# Docker Demystified

Learn How to Develop and Deploy Applications Using Docker

SAIBAL GHOSH

bpb

# Docker Demystified

Learn How to Develop and Deploy Applications Using Docker

SAIBAL GHOSH

bpb

# Docker Demystified

---

*Learn How to Develop and Deploy Applications Using Docker*

---

**Saibal Ghosh**

www.bpbonline.com

**FIRST EDITION 2021**



**ISBN: 978-93-89845-87-7**



**LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY**

The information contained in this book is true to correct and the best of author's & publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

**Distributors:**

**BPB PUBLICATIONS**

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

**MICRO MEDIA**

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

Ph: 22078296/22078297

**DECCAN AGENCIES**

4-3-329, Bank Street,

Hyderabad-500195

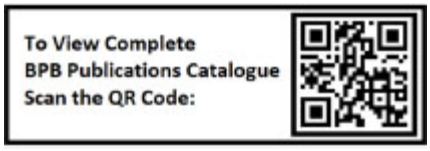
Ph: 24756967/24756400

**BPB BOOK CENTRE**

376 Old Lajpat Rai Market,

Delhi-110006

Ph: 23861747




Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

www.bpbonline.com

# Dedicated to

*All those who kept faith in me when they had no reason to*

# About the Author

**Saibal Ghosh** works as a Principal Architect in Ericsson India Ltd. He has donned many hats in his career, including being a database administrator, a technical consultant, a technical writer, an application developer, and a trainer.

He has a set of specific skills: understanding technology deeply and a keen sense of business pragmatism, and he thrives on matching these optimally. He has over the years, picked up different technologies, and uses them liberally to solve real-world problems in his profession. Of late, he is enamored by DevOps and spends a lot of his time working with Docker and Kubernetes and believes that DevOps is the future of Application Development. He obsesses about technical communication and tries very hard to ensure that such communication is correct, accurate, and to the point. He brings to the table more than twenty years of technical and business experience in and around infrastructure and security, presales, consulting, and solutioning.

Outside of work, Saibal loves reading, meditating, doing yoga, and spending time with his family.

# About the Reviewers

**Indrajit Nandi** is an IT Architect with over 17 years of experience in the mobile telecommunication industry. He is a TOGAF certified platform architect with documented success in creating IT infrastructures in the private and public cloud environments. He is experienced in areas like System Integration, Solution Architecting, Consulting in Large Scale Transformation, Integration and Implementation Engagements with leading Global organizations. He has expertise in IT Infrastructure, BSS implementation, Business Continuity Setup, Vendor Management etc.

He possess significant exposure in Telecom Billing and Mediation Product Implementation and System Integration. He is proficient in platform technologies like Chassis management, Blade Server, Storage, Linux, Virtualization, OpenStack, Public Cloud, Container and Container Orchestration. He is deft in providing technical consulting in multiple operating platforms and various telecommunication operators including Tier-I, Tier-II and Green-Field Operators.

His current role is majorly involved in Infrastructure requirement analysis, effort estimation, proposal making, client handling, high and low level solution design preparation, solution implementation in the Telecom Domain.

He has completed his Bachelors of Engineering (Electronics & Telecommunications), from Jadavpur University, India.

**Debasis Maity** has 15 years of experience in various DB technologies like Oracle, PostgreSQL, Cassandra, MongoDB, and MySQL. He also has worked in the containerization of various DB components in Docker and deployed in Kubernetes. He has profound knowledge in Devops using tools like Git, Jenkins, GitLab, and Ansible. Currently, he is working in Ericsson, USA as a database architect.

*Linkedin profile:* https://www.linkedin.com/in/debasis-maity-19007115/

**Sabyasachi Banerjee** has a penchant for new technologies with a focus on data driven architecture and automation. He has been in the industry for twenty plus years. Currently he is involved in Identity Management and Engineered System design and implementation projects. He is an early adopter of most things new. He is a B.Tech in Metallurgical Engineering from NIT, Warangal, who has dabbled in making steel to software development. He travels for work and enjoys hiking, traveling, and photography. Presently working as a Senior Consultant in New Zealand.

# Acknowledgements

This book would not have been possible without the blessings of my late parents, and the continued support of my wife Mouli and son Arpan, both of whom had to often put up with my brusque and curt manner, as I worked on the book late in the evenings and on weekends, cutting out on family time.

I would like to thank the technical reviewers of the book: Indrajit Nandi, Debasis Maity and Sabyasachi Banerjee, all of whom took time out from their busy schedules to technically review the book and give me the reassurance that the book indeed fulfils the objectives for which it was written. My gratitude also goes out to a lot of other people who encouraged me during the writing of this book, none more so than my brother Prabal, who helped me with all the diagrams in the book, as well as my friends and colleagues who kept me on my toes by often asking me about the progress of the book, thereby implicitly nudging me along.

This genesis of this book lies in my struggle in trying to decipher and understand Docker while I was learning it and trying to find the answers to questions and doubts that came to my mind while I tried to wrap my head around all the new concepts and terminologies, often facing a sea of doubts with no shoreline in sight.

If this book helps in ‘demystifying’ Docker for someone who is new to the technology, I would consider my job well done and my efforts more than adequately compensated.

Finally, I would like to thank BPB Publications for helping me throughout the book.

# Preface

Till a few years ago, all applications were monolithic in nature. That is, they were built as a single unit and were considered a composite whole. Any changes to be incorporated into the application meant bringing down the entire application stack, making changes to it, testing it out, and then bringing it up. The whole process was very cumbersome, unwieldy, and often error prone. Many times applications that ran perfectly on the test environment ran suboptimally on the production system or, worse, sometimes even didn't run at all!

And it frequently took days of painstaking analysis to find out that say, some of the libraries that were present in the development environment were missing in the production environment, resulting in the mismatch and all the subsequent problems.

Enter containers.

The way containers work is that they encapsulate an application into a single executable package along with all its related configuration files, libraries, and dependencies required for it to run. Containerized applications share the operating system of the host and use the run time engine of Docker to coordinate all the activities that let more than one container residing on the same host operating system share the host operating system's resources. This, of course, eliminates the requirement of having numerous operating systems and makes the containers fast, nimble, and agile.

These days most large-scale applications are microservice based. Instead of using a single monolithic codebase, applications are broken down into a collection of smaller components called microservices. The benefits of doing something like this is enormous. Each component can be individually developed, tested, deployed, and scaled. There is no dependency on other applications and enables the use of different programming languages, databases, and other tools for each microservice.

It is no idle coincidence that microservice based architecture has become very popular along with the growth of containers because it is containers that allow us the leeway for creating self-contained applications (microservice-based applications), which are loosely coupled and can be independently deployed and each separate application can work independently as well as be part of a composite whole, that is the microservices can be linked together to form a larger application.

Docker is an excellent tool for microservices. Each application can be deployed in separate containers, or they may even be broken down into separate processes running in separate Docker containers.

The main aim of this book is to provide you the necessary information and understanding to develop the skills to use Docker for your application development, deployment, and management. This book has a very hands-on approach, and there is a screenshot provided for almost every example done. Every concept is explained using a proof by example methodology so that the reader not only reads about the concept but also immediately sees a demonstration of the concept. In this book, you will learn about the following:

**Chapter 1** is an introductory chapter. This chapter introduces us to the world of containers, and talks about its benefits, and walks us through its difference with virtual machines, and then goes on to discuss the evolution of Docker and how it has become the tool of choice for application development. This chapter also discusses the Docker Registry, Microservices, and built-in security in Docker.

**Chapter 2** goes deeper into containers and its fundamental building block, images. In this chapter, we dissect containers and images and learn about them in detail. In this chapter, we will also learn to run containers, inspect them, check their logs, and then have a look at the container architecture. We also talk in detail about images, dockerfile, and how they all bind together.

**Chapter 3** Having understood containers and images, in this chapter, we look into storage drivers and volumes, which form a part of the building block of the docker ecosystem. This chapter goes

deep into storage drivers and volumes and the intricacies of their usage.

**Chapter 4** is a key chapter. In this chapter, we look into the theory behind Docker networking, which is thoroughly application driven. We also look at the structure of Docker networks and how it provides all the bells and whistles for the network to work smoothly, but at the same time keeping just the right level of abstraction for ease of use for application developers.

**Chapter 5** takes us into the world of Docker Swarm. This chapter explains the concept of orchestration and shows us how to harness the cumulative capabilities of numerous containers working together either on-premise or in the cloud.

**Chapter 6** goes deep into Docker networking, looking into the implementation of various networks in the Docker landscape. This chapter discusses the intricacies of the networks and how Linux networking features are leveraged to create a robust and secure networking infrastructure in Docker.

**Chapter 7** takes a look at the security features available in Linux and how they are utilized in Docker. This chapter focuses on the security capabilities of Linux and how they may be leveraged to create a safe and secure Docker environment.

**Chapter 8** details are securing our containerized applications combining the security features of Linux and the Docker Enterprise Edition. This chapter goes deep into the various components of the Docker Enterprise Edition and how we can best use these features along with the security features provided by Linux to secure our Docker containers.

# Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors if any, occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

**errata@bpbonline.com**

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

## BPB is searching for authors like you

If you're interested in becoming an author for BPB, please visit **www.bpbonline.com** and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Docker-Demystified**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at **https://github.com/bpbpublications**. Check them out!

## PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

## If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**.

## REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased

opinion to make purchase decisions, we at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

# Table of Contents

# CHAPTER 1

# Introduction to Containerization and Docker

## Introduction

Containerization is now in vogue in the software development world as an alternative or even as a complement to virtualization. Basically, containerization allows us to encapsulate (containerize) software with all its dependencies and the run-time environment in such a way that it runs uniformly across different infrastructures and platforms. Running software in containerized environments generally use fewer computing resources than running software within different virtual machines, since the latter requires a separate copy of the operating system to run on each virtual machine. In the coming days, we can hardly envisage any software being developed, which does not leverage the functionalities of containerization in some form.

## Structure

- Life before Containerization
- Concept of Containerization
- Benefits of Containerization
- Docker
- Docker engine
- Docker Hub and Docker Registry
- Linux and Windows Container
- Microservices and Containerization
- Security in Docker

## Objective

After studying this chapter, you should be able to understand what containers are, its usage and benefits, how Docker came into the picture, and how large applications are developed using a microservice-based architecture and Docker.

## Life before Containerization

Way back in the early 2000s, applications ran on servers, and we had to provision enough computing capacity to ensure that all our applications ran without any problems on these servers. However, it was not a static situation, and from time to time, businesses needed to grow their applications or add new applications, in which case they needed to increase the capacity of their servers, or worse go out and get new servers. So, at any given point of time, the IT team had to make an educated guess on how much capacity they would require in the coming months and provision accordingly, and this was a never-ending process. To ensure that the servers were big and powerful enough to take care of all the currently running applications as well as be prepared to meet the future growth, the IT operations team generally erred on the side of caution and often bought servers which were much bigger than the current requirement, resulting in a huge sub-optimal usage of those servers. It was not uncommon to see servers use as low as 10-15% of their full capacity. It was an enormous wastage of computing power, and by implication, company resources.

So, what next? Around 2005, we started hearing about something called virtualization.

Suddenly we had a situation where the next time we needed to add new applications or grow the current applications; we were not required to go out and buy new servers, instead, we could now leverage the functionalities afforded by the virtualization technology to use the spare capacity available on the servers to run numerous business applications and save money by optimizing the use of the servers. However, virtualization or running VMs on the servers came with its own set of downsides; every VM required its dedicated

**operating system** (**OS**). The OS, in turn, consumed computing resources, which kind of ate into scarce resources; each OS needed its maintenance, patching, and so on. The OS often required a separate license, and worst of all, for all VM based applications, there was a performance penalty, since the VM based applications were almost always slower than the applications running on bare metal. While virtualization was a step in the right direction, it was certainly not *manna from heaven!*

## Concept of Containerization

The foundation of containerization lies in the **Linux Container** (**LXC**) Format. LXC is an operating-system-level virtualization method for running multiple isolated Linux systems (container) on a control host using a single Linux kernel.

The fundamental difference between VMs and containers is that a container does not require its own OS, thus inherently making it much more lightweight than a VM. In fact, all containers on a server or a host share a single OS, thus freeing up a huge amount of computing resources such as storage, CPU, and RAM. A VM is slow to boot and takes a relatively long time to start up. Containers, on the other hand, are fast to startup and extremely lightweight. In fact, containers can have sub-second launch times!

So, with containers, we are not only guaranteed speed and agility but also, we save on potential license costs of the operating systems, as well as bypass the hassle of maintaining several operating systems. And this, to put it very simply, optimizes return on investments and translates as net savings for the company.

The container is lightweight because it shares the machine's operating system kernel and does not require the overhead of associating an operating system with each application. Naturally, containers are much smaller in size than a VM, thereby allowing us to have many more containers in the same host as compared to the number of VMs that we could potentially have.

This is just one part of it. The other part relates to software development. We have all had experiences where we developed a

piece of software, tested it out on our test systems, and everything ran just fine, still, when it was deployed in the production environment, something did not work out, and either the application ran slowly or sub-optimally, or worse, failed outright. It just did not run at all.

After numerous hours of doing a **root cause analysis**, we finally discovered that there was a library missing on the production environment, or the production environment had a different level of patching as compared to the test system, or something else did not match between the test and production environments leading to the fiasco.

With containerization, we have found a way out of this problem. Containerization simply eliminates this problem by bundling (containerizing) the software code together with the related configuration files, libraries, and other run-time dependencies required for it to run. The package of software code or container is separate from the operating system and other dependencies and can easily be ported to other environments and will run uniformly and consistently across platforms. We write the code once, containerize it, and run it a *thousand times* anywhere. This portability is extremely important as it allows us to develop code in any computing environment and then transfer it just about anywhere, secure in the knowledge that when we run it, we will get the same consistency, uniformity, and performance that we did in our original computing environment. This is indeed great news!

However, it is not as if the concept of containerization was invented overnight, and it became a sensation. The concept of containerization and process isolation has been around for some time now. But the emergence of the open source **Docker Engine** in 2013 -a lightweight, powerful open source containerization technology combined with the workflow for building and containerizing of applications really ushered in the era of containerization. The research firm Gartner projects that more than 50% of companies will use container technology by 2020. And frankly speaking, that’s quite an overwhelming number.

## Benefits of Containerization

So, lets now take a step back and see how containerization benefits us. As discussed earlier, compared to **virtual machines** (**VMs**), containers have a significantly smaller resource consumption footprint than a VM. Since the containers share the same OS kernel, we can pack in many more containers in the same host or server as virtual machines. Thus, it's really a no brainer that containers any day will be a more prudent choice for application development compared to VMs. Let's have a look at the following graphic. The graphic makes a minimalistic resource usage comparison between a virtual machine and a container.

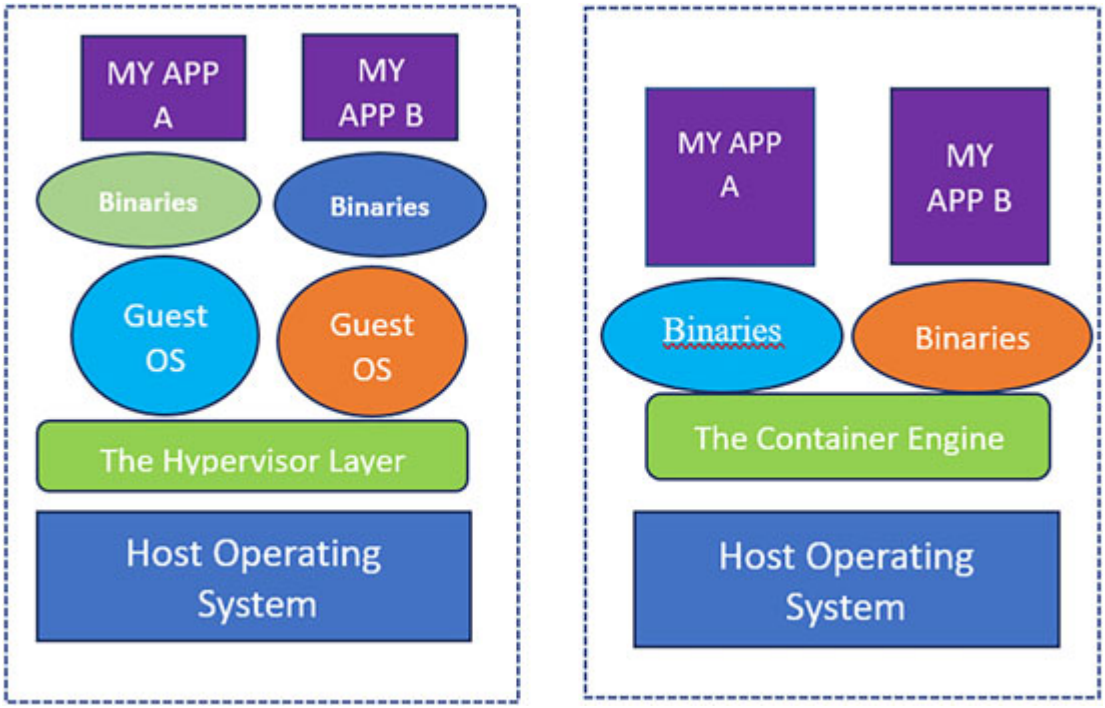



***Figure 1.1***

As we can see, the VMs have the hypervisor layer as well as individual operating systems for each application, thereby having more layers between the application and the host operating system. On the other hand, it is easy to see that there are fewer demands on the server in the case of containers. This is a key point. It simply means that we can fit in many more containers in the same server than VMs and continue to enjoy superior performance! As we can

see, the containers do not have their individual operating systems; it simply uses the host operating system kernel to get their work done.

The second benefit that the container provides us is because they are so lightweight, given the fact that they do not have a dedicated operating system for each container, they are much more nimble and agile than a traditional VM as far as starting and stopping go. As mentioned earlier, we can launch containers in sub-second times, and that is quite understandable-simply since there is no operating system to boot, our app starts loading immediately.

Now coming to security in containers-containers have a concept of namespaces. We are going to discuss this in detail later, but for the moment, let us understand that typically in a container, our applications are *sandboxed* and will not communicate with each other unless we configure them to. This provides a built-in security mechanism and helps to keep our applications safe. Additionally, if, by any chance, a malicious piece of software somehow finds its way into a container, it will not be propagated to other containers.

Another key advantage of a container over a VM is the developers can use the exact same environment for both development as well as production. We had briefly discussed this earlier as well. We all know that a common stumbling block in the development world is that an app developed on the developer's laptop doesn't run on the production server for the excellent reason that the environment on the developer's laptop and the production server do not match. With containers, we have just solved this issue in one step.

Since the run time environment is literally packaged along with the container, we have eliminated the variable of having a different run time environment in production, as compared to a test or a lab environment, which could impact how the application runs, provided it does so in the first place! Additionally, we can use containers for **continuous integration/continuous delivery** (**CI/CD**) pipeline integrations, helping developers become more productive and efficient.

Another advantage of containers is that it typically runs as a single service. For example, we can have a service for our database, a service for our web applications, a service for say, an application

running analytics, and so on and so forth. We have already seen that there is a benefit of keeping our services isolated, but at the same time, we would like to have another dimension to the story, so to say. We ought to be able to scale up/down our services as and when required. Fortunately for us, containers can both scale up and scale down, and we also have a mechanism in place to orchestrate and harmonize, such as scaling up or scaling down. Docker has a native clustering and orchestrating solution known as Docker Swarm. While there are other popular clustering and orchestration tools in the market, like *Mesos* and *Kubernetes*, *Docker Swarm* is a pretty nifty tool in itself.

## Docker

Ok, let's start with the basic question: What is Docker?

Docker is a product of *Docker Inc*., a San Francisco based organization started by Solomon Hykes. Initially, it was started as dotCloud to leverage the containerization movement.

The most common way to understand Docker is to consider it as a piece of software that creates, manages, administrates, and orchestrates container. It runs both on Linux as well as Windows. The software is developed as part of the open-source Moby project on *GitHub*. *Moby* is an open-source project created by Docker to advance the software containerization movement. It is a project for all container enthusiasts to experiment and exchange ideas. As Solomon Dykes, the CEO of *Docker Inc,* said, "*Docker uses the Moby Project as an open R&D lab.*"

To go a little deeper, Docker isn't a single monolithic application. Instead, it is made up of components like `containerd`, `runc`, `InfraKit`, and so on. The community works on the individual pieces as well as the composite whole, that is, Docker, and when it's time for a release, *Docker Inc*., the company, will package them as one homogeneous unit and release it.

Let' see how Solomon Hykes, the former CEO of *Docker Inc,* explains it:

*"*We needed our teams to collaborate not only on components but also on assemblies of components, borrowing an idea from the car industry where assemblies of components are reused to build completely different cars."

Hykes went on to explain that Docker releases would be built using Moby and its components. At the moment, there are more than eighty components that are combined into assemblies.

## The Docker Engine

The Docker Engine works on the principle of a client-server application. It can either be downloaded from *Docker Hub,* or we can build it manually from the source in GitHub. The Docker Engine can be deconstructed to have three main components:

1. The server, which is really the daemon process running.
2. The REST API, which takes stock of the programs that interface with the daemon process and instructs it about what is to be done.
3. And finally, the plain vanilla command-line interface-the the docker command.

## Docker Engine Components Flow

The Docker REST API provides a **command-line interface** (**CLI**) for interfacing with the Docker daemon. This may be done using the command line interface or creating scripts and running them. The **Application Programming Interfaces** (**APIs**) provided, are used by several applications, and the Docker daemon is tasked with the heavy lifting of creating, administering, and managing the Docker objects, such as containers, images, networks, storage, volumes, and so on.

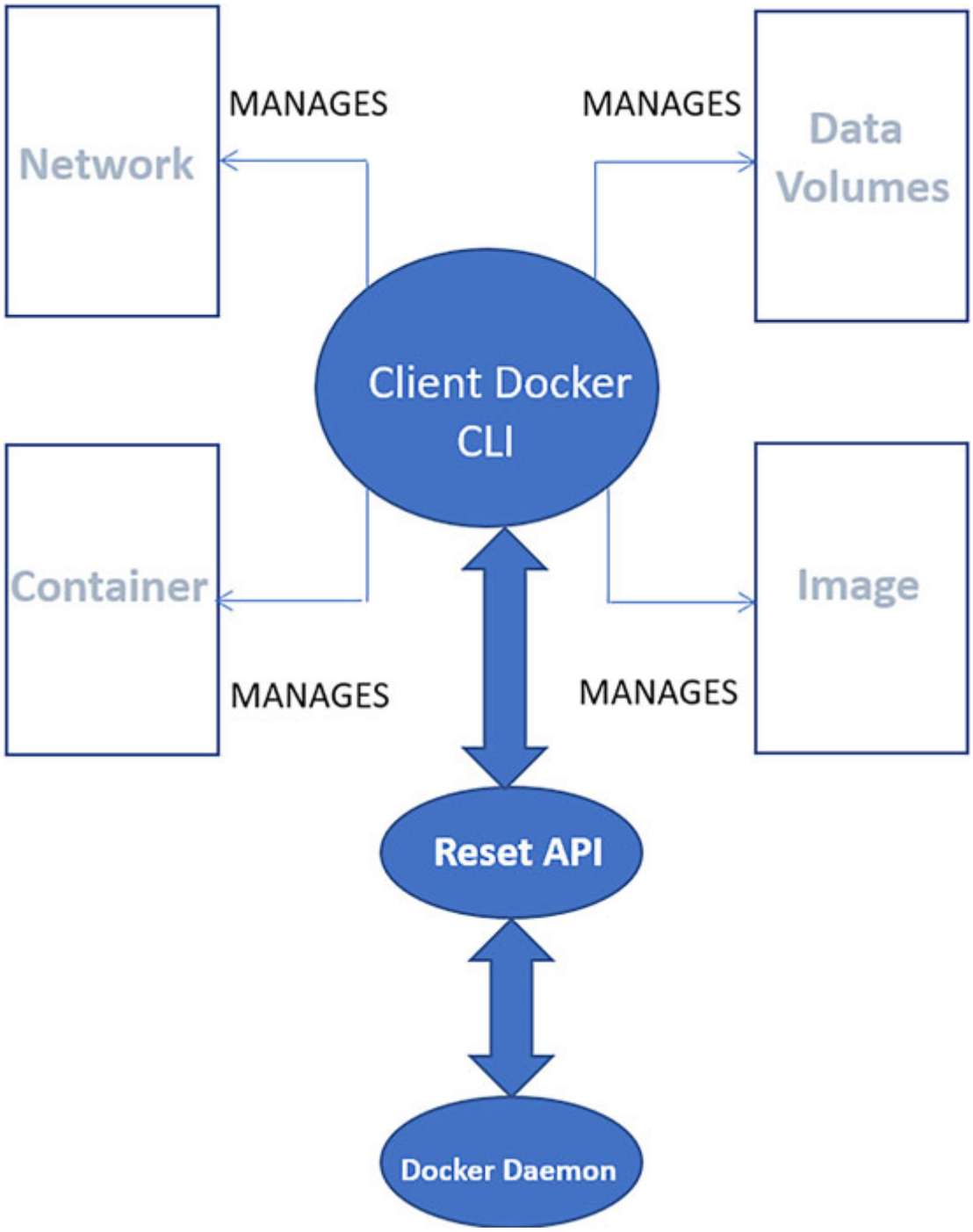



***Figure 1.2***

**Docker is licensed under the open source Apache 2.0 license.**

Docker uses a straight-forward client-server architecture, enabling the Docker client to talk to the Docker daemon, which does the hard work of building, running, maintaining, administering, and distributing

our container. The Docker client and daemon can run on the same system, or they may be part of a distributed architecture. And as pointed out above, the Docker client and daemon communicate using a REST API, over UNIX sockets, or a network interface. *Figure 1.2* instead of 'the following diagram' shows a very simple representation of how it is all put together and how it all works.

The Docker Engines comes in three different flavors:

**Docker Engine – Community Edition**: This is an ideal platform for individual developers or small teams who start out trying to get their hands dirty with Docker and try it for developing container-based applications. This has limited functionality as compared to the other two editions but is good enough for getting started with Docker.

**Docker Engine – Enterprise Edition**: This is a platform that is designed for enterprises that develop applications leveraging the functionalities provided by containerization. This has several built-in security safeguards and is capable of meeting enterprise-grade **service level agreements** (**SLAs**).

**A Service Level Agreement is an agreement between the vendor and customer regarding the level of service expected by the customer from the vendor based on measurable metrics. If the service level agreements are breached, there is usually a provision for penalties.**

**Docker Enterprise Edition** is designed for mission-critical applications. It is for building, shipping, and running business-critical applications at scale.

More details about the editions and the difference in the Docker documentation: https://docs.docker.com/install/overview/

## Docker Hub and Docker Registry

A registry may be defined as storage and content delivery system. In this case, they hold Docker images which are available in different tagged versions. So, Docker Registry is a service that is holding our Docker images. Docker Registry could also be hosted by a third party, as a public or private registry. Examples are Docker Hub, AWS Container Registry, Google Container Registry, and so on.

The same image may have several tagged versions, and we can push and pull images from the Registry. Typically, users interact with the registry using Docker push and pull commands.

The default storage driver is the local filesystem, which is perfectly fine for development and small deployments. However, if we require more storage, then we are in luck because Docker provides native support for several cloud-based storage drivers like *Microsoft Azure*, *S3*, *OpenStack*, and so on.

Docker Hub is a cloud-based repository in which Docker users and partners create, test, store, and distribute container images. Docker Hub is a cloud-hosted version of the Docker Registry.

Of course, we can run our Registry too. Our Registry can be classified as a Private Registry if it requires authentication to pull images. If we let anyone pull an image from our repository without any form of authentication, then it assumes the character of a Public Registry.

Our Registry could be a great way to integrate and complement our CI/CD system. Typically, a commit to the source revision control system would trigger a build in the CI system, and if the build is successful, then this would result in a push of a new image to our Registry. The notification from the Registry will automatically start a deployment on a staging environment, or even notify other systems that a new image is available.

## Linux and Windows Container

Docker runs both on Windows and Linux. Initially, Docker leveraged the Linux functionalities like namespaces, control groups, union filesystems, and so on. And the modern Container started in the Linux world. In most cases today, Docker runs on any Linux system with a Linux kernel of 3.10 or later. That kernel version is as of 2011, and so most Linux distributions released since then work with Docker. And the fact that a Container running on one Linux distro can be moved to another Linux distro with minimal effort is extremely convenient for developers.

## What about Container on Windows?

Docker and Microsoft have a joint engineering relationship to deliver a consistent Docker experience for developers and operators. Now, all *Windows Server 2016* and later versions come with *Docker Engine - Enterprise*. On top of this, developers can now leverage Docker natively, if they are running *Windows 10*, using *Docker Desktop*. There is no real difference between Docker on Linux and Docker on Windows. They work in the same way: same CLI, same APIs, same image format, and content distribution services.

The only thing that we see working against Windows is the GUI aspect. Docker was designed for containerizing applications that have a command-line interface. On both Windows and Linux, it lacks a native way of connecting to a graphical interface inside an application.

Using VNC or RDP to connect to GUIs require additional effort and maybe not also be risk-free from a security perspective.

This problem is indigenous to both Linux and Windows. However, given the fact that GUI is more common with Windows than with Linux, this limitation is more of a bottleneck on Windows. The Linux administrator will hardly use a GUI while the Windows sysadmin is much more accustomed to administering their systems with a GUI.

## Microservices and Containerization

Microservice is based on the modular approach to software development. To that extent, it is nothing new. It had earlier been called code modules or sub-routines or even software components - that is not important. What is important is that now with the microservice-based approach to software development, each piece of software has a well-bounded scope and can now be individually deployed, scaled, and upgraded. Combine it with the functionalities afforded by containerization, or more precisely, Docker, and you have an unbeatable combination, so to say.

Let’s go a little deeper. In the monolithic style of application development, the application is built as a single unit, typically consisting of a client interface, a database, and a server-side

application that will handle the business logic. Now assume there is a change in the system somewhere; the entire server-side application must be rebuilt and redeployed. And that's a massive task. Almost everything comes to a grinding halt as the entire application goes through the metamorphosis of having to be rebuilt, re-tested and redeployed, in a new avatar, so to say.

This is where the microservice-based architecture shows its worth. We don't have to change the entire application if we are changing a feature or adding new business logic, or changing something in the system, we change just the individual requirement, and that's it-we are good to go. This is based on the fact that microservices are modular. Each service runs on its own, separate from the others within the architecture. There is no dependency, and since there is no dependency, we can change, modify, enhance, delete from a particular module of the application without in any way impacting the rest of the application.

Thus, to put it slightly differently, we can make changes in a module of a microservice-based application, without affecting the rest of the application, as long as the underlying infrastructure supports it by providing the necessary computing resources.

Okay, now let's come to containers. As mentioned earlier, containers and microservices are a powerful combination. Since microservices are modular, each service can run on its own, independent of the rest of the application. Containers provide individual microservices their work environments, isolated from the rest of the application, making them independently deployable, modifiable, upgradable, and scalable. Microservices can be written in any language that doesn't really matter. What matters is that they can be readily deployed and scaled in containers.

So today, the trend is very quickly changing from the monolithic architecture to a microservice-based architecture, and large-scale applications are being deployed as microservice-based applications.

## Security in Docker

Since we have an entire chapter dedicated to security on Docker, we will just have a cursory look about it here. While reviewing Docker security, there are four major areas to consider:

1. The security at the kernel level and its support for namespaces and control groups (`cgroups`).
2. The surface area provided by the Docker daemon.
3. Security vulnerabilities in the Container configuration profile.
4. Hardening of the existing security features, and additionally, using third-party tools to do so.

Docker containers are very similar to LXC containers, and they have similar security features. We may be aware that LXC is an operating-system-level virtualization method for running multiple isolated Linux systems (container) on a control host using a single Linux kernel. When we start a container with docker run, behind the scenes, Docker creates a set of namespaces and control groups for the container. Now, what are namespaces and control groups?

Namespaces insulate our processes running inside a container from being affected in any way from processes running in any other container, or even on the host system.

Control groups, on the other hand, provide a mechanism to implement resource accounting and limiting. They provide a set of useful metrics that are used to ensure that each container gets its fair share of computing resources.

Additionally, each container has a full-fledged network setup, which means that a container has a self-contained network infrastructure in place and does not need access to another container's network infrastructure for doing its job. This is the default unless, of course, the host system is set up to have the containers talk to each other. From a network architecture point of view, all containers on a given Docker host are sitting on bridge interfaces. This means that they are just like physical machines connected through a common Ethernet switch.

So, fundamentally the container comes with some built-in security prudence. Thus, malicious programs will not automatically propagate

to other containers, or denial-of-service attacks cannot occur easily, and in general, consistent uptime is more or less guaranteed.

Now, coming to the Docker daemon surface attack area, when we are running containers, effectively, we are running the Docker daemon. And usually, we run the daemon with the root privilege. And it goes without saying that only trusted users should be allowed to control our Docker daemon.

However, even after doing the due diligence on users working with our Docker daemon, one of the potentially dangerous things that are indigenous to Docker, is that it lets us have common directories between the container and the host system. And that too, with unfettered privileges.

This means that the container if it so desires, can alter our host filesystem with absolute ease. This is similar to how virtualization systems allow filesystem resource sharing. Nothing prevents us from sharing our root filesystem (or even our root block device) with a virtual machine.

This has a strong security implication: for example, let us assume we have put in place a mechanism where containers are provisioned from a web server through an API. Now, we have to be extremely vigilant that no malicious users can pass on some virus, which passing through our Docker container infects the host system itself!

Coming to the problem of whether we ought to run docker as root or not, it is very simple to see that running Docker as non-root is very limiting and is rarely done in the real world.

However, there is an interesting point. Docker containers have an infrastructure around the container that takes care of a lot of stuff that would otherwise have needed the full-blown root privilege. This means that in most cases, containers do not need "*real*" root privileges at all. And therefore, containers can run with a reduced capability set; meaning thereby that "root" within a container is not really a full-blown root. It is root with much reduced privileges. And because this is a '*restricted root*' so to say, we are indirectly putting in a security layer that sort of ensures that the risk of attack by malicious users is somewhat stymied.

Let’s move on to the configuration profile. Docker supports the addition and removal of capabilities, allowing the use of a non-default profile. This may make Docker more secure through capability removal, or less secure through the addition of capabilities. The best practice is to practice the principle of least privileges, that is, remove everything except what is explicitly needed by the process.

Additionally, there is the Docker Content Trust Signature Verification. We have the mechanism to enable the Docker Engine to be configured only to run signed images. The Docker Content Trust Signature Verification feature is built directly into the dockerd binary.

Apart from leveraging the many security capabilities inside the Linux kernel, many well-known third-party tools can be used to add security features in Docker.

While Docker currently only enables capabilities, it doesn’t interfere with the other systems, which allow us to harden Docker in many ways, as the third-party tools don’t require Docker-specific configuration, since those security features apply system-wide, independent of container.

We can define our own policies using our favorite access control mechanism. The bottom line here is that there are lots of ways to harden the Docker without us having to customize or modify Docker itself in any way. We will discuss these and other security features in more detail in the chapter on security.

## Conclusion

In this chapter, we had a look at containerization and its inherent benefits. We also saw how Docker evolved and how it is an integral part of software development now. We learned about Docker and the many benefits it brings to the business, based on the fact that Docker can provide a containerization platform that packages our application and all its dependencies together in the form of a Docker container to ensure that our application works seamlessly in any environment. We also had a brief glimpse of how Docker gels with the microservice-based architecture paving the way for large scale

applications to be developed using microservices and Docker. And finally, we saw about security in the Docker environment and learned about Docker Hub and Docker Registry.

So, now with the basics out of the way, let's dive deeper into the architecture of Docker and understand it's building blocks in the next chapter.

## Points to Remember

- Containers encapsulate the run-time environment along with the application, thus providing a consistent environment for the applications to run.
- Containers are much more lightweight than virtual machines, thus making them far easier to spin up, run, and maintain.
- *Docker Inc.*, the company, provided ready-to-use container technology for commercial use.
- Docker runs both on Windows and Linux.
- Microservices and Docker are a great combination.
- Docker has built-in clustering and orchestrating technologies.
- Docker, in general, provides a fundamentally secure setup off the shelf.

## Multiple Choice Questions

**Choose the most appropriate answer:**

1. The Docker Container is lightweight compared to a Virtual Machine because:
    a. The Docker Containers are usually smaller than a Virtual Machine
    b. The Docker Container does not have a hypervisor
    c. The Docker Container does not have a hypervisor and additionally, uses the host's operating system
    d. The Docker Container does have a small hypervisor and additionally, uses the host's operating system

2. The Docker Swarm is:
    a. A Clustering Solution
    b. A Disaster Recovery Solution
    c. A Clustering and Orchestrating solution
    d. A Clustering and Backup solution
3. A Microservice-based architecture:
    a. Is bounded and has a defined scope
    b. Is architected to run only on bare-metal
    c. Is architected to run only on Containers
    d. Is open-ended and does not have a defined scope
4. A Docker Container:
    a. Uses the network stack of the host on which it runs
    b. Uses a common network stack allocated for each Container
    c. Has to have a network stack created manually
    d. Has its own independent network stack
5. A Docker Container:
    a. Can be easily scaled up or down
    b. Cannot be easily scaled up or down
    c. Can be easily compromised from a security perspective
    d. Will run even if the host system goes down

## Answers

1. c
2. c
3. a
4. d
5. a

## Questions

1. What is a Container?
2. How is it different from a Virtual Machine?
3. What is a Docker Registry?
4. What is Docker Swarm?
5. What are Microservices?
6. What is the native security setup in Docker?

## Key Terms

- Container: Docker is a piece of software that allows us to package our code and all the relevant run-time variables and dependencies so that the application runs consistently and uniformly across platforms.
- Docker Swarm: It is Docker's native clustering and orchestrating solution that helps to manage several Docker Containers running on the host(s).
- Microservices: An architecture where each piece of software has a well-bounded scope and can be individually deployed, scaled, and upgraded.
- Docker Registry: A Registry may be defined as storage and content delivery system. In the case of Docker, they hold Docker images which are available in different tagged versions. So, Docker Registry is a service that is holding our Docker images.

# CHAPTER 2

# Containers and Images

In this chapter, we will get deeper into containers and its fundamental building block, images. However, before we get started, we need to have the Docker software running on our system. This book focuses on Docker, on Linux, although almost all of the principles we talk about in this chapter and the rest of the book are true for windows as well.

## Structure

- Conceptualizing Containers
- Running Containers
- States of a Docker Container
- Getting inside a Container
- Inspecting a Container
- The Container Logs
- Basic Container Architecture
- What is a Docker image
- Getting deeper into images
- The Dockerfile

## Objective

After studying this chapter, you should be able to understand in some detail how containers and images work and the best practices to be followed while working with containers and images.

## Conceptualizing Containers

Containers are created from images, which may be considered as templates for creating containers. A container may be considered as a dynamic implementation of an image, which is a static tarball containing a filesystem. We will learn in detail about images later in the chapter, but for the moment, it will suffice for us to know that images are the building blocks of containers. So, where do we get the images?

One way is that we build our own images, which we learn about later in the chapter. but we also have an enormous number of images available in the public registries like Docker Hub, Google Container Registry, Amazon EC2 Container Registry, Azure Container Registry, .etc. When we install Docker, by default we get connected to the Docker Hub Registry.

## Running Containers

Let's get ready to explore the world of containers by running our first container. However, before we do that, let us see if we have docker running on our system. One quick and easy way to do it is to check for the version of the docker running on our system.

```
docker --version
```

or

```
docker -v
```

If Docker isn't installed in the system, we will get an output similar to the following screenshot:

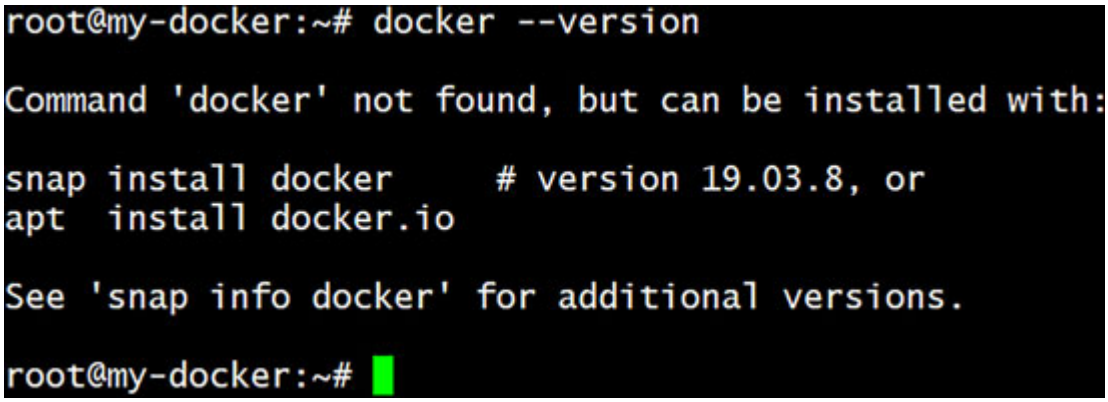


*Figure 2.1*

Else, if it is installed on the system, our output will be something like the following:

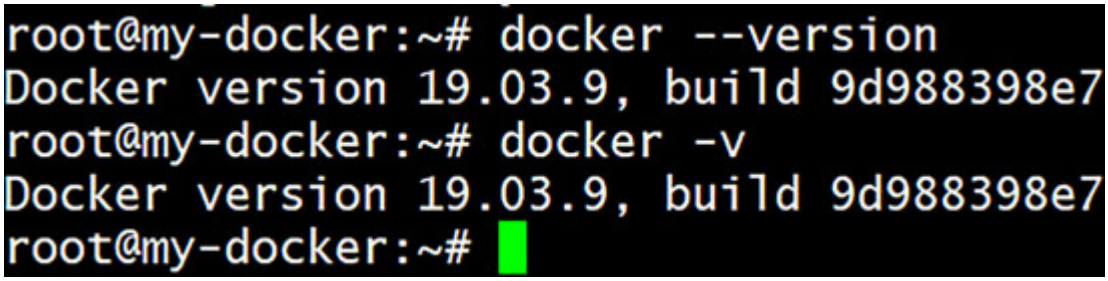



*Figure 2.2*

As we can see, we have `Docker version 19.03.9` running on our system.

