kube Control is the official command line interface into the Kubernetes API Server and the rest of the Kubernetes Control Plane. Using the `kubectl` tool, you can view the status of pods, access cluster resources, and even exec into running pods for troubleshooting purposes. In this portion of the chapter, we will look at the basics of using `kubectl` to manually describe deployments, scale pods, and create services to access the pods. This is beneficial to understanding the basic concepts of Kubernetes to understand how Ansible Container is able to automate many of these tasks using the native Kubernetes modules available to it.

Let's take a look at some of the most common `kubectl` options and syntax you are likely to run into working with Kubernetes:

- `kubectl get`: `kubectl get` is used to return resources that currently exist in the Kubernetes cluster. Commonly, this command is used to get a list of pods currently running or nodes in the cluster. Think of this command as being similar to the docker ps command. Examples of `get` commands are: `kubectl get pods` and `kubectl get deployments`.
- `kubectl describe`: `describe` is used to view more verbose details about a particular cluster resource. If you want to know the latest state of a resource or current details about how the resource is running you can use the describe command. `describe` is very helpful since you can call out a specific cluster resource, such as a pod, service, deployment, or replication controller to view the details pertaining directly to that instance. `describe` is also very useful for troubleshooting issues across Kubernetes environments. Examples of `describe` are: `kubectl describe pod`, and `kubectl describe node`.
- `kubectl run`: `kubectl run` functions quite similar to the `docker run` command we explored earlier in this book. Run is primarily used to start a new deployment and get pods up and running quickly in the Kubernetes cluster. The use case for run is rather limited, since more complex deployments are better suited for the `create` and `apply` commands. However, for testing or

getting containers running quickly and efficiently, `run` is a fantastic tool.

- `kubectl create`: Create is used to create new cluster resources such as pods, deployments, namespaces, or services. Create functions very similar to apply and run, with the caveat that it is used solely for launching new objects. Using create, you can use the `-f` flag to pass in a manifest file or direct URL to launch more complex items than you could with `kubectl run`.
- `kubectl apply`: Apply is often confused with `create` since the syntax and functionality is so similar. Apply is used to update Kubernetes resources that exist in the cluster, whereas `create` is used to create new resources. For example, if you created a series of pods based on a Kubernetes manifest that you launched using `kubectl` create, you could use `kubectl` apply to update any changes you may have made to the manifests. The Kubernetes Control Plane will analyze the manifest an attempt to make the changes necessary to bring the cluster resources into the desired state.

- `kubectl delete: delete` is rather self-explanatory since the primary function is used to delete objects from the Kubernetes cluster. Similar to create and apply, you can use the `-f` flag to pass in a Kubernetes manifest file that was created or updated previously and use that as an identifier to delete those resources from the cluster.

As you will notice, the `kubectl` uses a verb/noun syntax that is quite easy to remember. Everything you do with `kubectl` will take a verb argument (get, describe, create, apply), followed by the objects in kubernetes you wish to act on: (pods, namespaces, nodes, and other specific resources). There are a lot more command options available to you using the `kubectl` tool, but these are by far some of the most common options you are likely to use when starting with Kubernetes.

To view all of the possible parameters that Kubectl takes, you can use `kubectl --help or kubectl` subcommand `--help` to get help on a particular function or subcommand.

Since we now have access to the Kubernetes cluster from our Vagrant lab, we can use the `kubectl` tool to explore the cluster resources and objects. The first command that we will look at is the `kubectl get pods` command. We will use this to return a list of pods that exist in all namespaces across the cluster. Simply typing in `kubectl get pods` will return nothing since Kubernetes resources are separated by

namespaces. Namespaces provide a logical separation of Kubernetes resources based on DNS and networking rules, which allow users to have fine-grained control over multiple deployments that simultaneously exist in the same cluster. Currently, everything that exists in the Kubernetes cluster exists as running processes critical to the functionality of the Kubernetes control plane and exist in the `kube-system` namespace. To see a list of everything running in all namespaces, we can use the `kubectl get pods` command, passing in the `--all-namespaces` flag:
ubuntu@node01:~$ kubectl get pods --all-namespaces NAME READY STATUS RESTARTS AGE fluentd-gcp-v2.0-k8nrl 2/2 Running 0 17m fluentd-gcp-v2.0-l05dw 2/2 Running 0 17m fluentd-gcp-v2.0-svnfw 2/2 Running 0 17m heapster-v1.3.0-1288166888-cqpd3 2/2 Running 0 16m kube-dns-3664836949-sl69q 3/3 Running 0 17m kube-dns-3664836949-tbmvl 3/3 Running 0 17m kube-dns-autoscaler-2667913178-vdjc5 1/1 Running 0 17m kube-proxy-gke-cluster-1-default-pool-ca63b897-7pwx 1/1 Running 0 17m kube-proxy-gke-cluster-1-default-pool-ca63b897-d9cf 1/1 Running 0 17m kube-proxy-gke-cluster-1-default-pool-ca63b897-fnnt 1/1 Running 0 17m kubernetes-dashboard-2917854236-sctqd 1/1 Running 0 17m l7-default-backend-1044750973-68fx0 1/1 Running 0 17m

Your list may look slightly different to the output I have provided here, based on the version of Kubernetes your cluster is running and any changes the Google Container Engine platform may have introduced since the time of writing. However, what you will see is a list of containers that are running to support the Kubernetes Control Plane, such as the `kube-proxy`, `kube-dns`, and logging mechanisms using `fluentd`. The default output will show the name of the pods, how long they have been running (the age), the number of running replicas, and the number of times the pods have restarted.

You can use the `-o wide` flag to see more details, such as the namespace and overlay network IP addresses assigned to the pods. For example, `kubectl get pods -o wide --all-namespaces`.

Now that we have a firm understanding of listing pods, we can use the `kubectl run` command to start our first deployment. In `Chapter 3`, *Your First Ansible Container Project* we learned how to build an NGINX container using a community-developed container-enabled role and uploaded it to our personal Docker Hub repository. We can use the `kubectl run` command to download our container, quickly create a new Kubernetes deployment called `nginx-web` and get this pod

running in our cluster. In order to pull the pod from our repository, we will need to provide the fully qualified container name in the format: `image-repository/username/containername`. Furthermore, we need to map the port to port `8000` since the community-developed role leveraged that port by default. Finally, we will be launching this deployment in the `default` namespace, so no additional namespace configuration needs to be applied: **kubectl run nginx-web --image=docker.io/username/nginx_demo-webserver --port=8000**

Now, if we try running `kubectl get pods`, we will see a single NGINX pod running the default namespace: **ubuntu@node01:~$ kubectl get pods -o wide NAME READY STATUS RESTARTS AGE nginx-web-1202329523-qjkwp 1/1 Running 0 3m**

Similarly, we can use the `kubctl get deployments` function to see what the current state of deployments for the default namespace looks like: **ubuntu@node01:~$ kubectl get deployments NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE nginx-web 1 1 1 1 12m**

As you can see from the get pods and get deployments output, we have a single deployment called `nginx-web`, which consists of a single pod and a single container within that pod. This is in full agreement with the input we provided into the Kubernetes API server using the `kubectl run` command. If we attempt to delete this pod, there will be a delta between the desired state and the current status of our deployment. Kubernetes will then attempt to bring the cluster back into the desired state by recreating the deleted cluster resource. Let’s try doing a delete on the NGINX pod we created and see what happens. Usually, this happens so quickly, you will need to pay attention to the name of the pod as well as the age to see that the change has occurred: **ubuntu@node01:~$ kubectl delete pod nginx-web-1202329523-qjkwp pod "nginx-web-498735019-6llvp" deleted ubuntu@node01:~$ kubectl get pods NAME READY STATUS RESTARTS AGE nginx-web-498735019-xcp21 1/1 Running 0 15s**

If we wanted to actually delete the pods from the cluster permanently, we could use the delete command on the deployment itself, using the syntax: `kubectl delete deployment nginx-web` This would declare a new desired state, namely that we no longer want the deployment `nginx-web` present and all pods in that deployment should likewise be deleted.

Describing Kubernetes resources

Kubernetes can also be used to view detailed information about the pods or other objects we have instantiated in our cluster. We can do this using the `kubectl describe` command. Describe can be used to see a detailed view of almost any resource in our cluster.

Let’s take a moment to describe our NGINX pod and ensure that it is running as expected: **ubuntu@node01:~$ kubectl describe deployment nginx-web Name: nginx-web Namespace: default CreationTimestamp: Wed, 13 Sep 2017 19:36:48 +0000**

Labels: run=nginx-web Annotations: deployment.kubernetes.io/revision=1

Selector: run=nginx-web Replicas: 1 desired | 1 updated | 1 total | 1 available | 0 unavailable StrategyType: RollingUpdate MinReadySeconds: 0

RollingUpdateStrategy: 1 max unavailable, 1 max surge Pod Template:

Labels: run=nginx-web

Containers:

nginx-web:

Image: docker.io/aric49/nginx_demo-webserver Port: 8000/TCP

Environment: <none> Mounts: <none> Volumes: <none> Conditions:

Type Status Reason ---- ------ ------

Available True MinimumReplicasAvailable OldReplicaSets: <none>

NewReplicaSet: nginx-web-498735019 (1/1 replicas created) Events:

FirstSeen LastSeen Count From SubObjectPath Type Reason Message ---------- -------- ----- ---- ------------- -------- ------ -------

59s 59s 1 deployment-controller Normal ScalingReplicaSet Scaled up replica set nginx-web-498735019 to 1

As you can see describe displays a lot of pertinent information about our cluster, including details such as the namespace the pod is running in, any labels our pod is configured with, the name and location of the container image that is running, as well as the most recent events that have occurred to our pod. The describe output shows us a wealth of information that can help us to troubleshoot or optimize the deployments and containers in our cluster.

Exposing Kubernetes services

Since we now have a functional NGINX web server running in our cluster, we can expose this service to the outside world so that others can use our shiny new service. In order to do this, we can create a Kubernetes abstraction known as a service to control how our pod is granted outside access. By default, Kubernetes pods are assigned a cluster IP address by the overlay network fabric, which is only reachable within the cluster by other containers and across nodes. This is useful if you have a deployment that should never be directly exposed to the outside world. However, Kubernetes also supports exposing deployments using the service abstraction. Services can expose pods in a variety of ways, from allocating publicly routable IP addresses to the services and load balancing across the cluster to opening a simple node port on the master nodes, from which the service can listen. Google Container Engine provides native support for the `LoadBalancer` service type which can allocate a public virtual IP address to our deployment, making services extremely easy to expose. In order to allow our service to see the outside world, we can use the `kubectl expose deployment` command, providing the service type as `LoadBalancer`. Upon successful completion, you should see the message service `nginx-web` exposed.

```
ubuntu@node01:~$ kubectl expose deployment nginx-web –type=LoadBalancer
service "nginx-web" exposed
```

We can see our newly created service by running the `kubectl get services` command. You may notice that the `EXTERNAL IP` column may be in the pending state for a moment or two while Kubernetes allocates a public IP for the cluster. If you execute the `kubectl get services` command after a few minutes, you should notice it has an external IP and is ready to be accessed:

```
ubuntu@node01:~$ kubectl get services
NAME         CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
kubernetes   10.11.240.1     <none>        443/TCP          31m
nginx-web    10.11.255.240   <pending>     8000:32567/TCP   6s
```

After a minute or two:

```
ubuntu@node01:~$ kubectl get services
NAME         CLUSTER-IP      EXTERNAL-IP     PORT(S)          AGE
kubernetes   10.11.240.1     <none>          443/TCP          1h
nginx-web    10.11.241.144   35.202.165.54   8000:32567/TCP   1m
```

In this example, the external IP of `35.202.165.54` has been allocated to our deployment. You can access this IP address in a web browser to actually see the NGINX default web page in action. Remember, you have to access this service on TCP port `8000` since that is how the container-enabled role is configured out of the box. Bonus points if you want to go back and reconfigure your NGINX container to run on port `80`!

Google Cloud Platform has native integration with the Google Cloud virtual load balancer resources, which allow Kubernetes to assign external IP addresses to services. In baremetal environments or clusters running on other clouds, an extra configuration will be required to allow Kubernetes to seamlessly allocate publicly routed IP addresses.

Scaling Kubernetes pods

Now that we have pods running in our cluster, we can use the powerful Kubernetes primitives to scale out containers and running services across nodes for high availability. As mentioned previously, as soon as a desire state is declared that involves running more than one replica of a pod, Kubernetes will apply a bin-packing algorithm to the deployment in an effort to determine which nodes the service will run on. If you declare the same number of replicas as you have nodes in your cluster, Kubernetes will run one replica on each node by default. If you have more replicas declared then nodes, Kubernetes will run more than one replica on some of the nodes, and on others, it will run a single replica. This provides us with native high availability out of the box. One of the benefits of using Kubernetes is that, by leveraging these features and functionality, operators no longer worry as much about the underlying infrastructure the containers are running on as much as the cluster abstractions themselves.

NOTE: Kubernetes can also use labels to control where certain deployments should run. For example, if you have a high compute capacity node, you could label that node as a compute node. You can customize your deployment so that those pods will only run on nodes with that particular label.

To demonstrate how powerful of a functionality this is, we use `kubectl` to scale out our existing web server deployment. Since we are currently running a three-node cluster, let's scale out our NGINX deployment to four replicas. This way, we can best illustrate what decisions Kubernetes is making on where to place our containers. In order to scale our current deployment, we can use the `kubectl scale deployment` command to increase our replica count from one to four:
ubuntu@node01:~$ kubectl scale deployment nginx-web --replicas=4
deployment "nginx-web" scaled

Using `kubectl get deployments`, we can see that Kubernetes is actively reconfiguring our cluster towards the desired state. It might take a few seconds for Kubernetes to get all four pods running, depending on the configuration you have chosen for your cluster. Following we can see the desired number of pods, the current

number of pods running in the cluster, the number of pods that update, and the pods that are ready and available to accept traffic. It looks like our cluster is in our desired state: **ubuntu@node01:~$ kubectl get deployments NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE nginx-web 4 4 4 4 14m**

Running `kubectl get pods` with the `-o wide` flag, we can see that all four NGINX pods are running with different IP addresses allocated and on different cluster nodes. It is important to note that, since we specified four replicas and only have three nodes, Kubernetes made the decision to have two pod replicas running on the same host. Keep in mind that these are two separate and distinct pods with a different IP address, even though they are running the same host.

```
ubuntu@node01:~$ kubectl get pods -o wide
NAME   READY     STATUS    RESTARTS   AGE       IP          NODE
nginx  1/1       Running   0          2m        10.8.2.5    7pwx
nginx  1/1       Running   0          2m        10.8.1.10   fnnt
nginx  1/1       Running   0          15m       10.8.1.9    fnnt
nginx  1/1       Running   0          2m        10.8.0.5    d9cf
```

The preceding output is slightly truncated since the `-o` wide output is difficult to read properly in the context of a book page. Your output will be slightly more verbose than mine.

Accessing the public IP address again will result in the service now load balancing traffic across the pods in the cluster. Since we specified the service type as `LoadBalancer`, Kubernetes will use a round-robin algorithm to pass traffic to our pods with high availability. Unfortunately, this will not be obvious to the reader, since each pod is running the same NGINX test web page. One of the major benefits of Kubernetes is that services are usually tied to deployments. When you scale a deployment, the service will automatically scale and start passing traffic to the new pods!

Before we move forward to the next exercise, let’s delete the deployment we just created, as well as the exposed service. This will return our cluster to a fresh state:
ubuntu@node01:~$ kubectl delete deployment nginx-web deployment "nginx-web" deleted ubuntu@node01:~$ kubectl delete services nginx-web service "nginx-web" deleted

Creating deployments using Kubernetes manifests

Along with the ability to create services and other objects directly from the command line, Kubernetes also gives you the ability to describe desired states using a manifest document. Kubernetes manifests give you the freedom to provide more customization options in an easier to read, understand, and repeatable format, as opposed to the command line, which is rather limited in its format. Since this chapter is not designed to be a deep dive into Kubernetes, we will not spend a lot of time going into all of the various configuration options that can be used in a Kubernetes manifest. Rather, my intention is to show the reader what manifests look like and how they work at a basic level.

Since you are already familiar with creating deployments using the `kubectl` command-line tool, let's demonstrate what our `nginx-web` deployment would look like using a Kubernetes manifest. These examples are available in the official git repository for the book, under the `Kubernetes/nginx-demo` directory. Open your text editor and create a file: `webserver-deployment.yml`. The content of this file should resemble the following. In this example, we are going to continue to use our previously created NGINX container. However, feel free to use other container URLs if you wish to experiment with using other types of services and ports.

```
---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: webserver-deployment
spec:
  replicas: 4
  template:
    metadata:
      labels:
        app: http-webserver

    spec:
      containers:
        - name: webserver-container
          image: docker.io/USERNAME/nginx_demo-webserver
          ports:
            - containerPort: 8000
```

Like all the YAML documents we have looked at thus far, Kubernetes manifest

documents begin with three dashes to indicate the start of a YAML file. Kubernetes manifests always begin by specifying the API version, object kind, and metadata. This is colloquially known as the header data and indicates to the Kubernetes API the type of objects this document is going to create. Since we are creating a deployment, we will specify the `kind` parameter as `Deployment` and provide the name of the deployment as the metadata name. Everything listed underneath the `spec` section provides configuration option parameters that are specific to the pod object the document is creating. Since we are basically reverse engineering our previous deployment, we are specifying the number of replicas as `4`. The next few lines specify metadata we are going to configure our pods with, specifically a key-value pair label called, `app:http-webserver`. Keep this label in mind, as we are going to use it when we create the service resource next.

Finally, we have another `spec` section, which lists the containers that are going to run inside our pod. Earlier in the chapter, I mentioned that a pod can be one or more containers running using shared network and cluster resources. Containers in a pod share a pod IP address and localhost namespace. Kubernetes deployments allow you to specify more than one pod under the `containers:` section, passing them in as listed key-value pair items. This example, however, will create a single-pod container known as `webserver-container`. It is here that we will specify the container image version, as well as the container port (`80`).

This manifest can be applied using the `kubectl create` command, passing in the `-f` flag, which indicates a manifest object, as well as the path to the manifest:

```
kubectl create -f webserver-deployment.yml
```

Upon successful completion, you should see the pods getting created using `kubectl get pods`:

Creating services using Kubernetes manifests

In a similar way to creating our deployment using Kubernetes manifest, we can create other Kubernetes objects using manifests. The service we created earlier can be described using the following Kubernetes manifest: ---
apiVersion: v1
kind: Service
metadata:
name: webserver-service
spec:
type: LoadBalancer
selector:
app: http-webserver
ports:
- protocol: TCP
port: 80
targetPort: 8000

Notice in this example, we are specifying a different `kind` parameter to be `Service` as opposed to our previous example, which used `Deployment`. This tells the Kubernetes API to expect that the rest of the document will contain specifications that describe services instead of other types of Kubernetes objects. In the metadata section, we will name our service `webserver-service` (creative, no?). For the specification section, we will provide the type of service we are exposing, `LoadBalancer`, and provide the label we assigned to our deployment: `app: http-webserver`. When using `kubectl` to expose deployments, the service you create is inherently tied to the deployment you are exposing. When that deployment scales out, the service will be aware and will adjust according to how many backend pods are running. However, when creating a service using Kubernetes manifests, we can get more creative with how services are tied to the services they are exposing. In this example, we are creating a service that is associated with any pod that has the label `app: http-webserver`. In theory, this could be any number of pods across different namespaces and deployments. This allows for a

lot of flexibility when designing applications around a Kubernetes architecture.

The final section of our manifest describes the ports we will perform load balancing across. Remember how our NGINX container uses the fixed port `8000` by virtue of the fact we built this container using the community-written role? Using the load balancer service, we can expose any port we want on the frontend to forward traffic to any port on the backend pods. The protocol we will use will be TCP. The port we want to expose on the virtual IP address will be `80` for standard HTTP requests. Finally, we will list the port that NGINX is listening to internally on our pods to forward traffic to. In this case, `8000`.

Using the `kubectl create` command, we can create our service very similar to how we created our initial deployment:
ubuntu@node01:/vagrant/Kubernetes/nginx-demo$ kubectl create -f webserver-service.yml service "webserver-service" created

Using `kubectl get services`, we can see which external virtual IP address gets allocated to our cluster: **ubuntu@node01:$ kubectl get services NAME CLUSTER-IP EXTERNAL-IP PORT(S) webserver-service 10.11.251.238 104.197.85.153 80:30600/TCP**

Looking at the `PORTS` column, we can see that TCP port `80` is exposed in our cluster. Using a web browser again, we can access our new public IP on port `80` and see whether it's working:

Using Kubernetes manifests, we can describe in greater detail the ways we want our containerized applications to function. Manifests can easily be modified as well and reapplied using the `kubectl apply -f manifest.yml` command. If at any time, we wanted to update our application to a different version of the container

image, or modify exposed ports on the service, `kubectl` apply would only make the changes necessary to bring our application into the desired state. Feel free to tweak these manifests on your own and reapply them to see in what ways you can configure the services to run.

Next, we will look at deploying containers to Kubernetes using Ansible Container. Before we move forward, let's remove the pods in your Kubernetes cluster using the `kubectl delete` command, specifying the Kubernetes manifests we used to create or modify the deployment and service: **ubuntu@node01:$ kubectl delete -f webserver-deployment.yml deployment "webserver-deployment" deleted ubuntu@node01:$ kubectl delete -f webserver-service.yml service "webserver-service" deleted**

For now, we have finished working work the Kubernets cluster. If you wish to delete the cluster from Google Cloud, you can do so now. However, it is important to note that `Chapter 7`, *Deploying Your First Project* covers deploying projects to Kubernetes. I would suggest you keep your cluster active until you have finished working on the material in `Chapter 7`, *Deploying Your First Project*.