Our next step is to run a container. When we say, we run a container, effectively, what we are saying is that we create a container and start it on our system. In the example below, we are creating a `busybox` container and running it on the system. `busybox` is a software suite that provides several Unix utilities in a single executable file. The following is the command to create a busybox container.

```
docker run --name my_first_container busybox:latest
```

Let us see the output in the following screenshot:

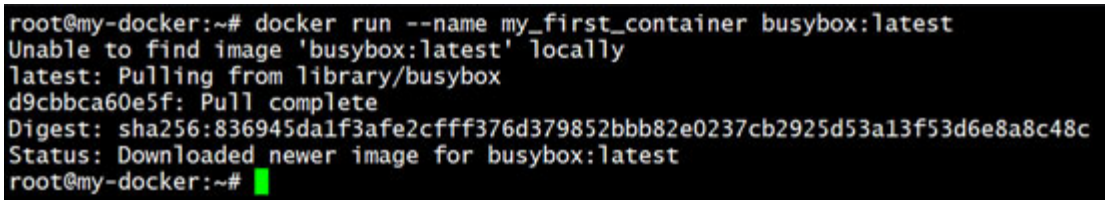



*Figure 2.3*

If we look carefully at the output, it says that it wasn't able to find the image locally. In other words, the image wasn't available on our local machine, and this implies that it was pulled from Docker Hub, which is the default registry to which a standard Docker installation gets connected automatically. Then we see some random characters and the pull completed. This is followed by another set of characters, which is a digest in the form *algorithm: hexadecimal* of the image

layers that go to build this container. We will discuss image layers in detail down the line, but for the moment, let us keep in mind that images are layers of files and directories that provide us a composite filesystem from which we can create containers.

Before we move ahead, let us discuss the command we used to run our first container.

```
docker run --name my_first_container busybox:latest
```

**`docker`**: This refers to the docker command-line interface which we usually use to communicate with the docker engine.

`run`: This refers to running a container. This is the context. A more verbose way of running a container is using docker container run.

**`--name`**: This refers to the name of the container; in this case, it is **`my_first_container`**.

**`busybox:latest`**: This is the image we are interested in. The tag latest signifies that it will only pull from the registry the latest image of BusyBox that is available there.

Now, let us run another container and use some flags to control the container at a more granular level. This time we are pulling an image of software named alpine, which is a Linux distribution built around BusyBox. We name our container alpine as well. See the following command:

```
docker run --detach --interactive --tty --name alpine
alpine:latest
```

And following is the output of the command:


***Figure 2.4***

Now, let us look at the various flags used in *Figure 2.4*.

`--detach` means the container runs in detached mode, that is, the background.

`--interactive` means that the container will accept input and provide output.

`--tty` means our container is in interactive mode, we will usually need to provide a terminal for getting the output.

The above flags can be abbreviated as `-d` for `--detach`, `-i` for `--interactive` and `-t` for `--tty`.

The above flags can be combined in the command as:

```
docker run -dit --name alpine alpine:latest
```

Before we end this section, there is one more thing that we need to pay attention to. We have been talking about running containers. Running a container implies that we create a container and then start it. We can also simply create a container and choose not to start it. We will discuss it below when we discuss the states of a container.

## States of a Docker Container

In this section, we talk about the states in which a docker container may exist. A docker container exists in any one of the following states:

- Running
- Paused
- Exited
- Restarting
- Dead
- Stopped
- Created

A container is in the running state when it is currently running. The status will show as ‘UP’.

Let us create a container in the running state and then check the state. We can use the command `docker ps --all` or the `docker`

`container ls` **-a** command to check the state of the docker container. See the following command:

```
docker run -dit --name my_container busybox:latest
```

The output is shown in the following screenshot:

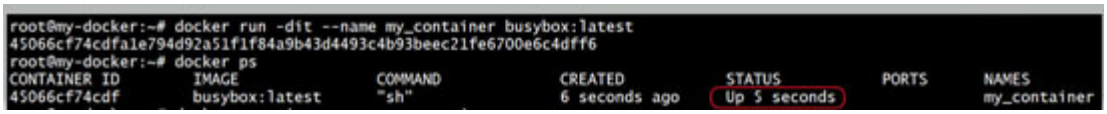



***Figure 2.5***

It is clear that the container is up and running. We can easily pause/unpause a running container, as shown below, by running the following command.

```
docker container pause my_container
```

See the output in the following screenshot:


***Figure 2.6***

A container might exist in an exited state if it completed what it was supposed to do. We will then need to manually restart the container for us to use it. Let us say we want to create a BusyBox container with the express purpose of pinging the localhost six times. We want to see the output on the terminal, so we will not run this container in detached mode. Let us run the following command on our host system:

```
docker run -it --name my_container1 busybox:latest ping -c 6
localhost
```

And we get the following output:


***Figure 2.7***

We can see that the container **`my_container1`** was created, and it pinged the localhost six times, and then it exited. So, we can see the status as exited. We also notice the other container **`my_container`** still in a paused state. Let us *unpause* **`my_container`** just to be clear on how the pause/unpause mechanism works by running the following:

```
docker container unpause my_container
docker container ls -a
```

See the output in the following screenshot:


***Figure 2.8***

Well, as expected, the container **`my_container`** is unpaused and is now up and running. The container is shown to be running for 53 minutes, although we know for a fact that it was paused for quite some time in between.

When we restart a docker container, we can catch the container in the process of being restarted. The state will then be shown as restarting.

A docker container would show a status of ‘dead’ if we were not successful in fully removing a container, when we attempted to do so, because resources may have been busy or unavailable when we were trying to remove the container, resulting in the container being only partially removed. Such containers cannot be brought up and may need to be manually cleaned up, although the docker daemon does try to remove dead containers when restarted.

Finally, let's see a couple of examples: one of stopping a running container by using the stop command, and another of using the create command to create a new container, without actually running it. See the following set of commands and the associated screenshot in *Figures 2.9* and *2.10*:

```
docker container stop my_container
```

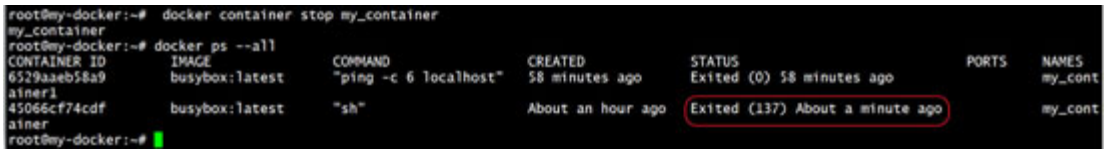



***Figure 2.9***

```
docker container create --name my_container2 alpine:latest
```


***Figure 2.10***

So, we see the stopped container with the status exited and a code of 137, as shown in *Figure 2.8*. In *Figure 2.9*, we see the status of the container as created. This implies that the container is not started as yet on the system.

## Getting inside a Container

Let us create a BusyBox container and get inside it. We use the `exec` option to get inside the container. See the commands and the screenshot below. We are in interactive mode with the `tty` option. Let us run the following commands:

```
docker run -dit --name test_cont busybox:latest
docker exec -i -t test_cont sh
```

See the following screenshot:

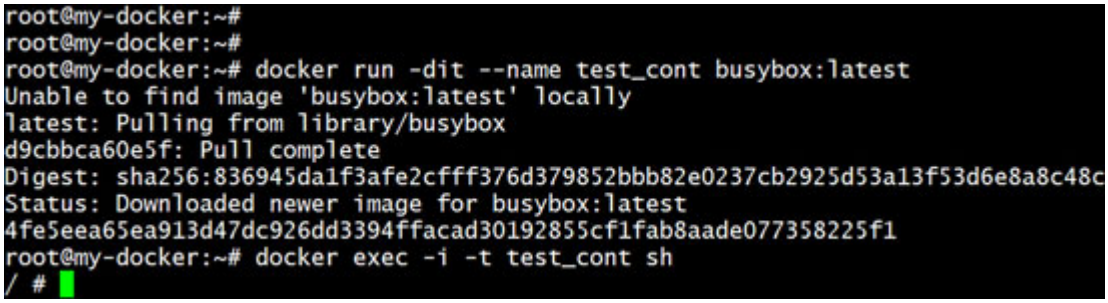



***Figure 2.11***

Inside the container, we can run commands pertaining to the container. For example, this is a BusyBox container, and this provides us with a program that provides all the commands required to make a good embedded Linux environment. This means that we can run commonly used Linux commands inside the container. Similarly, had we installed a database container like MySQL, we could have run database related commands inside it. As we discussed in *Chapter 1*, containers provide an insulated environment through the configuration of namespaces, for us to run our commands safely and securely.

Now that we are inside the container let us run a few simple Unix commands and see how things pan out. Check the screenshots in *Figures 2.12* and *2.13*.

```
ps
ls
mkdir test
cd test
touch myfile
```

See the following screenshots:

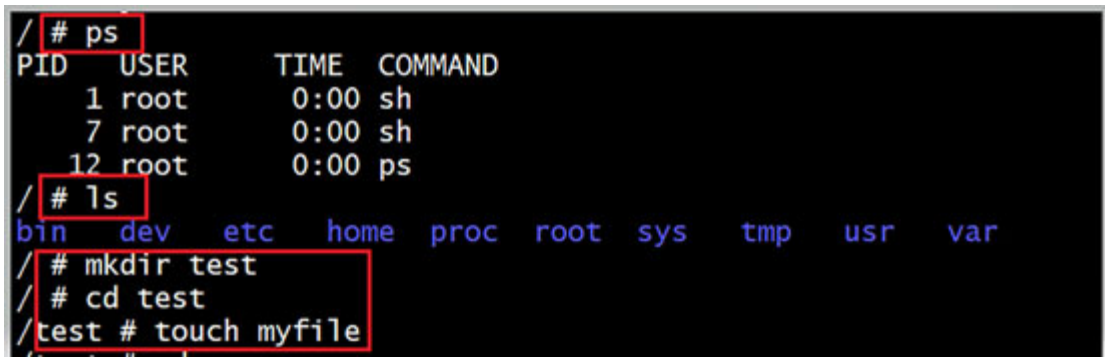


*Figure 2.12*

```
cd ..
ls
```


*Figure 2.13*

We see that we can run the commands as if we are on a Unix based operating system host. That is the big idea-*the containers provide us a mechanism to have our own database, our own operating system, our own web server, or just about anything that we can run on the host machine, encapsulated in a container*. While we can do something similar with virtual machines, the container is streets ahead of virtual machines as far as performance is considered, being much more lightweight and nimble, as it does not have a hypervisor layer or its separate operating system, which all adds to the performance degradation of virtual machines.

## Inspecting a Container

We can inspect a container by running the command `docker container inspect <container_id/container_name>` **or just** `docker inspect <container_id/container_name>`. The information is generated as a JSON object. There is a huge amount of information generated about the container id, about the image used to build the container, about the host configuration, about graph drivers, and network settings. If we know what to glean from this information, then this is really a treasure trove for us.

Suppose we want to inspect the container `test_cont` that we created a while ago. Let us run the following and check the output:

```
docker container inspect test_cont
[
  {
    "Id":
    "8a03514f6d3e9bb46c79f40220624ee690db23ed77fb65106a447c2d04fa
```

```
   fc66",
   "Created": "2020-05-28T04:49:50.071682396Z",
   "Path": "sh",
   "Args": [],
   "State": {
    "Status": "running",
    "Running": true,
    "Paused": false,
    "Restarting": false,
    "OOMKilled": false,
    "Dead": false,
    "Pid": 4386,
    "ExitCode": 0,
    "Error": "",
    "StartedAt": "2020-05-28T04:49:50.701489625Z",
    "FinishedAt": "0001-01-01T00:00:00Z"
   },
   "Image":
"sha256:78096d0a54788961ca68393e5f8038704b97d8af374249dc5c8faec
1b8045e42",

   "ResolvConfPath":
"/var/lib/docker/containers/8a03514f6d3e9bb46c79f40220624ee690d
b23ed77fb65106a447c2d04fafc66/resolv.conf",

   "HostnamePath":
"/var/lib/docker/containers/8a03514f6d3e9bb46c79f40220624ee690d
b23ed77fb65106a447c2d04fafc66/hostname",

   "HostsPath":
"/var/lib/docker/containers/8a03514f6d3e9bb46c79f40220624ee690d
b23ed77fb65106a447c2d04fafc66/hosts",

   "LogPath":
"/var/lib/docker/containers/8a03514f6d3e9bb46c79f40220624ee690d
b23ed77fb65106a447c2d04fafc66/8a03514f6d3e9bb46c79f40220624ee69
0db23ed77fb65106a447c2d04fafc66-json.log",

   "Name": "/test_cont",
```

```
    "RestartCount": 0,
    "Driver": "overlay2",
    "Platform": "linux",
    "MountLabel": "",
    "ProcessLabel": "",

……..
……..
```

The output has been truncated for brevity, but we get a sense of the amount and type of information that is available for us when we inspect a container. As we will see later, we can also use the inspect command to get detailed information about an image as well as other docker objects.

## The Container Logs

Running containers generate information that is logged. We can use the docker logs or the docker service logs (in case we are running a service-services are discussed in detail in *Chapter 5*) to get the information logged by the containers. We will be able to see the information on our terminal, because by default Linux outputs to the terminal (`STDOUT`), while error messages are outputted using `STDERR`.

Sometimes we need to do some additional configurations to ensure we get useful information from the docker logs. For example, if the logging driver we are using, sends our logs elsewhere, say to a file instead of `STDOUT` or `STDERR` then obviously looking into the default docker logging location will provide us with no useful information.

Before we move ahead, we need to understand what exactly a logging driver is. A logging driver is a piece of software that helps us get information from running containers and services. The Docker daemon has a default logging driver that our container uses to log information unless we configure our daemon to use a different logging driver, in which case our container uses the logging driver configured on the system at that point of time. The default logging driver for docker is json-file.

We can use the following command to check the logging driver configured on our system.

```
docker info --format '{{.LoggingDriver}}'
```

See the output of the command in the following screenshot:

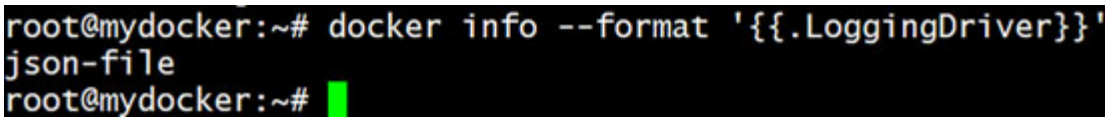



***Figure 2.14***

Docker supports different logging drivers, and we can simply edit the value of the log driver in `/etc/docker/daemon.json` followed by restarting the Docker daemon to reload its configuration.

The new driver settings will now apply to all containers which are created post-editing of the `daemon.json` file. However, there is another mechanism to override the default logging driver as well, and that is to run the container with `--log-driver` and `--log-opt` options.

Docker container logs are stored as JSON files in `/var/lib/docker/containers/<container_id>` directory on the host system. The logs are named as `<container_id>-json.log`. These logs come from the output streams, are annotated with the log origin, either `stdout` or `stderr`, and a timestamp. However, the thing to remember here is that each logfile that gets created contains information about only one container.

Now, let us check it for ourselves by creating a container and trying to access the logs by running the following set of commands:

```
docker run --name test -dit alpine:latest sh -c "while true; do
$(echo time) sleep\ 10; done"
cd /var/lib/docker/containers
ls
cd
0c485e553fe0c75ab51aa8d796d99b1681a666e4ecc8732cc395bbfbba30edd
f
ls -ltr
cat
0c485e553fe0c75ab51aa8d796d99b1681a666e4ecc8732cc395bbfbba30edd
f-json.log
```

Have a look at the following screenshot:

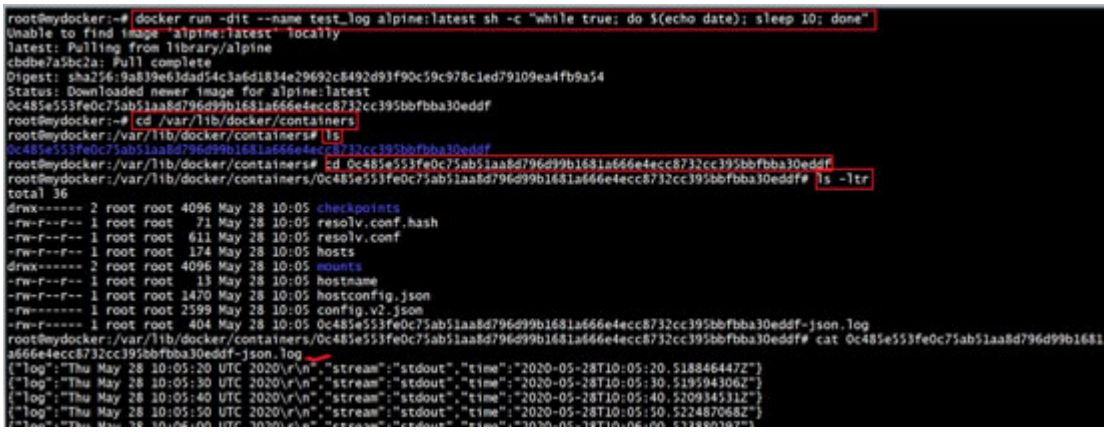



***Figure 2.15***

There we go. We can see the logs being generated every ten seconds showing the date and time on the system. These are the container logs.

We ought to be mindful of the fact that keeping these logs for any length of time on the system can consume substantial disk space, so it's always a best practice to use a log aggregator to collect the logs and store it in a central location, while at the same time enabling log rotation on the system.

One of the biggest challenges of docker logging is to consider is that we need to keep track of several sets of logs. There are logs generated for the application(s) inside the container as well as logs on the host system, so having a well-thought-out strategy for log management is essential for the successful implementation of docker in our setup.

Before we conclude this section, let us get acquainted with some typical commands relevant for log management. For this, let us create a container named `test_logs`, which echoes the date and time every 10 seconds. Then we use different commands to check the logs tailoring it to our needs. See the following set of commands.

```
docker run --name test_logs -dit alpine:latest sh -c \
"while true; do $(echo time) sleep 10; done"
docker ps
docker logs b0c4ee4a7b9f
```

Now, check out the following output:

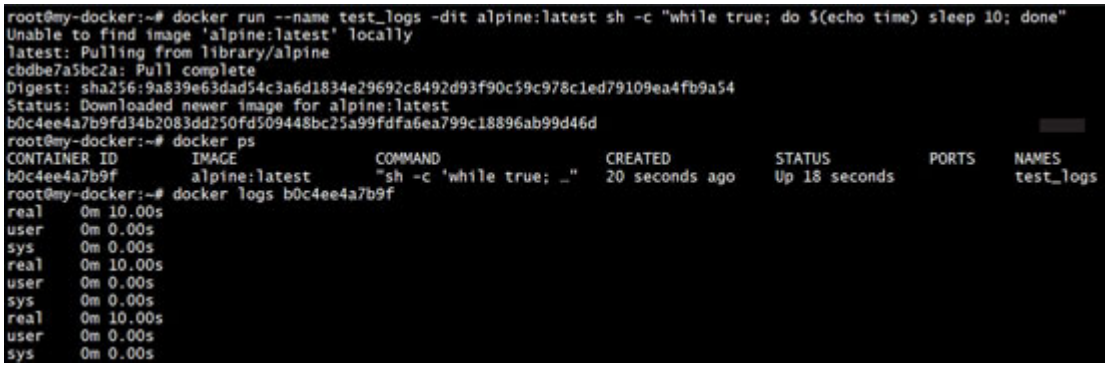



***Figure 2.16***

We can also choose to use the flag `--follow` or `-f` to literally '*follow*' the container logs as shown in the following example:

```
docker logs b0c4ee4a7b9f –follow
```

See the following screenshot:

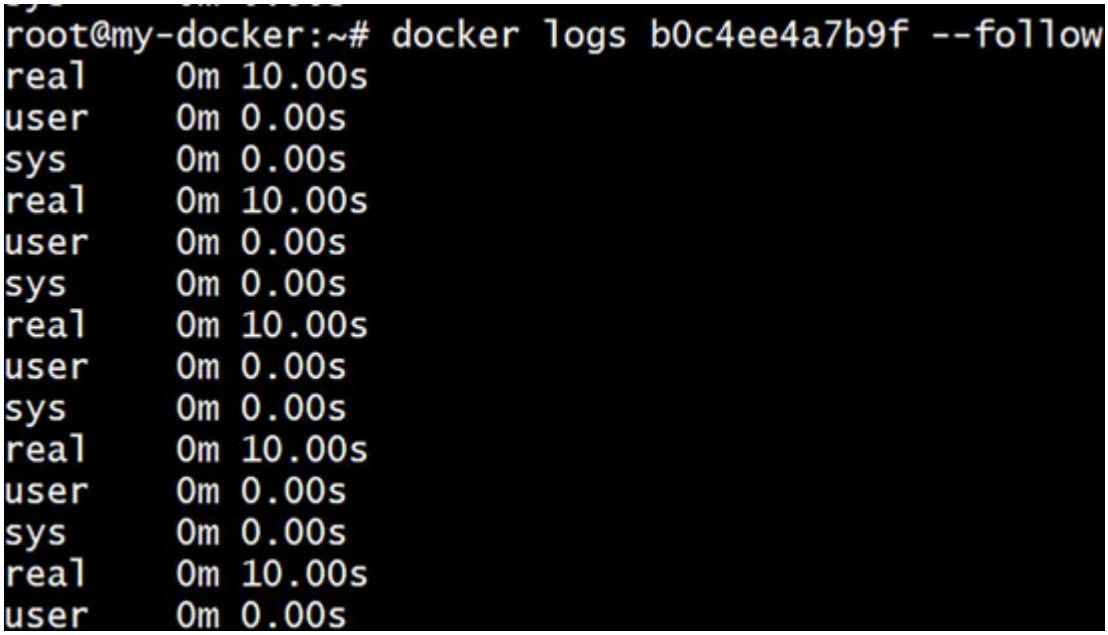



***Figure 2.17***

To get a more verbose output, we can use the `--details` flag. In the instant case, though, the output is similar to a non-verbose output because there aren't any additional details in the logs to be

displayed. Check the command and its following associated screenshot:

```
docker logs b0c4ee4a7b9f --details
```

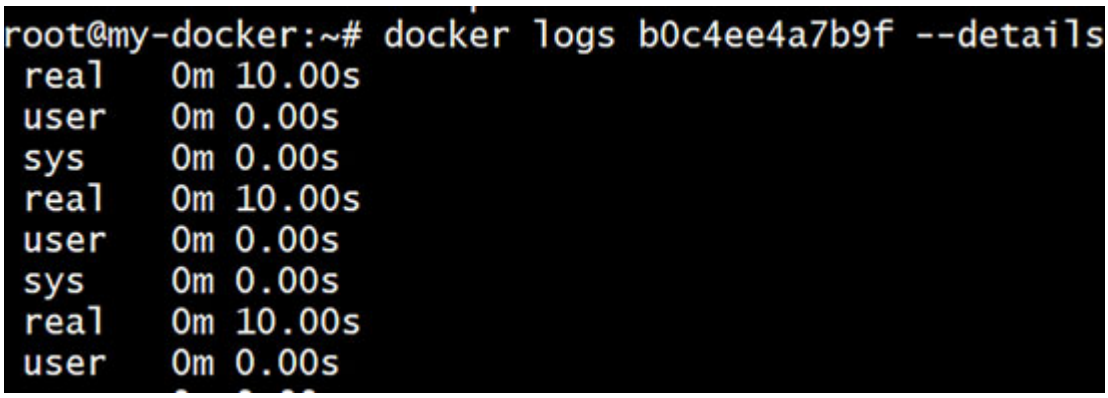



***Figure 2.18***

We can also tail the logs using the `--tail` command. Tailing the logs will show us the last '*n*' lines in the output. '*n*' is the number of lines of the output we want to see. In the following example, we see the last 8 lines of the logs.

```
docker logs b0c4ee4a7b9f --tail 8
```

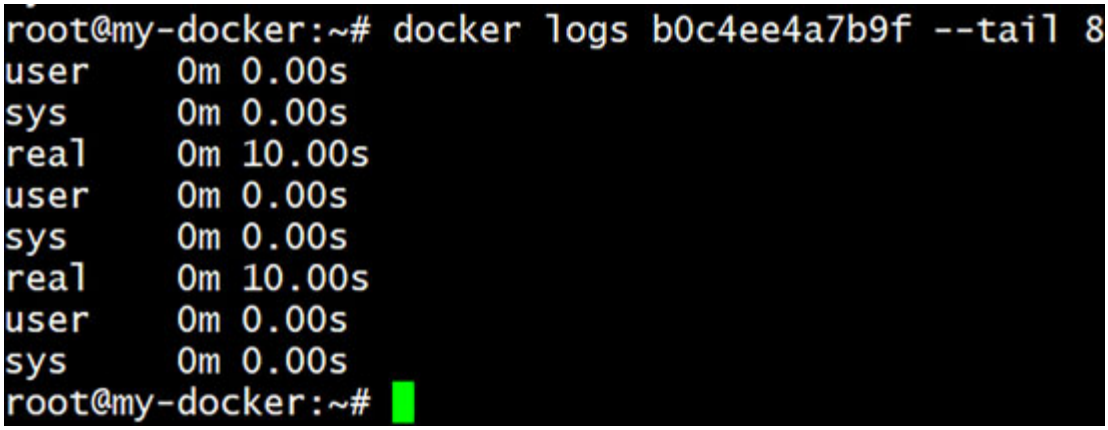



***Figure 2.19***

The flag `-t` allows us to see the timestamp of the logs. Let's have a look at the following example:

```
docker logs b0c4ee4a7b9f -t
```

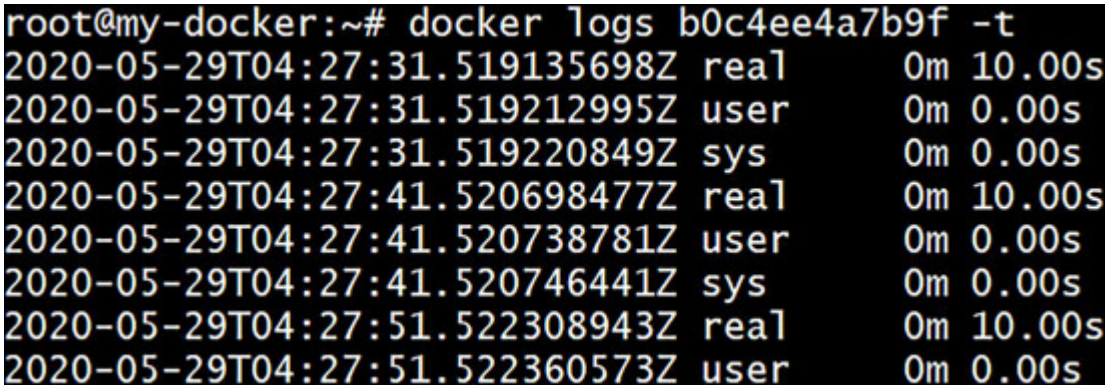



***Figure 2.20***

We can easily combine flags to get more meaningful outputs. For example, we can tail the output and use the timestamp flag as we do in the following example:

```
docker logs b0c4ee4a7b9f -t --tail 10
```

See the following screenshot:

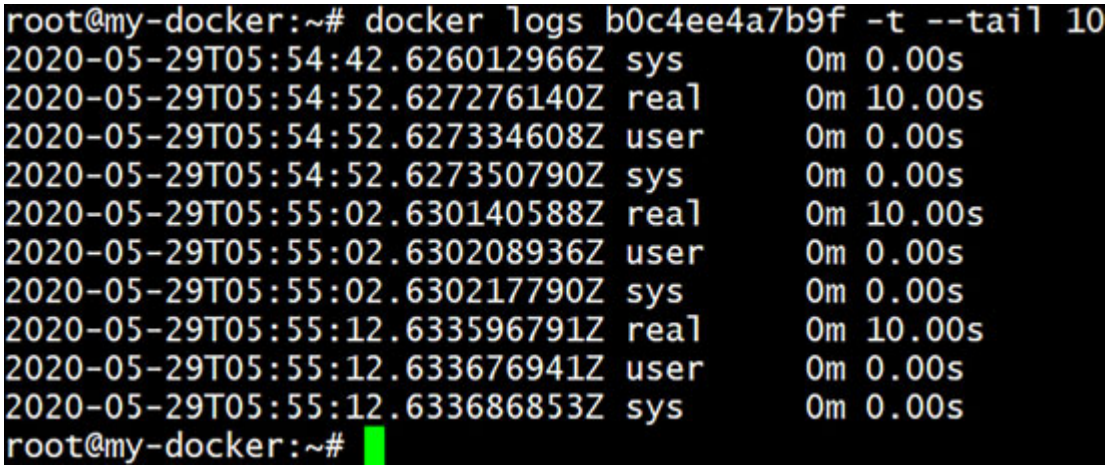



***Figure 2.21***

And we can also `grep` for a pattern in the logs or just look for errors by running the following commands:

```
docker logs <container_id> | grep pattern
docker logs <container_id> | grep -i error
```

Finally, log management in docker is a crucial function because with hundreds or even thousands of containers running on the system, successfully troubleshooting issues is directly predicated on how well

log management is configured on our system. Collating and combining information from so many containers to get a constructive input that will help in troubleshooting issues is no trivial stuff!

Although most of the log drivers can ship the logs to a central location or even to management tools, they don't allow us to parse and normalize the data. For that, we need a separate tool known as a log shipper. Examples of log shippers are Logstash and Filebeat. They can read data from multiple sources, parse and normalize them and send it to a central location in a format that is usable by tools like Kibana or Elasticsearch.

## Basic Container Architecture

To visualize the basic container architecture, let us have a look at the following diagram:

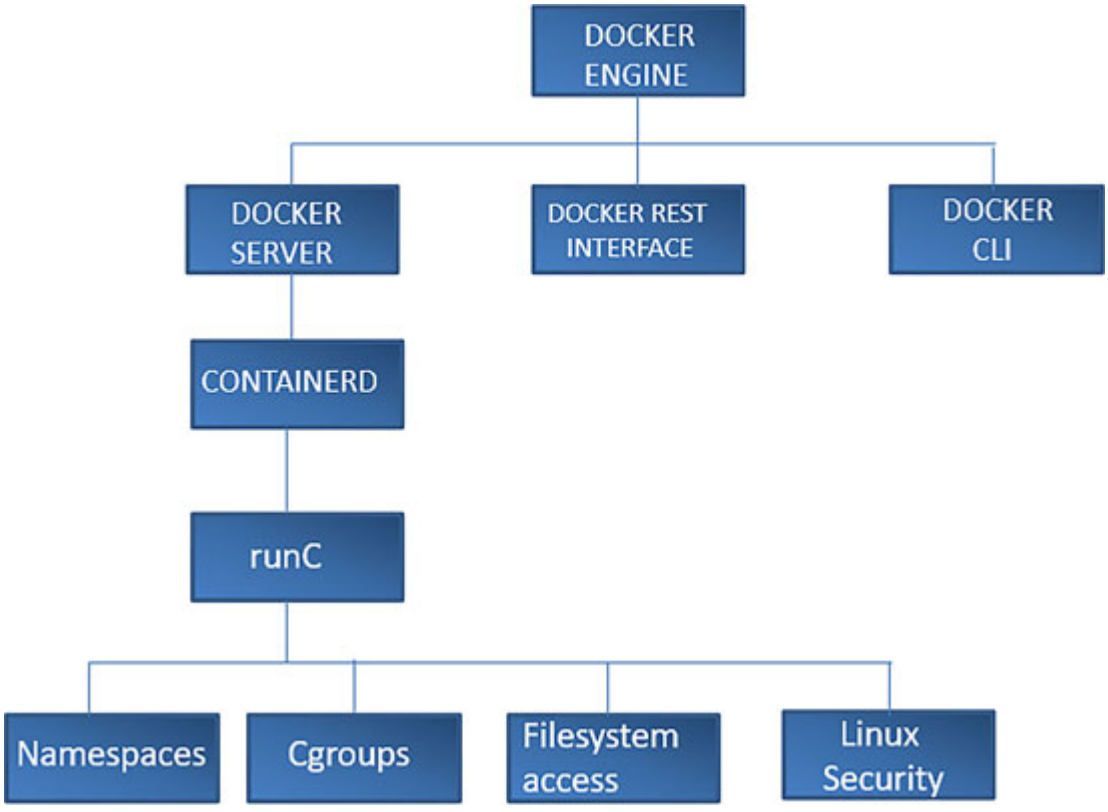



***Figure 2.22***

One of the central components of Docker is the Docker Engine. If we see the diagram above, we will see that the Docker Engine consists

of a docker command-line interface, a docker REST interface, and the docker server.

The docker server is a key component of the docker engine and consists of the dockerd or the docker daemon and is the piece of software that is responsible for the creation of Images, containers, networks, and so on.

The REST interface communicates with the docker server, while the docker CLI allows us to communicate with docker.

If we drill down deeper inside the docker server, we will get the containerd. We can consider the containerd as the piece of software that actually does all the heavy lifting for the docker server to create and manage containers.

However, the containerd, which is the container runtime, is dependent on another piece of software known as **runC**. We can consider runC as a low-level implementation of a container runtime; containerd builds on top of it and adds features like image management, container management, network management, and so on.

Getting back to runC, runC is a command line interface tool that follows the open container initiative as specified at https://www.opencontainers.org/. In other words, we can say that runC is basically a tool for spawning and running containers according to the specifications set out in the **Open Container Initiative** (**OCI**).

It is runC that allows us to leverage the Linux functionalities of namespaces, Cgroups, Linux-based security features like App Armor and SELinux, and filesystem access in the container ecosystem.

## What is a Docker Image

Let us begin by trying to describe what a docker image is. We can consider an image as a template in a read-only format that provides us a composite filesystem for creating containers. Images are made up of multiple layers of files and directories that are stacked in a particular order, and they form the basic building block of containers. The layers that form an image are '*immutable,*' or in other words,

once the layers are generated, they can never be changed. They may be physically deleted, but they cannot be modified. A container is created by adding a thin read-write layer on top of an image. This is extremely important as this opens up the possibility of creating *n* number of containers from a single image. As we will see later in the chapter, the sharing of images is a fundamental concept of Docker, and this is possible through mechanisms like copy-on-write that allows containers to change layers of an image by copying it up to its *read-write* layer and then incorporating whatever changes it requires to make.

## Getting deeper into Images

Images are not a composite block but are comprised of layers. Layers are stacked on top of each other. A layer is sometimes technically known as a '*diff'* because each layer '*builds'* on top of the layer below and is '*different'* from the layer(s) below.

Before Docker version 1.10, every layer corresponded to an image ID that could be presented in the format of either a short 12-digit hexadecimal string or a long 64-digit hexadecimal string. The image would be created by the mechanism of each layer referencing its parent layer below and the corresponding layer content until the base image was reached, which, of course, had no parent. The following diagram clarifies the concept:


***Figure 2.23***

However, since Docker version 1.10, layers do not have an image ID but are identified instead by a digest, which is in the form of Algorithm:Hexadecimal. See the following screenshot:

```
Digest: sha256:fe8d824220415eed5477b63addf40fb06c3b049404242b31982106ac204f6700
```

***Figure 2.24***

The biggest change now is that a layer is not synonymous with an image. Previously, that is, before version 1.10, a layer was synonymous with an image, or we could say that each layer corresponded to an image, and each image had an image id as an identifier.

The rationale for this change was that earlier there was no means to detect whether an image layer's content had been tampered with during a push or a pull from a registry. In other words, the security of an image layer was not guaranteed in any way. However, with the advent of the digest, it became much easier to check whether the image layer has been tampered with in any way or not by checking the computed digest value. The hexadecimal element is calculated by applying the algorithm (SHA256) to a layer's content. If the content changes, then the computed digest will also change,

meaning that it will become much easier to verify any security breach in the image layers.

*The docker image is now no longer created using the image of the parent; instead, the docker engine uses a list of layer digests ordered in a particular way to compile the image.*

In other words, an image now consists of a configuration object, which is nothing but the SHA digests in a particular order that the Docker Engine can utilize to create the filesystem to be used by the container, which will be created from this image. See the following diagram:

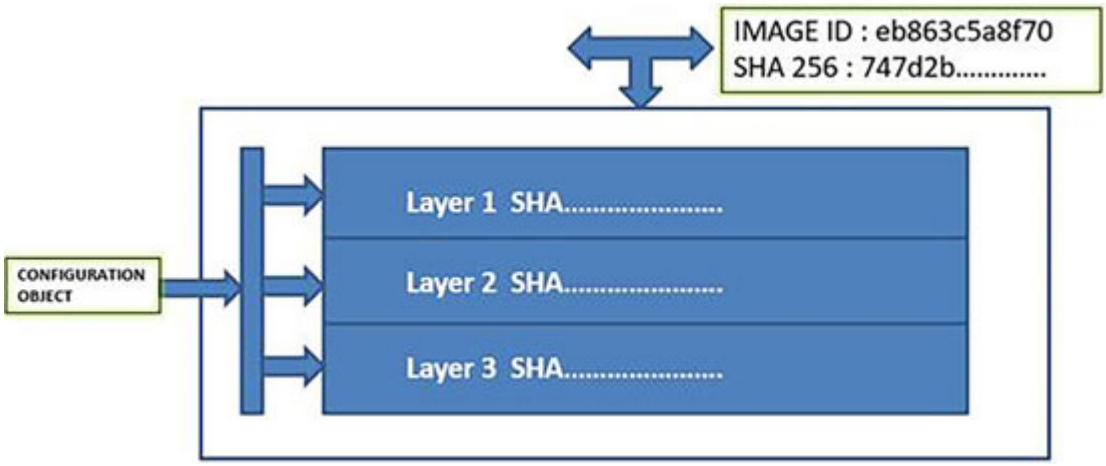



***Figure 2.25***

Let us create an image to understand how it all works. Let us run the following command and then check the output in our screenshot in *Figure 2.24*:

```
docker image pull ubuntu:latest
```

```
root@my-docker:~# docker image pull ubuntu:latest
latest: Pulling from library/ubuntu
d51af753c3d3: Pull complete
fc878cd0a91c: Pull complete
6154df8ff988: Pull complete
fee5db0ff82f: Pull complete
Digest: sha256:747d2dbbaaee995098c9792d99bd333c6783ce56150d1b11e333bbceed5c54d7
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
root@my-docker:~#
```

***Figure 2.26***

We have pulled an image of ubuntu from the Docker Hub registry, and as we can see, the image consists of three layers. The digest of the image is

`sha256:747d2dbbaaee995098c9792d99bd333c6783ce56150d1b11e333bbceed5c54d7` visible in the screenshot in *Figure 2.24*. The image will also have an identifier, technically known as an image id which we can see by running one of the following commands:

```
docker image ls
docker image ls --all
docker image ls --digests
```

We will use the last command as that produces the most comprehensive output:


***Figure 2.27***

We see the output showing both the image id as well as SHA digest, which we may have noticed when we were pulling the image from the registry.

Okay, let us now run the following command to see the output:

```
docker image history 1d622ef86b13
```


***Figure 2.28***

What strikes us is the **<missing>** in the IMAGE column. Although it's an unfortunate way of showing the output, there is no error here. Since docker version 1.10, each layer does not have a corresponding image id; therefore, that part is shown as **<missing>**.

The id shown in the first row of the output doesn't belong to the topmost layer of the image either; instead, the entire image is identified by that id displayed at the top. In other words, technically, that is the image id.

So, let's see where we stand now. We know that images are nothing, but a filesystem provided for us to create containers from. An image comes in layers, stacked one upon the other in a particular order, with each layer on the top storing only that what is different from the layer below it. Of course, the layer on top could be completely different than the layer below it, in which case the entire layer would be '*diff*'. But if the layer on the top needs some components which happen to be available in the layer below, then this layer will only store the unique (or different) components sharing the rest of it with the layer below.

The layer '*diffs*' are referenced using an SHA 256 hash of the '*diff's*' contents. We can create numerous containers from the same image because the containers have a thin read-write layer on the top, which is unique to the container, other than that the rest of the image remains the same.

## Copy on Write

Copy-on-write is a perfect example of sharing access to the same data amongst images and containers until the point any process wants to modify or write to the data, whereupon the operating system makes a copy of the data for that process to use. In other words, you don't get your own copy unless you have a requirement to write and make changes to an image. All other processes will continue accessing the original data.

If a container wants to make some changes to an image by *modifying a particular layer*, then that layer is copied up into the read-write layer where the changes are made. Docker makes use of the copy-on-write mechanism with both images as well as containers. To do this, changes between the image(s) and the running container(s) are tracked using storage drivers like *AUFS* or *OverlayFS* or something called *snapshotters,* which we learn about in the next chapter.

## Where are the images stored?

Typically, in Linux, the images are stored on the disk under `/var/lib/docker/<storage-driver>` directory. The layers will be stored as directories. Let's have a quick look at how they look.

We can run the docker info command to find out the storage driver we are using. See the command and the following screenshot:

```
docker info
```

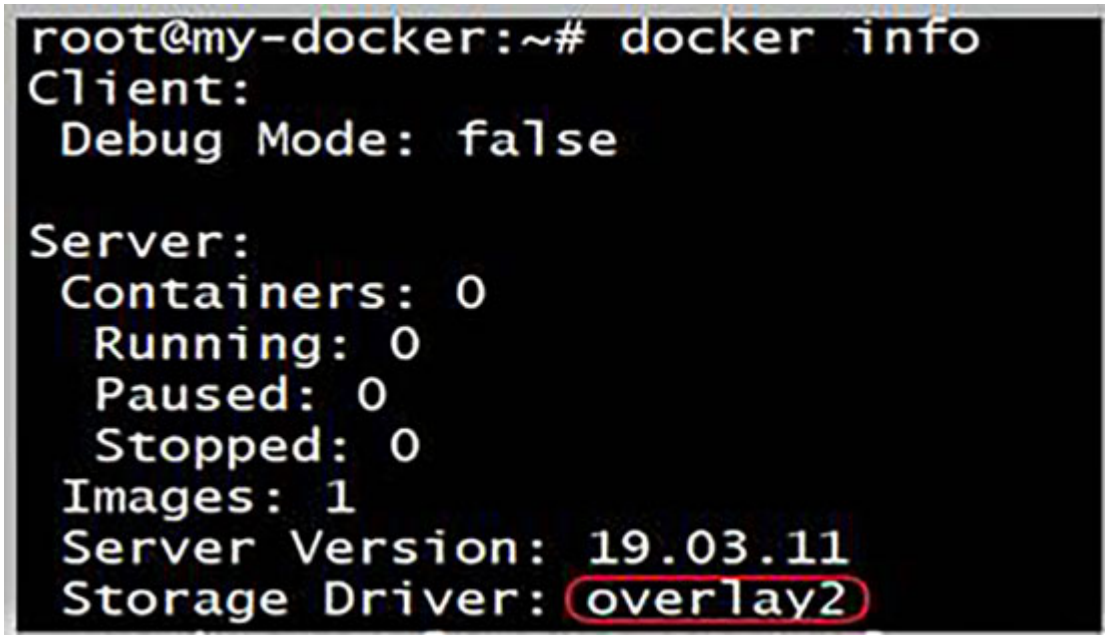



***Figure 2.29***

Output truncated for brevity. Let's check where the image is stored by running the following set of commands:

```
cd /var/lib/docker/overlay2
ls -ltr
```

We should get an output similar to the following:


***Figure 2.30***

So we can see the four layers plus one additional layer named ‘l’. We will look into more details about these in the next chapter.

## Inspecting an image

We will find an enormous amount of information when we run the command `docker image inspect <image_id>`. All that information may be somewhat unwieldy and not very reader-friendly. We may use the `jq` tool for formatting the output. `jq` is a lightweight and flexible command-line JSON processor. Since the output comes in json format, the jq-tool can be used to get an overview of the output and pick interesting parts. We can download the tool using the command `apt install jq`.

For example for checking the layers in an image we can run the following:

```
docker image inspect 1d622ef86b13 | jq .[0].RootFS
```

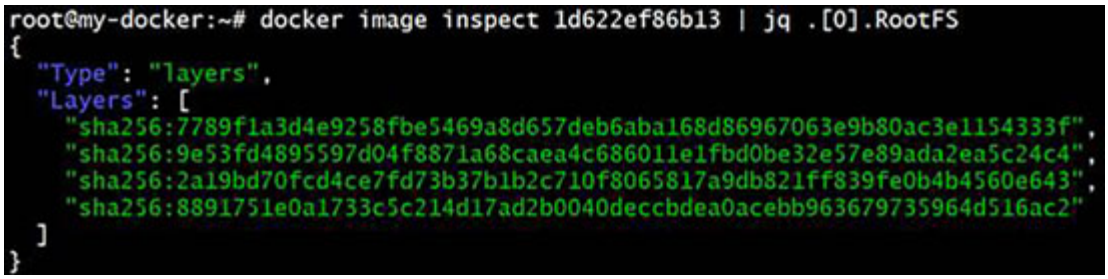



***Figure 2.31***

We can see all the image layers with their SHA digests. And no, they don’t match the directory names that we see in *Figure 2.28*.

## Saving an image

We can save an image as a tarfile. See the following commands and screenshot:

```
docker image pull nginx:latest
docker image ls
docker image save 4392e5dad77d > mynginx.tar
ls -ltr
ls -sh mynginx.tar
```


***Figure 2.32***

We can use the `--output` flag to create a backup that can then be used with docker load.

See the following command and screenshot:

```
docker save --output mynginx.tar nginx
```


***Figure 2.33***

## Using the COMMIT command to save an image

We can use the `COMMIT` command to create and save an image without using a Dockerfile. (we are going to see the Dockerfile later in the chapter).

To understand this concept, let us create an ubuntu container and get inside the container.

```
docker run -dit --name ubuntu ubuntu:latest
docker exec -it ubuntu sh
```

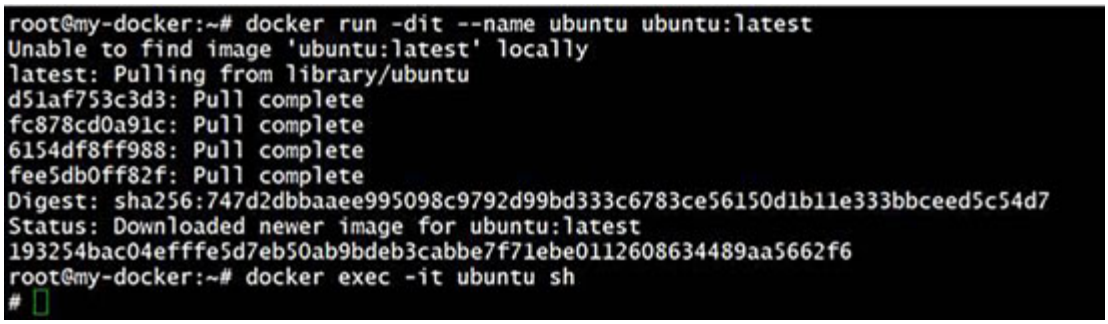


***Figure 2.34***

Now that we are inside the container let us try to ping google.com. But we find that the command fails for the excellent reason that the '*ping'* command is not installed on the Ubuntu OS that we used to create our container. See the following command and screenshot:

```
ping google.com
```

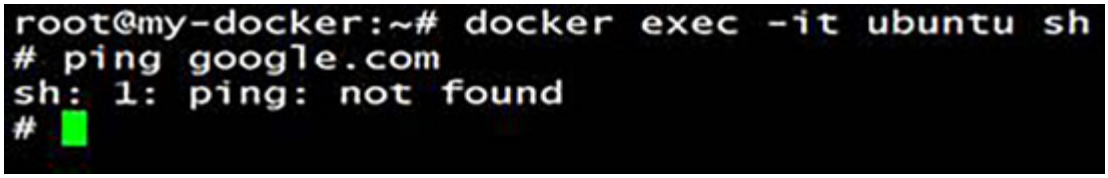



***Figure 2.35***

So, let us go ahead and install the '*ping'* command by running the following:

```
apt update && apt install iputils-ping -y
```

Once this command is installed successfully, let us try to ping google.com once again and see if it is working now.

```
ping google.com
```

And there we go-it works perfectly.

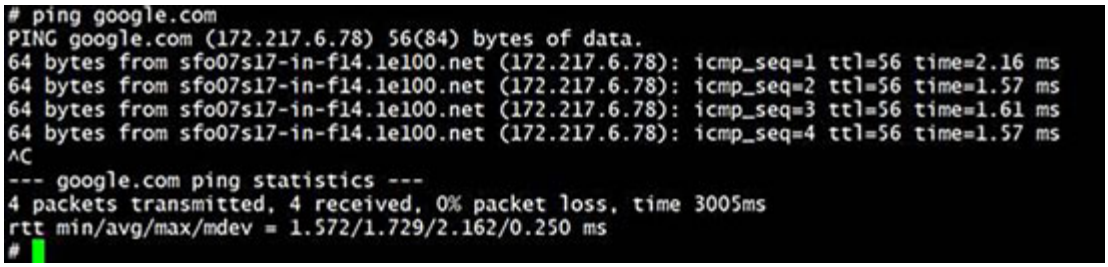



***Figure 2.36***

Now it would be great to have the '*ping'* command installed in our ubuntu image, so let us save this container into an image that we can use later. We will then not have to bother about installing the '*ping'* command, because it will be already installed in that image. We exit from the container on to the host system and just use the

commit command to save the container into an image. See the following commands and screenshot:

```
exit
docker commit ubuntu myubuntu
```

Here ubuntu is the name of our container, and `myubuntu` is the name of the image that we are creating.

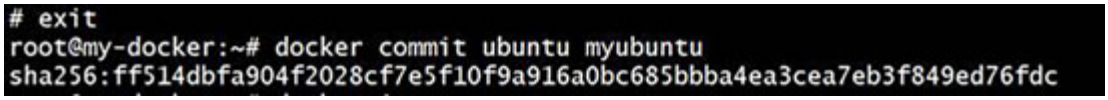



***Figure 2.37***

And now, if we check the images, we will find the new image named `myubuntu` available on the system. This can now be used for creating a container(s). Let's try doing it as follows:

```
docker images
```


***Figure 2.38***

## The Dockerfile

The Dockerfile is a text file in which we put in the commands that help us assemble an image when we '*build'* from that Dockerfile.

While the Dockerfile is a text file, unlike regular text files, it doesn't carry the `.txt` extension. When we save the Dockerfile, we save it without any extensions.

By now, we are conscious of the fact that images are nothing but layers of files and directories stored in a particular order that allows the Docker Engine to create a filesystem to be used by containers that will eventually be created from these images. In the Dockerfile, every line that we write creates a layer of the docker image.

Docker starts reading the Dockerfile from top to bottom and is case insensitive, although conventionally, the commands are written in uppercase. Docker reads each line of command in the Dockerfile

and executes them, resulting in a layer getting built for each line in the Dockerfile.

Let’s have a quick look at some of the commands that are typically used while creating a Dockerfile.

FROM: The FROM command is the first command that we will be putting into the Dockerfile, and this usually states what operating system we intend to use as the base image.

**We have to keep in mind that ARG is the only instruction that may precede FROM in the Dockerfile. The ARG declared before a FROM is technically outside our build stage, and none of the instructions after FROM will pick it up. If we want an ARG value declared before FROM to be used inside the build stage, we just need to use an ARG instruction without a value inside of a build stage.**

LABEL: The LABEL instruction adds metadata to an image. A LABEL is a key-value pair. Typically, we use the LABEL to provide more granular information about our image.

ENV: As we can guess, this command is used to set our environment variables while building the docker image. These variables will be there when we launch our container.

RUN: The RUN command is the major executing command in the Dockerfile. It will create a new layer on top of the current layer and commit the results. The RUN command is used to build the image. If we are using the RUN command in the shell form, we can use a \ (backslash) to continue a single RUN instruction onto the next line.

CMD: The CMD command has some similarities with the RUN command discussed earlier. However, the CMD command is not executed during a build but during the instantiation of the container using the image being built. It is the default command that gets executed when a container comes into existence based on this image.

ENTRYPOINT: It states what will happen every time a container is created using the image. For example, if we have installed a specific application inside an image, and we want that application to get started when we create a container from this image, we can state it

with ENTRYPOINT. However, it may seem that ENTRYPOINT is suspiciously similar to the CMD command; we can consider the ENTRYPOINT as the parent command. At the same time, CMD will be used to pass the parameters.

ADD: The ADD command just copies the files from the source (on the host) to the destination (container's filesystem). In case the source is a URL, then the contents of the URL gets downloaded and are placed at the destination.

EXPOSE: The EXPOSE instruction just lets Docker know about the ports on which the container will listen at runtime. By default, the protocol is TCP, although the UDP protocol may also be specified. But it must be remembered that EXPOSE does not actually publish the port, but just lets everyone know about the port or ports that can be published when the container is run.

WORKDIR: The WORKDIR directive sets out the working directory for any RUN, CMD, ENTRYPOINT, COPY, and ADD instructions that will follow in the Dockerfile. If WORKDIR doesn't exist, it will be created even though eventually, it may not be used in any of the Dockerfile instructions.

USER: The USER directive is very straightforward. It is used to set the UID (or username) or optionally the group of the user (GID) to be used when running the container created from the image based on any RUN, CMD, and ENTRYPOINT instructions that follow it in the Dockerfile.

VOLUME: The VOLUME instruction simply creates a mount point with the specified name. We can write the value of the volume as a JSON array or just as a plain string. The docker run command will initialize the newly created volume.

## The Build Cache

When we build an image, Docker goes through the instructions in our Dockerfile and executes the instructions step by step. However, there is a piece of optimization involved here. Before executing an instruction, Docker looks for an existing image in its cache, and if it exists, then it will use it rather than create a duplicate image. This

cache is known as the Build Cache and is an important step in the overall optimization that permeates across Docker.

In this context, it is pertinent to mention that the locally built images on a Docker host are treated slightly differently. While the basic content of a locally built image remains the same as an image pulled from a registry, the configuration object containing configuration items, including an ordered list of layer digests, differs slightly.

There is a concept of an '*intermediate image*' here. When we commit a layer during an image build, an intermediate image is created. This intermediate image has all the trappings of a '*real image*'. Just like a real image, it has a configuration item which, as we know, is a list of digests that the Docker Engine uses to create a filesystem. These intermediate images also contain an ID of the parent image, which is the image of the layer immediately below it.

The purpose of the intermediate images and the reference to parent images is to allow a *match to be made with existing images in the Build Cache*. If a match is found, Docker will use that instead of regenerating the content needlessly, and continue ahead.

## Use Multi-Stage Builds

Another optimization that we can do while building our image is to go for multi-stage builds, which reduces the size of our images quite substantially. With multi-stage builds, we use multiple `FROM` statements in our Dockerfile. Each `FROM` instruction uses a separate base and begins a new stage of the build. We can then selectively copy whatever we require from one stage to another, leaving behind everything we don't require in our final image. Multi-stage builds allow us to substantially reduce the size of our final image, without always having to keep an eye out to reduce the number of layers in the image.

Let's write a Dockerfile now and build it.

```
vi Dockerfile
FROM ubuntu:latest
ENV HOME /root
LABEL ubuntu=myubuntu
```

```
ENTRYPOINT [“sleep”]
CMD [“50”]
RUN useradd -m -G root testuser
USER root
RUN apt-get update && apt-get install net-tools -y
RUN apt-get install iputils-ping -y
```

Let us save and exit the editor and then run the build command as follows:

```
docker build --tag myubuntu_image.
```

The build commands build our image from the current directory (don’t miss the dot ‘.’ at the end of the command). If you notice carefully, we have deliberately not optimized the image by running several `RUN` commands separately, each resulting in a separate layer of the image being built and saved.

But when we build the same image more optimally, we will have just one `RUN` command, and all the `RUN` items will be clubbed together, separated only by double ampersands (`&&`). There will be a smaller image. See the following screenshots:

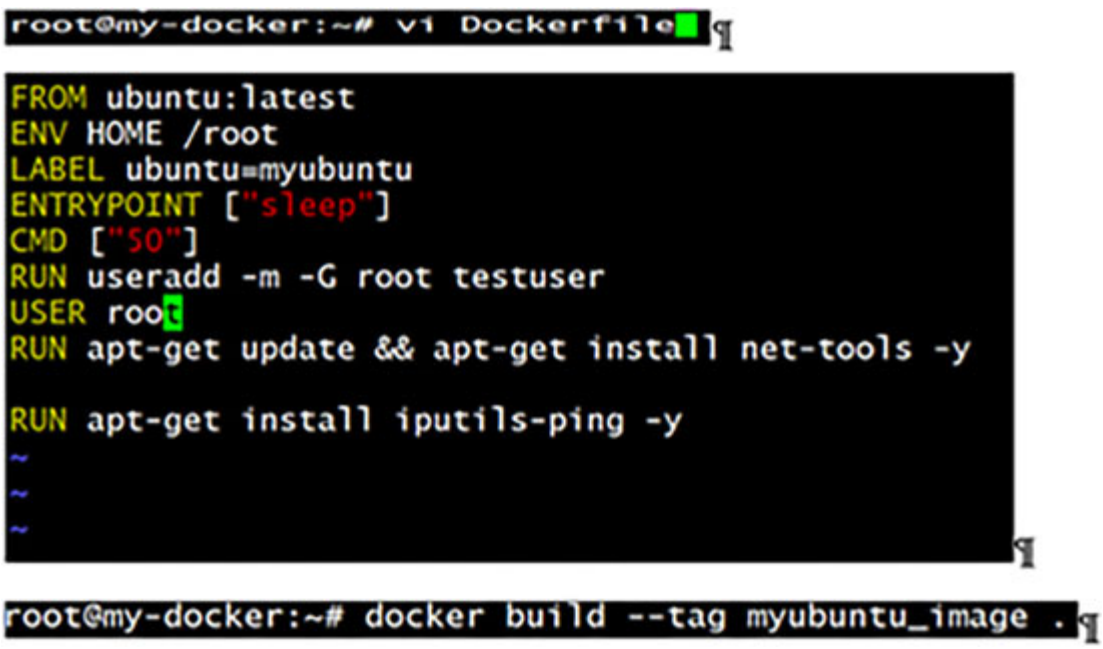



***Figure 2.39***

```
root@my-docker:~# docker build --tag myubuntu_image.
```

```
Sending build context to Docker daemon 16.38kB
Step 1/9 : FROM ubuntu:latest
latest: Pulling from library/ubuntu
d51af753c3d3: Already exists
fc878cd0a91c: Already exists
6154df8ff988: Already exists
fee5db0ff82f: Already exists
Digest:
sha256:747d2dbbaaee995098c9792d99bd333c6783ce56150d1b11e333bbce
ed5c54d7
Status: Downloaded newer image for ubuntu:latest
 ---> 1d622ef86b13
Step 2/9 : ENV HOME /root
 ---> Running in 37112affa648
Removing intermediate container 37112affa648
 ---> 9646522325be
Step 3/9 : LABEL ubuntu=myubuntu
 ---> Running in 9801e9252462
Removing intermediate container 9801e9252462
 ---> bd68bd81ff42
Step 4/9 : ENTRYPOINT [“sleep”]
 ---> Running in 94d8fbefddc5
Removing intermediate container 94d8fbefddc5
 ---> d79ce88560e6
Step 5/9 : CMD [“50”]
 ---> Running in 30bcc2a0452e
Removing intermediate container 30bcc2a0452e
 ---> 0c47c79e948e
Step 6/9 : RUN useradd -m -G root testuser
 ---> Running in f0ee56d8ce0d
Removing intermediate container f0ee56d8ce0d
 ---> b5d03fc71fd8
Step 7/9 : USER root
 ---> Running in d4b5afccc880
Removing intermediate container d4b5afccc880
 ---> 86f88926749a
Step 8/9 : RUN apt-get update && apt-get install net-tools -y
```

```
 ---> Running in 8b5033e1a4c9
Get:1 http://archive.ubuntu.com/ubuntu focal InRelease [265 kB]
Get:2 http://archive.ubuntu.com/ubuntu focal-updates InRelease
[107 kB]
Get:3 http://archive.ubuntu.com/ubuntu focal-backports
InRelease [98.3 kB]
Get:4 http://security.ubuntu.com/ubuntu focal-security
InRelease [107 kB]
Get:5 http://archive.ubuntu.com/ubuntu focal/restricted amd64
Packages [33.4 kB]
Get:6 http://archive.ubuntu.com/ubuntu focal/main amd64
Packages [1275 kB]
Get:7 http://archive.ubuntu.com/ubuntu focal/multiverse amd64
Packages [177 kB]
Get:8 http://archive.ubuntu.com/ubuntu focal/universe amd64
Packages [11.3 MB]
Get:9 http://archive.ubuntu.com/ubuntu focal-updates/multiverse
amd64 Packages [1079 B]
Get:10 http://archive.ubuntu.com/ubuntu focal-
updates/restricted amd64 Packages [10.6 kB]
Get:11 http://archive.ubuntu.com/ubuntu focal-updates/main
amd64 Packages [224 kB]
Get:12 http://archive.ubuntu.com/ubuntu focal-updates/universe
amd64 Packages [126 kB]
Get:13 http://archive.ubuntu.com/ubuntu focal-
backports/universe amd64 Packages [2903 B]
Get:14 http://security.ubuntu.com/ubuntu focal-
security/restricted amd64 Packages [10.6 kB]
Get:15 http://security.ubuntu.com/ubuntu focal-security/main
amd64 Packages [117 kB]
Get:16 http://security.ubuntu.com/ubuntu focal-
security/multiverse amd64 Packages [1079 B]
Get:17 http://security.ubuntu.com/ubuntu focal-
security/universe amd64 Packages [39.8 kB]
Fetched 13.9 MB in 2s (8010 kB/s)
Reading package lists…
Reading package lists…
```

```
Building dependency tree…
Reading state information…
The following NEW packages will be installed:
 net-tools
0 upgraded, 1 newly installed, 0 to remove and 5 not upgraded.
Need to get 196 kB of archives.
After this operation, 864 kB of additional disk space will be
used.
Get:1 http://archive.ubuntu.com/ubuntu focal/main amd64 net-
tools amd64 1.60+git20180626.aebd88e-1ubuntu1 [196 kB]
debconf: delaying package configuration, since apt-utils is not
installed
Fetched 196 kB in 0s (3553 kB/s)
Selecting previously unselected package net-tools.
(Reading database … 4122 files and directories currently
installed.)
Preparing to unpack …/net-tools_1.60+git20180626.aebd88e-
1ubuntu1_amd64.deb …
Unpacking net-tools (1.60+git20180626.aebd88e-1ubuntu1) …
Setting up net-tools (1.60+git20180626.aebd88e-1ubuntu1) …
Removing intermediate container 8b5033e1a4c9
 ---> d986a915ba26
Step 9/9 : RUN apt-get install iputils-ping -y
 ---> Running in 6d4c5f56864d
Removing intermediate container 6d4c5f56864d
 ---> efda6ce9dfd0
Successfully built efda6ce9dfd0
Successfully tagged myubuntu_image:latest
root@my-docker:~#
```

Some portions of the output have been truncated for brevity, but nonetheless we get a sense of how Dockerfile is used to create an image. Let us see the following command and its output:

```
docker images ls --digests
```


***Figure 2.40***

In the next section, we have somewhat optimized the Dockerfile by clubbing all the RUN commands together separated by double ampersands, as shown in the following commands:

```
FROM ubuntu:latest
ENV HOME /root
LABEL ubuntu=myubuntu
ENTRYPOINT [“sleep”]
CMD [“50”]
USER root
RUN apt-get update && apt-get install net-tools -y && apt-get
install iputils-ping -y && useradd -m -G root testuser
```

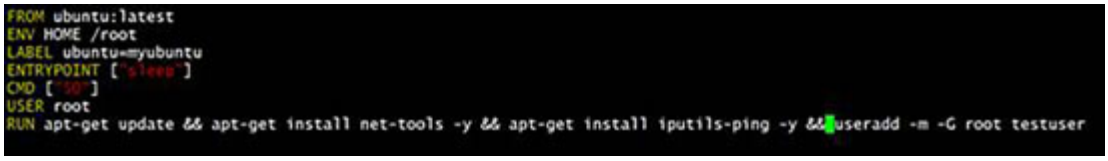



***Figure 2.41***

We notice that even by just ensuring that all the ‘RUN’s are clubbed together we get some benefit in the overall size of the image as can be envisaged from the following screenshot:


***Figure 2.42***

However, just to drive the point home, let us take a more potent example of how the size of an image can be influenced by combining the ‘RUN’s in a single line. We will do two examples. In the first example, we create a file of size 100 MB, and in the next run command, remove the file. In the second example, similarly, we

create a file of size 100 MB, and in the same line (layer), we delete the file.

Before we do these examples, let us make sure we have cleaned up our system by running the command `docker image rm <image_id>`.

Now let us run the following and check the screenshot 2.41 screenshot in *Figure 2.43*.

```
vi Dockerfile
FROM ubuntu:latest
RUN truncate -s 100M mytext.txt
RUN rm -rf mytext.txt
```

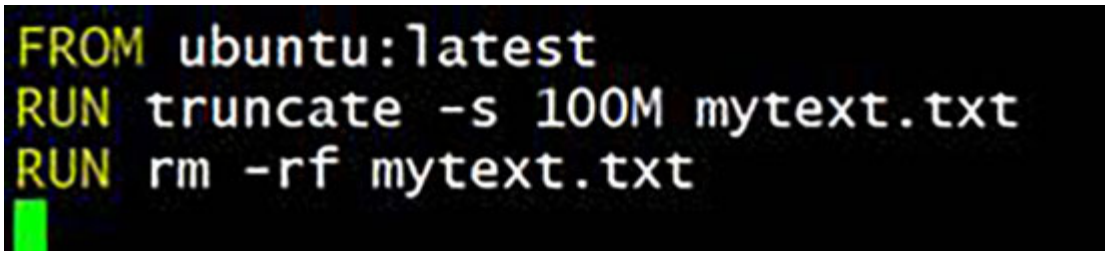



***Figure 2.43***

We save and exit from the editor and then run the following:

```
docker build -t test_size.
```

We see from the following screenshot that the Dockerfile was built successfully.

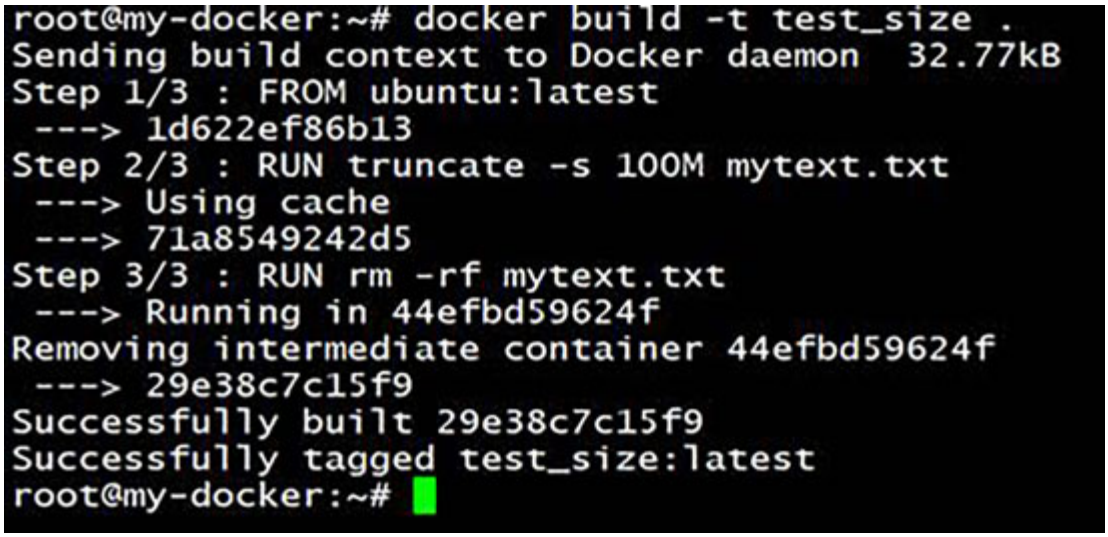


***Figure 2.44***

Now, let us go ahead and edit the Dockerfile and then save it with a different name.

```
vi Dockerfile
FROM ubuntu:latest
RUN truncate -s 100M mytest.txt && rm -rf mytext.txt
```


***Figure 2.45***

Save and exit from the editor and then run the following:

```
docker build -t test_size1.
```

This Dockerfile was gets built successfully as is evidenced by the following screenshot:


***Figure 2.46***

Now, with both the images having been successfully built, let us compare the size of the images by running the following command:

```
docker images
```


***Figure 2.47***

We see a huge difference in the size of the images `test_size` and `test_size1`. It is obvious that after creating a file and deleting it in the same layer, we do not add to the size of the image, but when we delete the file in a different layer, the size of the image gets inflated, simply because once the layer is created, it is immutable. Can the reader guess why the size of the image **ubuntu** and `test_size1` are exactly the same?

## Conclusion

In this chapter, we delved in some detail into the world of containers and images. We learned about how containers are created and run. We talked about the various states of a container and the implication of each state. We also learned about getting inside a container and running commands from therein. We talked briefly about the architecture of Docker and also about logging into containers as well as inspecting them. We also understood in some detail the anatomy of images, how they are created, and the relationship between containers and images. We also talked about the best practices about image creation, learned about the build cache and Dockerfile, and also had a peek at inspecting an image. Generally speaking, this chapter helped us develop an awareness of the container ecosystem.

In the next chapter, you will be learning about storage drivers and volumes.

## Points to Remember

- Containers are created from images, which may be considered as templates for creating containers.
- An image may be used to create an '*n*' number of containers.

- A container creates just a thin read-write layer on top of an image.
- Containers can be treated as ephemeral and can be spun up quickly.
- A container can exist in various states.
- The containers provide us a mechanism to have our own database, our own operating system, our own web server, or just about anything that we can run on the host machine, encapsulated in a container.
- It is crucial that proper log management be done on the containers to ensure container troubleshooting can be done optimally.
- Images are a template from which containers are spun up, but images themselves are immutable, that is, they cannot be changed or modified in any way once they are created.
- Images are layer-based.
- Layers have an SHA digest associated with them.
- An Image id itself is a digest that contains an SHA 256 hash of the image's JSON configuration object.
- Docker creates intermediate images to take advantage of the build cache.
- Images can be saved as a tarfile.
- Containers can be saved as images.
- Images can either be pulled from a registry or assembled in a Dockerfile.
- Dockerfiles are text files without any extensions.
- Dockerfiles are built into images.
- Images can be optimized by layering the image in such a way that the layers that the more volatile layers are put near the top while the more stable layers are put towards the bottom of the image stack.
- We should try to minimize the number of layers in the image to make it smaller in size, and by implication, more optimized

overall.

## Multiple Choice Questions

**Choose the most appropriate answer:**

1. The Dockerfile is
   a. An Image file
   b. A document
   c. A text file
   d. None of the above
2. A Container creates
   a. A thin read-only layer on the top of the image
   b. A thin read-write layer on the bottom of the image
   c. A thin read-only layer anywhere in the image
   d. A thin read-write layer on the top of an image
3. Images are stored in the directory
   a. /var/lib/docker/<storage_driver>
   b. /var/lib/docker/
   c. /var/lib/docker/netns
   d. None of the above
4. Docker logs are stored as JSON files in
   a. /var/lib/docker/<container_id>
   b. /var/lib/docker/containers/<container_id>
   c. /var/lib/docker/containers
   d. None of the above
5. Every layer in an image has:
   a. An image id
   b. An image name
   c. An SHA digest

d. An image id and an SHA digest

6. The docker server is a key component of the docker engine and consists of:
   a. Containerd
   b. Dockerd
   c. runC
   d. None of the above
7. Copy-on-Write is a mechanism:
   a. That lets us copy an image
   b. That creates a copy of an image only if the image is being modified.
   c. That creates a copy of an image only if the image is not being modified.
   d. That creates a copy of an image only if the image is being deleted.
8. Layer digests ordered in a particular way to compile the image is known as.:
   a. The configuration object
   b. The SHA digest
   c. The filesystem
   d. None of the above
9. Multi-stage builds allow us to use:
   a. Several RUN commands in the same Dockerfile
   b. Several CMD commands in the same Dockerfile
   c. Several Dockerfiles together
   d. Several FROM commands in the Dockerfile
10. A docker container will show a status of dead if we were not successful:
    a. In running the container

b. In starting up the container
c. In fully cleaning up the container because resources were unavailable or busy at that point.
d. In creating the container cleanly

## Answers

1. c
2. d
3. a
4. b
5. c
6. b
7. b
8. a
9. d
10. c

## Questions

1. Describe your understanding of a container.
2. What are images, and how are they related to containers?
3. What are the various states in which a container may exist?
4. What do you understand by the term immutable in the context of images?
5. What are some of the best practices regarding log management?
6. Describe your understanding of a Dockerfile.
7. What is a configuration object in an image?
8. What is a build cache?
9. What do you understand by Multi-stage builds?
10. What is your understanding of intermediate images?

## Key Terms

- Containers
- Images
- Copy-on-write
- Thin read-write layer
- Container logs
- Inspecting containers
- Immutable
- Image id
- SHA digest
- Configuration object
- Dockerfile
- Build cache
- Multi-stage builds
- Intermediate image

# CHAPTER 3

# Storage Drivers and Volumes

In this chapter, we look into storage drivers and volumes. Typically, we write relatively less amount of data to the container's read-write layer, and if we are saddled with a write-heavy workload, we will prefer using volumes instead. Volumes are discussed later in the chapter. However, there will always be some workloads that need us to write to the writeable layer of the container, and in this context, we will use storage drivers. However, there are a lot of intricacies as far as storage drivers and volumes are concerned, and we must have a good understanding of these as we go ahead.

## Structure

- Docker Storage Drivers
- Supported Storage Drivers
- Backing Filesystem support
- Overlay and Overlay2 Storage Drivers
- Docker Volumes

## Objective

After studying this chapter, you should be able to develop a good understanding of storage drivers and volumes. For example, you ought to be able to make an informed decision on which storage driver to use for your workloads if your kernel supports multiple storage drivers. Similarly, you should become proficient in understanding volumes as the preferred mechanism for data persistence used by Docker containers and the best practices that go alongside it.

## Docker Storage Drivers

For heavy writes, we use docker volumes, which, as mentioned earlier, we will see later in the chapter, but for smaller writes which occur in the read-write layer of the container, the storage drivers come into the picture. Storage drivers are the software that helps us write stuff to the docker container's read-write layer.

Using a pluggable architecture, Docker provides support for several different storage drivers. In fact, the storage driver we use, in a way, does impact how images and containers are stored and managed on our Docker host.

If our kernel supports multiple storage drivers, then we ought to use the driver that has the best overall stability and performance. That said, Docker already provides a list of prioritized storage drivers if we do not have a storage driver configured on our system. However, we need to ensure that the storage driver we choose meets our overall requirements.

## Supported Storage Drivers

Let's list out the storage drivers that are currently supported in Docker:

- **Overlay2**: Overlay2 is the preferred storage driver for all Linux distros. It requires no additional configuration, and we can just use it directly.
- **aufs**: For versions of Docker older than version 18.06 running on Ubuntu, aufs is the preferred storage driver, only in case if we are on *Ubuntu 14.04* on *kernel 3.13* which has no support for overlay2.
- **Devicemapper**: Earlier devicemapper was the preferred storage driver for both *CentOS* and *RHEL*, as their kernel drivers did not support overlay2, but now both have support for overlay2, which is now the de facto recommended driver.

  Devicemapper requires direct-lvm, which entails additional configuration for the creation of a thin pool using block devices. We may use loopback-lvm, as it requires no additional

configuration, but the problem is it not performant at all and cannot be recommended for use in production environments.

- **btrfs and zfs**: The btrfs and zfs storage drivers may be used if the filesystem on which the host is installed (technically known as the backing filesystem) have also got btrfs and zfs installed on them. But once we have these running on our systems, we may use them for sophisticated stuff like creating snapshots, and so on. But it all needs additional configurations to be done.
- **vfs:** This storage driver is not suitable for production setups because it has a very poor performance and does not support the copy-on-write mechanism.

**We see a general move towards the overlay2 storage driver, with the overlay storage driver deprecated in Docker Engine-Enterprise 18.09 and Devicemapper deprecated in Docker Engine 18.09.**

Of course, we can change storage drivers by following certain steps detailed below. However, some drivers may require additional configuration, including configuration on the Docker host (at the physical or logical disk level).

## Backing Filesystem Support

The table below shows us the backing filesystem support that is available for different storage drivers. But before we go any further, let us try to understand what a backing filesystem is. Docker needs a filesystem to store data. Typically, it is under `/var/lib/docker/` This is the backing filesystem.

| Storage Driver | Filesystem Support | Remarks |
|---|---|---|
| overlay, overlay2 | xfs, ext4 | xfs ought to have ftype=1 |
| Aufs | xfs, ext4 | aufs cannot use the backing filesystems like aufs, btrfs, or ecryptfs. |
| Devicemapper | direct-lvm | devicemapper support need to be included in the kernel, and specific |

| | | |
|---|---|---|
| | | configurations are required to make it work with Docker. |
| Btrfs | btrfs | Backing filesystem has to be btrfs |
| Zfs | Zfs | Backing filesystem has to be zfs |
| Vfs | any filesystem | While vfs will work with any filesystem, it is a rarely used storage driver because of its performance issues and the fact that it doesn't support copy-on-write mechanism. |

***Table 3.1***

## Overlay and Overlay2 Storage Drivers

The OverlayFS filesystem is a modern implementation of a union filesystem similar to aufs, but more performant and simpler to implement overall. By default, docker provides two overlay related storage drivers: overlay and overlay2. Overlay had a problem with inode exhaustion, which was taken care of in overlay2. Overlay2 is considered more stable and performant. So, let us focus on overlay2.

Overlay2 is supported both on *Docker Community Edition* and *Docker Enterprise Edition 17.06.02-ee5* and higher. Overlay2 is supported even on an xfs backing filesystem, provided the flag `-n ftype` is set to 1.

Let us start by checking whether the Linux kernel driver OverlayFS is available on our system by running the following command and observing the screenshot in *Figure 3.1*.

```
lsmod | grep overlay
```


***Figure 3.1***

Yes, it is there, so let us go ahead and check our current storage driver. For this, we need to run the following command and check the output in *Figure 3.2*:

```
docker info | grep Storage
```

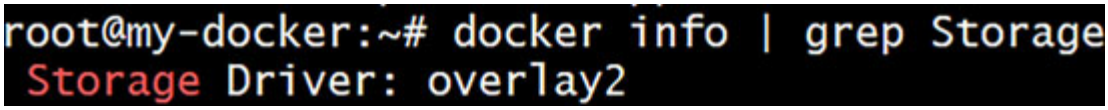



***Figure 3.2***

Since we have checked and found that we are using the overlay2 storage driver, which is the default in most cases, we ought to be fine. But, just for understanding how it works, let us try to change the storage driver from overlay2 to overlay.

**Step 1)** We need to stop docker first.

```
systemctl stop docker
```

**Step 2)** We need to copy the contents of `/var/lib/docker` directory to another location.

```
cp -au /var/lib/docker /var/docker.bkp
```

**Step 3)** We now need to edit a file named `daemon.json` under the `/etc/docker/` directory. If the file is not there, we need to create that file and pass an entry similar to the one below.

```
{
  "storage-driver": "overlay"
}
```

After passing the entry or editing the file, as the case may be, we go ahead and save and close the file.

**Step 4)** We now need to start docker.

```
systemctl start docker
```

**Step 5)** Now if we check for the storage driver by running the command below, we ought to get an output similar to the one shown in *Figure 3.3*.

```
docker info | grep Storage
```

```
root@my-docker:~# docker info | grep Storage
 Storage Driver: overlay
WARNING: No swap limit support
WARNING: the overlay storage-driver is deprecated, and will be removed in a future release.
root@my-docker:~#
```

***Figure 3.3***

There we go. The storage driver is changed to overlay from overlay2. We can similarly change back to overlay2 again. And it is all pretty simple; however, if we were changing into a storage driver like devicemapper or btrfs, it would have meant lots of additional configurations apart from ensuring that all of the prerequisites the drivers require were being met.

## Going deeper into the overlay2 storage driver

The fundamental concept of the *OverlayFS* is to merge directories and/or filesystems in such a way that the filesystem or directory in the lower layer is not written into., but all changes get written into the upper layer. Technically, the *lower layer* is called the lowerdir, and the *upper layer* is known as the upperdir. These layers are merged or unified, and the process is known as a union mount. The unified or merged directory is known as, what else, merged. The overlay2 driver can support upto 128 layers in the lowerdir!

So, let's do a quick example to understand the concept clearly. Let us pull a redis image and see how everything pans out. See the output in *Figure 3.4*.

```
docker image pull redis
```

```
root@my-docker:~# docker image pull redis
Using default tag: latest
latest: Pulling from library/redis
8559a31e96f4: Pull complete
85a6a5c53ff0: Pull complete
b69876b7abed: Pull complete
a72d84b9df6a: Pull complete
5ce7b314b19c: Pull complete
04c4bfb0b023: Pull complete
Digest: sha256:800f2587bf3376cb01e6307afe599ddce9439deafbd4fb8562829da96085c9c5
Status: Downloaded newer image for redis:latest
docker.io/library/redis:latest
root@my-docker:~#
```

***Figure 3.4***

Next, let us go inside the *overlay2* directory and see what is inside. See the output in *Figure 3.5*.

```
cd /var/lib/docker/overlay2
ls -ltr
```


***Figure 3.5***

If we look carefully, we will see that the redis image had six layers, while there are seven directories under `/var/lib/docker/overlay2`. There is a directory named '`l`' (lowercase L), which is an interesting one because if we go inside, we will see that it contains shortened layer identifiers as symbolic links. It is not without reason that this has been done-the shortened identifiers allow us to side-step, hitting the limitation of page size when we use the mount command with arguments.