References

- **Kubernetes Documentation**: https://kubernetes.io/docs/home/
- **Google Cloud Platform**: https://cloud.google.com
- **Running Kubernetes locally with Minikube**: https://kubernetes.io/docs/getting-started-guides/minikube/

Summary

Kubernetes is quickly becoming one of the most robust, flexible, and popular container deployment and orchestration platforms that is taking the IT industry by storm. Throughout this chapter, we have taken a close look at Kubernetes, learning about how it works as a platform and some of the key features that make it so useful and versatile. If you have worked in or around containers for very long, it will be clear to you that Kubernetes is rapidly being adopted by organizations throughout the world due to the extremely sophisticated mechanisms it uses to deploy and manage containers at scale.

Due to the native support for Kubernetes in Ansible Container, we can use the same workflow to build, run, test, and destroy containerized applications that we can deploy to robust services such as Kubernetes. Ansible Container truly provides the right tools to help drive complex deployments using a unified and reliable framework.

However, Google Cloud and the Kubernetes framework are not the only cloud-based container orchestration solutions on the market in today's world. OpenShift is quickly gaining popularity as a managed solution built by Red Hat that functions on top of the Kubernetes platform. Next, we will apply the Kubernetes concepts we learned in this chapter to deploy applications to the OpenShift software stack, using the powerful tools offered to us by the Ansible Container platform to drive large-scale application workloads.

Managing Containers with OpenShift

Using a comprehensive suite of some of the best and most resilient open source tooling available, Kubernetes is rapidly changing the way that software applications are being built and deployed across organizations and in the cloud. Kubernetes brings with it lessons learned from deploying a containerized infrastructure across a company with one of the largest and most robust infrastructure footprints: Google. As we saw in the previous chapter, Kubernetes is an incredibly flexible and reliable platform for deploying containers at a very high scale, bringing with it the features and functionality to deploy highly available applications across clusters of servers, by running on top of native container engines such as `Docker`, `rkt`, and `Runc`. However, with the great power and flexibility Kubernetes brings, also comes great complexity. Arguably, one of the biggest downsides to deploying a containerized infrastructure using Kubernetes is the high degree of knowledge regarding the Kubernetes architecture that one must acquire prior to migrating workloads over to Kubernetes.

There is a solution to the high degree of overhead and technical complexity that puts Kubernetes out of the reach of many organizations today. In recent years, Red Hat has developed an answer to the problem of simplifying Kubernetes concepts and making the platform more accessible for software developers and DevOps engineers to quickly deploy and rapidly build upon. OpenShift is a suite of tools developed by Red Hat that runs on top of the Red Hat distribution of Kubernetes that provides a sophisticated, yet easy to understand platform for automating and simplifying the deployment of containerized applications. The aim of OpenShift is to provide a reliable and secure Kubernetes environment that provides users with a streamlined web interface and command-line tool used for deploying, scaling, and monitoring applications running in Kubernetes. Furthermore, OpenShift is the second of the major cloud providers currently supported by the Ansible Container project (Kubernetes and OpenShift).

In this chapter we will cover the following topics:

- What is OpenShift?
- Installing Minishift locally
- Deploying containers from the web interface
- OpenShift web interface tips
- An introduction to the OpenShift CLI
- OpenShift and Ansible Container

What is OpenShift?

OpenShift is a suite of products available from Red Hat for building a production-ready, reliable, and secure Kubernetes platform. Using OpenShift, developers have a tremendous amount of freedom when deploying containerized applications using the OpenShift API, or accessing the Kubernetes API to fine-tune functionality and features. Since OpenShift uses the same underlying container runtime environments, Docker containers can be developed locally and deployed to OpenShift, which leverages all of the Kubernetes primitives, such as namespaces, pods, and deployments, to expose services to the outside world. At the time of writing, Red Hat offers the OpenShift platform with the following configuration options:

- **OpenShift Origin**: A fully free and open source version of OpenShift that is community-supported. OpenShift Origin can be deployed locally using a project known as **Minishift**.
- **OpenShift Online**: OpenShift Online is a fully hosted public cloud offering from Red Hat that allows individuals and organizations to take advantage of OpenShift Origin without committing hardware resources to deploying OpenShift on-premise. Users can sign up for OpenShift online free-tier accounts that allow for application deployments up to 1 gigabyte of RAM, and two vCPUs.
- **OpenShift Dedicated/Container Platform**: OpenShift Dedicated and the OpenShift Container Platform provide robust and scalable deployments of OpenShift that are managed and supported by Red Hat either on-premises or through public cloud providers such as Google Cloud, Azure, or AWS.

Throughout the course of this chapter, and going forward into the next chapters, we will be using the fully free and open source OpenShift Origin to deploy a local Minishift cluster. Unlike the previous chapter, the free-tier version of OpenShift is unfortunately too limited to cover the breadth of examples this chapter is going to cover. In an effort to fully demonstrate the capabilities of OpenShift, I have opted to walk the user through a local installation of Minishift

that is only limited by the hardware running on your local workstation. If you have been tracking with us thus far, Minishift is not much more complicated to set up on VirtualBox than the local Vagrant lab environment we have been using. However, if you want to use the free tier of OpenShift Online, most of these examples can replicated there, albeit in a more limited way than running Minishift in your local environment.

Installing Minishift locally

Minishift is a local OpenShift cluster that can be downloaded and run on your local PC to function as a development environment. The primary use case for Minishift is to provide a sandbox that gives developers a functional development environment that can be launched on your laptop. Minishift is also fully compatible with the **OpenShift Client** (**OC**) CLI that is used to work with OpenShift clusters using a command-line interface. In this portion of the chapter, we will learn how to install Minishift and the OC in order to get it running in your local environment. Before proceeding, you need to have VirtualBox installed on your PC; it will function as a hypervisor for launching the Minishift VM. For the purposes of this demonstration, we will be using Minishift version 1.7.0. Since the time of writing, newer versions of Minishift will have undoubtedly been released. To have the best experience working with Minishift, I would suggest you download the 1.7.0 release, although newer releases will most likely work just as well.

Furthermore, Minishift and the OC are available cross-platform on Windows, Linux, and macOS. This example is going to demonstrate downloading and installing Minishift on Linux. For more information about installing Minishift on other platforms, I have provided the following link: `https://docs.openshift.org/latest/minishift/getting-started/installing.html`.

aric@local:~/minishift$ minishift stop
Stopping local OpenShift cluster...
Cluster stopped.

Deploying containers using the web interface

As you may have noticed, when the `minishift start` command completed, it provided an IP address that you can use to access the web user interface. One of the biggest benefits of using OpenShift over standard Kubernetes is that OpenShift exposes almost all of Kubernetes' core functionality through an intuitive web interface. The OpenShift console works in a similar way to other cloud or service dashboards you have used in the past. At a glance, you can see which deployments are running, triggered alarms caused by failed pods, or even new deployments that other users have triggered in the project. To access the web interface, simply copy and paste the IP address in the output of the `minishift start` command in any modern web browser installed on your local machine. You may then have to accept the self-signed SSL certificate that comes with Minishift by default, after which you will be prompted with a login screen similar to the following screenshot:

Figure 1: OpenShift login page The default credentials to access Minishift are the username `developer` and any password you want. It is not important to remember the password you enter, as each time you authenticate as the developer user, you can simply supply any password. Upon successfully logging in, you will be asked to access a project. The default project that Minishift provides for you is called My Project. For the sake of simplicity, we will use this project for the following lab examples, which you can follow along with.

The web interface is laid out by two primary navigation bars along the left and top of the screen, while the central portion of the interface is reserved for showing details about the environment you are currently accessing, modifying settings, or deleting resources:

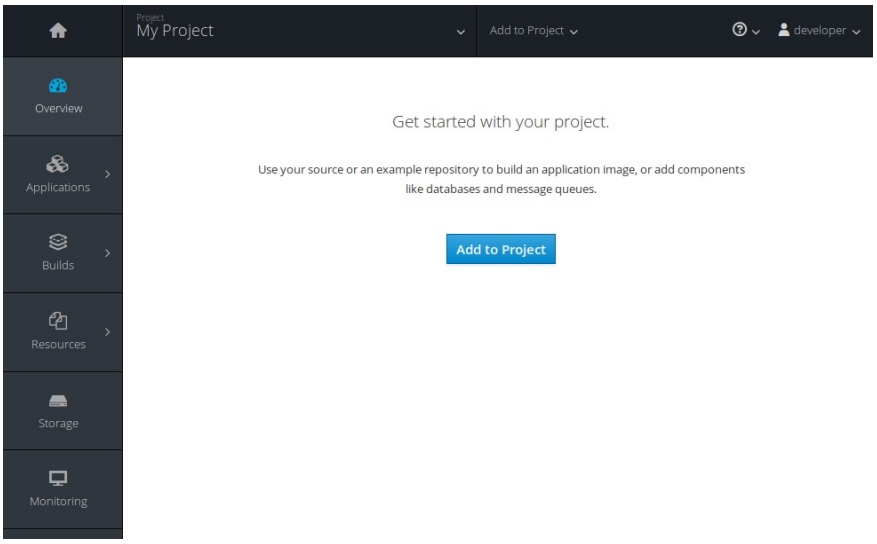


Figure 2: Initial OpenShift project Now that you are familiar with the OpenShift user interface, let's create a deployment and see what it looks like when pods are running in the cluster. The functionality for creating new pods and deployments can be found towards the top of the screen by selecting the Add to Project drop-down box:

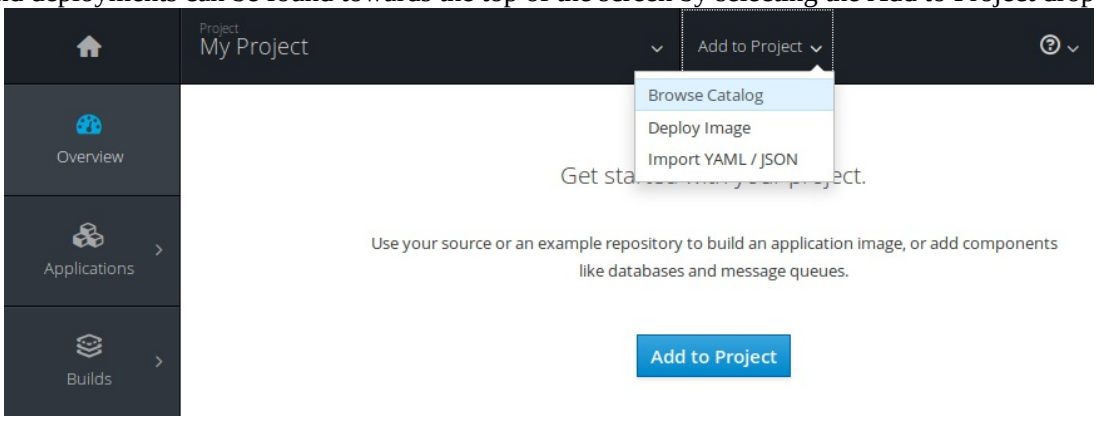


Figure 3: Adding services to your project We can create a new deployment in a variety of different ways. The default options OpenShift provides is to browse a catalog of pre-built images and services, deploy an image based on a container registry URL, or importing a YAML or JSON manifest that describes the services we are building. Let's deploy one of the catalog services found in the web interface. Selecting the Browse Catalog option from the Add to Project drop-down will open a screen for the OpenShift catalog. Any of the OpenShift examples found in the catalog will run well in OpenShift, but for demonstration purposes let's deploy the framework for a simple Python application. To do this, click on the catalog option

for Python, then the Python 3.5 Source Code Application:

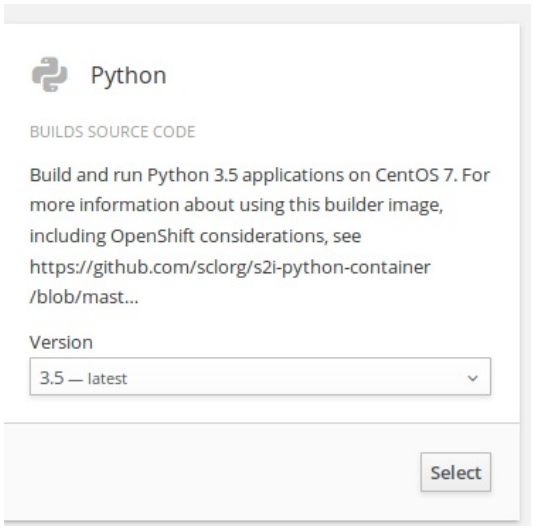


Figure 4: Creating a simple Python service from the OpenShift catalog On the next screen, OpenShift will prompt you for options to deploy your application with, namely a source code repository, which contains Python source code, and an application name. You can choose any name for the Python application. For this example, I will name mine `oc-test-deployment`. Since we do not have a pre-developed Python application, we can click on the Try It link below the Git Repository URL textbox to use the demo application provided by OpenShift:

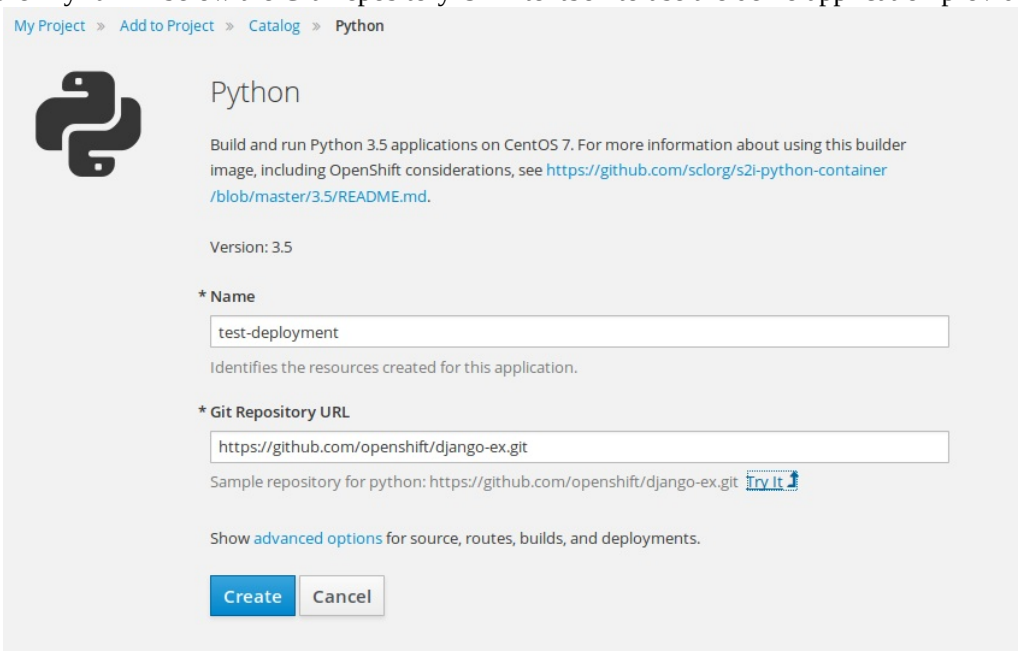


Figure 5: Modifying the attributes of the Python application If you are a Python developer and have a Django application you would like to deploy instead, feel free to use another Git repository in place of the demo one!

Clicking on the blue Create button will initiate the deployment and launch the container. Depending on the specifications of your workstation, it might take a while for the service to fully deploy. While it is deploying, you can watch the

various stages of the deployment by clicking through the pages of the user interface. For example, clicking on Pods in the sidebar will show pods as they are created and go through the various stages to become available in Kubernetes. OpenShift shows circular graphs that describe the state of the resources that are running. Pods that are healthy, responding to requests, and running as intended are shown by blue circles. Other Kubernetes resources that might not be running as intended, throwing errors or warnings instead, are represented by yellow or red circles. This provides an intuitive way to understand how the services are running at a glance:

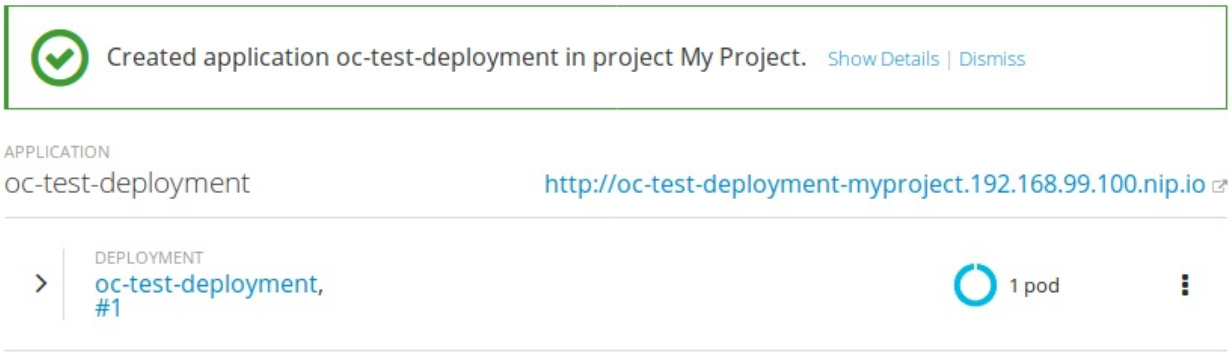


Figure 6: A successfully created test Python application Once the service has been deployed fully, OpenShift will provide a link to access the service. In OpenShift vernacular, this is known as a **route**. Routes function in a similar way to exposed services in Kubernetes, with the exception that they leverage `nip.io` DNS forwarding by default. You might notice that the service route pointing to the service we just created has the fully qualified domain name `servicename.ipaddress.nip.io`. This provides the user with routable access to reach the Kubernetes cluster without the hassle of configuring external load balancers or service proxies. Accessing that URL in a web browser should open a page that looks similar to the following:

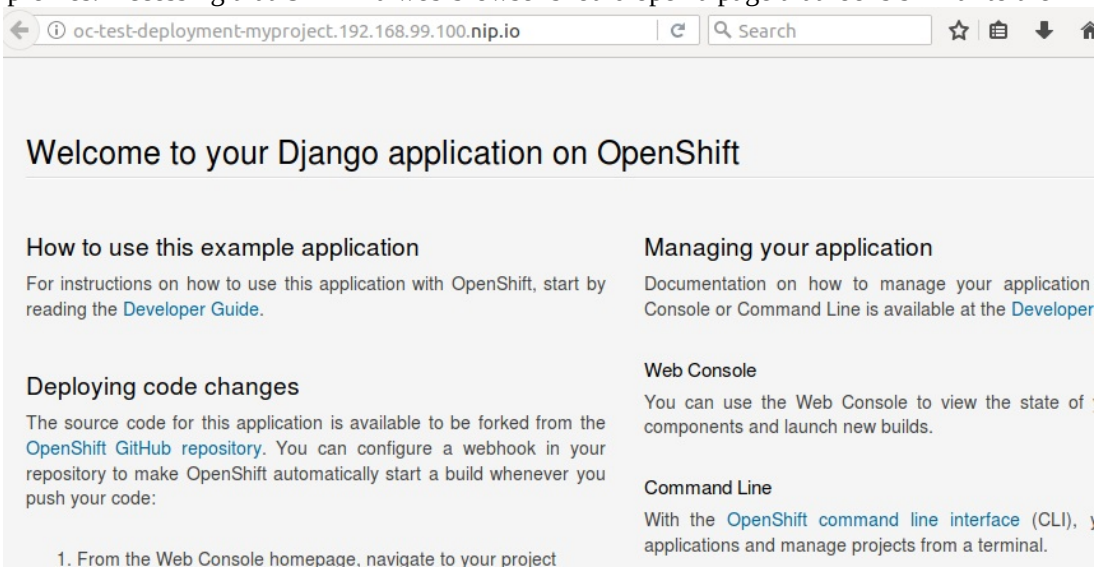


Figure 7: Python application test page This is the default Python-Django index page for this simple demo

application. If we click back on the OpenShift console, we can go into more detail regarding the pods that are running in this deployment. Similar to kubectl, OpenShift can give us details about the deployment, including running pods, log events, and even allow us to customize the deployment from the web user interface. To view these details, select Applications | Deployments and click on the deployment you wish to look up. In this case, we will look at the details of the only running deployment we have, `oc-test-deployment`:

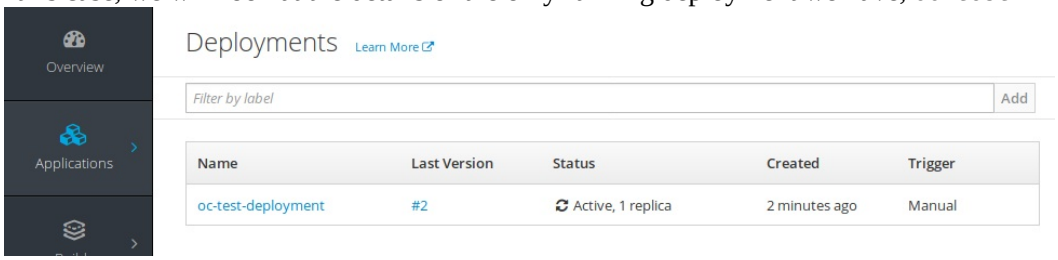


Figure 8: OpenShift deployments page From this page, we can view the history of containers, modify the configuration, check or change environment variables, and view the most recent event logs from the deployment. In the upper-right corner of the screen, the Actions drop-down box gives us even more options for adding storage, autoscaling rules, and even modifying the Kubernetes manifest used to deploy this service:

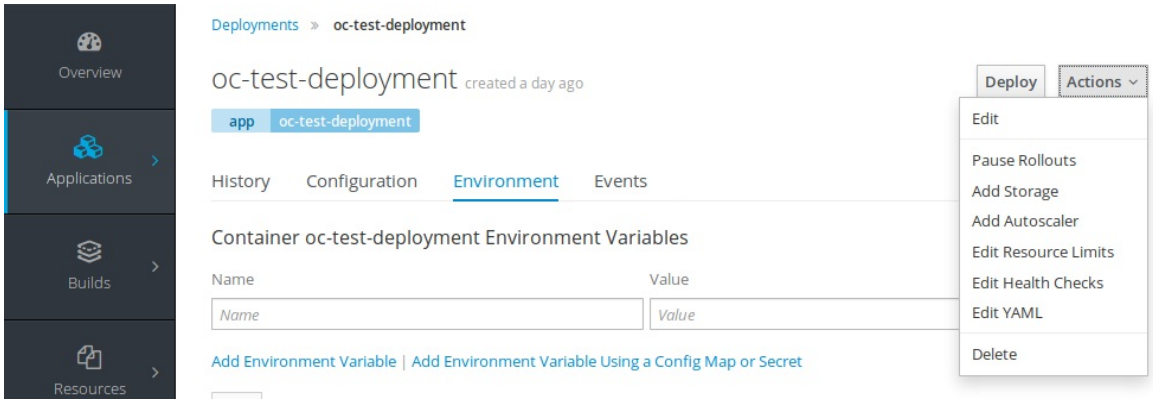


Figure 9: Managing OpenShift applications and deployments OpenShift provides a great interface for tweaking manifests and experimenting with changes in real time. The OpenShift user interface will give you feedback on changes you make and let you know when there are potential problems.