See the following screenshot to understand the 'l' layer.

```
cd l
ls -ltr
```

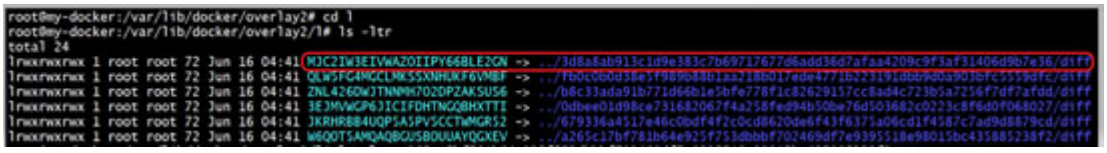



***Figure 3.6***

The lowest layer indicated in red in *Figure 3.6* will contain a file named link, which contains the identifier in short form, and a directory called `diff` which has all of the layer's contents. Check the screenshots in *Figures 3.7*, *Figure 3.8*, and *3.9*.

```
ls -ltr
3d8a8ab913c1d9e383c7b69717677d6add36d7afaa4209c9f3af31406d9b7e3
6
```

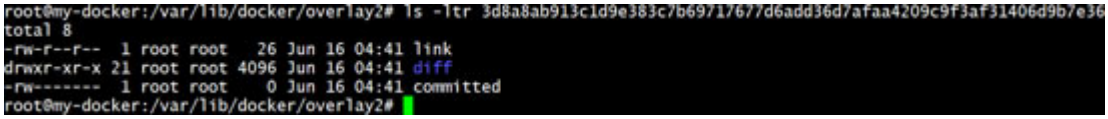



***Figure 3.7***

```
cd
3d8a8ab913c1d9e383c7b69717677d6add36d7afaa4209c9f3af31406d9b7e3
6
ls -ltr
cat link
```

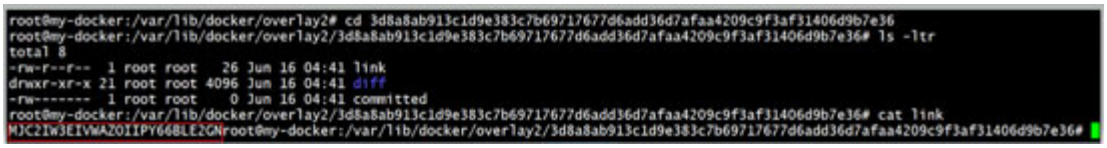



***Figure 3.8***

```
cd diff
ls
```

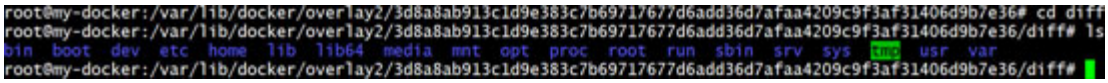



***Figure 3.9***

Layers above the lowest layer will contain a file called lower. The lower layer will denote its parent while the diff directory will contain its contents. There will also be a work directory which is used internally by OverlayFS. Finally, there will a merged layer where the contents are merged.

## Docker Volumes

If we just use the writeable layer of the container to write all our data, then there are several potential issues. The first and foremost one is the data will not persist once the container is gone. Second, because the writeable layer uses a storage driver to provide a union filesystem-this, additional abstraction may be a drag on the performance of the container. Third, as the writeable layer is so tightly coupled with a container, manipulating the data written therein is non-trivial. And finally, containers are supposed to lightweight and

nimble. If all of our writes go into our container, then it will neither remain lightweight, nor nimble.

Volumes, on the other hand, are designed to store data permanently. For write-heavy workloads on a container, volume and its close siblings bind mount, and tmpfs is the preferred mechanism. See the following diagram:

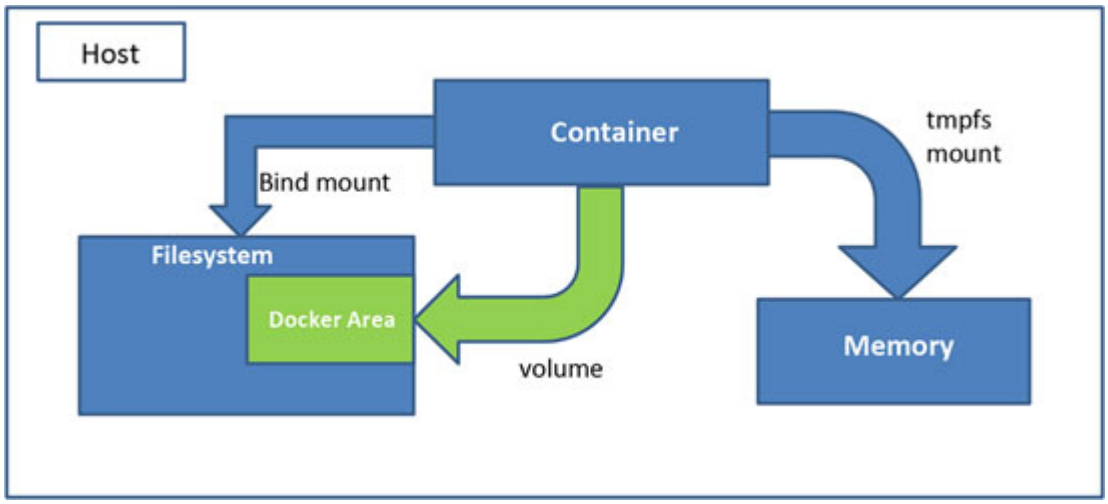



***Figure 3.10***

Now, let us try to understand what a volume is.

Volumes are a means of establishing a connection between the host system and the container. A directory on the docker area of the host system is connected to a directory in the container, thereby enabling the data to be persistent and available. Even if the containers are removed, yet the data will continue persisting. In the case of a bind mount, however, we don't bother with the docker area, we just map a random directory on the host filesystem to a directory in the container. In `tmpfs`, however, the mapping of a directory in the container is in the memory of the host.

With a volume, neither the persistence of data nor it's transfer remain an issue. And without a shadow of doubt, volumes are more performant than a container's writeable layer. Additionally, we can mount a volume for multiple containers simultaneously, and this is especially helpful where the containers need to share data amongst themselves.

Volumes also support the use of volume drivers that allow us to do a lot of the things with the data, such as encrypting it and/or moving it to the cloud or remote servers.

So, before moving ahead, let us summarize what we understood about volumes.

Volumes are nothing but the physical filesystem on the host being plugged into the virtual filesystem of docker, such that any changes/modifications made at the container level gets written into the host system. And it would be the other way round as well, any changes made on the host level will be reflected at the container level.

Let us do a few examples to understand the concept better.

First, let's see an example using a bind mount. To start with, let us create on our host system a directory named testing under the root directory. Since this is a bind mount, we don't have anything to do with the *docker area*. Then we will attach this newly created directory to a directory named `test1` inside the container. The `test1` directory will be created on the fly as we create the container.

On the host system:

```
mkdir -p /root/testing
ls
```

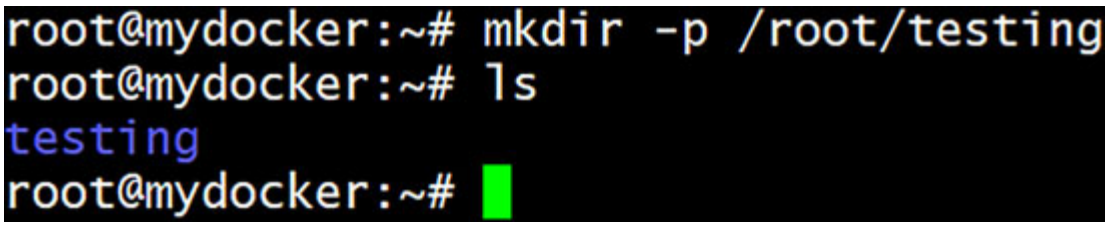



***Figure 3.11***

Next, let us create the container.

```
docker run -it --name myalpine -v /root/testing/:/test1
alpine:latest
ls -ltr
```

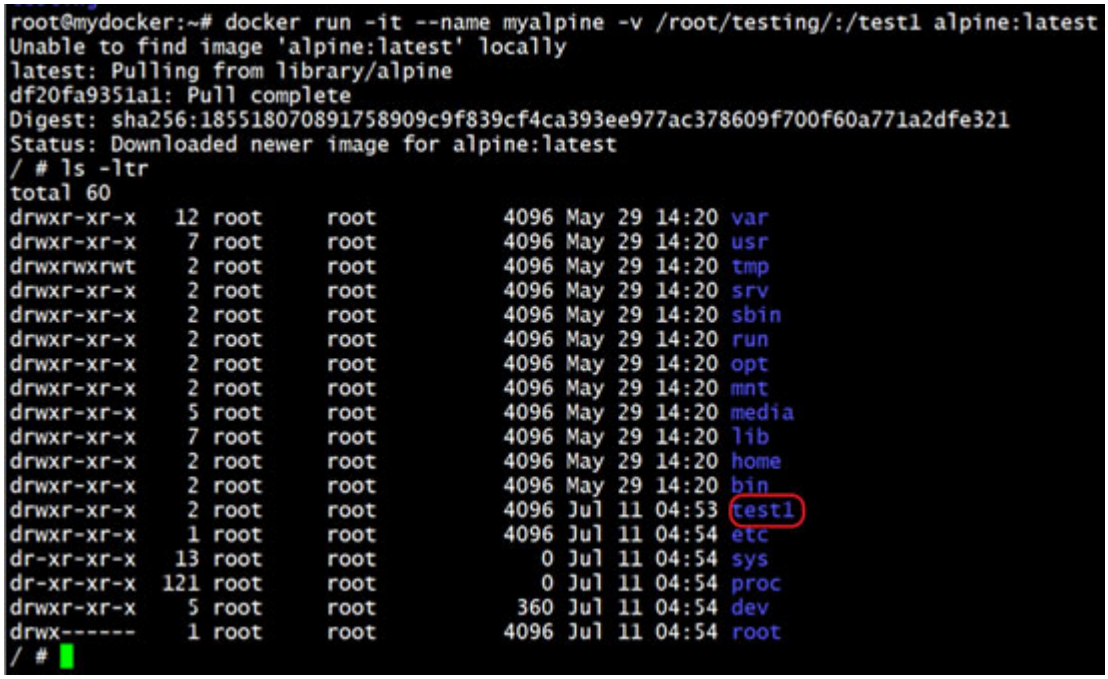



***Figure 3.12***

The command creates a container named myalpine in interactive mode (`-i`) with a terminal attached (`-t`). We don't create the container in detached (`-d`) mode, because we don't want it to get started in the background, as we would like to have it run in the foreground as we would like to get inside the container directly. Additionally, we use the `-v` (or `--volume`) flag to attach the directory `/root/testing/` on the host system to the directory named `test1` in the container, which as we noted earlier, will be created on the fly as we create the container. See the screenshot in *Figure 3.12*.

So, expectedly the directory named `test1` is there in the container. Now, we will create a file inside this directory and write something in the file. We expect the write to be mapped to the directory on the host system. Then we will drop the container and note the persistence of the data.

```
cd test1
vi myfile
```

We save and close the file after putting in a line like, " *This is a demo to check for data persistence in a container.*"

When we check the file inside the container, we should get something similar to the following screenshot:

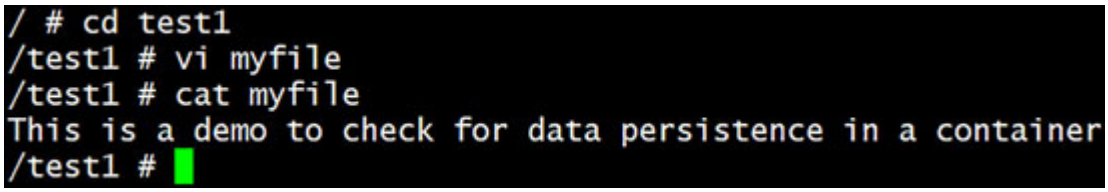



***Figure 3.13***

Now, let us see whether the write in the container has been propagated to the host system in `/root/testing/` directory.

```
exit
cd /root/testing/
ls -ltr
cat myfile
```

And sure enough, we have the write on the host system as well. See the following screenshot:

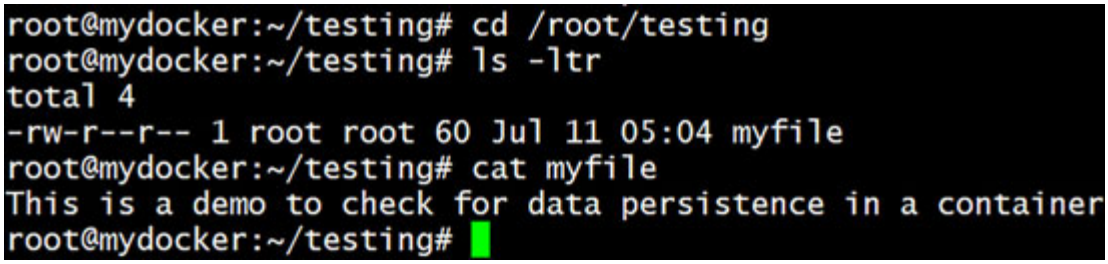



***Figure 3.14***

Before we move ahead, let us check the details of how the directory on the host system, technically known as the source (`/root/testing`), is mapped to a directory in the container technically known as the destination (`/test1`). We can also discern from the screenshot below that we are using a bind mount by the fact that the "*Type*" shows "*bind*".

```
cd ~
docker container ls -a
docker inspect -f '{{json .Mounts}}' 07484e3aaff7
```


***Figure 3.15***

The next thing for us to check is whether after removing the container, we can still access the data or not.

```
docker ps
docker container rm myalpine -f
cd /root/testing
ls -ltr
cat myfile
```

See the following screenshot:

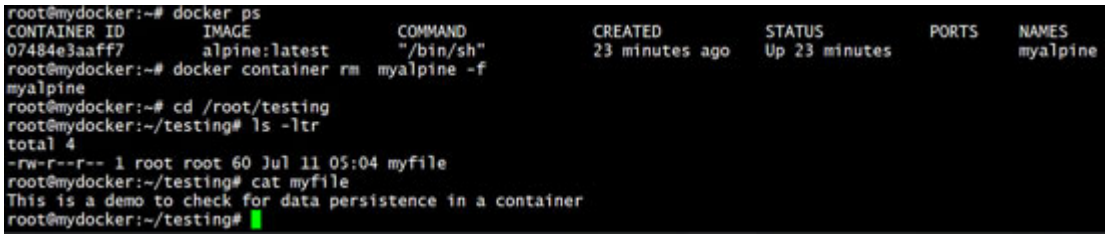



***Figure 3.16***

And there we are! The data is still there, and it clearly shows us the persistence of data.

Now, let us take an example where we use the area on the host system specified as the *docker area*. In this example we will just specify the destination directory, that is, the directory in the container and let docker in the docker area handle the source part. This is technically known as creating an anonymous volume.

```
docker run -dit --name test_vol --volume /myapp alpine:latest
docker container ls -a
```

`apt update && apt install jq -y` (This command is required to install `jq`, which is a tool used as command-line JSON processor to format the JSON outputs nicely. In my case, I have already installed `jq` on the system, so I need not run this command again.)

```
docker inspect -f '{{json .Mounts}}' 8dc17e8c885d | jq
```

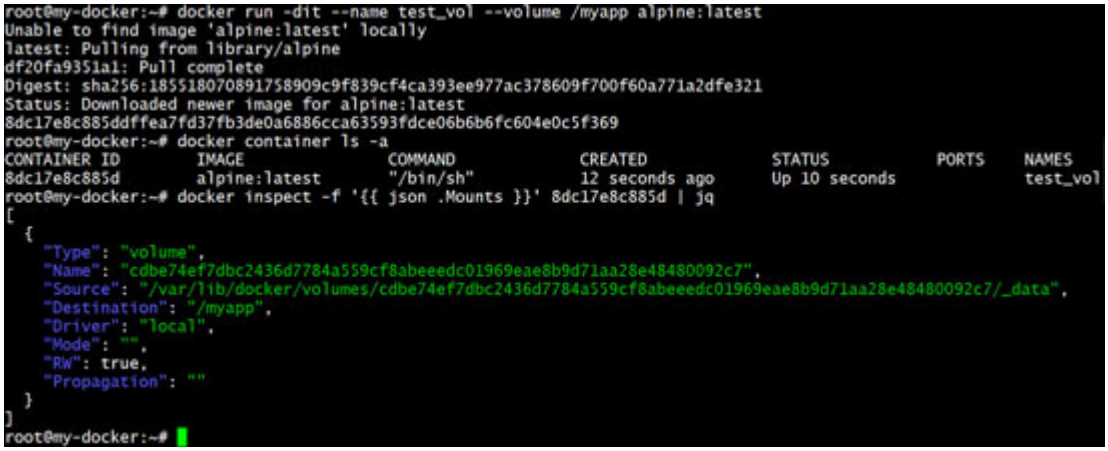



***Figure 3.17***

If we look carefully, we will see that in *Figure 3.17*, the type is volume, and the source is a directory which we can guess would be in the area designated as *docker area* (Refer *Figure 3.10*). The destination in the container is the directory we mentioned while creating the container.

So, now let us go ahead and test the persistence of the data, as we did earlier. We will create a file with something written in it inside the container, and then check whether the file persists when we remove the container.

```
docker exec -it test_vol sh
cd myapp
vi test_anonymous_vol
```

We put in a message like “We are testing data persistence in an anonymous volume.” and then save and close the file.

```
cat test_anonymous_vol
exit
cd
/var/lib/docker/volumes/cdbe74ef7dbc2436d7784a559cf8abeeedc0196
9eae8b9d71aa28e48480092c7/_data
ls
cat test_anonymous_vol
cd ~
docker container rm 8dc17e8c885d -f
```

```
cd
/var/lib/docker/volumes/cdbe74ef7dbc2436d7784a559cf8abeeedc0196
9eae8b9d71aa28e48480092c7/_data
ls
cat test_anonymous_vol
```

The output is shown in the following screenshot:

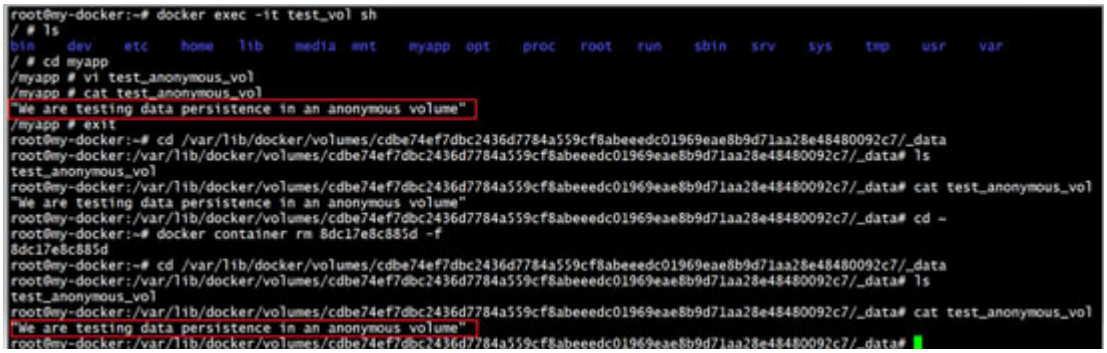



***Figure 3.18***

While this works fine, there is a better way of doing things. We can use a named volume, which we can create independent of a container. The main advantage of a named volume over an anonymous one is that we don't have to muck around with long directory names, and it is much easier to handle, especially in production environments where we may be dealing with hundreds of containers requiring stringent control through proper naming conventions and other best practices.

Okay, let's begin by creating a volume.

```
docker volume create myvolume
```

See the output in the following screenshot:

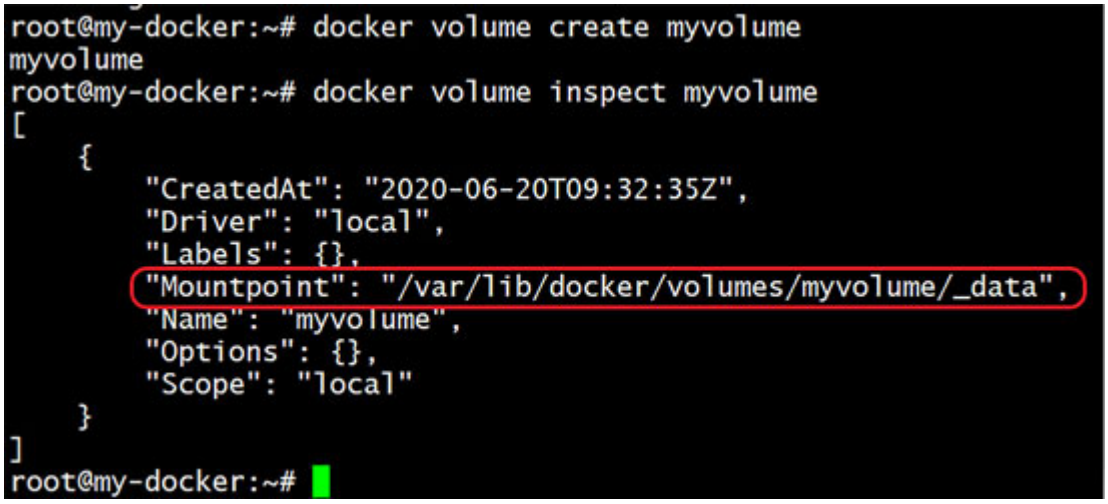



***Figure 3.19***

And as we can see, the volume `myvolume` points to the directory `/var/lib/docker/volumes/myvolume/_data`. This would be our source directory when we map it to a container. So, let's go ahead and do it.

```
docker container run -dit --name mycentos --volume
myvolume:/test centos:latest
```

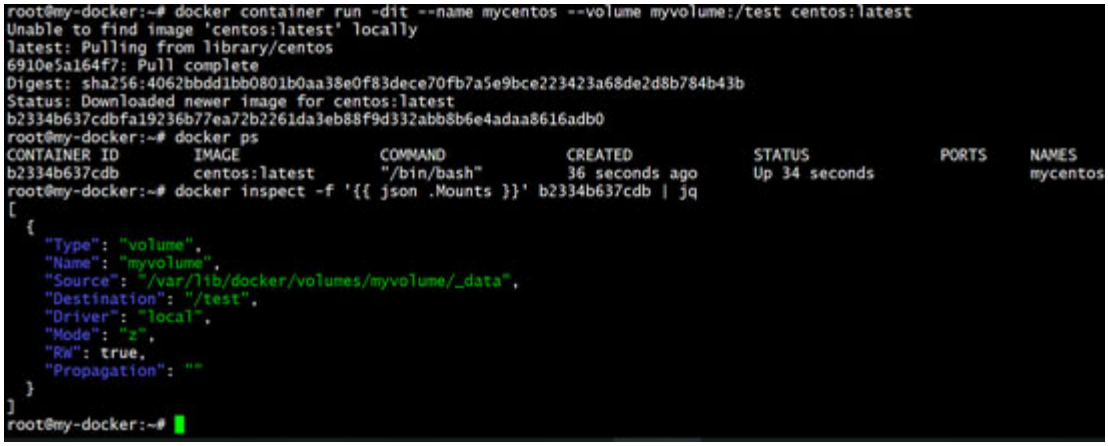



***Figure 3.20***

So, it should be apparent that we used the volume that we created earlier to map it to the container named `mycentos`. The interesting thing is that not only will the data persist after the container has been removed, but we can map more than one container to the same volume. This is especially helpful where data needs to be shared amongst containers.

Before we move ahead, there is one thing that we should note. The `-v` or `--volume` flag was used only for swarm services prior to *Docker 17.06*, but from version 17.06, the `--mount` can be also be used for stand-alone containers as well. In fact `--mount` is simpler, easier to understand, and more verbose and, in general, ought to be our preferred syntax.

So, let us do a quick example using the `--mount` syntax.

This time we are not going to create a volume beforehand, but let docker create the volume on the fly for us. This is also a facility provided to us by docker.

Let us run the commands below and see the output in the screenshot in *Figure 3.21*.

```
docker run -dit --name mynginx --mount
source=myvol2,target=/test nginx:latest
docker ps
docker inspect -f '{{json .Mounts}}' af28f181a46d | jq
```

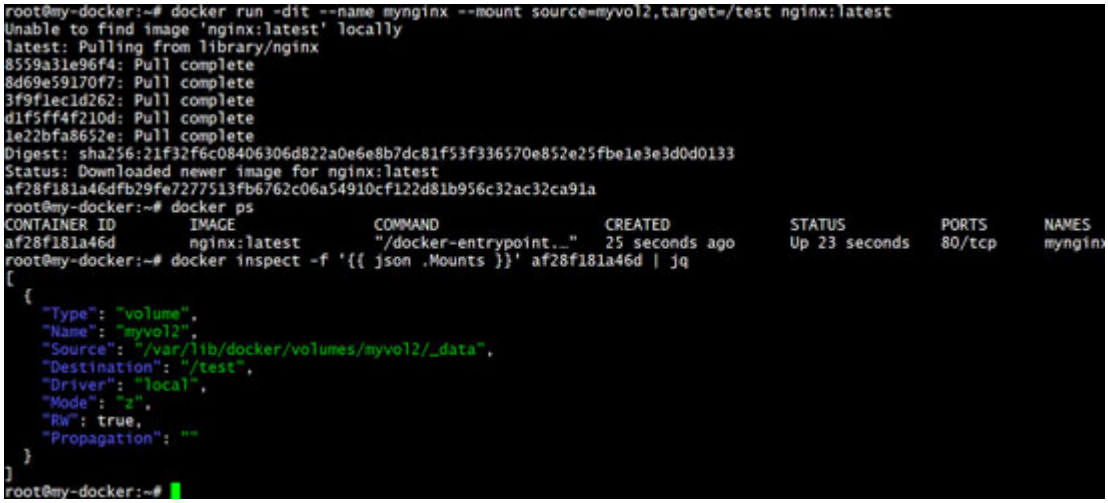



***Figure 3.21***

There is another mechanism to write files outside of the container's read-write layer, and that is known as `tmpfs`. These files are created in the memory of the host system instead of the writeable layer of the container. However, unlike bind mounts and volumes, the `tmps` mount is temporary and will persist only as long as the container is running. In other words, as soon as the container is stopped, the

mount gets removed. Therefore, there is no persistence of files in this case.

Let us take a look at an example of `tmps` usage by running the commands below and checking the output in *Figure 3.22*.

```
docker run -it --name test_tmpfs --mount
type=tmpfs,destination=/testing\ centos:latest
ls
```

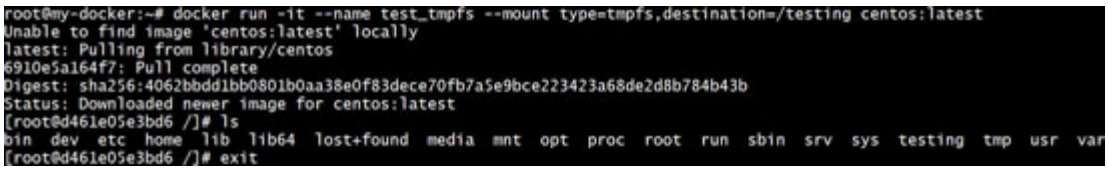



***Figure 3.22***

We do need to mention the type explicitly as `tmpfs`, but the rest of the command should look familiar to us by now. So, there we are- `tmpfs` provides us with a temporary location if we need to write outside of the container's read-write layer. `tmpfs` may be considered an option, albeit with limited utility.

# Conclusion

In this chapter, we had a look at the container's read-write layer and how specialized software is known as storage drivers' aid in such writing. We learned in brief about the different storage drivers available in docker and had an in-depth look into the storage driver recommended by docker, that is, overlay2. We also understood the concept of persistence of data and how volumes and bind mounts help in persisting data even after the container is removed. We also understood about the performance implications of volumes vis-à-vis the container's read-write layer and how volumes ought to be an integral part of our production setup. Overall, after this chapter, we ought to have gained a good understanding of how storage is managed in containers. In the next chapter, we get into another building block of docker-docker networking. The next chapter will build up our conceptual understanding of how docker networking works from the ground up.

## Points to Remember

- All writes are made in the thin read-write layer, which comes into existence when a container is created.
- Typically, relatively less amount of data is supposed to be written into the container's read-write layer because it is not designed for write-intensive workloads.
- Storage drivers are specialized software designed to help write to the container's read-write layer.
- If our kernel supports multiple storage drivers, then we ought to use the driver that has the best overall stability and performance.
- Docker already provides a list of prioritized storage drivers if we do not have a storage driver configured on our system. However, in general, we would be pretty well served if we use the overlay2 storage driver.
- The overlay2 storage driver merges directories and/or filesystems in such a way that the filesystem or directory in the lower layer is not written into. Still, all changes get written into the upper layer(s). Technically, the lower layer is called the `lowerdir`, and the upper layer is known as the `upperdir`. These layers are merged or unified, and the process is known as a **union mount**.
- Only the '*diff*' is written into the upper layers so that we can continue to keep the image size as small as possible.
- Volumes and Bind Mounts are used to persist data on the system because one of the characteristics of a container is that it can be ephemeral in nature-that is another container can replace this container at any point in time.. We would want this to happen without impacting the data stored on the host system.
- `tmpfs` allow us to have writes outside the container's read-write layer in the host's memory.
- `tmpfs` does not allow us to persist data. When the container stops, the `tmpfs` mount is removed, and files written there are no longer available.

## Multiple Choice Questions

**Choose the most appropriate answer:**

1. The read-write layers of containers are used for:
    a. Write intensive workload
    b. Read intensive workloads
    c. Small writes
    d. None of the above
2. The Storage driver is a piece of software that facilitates:
    a. Creation of an image
    b. Creation of a container
    c. Writes to the read-only layer of the container
    d. Writes to the read-write layer of the container
3. The tmps mount:
    a. Stores information inside the container
    b. Stores information in a volume
    c. Stores information in the host system's memory
    d. Stores information in the host system's storage
4. In the case of a Bind Mount:
    a. A file or directory on the host machine is mounted into a container
    b. A file or directory from the 'docker area' is mounted into a container
    c. A file or directory from the host machine is mapped to a container's memory
    d. There is no mapping between a host machine's filesystem and the container
5. Volumes support the use of volume drivers that:
    a. Allow us to store data on the host machine's disk

b. Allows us to store data in the container's filesystem
c. Allows us to encrypt data and port it to other locations.
d. Allow us to compress data

6. The command `docker run -it --name myalpine -v /root/test/:/test\ alpine:latest`
   a. Creates a container and mounts the /root/test/ directory on the container to /test directory on the host machine
   b. Creates a container and mounts the /root/test/ directory on the host machine to the /test directory on the container
   c. Does not create a container but mounts the /root/test/ directory on the host machine to the /test directory on the container
   d. Does not create a container but mounts the /root/test/ directory on the container to /test directory on the host machine
7. The information written in the read-write layer of the container:
   a. Is not lost when the container is stopped and started
   b. Is lost when the container is stopped and started
   c. Some of the information may be lost
   d. None of the above
8. The btrfs and zfs storage drivers may be used if:
   a. The backing filesystem supports btrfs and zfs
   b. The backing filesystem supports direct-lvm
   c. The backing filesystem supports loopback-lvm
   d. The backing filesystem also has btrfs and zfs installed on it
9. The storage driver overlay2 is preferred over overlay is because:
   a. overlay2 is definitely more performant than overlay
   b. RHEL and CentOS now support overlay2
   c. overlay2 does not support from inode exhaustion

d. overlay2 is easier to install

10. The storage driver overlay2 is preferred over overlay is because:

    a. overlay2 is definitely more performant than overlay
    b. RHEL and CentOS now support overlay2
    c. overlay2 does not support from inode exhaustion
    d. overlay2 is easier to install

## Answers

1. c
2. d
3. c
4. a
5. c
6. b
7. a
8. d
9. c
10. c

## Questions

1. What do you understand by storage drivers?
2. Describe briefly the different types of storage drivers available in docker.
3. On which Linux filesystem is the overlay group of storage drivers based?
4. Explain the concept of '*diff*' from the perspective of image layers.
5. Explain how the overlay2 storage driver works.
6. What do you understand by a volume?

7. What do you understand by a bind mount?
8. Explain the difference between a volume and a bind mount.
9. What would be a good use case for sharing volumes between containers?
10. Explain the concept of tmpfs.

## Key Terms

- Storage Drivers
- Read-Write Layer
- Overlay and Overlay2
- Aufs
- Layered filesystem
- Backing filesystem
- Device Mapper
- Union Mount
- Lowerdir, upperdir
- Merged
- Data Persistence
- Storage
- Volumes
- Volume driver
- Bind mount
- tmpfs

# CHAPTER 4

# The Container Network Model and the Docker Bridge

This chapter covers the fundamentals of docker networking from the perspective of understanding the implementation of the network in real-life docker deployments. This chapter will give you a good grounding of the fundamentals on which the docker network stack is based and the spin that docker gives to it to make it a robust and performant infrastructure.

## Structure

- The Container Network Model (CNM)
- The CNM Driver Interfaces
- Libnetwork
- Docker Drivers
- The Docker Bridge Network
- Conclusion
- Points to remember
- Multiple-choice questions
- Questions
- Key Terms

## Objective

In this chapter, we will look into the theory behind Docker networking. Docker networking is thoroughly application-driven. While on the one hand, it provides all the bells and whistles for the network to work smoothly; on the other hand, it provides just the right level of abstraction for application developers.

## The Container Network Model

Docker networking is built on an architecture called the **Container Network Model** (**CNM**). The CNM is an architecture that allows us to provide a networking model that is simple and efficient, while at the same time ensuring that the applications running in the containers are safe, secure, and portable. Let us have a look at the CNM model.

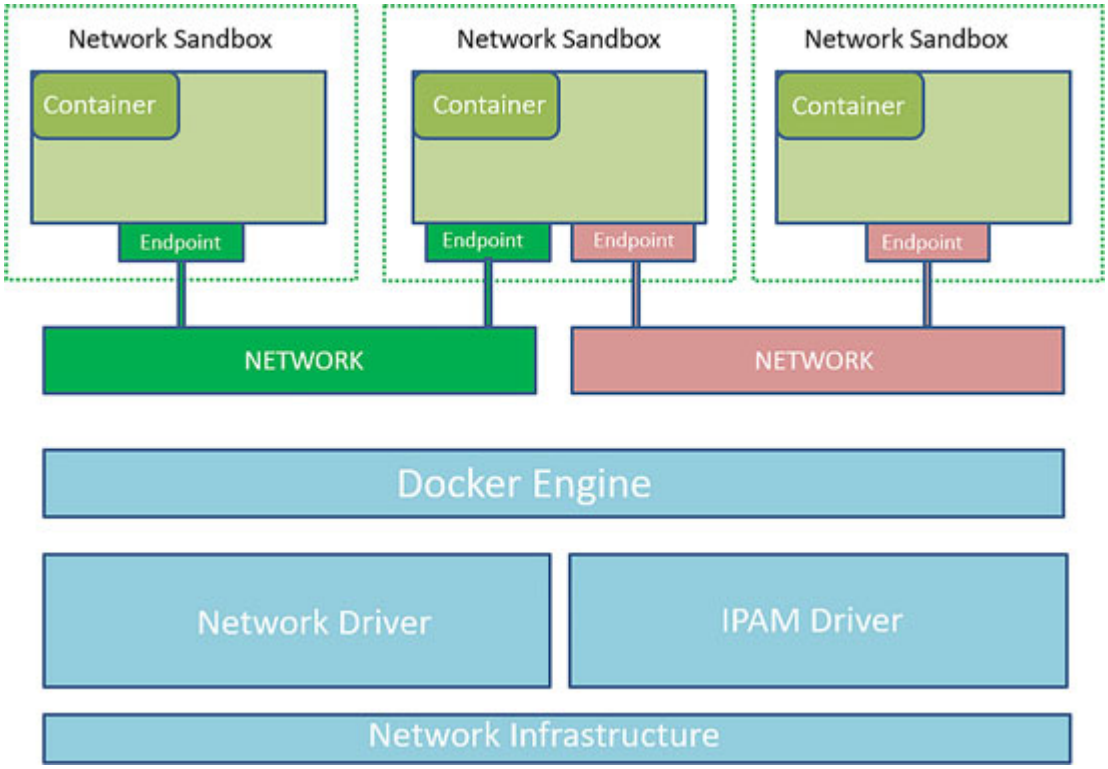



***Figure 4.1***

The CNM typically have several constructs and let us have a quick look at those.

- **Sandbox**: The Sandbox is a construct that contains a full configuration of the container's network stack. This means that it contains interfaces, routing tables, DNS settings, a. etc. However, the Sandbox is completely insulated from the outside world save for endpoints. Endpoints are described below.
- **Endpoints**: The Endpoints join the Sandbox to the outside world via networks. We can think of endpoints as a gateway to the outside world, which in this case, are networks.

- **Network**: The network here could be a complex network created by the user, or a Linux bridge or a VLAN. The important thing to note here is that endpoints connected to the same network will have connectivity amongst themselves. In other words, endpoints connected to the same network can communicate with each other as well.

## The CNM Driver Interfaces

We can think of drivers as pieces of software that actually make the Docker network work. The beauty of these drivers is that they follow the '*plug and play*' model; that is, we can plug in different drivers based on our requirements and start using the network. However, the docker network can be instantiated only through a single driver.

While Docker comes with a rich set of native drivers, it also supports third-party or remote network drivers as well.

We have another class of drivers as well, known as**IP Address Management** (**IPAM**) drivers, as shown in *Figure 4.1*. These drivers provide default subnets or IP addresses where these are not provided or assigned externally.

## The Libnetwork

Libnetwork may be considered as the implementation of the Container Network Model. It is a library for creating and managing network stacks for containers as well as an attempt by Docker to standardize interfaces to connect the Docker daemon to the network drivers.

## Docker Drivers

Docker Drivers may be classified as:

- Docker Native Network Drivers
- Docker Remote Network Drivers

Let's first discuss the Docker Native Drivers.

The Docker native network drivers are built-in drivers and are a part of the Docker Engine setup. We just use them using standard Docker commands. No extra configurations are required.

The following table shows the driver name along with a brief description:

| Name | Description |
|---|---|
| Bridge | A Docker bridge is created on the host using the Linux bridge. Containers connected by a Docker bridge can talk to each other. |
| Host | The container does not have a separate network namespace and just uses the network of the host on which it is residing. |
| None | The none driver provides a network stack of sorts inside the container but doesn't provide an interface to connect to the outside world. So, effectively the container is completely isolated. |
| Overlay | The Overlay network creates a network that can connect multi-hosts without any additional configurations. It makes use of the traditional Linux Bridge as well as VXLAN to enable containers to communicate over the network infrastructure. |
| MACVLAN | The MACVLAN driver uses the MACVLAN bridge in Linux to establish a connection between interfaces. |

***Table 4.1***

**Docker Remote Network Drivers**: These are third party drivers that are compatible with CNM and can be added to the Docker host as a plugin. *Contiv* and *Weave* are two of the more popular ones. The following diagram should clarify how everything is stacked up.

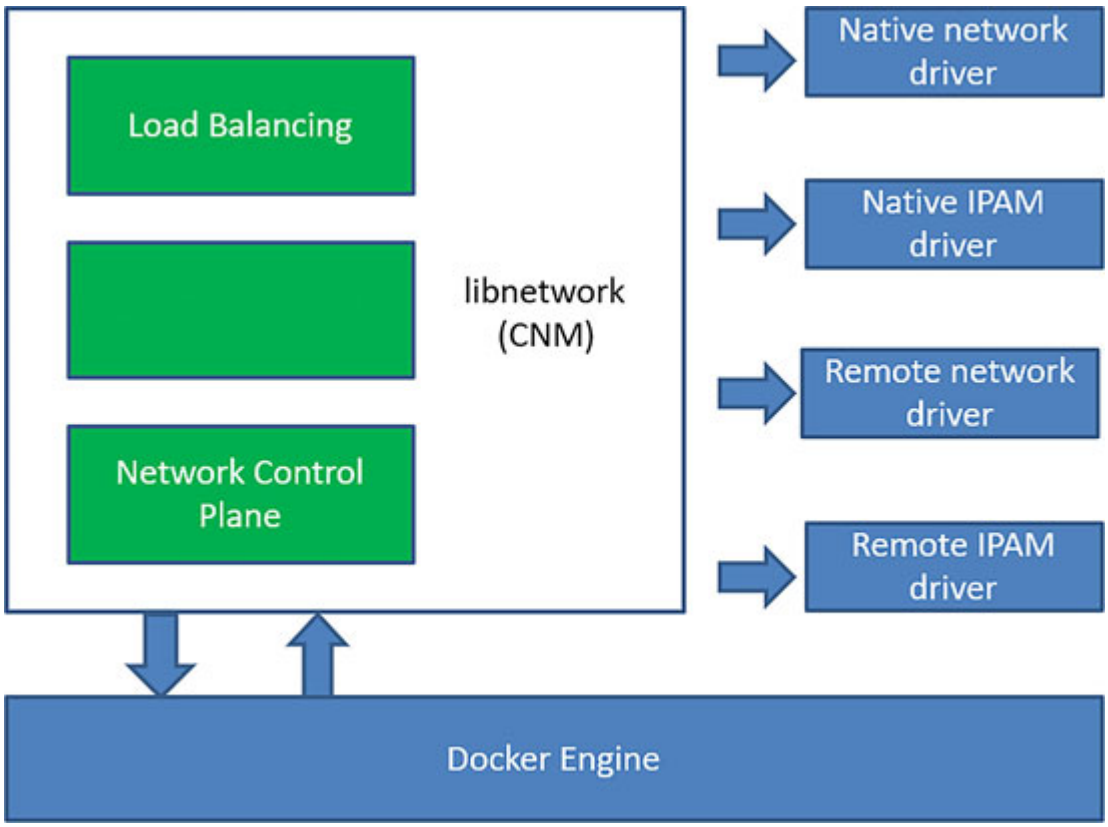



***Figure 4.2***

## The Docker Bridge Network

If we simply talk in network terms, then we can conceptualize a bridge network as a link layer device which is used as a mechanism to transmit traffic between network segments. When docker is installed, rules are automatically put in place so that the containers on a bridged network cannot communicate directly to each other but need to go through the bridge for inter-container communication.

## The Concept of Linux Namespaces

Docker draws heavily from Linux in setting up its network constructs. For example, docker liberally uses network namespaces, Linux bridges, and iptables in creating its network topology. The docker network driver bridge is a somewhat higher-level implementation of the Linux bridge. Put simply; a Linux bridge is Layer 2 device. We

can think of it as a virtual implementation of a physical switch inside the Linux kernel.

Containers provide network isolation based on namespaces. In a single-user computer, we are fine not having to bother overmuch about isolation. Still, where we are running multiple services, it is essential for security and stability that the services are as isolated from each other as possible. And containers use the concept of the namespace to provide that isolation.

A network namespace may be defined as a self-contained unit of network stack having all the components that go into a network stack, like its own routes, firewall rules, and network devices.

Before we delve deeper into container namespaces and the bridge network, let us try to understand the concept of a Linux network namespace.

As we know, network namespaces are self-contained units of the network stack. To start with, let us check whether we have any network namespaces defined on our Linux system by running the following command and checking the screenshot in *Figure 4.2*.

```
ip netns list
```

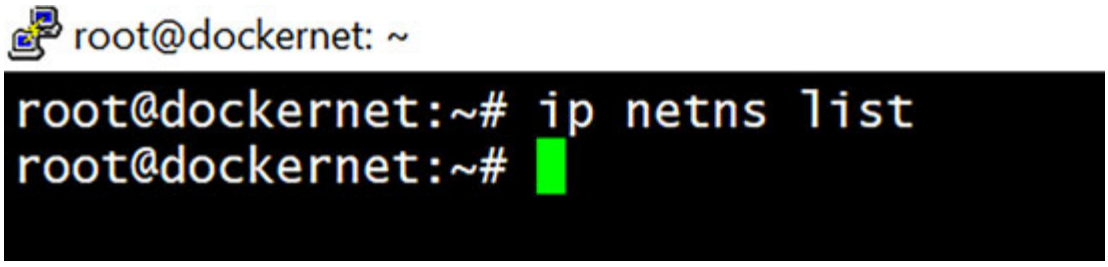



***Figure 4.3***

We see that we don't have any network namespaces defined. Typically, we shouldn't expect to find network namespaces defined.

So, let us create a network namespace. It is pretty straightforward. Let us name the namespace `mynetspace`. Let us run the following commands in sequence and check the screenshot in *Figure 4.3*:

```
ip netns add mynetspace
ip netns add mynetspace1
```

```
ip netns list
```

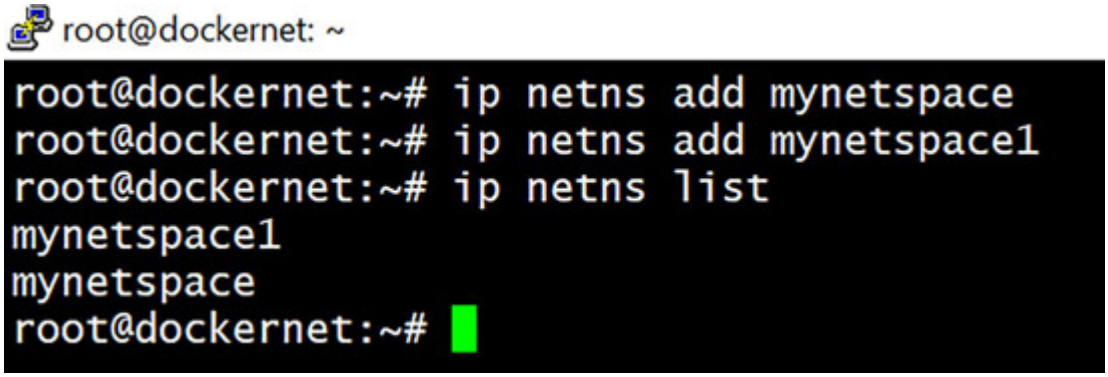



***Figure 4.4***

So, the network namespaces have been created. Next, let us create a veth pair. Now, what is a **virtual ethernet device pair** (**veth**)?

Think of veth as a connecting strand between two namespaces as shown in the diagram:

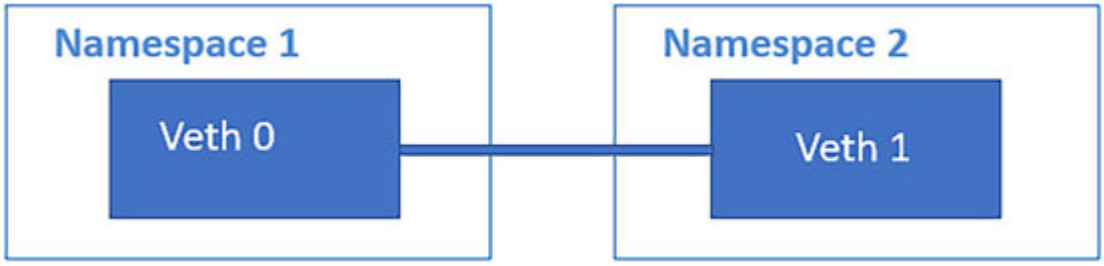



***Figure 4.5***

We often speak about veths in terms of veth pairs. The '*pair'* word comes into the picture because it is a full-duplex link and has an interface in each of the *namespaces*. So, traffic is routed through the veth pair from one interface to the other.

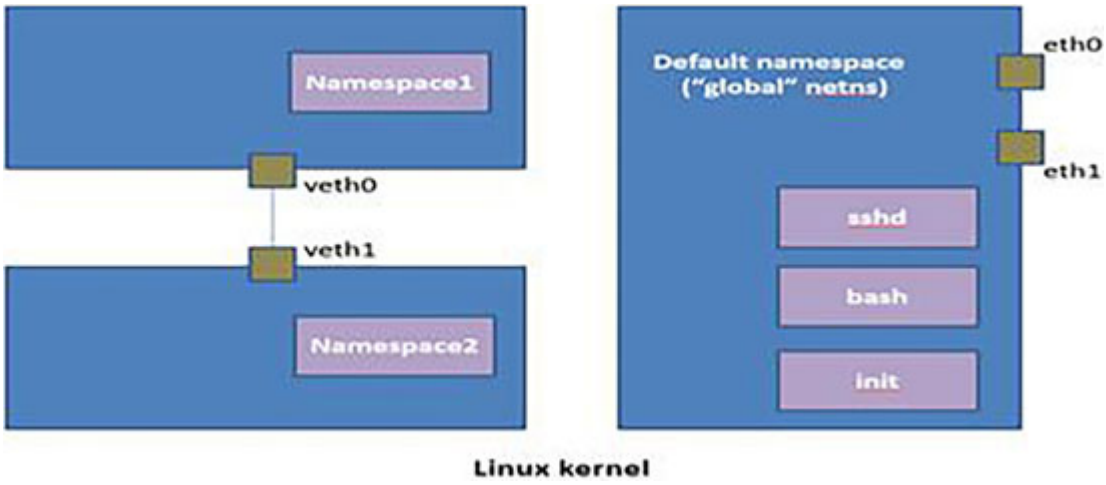



***Figure 4.6***

Let us create one veth pair now, by running the following set of commands:

```
ip link add veth0 type veth peer name veth1
ip link list | grep veth
```

The output is as follows:

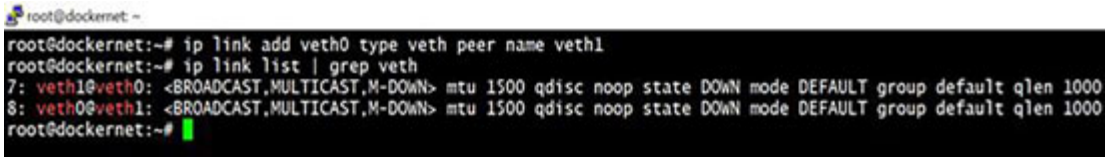



***Figure 4.7***

So, there we go. The veth pair has been created, and we can see it. However, the veth pair has just been created globally and is yet to be attached to the network namepaces that we created. The next step is to attach one end to each of our veth pairs to the namespaces we created. See the commands and the following associated screenshot:

```
ip link set veth0 netns mynetspace
ip link set veth1 netns mynetspace1
```

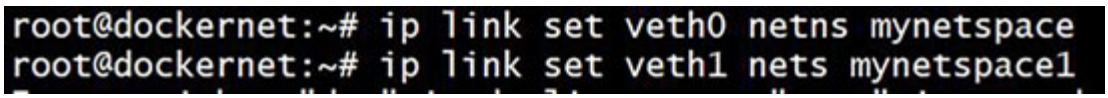



***Figure 4.8***

Next, we `grep` for `veth`, as shown below:

```
ip link | grep veth
```

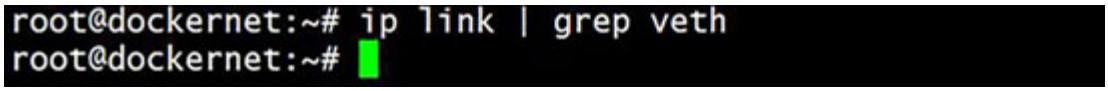



***Figure 4.9***

The interesting thing to note here is that grepping shows us nothing because the veth has been moved from the host namespace to the namespaces we created. We can, of course, see the veth pair going individually inside of our namespaces by running the following set of commands and checking the screenshot in *Figure 4.8*:

```
ip netns exec mynetspace ip link | grep veth
ip netns exec mynetspace1 ip link | grep veth
```

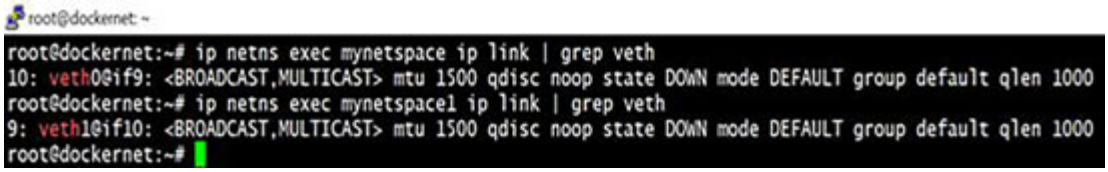



***Figure 4.10***

Now that we have plumbed one end each of the veth interfaces on our namespaces we created to let us add IP addresses and bring them up, or in other words, make our namespaces operational by running the following set of commands and checking the output in *Figure 4.11*:

```
ip netns exec mynetspace ip addr add 10.0.0.15/24 dev veth0
ip netns exec mynetspace1 ip addr add 10.0.0.16/24 dev veth1
ip netns exec mynetspace ip link set veth0 up
ip netns exec mynetspace1 ip link set veth1 up
```

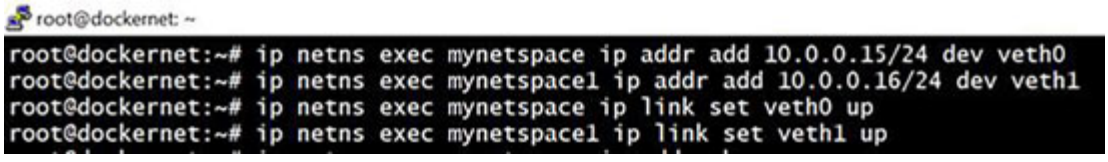



***Figure 4.11***

So, now we have a fully operational network namespaces with a complete network stack, and its IP address. Let's take a look at them. We run the following commands and check the associated screenshots:

```
ip netns exec mynetspace ip addr show
ip netns exec mynetspace1 ip addr show
```


***Figure 4.12***

To do a quick check on whether the interfaces are up and running, we can try to ping one namespace from the other. Let us run the following commands and check the screenshots in *Figures 4.13* and *4.14*:

```
ip netns exec mynetspace ping 10.0.0.16
```

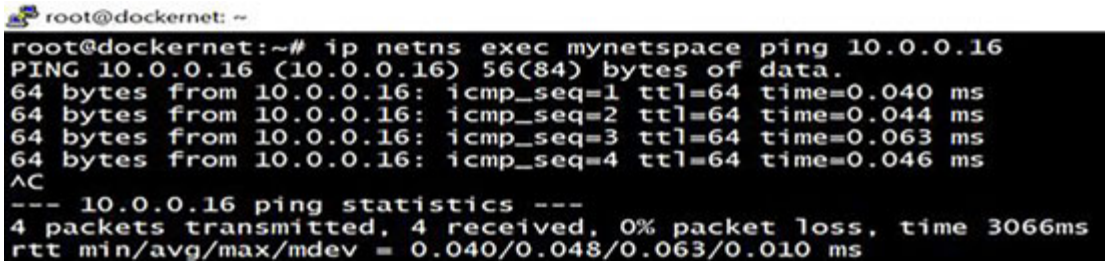



***Figure 4.13***

Pinging the other server and checking the screenshot in *Figure 4.14*.

```
ip netns exec mynetspace1 ping 10.0.0.15
```

```
root@dockernet:~# ip netns exec mynetspace1 ping 10.0.0.15
PING 10.0.0.15 (10.0.0.15) 56(84) bytes of data.
64 bytes from 10.0.0.15: icmp_seq=1 ttl=64 time=0.047 ms
64 bytes from 10.0.0.15: icmp_seq=2 ttl=64 time=0.060 ms
64 bytes from 10.0.0.15: icmp_seq=3 ttl=64 time=0.046 ms
64 bytes from 10.0.0.15: icmp_seq=4 ttl=64 time=0.043 ms
^C
--- 10.0.0.15 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3049ms
rtt min/avg/max/mdev = 0.043/0.049/0.060/0.006 ms
root@dockernet:~#
```

***Figure 4.14***

Everything works perfectly, and this seems a very nice way of implementing network isolation, except for the fact that this is not a very scalable setup. If we think a little deeper, we will realize that every time we create namespaces and connect them, we need to create veth pairs, and creating, managing, and maintaining veth pairs are not exactly trivial work. Lots of namespaces would mean lots of stand-alone networks, and assuming each network would need to communicate with the other, we would just be overwhelmed with the amount of veth pair connectivity we need to create and keep track of.it.

This is where the tried and trusted Linux bridge comes to the rescue. Instead of creating individual pipes (veths) amongst all namespaces, we just connect our namespaces to the Linux bridge using a veth pair, and that's it. We are good to go. Each namespace connected to the Linux bridge will be able to communicate with the other. See the following diagram:

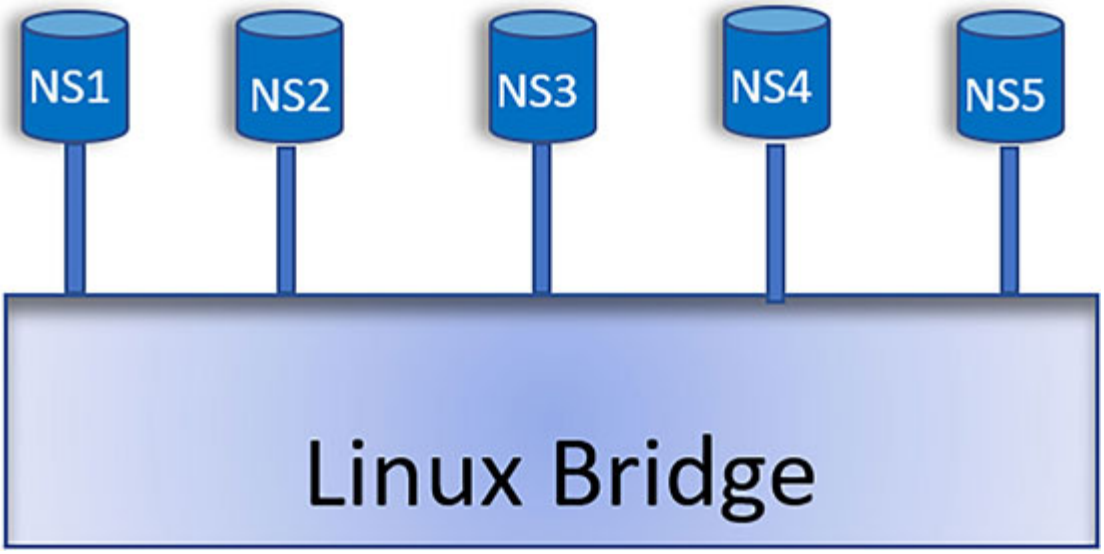


***Figure 4.15***

From the diagram, it is clear how docker containers communicate with each other using the bridge network. A Linux host running a Docker Engine automatically creates a bridge network. This bridge network adapts a Linux bridge for its use and has a default name of docker0.

For stand-alone networks, by default, docker will use a bridge network. Inside the container, one end of the veth pair is hooked and is given the name eth0, while the other end is plugged into the *docker0* bridge and has a name starting with veth. For the container, if we do not specify any external IP address, it will be given an address by the Docker native IPAM driver. However, before we proceed, let us clean up the namespaces we created, by running the following commands:

```
ip netns del mynetspace
ip netns del mynetspace1
```

## The Docker Bridge

Let's first check the interfaces available on our Linux system on which we have installed Docker by running the following and checking the associated screenshot:

```
ip addr
```

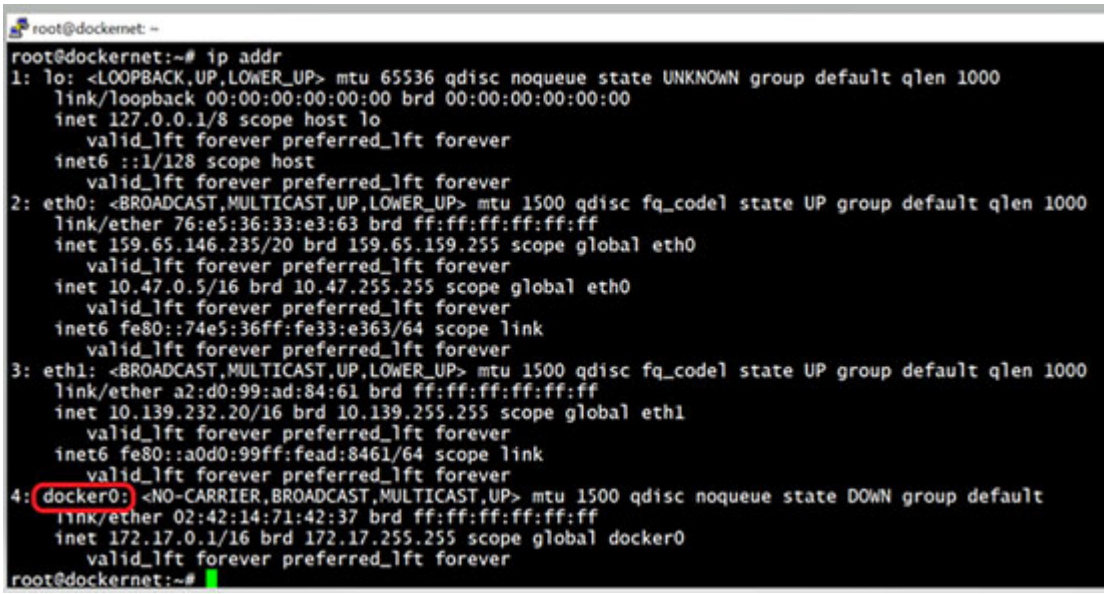



***Figure 4.16***

Indeed, we see that the *docker0* interface is already there as part of the docker setup.

## Conclusion

In this chapter, we had a look at the network model that forms the bulwark of Docker networking. Additionally, we understood the concept of drivers and also had a detailed look into network namespaces. Besides network namespaces, we also focused on understanding the Linux bridge, and how the *docker0* bridge leverages the Linux bridge and provides us the functionality of container to container communication by having one end of the VETH pair hooked into the docker0 bridge. In contrast, the other end is plugged into individual containers.

In the next chapter, we look into the practical implementation of networking in dockers.

## Points to Remember

- The Container Network Model is the basis for the Docker networking architecture.

- The Libnetwork is simply an implementation of the Container Network Model.
- Docker Drivers are pieces of software that make the Docker network work.
- Network namespaces provide the isolation that allows containers to function independently without encroaching on each other's network territory.
- The Linux Bridge is the foundation on which *docker0* is built.
- VETH pair are connecting strands between two or between a container and the *docker0* bridge.
- Containers connected to a docker0 bridge can communicate amongst themselves.

## Multiple Choice Questions

**Choose the most appropriate answer:**

1. The Linux Bridge is a:
   a. Layer 1 device
   b. Layer 3 device
   c. Layer 2 device
   d. The Linux Bridge doesn't belong to the Open Systems Interconnection (OSI) model
2. Containers provide network isolation based on:
   a. The Container Network Model
   b. The VETH pair
   c. The network namespace
   d. The network drivers
3. When we have a Linux host running a Docker engine, we automatically:
   a. Create a Bridge Network
   b. Create a VETH pair

c. Create network namespaces
d. Install all the native network drivers

4. A Network Sandbox is completely isolated save for:
   a. Being connected to a network stack
   b. Being connected to endpoints
   c. Being connected to IPAM drivers
   d. Being connected to a network stack
5. A Libnetwork may be considered:
   a. As a network interface
   b. As a repository for native drivers in Docker
   c. As an implementation of IPAM
   d. As an implementation of the CNM
6. Which network driver will allow us to connect to multiple hosts without any additional configuration?
   a. The MACVLAN driver
   b. The Overlay driver
   c. The Bridge driver
   d. None of the above

## Answers

1. c
2. c
3. a
4. b
5. d
6. b

## Questions

1. Explain the concept of network namespaces

2. Explain how a VETH pair works.
3. Draw and explain the Container Network Model.
4. What is Libenetwork? What is its purpose?
5. Explain in detail the concept of a Linux bridge.
6. What are network drivers?

## Key Terms

- Container Network Model
- Libnetwork
- VETH
- Sandbox
- Endpoints
- Network Stack
- Container Isolation
- Network Namespace
- Linux Bridge
- Docker0

# CHAPTER 5

# Docker Swarm

## Introduction

When lots of containers are running, then obviously we need to have a mechanism in place to have them work together in harmony to leverage the cumulative capabilities of all these containers running together. Docker Swarm is the answer. Docker Swarm provides the technology for native clustering for Docker. Putting it differently, Docker Swarm, often known as swarm mode, is Docker's native support for orchestrating clusters of Docker engines. So, all the containers are orchestrated and run as a composite whole.

## Structure

- Docker Swarm defined
- Benefits of using Docker Swarm
- Setting up a Swarm
- The concept of service in Swarm
- Locking and unlocking a Swarm Cluster
- Networking in Docker Swarm
- Troubleshooting Docker Swarm

## Objective

In this chapter, you are going to learn about the orchestration of containers using Docker's native clustering solution known as *Docker Swarm*. You will learn how to set up a Docker Swarm, how to use it as a high availability solution, and how to leverage the capabilities inherent in a swarm to build safe, secure, and optimized solutions for your applications.

## Docker Swarm defined

Docker swarm is a platform for deploying applications across multiple hosts, using a rich API that allows for deploying complicated applications easily. Typically, the applications will be written in a manifest file, which we can then deploy easily to the cluster using native Docker commands. The whole process is automated, and Swarm can route the request from the client to the swarm cluster. Once the Swarm cluster receives the request, it takes charge of the application and manages and administers the application. As of Docker version 1.12, Docker includes swarm mode for natively managing a cluster of Docker Engines, which we now know is technically called a swarm, or to be more precise, Docker Swarm. We use the Docker CLI to create the swarm, deploy application services to it, and manage its behavior.

## Benefits of using Docker Swarm

There are a host of benefits of using Docker Swarm. Let us have a look at them:

1. **Declaration of the desired state**: It lets you declare what and how you want it. For example, the Docker Engine will let us define how many replicas of a particular application we want, or how the application will be structured. We may want a web-based frontend and a database running in the backend, and so on and so forth.
2. **Scaling**: One of the major benefits of running Docker Swarm is the facility of scaling up or down, provided natively by the software. We can easily scale up the number of replicas of an application that we need, and just as easily scale down, when the requirement goes down.
3. **No separate Orchestrator**: We don't need to find any third-party orchestration software for managing our cluster. It is already in-built. In other words, cluster management is integrated with the Docker Engine.
4. **Desired state reconciliation**: The swarm manager node constantly monitors the cluster state and reconciles any

differences between the actual state and our desired state. For example, if we have set up the desired state to have five replicas, and for some reason, one of the replicas becomes unavailable, because the machine (worker) hosting it crashed, Docker Swarm will ensure that one more replica is created on one of the surviving machines.

5. **Networks multiple hosts**: Docker Swarm provides us an off-the-shelf overlay network for our services. Addresses are automatically assigned to the containers on the overlay network when an application is started, modified, or updated.
6. **Service discovery**: There is a DNS server embedded in the swarm. The Swarm manager uses this DNS server in providing an identifier to each of the services running on the swarm. On top of it, the Swarm Manager also load balances the running containers, thereby ensuring that no one container is overloaded.
7. **Built-in security**: Docker Swarm natively uses **Transport Layer Security** (**TLS**) mutual authentication and encryption to ensure secure communication between nodes.
8. **Rolling updates**: This is a very interesting feature. Service updates to nodes can be applied incrementally to nodes. This means that we can stagger the service updation to various sets of nodes. If anything goes wrong, post update, we can easily roll back the update, and we will be none the worse for it.

## Setting up a Swarm

Setting up a swarm is super-easy. We need just a single line of command to set it up. The nodes are configured either as managers or workers. The manager(s) orchestrates the cluster, allocates work to the worker nodes, and in general, controls the cluster.

All the metadata regarding the cluster is stored in a distributed, etcd database located on all the managers. The etcd database is completely self-reliant and works on its own and requires no configuration at our end.

On the application orchestration front, we use what is technically known as a service. A service is a homogeneous unit of work on the swarm. A service can be scaled up or down, updated, rolled back, and so on. So, when we talk about allocating work to a swarm, the work is allocated as a service to the swarm. When we create a service, we specify the container image to be used, the commands to be executed inside running containers, and so on. Aside from this, we may also specify things like port, network, CPU, . etc.

When a service is allocated to a swarm, we call it a task or replica. To put it differently, we can think of service as a description of the desired state and a task as the work being done. There is a sequence of work that is scheduled for worker nodes when a service is created.

1. First, the request goes to the Docker manager.
2. The Docker manager then schedules the service to run on a set of nodes.
3. The service then runs on those nodes as individual tasks, since a service may translate into multiple individual tasks. Tasks are execution units that run once to completion. When a task stops, it isn't executed again, but a new task may take its place. Each task will have a lifecycle state like *NEW*, *PENDING*, *RUNNING*, *COMPLETE*, and so on.

Ok, let's get started. Let's build a basic swarm. This swarm will consist of one manager and two worker nodes. However, there are a few prerequisites to take care of. First, we need to ensure the following ports are open:

*2377 tcp for the client to daemon connection*

*7946/tcp 7946/udp for control plane group*

*4789/udp for VXLAN based overlay networks*

Next, we have to identify the node that we would like to designate as the manager node and obtain its IP address. It is quite simple. We just type the command:

```
ifconfig -a eth0
```

Let's see the output in the following screenshot:

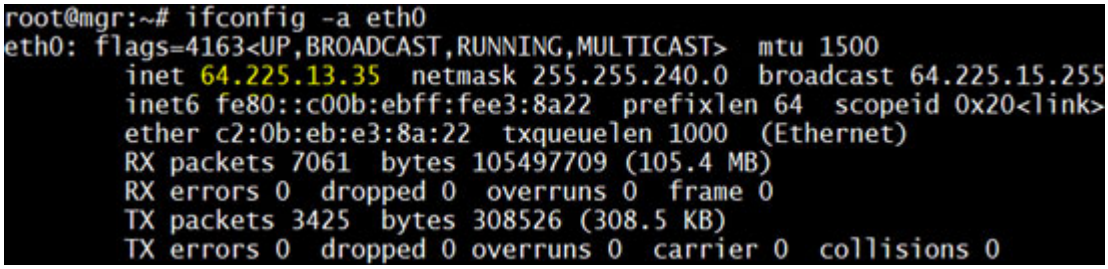



***Figure 5.1***

So, we know the IP address we need is `64.225.13.35`. Now, let us initialize this node as the manager node. We just need to run the following command, and we are good to go. Check out the screenshot in the following *Figure 5.2*:

```
docker swarm init --advertise-addr 64.225.13.35:2377 --listen-addr 64.225.13.35:2377
```


***Figure 5.2***

Let's try to make sense of the command we just ran:

`docker swarm init` simply tells docker to initialize a swarm.

`--advertise-addr` is the IP and port through which the other nodes will connect to this manager. `--advertise-addr` is optional and may be skipped because, by default, docker will pick up the IP available on `eth0`.

`--listen-addr` is the IP and port that we want to designate for listening for swarm traffic. Typically, it will be the same IP as the `--advertise-addr`, unless we have multiple IPs, and we feel the need to have different IPs for connecting and listening.

In the screenshot above, we will also notice that the output provides us with a token key for adding a node to the swarm as a worker. It also provides us with the command to add a manager to the swarm and tells us to follow the instructions after that. We can easily get the

tokens for either a manager or a worker by simply running the following commands:

```
docker swarm join-token manager
docker swarm join-token worker
```

Ok, let's move ahead now and join a couple of nodes as workers to this swarm. We will log on first to the worker node 1 and join it to the swarm, followed by logging in to the worker node 2 and joining it to the swarm. As we can see below, it's all very straightforward. Check the following screenshot:


***Figure 5.3***

Adding a second worker node. Check the screenshot in following *Figure 5.4*:


***Figure 5.4***

So, here we are. We have set up our swarm with one manager and two workers. Let us log back into the manager node and do some quick checks which we can see in the following *Figure 5.5*:

```
docker node ls
docker node inspect self
docker node inspect --pretty self
docker node inspect --pretty wkr01
```

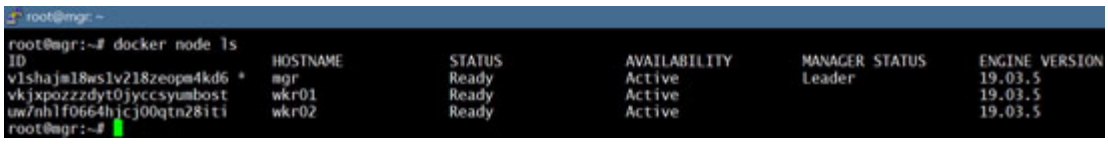



***Figure 5.5***

`docker node ls` provide us a list of nodes in the swarm cluster. This command cannot be run from the worker nodes, however. The node from which the command is run has an asterisk (*) beside the ID.

The *MANAGER STATUS* column shows the manager as a *LEADER*, which simply means that the manager is currently active. If we have more than one manager in a swarm, only one of them can be active at a given point in time. The *MANAGER STATUS* for the worker nodes is empty, which implies that these nodes are not manager nodes-they are worker nodes. We can run the following command to have a detailed look at the manager node.

```
root@mgr:~# docker node inspect self
[
  {
    “ID”: “v1shajm18ws1v218zeopm4kd5”,
    “Version”: {
    “Index”: 9
    },
    “CreatedAt”: “2019-12-15T05:46:69.372129773Z”,
    “UpdatedAt”: “2019-12-15T05:46:69.979553725Z”,
    “Spec”: {
      “Labels”: {},
      “Role”: “manager”,
      “Availability”: “active”
    },
    “Description”: {
      “Hostname”: “mgr”,
    “Platform”: {
      “Architecture”: “x85_64”,
      “OS”: “linux”
    },
    “Resources”: {
      “NanoCPUs”: 4000000000,
      “MemoryBytes”: 8364035096
    }
```

Output truncated for brevity.

The output of the above command is somewhat unwieldy and difficult to read but contains a wealth of information regarding the node. However, a much more readable output is produced when we

use the `--pretty` flag, and it ought to suffice in most cases. Let's check it out in the following *Figure 5.6*:

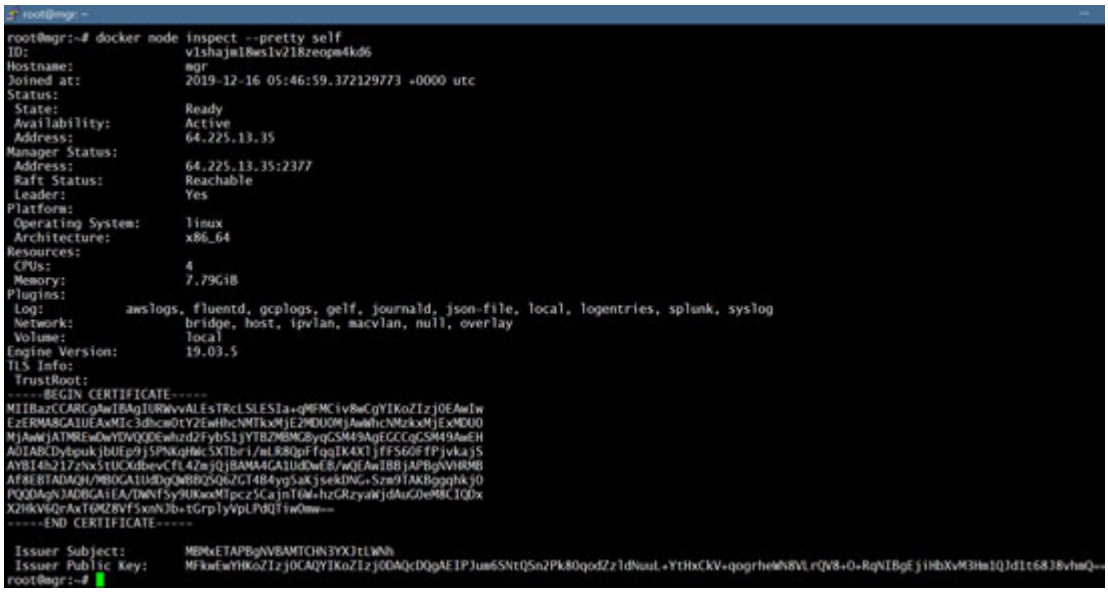



***Figure 5.6***

Similarly, we can check the other nodes from the manager node as well.

So, we have the swarm all configured, which implies that we now have native support for high availability. This means that if one or more nodes fail, the survivors will keep the swarm running. The Leader is the only one that will issue live commands against the swarm. Just for our information, if we do have more than one manager, and the passive manager receives a request, it will transfer that request across to the leader.

## Consensus in Docker Swarm

It is generally recommended to have an odd number of managers and not to have more than five managers. Now, let us try to understand why this recommendation is there.

When we have multiple servers operating together, the servers need to have a consensus. Consensus means all/most of the servers agreeing to the same information, at the same time, something which is mandatory for designing fault-tolerant distributed systems. If all the servers are not storing the same consistent state, we will not

be in a position to have any of the surviving managers pick up the task and restore to a stable state, if the Leader happens to die for some reason.

The recommendation of having an odd number of managers is made to avoid having a split-brain condition. In a split-brain condition, we end up with the manager(s) not having the quorum to make a decision, leading to an eventual breakdown of the clustered structure.

Docker Swarm is based on the Raft Consensus Algorithm. The Raft can tolerate up to *(N-1)/2* failures, that is, say, if we have *five* managers, then the *Raft Algorithm* will be able to tolerate *(5-1)/2*, that is, loss of up to *2* nodes. In case we have *three* managers, then the algorithm will allow us to lose just one node at most *(3-1)/2=1*.

To take another step forward, let us assume that in a cluster of *5* managers, we lost one manager, then as per the formula above, we should be fine. However, it now requires *(N/2) +1*, that is, *(4/2) +1=3* nodes to have a consensus for any decision to be taken in the cluster.

In the unfortunate event of having a split-brain condition, the existing tasks keep running, but the scheduler cannot rebalance tasks to cope with failures if the manager set is not healthy.

Having too many managers will impact performance adversely, simply because achieving the consensus of so many nodes is a lot of work under the covers, and there is no proportionate value-addition. So, it is best to stick to either 3 or 5 managers at most.

## The concept of service in swarm

We have already touched on the concept of service earlier. Just to reiterate, when we deploy an application, and the Docker engine is in swarm mode, we create a service. When we create a service, we can specify, amongst other things, which container image to use, which network to use, which commands to use, and which commands to execute inside running containers. When we schedule a service, the orchestrator responds by creating tasks or replicas for us.

Let's start by creating a service. Check the output in *Figure 5.7*:

```
docker service create --name myredis redis:latest
```

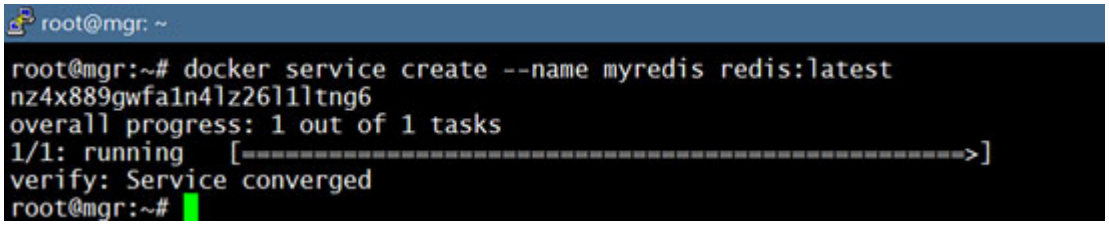



***Figure 5.7***

Let's do a few checks on the service and check what we get in *Figure 5.8*:

```
docker service ls
docker service ps myredis
docker service inspect --pretty myredis
```

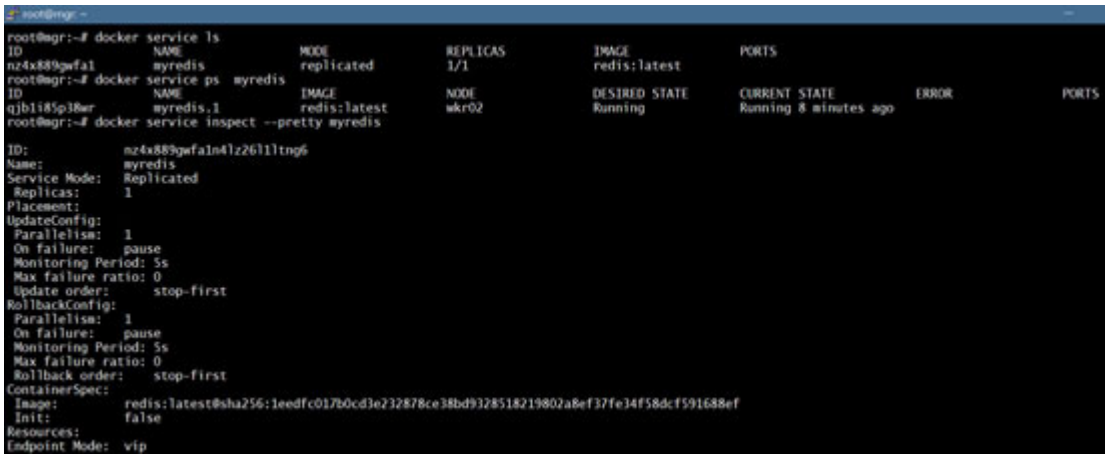



***Figure 5.8***

`docker service ls` shows us that the service created is replicated once, and it was created using the `redis:latest image`.

`docker service ps myredis` provides us some more information like the node in which the service is running, what is the desired state, what is the current state, and so on.

`docker service inspect --pretty myredis` gives us a very readable output of information about the service.

## Creating replicas

Ok, so far, so good. Let's now try to get a little deeper and try to see how replicas are created and how the services scale up and down by running the following commands and seeing the output in the screenshot in *Figure 5.9*:

```
docker service create --name myredis_replica --replicas=5
redis:latest
docker service ps myredis_replica
```


***Figure 5.9***

It is interesting to see that the replicas are spread out amongst the nodes, including the manager node. We can consider each replica as a task, and each task maps to a container. From the screenshot above, we can see two tasks (containers) are running on the manager node, two tasks (containers) are running on the first worker node, `wkr01`, and one task (container) is running on the second worker node, `wkr02`.