Information related to the running pods within the deployment can also be accessed through the web user interface. From the Applications menu, select Pods and click on the pod you wish to view information for. In this case, we have a single running pod, `oc-test-deployment-1-l18l8`:

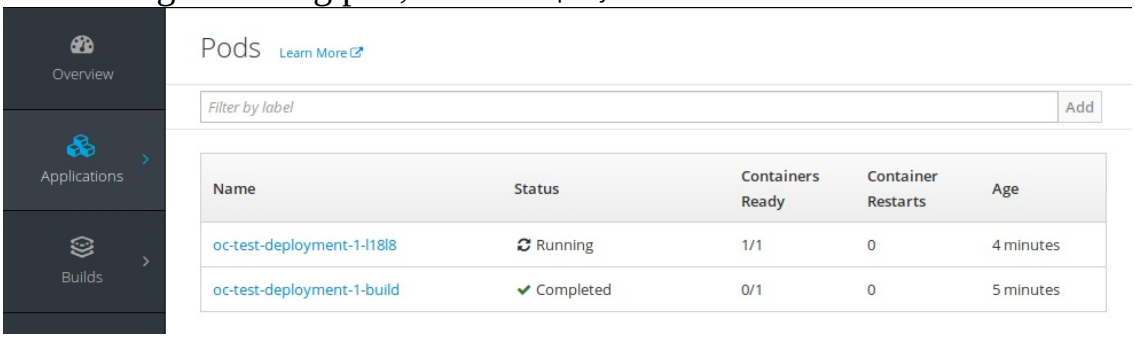


Figure 10: Viewing pods within application deployments On this page, we can view almost any pertinent detail regarding the pods that are running within any of our deployments. From this screen you can view environment configurations, access container logs in real time, and even log into containers through a fully functional terminal emulator:

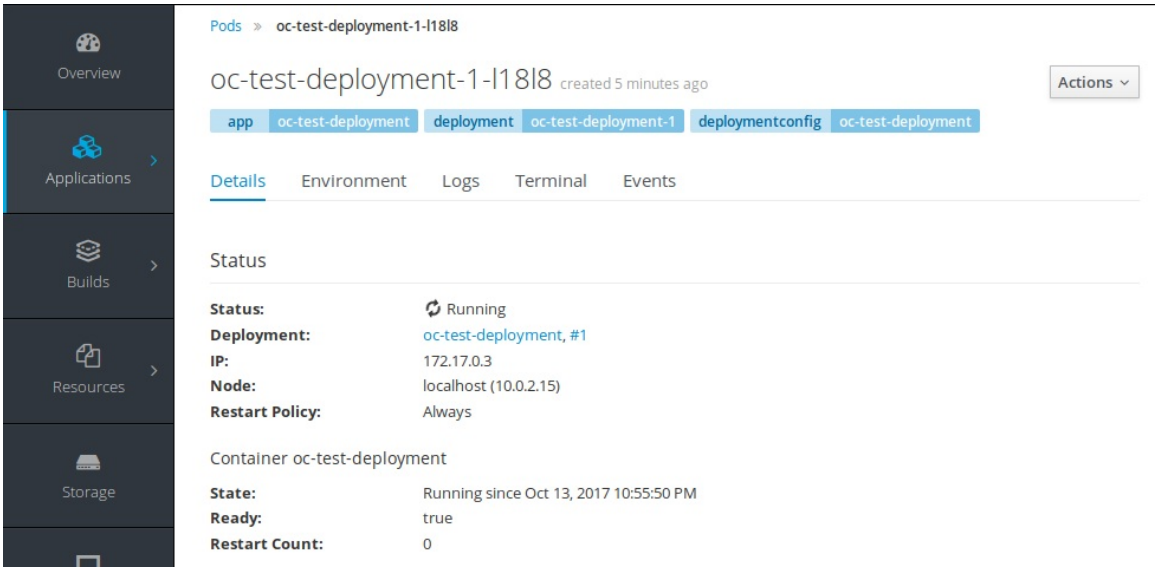


Figure 11: Viewing pod-specific details in OpenShift Similar to the Deployments menu, we can select the Actions drop-down menu from this screen as well to modify container settings in the YAML editor, or mount storage volumes inside the container.

Finally, using the OpenShift web interface, we can delete deployments and pods. Since OpenShift is essentially a layer that functions on top of Kubernetes, we need to apply many of the same concepts. In order to delete pods, we must first delete the deployment in order to set a new desired state within the Kubernetes API. Within OpenShift this can be accomplished by selecting Applications | Deployments | Your Deployment (oc-test-deployment). From the Actions drop-down menu, select Delete Deployment. OpenShift will prompt you to make sure this is something you really want to do; click Delete to finalize the operation:

Figure 12: Deleting deployments in OpenShift In a similar fashion, you will have to go to Applications | Service and then Applications | Routes in order to delete the services and routes that OpenShift created for

the service. Once this is complete, the screen produced by clicking on the Overview button in the left menu bar will once again be blank, showing that nothing is currently running in the OpenShift cluster.

OpenShift web user interface tips

The preceding example walked the user through some of the major OpenShift and Kubernetes workflow steps for creating a new deployment, managing the deployment, and ultimately deleting the deployment and other resources. OpenShift exposes far more functionality through the web user interface than this book has time to delve into; I suggest you take time to explore the web interface for yourself to truly become familiar with the features that OpenShift provides. For the sake of not being monotonous, I have provided a few key features to keep your eyes open for in the OpenShift web interface:

- **Overview dashboard**: The Overview dashboard can be accessed from the navigation bar on the left side of the screen. The overview dashboard shows information about the most recent activity inside the OpenShift cluster. This is useful for accessing the latest deployments and having single-click access to various cluster resources.
- **Applications menu**: The Applications menu is a single location to view or modify any deployments or pods that are running across the OpenShift cluster. From Applications, you can access information related to deployments, pods, stateful sets, services, and routes. Think of the Applications menu as a single stop for anything related to the configuration of containers running within the cluster.
- **Builds dashboard**: The Builds dashboard features a light **continuous integration continuous delivery** (**CICD**) workflow for Kubernetes. This is useful for triggering image builds, establishing Jenkins-enabled workflow pipelines, and building automation into OpenShift projects.
- **Resources menu**: The Resources menu is used primarily to define quotas and user account privileges used to manage access and limits for users and projects within the OpenShift cluster. Also defined here is a lightweight secret storage interface, as well as config map options to define the configurations for containers within OpenShift projects.
- **Storage dashboard**: The Storage dashboard is used to display information regarding persistent volume claims used by containers and deployments on

the underlying hardware or VM OpenShift is running on. Volume claims can be created or deleted from this portion of the web interface, as well as managed or modified depending on new or changing requirements.

- **Monitoring dashboard**: Finally, the Monitoring dashboard provides the user with details about running pods, triggered events, as well as the historical context regarding the changes in the environment leading up to the events listed. Monitoring can be easily tied to build a pipeline or even used to report on configured service health checks.

Leveraging the robust suite of tools provided by OpenShift helps to abstract and simplify many of the Kubernetes concepts we learned about in Chapter 5, *Containers at Scale with Kubernetes*.

An introduction to the OpenShift CLI

The second primary way that users can interact with the OpenShift platform is through the OpenShift command-line interface, OC (short for OpenShift Client). The OpenShift command-line interface uses many of the same concepts we explored in `Chapter 5`, *Containers at Scale with Kubernetes*, using the `kubectl` command line interface for managing pods, deployments, and exposing services. However, the OpenShift client also supports many of the features that are specific to OpenShift, such as creating routes to expose deployments and working with integrated security context constraints for access control. Throughout this section, we will look at how to use the OC to accomplish some of the basic workflow concepts we explored through the web user interface, such as creating deployments, creating routes, and working with running containers. Finally, we will look at some tips for diving deeper with the OC and some of the more advanced features that are available. Before proceeding, ensure the OC is installed (see the beginning of this chapter for installation details):

1. **Logging into OpenShift**: Similar to the web user interface, we can use the CLI to log into our local OpenShift cluster using the `oc login` command. The basic syntax for this command is `oc login URL:PORT`, where the user replaces the URL and port with the URL and port number of the OpenShift environment they are logging into. Upon successful login, the prompt will return `Login Successful` and grant you access to your default project. In this case, we will log into our local environment at `192.168.99.100`, using the `developer` username and anything as the password:

```
aric@local:~$ oc login https://192.168.99.100:8443


Authentication required for https://192.168.99.100:8443 (openshift)
```

Username: developer

Password:

Login successful.

You have one project on this server: "myproject"

Using project "myproject".

2. **Check status using OC status**: The `oc status` command is used in a similar way to the Overview dashboard in the web user interface to show critical services deployed in the environment, running pods, or anything in the cluster that might be triggering an alarm. Simply typing `oc status` will not return anything, since we deleted the deployments, routes, and services we created through the web user interface:

```
aric@local:~$ oc status
```

In project My Project (myproject) on server https://192.168.99.100

You have no services, deployment configs, or build configs.

Run 'oc new-app' to create an application.

3. **Create an OpenShift deployment**: Deployments and other cluster resources can easily be created using the `oc create` command. Similar to `kubectl`, you can create deployments by using the `oc create deployment` command and referencing the name of the container image you wish to use. It should be noted that deployment names are sensitive to using special characters such as underscores and dashes. For the purposes of simplicity, let's re-create our example from `Chapter 5`, *Containers at Scale with Kubernetes*, and create a simple NGINX pod using the official NGINX Docker image using the `oc create` command and specifying the object as `deployment`:

```
aric@local:~$ oc create deployment webserver --image=nginx
```

```
deployment "webserver" created
```

Another similarity to kubectl is that OpenShift supports creating deployments based on Kubernetes manifest files using the `-f` option.

4. **List pods and view the OC status**: Now that we have a deployment and pod running in the OpenShift cluster, we can view running pods using the `oc get pods` command, and check the output of the `oc status` command to see an overview of the running pods in our cluster:

```
aric@local:~/Development/minishift$ oc get pods
```

```
NAME                         READY     STATUS    RESTARTS   AGE
webserver-1266346274-m2jvd   1/1       Running   0          9m
```

aric@local:~/Development/minishift$ oc status

In project My Project (myproject) on server https://192.168.99.100:

pod/webserver-1266346274-m2jvd runs nginx

You have no services, deployment configs, or build configs.

Run 'oc new-app' to create an application.

5. **View verbose output using** `oc describe`: Aside from simply listing objects that are created and available in the OpenShift cluster, verbose details about specific objects can be viewed using the `oc describe` command. Similar to `kubectl describe`, `oc describe` allows us to view pertinent details about almost any object defined in the cluster. For example, we can use the `oc describe deployment` command to view verbose details about the web server deployment we just created:

```
aric@local:~/Development/minishift$ oc describe deployment webserver
```

Name: webserver

Namespace: myproject

CreationTimestamp: Sun, 15 Oct 2017 15:17:30 -0400

Labels: app=webserver

Annotations: deployment.kubernetes.io/revision=1

Selector: app=webserver

Replicas: 1 desired | 1 updated | 1 total | 1 available | 0 unav

StrategyType: RollingUpdate

MinReadySeconds: 0

RollingUpdateStrategy: 1 max unavailable, 1 max surge

Pod Template:

Labels: app=webserver

Containers:

nginx:

Image: **nginx**

Port:

Environment: **<none>**

Mounts: **<none>**
Volumes: **<none>**

Conditions:

Type **Status** **Reason**

---- **------** **------**

Available **True** **MinimumReplicasAvailable**

OldReplicaSets: <none>

```
NewReplicaSet:  webserver-1266346274 (1/1 replicas created)

Events:

  FirstSeen    LastSeen    Count   From                    SubObjectPat
  ---------    --------    -----   ----                    -------------   --------    --

  11m          11m         1       deployment-controller   Normal Scaling
```

6. **Create an OpenShift service**: In order to expose pods that are running in the OpenShift cluster, we must first create a service. You may remember from Chapter 5, *Containers at Scale with Kubernetes*, that Kubernetes services are abstractions that work in a similar way to internal load balancers inside of the Kubernetes cluster. Essentially, we are creating a single internal (or external) IP address from which traffic from other pods or services can access any number of pods matched to a given selector. In OpenShift, we will create a service simply called `webserver` that will use an internally routed cluster IP address to intercept web server traffic and forward it to the web server pod we created as apart of our deployment. By naming our service `webserver`, it will by default use a selector criteria that matches the label `app=webserver`. This label was created by default when we created the deployment in OpenShift. Any number of labels or selector criteria can be created, which allows Kubernetes and OpenShift to select pods to load-balance traffic against. For the purposes of this example, we will use an internal cluster IP and map the selector criteria based on naming our service with the same name we named our deployment. Finally, we will select the ports we want to forward from our service externally, to the ports the service is listening on inside the container. To keep things simple, we will forward traffic destined to port `80` to the pod port `80`:

```
aric@local:~$ oc create service clusterip webserver --tcp=80:80
```

```
service "webserver" created
```

We can check the service configuration using the `oc get services` command. We can see that our service was created with an internally routed cluster address of `172.30.136.131`. Yours will most likely differ as these addresses are pulled from the CNI subnet within Kubernetes:

```
aric@lemur:~$ oc get services

NAME      CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE

webserver   172.30.136.131   <none>        80/TCP    18m
```

7. **Create a route to enable access**: Finally, we can create a route to access our service using the `oc expose` command, followed by the service name we are exposing (`webserver`). To make this routable from our workstation, OpenShift uses the `nip.io` DNS forwarding based on the IP address of the VM. We can enable this by specifying the `--hostname` flag to be any name we want the service to be accessed by, followed by the IP address of the VM, concluding with the suffix `nip.io`:

```
aric@local:~$ oc expose service webserver --hostname="awesomewebapp.192.168.99.

route "webserver" exposed
```

Executing the `oc get routes` command will display the route we just created:

```
aric@local:~$ oc get routes

NAME      HOST/PORT                    PATH      SERVICES
```

```
webserver   awesomewebapp.192.168.99.100.nip.io          webs
```

To ensure the route is working, we can use a web browser and navigate to the forwarded DNS address we assigned to the route. If everything is working, we will be able to see the NGINX welcome screen:

Feel free to continue on deploying more complex containerized applications using your local Minishift cluster. When you are finished, make sure you stop the Minishift instance using the `minishift stop` command.

OpenShift and Ansible Container

As we have seen throughout this chapter, OpenShift is a rich platform that provides valuable abstractions on top of Kubernetes. As such, Ansible Container provides ample support for deploying and running containerized application life cycles through OpenShift. Since OpenShift and Ansible Container are both products of the same parent company, Red Hat, it is apparent that OpenShift and Ansible Container will have excellent compatibility. So far, we have primarily discussed building containers using Ansible Container and running them locally on a Docker host.

Now that we have a firm foundation from which to understand Kubernetes and OpenShift, we can combine the knowledge we have gained so far with Ansible Container to learn how to use Ansible Container as a true end-to-end production-ready deployment and life cycle management solution. Things are about to get interesting!

References

- **OpenShift project**: https://www.openshift.com/
- **MiniShift project**: https://www.openshift.org/minishift/
- **Installing MiniShift**: https://docs.openshift.org/latest/minishift/getting-started/installing.html

Summary

Container orchestration platforms such as Kubernetes and OpenShift are rapidly being adopted by organizations to ease the complex process of scaling out applications, deploying updates, and ensuring maximum reliability. With the increasing popularity of these platforms, it is even more important that we understand the implications of adopting these technologies in order to support the organizational and cultural shift of mentality these technologies bring to the table.

OpenShift is a platform built on top of the Red Hat distribution of Kubernetes that aims to provide the best experience for working with Kubernetes. At the beginning of the chapter we learned what OpenShift is, and the various capabilities that Red Hat is working to deliver with the OpenShift platform. Next, we learned how to install the Minishift project, which is a developer-oriented solution for deploying OpenShift locally.

Once we had Minishift installed and working locally, we learned how to run pods, deployments, services, and routes from the Minishift web user interface. Finally, we learned about the OpenShift command-line interface, OC, and how it functions in a similar capacity to kubectl to provide CLI access to OpenShift and the innovative functionality that OpenShift builds on top of Kubernetes.

In the next chapter, my aim is to tie our knowledge of OpenShift and Kubernetes back into Ansible Container to learn about the final step in the Ansible Container workflow, deployment. The deployment functionality sets Ansible Container apart as a truly robust tool for not only building and testing container images, but also for deploying them all the way through to containerized production environments running on Kubernetes and OpenShift.

Deploying Your First Project

Throughout this book so far, we have looked at the various ways we can run builds and containers using the Ansible Container workflow. We learned about running containers in a local Docker daemon, pushing built containers to a remote Docker image repository, managing container images, and even running containers at scale using container orchestration tools such as Kubernetes and OpenShift. We have almost come full circle, demonstrating the rich capabilities of Ansible Container and how it can be leveraged as a fully functional tool for building, running, and testing container images throughout an application's life cycle.

However, there is one aspect of the Ansible Container workflow we have not yet looked at in depth. In previous chapters, we alluded to the `deploy` command, and how `deploy` can be leveraged to run containers in production environments, or on remote systems. Now that we have covered a lot of the basics of how Docker, Kubernetes, and OpenShift work, it is time we turned our attention to the final Ansible Container workflow component: `ansible-container deploy`. It is my goal that, by reading through this chapter and following along with the examples, it will become evident to the reader that Ansible Container is more than a tool used to build and run container images locally. It is a robust tool for complex containerized application deployments across a variety of popular container platforms.

Throughout this chapter, we will cover the following topics:

- Overview of ansible-container deploy
- Deploying containers to Kubernetes
- Deploying containers to OpenShift

Overview of ansible-container deploy

The `ansible-container deploy` command is the component of the Ansible Container workflow that is responsible for, you guessed it, deploying containers to remote container service engines. At the time of writing, these engines include Docker, Kubernetes, and OpenShift. By leveraging configuration in the `container.yml` file, Ansible Container has the ability to authenticate to these services and leverage API calls to start containers according to the configuration specified by the user. Deployment with Ansible Container is a two-step process. First, Ansible Container pushes the built container images to a remote image registry, similar to Docker Hub or `Quay.io`. This enables the remote container runtime service to have access to the containers during the deployment process. Second, Ansible Container generates deployment playbooks that can be executed locally and performs the deployment using the `ansible-container run` command. Working through the deploy process can be a little confusing at first. The following flowchart demonstrates the deployment process after first building and running a project locally:

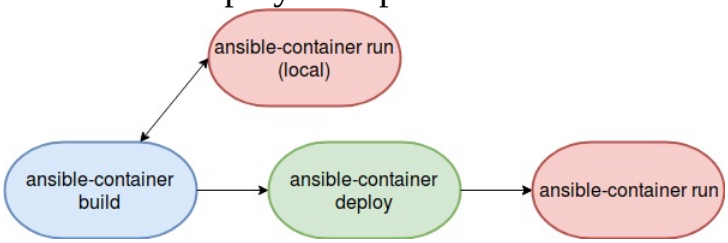


Figure 1: ansible-container deploy workflow We are going to start out by looking at examples using our simple NGINX container project. Later, we are going to look at deploying examples to Kubernetes and OpenShift using the MariaDB project we built in `Chapter 4`, *What's in a Role?*

ansible-container deploy

Before we start looking at `ansible-container deploy`, let's first rebuild the NGINX project we created earlier. In your Ubuntu Vagrant lab VM, navigate to the `/vagrant/AnsibleContainer/nginx_demo` directory; or, if you built this example yourself in another directory, navigate to it and run the `ansible-container build` command. This will make sure that the lab VM has a fresh build of the project:

```
ubuntu@node01:/vagrant/AnsibleContainer/nginx_demo$ ansible-container build


Building Docker Engine context...


Starting Docker build of Ansible Container Conductor image (please be p


Parsing conductor CLI args.


Docker™ daemon integration engine loaded. Build starting.       project=n


Building service...     project=nginx_demo service=webserver




PLAY [webserver] ***************************************************
```

TASK [Gathering Facts] *

ok: [webserver]

TASK [ansible.nginx-container : Install epel-release] ****************

changed: [webserver]

TASK [ansible.nginx-container : Install nginx] ***********************

changed: [webserver] => (item=[u'nginx', u'rsync'])

TASK [ansible.nginx-container : Install dumb init] *******************

changed: [webserver]**

TASK [ansible.nginx-container : Update nginx user] **************

changed: [webserver]

TASK [ansible.nginx-container : Put nginx config] ****************

changed: [webserver]

You can validate that the project has successfully been built and the container images are cached by running the `docker images` command:

```
ubuntu@node01:/vagrant/AnsibleContainer/nginx_demo$ docker images

REPOSITORY  TAG   IMAGE ID  CREATED  SIZE

aric49/nginx_demo-webserver 20171022202358 09f7b7cc3e3e 10 minutes
```

Now that we have the container cached locally, we can use the `ansible-container deploy` command to simulate project deployment. Without providing any arguments about what engine we will deploy our container to, `ansible-container deploy` will generate playbooks that can be used to deploy our project onto a local or remote host that is running Docker. It will also push our project to the registries configured in the `container.yml` file located in the `root` directory of our project. Due to the fact that `deploy` leverages much of the same functionality as `ansible-container push`, we will provide `deploy` with the same flags we would provide the `push` command concerning our container image registry. In this case,

we will tell it to push to our Docker Hub registry, as we will provide the username for our account and any tags we want to use to differentiate this version of the container from previous versions. For the purposes of demonstration, we will use the `deploy` tag:

```
ubuntu@node01:/vagrant/AnsibleContainer/nginx_demo$ ansible-container deploy --push-to
```

Enter password for aric49 at Docker Hub:

Parsing conductor CLI args.