Let's stop docker on the `wkr02` node and see what happens. We will log in to `wkr02` and run the following commands and then check the output in the screenshot in *Figure 5.10*:

```
docker container ls -a
systemctl stop docker
```

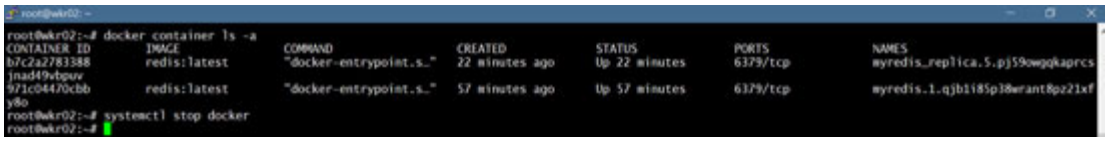



***Figure 5.10***

So, we have taken down the node `wkr02`. Now, let's find out whether the two replicas that were *hitherto* running on `wkr02` have come upon the surviving nodes or not. We can easily find this out by running the `docker service ps myredis_replica` command on the manager node. Let us check the following screenshot:

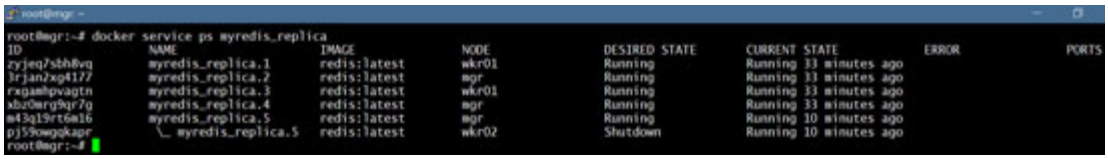



***Figure 5.11***

From the screenshot, it is apparent that the `wkr02` is in a shutdown state, and three tasks are running on the manager and two running on `wkr01`. So, we can see that the task migrated from `wkr02` to the `mgr` node when the docker was stopped on `wkr02`. However, when the docker is started up in `wkr02`, the task will not migrate back to the node.

## Scaling a Service

Scaling a service up or down is very simple in Docker Swarm. It's just a one-line command. So, let us create a service with four replicas and then scale it up to seven. Then, we will scale it down to five. Let us run the following set of commands and check the output in the screenshots taken in *Figures 5.12* and *5.13*:

```
docker service create --name mynginx --replicas=4 nginx:latest
docker service ps mynginx
docker service scale mynginx=7
docker service ps mynginx
```

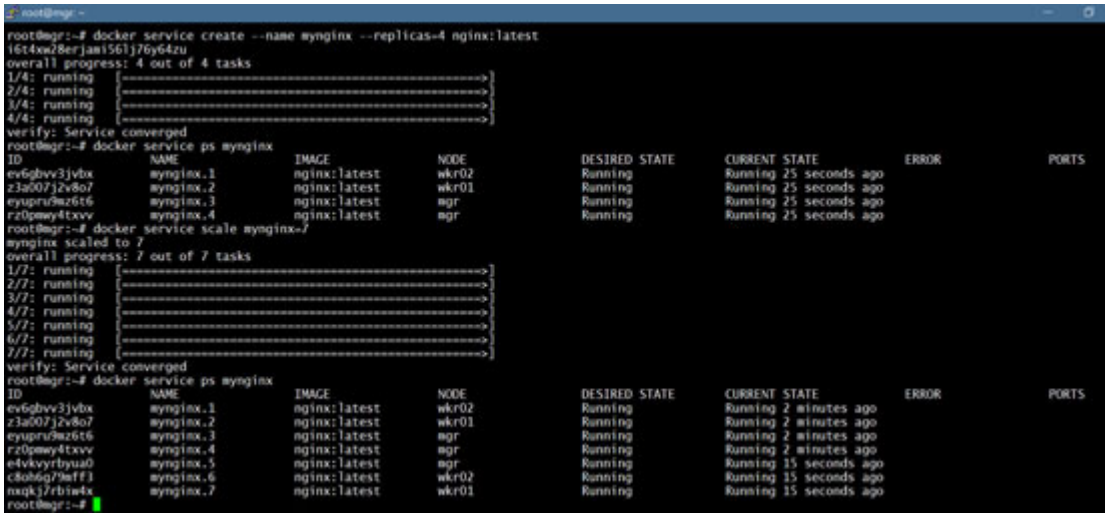



***Figure 5.12***

```
docker service scale mynginx=5
```

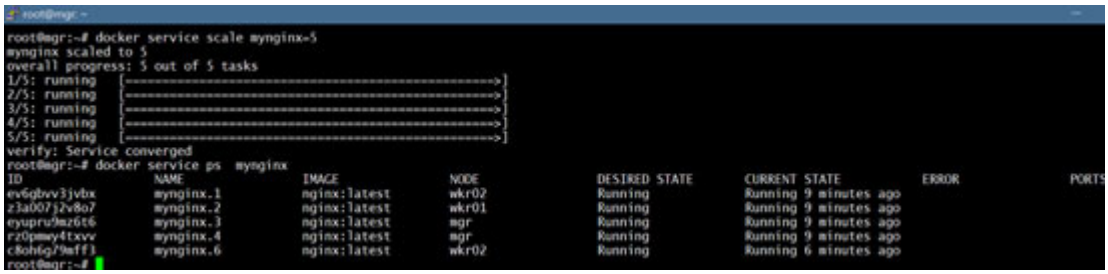



***Figure 5.13***

We had created a service with four replicas, and it was easy to check how the replicas were distributed amongst the nodes. We then scaled up the number of replicas to seven, checked how the new replicas were accommodated amongst the nodes, and then just as easily scaled down the replicas from seven to five, and again we saw the whole process was done with consummate ease under the hood. Scaling up or down in Docker Swarm is very simple and intuitive.

## Replicated and Global Services

In Docker Swarm, there are two types of services: *Replicated* and *Global*. For replicated services, we just specify the number of replicas needed and leave it to the swarm manager to distribute the replicas on the available nodes. On the other hand, for global services, the swarm manager places exactly one task on each

available node factoring in resource requirements and other constraints. This implies that when a node is added to the cluster, and if there is a global service running, then a replica of that service is created on the newly added node.

Let us test these out by running the following set of commands and then checking the screenshot in *Figure 5.14*:

```
docker service create --name replicated --replicas 3 --mode
replicated redis:latest
docker service ps replicated
```

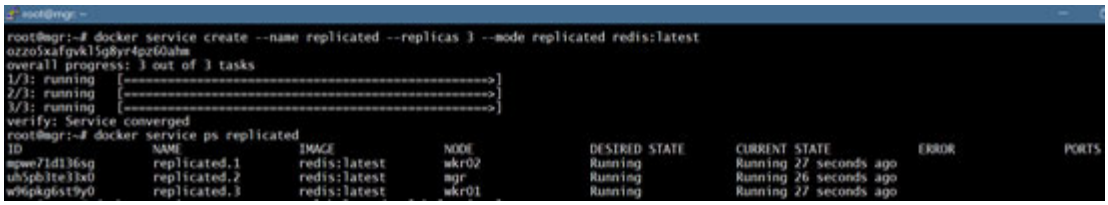



***Figure 5.14***

So, we created a service with three replicas and explicitly mentioned the default mode that is, replicated, and as expected, we had three replicas created, one on each of the nodes. Let us now create a service with the mode set as global and check what happens in the screenshot in *Figure 5.15*:

```
docker service create --name global --mode global nginx:latest
docker service ps global
```


***Figure 5.15***

Again, everything is pretty intuitive. A service gets created on each of the nodes, and up until this point, there will be no difference between a replicated service and a global one. Things start to get interesting when we add a new node to the swarm. Let's do it and check. *Figure 5.16* shows worker node 3 being added to the swarm:


***Figure 5.16***

We notice immediately that after the node is added to the swarm, a container, that is, a replica is created and running, as we can see from the above screenshot. We can confirm this by logging in to the manager node and running the following command. Check out the screenshot in *Figure 5.17*:

```
docker service ps global
```

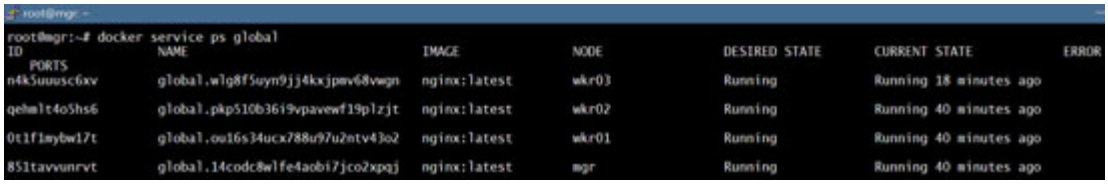



***Figure 5.17***

We see a service running on each of the nodes, including the last node added, that is, `wkr03`. So, we can conclusively say that for services running in the global mode, every time a new node becomes available, the scheduler places a task for the global service on the new node.

## Draining a Swarm

The default mode of a swarm is running with what is technically called *ACTIVE* availability. In an *ACTIVE* mode, the node is all set up to receive and execute any task assigned by the swarm manager.

Sometimes, during planned maintenance, we might need to set the node to *DRAIN* availability, which prevents the node from receiving new tasks from the swarm manager. Additionally, all running tasks are stopped and recreated on the other node having *ACTIVE* availability.

So, let us check this out with a practical example. We will create a service with two replicas. Then we will drain the node which has a replica of the running service and check whether that replica came

upon an available node or not. Let us run the following set of commands and check out the screenshot in *Figure 5.18*:

**We need to be aware that setting a node to DRAIN availability does not remove any stand-alone containers that we may have created earlier with the docker run, docker compose, or with the Docker Engine API. By putting a node in DRAIN mode, we immediately stop it from accepting any new tasks, and existing tasks are moved to available nodes.**

```
docker service create --name myservice --replicas 2
nginx:latest
docker service ps myservice
```

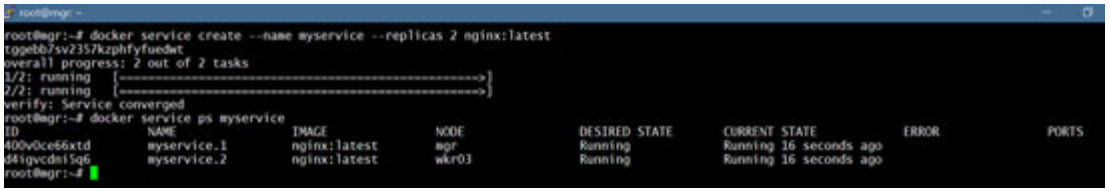



***Figure 5.18***

So, we can see that two replicas of the service are created on the `mgr` and `wkr03` nodes.

Let us now drain one of the nodes and validate the replica is actually moved to another available node. In the instant case, we are draining the `wkr03` node. Check out *Figure 5.19*:

```
docker node update --availability drain wkr03
docker service ps myservice
```


***Figure 5.19***

Indeed, the replica has come up now on `wkr01`, while the node `wkr03` is shown in a shutdown state. Now, let us make the node `wkr03` available once again and see if anything changes. Check out the screenshot in *Figure 5.20*:

```
docker node update --availability active wkr03
```

```
docker node ps myservice
```


***Figure 5.20***

The node `wkr03` continues to show a shutdown state, but in reality, it is in an active state as we can see in *Figure 5.21* by running the `docker node inspect --pretty wkr03` command:

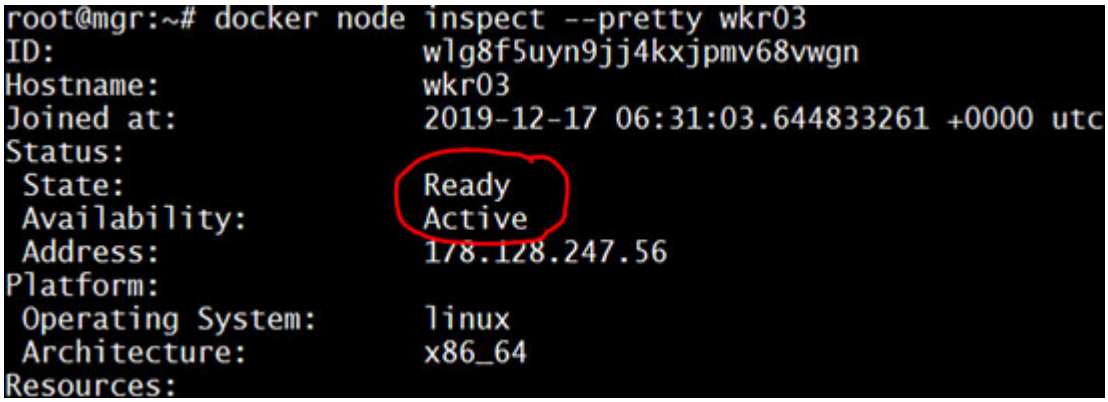



***Figure 5.21***

When we set the node back to *ACTIVE* availability mode, it is set up to receive and execute new tasks for:

- Scaling up/down service during a service update
- During a rolling update
- When we drain another node
- As a HA node for a task failure for any other node

## Locking and unlocking a Swarm cluster

Since Docker 1.13, with the introduction of secret management, the raft logs are now stored in an encrypted manner on the disk. Why are these logs so important? These logs are vital because each of the managers must have access to the same version of logs to ensure that if the current leader somehow becomes unavailable, any of the other available managers can take up the role of the leader.

With the advent of the Docker secrets feature, it becomes feasible for us to provide at-rest encryption of the Raft logs. This provides a layer of security for these logs in case if somehow attackers gain access to these logs.

The key pair used for the encryption is also stored alongside the encrypted logs. The private key is used to encrypt the Raft logs and ensure the communication between nodes is secure.

When Docker restarts, both the keys-one relating to encrypting and decrypting Raft logs on disk and the other relating to secure **Transport Layer Security** (**TLS**) communication between the nodes are cached into each manager node's memory.

From *Docker 1.13* onwards, we can take ownership of these keys and manually lock or unlock a swarm. This feature is technically known as **autolock**. Autolock forces managers that have been restarted to present the cluster unlock key before being permitted back into the cluster.

**We don't need to unlock the swarm when a new node joins the swarm, because the key is propagated to it over mutual TLS.**

When Docker restarts, we must unlock the swarm first, using an encryption key generated by Docker when the swarm was locked. We can rotate this encryption key at any time.

As we have been doing all along, let's do a practical example to understand the concept better.

We start by checking whether autolock is set to true or false. It defaults to false.

```
docker info | grep Autolock
```

So, we proceed to set autolock to true. When we set autolock to true, it will automatically generate the key we have to use to unlock the swarm. The key should be stored safely because if the key is lost or misplaced, we cannot unlock the swarm.

```
docker swarm update --autolock=true
```

The following screenshot is self-explanatory:

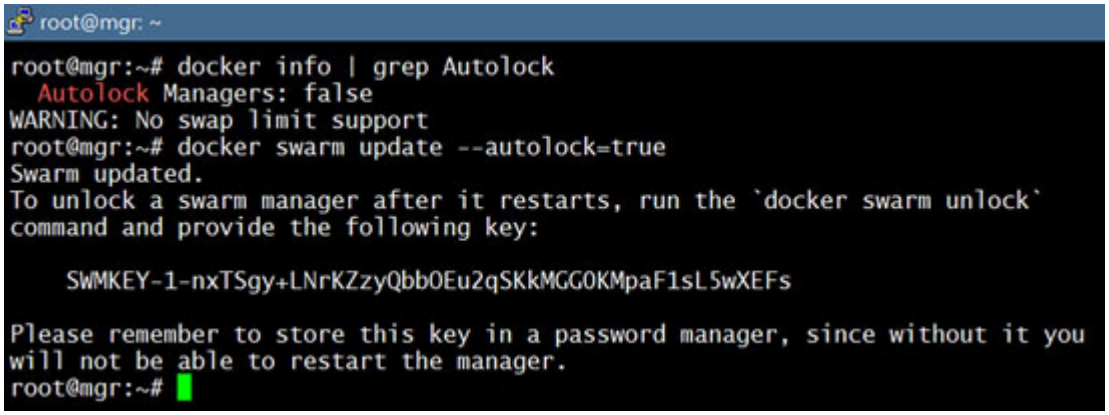



***Figure 5.22***

However, despite running the above commands, nothing changes on the ground, unless we restart docker, although running `docker info | grep Autolock` shows the value to have changed to true. Again, let us test it by running `docker node ls`, and we notice that the command executes without a hitch, which implies that the swarm is not actually locked. We will now go ahead and restart the docker. And now, as expected, the swarm will be well and truly locked! Let's test it out. Check out the screenshot in *Figure 5.23*:

```
docker node ls
systemctl restart docker
```


***Figure 5.23***

We have restarted docker, and now, if we try to do a `docker node ls`, we will get the message "*Swarm is encrypted and needs to be unlocked before it can be used. Please use "docker swarm unlock" to unlock it*."

Let us check out the following screenshot:


***Figure 5.24***

Once we are aware that the swarm is locked, let us see how a locked swarm manager impacts us. We will have a worker node leave the swarm and then try to join it back. For this, we log in to `wkr02` and run the command:

```
docker swarm leave
```

This causes the node to leave the swarm. We now use the token key that we had saved earlier for joining a node as a worker node to the swarm, to try to join back this node to the swarm. But expectedly, it doesn't work. See the following screenshot:

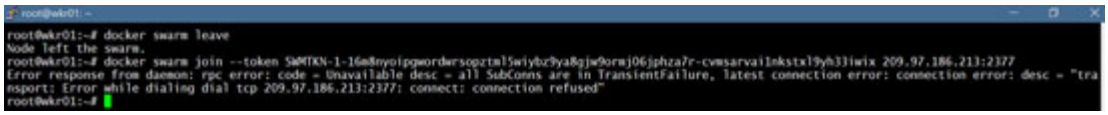



***Figure 5.25***

However, in both on the manager as well as workers participating in the swarm, we can still run all commands which are non-swarm-specific. In other words, for example, we can create a container on the manager or the worker without any issues. This is crucial for our understanding. We cannot run swarm-specific commands on a locked swarm cluster.

So, we now go back and unlock the swarm using the unlock key and then try to make the worker node join back to the swarm.

From the manager node, we run the command docker swarm unlock. We are prompted for the unlocking key, which we supply, and then we can see that our swarm is unlocked. Now we log back to the `wkr02` node and attempt to add it back to the swarm. It works perfectly! The commands and screenshots are displayed in the following *Figures 5.26* and *5.27*:

From the manager node:

```
docker swarm unlock
docker node ls
```

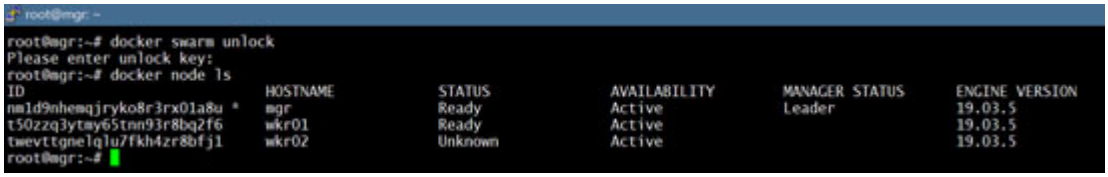



***Figure 5.26***

From the worker node:

```
docker swarm join token <token>
```


***Figure 5.27***

## Networking in Docker Swarm

Networking is a very important concept in Docker Swarm. Although we cover docker networking in detail in the chapter on networking, we will have a look at swarm related networking here.

When we create a swarm, the following happens:

1. A bridge network called the `docker_gwbridge` is created. This helps in connecting containers which are part of a swarm cluster, to the outside world.
2. An overlay network called ingress is also created. This is used to handle containers to container communication.

If no user-defined overlay network is created, the default overlay network, named ingress, is used for control and data traffic for the swarm.

We can create both user-defined overlay networks as well as user-defined bridge networks, and services or containers can easily remain connected to more than one network at the same time.

**Firewall rules for Docker daemons using overlay networks**

**We need the following ports open to traffic to and from each Docker host participating on an overlay network:**

- **TCP port 2377 for cluster management communications**
- **TCP and UDP port 7946 for communication among nodes**

- **UDP port 4789 for overlay network traffic**

Ok, let's have a quick look at the networks available for our swarm manager by running the following command and checking the output in *Figure 5.28*:

```
docker network ls
```

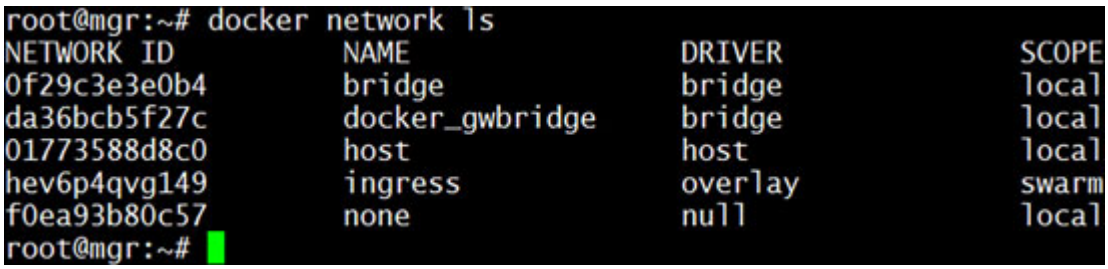



***Figure 5.28***

Expectedly, we see both the `docker_gwbridge` and the ingress networks are available.

Now, let's move ahead and create a new overlay network and see how it all pans out. Check out the screenshot in *Figure 5.29*:

```
docker network create --driver overlay new_overlay
```

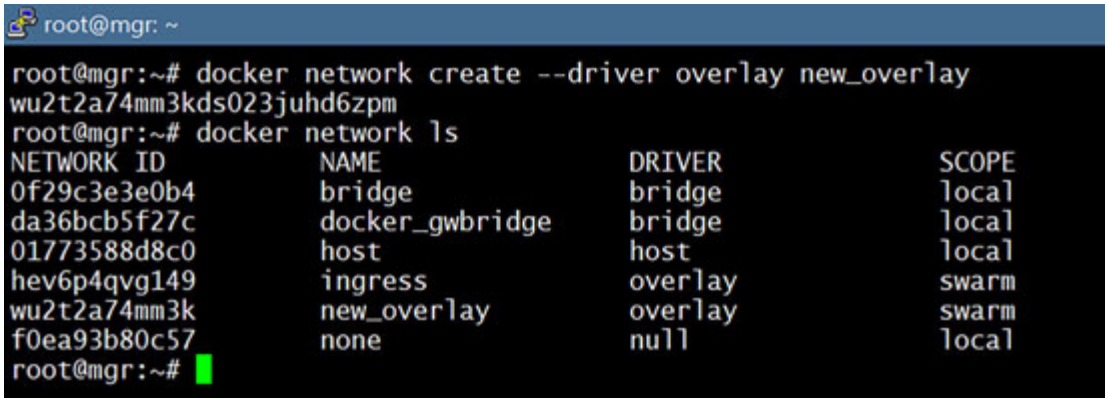



***Figure 5.29***

So, we have a new overlay network named `new_overlay` available now. The interesting thing is that we don't have to create the network again on the worker nodes separately, they are automatically propagated there. However, initially, it is not there-it only comes into existence after we create a service and associate it with the new

network. Before we get started, let us check out the networks on one of the worker nodes. See the screenshot in *Figure 5.30*:

```
docker network ls
```

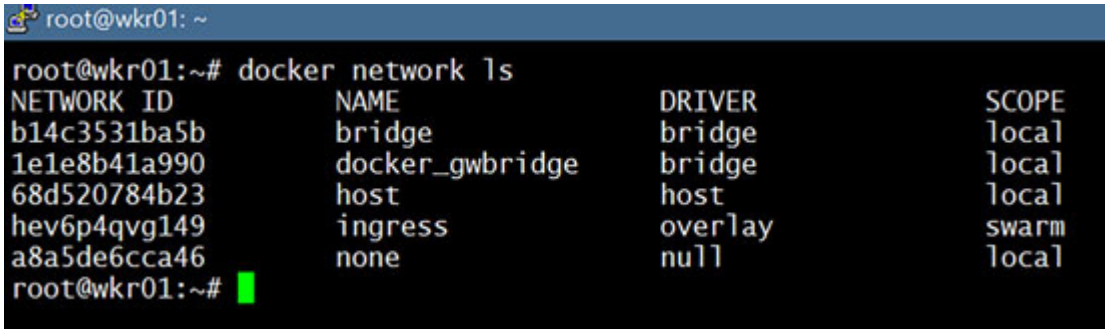



***Figure 5.30***

And sure enough, the new network that we created-`new_overlay`-is not there.

Let's move ahead and create a service and associate it with the overlay network we created. From the manager node, we run the following. Refer to *Figure 5.31*:

```
docker service create --name test --replicas 4 --network
new_overlay ubuntu:latest sleep infinity
```

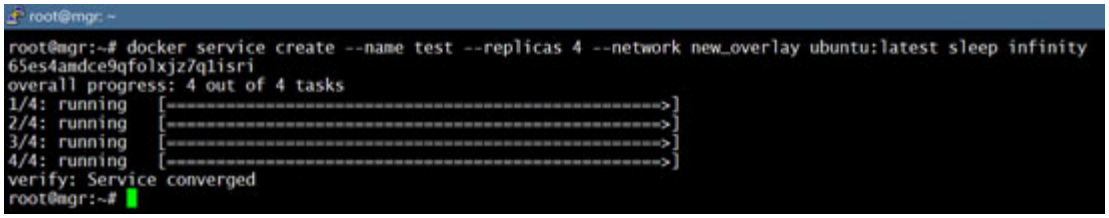



***Figure 5.31***

The service is created, so we now need to log back to `wkr01` node and check the network by running the following command and checking the output in *Figure 5.32*:

```
docker network ls
```

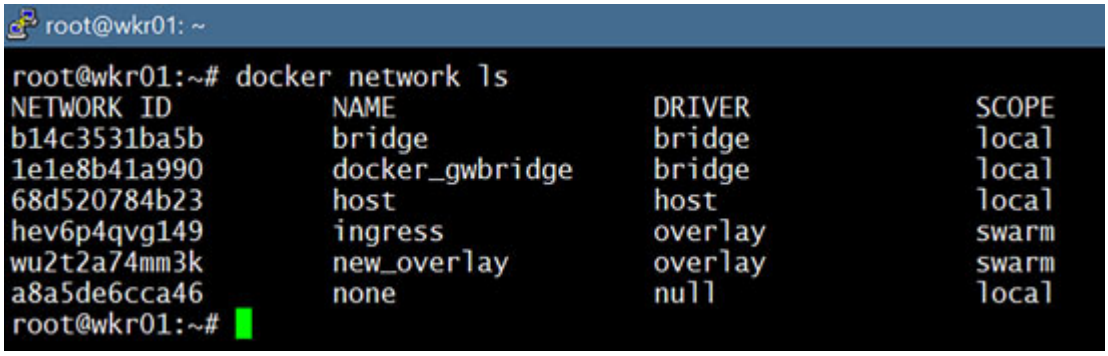



***Figure 5.32***

And, indeed, we see the network is now available on the worker node.

Ok, so far, so good. Let us delve a little deeper and check the private IP address, subnet, and gateway information. We need to run a few commands. For testing purposes, we will run it from the manager node and, subsequently, from one of the worker nodes as well. From the manager node, the output we see is shown in *Figure 5.33*:

```
docker network inspect new_overlay
```

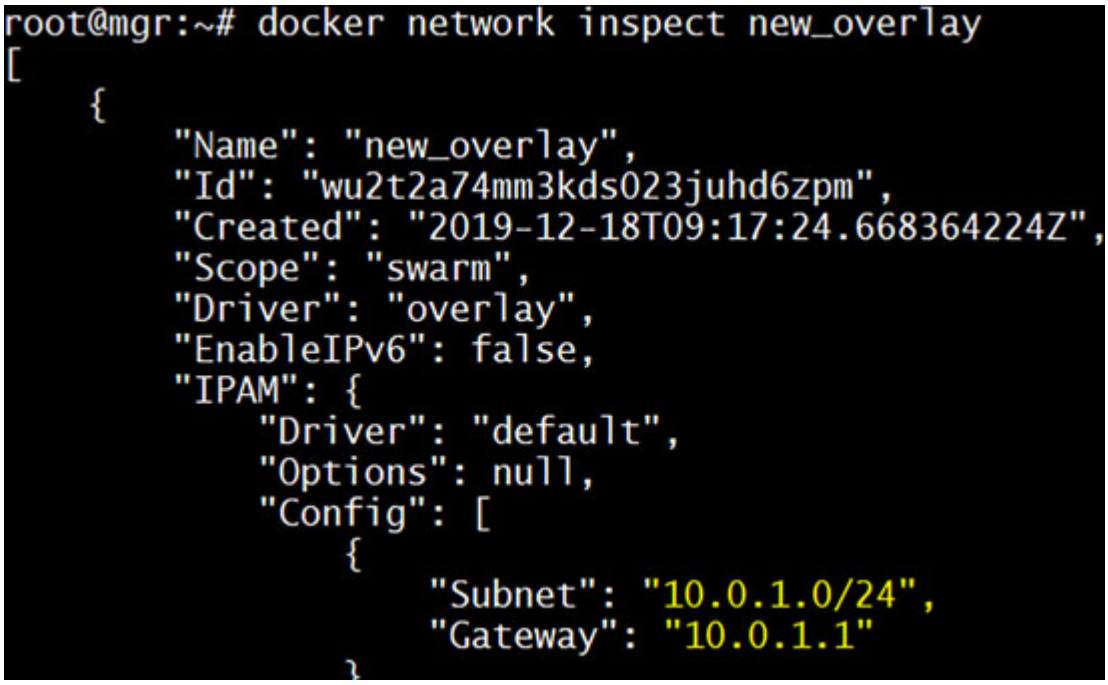



***Figure 5.33***

Next, we check the IP address by running the following command and checking the associated screenshot in *Figure 5.34*:

```
docker inspect --format='{{range .NetworkSettings.Networks}}
{{.IPAddress}}{{end}}' <container_id>
```

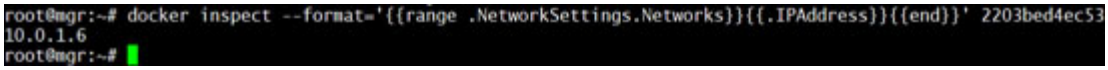



***Figure 5.34***

We can get the same information from the worker node also. Check the screenshot in *Figure 5.35*:

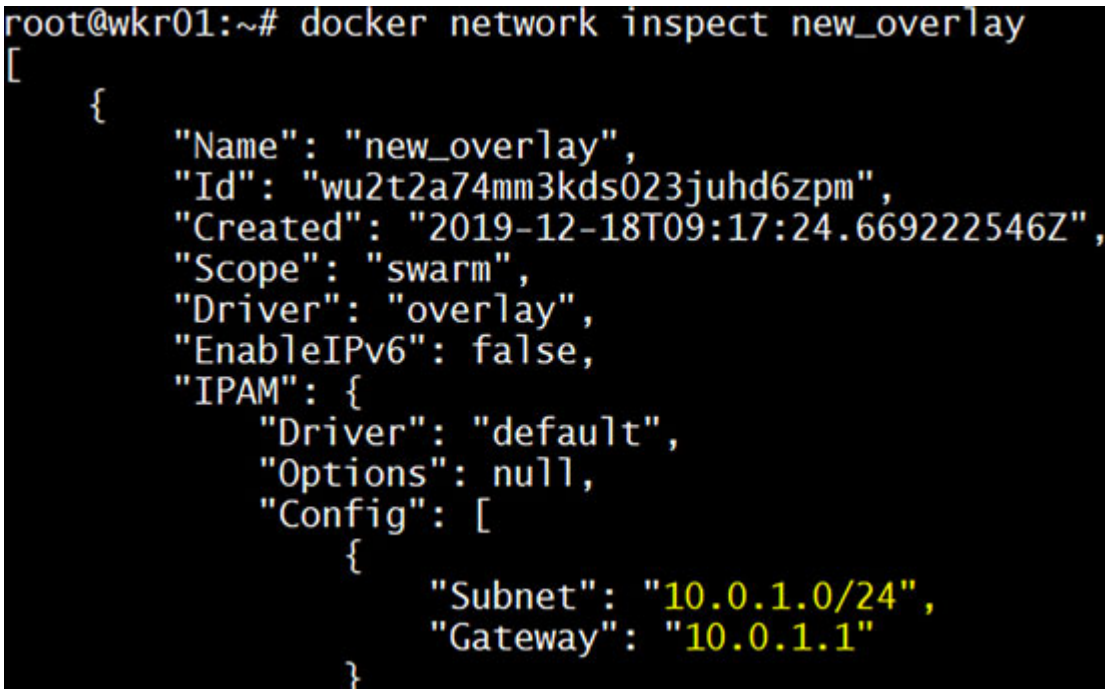



***Figure 5.35***

We can also get the IP addresses of the containers pretty easily as can be seen from the following screenshot:

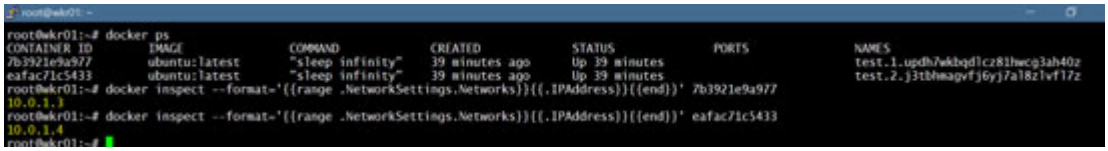



***Figure 5.36***

So, we see the information is all there. We were able to see the network information pretty conveniently. Taking a step ahead, let us get inside the manager node and check whether we can ping the worker node, and vice-versa. The idea of this exercise is to validate whether the nodes in a swarm cluster are internally accessible or not.

We need to run the first command to get the `container_id`, which will be required to run the second command below. The second command helps us to get inside the container.

```
docker container ls -a
docker container exec -it <container_id> sh
```

The following command needs to be run to install ping and traceroute commands, which may not be natively available inside the container.

```
apt-get update && apt-get install iputils-ping -y && apt-get
install traceroute
ping <worker_node_ip>
```

Following is the screenshot of the ping and traceroute commands being executed successfully from the manager node to one of the worker nodes:

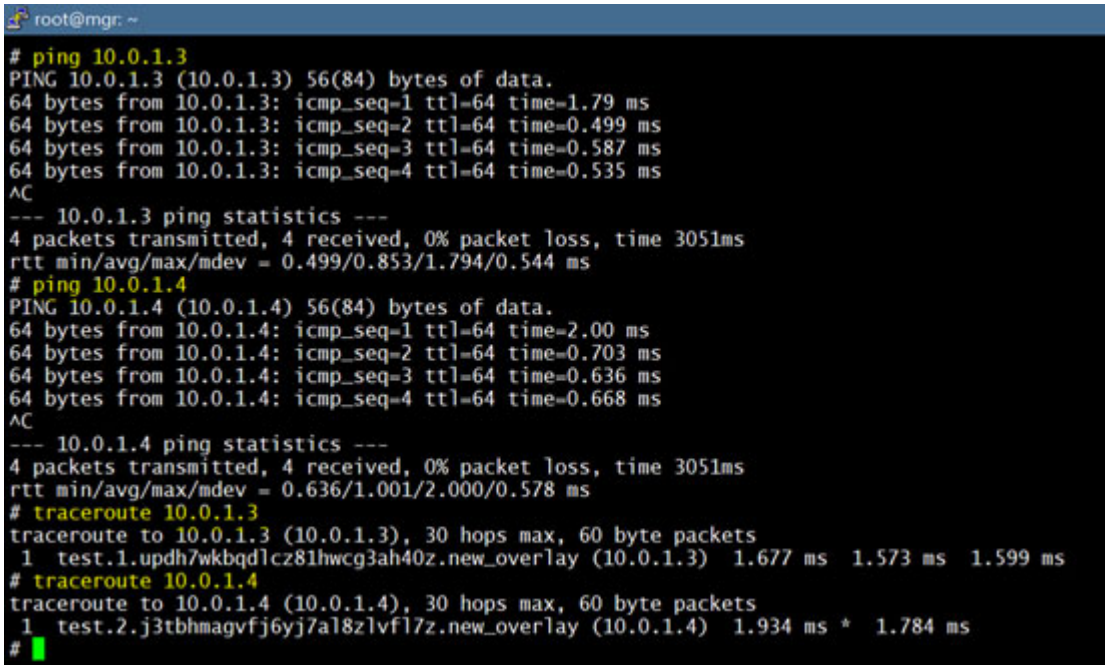


***Figure 5.37***

Let us now try to execute the same commands from one of the worker nodes (`wkr01`) to the manager node. Check the screenshot in *Figure 5.38*:

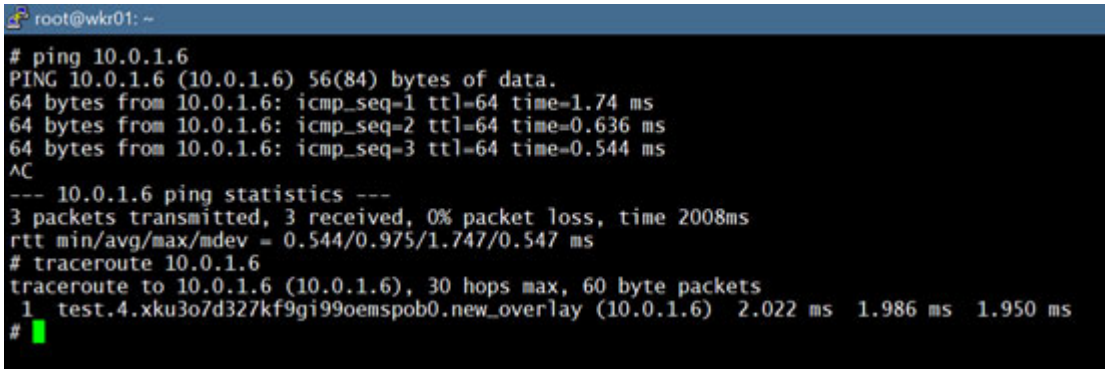



***Figure 5.38***

It ought to be amply clear now that we can ping from one node to the other inside a swarm cluster. Now let us see how we can publish ports while creating a service.

## Creating a service with a published port

Swarm services connected over an overlay network do not have a problem connecting at any level, because they expose their ports to each other. Similarly, for a client connecting to the swarm from outside the cluster, it will use the available default ports unless we explicitly specify through which port we want the connection to be established inside the cluster. To do so, we must publish the port using either the -p or the published flag. Both the legacy colon-separated syntax and the newer comma-separated value syntax are supported. The longer syntax is preferable because it is more intuitive.

Let's run the following set of commands and check the screenshot in *Figure 5.39*:

```
docker service create --name myserv_ingress -p 8081:80 nginx
docker service create --name myservice1_ingress -p
published=8080,target=80 nginx
```

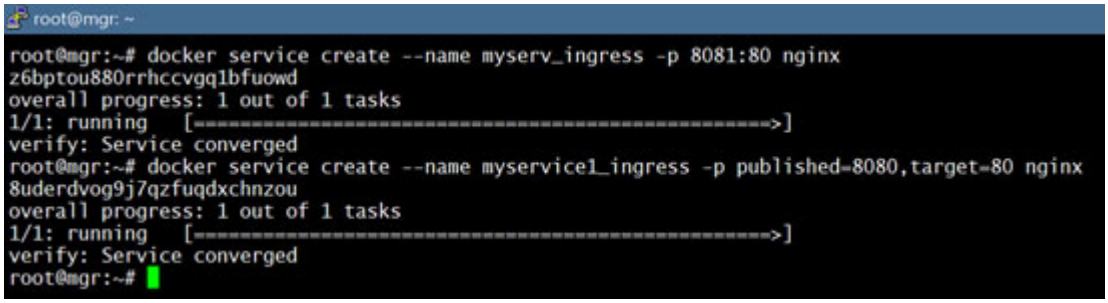



***Figure 5.39***

We can be even more specific and specify protocols as well. We will run a few examples just so we are absolutely clear on this. Check the output in *Figure 5.40*:

```
docker service create --name myservice_ingress2 -p 8082:80/udp
nginx
docker service create --name myservice_ingress3 -p 8083:80/udp
-p 8083:80/tcp nginx
docker service create --name myservice_ingress4\
 -p published=8084,target=80,protocol=udp nginx
docker service create --name myservice_ingress5 -p\
published=8085,target=80,protocol=udp -p
published=8085,target=80,protocol=tcp nginx
```

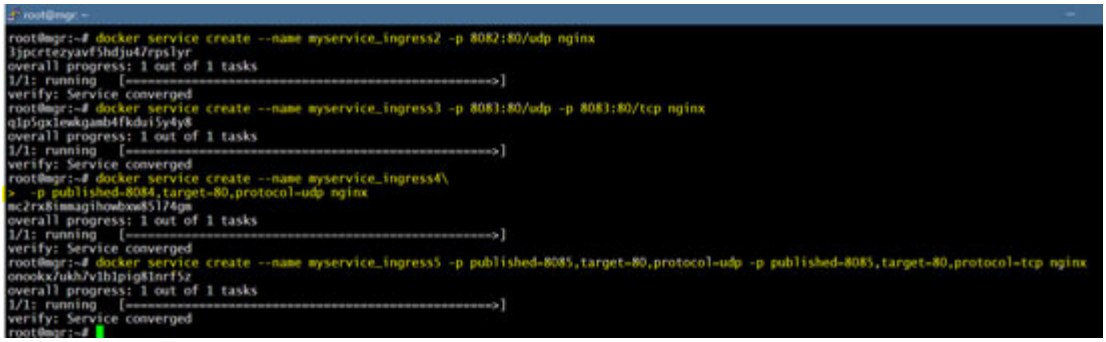



***Figure 5.40***

## Bypassing the routing mesh for a Swarm

The routing mesh uses an internal algorithm to route a service request to a worker node. So, when a client sends in a service request, there are absolutely no guarantees on which node will actually service the request. In a way, docker does a load-balancing

job under the covers. Even when we use the `--global` flag, which puts a service each on the available nodes, the routing mesh is still used.