Engine integration loaded. Preparing push. engine=Docker™ daemon

Tagging aric49/nginx_demo-webserver

Pushing aric49/nginx_demo-webserver:deploy...

The push refers to a repository [docker.io/aric49/nginx_demo-webserver]

Preparing

Pushing

Pushed

Pushing

Pushed

20171022202358: digest: sha256:74948d56b3289009a6329c0c2035e3217d(

Conductor terminated. Cleaning up. command_rc=0 conductor_id=4c'

er=False

Parsing conductor CLI args.

Engine integration loaded. Preparing deploy. engine=Docker™ daemon

Verifying image for webserver

Conductor terminated. Cleaning up. command_rc=0 conductor_id=ac

The deploy process, in a similar fashion to the push process, will prompt you for the password for your Docker Hub account. Upon successful authentication, it will push your container image layers to the container image registry. So far, this might look exactly identical to the push process. However, you may notice that, in the `root` directory of your project, a new directory called `ansible-deployment` now exists. Within this directory, you will find a single Ansible playbook that is named identically to that of your project, `nginx_demo`. Here is a sample of what this playbook looks like:

```
- name: Deploy nginx_demo
```

```
hosts: localhost

gather_facts: false

tasks:

  - docker_service:

      definition:

        services: &id001

          webserver:

            image: docker.io/aric49/nginx_demo-webserver:deploy

            command: [/usr/bin/dumb-init, nginx, -c, /etc/nginx/nginx.conf

            ports:

              - 80:8000
```

```
                user: nginx

            version: '2'

        state: present

        project_name: nginx_demo

    tags:

      - start

- docker_service:

        definition:

            services: *id001

            version: '2'

        state: present

        project_name: nginx_demo
```

```
        restarted: true

    tags:

      - restart

TRUNCATED
```

You may have to ensure the `image` line reflects the image path in this format, `docker.io/username/containername:tag`, as some versions of Ansible Container supply the wrong path as input in the playbook. If this is the case, simply modify the playbook in a text editor.

The deploy playbook works by making calls to the `docker_service` module running on the target hosts. By default, the container uses `localhost` as the target host for deployment. However, you can easily provide a standard Ansible inventory file to have this project run on remote hosts. The deploy playbook supports full Docker life cycle application management, such as starting the container, restarting, and ultimately destroying the project by providing a series of playbook tags to conditionally execute the desired functionality. You may notice that the playbook inherits many of the settings we configured in the `container.yml` file. Ansible Container uses these settings so that the playbooks can be executed independently of the project itself.

Since we have already looked at using `ansible-container run` to run our containers locally throughout this book, let's try executing the playbook directly to start the container. This mimics the same process used if you want to manually run a deployment outside of the Ansible Container workflow. This can be accomplished by using the `ansible-playbook` command with the `start` tag to deploy a project on our localhost. You may notice that this process is exactly the same process as running the `ansible-container run` command. It is important to note that any of the core Ansible Container functionality (run, restart, stop, and destroy)

can be executed independently of Ansible Container by running playbooks directly and supplying the appropriate tag according to the functionality you are trying to achieve:

```
ubuntu@node01:$ ansible-playbook nginx_demo.yml --tags start

[WARNING]: Host file not found: /etc/ansible/hosts

 [WARNING]: provided hosts list is empty, only localhost is available

PLAY [Deploy nginx_demo] ***************************************

TASK [docker_service] ******************************************

changed: [localhost]
```

```
PLAY RECAP *********************************

localhost                  : ok=1    changed=1    unreachable=0    failed=0
```

Once the playbook execution has completed, `PLAY RECAP` will show that one task has executed a change on your localhost. You can execute the `docker ps -a` command to confirm the project has successfully been deployed:

```
ubuntu@node01:$ docker ps -a

CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS  PORTS

4b4b3b032c61        aric49/nginx_demo-webserver:deploy   "/usr/bin/duml
```

In a similar way, we can run this playbook again, passing in the `--restart` tag to restart the container on the Docker host. After executing the playbook for a second time, you should see that a single task has once more changed, indicating the container has been restarted. This mimics the functionality that the `ansible-container restart` command provides. The `status` column in `docker ps -a` will show that the container has only been up for a handful of seconds after executing the restart:

```
ubuntu@node01:$ docker ps -a

CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS  PORTS

4b4b3b032c61        aric49/nginx_demo-webserver:deploy "/usr/bin/dumb
```

The `stop` tag can be passed into the `ansible-playbook` command to temporarily stop the running container. Similar to `restart`, the `docker ps -a` output will show that the container is in an `exit` status:

```
ubuntu@node01:$ ansible-playbook nginx_demo.yml --tags stop

[WARNING]: Host file not found: /etc/ansible/hosts

 [WARNING]: provided hosts list is empty, only localhost is available

PLAY [Deploy nginx_demo] ************************************

TASK [docker_service] ***************************************

changed: [localhost]
```

TRUNCATED

**PLAY RECAP **

localhost: ok=4 changed=2 unreachable=0 failed=0

We can now check the status using `docker ps -a` command:

```
ubuntu@node01:$ docker ps -a
```

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS

4b4b3b032c61 aric49/nginx_demo-webserver:deploy "/usr/bin/duml

Finally, the project can be removed entirely from our Docker host by passing in the `destroy` tag. Running this `playbook` tag will execute a few more steps in the playbook, but will ultimately remove all traces of your project from the host:

```
ubuntu@node01:$ ansible-playbook nginx_demo.yml --tags destroy
```

[WARNING]: Host file not found: /etc/ansible/hosts

[WARNING]: provided hosts list is empty, only localhost is available

PLAY [Deploy nginx_demo] ************************************

TASK [docker_service] *************************************

changed: [localhost]

TASK [docker_image] *

ok: [localhost]

```
TASK [docker_image] ************************************************

TASK [docker_image] ************************************************
ok: [localhost]

TRUNCATED

PLAY RECAP *********************************************************
localhost: ok=7    changed=2    unreachable=0    failed=0
```

Behind the scenes, when any of the core Ansible Container commands are executed, they are essentially wrappers around the same playbook that gets generated as a part of your project. The purpose of this portion of the chapter was to demonstrate to the reader the overall flow of deploying projects using Ansible Container locally, and to build upon these skills deeper in the lesson. Where deployment gets really interesting is when using Kubernetes and OpenShift as the target deployment engines. Using the Ansible Container workflow commands with the corresponding container platform engine, we can

manage containerized deployments directly using the Ansible Container workflow commands instead of executing the playbooks directly.

Deploying containers to Kubernetes

One of the many aspects that makes the Ansible Container workflow so flexible and appealing to organizations and individuals looking to adopt Ansible as the native support for remote deployments using Kubernetes and OpenShift. In this section, we will look at using the `ansible-container deploy` command to deploy our containers to our Google Cloud Kubernetes cluster we created in `Chapter 5`, C*ontainers at Scale with Kubernetes*.

As we discussed in the previous section, running `ansible-container deploy` by itself will, by default, push your container to any image registries you have configured in your `container.yml` file and generate a new directory in the `root` of your project called `ansible-deployment`. Inside of this directory, a single YAML playbook file named after the project will be present. This playbook is used to deploy your container to any Docker host, quite similar to the `ansible-container run` command. For the purposes of this example, we are going to run `ansible-container deploy` using the Kubernetes engine so we can leverage Ansible Container as a deployment tool for Kubernetes, creating service definitions and deployment abstractions automatically.

In order to enable the Kubernetes to deploy functionality in Ansible Container, we will add a couple of new parameters to the project `container.yml` file. Specifically, we need to point our project to our Kubernetes authentication configuration file, and define the namespace our container operates in within Kubernetes. For this example, I will use our previously used MariaDB project, but with a few modifications to the `container.yml` file to support Kubernetes. For reference, this project can be found in the official Git repository for this book, in the `Kubernetes/mariadb_demo_k8s` directory. Feel free to follow along, or modify the existing MariaDB project to support Kubernetes. In the settings section, we will add the `k8s_namespace` and `k8s_auth`.

The Kubernetes `namespace` will contain the name of the namespace in our cluster we want to deploy our project into, and the `auth` section will provide the path to

our Kubernetes authentication configuration file. The default location of the Kubernetes authentication configuration is `/home/user/.kube/config`. If you are following along using the Vagrant lab, Google Cloud SDK places this configuration file at `/home/ubuntu/.kube/config`.

Before we can begin, though, we need to set up a default access token in order for Ansible Container to access the Google Cloud API. We can do this by executing the `gcloud auth application-default login` command. Executing this command will provide you with a hyperlink you can use to allow permissions to the Google Cloud API using your Google login credentials, similar to what we did in `Chapter 5`, *Containers at Scale with Kubernetes*. The Google Cloud API will give you a token you can enter at the command line that will generate an application default credentials file, located at `/home/ubuntu/.config/gcloud/application_default_credentials.json`:

```
ubuntu@node01:$ gcloud auth application-default login
Go to the following link in your browser:

    https://accounts.google.com/o/oauth2/TRUNCATED


Enter verification code: XXXXXXXXXXXXXXX

Credentials saved to file: [/home/ubuntu/.config/gcloud/application_default_credentials
```

These credentials will be used by any library that requests access to any Google Cloud resources, including Ansible Container.

The Google Container Engineer-specific steps are only required if you are using a Google Cloud Kubernetes cluster. You can skip these step if you are using a local Kubernetes environment such as Minikube.

Now that we have the proper permissions set in the Google Cloud API, we can modify the `container.yml` file of our MariaDB project to support the Kubernetes deployment engine:

```
version: "2"
settings:
  conductor_base: ubuntu:16.04
  project_name: mariadb-k8s
  roles_path:
    - ./roles/
  k8s_namespace:
    name: database
  k8s_auth:
```

```
    config_file: /home/ubuntu/.kube/config
services:
  mariadb-database:
    roles:
      - role: mariadb_role

registries:
  docker:
    url: https://index.docker.io/v1/
    namespace: username
```

You will notice the following changes we have made to support the Kubernetes deployment:

- `project_name`: For this example, we have added a field in the `settings` block called `project_name`. Throughout this book, we have allowed our projects to take the default name of the directory that it is built in. Kubernetes is limited as to the characters it can use to define services and pods, so we want to ensure we do not use illegal characters in our project name by overriding them in the `container.yml` file.
- `k8s_namespace`: This defines the Kubernetes namespace we will deploy our pods into. Leaving this stanza blank will cause Ansible Container to use the default namespace that we used in our NGINX deployment earlier in the chapter. In this example, we will use a different namespace called `database`.
- `k8s_auth`: This is where we specify the location of our Kubernetes authentication configuration file. Within this file, Ansible Container is able to extract the IP address of our API server, the access credentials to create resources in the cluster, as well as the SSL certificates required to connect to Kubernetes.

Once these changes have been placed in your `container.yml` file, let's build the project:

```
ubuntu@node01:$ ansible-container build
Building Docker Engine context...
Starting Docker build of Ansible Container Conductor image (please be patient)...
Parsing conductor CLI args.
Docker™ daemon integration engine loaded. Build starting.       project=mariadb-k8s
Building service...     project=mariadb-k8s service=mariadb-database

PLAY [mariadb-database] ********************************************************

TASK [Gathering Facts] *********************************************************
ok: [mariadb-database]

TASK [mariadb_role : Install Base Packages] ************************************
changed: [mariadb-database] => (item=[u'ca-certificates', u'apt-utils'])

TASK [mariadb_role : Install dumb-init for Container Init System]
```

Once the project has finished building, we can use the `ansible-container deploy` command, specifying the `--engine` flag to use `k8s`, and providing the details for the Docker image registry we want to push to as configured in our `container.yml` file. For the sake of separation, let's also tag the image version with `kubernetes` so we can keep this version separate in our repository. Ansible Container will then push our image to the Docker Hub repository and generate the deployment playbooks specific to Kubernetes:

K8s is shorthand for Kubernetes since Kubernetes comprises the letter K with 8 letters after it. This is commonly pronounced in the community as ***Kay-Eights****.*

```
ubuntu@node01:$ ansible-container --engine k8s deploy --push-to docker --tag kubernetes
Parsing conductor CLI args.
Engine integration loaded. Preparing push.      engine=K8s
Tagging aric49/mariadb-k8s-mariadb-database
Pushing aric49/mariadb-k8s-mariadb-database:kubernetes...
The push refers to a repository [docker.io/aric49/mariadb-k8s-mariadb-database]
Preparing
Waiting
Layer already exists
Pushing
Pushed
Pushing
Pushed
kubernetes: digest: sha256:563ec4593945b13b481c03ab7813bb64c0dc1b7a1d1ae8c4b61b744574df
Conductor terminated. Cleaning up.      command_rc=0 conductor_id=27fd42d3920deb12f8c81
Parsing conductor CLI args.
Engine integration loaded. Preparing deploy.    engine=K8s
Verifying image for mariadb-database
ansible-galaxy 2.5.0
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/root/.ansible/plugins/modules', u'/usr/share/ansi
  ansible python module location = /usr/local/lib/python2.7/dist-packages/ansible
  executable location = /usr/local/bin/ansible-galaxy
  python version = 2.7.12 (default, Nov 19 2016, 06:48:10) [GCC 5.4.0 20160609]
Using /etc/ansible/ansible.cfg as config file
Opened /root/.ansible_galaxy
Processing role ansible.kubernetes-modules
Opened /root/.ansible_galaxy
- downloading role 'kubernetes-modules', owned by ansible
https://galaxy.ansible.com/api/v1/roles/?owner__username=ansible&name=kubernetes-module
https://galaxy.ansible.com/api/v1/roles/16501/versions/?page_size=50
- downloading role from https://github.com/ansible/ansible-kubernetes-modules/archive/m
- extracting ansible.kubernetes-modules to /vagrant/Kubernetes/mariadb_demo_k8s/ansible
- ansible.kubernetes-modules (master) was installed successfully
Conductor terminated. Cleaning up.      command_rc=0 conductor_id=d66eb0a142190022fee50
```

Once this process has completed, you will notice that, similar to the last example, a new directory has appeared in your project called `ansible-deployment`.

Inside of this directory, you will find a single playbook and a `roles` directory that is responsible for performing the actual deployment of our service to Kubernetes. As with our previous localhost example, this playbook is likewise divided up based on tags that control starting, stopping, and restarting the service in our cluster. Since we have not yet deployed our service, we can start the deployment using the `ansible-container run` command with the `--engine k8s` flag to indicate a Kubernetes deployment. Assuming we have configured everything correctly in the `container.yml` file, you should see a successful playbook run, indicating the container has been deployed to the Kubernetes cluster:

```
ubuntu@node01:/vagrant/Kubernetes/mariadb_demo_k8s$ ansible-container --engine k8s run
Parsing conductor CLI args.
Engine integration loaded. Preparing run.       engine=k8s
Verifying service image service=mariadb-database
PLAY [Manage the lifecycle of mariadb-k8s on K8s] ************************************

TASK [Create namespace database] *****************************************************
ok: [localhost]

TASK [Create service] ****************************************************************
ok: [localhost]

TASK [Create deployment, and scale replicas up] **************************************
changed: [localhost]

PLAY RECAP ***************************************
localhost: ok=3    changed=1    unreachable=0    failed=0
```

Using the `kubectl get pods` command from earlier, we can validate that our Kubernetes pod has been deployed and is successfully running. Since we deployed this particular pod in its own namespace, we need to use the `--namespace` flag to see pods that are running in other namespaces:

```
ubuntu@node01:$ kubectl get pods --namespace database
NAME                                READY     STATUS    RESTARTS   AGE
mariadb-database-1880651791-979zd   1/1       Running   0          3m
```

Using the `ansible-container run` command with the Kubernetes engine is a powerful tool for creating cloud-native services that run on a Kubernetes cluster. Using Ansible Container, you have the flexibility to choose how you want to deploy applications, by executing the playbooks directly or automatically using the Ansible Container workflow. If you wish to delete the current deployment from your Kubernetes cluster, you can simply run the `ansible-container --engine k8s destroy` command to completely remove the pods and deployment artifacts from the cluster. It is important to note that the other Ansible Container workflow commands (start, stop, and restart) are perfectly applicable suffixes to use with

the Kubernetes deployment engine. For the sake of reducing redundancy, let's take a look at how `ansible-container deploy` and the workflow commands work with the OpenShift deployment engine. Functionally, the Ansible Container workflow commands are identical for Kubernetes and OpenShift.

Deploying containers to OpenShift

The `ansible-container deploy` command also supports deployments directly to OpenShift using the native OpenShift APIs. Since OpenShift is built on top of Kubernetes, you will discover that deploying containers to OpenShift is quite a similar process to Kubernetes deployments, since OpenShift authentication works very similarly to Kubernetes on the backend. Before beginning, these examples are going to use the Vagrant lab VM running at the same time as the Minishift VM we created in `Chapter 6`, *Managing Containers with OpenShift*. This can get quite CPU and RAM intensive. If you're attempting to run these examples with 8 GB of RAM or higher, you should get good performance.

However, if you are constrained on resources, these examples can run reasonably well using the OpenShift free tier cloud account, although you may run into issues with the limited quotas provided.

Before beginning, we need to first ensure that the Vagrant lab environment, as well as our Minishift VM, are running and reachable from the VirtualBox network. Since the hypervisors used to create our Vagrant lab environment and our Minishift cluster are both using VirtualBox, we should by default have network connectivity between the two VMs. We can validate this by attempting to ping the Minishift VM from our Vagrant lab VM. First, we need to start the Minishift VM using reasonable specifications for your local workstation. In this example, I am going to start the Minishift VM allocating 8 GB of RAM and 50 GB virtual hard disk storage for it. If you are running both VMs simultaneously, you may only be able to allocate the minimum 2 GB of RAM for Minishift:

```
aric@lemur:~/Development/minishift$ minishift start --vm-driver=virtualbox --disk-size=
```

-- Starting local OpenShift cluster using 'virtualbox' hypervisor ...

-- Starting Minishift VM ..... OK

-- Checking for IP address ... OK

-- Checking if external host is reachable from the Minishift VM ...

Pinging 8.8.8.8 ... OK

-- Checking HTTP connectivity from the VM ...

Retrieving http://minishift.io/index.html ... OK

-- Checking if persistent storage volume is mounted ... OK

-- Checking available disk space ... 7% used OK

-- OpenShift cluster will be configured with ...

Version: v3.6.0

-- Checking `oc` support for startup flags ...

host-config-dir ... OK

host-data-dir ... OK

host-pv-dir ... OK

host-volumes-dir ... OK

routing-suffix ... OK

Starting OpenShift using openshift/origin:v3.6.0 ...

OpenShift server started.

The server is accessible via web console at:

https://192.168.99.100:8443

At the end of the startup process, you should receive an IP from which the OpenShift Web UI is accessible. We need to ensure this IP address is reachable from our Vagrant lab node:

```
ubuntu@node01:~$ ping 192.168.99.100
```

PING 192.168.99.100 (192.168.99.100) 56(84) bytes of data.

64 bytes from 192.168.99.100: icmp_seq=1 ttl=63 time=0.642 ms

64 bytes from 192.168.99.100: icmp_seq=2 ttl=63 time=0.602 ms

64 bytes from 192.168.99.100: icmp_seq=3 ttl=63 time=0.929 ms

^C

--- 192.168.99.100 ping statistics ---

3 packets transmitted, 3 received, 0% packet loss, time 2003ms

rtt min/avg/max/mdev = 0.602/0.724/0.929/0.147 ms

If this IP address is not pingable, you may have to configure your VirtualBox networking so that network connectivity is available. A great resource to learn more about configuring and debugging VirtualBox networks is the official VirtualBox documentation: `https://www.virtualbox.org/manual/ch06.html`.

Once networking has been established and verified between the Minishift VM and the Vagrant lab VM, next we will need to install the OC on our Vagrant lab VM to allow us to authenticate to OpenShift. This will be the exact same process we completed in `Chapter 6`, *Managing Containers with OpenShift*:

1. Download the OC binary packages using `wget`:

```
ubuntu@node01:~$ wget https://mirror.openshift.com/pub/openshift-v3/clients/3.6
```

Resolving mirror.openshift.com (mirror.openshift.com)... 54.173.18

Connecting to mirror.openshift.com (mirror.openshift.com)|54.173.

HTTP request sent, awaiting response... 200 OK

Length: 36147137 (34M) [application/x-gzip]

Saving to: ‘oc.tar.gz'

2017-10-28 15:21:24 (6.79 MB/s) - ‘oc.tar.gz' saved [36147137/3614

2. Extract the TAR archive using the `tar -xf` command:

```
ubuntu@node01:~$ tar -xf oc.tar.gz
```

oc oc.tar.gz

3. Copy the binary to a `$PATH` location:

```
ubuntu@node01:~$ sudo cp oc /usr/local/bin
```

If you have existing Kubernetes credentials in `/home/ubuntu/.kube/config`, you will need to back them up to another location so the OC does not overwrite them (or simply delete the config file if you have no further use for it any longer: `rm -rf ~/.kube/config`).

Next, we need to authenticate to the local OpenShift cluster using the `oc login` command in order to generate our Kubernetes credential file that Ansible Container will leverage. The `oc login` command takes the URL endpoint of the OpenShift cluster as a parameter. By default, the OC will write a Kubernetes configuration file to `/home/ubuntu/.kube/config`. This file will serve as our means of authenticating to OpenShift Kubernetes to perform automated deployments. Remember to log in as the `developer` user, which uses any user-provided password to log in:

```
ubuntu@node01:$ oc login https://192.168.99.100:8443


The server uses a certificate signed by an unknown authority.


You can bypass the certificate check, but any data you send to the server c


Use insecure connections? (y/n): y




Authentication required for https://192.168.99.100:8443 (openshift)


Username: developer


Password:


Login successful.
```

You have one project on this server: "myproject"

Using project "myproject".

Welcome! See 'oc help' to get started.

Once you have successfully authenticated, you should notice there is now a new Kubernetes configuration file written to the path `/home/ubuntu/.kube/config`. This is the configuration file that Ansible Container will use for access to OpenShift:

```
ubuntu@node01:~$ cat .kube/config

apiVersion: v1

clusters:

- cluster:

    insecure-skip-tls-verify: true

    server: https://192.168.99.100:8443
```

```
  name: 192-168-99-100:8443

contexts:

- context:

    cluster: 192-168-99-100:8443

    namespace: myproject

    user: developer/192-168-99-100:8443

  name: myproject/192-168-99-100:8443/developer

current-context: myproject/192-168-99-100:8443/developer

kind: Config

preferences: {}

users:
```

```
- name: developer/192-168-99-100:8443

  user:

    token: TOKEN-CENSORED
```

Let's test authentication to the local OpenShift instance by using the `oc get all` command. If authentication has been successful, you should see a list of pods, deployments, services, and routes currently running in your local OpenShift environment:

```
ubuntu@node01:~$ oc get all

NAME   DOCKER REPO   TAGS    UPDATED

is/oc-test-deployment   172.30.1.1:5000/myproject/oc-test-deployment   lat

is/test-deployment      172.30.1.1:5000/myproject/test-deployment      latest




NAME              DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   A

deploy/webserver  1         1         1            1           12d
```

```
NAME              HOST/PORT                       PATH     SERVICES   PO
routes/webserver  awesomwebapp.192.168.99.100.nip.io          webserve


NAME          CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
svc/webserver 172.30.136.131  <none>        80/TCP    12d


NAME                  DESIRED   CURRENT   READY     AGE
rs/webserver-1266346274 1       1         1         12d


NAME                      READY     STATUS    RESTARTS   AGE
po/webserver-1266346274-m2jvd 1/1       Running   3          12d
```

OpenShift, by default, leverages the same authentication mechanism that

Kubernetes uses in our `container.yml` file. The only thing we need to provide is the path to our Kubernetes configuration file, as well as the Kubernetes namespace the project will be deployed into. Since we have previously configured this in our MariaDB project in the last section, let's reuse this same configuration to deploy our project to OpenShift. As a review, let's look at the content of our MariaDB project in the Vagrant Lab VM (`/vagrant/Kubernetes/mariadb_demo_k8s`), and look at the contents of the `container.yml` file:

```
version: "2"

settings:

  conductor_base: ubuntu:16.04

  project_name: mariadb-k8s

  roles_path:

    - ./roles/

  k8s_namespace:

    name: database

  k8s_auth:
```

```
    config_file: /home/ubuntu/.kube/config

services:

  mariadb-database:

    roles:

      - role: mariadb_role



registries:

  docker:

    url: https://index.docker.io/v1/

    namespace: aric49
```

The only difference here is that the `k8s_namespace` parameter will define which OpenShift project you want to deploy your container into. In OpenShift terminology, `project` and `namespace` are essentially identical. For now, let's leave the configuration as is and look at how to deploy our project using the OpenShift engine. Deploying projects using OpenShift is very similar to how we deployed using Kubernetes, with the exception that we will prefix our Ansible Container

commands with the `--engine openshift` flag so that our project will know to talk to the OpenShift API directly. The same syntax rules apply here as well. We will give our `deploy` command the name of the repository defined in the `container.yml` file to push our container image to, and give it a unique tag to reference later:

```
ubuntu@node01:/vagrant/Kubernetes/mariadb_demo_k8s$ ansible-container --engine openshif

Parsing conductor CLI args.

Engine integration loaded. Preparing push.      engine=OpenShift™

Tagging aric49/mariadb-k8s-mariadb-database

Pushing aric49/mariadb-k8s-mariadb-database:openshift...

The push refers to a repository [docker.io/aric49/mariadb-k8s-mariadb-d

Preparing

Waiting

Layer already exists

openshift: digest: sha256:99139cdd73b80ed29cedf8df4399b368f22e747f18
```

Conductor terminated. Cleaning up. command_rc=0 conductor_id=9b

Parsing conductor CLI args.

Engine integration loaded. Preparing deploy. engine=OpenShift™

Verifying image for mariadb-database

Conductor terminated. Cleaning up. command_rc=0 conductor_id=76

Once our container image has been pushed, we can validate the deployment playbooks have been generated in the `ansible-deployment` directory (`Kubernetes/mariadb_demo_k8s/ansible-deployment/mariadb-k8s.yml`):

```
- name: Manage the lifecycle of mariadb-k8s on OpenShift™

  hosts: localhost

  gather_facts: no

  connection: local

  # Include Ansible Kubernetes and OpenShift modules

  roles:
```

```
  - role: ansible.kubernetes-modules

vars_files: []

# Tasks for setting the application state. Valid tags include: start, stop, restar

tasks:

  - name: Create project myproject

    openshift_v1_project:

      name: myproject

      state: present

    tags:

      - start

  - name: Destroy the application by removing project myproject

    openshift_v1_project:
```

```
        name: myproject

        state: absent

      tags:

        - destroy

TRUNCATED
```

Similar to the Docker and Kubernetes deployment engines, these playbooks can be executed independently using the `ansible-playbook` command, or by using the `ansible-container run` command. Let's run our project and deploy it into OpenShift using the `ansible-container run` command:

```
ubuntu@node01:/vagrant/Kubernetes/mariadb_demo_k8s$ ansible-container --engine openshif


Parsing conductor CLI args.

Engine integration loaded. Preparing run.       engine=OpenShift™

Verifying service image service=mariadb-database
```

PLAY [Manage the lifecycle of mariadb-k8s on OpenShift?] *********

TASK [Create project database] *********

changed: [localhost]

TASK [Create service] *********

changed: [localhost]

TASK [Create deployment, and scale replicas up] *********

changed: [localhost]

TASK [Create route] *********

changed: [localhost]

PLAY RECAP *

localhost : ok=4 changed=4 unreachable=0 failed=0

All services running. playbook_rc=0

Conductor terminated. Cleaning up. command_rc=0 conductor_id=ac

Upon successful completion of the playbook run, we can log in to the OpenShift web user interface to look at the deployment we just executed. In a web browser, navigate to the URL provided in the output of the `minishift start` command (in my case, it is `192.168.99.100`), accept the self-signed certificate, and log in as the `developer` user:

Figure 2: Logging into the OpenShift console

Upon logging in, you should see a new project has been created, called database.

In this project you can see everything that the Ansible Container deployment has generated by default:

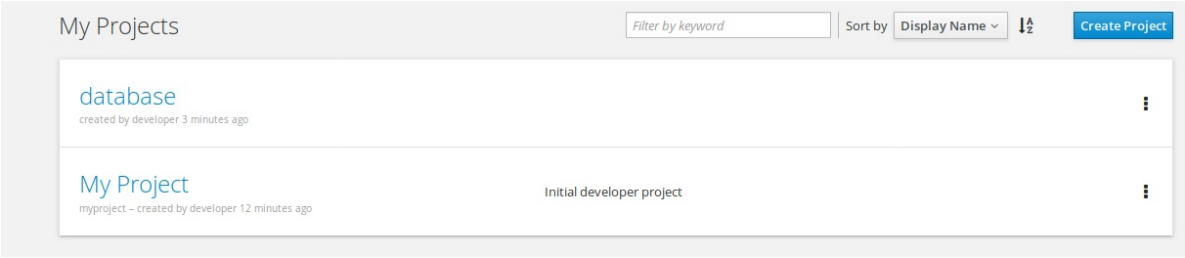


Figure 3: The database project has been created under My Projects in the OpenShift console

Clicking on the project, `database` will bring you to a dashboard showing the relevant details for the deployment:

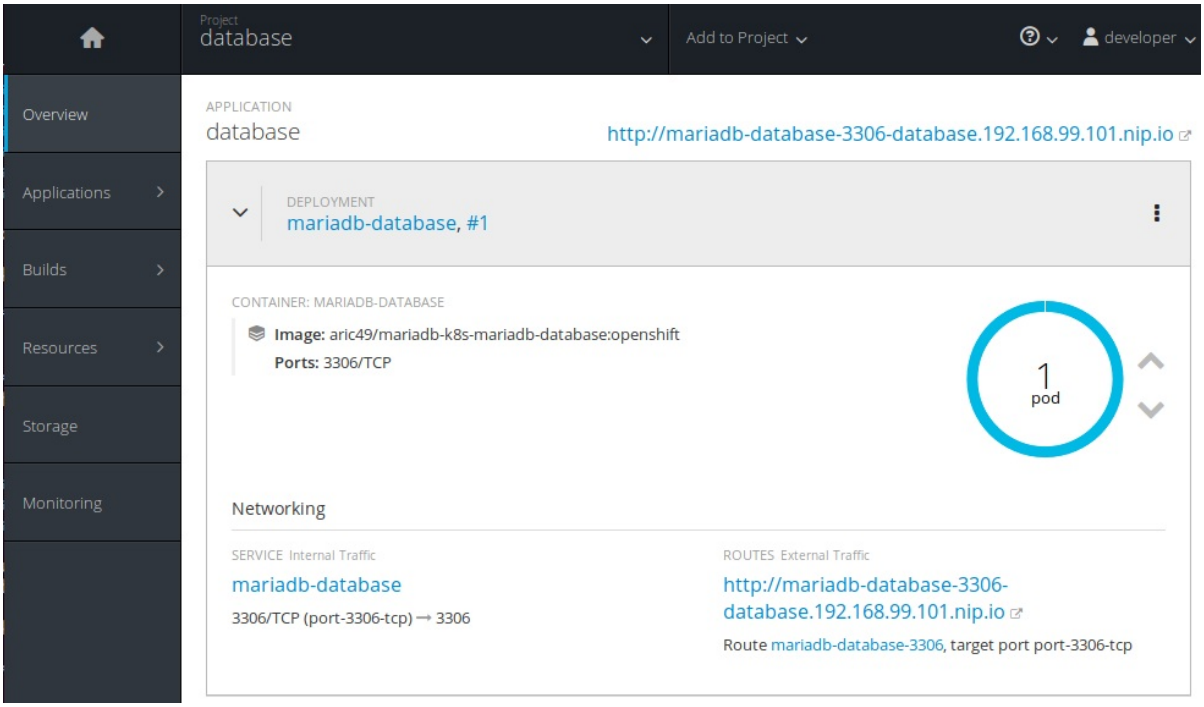


Figure 4: MariaDB database deployment

As you can see, by default the Ansible Container playbooks used to deploy OpenShift run with a very useable set of default configuration options. Right away, we can see that Ansible Container has created a new project for our project, called `database`. Within this project, a default deployment exists that has our MariaDB pod created and running. It has even taken the steps for us to create a default service with a pre-configured set of labels, and created a route to access

the service using the `nip.io` DNS service. Essentially, our new service is deployed and ready to go right out-of-the-box. In order to use the OpenShift deployment engine, we didn't actually have to change any of the `container.yml` configuration; we used exactly the same configuration we used to deploy to Kubernetes, with the exception of using a different Kubernetes config file, and specifying the OpenShift engine in our `run` command. As I'm sure you can see, having the ability to deploy to OpenShift or Kubernetes transparently is immensely powerful. This allows Ansible Container to function seamlessly no matter what target architecture your service is configured to use.

We can also validate the deployment by using the OC command-line interface client. From the Vagrant lab VM, you can use the `oc project` command to switch to the `database` project:

```
ubuntu@node01:/vagrant/Kubernetes/mariadb_demo_k8s$ oc project database


Now using project "database" on server "https://192.168.99.100:8443".
```

Once we have switched to a new project context, we can use the `oc get all` command to show everything configured to run in this project, including the pods, services, and route configuration generated by Ansible Container:

```
ubuntu@node01:/vagrant/Kubernetes/mariadb_demo_k8s$ oc get all


NAME                   REVISION   DESIRED   CURRENT   TRIGGERED B

dc/mariadb-database    1          1         1         config




NAME                   DESIRED   CURRENT   READY     AGE


```

```
rc/mariadb-database-1   1         1         1         27s


NAME                          HOST/PORT                                  PATH

routes/mariadb-database-3306   mariadb-database-3306-database.192.168


NAME                 CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE

svc/mariadb-database   172.30.154.77   <none>        3306/TCP   28s


NAME                        READY     STATUS    RESTARTS   AGE

po/mariadb-database-1-bj19h   1/1       Running   0          26s
```

Along with `ansible-container run`, we can also use the standard Ansible Container workflow commands to manage our deployment, such as `stop`, `restart`, and `destroy`. As we discussed earlier, these workflow commands function identically with the Kubernetes engine. Let's first start the `ansible-container stop` command. `stop` will gracefully stop all running pods in the deployment, while keeping the other resources deployed and active. Let's try stopping the deployment and re-running

the `get all` command to learn what happens:

```
ubuntu@node01:$ ansible-container --engine openshift stop

Parsing conductor CLI args.

Engine integration loaded. Preparing to stop all containers.    engine=Ope



PLAY [Manage the lifecycle of mariadb-k8s on OpenShift?] ***********



TASK [Stop running containers by scaling replicas down to 0] **********

changed: [localhost]



PLAY RECAP *********************************************************

localhost                  : ok=1    changed=1    unreachable=0    failed=0
```

All services stopped. playbook_rc=0

Conductor terminated. Cleaning up. command_rc=0 conductor_id=83

Once `stop` has completed successfully, re-run the `oc get all` command:

```
ubuntu@node01:/vagrant/Kubernetes/mariadb_demo_k8s$ oc get all
```

NAME REVISION DESIRED CURRENT TRIGGERED B

dc/mariadb-database 2 0 0 config

NAME DESIRED CURRENT READY AGE

rc/mariadb-database-1 0 0 0 7m

rc/mariadb-database-2 0 0 0 1m

NAME HOST/PORT PATH

```
routes/mariadb-database-3306   mariadb-database-3306-database.192.168


NAME                  CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE

svc/mariadb-database   172.30.154.77   <none>        3306/TCP   7m
```

From the preceding output, we can see that OpenShift has created a new revision for the configuration change we deployed (`REVISION 2`), which describes the deployment as having zero running pod replicas, indicative of the deployment existing in the stopped state (`Current 0`, `Desired 0`, `Ready 0`). However, the route and service artifacts still exist and are running in the cluster. One of the major benefits of OpenShift is the nature of OpenShift to readily track the changes made to the project under various revision definitions. This makes it very easy to roll back to a previous deployment should a change fail or need to be rolled back. Complementary to the `stop` command is the `restart` command, which ensures the current revision is in a running state, after first stopping the service.

Unlike `stop`, `restart` does not create a new revision, since our current revision is already scaled down to zero replicas, but instead will scale up the current revision to ensure that the desired number of pods is running in the project. Let's execute the `ansible-container restart` command for the OpenShift engine and see how this affects our deployment:

```
ubuntu@node01:/vagrant/Kubernetes/mariadb_demo_k8s$ ansible-container --engine openshif


Parsing conductor CLI args.


Engine integration loaded. Preparing to restart containers.     engine=Ope
```

PLAY [Manage the lifecycle of mariadb-k8s on OpenShift?] *********

TASK [Stop running containers by scaling replicas down to 0] *******

ok: [localhost]

TASK [Create deployment, and scale replicas up] *******************

changed: [localhost]

PLAY RECAP *

localhost : ok=2 changed=1 unreachable=0 failed=0

All services restarted. playbook_rc=0

Conductor terminated. Cleaning up. command_rc=0 conductor_id=d5

Executing the `oc get all` command once more, we will see that our current revision (#2) is now running with the desired number of pods:

```
ubuntu@node01:/vagrant/Kubernetes/mariadb_demo_k8s$ oc get all

NAME                  REVISION   DESIRED   CURRENT   TRIGGERED B

dc/mariadb-database   2          1         1         config


NAME                    DESIRED   CURRENT   READY     AGE

rc/mariadb-database-1   0         0         0         31m

rc/mariadb-database-2   1         1         1         26m
```

```
NAME                    HOST/PORT                                PATH

routes/mariadb-database-3306   mariadb-database-3306-database.192.168


NAME                CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE

svc/mariadb-database   172.30.154.77   <none>        3306/TCP   31m


NAME                    READY     STATUS    RESTARTS   AGE

po/mariadb-database-2-g7r7f   1/1       Running   0          6m
```

Finally, we can use the `ansible-container destroy` command to completely remove all traces of our service from the OpenShift (or Kubernetes) cluster. Keep in mind that this will also remove the project as well as any other containers that are also running within the project that may have been deployed manually or by other means outside of Ansible Container. This is why it is important to separate application deployments by OpenShift project and Kubernetes namespace, especially when running commands such as `ansible-container destroy`:

```
ubuntu@node01:/vagrant/Kubernetes/mariadb_demo_k8s$ ansible-container --engine openshif


Parsing conductor CLI args.
```

Engine integration loaded. Preparing to stop+delete all containers and bu

PLAY [Manage the lifecycle of mariadb-k8s on OpenShift?] *********

TASK [Destroy the application by removing project database] *******

changed: [localhost]

PLAY RECAP *******

localhost : ok=1 changed=1 unreachable=0 failed=0

All services destroyed. playbook_rc=0

Conductor terminated. Cleaning up. command_rc=0 conductor_id=dd

According to the task execution, it appears that a single task that was run deleted the entire OpenShift project. This is reflected if we execute the `oc get all` command one final time:

```
ubuntu@node01:/vagrant/Kubernetes/mariadb_demo_k8s$ oc get all
```

Error from server (Forbidden): User "developer" cannot list buildconfigs

Error from server (Forbidden): User "developer" cannot list builds in pr

Error from server (Forbidden): User "developer" cannot list imagestrean

Error from server (Forbidden): User "developer" cannot list deployment

Error from server (Forbidden): User "developer" cannot list deployments

Error from server (Forbidden): User "developer" cannot list horizontalp

Error from server (Forbidden): User "developer" cannot list replicationc

Error from server (Forbidden): User "developer" cannot list routes in pr

Error from server (Forbidden): User "developer" cannot list services in p

> **Error from server (Forbidden): User "developer" cannot list statefulsets.**
>
> **Error from server (Forbidden): User "developer" cannot list jobs.batch i**
>
> **Error from server (Forbidden): User "developer" cannot list replicasets.e**
>
> **Error from server (Forbidden): User "developer" cannot list pods in proj**

These errors indicate that our user can no longer list anything that exists inside of the `database` project due to the fact that it no longer exists. All traces of the project, deployments, services, pods, and routes, have been deleted from the cluster. This is also apparent from the web interface because refreshing the web page will indicate that the projects no longer exist.

References

- **Ansible Container Deployment Guide**: https://docs.ansible.com/ansible-container/reference/deploy.html
- **VirtualBox Networking Guide**: https://www.virtualbox.org/manual/ch06.html

Summary

Over the course of this chapter, we looked at the final Ansible Container workflow command: `ansible-container deploy`. `deploy` is one of the most versatile commands available in the Ansible Container arsenal since it allows us to run and manage containers in production-grade Kubernetes and OpenShift environments. `deploy` opens a new path in our journey to enable the flexibility and agility that containers give our infrastructure. We can now truly use a single tool to not only build and debug containerized applications locally, but also to deploy and manage these same applications in production. Having the ability to use the same expressive Ansible Playbook language to truly build reliable and scalable applications means that deployments can be built around DevOps and automation best practices from day one, instead of the painstaking task of re-engineering deployments so they are automated after the fact.

Just because we have finished learning about the major Ansible Container workflow components does not mean that our journey has ended. So far in this book, we have looked at using Ansible Container to deploy single-function microservices that require no dependencies on other services. Ansible Container being as powerful as it is also has the innate ability to build and deploy multiple containerized applications by expressing links and dependencies on other services. In the next chapter, we will look at how to build and deploy multi-container applications.