Now, can we bypass the routing mesh, or closer to home, do we have a situation where we want to bypass the routing mesh?

Well, if we have our own load balancer and we set it up in front of our service, we may want to bypass the routing mesh. To do so, we need to start a service using **DNS Round Robin** (**DNSRR**). This can be achieved by setting the flag `--endpoint-`mode to dnsrr.

## Traffic encryption on an overlay network

All management traffic in a swarm service is encrypted by default., and it uses the AES algorithm to encrypt the stuff. Key rotation is also available, and typically, the manager node in the swarm rotates the key every twelve hours.

But, if we want to encrypt the application data as well, there is a mechanism available. We can use the `--opt` encrypted flag while creating or updating the network. This enables IPSEC encryption at the level of the VXLAN. Docker creates IPSEC tunnels between all the nodes where tasks are scheduled for services attached to the overlay network. These tunnels also use the AES algorithm in GCM mode, and manager nodes automatically rotate the keys every 12 hours. However, this encryption doesn't come for free. There is a performance penalty that is imposed because of the encryption, and we need to factor that in when we are thinking about this option in our production environment.

## Troubleshooting Docker Swarm

If we need to troubleshoot a problem in docker swarm, a good place to start would be the service logs, provided our driver supports the docker service logs command. By default, docker is configured to use the *json-file log* driver, which supports the docker service logs command. The logs may be somewhat unwieldy, and so it is a good practice to follow the logs (`--follow`) or tail them (`--tail`).

## Conclusion

In this chapter, we learned about docker swarm, how it helps to create a **high availability** (**HA**) through the creation of a cluster, such that if a node fails, the task(s) running on it will come upon one of the surviving nodes. We also talked about the consensus algorithm underlying the clustered set up and how everything is orchestrated inside the swarm. We had a glimpse of the load-balancing capability of the docker swarm through the use of a routing mesh. We looked into how docker swarm allows us to scale up or scale down the number of replicas, and we also had a look into the security mechanism available via autolocking and encryption. We also had a deep look inside the networking in a docker swarm, especially relating to creating services on non-default ports, and using non-default protocols. Overall, this chapter gave us a good idea about what is docker swarm, how it works, and how everything relating to it is tied together.

## Points to Remember

- Docker swarm is docker's native orchestration and HA solution.
- Docker swarm is tightly integrated into the Docker ecosystem and uses its own API.
- Docker swarm is easy to set up and manage, and pretty intuitive.
- Docker swarm allows us to scale up or scale down the number of replicas very easily, making it very simple to set up containerized applications and get it running.
- Docker has several built-in security measures, which allows us to use docker off-the-shelf without having to plug in additional security patches.
- Docker swarm allows us several options to customize the network, although it has a robust default network setup going.
- Docker gives us the flexibility for quick container deployments and scaling up quickly, even with very large clusters.

## Multiple Choice Questions

**Choose the most appropriate answer:**

1. Docker Swarm allows us to deploy applications quickly across multiple hosts using:
    a. A minimal API
    b. A hypervisor
    c. A rich API
    d. Secure networking
2. Docker Swarm is
    a. A backup solution
    b. An orchestrator
    c. A load balancer
    d. An orchestrator and load balancer
3. Docker Swarm automatically creates
    a. An overlay network
    b. An overlay network and a bridge network
    c. A bridge network and a host network
    d. An underlay network
4. Replicas can be created in the following modes:
    a. Universal and global
    b. Universal and replicated
    c. Replicated and global
    d. We cannot specify modes for creating replicas
5. To implement autolock in Docker Swarm
    a. We need to update autolock to true
    b. Autolock is on by default on docker swarm
    c. We need to update autolock to true and restart docker
    d. We need to update autolock to true and rotate the keys

6. When a Replica is created with the global mode flag
    a. A new replica is automatically created when a node is added to the cluster
    b. A new replica has to be manually created when a node is added to the cluster
    c. No new replicas can be created on the newly added node
    d. A newly added node cannot become part of an existing cluster
7. When a swarm has been locked
    a. A new node cannot join the swarm unless the swarm has been unlocked
    b. A new node can join the swarm right away
    c. A new node can join the swarm if at least one manager node is available
    d. A new node can join the swarm only as a manager node
8. When a swarm has been set to DRAIN availability
    a. Existing tasks will continue to completion, but no new tasks can be accepted
    b. Existing tasks will be terminated right away, and they will not come up in other nodes
    c. Existing tasks will be terminated right away, and they will be recreated in other nodes having ACTIVE availability
    d. Existing tasks will continue to completion, and new tasks can also be accepted

## Answers

1. c
2. d
3. b
4. c
5. c

6. a
7. a
8. c

## Questions

1. What is a Docker Swarm?
2. What do you understand by the term Service? How is it different from a task or a replica?
3. Explain the Routing Mesh.
4. What is the procedure to be followed to lock a Docker Swarm?
5. Explain Consensus in the context of Docker Swarm.
6. Explain the difference between Replicated and Global modes.
7. Explain the concept of overlay networks in Docker Swarm.
8. Explain how we can publish non-default ports and protocols in a Docker Swarm network.

## Key Terms

- Docker Swarm
- Swarm manager and worker nodes
- Service
- Task
- Replica
- Tokens and Keys
- Consensus algorithm
- Overlay network
- Bridge network
- Routing Mesh
- Autolock
- Replicated and Global services
- DRAIN and ACTIVE availability

- Published ports

# CHAPTER 6

# Docker Networking

## Introduction

This chapter deals with docker networking. Docker containers need to communicate with one another, and they do so over networks that are established when the Docker software is installed. The Docker software, for the most part, leverages Linux network capabilities to establish the network infrastructure that we are going to learn about in detail in this chapter.

## Structure

- Docker Networks
- The Bridge Network
- The Host Network
- The None Network
- Using an existing container's namespace
- Port Mapping
- MACVLAN Network
- Overlay Network

## Objective

The objective of this chapter is to go deep inside the docker networking infrastructure and understand how it all works. In this chapter, we will look into the different networking options available in Docker and their main features, so that by the time we reach the end of the chapter we ought to have a thorough understanding of all the networks, such that we are in a position to leverage their capabilities to the hilt.

# Docker Networks

Let us start with by having a peek at the networks available on the system at this point by running the following command and checking the following screenshot in *Figure 6.1*:

```
docker network ls
```

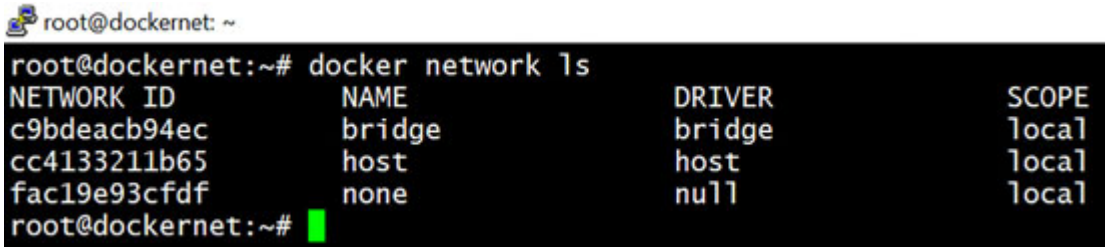



***Figure 6.1***

There will be three standard networks available on any docker installation unless we have installed docker swarm, in which case some other networks also come into the picture, which we will discuss later in the chapter.

# The Bridge Network

Let us explore the bridge network by creating a container and see how things pan out from the network perspective. But, before we do so, let's quickly understand a little bit about the bridge network. For the moment, let us think about the bridge network as a linking device operating at the link layer to forward traffic to other networks. When we install the docker software, the software automatically installs the docker bridge, which sets up rules, so that all containers automatically get connected to the docker bridge, and this helps to establish a communication channel amongst the containers.

Let us now go ahead and create a busybox container and name it `bbox`.

Check out the following commands and the attached screenshot in *Figure 6.2*:

```
docker run -dit --name bbox busybox:latest
docker container ls -a
```

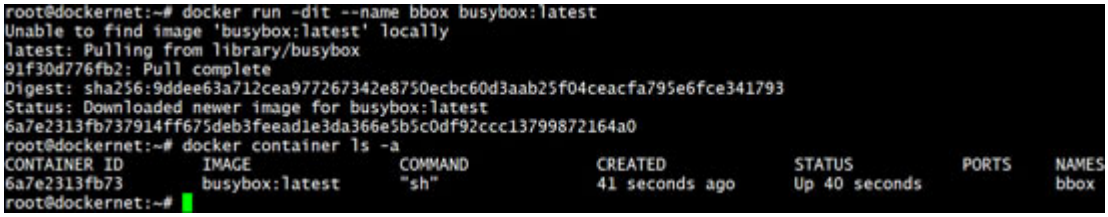



***Figure 6.2***

The container is created. Now, let us get inside the container and have a look using the following command and checking the attached screenshot:

```
docker exec -it bbox sh
ip addr
```

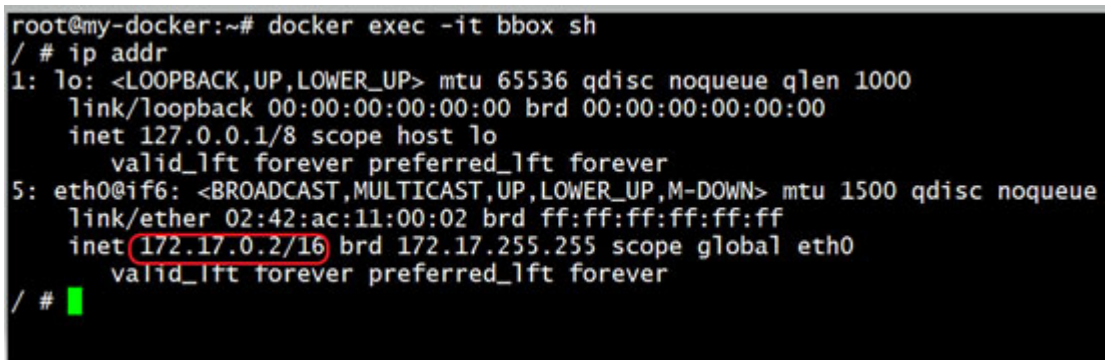



***Figure 6.3***

The container has an IP address of `172.17.0.2`. This IP address is generated by Docker's IP Address Management System or more commonly known as IPAM. Typically, the first IP address in this series, that is, `172.17.0.1`, will be assigned to the docker0 bridge And the broadcast IP will be *172.17.255.255*. All the IPs in between these two IPs are assignable to containers. Additionally, we note that one end of the veth is hooked in here and has the name `eth0@if6`. The other end will be found plugged on the host system. We will look at it in a moment. Let us now exit from inside this container by running exit. This will bring us back to the host. Now, let us run the `ip addr` command and check the output. The following screenshot follows the command in *Figure 6.4*:

```
ip addr
```

```
root@dockernet:~# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 1a:03:65:ea:b3:32 brd ff:ff:ff:ff:ff:ff
    inet 165.22.218.221/20 brd 165.22.223.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet 10.47.0.5/16 brd 10.47.255.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::1803:65ff:feea:b332/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether ea:bd:0f:75:61:90 brd ff:ff:ff:ff:ff:ff
    inet 10.139.0.2/16 brd 10.139.255.255 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::e8bd:fff:fe75:6190/64 scope link
       valid_lft forever preferred_lft forever
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:00:58:a7:65 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:ff:fe58:a765/64 scope link
       valid_lft forever preferred_lft forever
6: veth5adbd98@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
    link/ether 8e:32:7d:2a:34:87 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::8c32:7dff:fe2a:3487/64 scope link
       valid_lft forever preferred_lft forever
root@dockernet:~#
```

***Figure 6.4***

We see the other end of the veth pair hooked on the system here, and it has the name veth5adbd98@if5. Let's take a step back and see how it all looks:

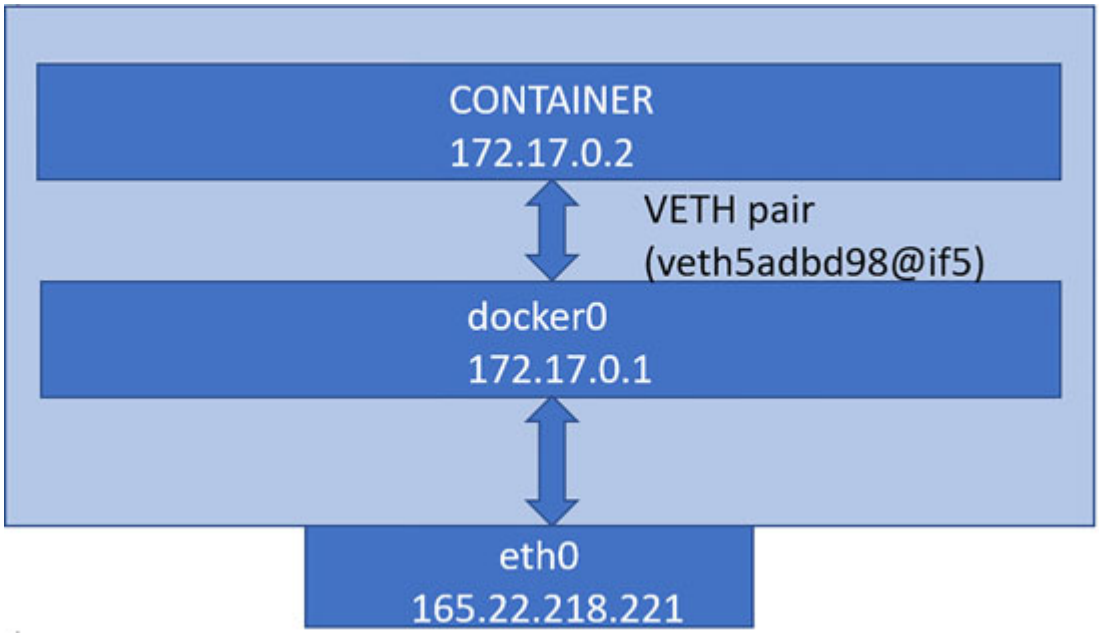



***Figure 6.5***

The host network namespace has the docker0 interface and the eth0 interface, which acts as an endpoint. Typically, eth0 may be NIC or a virtual NIC if the host is running on a VM. The host and the container

are connected by the veth pair, as shown in the diagram in *Figure 6.5*.

All traffic will flow through the eth0 interface and then will be routed to the respective containers by docker0.

So far, so good. Now that we have a container on our system let us check whether we can see the network namespace from the host. However, before we do that, let us be clear that a network namespace is an object typically under `/var/run/netns/NAME`.

However, unfortunately, the docker does not create the network namespace in the default location. We can easily find out where it is created by running the command:

```
docker inspect bbox --format '{{.NetworkSettings.SandboxKey}}'
```

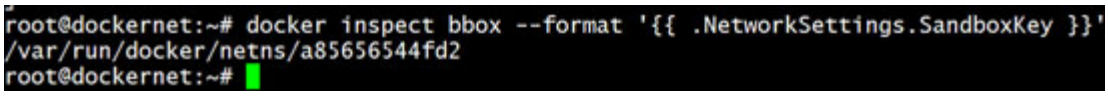



***Figure 6.6***

So, the network namespace is under `/var/run/docker/netns` instead of `/var/run/netns/`. This implies if we try to check for the docker network namespace by running the command `ip netns list`, we will not get output for the docker network namespace, which is perfectly understandable.

The following screenshot proves this:

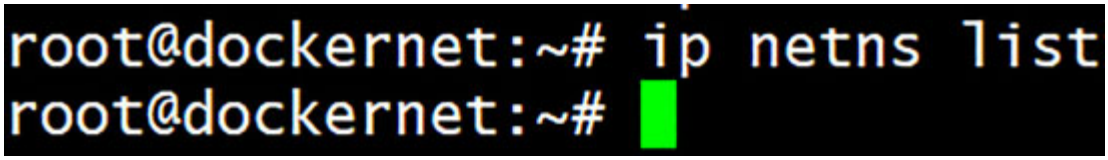



***Figure 6.7***

The way out here would be to create a symbolic link. Let us create the symbolic link by running the following set of commands:

First, we set a few environment variables.

```
container_id=bbox
container_netns=$(docker inspect ${container_id} --format\
  '{{.NetworkSettings.SandboxKey}}')
```

Now, we run the command for actually creating the symbolic link and check the output in *Figure 6.8*.

```
ln -sv ${container_netns} /var/run/netns/${container_id}
```


***Figure 6.8***

Let us check whether we can see the network namespace now by running the following command and checking the output in *Figure 6.9*:

```
ip netns list
```

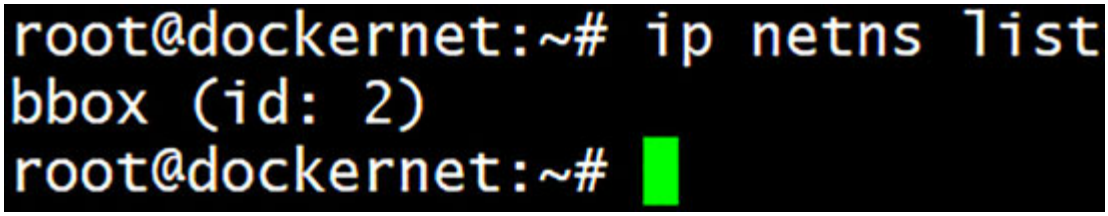



***Figure 6.9***

Thus, we can see the docker network namespace from the Linux level.

Now, let us look at how containers talk to each other over a bridge network. Let us get rid of this container and create a couple of new ones.

```
docker container rm bbox –force
docker container ls
```


***Figure 6.10***

Now, let us create a couple of containers, which, as we now know, by default, would be using the docker0 bridge network.

```
docker run -dit --name mybox busybox:latest
docker run -dit --name mybox1 busybox:latest
```

See the output in the following screenshot:

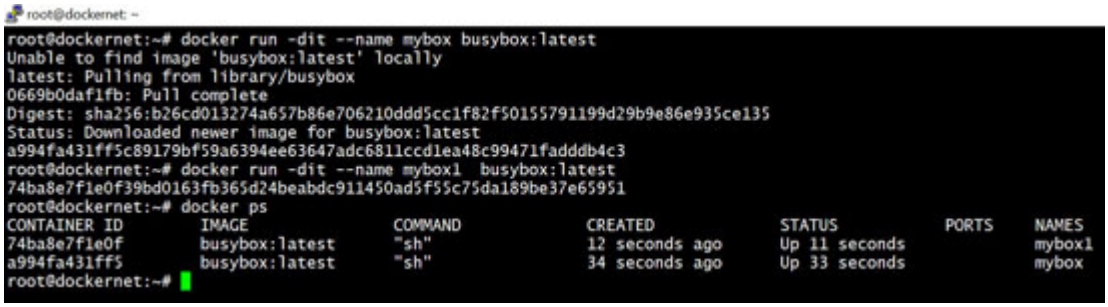



***Figure 6.11***

Now, let us get inside the containers and get the IP address of each of the containers, as shown in the screenshot in *Figure 6.12*:

```
docker exec -it mybox sh
ifconfig
```

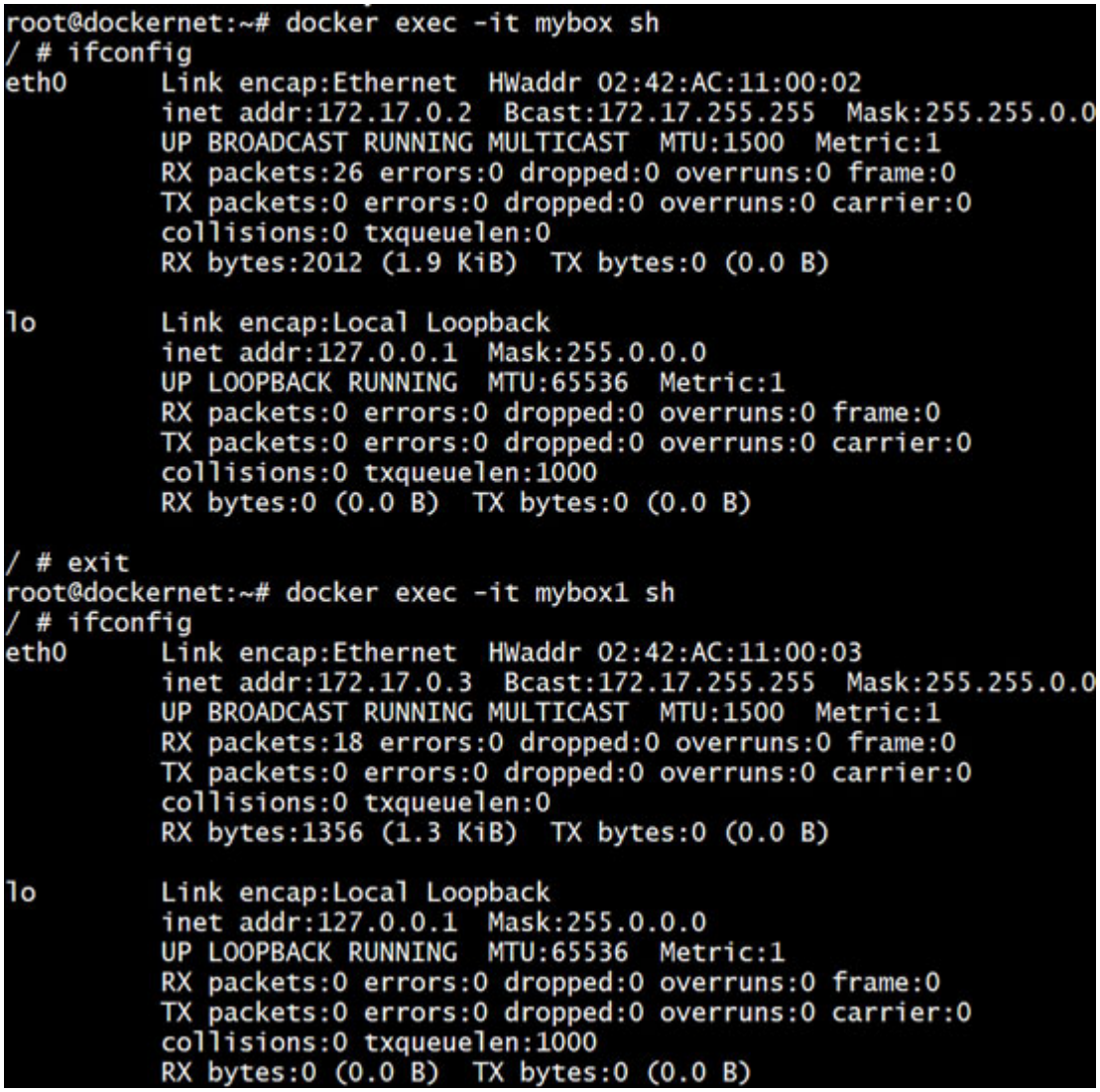



***Figure 6.12***

Now let us try to ping from the second container to the first one and vice versa, and see the output in *Figure 6.13*:

```
ping 172.17.0.2
exit
docker exec -it mybox sh
ping 172.17.0.3
```

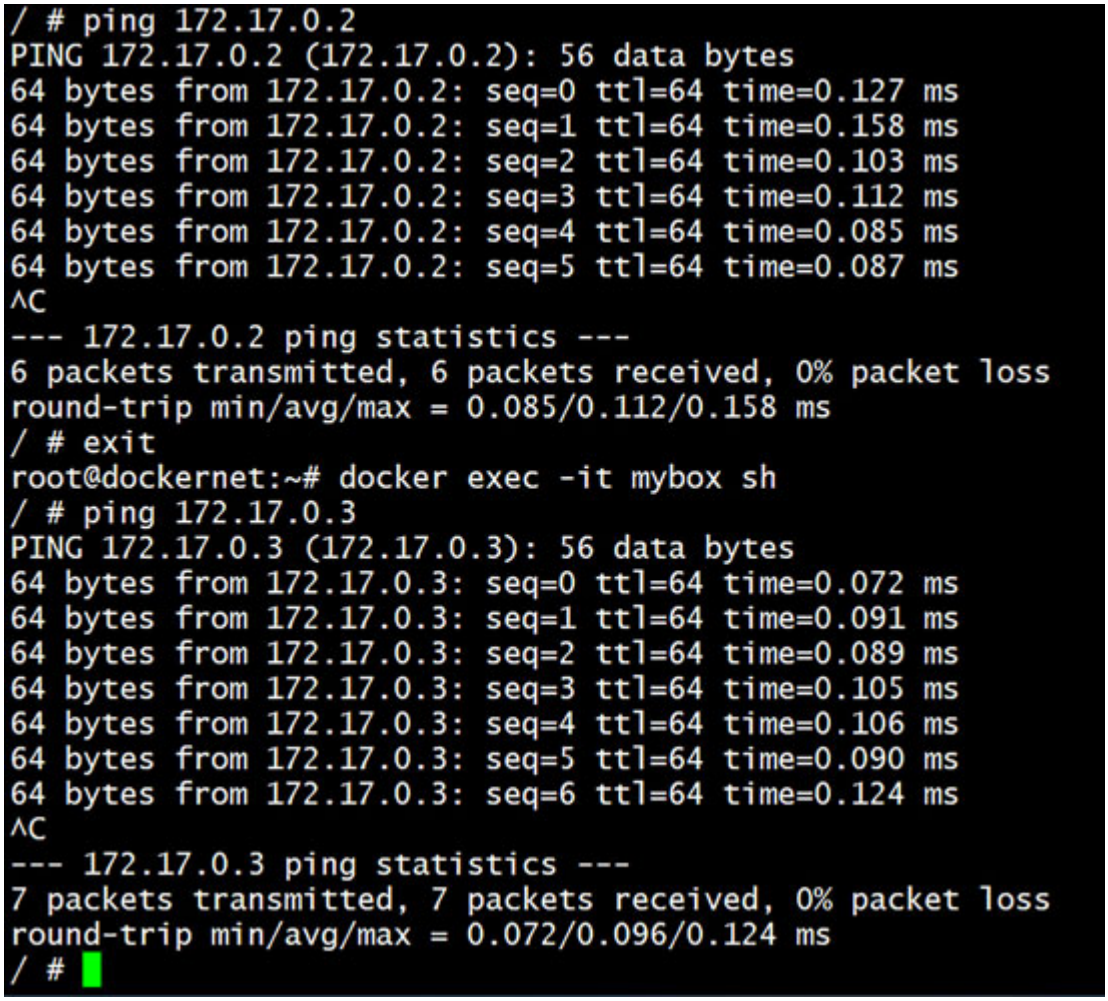



***Figure 6.13***

So, everything works as expected. However, when we try to ping another container using that container's name, we are not able to do so. See the following screenshot:

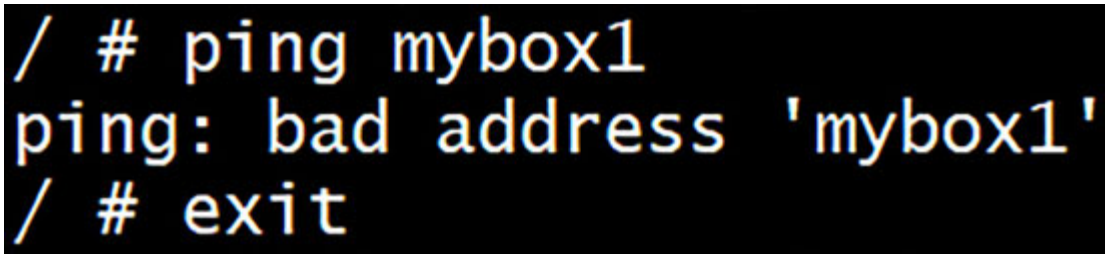



***Figure 6.14***

An interesting thing to note here is that if we create our own bridge network and attach containers to it, we can ping from one container

to the other using it's name. We can check it for ourselves below. The output is in *Figure 6.15*:

```
docker network create -d bridge mybridge
docker network ls
docker run -dit --name mybox3 --net mybridge busybox:latest
docker run -dit --name mybox4 --net mybridge busybox:latest
docker exec -it mybox3 sh
ping mybox4
```

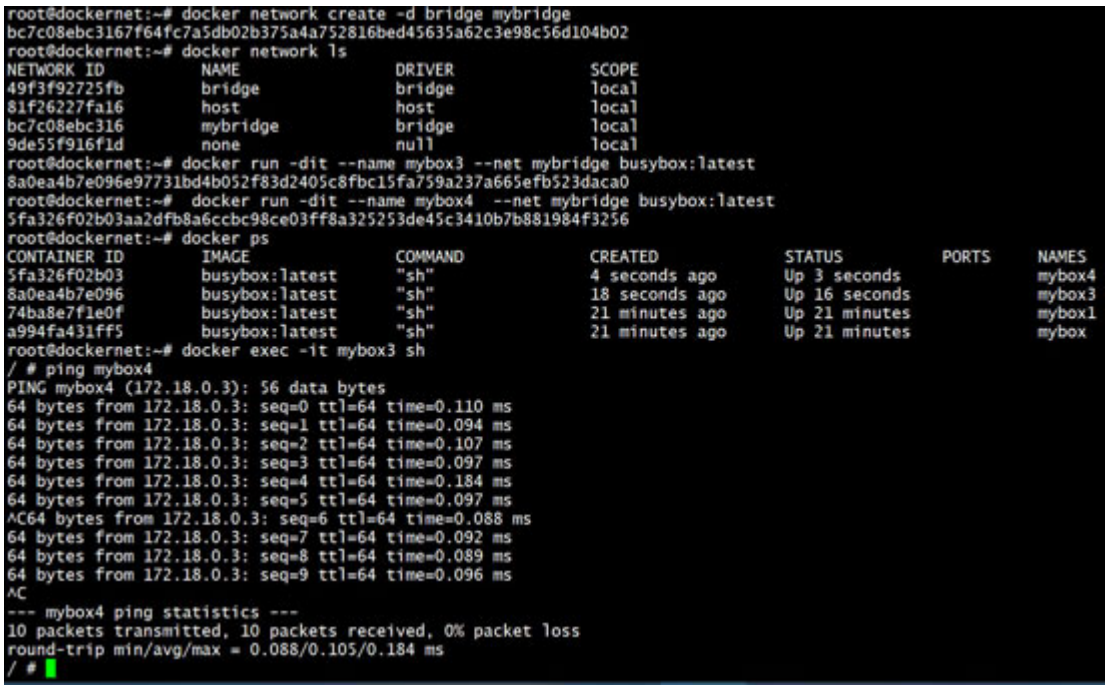



***Figure 6.15***

## The Host Network

We can also use the host network to create a container. If we do so, the container is created without its network namespace and uses the host's networking namespace. On top of it, the container is not allocated its own IP address. Now, why would we want to do that? Normally, there is no real reason unless we are trying to debug or analyze the traffic flowing through the host network.

**Since the container does not have its own network stack and it uses the host's network stack, there is no scope for port-**

**mapping, and all port mapping options are ignored and produce a warning message instead.**

Running a container using the host's namespace is pretty simple; we just need to attach the container to the host's network, as shown in the following *Figure 6.16*:

```
docker run -dit --name hostnet --network host busybox:latest
```

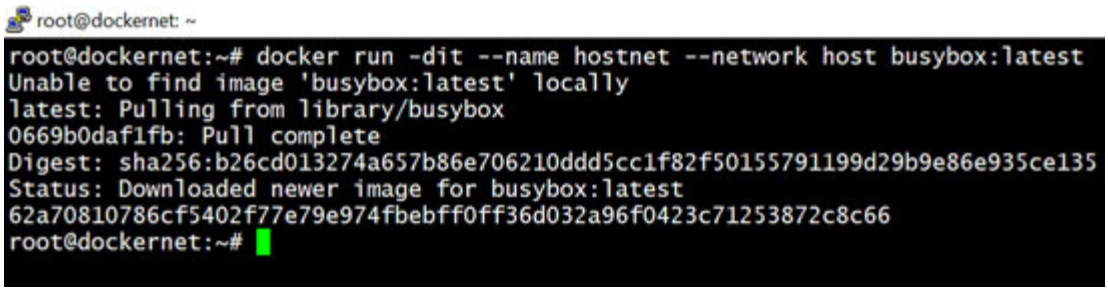



***Figure 6.16***

Now, let us pull up the IP address of the host and match it with the IP address that is visible from inside the container. The highlighted parts of *Figure 6.17* tell us the story. This is the expected behavior.

```
ifconfig -a eth0
docker exec -it hostnet sh
ip addr
```

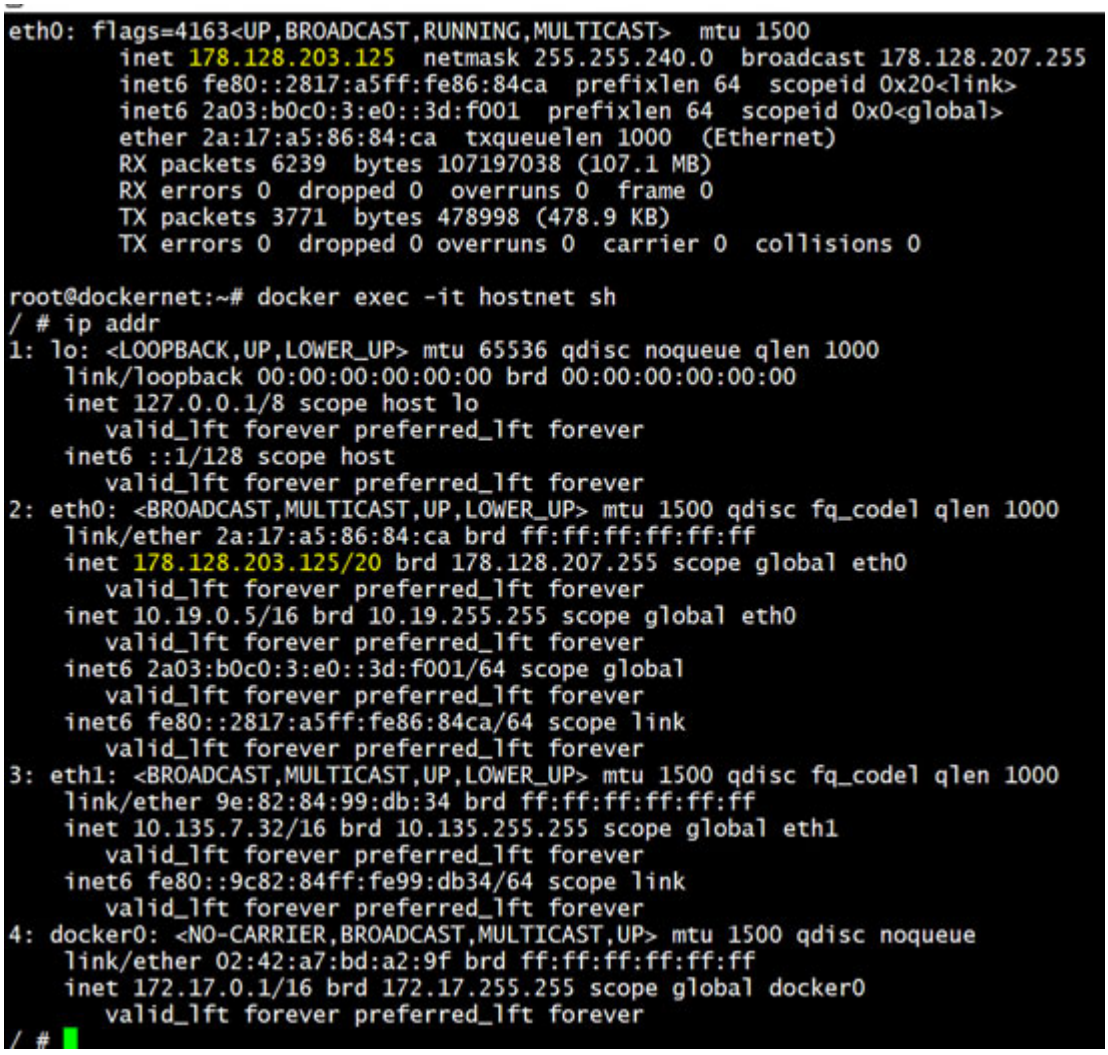



***Figure 6.17***

So, we see that the entire network stack of the host is visible from inside the container; and the IP address is the same as well. As discussed earlier, the container has no network stack of its own.

## The None Network

When we don't have any requirement for the container to connect to the outside world, we can use the none network. This container will be completely isolated from the outside world, and applications running inside the container will be insulated from anywhere outside the container. Let's have a quick look at it before moving ahead.

See the following command and the screenshot in *Figure 6.18*:

```
docker run -dit --name nonenet --network none busybox:latest
```

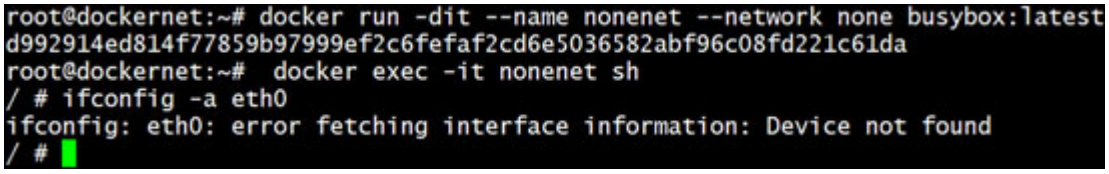



***Figure 6.18***

As expected, the container doesn't have an interface for connecting to the outside world.

## Using an existing container's namespace

There is a mechanism by which we can have a container use the network stack of an existing container. In other words, the new container will not have a network stack of its own; instead, it will start using the existing container's network stack. The new container will have its processes and filesystem but will share the same IP address and port numbers as the first container, and the two containers will be able to talk over the loopback interface.

## Port Mapping

Port mapping is very important, because the outside world needs to communicate with the container, and it can only do so when it goes through a host port. The host port needs to be mapped to a port in the container. See the following diagram:

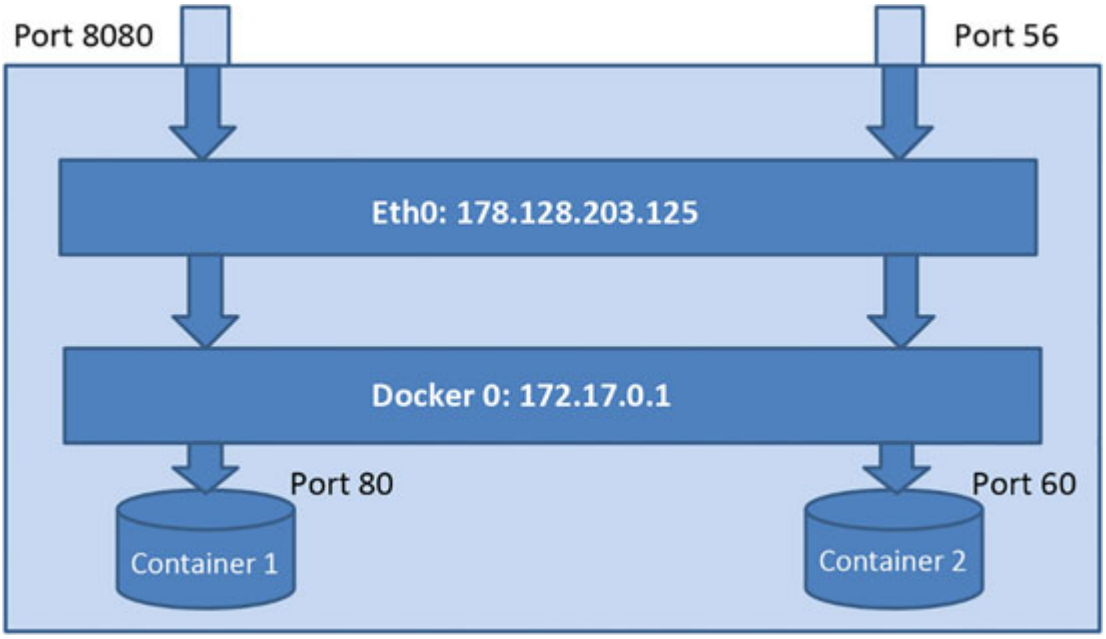



***Figure 6.19***

If we look at the diagram, we will see that the port 8080 on the host is mapped to the port 80 on the container while the port 56 on the host is mapped to port 60. So, all external traffic will be channeled through the host port to the container port to create a connection.