Building and Deploying a Multi-Container Project

So far, throughout the course of this book, we have explored the many facets of Ansible Container and containerized application deployments. We have looked at building Docker containers from basic Dockerfiles, using Ansible Container to install roles, build containers, and even deploy applications to cloud solutions such as Kubernetes and OpenShift. However, you may have noticed that our discussion so far has been centered around deploying single microservice applications such as Apache2, Memcached, NGINX, and MariaDB. These applications can be deployed standalone, without any dependency on other services or applications aside from a basic Docker daemon. While learning containerization from building single-container microservices is a great way to learn the core concepts of containerization, it isn't an accurate reflection of real-world application infrastructures.

As you may already know, applications usually comprise stacks of interconnected software that work together to deliver a service to end users. A typical application stack might involve a web frontend that receives input from a user. The web interface might be responsible for knowing how to contact a database backend to store the data provided to it by the user, as well as retrieve previously stored data. Big data applications might periodically analyze the data within the database in an attempt to figure out trends in data, analyze usage, or perform other functions that give data scientists insight into how users are operating the application. These applications live in a delicate balance that's dependent on network connectivity, DNS resolution, and service discovery in order to talk to each other and perform their overarching functions.

The world of containers is not very different at the outset. After all, the containerized software still draws dependencies on other containerized and non-containerized applications to store, retrieve, and process data, and perform distinct functions. As we touched on in `Chapter 5`, *Containers at Scale with Kubernetes,* and `Chapter 6`, *Managing Containers with OpenShift*, containers bring a lot more versatility and much reduced management complexity to the problem

of deploying and scaling multi-tiered applications.

In this chapter we will cover the following topics:

- Defining complex applications using Docker networks
- Exploring the Ansible Container django-gulp-nginx project
- Building the django-gulp-nginx project
- Development and production configurations
- Deploying the project to OpenShift

Defining complex applications using Docker networking

Containerized environments are dynamic and apt to change state quickly. Unlike traditional infrastructure, containers are continually scaling up and down, perhaps even migrating between hosts. It is critical that containers are able to discover other containers, establish network connectivity, and share resources quickly and efficiently.

As we touched on in previous chapters, Docker, Kubernetes, and OpenShift have the native functionality to automatically discover and access other containers using various networking protocols and DNS resolution, not unlike bare-metal or virtualized servers. When deploying containers on a single Docker host, Docker will assign each container an IP address in a virtual subnet that can be used to talk to other container IP addresses in the same subnet. Likewise, Docker will also provide simple DNS resolution that can be used to resolve container names internally. When scaled out across multiple hosts using container orchestration systems such as Kubernetes, OpenShift, or Docker Swarm, containers use an overlay network to establish network connectivity between hosts and run as though they exist on the same host. As we saw in `Chapter 5`, *Containers at Scale with Kubernetes*, Kubernetes provides a sophisticated internal DNS system to resolve containers based on namespaces within the larger Kubernetes DNS domain. There is a lot to be said about container networking, so for the purposes of this chapter, we will look at Docker networking for service discovery. In this section, we will create a dedicated Docker network subnet and create containers that leverage DNS to establish network connectivity to other running containers.

To demonstrate basic network connectivity between Docker containers, let's use the Docker environment in our Vagrant lab host to create a new virtual container network using the bridge networking driver. Bridge networking is one of the most basic types of container networks that is limited to a single Docker host. We can create this using the `docker network create` command. In this example, we will create a network called `SkyNet` using the `172.100.0.0/16` CIDR block, with the `bridge` networking driver:

```
ubuntu@node01:~$ docker network create -d bridge --subnet 172.100.0.0/16 SkyNet

2679e6a7009912fbe5b8203c83011f5b3f3a5fa7c154deebb4a9aac7af80a6aa
```

We can validate this network has been successfully created using the `docker network ls` command: **ubuntu@node01:~$ docker network ls NETWORK ID NAME DRIVER SCOPE 2679e6a70099 SkyNet bridge local truncated..**

We can see detailed information about this network in JSON format using the `docker network inspect` command: **ubuntu@node01:~$ docker network inspect SkyNet [{ "Name": "SkyNet", "Id": "2679e6a7009912fbe5b8203c83011f5b3f3a5fa7c154deebb4a9aac7af80a6aa", "Created": "2017-11-05T02:26:22.790958921Z", "Scope": "local", "Driver": "bridge", "EnableIPv6": false, "IPAM": { "Driver": "default", "Options": {}, "Config": [{ "Subnet": "172.100.0.0/16" }] },**

Now that we have established a network on our Docker host, we can create containers to connect to this network to test the functionality. Let's create two Alpine Linux containers to connect to this network and use them to test DNS resolution and reachability. The Alpine Linux Docker image is an extremely lightweight container image that can be used to quickly spin up containers for testing purposes. In this example, we will create two Alpine Linux containers named `service1` and `service2`, connected to the SkyNet Docker network using `--network` flag: **ubuntu@node01:~$ docker run --network=SkyNet -itd --name=service1 alpine Unable to find image 'alpine:latest' locally latest: Pulling from library/alpine b56ae66c2937: Pull complete Digest: sha256:d6bfc3baf615dc9618209a8d607ba2a8103d9c8a405b3bd8741d88b4be Status: Downloaded newer image for alpine:latest 5f1fba3964fae85e90cc1b3854fc443de0b479f94af68c14d9d666999962e25a**

In a similar way, we can start the `service2` container, using the `SkyNet` network:

```
ubuntu@node01:~$ docker run --network=SkyNet -itd --name=service2 alpine
8f6ad6b88b52e446cee44df44d8eaa65a9fe0d76a2aecb156fac704c71b34e27
```

> *Although these containers are not running a service, they are running by allocating a pseudo-tty instance to them using the `-t` flag. Allocating a pseudo-tty to the container will keep it from immediately exiting, but will cause the container to exit if the TTY*

session is terminated. Throughout this book, we have looked at running containers using command and entrypoint arguments, which is the recommended approach. Running containers by allocating a pseudo-tty is great for quickly spinning up containers for testing purposes, but not a recommended way to run traditional application containers. Application containers should always run based on the status of the ***process ID (PID)*** *running within it.*

In the first example, we can see that our local Docker host pulled down the latest Alpine container image and ran it using the parameters we passed into the `docker run` command. Likewise, the second `docker run` command created a second instance of this container image using the same parameters. Using the `docker inspect` command, we can see which IP addresses the Docker daemon assigned our containers: **ubuntu@node01:~$ docker inspect service1**
TRUNCATED..
"NetworkID":
"2679e6a7009912fbe5b8203c83011f5b3f3a5fa7c154deebb4a9aac7af80a6aa",
"EndpointID":
"47e16d352111007b9f19caf8c10a388e768cc20e5114a3b346d08c64f1934e1f",
"Gateway": "172.100.0.1",
"IPAddress": "172.100.0.2",
"IPPrefixLen": 16,
"IPv6Gateway": "",
"GlobalIPv6Address": "",
"GlobalIPv6PrefixLen": 0,
"MacAddress": "02:42:ac:64:00:02",
"DriverOpts": null

And we can do the same for `service2`:

```
ubuntu@node01:~$ docker inspect service2
TRUNCATED..
"NetworkID": "2679e6a7009912fbe5b8203c83011f5b3f3a5fa7c154deebb4a9aac7af80a6aa",
"EndpointID": "3ca5485aa27bd1baffa826b539f905b50005c9157d5a4b8ba0907d15a3ae7a21",
"Gateway": "172.100.0.1",
"IPAddress": "172.100.0.3",
"IPPrefixLen": 16,
"IPv6Gateway": "",
"GlobalIPv6Address": "",
"GlobalIPv6PrefixLen": 0,
"MacAddress": "02:42:ac:64:00:03",
"DriverOpts": null
```

As you can see, Docker assigned the IP address of `172.100.0.2` to our `service1` container, and the IP address of `172.100.0.3` to our `service2` container. These IP addresses provide network connectivity exactly as you would expect between two hosts attached to the same network segment. If we use `docker exec` to log into the `service1` container, we can check to see whether `service1` can ping `service2` using the IP addresses Docker assigned: **ubuntu@node01:~$ docker exec -it service1 /bin/sh / # ping 172.100.0.3 PING 172.100.0.3 (172.100.0.3): 56 data bytes 64 bytes from 172.100.0.3: seq=0 ttl=64 time=0.347 ms 64 bytes from 172.100.0.3: seq=1 ttl=64 time=0.160 ms 64 bytes from 172.100.0.3: seq=2 ttl=64 time=0.159 ms**

Since these containers are running using a pseudo-tty instead of a command or entrypoint, simply typing `exit` in the container shell will kill the TTY session and stop the container. To keep the container running when exiting the shell, use the Docker escape sequence from your keyboard: Ctrl + P Ctrl + Q.

We can as well do this test likewise from the `service2` container: **ubuntu@node01:~$ docker exec -it service2 /bin/sh / # ping 172.100.0.2 PING 172.100.0.2 (172.100.0.2): 56 data bytes 64 bytes from 172.100.0.2: seq=0 ttl=64 time=0.175 ms 64 bytes from 172.100.0.2: seq=1 ttl=64 time=0.157 ms**

It is easy to see that IP-based networking works well to establish network connectivity between running containers. The downside of this approach is that we cannot always know ahead of time what IP addresses the container runtime environment will assign our containers. For example, a container may require an entry in a configuration file to point to a service it depends on. Although you might be tempted to plug an IP address into your container role and build it, this container role would have to be rebuilt for each and every environment it is deployed into. Furthermore, when containers get stopped and restarted, they could take on different IP addresses, which will cause the application to break down. Luckily, as a solution to this issue, Docker provides a DNS resolution based on the container name, which will actively keep track of running containers and resolve the correct IP address in the event that a container should change IP addresses. Container names, unlike IP addresses, can be known in advance and be used to point containers to the correct services inside of configuration files, or stored in memory as environment variables. We can see

this in action by logging back into the `service1` container and using the `ping` command to ping the name `service2`: **ubuntu@node01:~$ docker exec -it service1 /bin/sh / # ping service2 PING service2 (172.100.0.3): 56 data bytes 64 bytes from 172.100.0.3: seq=0 ttl=64 time=0.233 ms 64 bytes from 172.100.0.3: seq=1 ttl=64 time=0.142 ms 64 bytes from 172.100.0.3: seq=2 ttl=64 time=0.184 ms 64 bytes from 172.100.0.3: seq=3 ttl=64 time=0.263 ms**

Furthermore, we can create a third service container and check to see if the new container has the ability to resolve the names of `service1` and `service2` respectively: **ubuntu@node01:~$ docker run --network=SkyNet -itd --name=service3 alpine 8db62ae30457c351474d909f0600db7f744fb339e06e3c9a29b87760ad6364ff ubuntu@node01:~$ docker exec -it service3 /bin/sh / # ping service1 PING service1 (172.100.0.2): 56 data bytes 64 bytes from 172.100.0.2: seq=0 ttl=64 time=0.207 ms 64 bytes from 172.100.0.2: seq=1 ttl=64 time=0.165 ms 64 bytes from 172.100.0.2: seq=2 ttl=64 time=0.159 ms ^C --- service1 ping statistics --- 3 packets transmitted, 3 packets received, 0% packet loss round-trip min/avg/max = 0.159/0.177/0.207 ms / # / # ping service2 PING service2 (172.100.0.3): 56 data bytes 64 bytes from 172.100.0.3: seq=0 ttl=64 time=0.224 ms 64 bytes from 172.100.0.3: seq=1 ttl=64 time=0.162 ms 64 bytes from 172.100.0.3: seq=2 ttl=64 time=0.146 ms**

Finally, if we log into the `service2` container, we can use the `nslookup` command to resolve the IP address of the newly created `service3` container: **ubuntu@node01:~$ docker exec -it service2 /bin/sh / # nslookup service3 Name: service3 Address 1: 172.100.0.4 service3.SkyNet**

Docker creates DNS resolution using the name of the Docker network as a domain. As such, the `nslookup` results are showing the fully qualified domain name of `service3` as `service3.SkyNet`. However, as I'm sure you could imagine, having DNS resolution for containers is an incredibly powerful tool for building reliable and robust containerized infrastructures. Just by knowing the name of the container, you can establish links and dependencies between containers that will scale with your infrastructure. This concept extends far beyond learning the individual IP addresses of containers. For example, as we saw in `Chapter 5`, *Containers at Scale with Kubernetes*, and `Chapter 6`, *Managing Your Applications with OpenShift*, Kubernetes and OpenShift allow for the creation of services that logically connect to backend pods using labels or other identifiers. When other

pods pass traffic to the service DNS entry, Kubernetes will load-balance traffic to the running pods that match the label rules configured in the service entry. The only thing the containers that rely on that service need to know is how to resolve the service FQDN, and the container orchestrator takes care of the rest. The backend pods could scale up or down, but as long as the container orchestrator's DNS service is able to resolve the service entry, the other containers calling the service will not notice a difference.

Exploring the Ansible Container django-gulp-nginx project

Now that we have a basic understanding of container networking concepts and Docker DNS resolution, we can build projects that have multi-container dependencies. Ansible Container has a concept of creating fully reusable full stack containerized applications, aptly named Container Apps. Container Apps are able to be downloaded and deployed quickly from Ansible Galaxy very similar to container-enabled roles. Container Apps have the benefit of allowing users to get started developing quickly against fully functional multi-tier applications that run as separate microservice containers. In this example, we will use a community-developed web application project that spins up a Python-based Django, Gulp, and NGINX environment we can deploy locally and to a container orchestration environment such as OpenShift or Kubernetes.

You can explore a wide range of container apps using Ansible Galaxy by simply going to the Ansible Galaxy website at `https://galaxy.ansible.com`, selecting BROWSE ROLES, clicking on Role Type from the Keyword drop-down box, and selecting Container App from the search dialog:

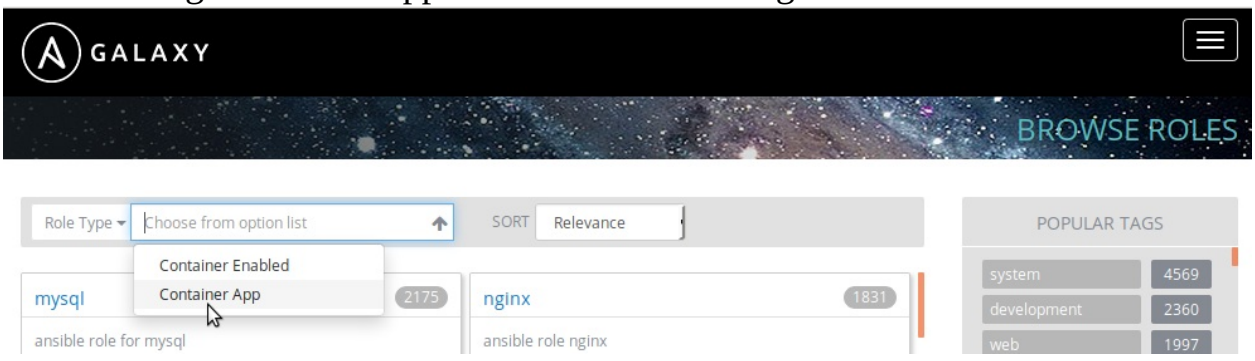


Figure 1: Searching for Container Apps in Ansible Galaxy In this example, we are going to leverage the pre-built Ansible `django-gulp-nginx` Container App, which is an official Ansible Container project. This container app creates a containerized Django framework web application that leverages NGINX as a web server, Django and Gulp as a framework, and PostgreSQL as a database server. In this project is an entirely self-contained demo environment we can use to explore how Ansible Container works with other services and dependencies.

In order to get started with using this project, we need to first install it in a clean directory on our Vagrant Lab VM. First, create a new directory (I will call mine `demo`), and run the `ansible-container init` command followed by the name of the Container App we want to install, `ansible.django-gulp-nginx`. You can find the full name for this project on Ansible Galaxy, using the preceding steps to search for Container Apps. Following code demonstrates creating a new directory and initializing the Django-Gulp-NGINX project: **ubuntu@node01:~$ mkdir demo/**

ubuntu@node01:~$ cd demo/

ubuntu@node01:~$ ansible-container init ansible.django-gulp-nginx

Ansible Container initialized from Galaxy container app 'ansible.django-gulp-nginx'

Upon successfully initializing the project, you should see the Ansible Container initialized from Galaxy Container App `ansible.django-gulp-nginx` message appear. This indicates that the container app was successfully installed from Ansible Galaxy. Executing the `ls` command in the `demo/ directory` should display project files similar to the following:

```
ubuntu@node01:~/demo$ ls bower.json     dist         Makefile   meta.yml      package.j
```

A lot of the files listed are configuration files that support the Gulp/Django framework for the application we are going to create. The primary file we are concerned with for the purposes of this demonstration is the core file in all Ansible Container projects: `container.yml`. If you open the `container.yml` file in a text editor, it should resemble the following: version: '2'

settings:

conductor:

base: 'centos:7'

volumes:

- temp-space:/tmp # Used to copy static content between containers

```
k8s_namespace:
name: demo
display_name: Ansible Container Demo
description: Django framework demo
defaults:
POSTGRES_USER: django
POSTGRES_PASSWORD: sesame
POSTGRES_DB: django
DJANGO_ROOT: /django
DJANGO_USER: django
DJANGO_PORT: 8080
DJANGO_VENV: /venv
NODE_USER: node
NODE_HOME: /node
NODE_ROOT: ''
GULP_DEV_PORT: 8080
services:
django:
from: 'centos:7'
roles:
```

- role: django-gunicorn

environment:

DATABASE_URL: 'pgsql://{{ POSTGRES_USER }}:{{ POSTGRES_PASSWORD }}@postgresql:5432/{{ POSTGRES_DB }}'

DJANGO_ROOT: '{{ DJANGO_ROOT }}'

DJANGO_VENV: '{{ DJANGO_VENV }}'

expose:

- '{{ DJANGO_PORT }}'

working_dir: '{{ DJANGO_ROOT }}'

links:

- postgresql

user: '{{ DJANGO_USER }}'

command: ['/usr/bin/dumb-init', '{{ DJANGO_VENV }}/bin/gunicorn', -w, '2', -b, '0.0.0.0:{{ DJANGO_PORT }}', 'project.wsgi:application']

entrypoint: [/usr/bin/entrypoint.sh]

dev_overrides:

volumes:

- '$PWD:{{ DJANGO_ROOT }}'

command: [/usr/bin/dumb-init, '{{ DJANGO_VENV }}/bin/python', manage.py, runserver, '0.0.0.0:{{ DJANGO_PORT }}']

depends_on:

- postgresql

```
gulp:
from: 'centos:7'
roles:
- role: gulp-static
working_dir: '{{ NODE_HOME }}'
command: ['/bin/false']
environment:
NODE_HOME: '{{ NODE_HOME }}'
dev_overrides:
entrypoint: [/entrypoint.sh]
command: [/usr/bin/dumb-init, /usr/local/bin/gulp]
ports:
- '8080:{{ GULP_DEV_PORT }}'
- 3001:3001
links:
- django
volumes:
- '$PWD:{{ NODE_HOME }}'
openshift:
```

state: absent

nginx:

from: 'centos:7'

roles:

- role: ansible.nginx-container

ASSET_PATHS:

- /tmp/dist

PROXY: yes

PROXY_PASS: 'http://django:8080'

PROXY_LOCATION: "~* /(admin|api)"

ports:

- '{{ DJANGO_PORT }}:8000'

links:

- django

dev_overrides:

ports: []

command: /bin/false

postgresql:

```
# Uses a pre-built postgresql image from Docker Hub
from: ansible/postgresql:latest
environment:
- 'POSTGRES_DB={{ POSTGRES_DB }}'
- 'POSTGRES_USER={{ POSTGRES_USER }}'
- 'POSTGRES_PASS={{ POSTGRES_PASSWORD }}'
- 'PGDATA=/var/lib/pgsql/data/userdata'
volumes:
- postgres-data:/var/lib/pgsql/data
expose:
- 5432


volumes:
postgres-data:
docker: {}
openshift:
access_modes:
- ReadWriteMany
requested_storage: 3Gi
```

temp-space:

docker: {}

openshift:

state: absent

registries:

local_openshift:

url: https://local.openshift

namespace: demo

pull_from_url: 172.30.1.1:5000

The output shown here is a reflection of the contents of `container.yml` at the time of writing. Yours may look slightly different if updates have been made to this project since the time of writing.

As you can see, this `container.yml` file contains many of the same specifications we have covered already in previous chapters of the book. Out of the box, this project contains the service declarations to build the Gulp, Django, NGINX, and Postgres containers, complete with the role paths and various role variables defined to ensure the project is able to run in a completely self-contained format. Also built into this project is support for deploying this project to OpenShift. One of the benefits of this project is that it exposes virtually every possible configuration option available in an Ansible Container project, as well as the proper syntax to activate these features. Personally, I like to use this project as a reference guide in case I forget the proper syntax to use in my project's `container.yml` files. Following is a list of sections from the `container.yml` that are useful for the user to have an understanding of, starting from the top and moving towards the bottom:

- `conductor`: As we have seen throughout this book, this section defines the

conductor container and the base container image to build the conductor from. In this case, the conductor image will be a Centos 7 container that leverages a volume mount from the `temp-space` directory in the `root` of the project to the `/tmp` directory inside of the container. It is important to note here that the conductor image can leverage volume mounts in order to store data during the build process.

- `defaults`: This section is known as the top-level defaults section and is used to instantiate variables that can be used throughout the project. Here, you can define variables that can be used in the service section of the project as role variable overrides, or simply in place of hardcoding the same values over and over again in the `container.yml` file. It is important to note that in the order that, Ansible Container evaluates variable precedence, the top-level defaults section has the lowest precedence.
- `services`: In the `services` section, we see entries for the core service that will run in this stack (`django`, `gulp`, `nginx`, and `postgresql`). This section, for the most part, should be reviewed based on what we have covered in previous chapters up until this point. However, you will notice that, in the container definition for the `django` container, there is a `link` line that specifies the `postgresql` container name. You will notice this as well in the other container definitions that list the name of the `django` container. In previous versions of Docker, links were a way of establishing networking connectivity and container name resolution for individual containers. However, recent versions of Docker have deprecated the `link` syntax in favor of the native container name resolution built into the Docker networking stack. It is important to note that many projects still use links as a way to establish network dependencies and container name resolution, but will most likely be removed in future versions of Docker. Container orchestration tools such as Kubernetes and OpenShift also ignore the `link` syntax since they only use native DNS services to resolve other containers and services. Another aspect I would like to draw the readers attention to in the `services` section is the `nginx`, `gulp`, and `django` containers have a new sub-section titled `dev-overrides`. This section is for specifying container configuration that will only be present when building testing containers locally. Usually, developers use `dev-overrides` to run containers with verbose debugging output turned on, or other similar logging mechanisms are used to troubleshoot potential issues. The `dev-override` configuration will be ignored when using the `--production` flag when executing `ansible-container run`.

- `volumes`: The top-level volumes section is used to specify **persistent volume claims** (**PVCs**) that continue to exist even if the container is stopped or destroyed. This section normally maps volumes that have already been created in the container-specific services section of the `container.yml` file to provide a more verbose configuration for how the container orchestrator should handle the persistent volume claim. In this case, the `postgres-data` volume that has been mapped in the PostgreSQL container is given the OpenShift specific configuration of `ReadWriteMany` access mode, as well as 3 GB of storage. PVCs are usually required for applications dependent on storing and retrieving data, such as databases or storage APIs. The overall goal of PVCs is that we do not want to lose data if need to redeploy, upgrade, or migrate the container to another host.

Building the django-gulp-nginx project

Now that we have a firm understanding of some of the more advanced Ansible Container syntax that is commonly found in Container Apps, we can apply the knowledge we have learned so far of the Ansible Container workflow to build and run the container App. Since container apps are full Ansible Container projects complete with roles, a `container.yml` file, and other supporting project data, the same Ansible Container workflow commands we used previously can be used here with no modifications. When you are ready, execute the `ansible-container build` command in the `root` directory of the project:

ubuntu@node01:~/demo$ ansible-container build Building Docker Engine context...

Starting Docker build of Ansible Container Conductor image (please be patient)...

Parsing conductor CLI args.

Docker™ daemon integration engine loaded. Build starting. project=demo Building service... project=demo service=django

PLAY [django] *

**TASK [Gathering Facts] **

ok: [django]

TASK [django-gunicorn : Install dumb init]
****

changed: [django]

TASK [django-gunicorn : Install epel]
****

changed: [django]

TASK [django-gunicorn : Install python deps]
****

changed: [django] => (item=[u'postgresql-devel', u'python-devel', u'gcc', u'python-virtualenv', u'nc', u'rsync'])

TASK [django-gunicorn : Make Django user]
****

changed: [django]

TASK [django-gunicorn : Create /django]
****

changed: [django]

TASK [django-gunicorn : Make virtualenv dir]
****

changed: [django]

TASK [django-gunicorn : Setup virtualenv]

****

changed: [django]

TASK [django-gunicorn : Copy core source items]
****

changed: [django] => (item=manage.py)

changed: [django] => (item=package.json) changed: [django] => (item=project)

changed: [django] => (item=requirements.txt) changed: [django] => (item=requirements.yml) TRUNCATED...

Since the Container App is building four service containers, it may take a little longer than usual for the build process to complete. If you are following along, you will see Ansible Container go through each playbook role individually as it creates the containers and works to bring them into the desired state described in the playbooks. When the build has completed successfully, we can execute `ansible-container run` command to start the containers and bring our new web service online: **ubuntu@node01:~/demo$ ansible-container run Parsing conductor CLI args.**

Engine integration loaded. Preparing run. engine=Docker™ daemon Verifying service image service=django

Verifying service image service=gulp

Verifying service image service=nginx

PLAY [Deploy demo]
****

TASK [docker_service] *

changed: [localhost]

**PLAY RECAP **

localhost : ok=1 changed=1 unreachable=0 failed=0

All services running. playbook_rc=0

Conductor terminated. Cleaning up. command_rc=0 conductor_id=3109066bdb82a46e0b44fdbbbeaaa02fe8daf7bc18600c0c8466e save_container=False

When the run playbooks have finished executing, the service containers should be running on the Vagrant VM in developer mode, since the `container.yml` file specifies `dev-overrides` for many of the services. It is important to note that `ansible-container run` will, by default, run the service containers according to any `dev-override` configuration listed in the `container.yml` file. For example, one developer override configured is to not run the NGINX container when running in developer mode. This is accomplished by setting a developer override option for the NGINX container so that it will run `/bin/false` as the initial container command, immediately killing it. Executing the `docker ps -a` command will show that the `postgresql`, `django`, and `gulp` containers are running, with NGINX in a stopped state. Using the developer overrides, NGINX is stopped and `gulp` is responsible for serving up the HTML page:

```
ubuntu@node01:~/demo$ docker ps -a CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS  PORTS

c3e3c2e07427  demo-gulp:20171107030355    "/entrypoint.sh /u..."   56 se
```

Once the containers have started, the `django-gulp-nginx` Container App will be

listening on the Vagrant lab VM's localhost address at port `8080`. We can use the `curl` command to test the application and ensure we are able to get the default Hello World simple HTML page response the service is designed to provide:
ubuntu@node01:~/demo$ curl http://localhost:8080

**<!DOCTYPE html><html lang="en-US"><head><title></title><meta charset="UTF-8"><meta http-equiv="X-UA-Compatible" content="IE=edge"><meta name="viewport" content="width=device-width,initial-scale=1"><link rel="stylesheet" href="style.css"></head> <body> Hello <ul> world !
users !
you!
everybody ! <script src="js/bundle.min.js"></script></body></html>**

Development versus production configurations

By default, executing the `ansible-container run` command on a project that specifies developer-overrides for a given service will run the service with the developer overrides active. Often, the developer overrides expose verbose logging or debugging options in an application that a developer would not want a general end user to be exposed to, not to mention that it can be quite resource-intensive to run applications with verbose logging stack tracing running constantly. The `ansible-container run` command has the ability to be run with the `--production` flag to specify when to run services in a mode that mimics a production-style deployment. Using the `--production` flag ignores the `dev_overrides` sections in the `container.yml` file and runs the services as explicitly defined in the `container.yml` file. Now that we have verified that our web service is able to run and function in developer mode, we can try running our service in production mode to mimic a full production deployment on our local workstation.

First, we will need to run `ansible-container stop` in order to stop all running container instances in developer mode: **ubuntu@node01:~/demo$ ansible-container stop Parsing conductor CLI args.**

Engine integration loaded. Preparing to stop all containers. engine=Docker™ daemon

PLAY [Deploy demo] *

TASK [docker_service] *

changed: [localhost]

PLAY RECAP
****

localhost : ok=1 changed=1 unreachable=0 failed=0

All services stopped. playbook_rc=0

Conductor terminated. Cleaning up. command_rc=0 conductor_id=8ab40a594ec72012afdf0abc31ff527925fc5960e4ecbb40eeb1676 save_container=False

Next, let's re-run the `ansible-container run` command, this time providing the `--production` flag to indicate that we wish to ignore the developer overrides and run this service in production mode: **ubuntu@node01:~/demo$ ansible-container run --production Parsing conductor CLI args.**

Engine integration loaded. Preparing run. engine=Docker™ daemon Verifying service image service=django

Verifying service image service=gulp

Verifying service image service=nginx

PLAY [Deploy demo]
****

TASK [docker_service]
****

changed: [localhost]

PLAY RECAP
****

localhost : ok=1 changed=1 unreachable=0 failed=0

All services running. playbook_rc=0

Conductor terminated. Cleaning up. command_rc=0 conductor_id=1916f63a843d490ec936672528e507332ef408363f65387256fe8a7 save_container=False

If we now look at the services running, you will notice that the NGINX server container is now running and acting as the frontend service for the web traffic on port `8080` instead of the Gulp container. Meanwhile, the Gulp container has been started with the default command `/bin/false`, which instantly kills the container. In this example, we have introduced a production configuration that terminates a development HTTP web server, in favor of a production-ready NGINX web server: **ubuntu@node01:~/demo$ docker ps -a CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES**

1aabc9745942 demo-nginx:20171107031508 "/usr/bin/dumb-ini..." 7 seconds ago 16154bbfae54 demo-django:20171107031311 "/usr/bin/entrypoi..." 14 seconds ago ea2ec92e9c50 demo-gulp:20171107030355 "/bin/false" Exited (1) 15

9660b816e86f ansible/postgresql:latest "/usr/bin/entrypoi..." 20 minutes ago

We can finally test the web service once more to ensure that the service is reachable and running on the Vagrant Lab VM on localhost port `8080`: **ubuntu@node01:~/demo$ curl http://localhost:8080**

**<!DOCTYPE html><html lang="en-US"><head><title></title><meta charset="UTF-8"><meta http-equiv="X-UA-Compatible" content="IE=edge"><meta name="viewport" content="width=device-width,initial-scale=1"><link rel="stylesheet" href="style.css"></head> <body> Hello **

world !
users !
you!
everybody !
<script src="js/bundle.min.js"></script></body></html>

Deploying the project to OpenShift

So far, we have looked at how to run the demo web application locally using the production and development configurations provided by the `dev_override` syntax. Now that we have an understanding of how the web application functions and leverages other services, we can look at how to deploy this application in a production-grade container orchestration environment such as OpenShift or Kubernetes. In this section of the book, we will deploy this project using the production configuration into the local Minishift cluster we created in `Chapter 6`, *Managing Applications with OpenShift*. Prior to starting this example, make sure you have a valid OpenShift credentials file that works with your local cluster, in the `/home/ubuntu/.kube/config` directory. If new OpenShift credentials need to be created, be sure to turn back to `Chapter 7`, *Deploying Your First Project*, for more details.

In order to ensure our application can be deployed to OpenShift, we need to modify the container app's `container.yml` file so it points to our Kubernetes configuration file as well as to the Docker Hub registry for pushing our container images.

OpenShift comes with an integrated container registry you can use to push container images to during the `ansible-container deploy` process. However, it requires some additional configuration that is beyond the scope of this book. For now, it will be sufficient to use the Docker Hub registry as we have throughout this book so far.

In the `settings` section of the `container.yml` file, we will add a `k8s_auth` stanza to point to the Kubernetes configuration file the OC generated: k8s_namespace: name: demo display_name: Ansible Container Demo description: Django framework demo k8s_auth: config_file: /home/ubuntu/.kube/config

Next, in the `registries` section, we will add an entry for the Docker Hub container registry, using our user credentials: registries: docker: url:

https://index.docker.io/v1/ namespace: aric49

Now that we have OpenShift and Docker Hub configured in our project, we can use the `ansible-container deploy` command with the `--engine openshift` flag to generate the OpenShift deployment and push the image artifacts to Docker Hub. In order to differentiate the images, we can push them to Docker Hub using the `containerapp` tag. Since we are pushing multiple images to Docker Hub, depending on your internet connection speed, it may take a few minutes for this process to complete:

```
ubuntu@node01:~/demo$ ansible-container --engine openshift deploy --push-to docker --us
Enter password for aric49 at Docker Hub:
Parsing conductor CLI args.
Engine integration loaded. Preparing push.      engine=OpenShift™
Tagging aric49/demo-django
Pushing aric49/demo-django:containerapp...
The push refers to a repository [docker.io/aric49/demo-django]
Preparing
Pushing
Mounted from library/centos
Pushing
Pushed
containerapp: digest: sha256:983afc3cb7c0f393d20047d0a1aa75a94a9ab30a2f3503147c09b55a81
Tagging aric49/demo-gulp
Pushing aric49/demo-gulp:containerapp...
The push refers to a repository [docker.io/aric49/demo-gul
TRUNCATED...
```

Once the deploy process has completed successfully, we can use the `ansible-container run` command with the `--engine openshift` flag to launch our application and run it in our simulated OpenShift production environment. Don't forget to specify the `--production` flag so that our service gets deployed using the production configuration and not the developer overrides:

ubuntu@node01:~/demo$ ansible-container --engine openshift run --production Parsing conductor CLI args. Engine integration loaded. Preparing run. engine=OpenShift™ Verifying service image service=django Verifying service image service=gulp Verifying service image service=nginx PLAY [Manage the lifecycle of demo on OpenShift?]

********** **TASK [Create project demo]**

********** **changed: [localhost] TASK [Create service]**

****

changed: [localhost] TASK [Create service]

****

changed: [localhost] TASK [Create service]

****

changed: [localhost] TASK [Remove service]
********** **ok: [localhost] TASK [Create deployment, and scale replicas up]**
********** **changed: [localhost] TASK [Create deployment, and scale replicas up]** **********
changed: [localhost] TRUNCATED..

Once the process has completed successfully, we can log into the OpenShift web console to validate the service is running as expected. Unless it's otherwise changed, the Container App was deployed into a new project called `demo`, but will be displayed with the name `Ansible Container Demo` in the web interface, as per our `container.yml` configuration:

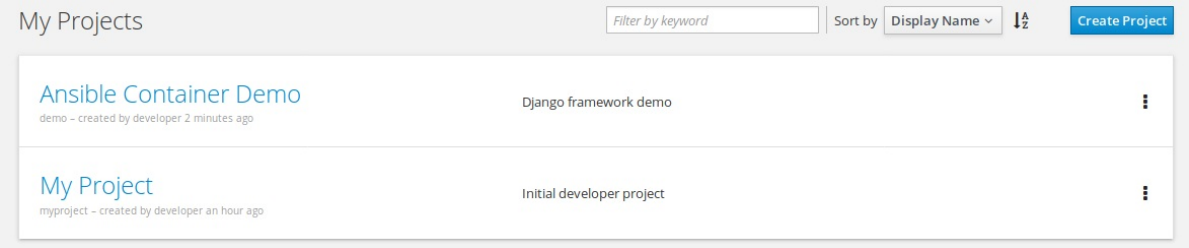


Figure 2: The Ansible Container Demo project deployed to OpenShift Clicking on the Ansible Container Demo project display name will show you the standard OpenShift dashboard demonstrating the running pods according to the production configuration. You should see the `django`, `ngnix`, and `postgresql` pods running, along with a link to the route created to access the web application in the upper-right corner of the console display:

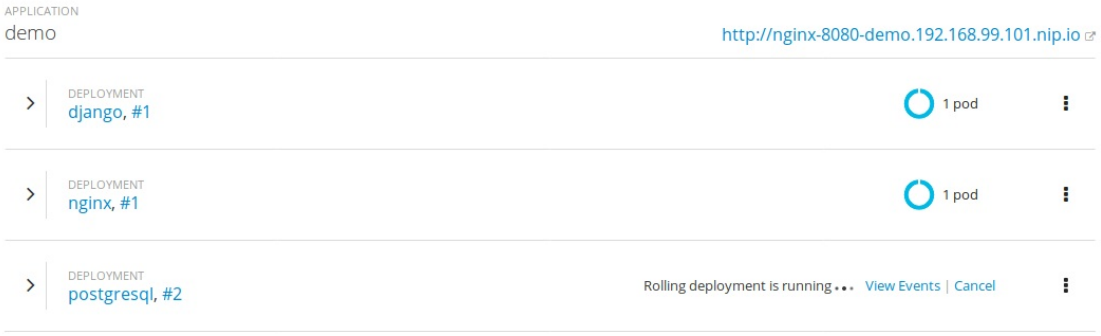


Figure 3: Running pods in the demo project We can test to ensure our application is running by clicking on the `nip.io` route created in OpenShift and ensuring the NGINX web server container is reachable. Clicking on the link should show the simple `Hello you!` Django application in its full glory:

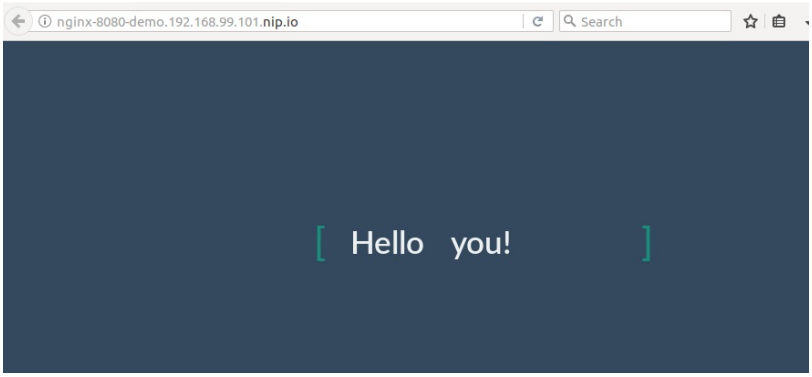


Figure 4: The Hello World page as viewed running in OpenShift That looks a lot nicer then the `curl` testing we were running in the local Vagrant lab, don't you think? Congratulations, you have successfully deployed a multi-container application into a simulated production environment!

From the OpenShift console, we can validate that the various aspects of our deployment are present and functioning as intended. For example, you can click on the `Storage` link in the left-hand navigation bar to validate that the PVC Postgres data was created and is functional in OpenShift. Clicking on postgres-data will show the details of the PVC object, including the allocated storage (3 GiB), and the access modes configured in the `container.yml` file, `Read-Write-Many`:

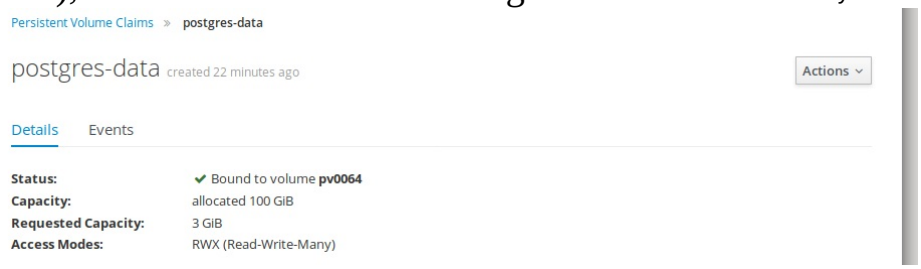


Figure 5: PostgreSQL PVC

References

- **Ansible django-gulp-nginx project**: https://github.com/ansible/django-gulp-nginx/
- **Docker networking documentation**: https://docs.docker.com/engine/userguide/networking/

Summary

As we are nearing the end of our journey with Ansible Container, we have covered what is perhaps the final hurdle in our quest to learn about automating containers using the Ansible Container project, working with multi-container projects. Due to the inherent networking functionality available in almost all container runtime environments, such as Docker, Kubernetes, and OpenShift, building streamlined microservice software stacks is a breeze. As we have seen throughout this chapter, microservice containers can easily be connected with Lego-like efficiency to build and deploy robust applications in production.

Throughout this section, we have looked at how container runtime environments establish dependencies on other containers using the container networking fabric, as well as creating link dependencies. We observed how these concepts work together to build a rather complex multi-container application using Gulp, Django, NGINX, and Postgres containers. We tested this stack in developer mode using `dev_overrides`, as well as in production mode according to the project configuration. Finally, we deployed this application into our local OpenShift cluster to simulate a real-world production deployment, complete with container networking and persistent volume claims.

The final chapter of the book will cover ways in which you can expand your knowledge of Ansible Container and cover some practical tips on how to go forward in your knowledge of Ansible Container, carrying forward the knowledge you have obtained so far in this book.

Going Further with Ansible Container

In the introductory chapters of this book, we learned how trends in the IT industry have fundamentally shifted and shaped the ways in which applications and services are designed and deployed. With the rise of consumption of high-CPU and bandwidth-intensive services, consumers are regularly demanding more features, have zero-tolerance to outages, and want more options to consume services and applications. In response to this shift, no longer can monolithic application deployments and static servers be the backbone of this aging infrastructure. Even configuration management and automation tools, as dynamic as they are, cannot keep up with the need for organizations to continually scale out existing infrastructure footprints across a variety of platforms.

In response to this trend, containerization platforms such as Docker rose to the challenge to address the need deploy and manage applications consistently and reliably. Docker containers enable businesses and organizations to adopt a modular infrastructure footprint in which applications can be built entirely self-contained and guaranteed to run on any system that uses a compatible container runtime environment. This allows software developers and DevOps engineers to rapidly build microservice applications, similar in many respects to Lego bricks that can be stuck together to design large and complex software stacks.

While microservice applications seem to be the answer to many problems plaguing the industry today, conventional methods of building and deploying microservice applications are proving to be less robust and give the operators of these services fewer options to truly build and configure container images that meet the needs of organizations. The Ansible Container project seeks to meet this need by filling the gap between traditional configuration management and the container build and deploy pipeline. As we have seen throughout this book, Ansible Container can be leveraged to not only use the power of Ansible to build truly customized container images, but also to manage the life cycle of containerized software from development all the way to production.

In this, the final chapter of this book, I want to provide the reader with resources they can use to move forward with building and deploying containerized projects using Ansible Container, far beyond the scope of this book. Although we have covered all of the functional aspects of using and working with Ansible Container, microservice architecture is a rapidly growing field that is constantly changing and growing with the open source communities that author them. This chapter aims to point the reader in helpful directions that can be used as a starting point from which to grow in your knowledge of containerized software, provide helpful hints and tips for managing containers at scale, and also use your newly-found knowledge to help grow the open source communities around these projects.

In this final chapter, we will cover the following topics:

- Tips for writing roles and container apps
- Building powerful deployment playbooks with Ansible Container
- Tips for troubleshooting application containers
- CICD deployments with Jenkins or Travis CI
- Sharing roles and apps on GitHub and Ansible Galaxy
- Containerize everything

Tips for writing roles and container apps

If you are new to Ansible and playbook syntax, it can be quite easy to get confused when writing playbooks for the first time. Although Ansible is inherently a very readable and non-programmer friendly language to approach, there are a few *gotchas* one should keep in mind when writing roles or container apps to maximize usability.

Use full YAML syntax

A personal pet peeve of mine when working with Ansible code is when the author uses what I call the *condensed method* of writing a playbook. Essentially, functional Ansible code can be written so that module calls and attributes can be written on the same line, using an equals sign (=) to separate attributes and values, as seen in the following code.

Condensed method sample code:

```
- name: Deploy configuration file
  template: src=ConfigFile.j2 dest=/etc/myApp/myConfig.yml mode=0644

- name: Install Package
  apt: name=myApp state=present update_cache=true
```

Proper YAML syntax sample code:

```
- name: Deploy configuration file



  template:



    src: ConfigFile.j2



    dest: /etc/myApp/myConfig.j2



    mode: 0644
```

```
- name: Install Package

  apt:

    name: myApp

    state: present

    update_cache: true
```

As you can see, the condensed method and proper YAML syntax are functionally the same, but they are visually different. Using proper YAML syntax defines module calls across multiple lines and requires the user to indent module attributes underneath the definition of the module call itself. This makes playbooks much easier to read at a glance, and a lot easier to debug when looking for errors. Using proper YAML syntax also ensures that your text editor can perform proper syntax highlighting of your code since it conforms to standard YAML conventions. Using the condensed method, however, one can write Ansible code more *quickly*, but at the cost of readability and usability for others who might use your playbooks in the future. Not only is it visually unappealing, but makes reading the code and understanding the functionality a difficult experience. A best practice is to get into the habit of writing Ansible playbooks and roles using full YAML syntax and indentation. Others who use your code and contribute to it will thank you.

Use Ansible modules

When first starting writing Ansible playbooks and roles, it is quite tempting to use the `shell` or `command` modules for almost every task. If one has a firm understanding of BASH and the suite of GNU/Linux tools and utilities that come natively with most Linux-based operating systems, it is logical to want to build playbooks using the shell or command modules. The problem with this approach is that it ignores the family of over a thousand unique Ansible modules that ship out of the box with Ansible.

While `shell` and `command` do have their place under some circumstances, you should look first to see if there is an Ansible module that can programmatically do what you are trying to accomplish. The benefit of using an Ansible module, instead of running commands directly on the shell, is that Ansible modules have the ability to evaluate idempotency and take action only if the target is not in the desired state. While it is possible to use the command-line modules idempotently, it is far more difficult. Furthermore, Ansible modules have the unique ability to store and retrieve metadata about tasks in memory. For example, you could add the register line in a task definition to store the task metadata to a variable named `task_output`. Later in the playbook, you could check if that task performed a change on the system by checking if `task_output.changed == true` and take action accordingly. Similarly, this same logic can be used to check the return codes of tasks, search for metadata, or take actions if tasks have failed. Using modules gives you the freedom to leverage Ansible to work exactly the way you want it to.

Build powerful deployment playbooks with Ansible Core

As we have looked at throughout this chapter, Ansible Core is essentially the engine that works behind the scenes during the execution of a deployment. We looked at extracting these playbooks from the `ansible-deployment` directory and running them manually, passing in the corresponding tags to manually execute the run, stop, restart, and destroy functionality. However, these playbooks are generally limited and quite basic in form and function. Don't think for a moment that for deploying projects you are only limited to running the `ansible-container deploy` command, or executing the deployment playbooks manually. If you look at the deployment playbooks that are automatically generated, you will notice that they make calls to the `docker_service` module, which is a module featured in Ansible Core. Using a similar methodology, you can write your own playbooks to build completely custom deployments outside the scope of Ansible Container.

An excellent use case for this scenario might be that you have other services that are dependent on the status of the containerized project you built with Ansible Container. These services could be monitoring services, a database cluster, or even an external infrastructure API that you want your containers to pull data from during the launch process. Using a separate Ansible Core playbook, you could completely orchestrate the process of launching your containers and interacting with the dependent services. Here is an example code snippet, to help give you some inspiration. Notice we are defining the project name as a variable that we are also passing into the REST API call to register the service:

```
- name:Deploy New Service


  hosts: localhost
```

```
connection: local

gather_facts: no

vars:

  ProjectName: "MyAwesomeApp"


tasks:



  #Start the Container Service using the variable defined above

  - docker_service:

      project_name:"{{ ProjectName }}"

      definition:

          App:
```

```
            image: MyContainer:tag

            command: /usr/bin/dumb-init AwesomeApp

      register: ServiceStarted



#Register the service only when the container is updated (changed)

- name: Register Service in API

  uri:

    url: https://your.service.example.com/api/v2/

    method: POST
    body: "service: {{ ProjectName }}, state: deployed"
    body_format: yaml
  when: ServiceStarted.changed
```

As you can see, using Ansible Core playbooks to deploy containerized infrastructure can be an immensely powerful tool. When Ansible Core playbook modules to abstract your deployments, you are only limited by your imagination.

Troubleshooting application containers

Inevitably during the course of building containerized services and applications, you will encounter the need to troubleshoot misbehaving containers. Sometimes, containers fail to start due to a misconfigured `container start` command or entry point. Other times, the container itself starts to throw errors that need to be debugged or diagnosed. Most of the time, these issues can be looked at by examining the container logs or viewing the container runtime details in OpenShift, Docker, or Kubenetes. Following is a list of commands for the respective containerized runtime environment I have found the most helpful over the years:

- **Docker**:
  - `docker logs`: Use the `docker logs` command to view the standard out logs for any stopped, running, or exited container. Oftentimes, when containers have stopped, the last message they logged to standard out will confirm the cause of the container stop. Docker logs are also useful for debugging errors as they happen in the container in real-time. The full syntax for this command is `docker logs [container name or ID]`.
  - `docker inspect`: This can be used to view the all attributes and configuration details for almost any Docker resource, such as containers, networks, or storage volumes, in JSON format. `inspect` is useful for understanding how Docker itself sees the respective resource and whether it is picking up certain configuration parameters. The full syntax for inspect is `docker inspect [docker resource name or ID]`.
- **Kubernetes**:
  - `kubectl logs`: `kubectl logs` is used to view the logs of a pod running in Kubernetes. Often, the `kubectl logs` output will be similar to the `docker logs` output, if Kubernetes is using Docker as the underlying container runtime environment. However, using Kubernetes to natively relay the

log output allows the user to retrieve the logs using native Kubernetes abstractions, which may give an indication where the issue rests. It is also helpful to view the container runtime logs in conjunction with the Kubernetes logs. The full syntax for `kubectl logs` is `kubectl logs [full pod name] --namespace [namespace name]`.

- `kubectl describe`: `describe` is useful for viewing verbose output and configuration parameters for almost any Kubernetes resource. It can be used on cluster nodes, pods, namespaces, replica sets, and services, to name a few. Using `describe`, one can view resource labels, event messages, and other configuration options. The major benefit of `kubectl describe` is being able to describe almost any cluster resource to troubleshoot issues, whether it is a cluster node or a misbehaving pod. The full syntax for describe is `kubectl describe [resource type] [resource name] --namespace [namespace name if applicable]`.

- **OpenShift**:
  - `oc logs`: `oc logs` is quite similar to `kubectl logs` and `docker logs` in that it allows the user to view the logs specific to a specific running or restarting pod. Similar to Kubernetes, it is often quite helpful to compare the `oc logs` output with the output of `docker logs` to try and pinpoint the source of the issue. The full syntax for `oc logs` is `oc logs [pod or resource name]`.
  - `oc debug`: `debug` is used to create an exact copy of the troublesome pod or deployment to examine without interrupting the running service. Using `debug`, you can pass commands into the copied pod that can be used for debugging purposes, such as opening a shell or dumping environment variables. The full syntax for debug is `oc debug [pod, deployment, or resource name] -- [command to execute]`.

Create a build pipeline using Jenkins or TravisCI

In recent years, the idea of **continuous integration continuous deployment** (**CICD**) has taken the software development community by storm. Rich projects such as Jenkins, TeamCity, and TravisCI have given developers automated frameworks from which code changes can automatically be built, tested, and upon passing, deployed across the infrastructure. Leveraging CICD tools, developers using rapid software development methodologies such as Agile can deploy software faster and more reliably than ever before.

Most CICD tools and workflows function by defining jobs that have specific triggers they are listening for. These triggers could be code check-ins in a Git repository, users manually kicking off builds, or automated processes that call the CICD API directly. These jobs perform very specific and automated functions such as building code, running tests against the newly built code, and even handling the deployment of code changes. The CICD jobs even have the capability of sending notifications to chatrooms or email if a build, testing, or deploy step fails. Some are even sophisticated enough to notify the user who checked in the code if that particular change is breaking the build. The major benefit of CICD rests in the automation it provides to software developers to ensure these jobs are performed reliably, consistently, and regularly.

Imagine for a moment what a huge benefit CICD tools could provide to Ansible Container projects:

- `ansible-container build` automatically executed upon checking code into a certain Git repository branch
- `ansible-container run` executed on the CICD host to bring up the containers locally and fire off smoke tests to ensure the containers are running as expected
- If the build and testing steps pass, `ansible-container push` could be executed on the images, which tags them with a specific build number and pushes them to a container image registry such as Docker Hub

- Automatically kicking off `ansible-container deploy` and `ansible-container run` to deploy projects into specific environments (even production!)

Using automated CICD tools, you can take steps towards moving your infrastructure further towards a fully automatic build and deploy pipeline that does not involve human intervention at all. If you are curious to see what CICD tools can do for you and your workflow, I would suggest that you sign up for a free Travis CI account at `https://travisci.org`. If you are interested in a fully free and open source solution to deploy in your own infrastructure, I would recommend Jenkins: `https://jenkins.io`.

Additionally, you can look at the Travis CI build pipelines that build some of the Ansible Container projects we worked through throughout this book. For example, you can find the `django-gulp-nginx` project here: `https://travis-ci.org/ansible/django-gulp-nginx`.

Share roles and apps on GitHub and Ansible Galaxy

Ansible Galaxy is an absolutely fantastic resource to leverage to reuse some of the best playbooks and roles developed by the community. As we have seen throughout the book, Ansible Galaxy hosts hundreds of Ansible Core roles, container roles, and container apps. These projects are developed and shared with the community due to the friendly and altruistic nature of the stellar and incredibly smart people who make up the vast Ansible ecosystem. Sharing Ansible roles on Galaxy, however, is not just reserved for a select few Ansible veterans. Anyone can sign up for an account on `https://galaxy.ansible.com` and share a project already hosted in a GitHub repository. In my experience working with the Ansible community, if you develop a super-cool role or app that solves a problem, it is almost guaranteed there are others out there struggling with the same problem. By contributing your code to Ansible Galaxy, you are not only helping others, you are also opening the door to allow others to help you as well. Oftentimes, others will contribute to your code directly or by leaving feedback for ideas and suggestions for how things can be improved or fixed. Newer and better versions of your code can be contributed from community members and reused to help make your life even easier. That's one of the most powerful things I love so much about open source software: we can achieve as a community what is oftentimes very difficult to achieve on our own.

At the very least, you should check your code into a GitHub repository to share your projects and make your code accessible to other users in the future. GitHub repositories are also useful for using version control on your projects and tracking changes. To use GitHub, sign up for an account at `https://github.com`:

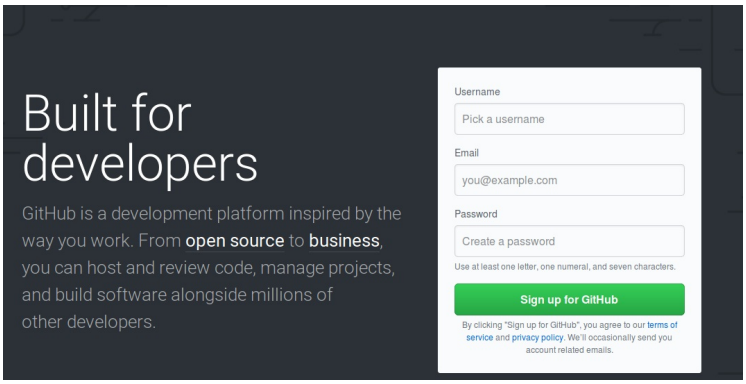


Figure 1: github.com homepage From the `https://github.com/` homepage, you can sign up for a free account by supplying a username, password, and valid email address. Once your account has been created and you have signed in for the first time, you can create a GitHub repository by clicking on the + (plus sign) dropdown in the upper-right corner of the screen and selecting New Repository. This will direct you to a form from which you can provide the details about the new project you wish to create:

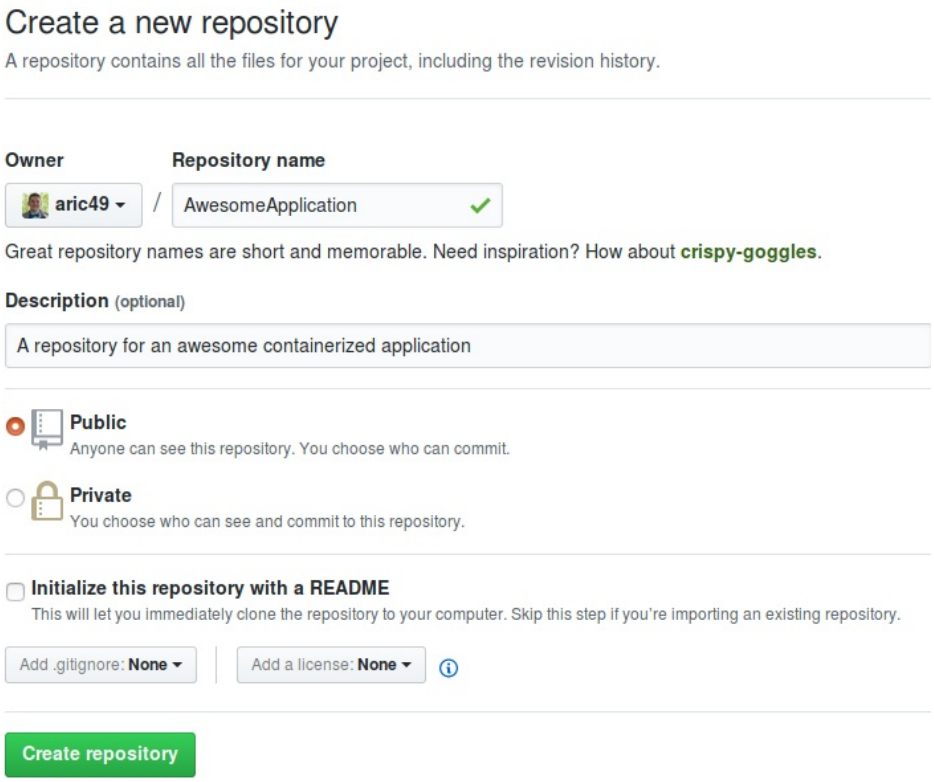


Figure 2: Creating a new GitHub Repository Once the new GitHub repository has been created, you can clone the repo using the SSH or HTTPS link and start contributing code. Each GitHub repository has a unique public URL that can be used to share the link to the GitHub repository. This link is required to share your code on Ansible Galaxy:

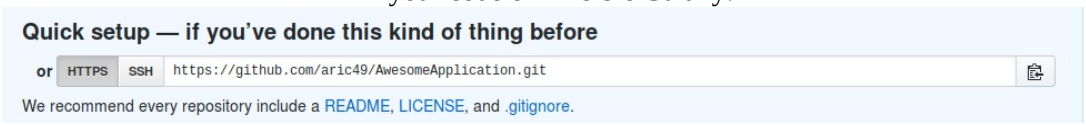


Figure 3: Cloning an empty Git repository The following example demonstrates a sample Git workflow: cloning the repository, creating initial files, and committing them into the repository.

```
aric@local:$ git clone https://github.com/aric49/AwesomeApplication.git
                Cloning into 'AwesomeApplication'...
        warning: You appear to have cloned an empty repository.

                        aric@local:$ git status
  fatal: Not a git repository (or any of the parent directories): .git
                        aric@local:$ cd AwesomeApplication/
                aric@local:AwesomeApplication$ touch file.txt
                  aric@local:AwesomeApplication$ git status
                              On branch master

                               Initial commit

                              Untracked files:
      (use "git add <file>..." to include in what will be committed)

                                      file.txt

nothing added to commit but untracked files present (use "git add" to track)

              aric@local:AwesomeApplication$ git add file.txt

 aric@local:AwesomeApplication$ git commit -m "Adding code to my git repo"
        [master (root-commit) 8bfd103] Adding code to my git repo
             1 file changed, 0 insertions(+), 0 deletions(-)
                       create mode 100644 file.txt
```

For more information about Ansible Galaxy and sharing roles or apps online, please read the Ansible Galaxy documentation located at https://galaxy.ansible.com/intro.

Containerize everything!

Progressing on your journey to a fully modular containerized infrastructure is an exciting one. I'm sure as you have worked your way through the examples contained within this book, you have undoubtedly caught container fever. Hopefully, you have seen how powerful containers can be, especially when run within container orchestration solutions such as Kubernetes and OpenShift. My last word of advice to you before our journey concludes is to keep it up! Now that you have a full understanding of the entire Ansible Container workflow, keep building projects and containerize as much as you possibly can. If the last few years have proven anything in the world of DevOps and infrastructure, it is that containers are the future of software as we know it. As more and more businesses and organizations adopt containerized solutions, the demand for qualified developers who have a firm understanding of building and deploying containers continues to grow.

It is my personal rule of thumb that I try to containerize everything, as often and as much as possible. Having reusable Docker containers for the applications I am working on gives me the ability to quickly build, destroy, deploy, and redeploy entire projects completely on my laptop at a moments notice. Using Ansible Container allows me, as a DevOps engineer, to build containers that speak the same language that my infrastructure speaks: Ansible. Using Ansible, I no longer have to mentally shift gears when working on infrastructure automation and building containerized projects. I can just as quickly develop containers as I can any other Ansible project I'm working on. This has given me tremendous drive and motivation to adapt my previously written Ansible roles to build and deploy Ansible containers that can run in any containerized environment, regardless of the platform or operating system. Subsequently, this also makes me think about current projects I am working on and how I can best adapt my automation strategy to fit into a containerized context.

Looking at projects out there, such as CoreOS's Container Linux (`https://coreos.com/why/`) and other fully containerized operating system platforms, it is apparent that everything, even entire operating systems, will eventually be containerized. Go ahead. Start now! By containerizing everything you are working on, you will

make your work more efficient, repeatable, and locally testable. Not only that, but you are ensuring your platforms, applications, and infrastructure is future-proof. Even if your team isn't currently thinking about containers yet, you should be. The world of cloud infrastructure and containerization is moving so quickly that without embracing a truly modular software development methodology, you will surely be left in the dust.

References

- **Travis CI**: https://travis-ci.org/
- **Jenkins**: https://jenkins.io/
- **Ansible Container Freenode IRC**: http://docs.ansible.com/ansible-container/
- **Container Linux**: https://coreos.com/why/

Summary

When I first started working with Docker containers around two years ago, I thought at first that building and managing containers was quite a pain, given how quickly and efficiently I could write Ansible playbooks that would do exactly the same thing. Eventually, I got the hang of the Dockerfile syntax and started to finally see how much power there was in deploying applications inside containers. What finally convinced me how awesome containers are was when I built out an entire OpenStack cloud using the Kolla project (https://docs.openstack.org/kolla/latest/). The Kolla project aims to deploy a full OpenStack cloud solution using Ansible Playbooks to deploy OpenStack services in Docker containers. Using Kolla, I could deploy an entire multi-node OpenStack cluster in approximately 30 minutes. Coming from a background of having previously automated various OpenStack components, using Chef and Ansible, I was completely amazed.

Around a year ago, I started following the Ansible Container project as a supplement to the Docker, OpenStack, and Kubernetes work I was engaged in at the time. At the time, I saw Ansible Container as the missing piece of the puzzle that would allow me to use Ansible as a full end-to-end development and deployment solution to my container work. It has been an absolutely amazing journey so far. With the support of the community, I have used Ansible Container both personally and professionally to automate and deploy a multitude of projects over the last year or so. I have been absolutely amazed with how much more flexibility Ansible Container gives me as a DevOps engineer compared to building standard Dockerfiles.

It is my hope that you will finish reading this book with a sense of enthusiasm to move forward with your work and career no matter what segment or industry you work in. The thing that never ceases to amaze me with open source software is the sheer number of diverse people in various industries that adopt these technologies. I hope that, as you progress with Ansible Container, you will keep an eye on the future and an ear to the ground. As more and more people adopt

these technologies, amazingly smart people will keep contributing features to make these platforms even better. Thank you for taking the time to read this book. I wish to also thank the amazing people who write Ansible Container: Chris Houseknecht (`@chouseknecht`) and Joshua Ginsberg (`@j00bar`), who have supported me on many of these examples, fixing bugs, and providing fantastic support for a project they are quite obviously passionate about. If this book has been helpful to you, please drop me a line on Twitter or Freenode IRC: `@aric49`.