But none of this is required when the container wants to communicate with the outside world. This is possible because the container's IP address is hidden behind the host's IP address. It is as if the host itself is talking to other applications. To do this outbound **Network Address Translation** (**NAT**), Docker uses the Linux netfilter framework. We can see the rules using the netfilter command-line tool iptables. Check the output of the command `iptables -t nat -L` in *Figure 6.20*:

```
iptables -t nat -L
```


***Figure 6.20***

As we can see, we have a rule in the *POSTROUTING* chain that masquerades or hides anything sourced from our *docker0* bridge-`172.17.0.0` behind the host's interface. This allows us to communicate to the outside world from inside the container, as shown in the following screenshot:

```
docker run -dit –name test_con busybox:latest
```

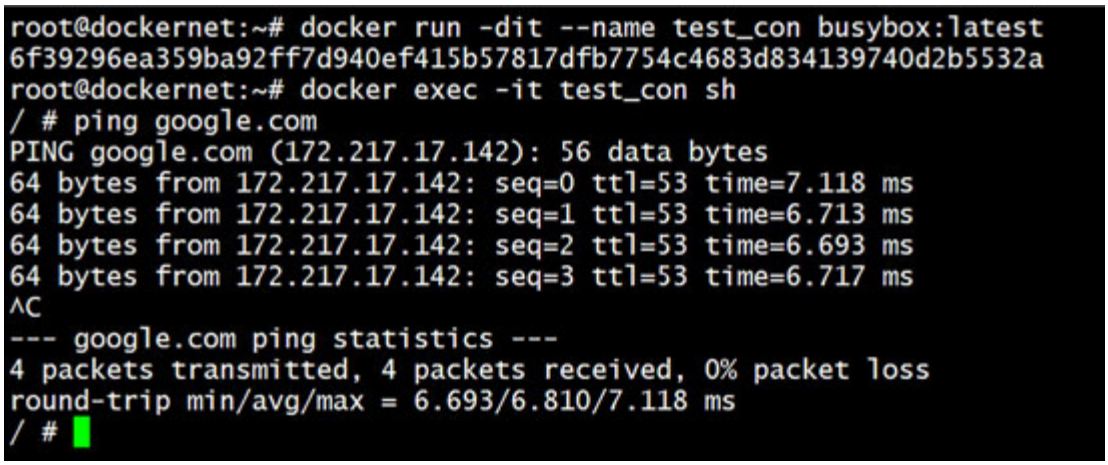



***Figure 6.21***

So, now that we understand how the entire mechanism works let us see the options of mapping ports to containers.

```
docker run -dit --name test1 -P nginx:latest
docker ps
```

See the output in the screenshot in *Figure 6.22*:


***Figure 6.22***

As we can see, the *nginx* application is exposed on port 80 of the container, and the `-P` flag has randomly mapped the port to port `32778` on the host. `-P` flag will randomly map a container port to a port on the host. We can easily check if everything is working fine or not by running the following command and checking the output in *Figure 6.23*:

```
curl localhost:32778
```

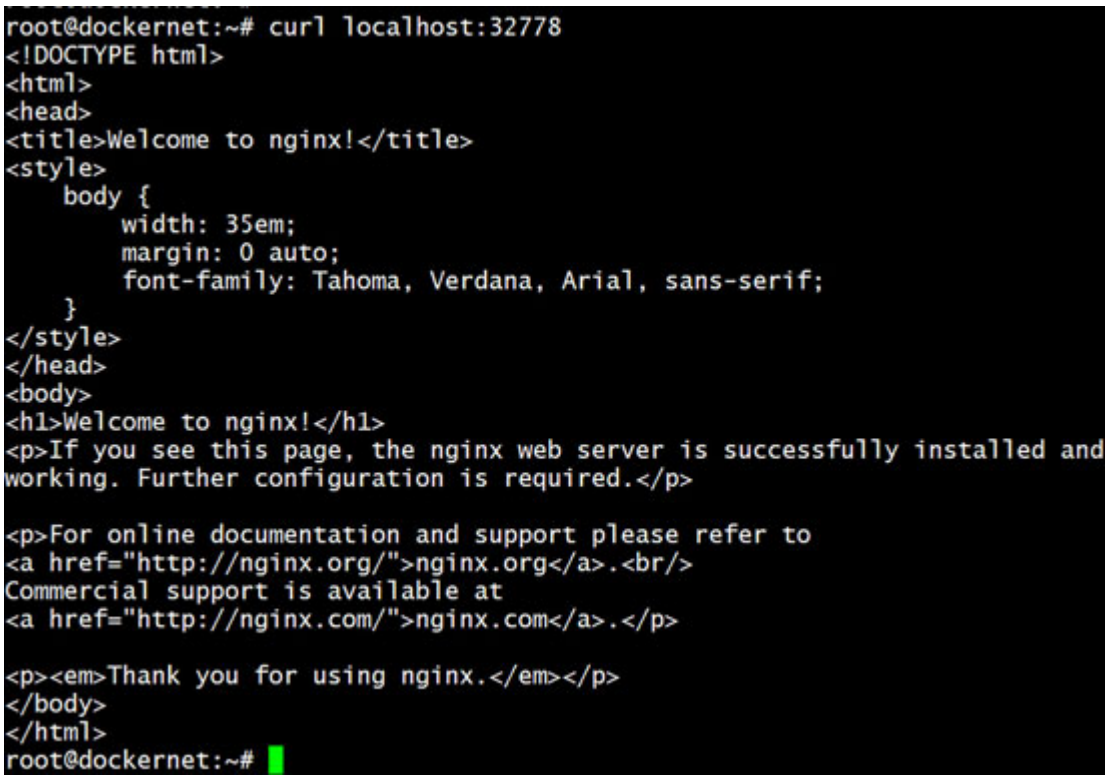



***Figure 6.23***

Now, let us map a specific port on the host to a specific port on the container and check how it all pans out. See the screenshot in *Figure 6.24*:

```
docker run -dit --name test2 -p 8080:80 nginx:latest
```

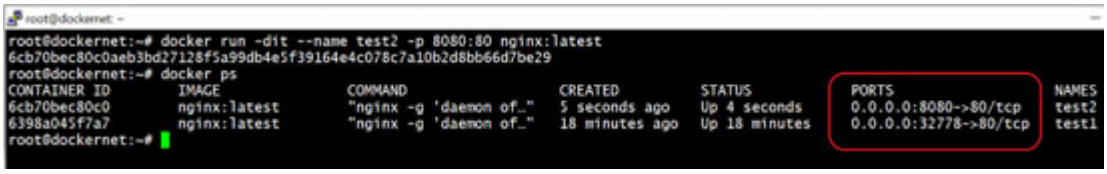



***Figure 6.24***

We can map the ports of containers easily to an available port on the host. Not only that, but we can also map ports with protocols. For example, we can map port 80 on the container to port 8081 for both tcp and udp protocols. Let's see the following example:

```
docker run -dit --name test3 -p 8081:80/tcp -p 8081:80/udp
nginx:latest
```


***Figure 6.25***

## MACVLAN Network

Sometimes there may be a requirement for applications to be connected to the physical network, because such applications may require a physical network to run. The *MACVLAN* network is tailor-made for such situations.

The *MACVLAN* interface is created by attaching it to a physical interface, and it is provided its own MAC address and IP address. There are a few important things to understand when we talk about *MACVLAN*.

1. *MACVLAN* interfaces have to be mapped to a parent interface and will use the host's gateway to access external networks.
2. The *MACVLAN* driver only works on Linux hosts and is not supported on Docker Desktop for Mac, Docker Desktop for Windows, or Docker EE for Windows Server.
3. For using *MACVLAN*, we require at least *version 3.9* of the Linux kernel, but *version 4.0* or higher is recommended.

4. The *MACVLAN* interface needs to be assigned to the same subnet as the parent interface.
5. The host NIC has to be in promiscuous mode.
6. We can define one *MACVLAN* type network per parent interface in the same subnet.
7. We can specify an IP range for the container as well as any auxiliary address that we don't want Docker IP Management system to start allocating for containers.

A *MACVLAN* network can be either in bridge mode or 802.1q trunk bridge mode. The difference is that in the bridge mode, traffic is routed through a physical interface on the host, while in 802.1q trunk bridge mode, a subinterface is created by docker, which is part of the parent interface. The only advantage of using the latter is it allows us to control the network-related stuff at a somewhat more granular level.

## The MACVLAN Bridge Network

Let us create a *MACVLAN* network using `eth0` as our parent interface. To start with, let us a look at the IP address and subnet associated with the eth0 interface. We can do so by running the ifconfig -a eth0 command and checking the output in the screenshot in *Figure 6.26*.

```
ifconfig -a eth0
```

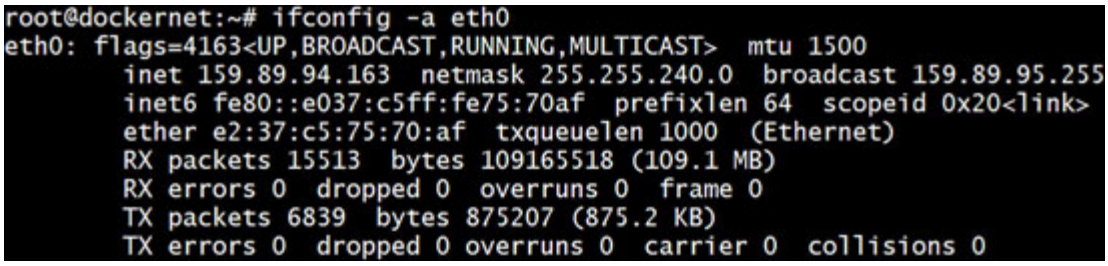



***Figure 6.26***

We can glean the following from the above screenshot:

```
Address: 159.89.94.163
Netmask: 255.255.240.0 = 20
```

```
Network: 159.89.80.0/20
Broadcast: 159.89.95.255
HostMin: 159.89.80.1 <===This can be used as the Gateway
HostMax: 159.89.95.254
```

**You might want to use an online subnet calculator like http://www.subnet-calculator.com/ to get the network, broadcast and other values easily.**

Let us now create the *MACVLAN* network using the eth0 interface and its associated subnet. See the screenshots in *Figures 6.27* and *6.28*.

```
docker network create --driver=macvlan --
subnet=159.89.94.163/20 \
--gateway=159.89.80.1 --ip-range=158.89.80.2/24 \
-o parent=eth0 macvnet
docker network ls
```

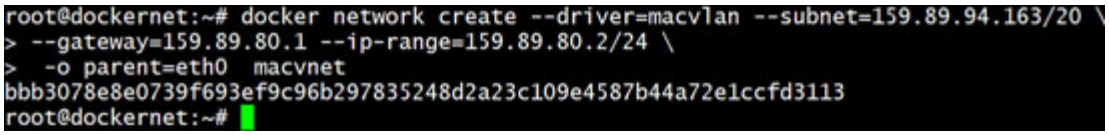



***Figure 6.27***

A quick listing of the networks shows our network has been duly created. See the screenshot in *Figure 6.28*:

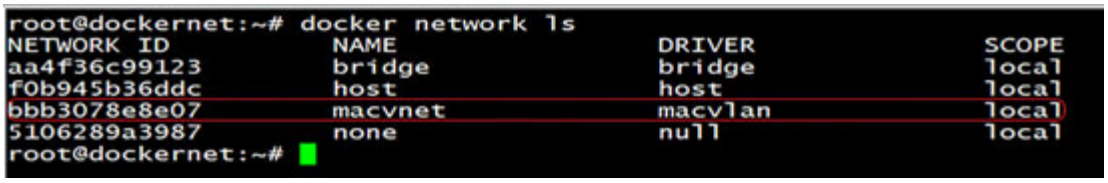



***Figure 6.28***

Before we move ahead, let us get an understanding of the command used:

1. We create a network using the *MACVLAN* driver.
2. We provide the subnet information by mapping it to the `eth0` interface subnet.
3. We provide the gateway information.

4. We provide a range of IPs assignable for the containers attached to this network. Our range is `159.89.80.1-254`. If we didn't put in this flag, docker would have picked up an arbitrary IP address in the subnet of the eth0 interface.
5. The final step in the above command is to connect the eth0 interface as the parent.

Let us drill down into the network and see how things are stacked up down there. See the screenshot in *Figure 6.29*:

```
docker inspect macvnet
```

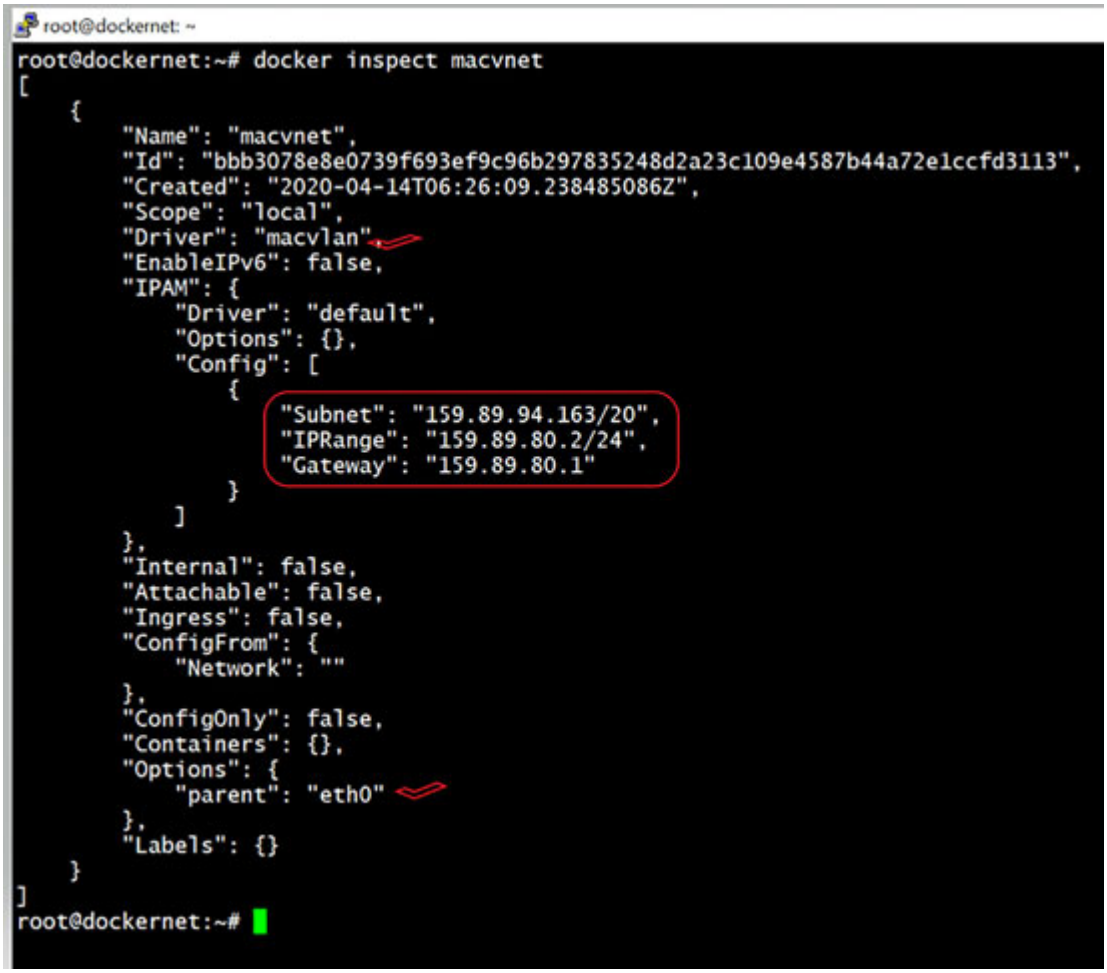



***Figure 6.29***

As expected, we see the driver as *MACVLAN*. We also see the subnet, IP range, and gateway mentioned.

So far, so good. Let us now go ahead and create a container and attach it to our newly created *MACVLAN* network. See the screenshot in *Figure 6.30*:

```
docker run -dit --name mymac --network macvnet busybox:latest
docker exec -it mymac sh
ip addr
ip route
exit
```

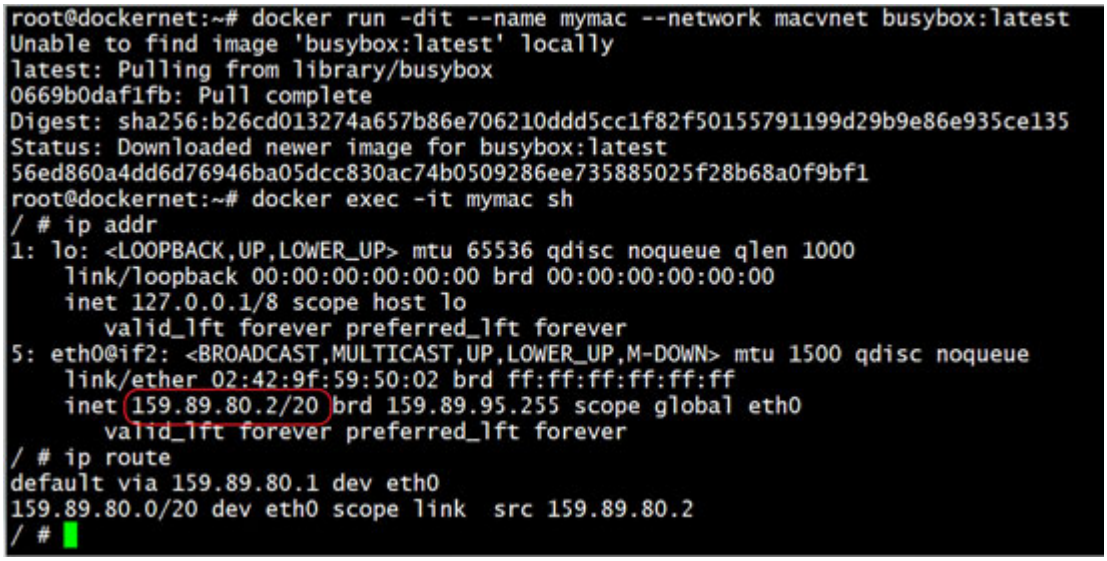



***Figure 6.30***

So, we see quite a bit of information here. The first piece of information is the name of the interface, i.e., `eth0@if2`. We also notice that docker has assigned the first available IP in the range of IPs we had provided. (`159.89.80.1` being the gateway IP). Also, the route information clearly shows that we are using the `eth0` network.

Now, let us look into the container a little deeply. Let us ping the IP address of the container as well as the `eth0` interface and see what happens. Check the screenshot in *Figure 6.31*:

```
ping 159.89.80.2
ping 159.89.94.163
```

```
       valid_lft forever preferred_lft forever
6: eth0@if2: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:9f:59:50:02 brd ff:ff:ff:ff:ff:ff
    inet 159.89.80.2/20 brd 159.89.95.255 scope global eth0
       valid_lft forever preferred_lft forever
/ # ping 159.89.80.2
PING 159.89.80.2 (159.89.80.2): 56 data bytes
64 bytes from 159.89.80.2: seq=0 ttl=64 time=0.062 ms
64 bytes from 159.89.80.2: seq=1 ttl=64 time=0.065 ms
64 bytes from 159.89.80.2: seq=2 ttl=64 time=0.080 ms
^C
--- 159.89.80.2 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.062/0.069/0.080 ms
/ # ping 159.89.94.163
PING 159.89.94.163 (159.89.94.163): 56 data bytes
^C
--- 159.89.94.163 ping statistics ---
3 packets transmitted, 0 packets received, 100% packet loss
/ #
```

***Figure 6.31***

So, while we can ping the IP address of the container, we see that we cannot ping the IP address of the host. This is the default behavior. Not only this, but we cannot even ping the container from the host even though they are both on the same interface. See the following screenshot:

```
root@dockernet:~# ping 159.89.80.2
PING 159.89.80.2 (159.89.80.2) 56(84) bytes of data.
^C
--- 159.89.80.2 ping statistics ---
7 packets transmitted, 0 received, 100% packet loss, time 6149ms

root@dockernet:~#
```

***Figure 6.32***

So, let's take a step back and see where we stand. The following diagram shows that the container `mymac` has its own MAC address, its IP address, and its namespace. The container is plugged into the `eth0` interface, which in turn is connected to the external network, as shown in *Figure 6.33*:

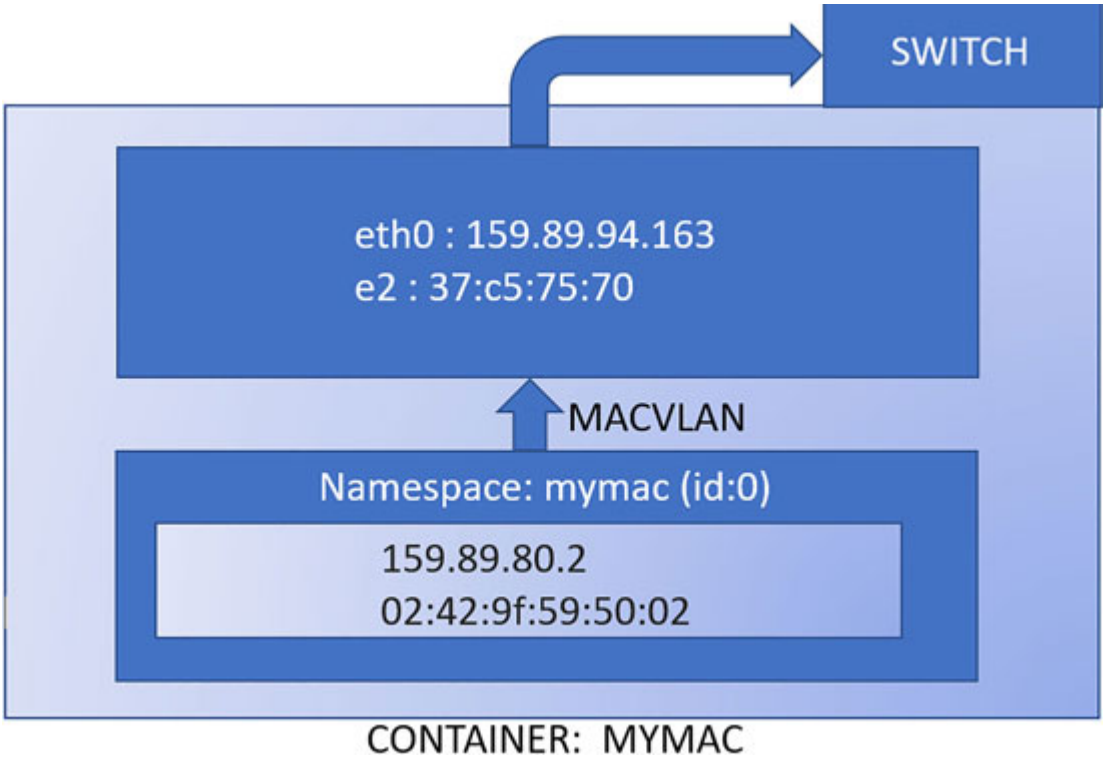



***Figure 6.33***

Getting back to the point of not being able to ping the container from the host or vice-versa, we can get it to work, but before that, let us create another container on the same *MACVLAN* network and see if they can communicate with each other. Let us run the following commands and check the screenshot in *Figure 6.34*:

```
docker run -dit --name mymac --network macvnet busybox:latest
docker exec -it mymac sh
ip addr
ping 159.89.80.2
```

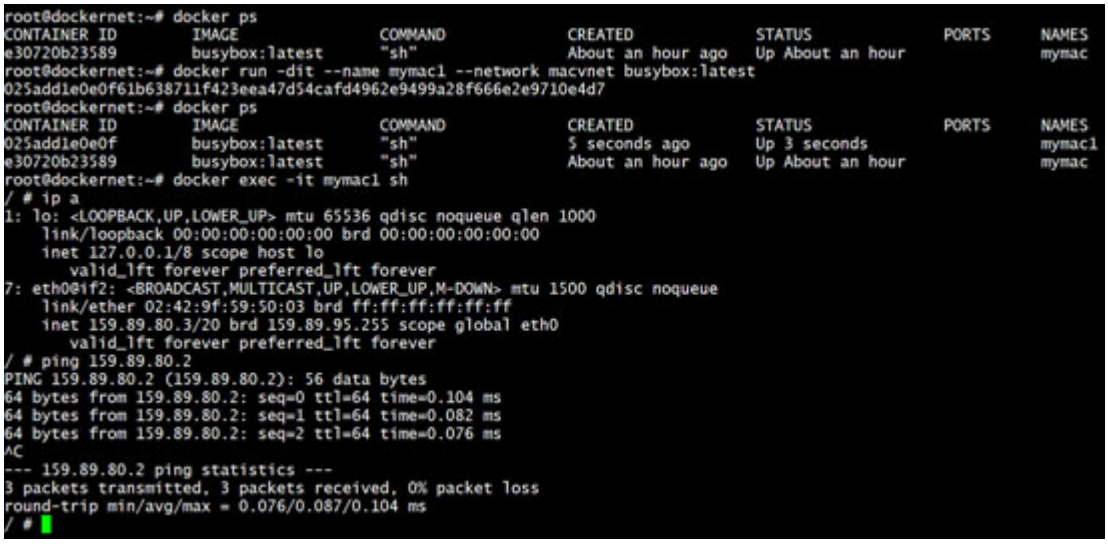



***Figure 6.34***

Well, the containers can communicate with each other as they are connected to the same *MACVLAN* network. The containers are in the *MACVLAN* bridge mode, which mimics the Linux bridge.

Now, let us address the question of pinging the container from the host. For this to work, we need to do the following. See the screenshot in *Figure 6.35*:

1. `ip link add macnet0 link eth0 type macvlan mode bridge`
2. `ip addr add 159.89.80.2/20 dev macnet0`
3. `ip link set macnet0 up`

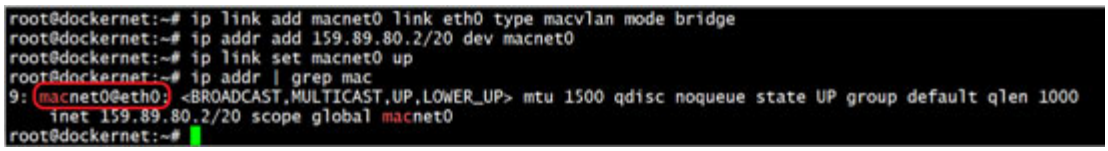



***Figure 6.35***

What we did here was to create a connection from the interface `eth0` to the container, adding the containers' IP address and setting it up. In other words, we have simply established one-way connectivity from the host to the container. Let us now try to ping the container from the host. We expect this to work, and sure enough, it does. See the following screenshot:

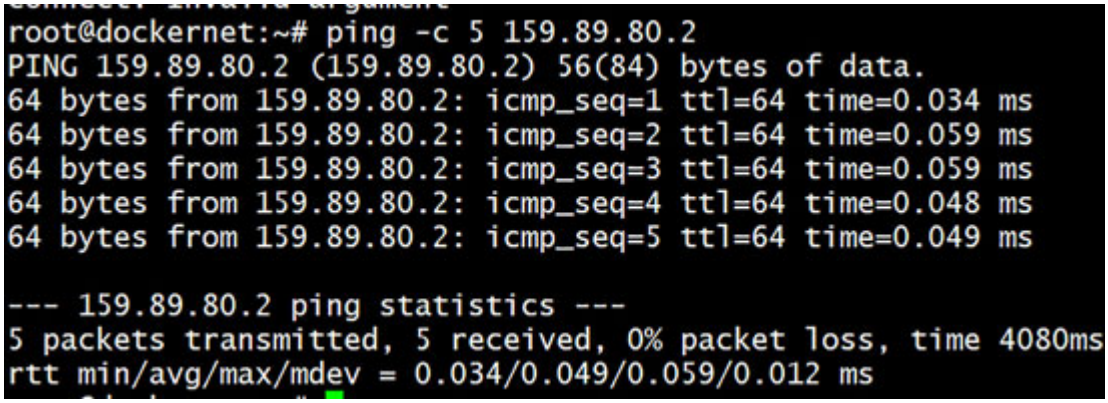



***Figure 6.36***

# The 802.1q Trunk Bridge Network

Let us create a network using the *802.1q* trunk bridge. This time let us use the `eth1` interface as the parent interface. See the screenshot in *Figure 6.37*:

```
docker network create --driver=macvlan --
subnet=10.136.11.242/16 \
--gateway=10.136.0.1 --ip-range=10.136.0.2/24 \
-o parent=eth1.50 macvnet1
docker network ls
```

```
Address: 10.136.11.242
Netmask: 255.255.0.0 = 16
Network: 10.136.0.0/16
Broadcast: 10.136.255.255
HostMin: 10.136.0.1 <====== Gateway
HostMax: 10.136.255.254
```

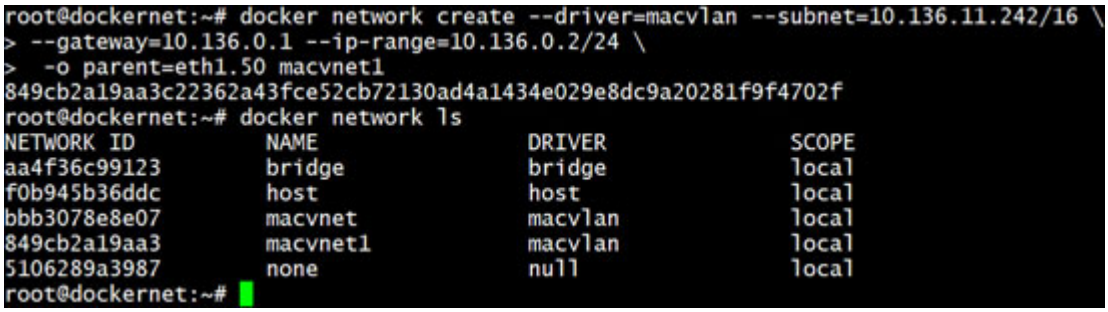



***Figure 6.37***

The next logical step to inspect this network set up. We do so by running the `docker network inspect macvnet1` command and checking the screenshot in *Figure 6.38*:

```
docker network inspect macvnet1
```

We see that the name of the parent is the subinterface eth1.50. Otherwise, it is the same as the normal *MACVLAN* bridge network.

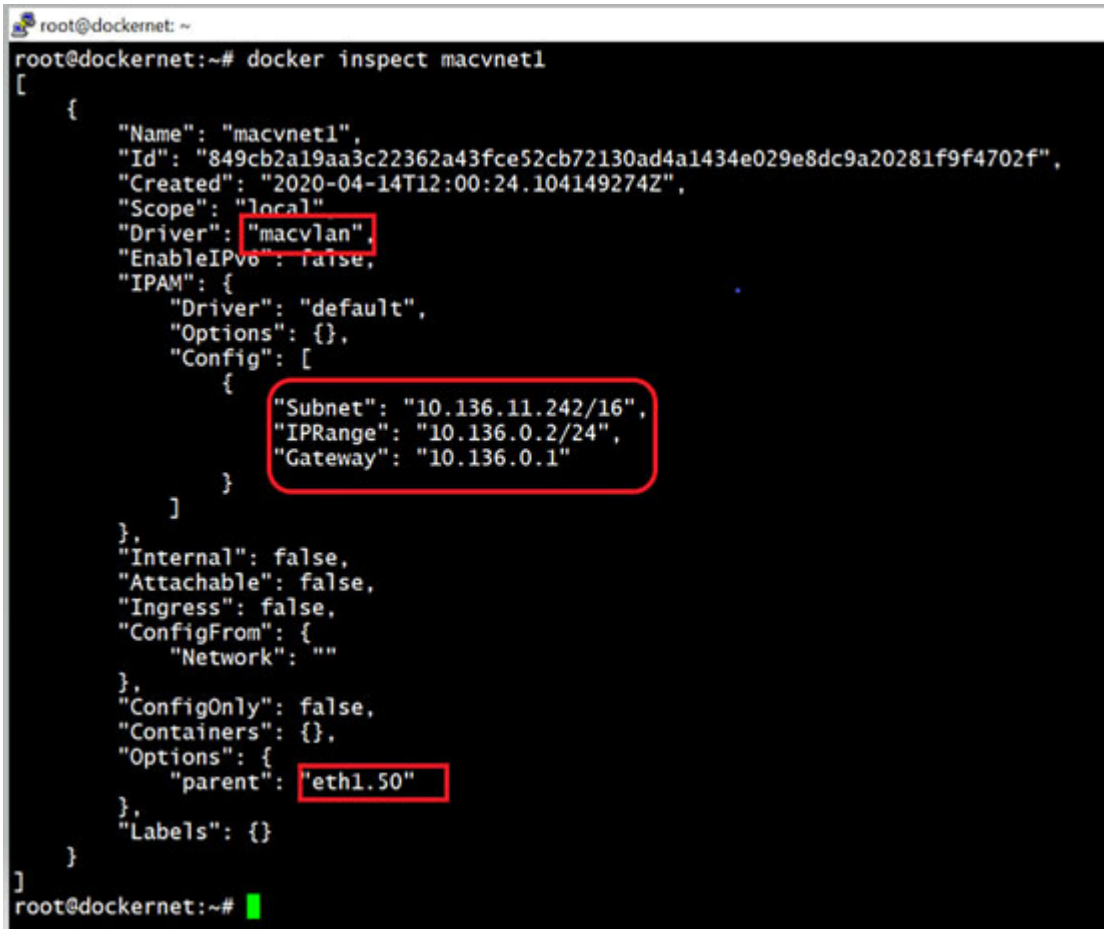



***Figure 6.38***

## Overlay Network

The overlay network driver connects multiple containers over host-specific networks. In other words, overlay networks are a mechanism to build an isolated network on top of whatever existing network there is for the host.

However, installing docker doesn't get an overlay network initiated by default. It's only when we initialize a swarm or join a node to a docker swarm that the overlay network gets created. However, not only do we get an overlay network created, which by the way, is named ingress, but we also get another network by the name of `docker_gwbridge` created as well. Looking at the name of the second network, it is easy to guess that it is a network of type bridge.

*We can connect both individual containers as well as docker swarm services (discussed earlier in Chapter 5) to the overlay network. However, to connect individual containers to an overlay network, we need to create the network with the* attachable *flag.*

To start with, let us create a docker swarm configuration with one manager node and a couple of worker nodes and check the network configuration. See the screenshot in *Figure 6.39*:

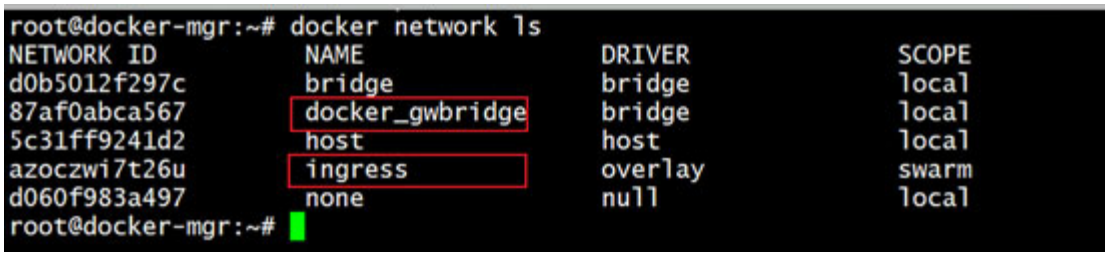



***Figure 6.39***

Sure enough, we see two new networks have joined the party- `docker_gwbridge` of type bridge and ingress of type overlay.

And we see the same output when we run the command from one of the worker nodes. See the output in *Figure 6.40*:

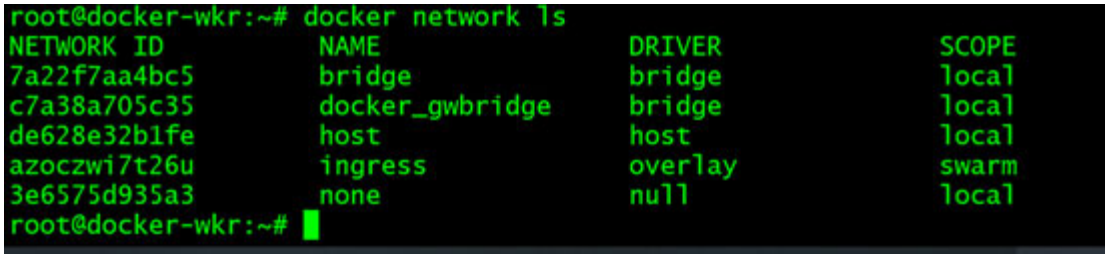



***Figure 6.40***

Let us try to create a container and see what happens when we try to attach it to the ingress network. See the screenshot in *Figure 6.41:*

```
docker run -dit --name test_container --network ingress
busybox:latest
```

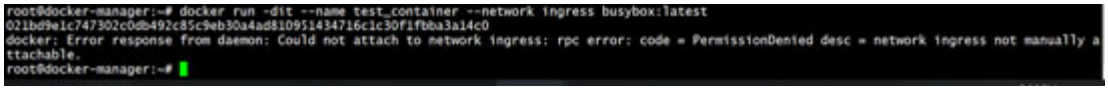



***Figure 6.41***

We cannot create the container and attach it to the ingress network simply because the "*swarm*" scoped default overlay network cannot be used for docker run. Strangely, neither can it be used for docker services? See the following screenshot:


***Figure 6.42***

However, a swarm scoped user-created network allows us to attach a container to it, provided we create the network as an attachable network.

When we delve a little deeper and inspect the ingress network, sure enough, we see that is it scoped only for swarm. Check the screenshot in *Figure 6.43*:

```
docker inspect ingress
```

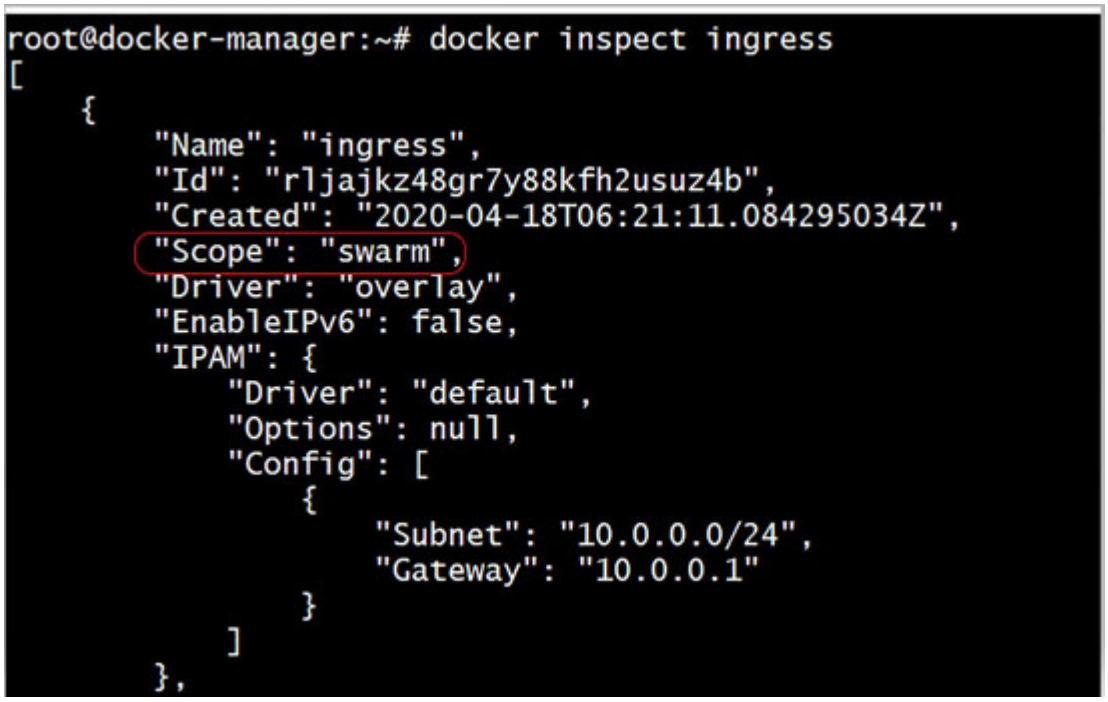



***Figure 6.43***

At this point, let us also look at the namespace of the overlay network. As discussed earlier, our network namespace in docker doesn't lie in the same path as the standard Linux network namespace, and as such, we will not be able to use a Linux command like `ip netns` list to list out the docker network namespaces. Earlier, we had created a soft link to point to the path where the docker network namespace lay and thus was able to use the `ip netns` list command to list out the namespaces.

Let's do it directly this time and check the network namespaces in docker. Check the screenshot in *Figure 6.44*:

```
docker network ls
ls -ln /var/run/docker/netns
```

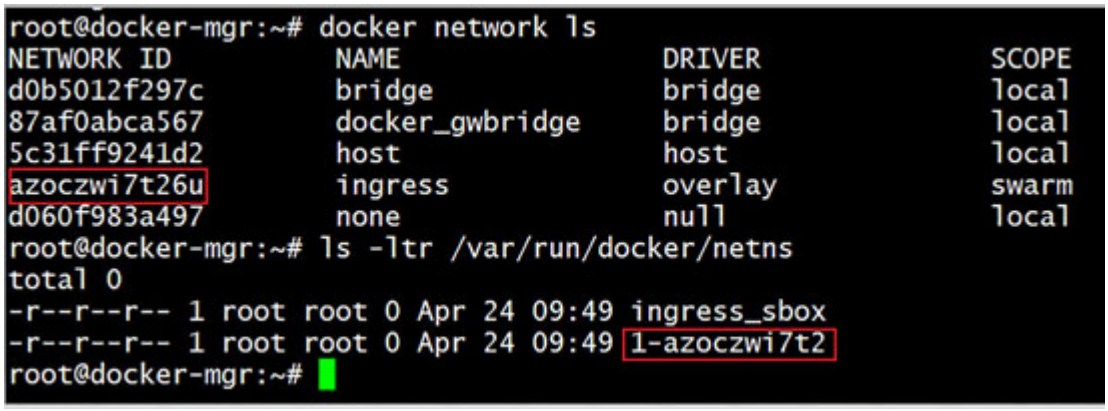



***Figure 6.44***

We can match the namespace name with the network id of the overlay namespace. Additionally, we also see something called an `ingress_sbox`. Now, what is this? This is a hidden container, which has one end hooked onto the host network and the other tied to the overlay network. Its main function is related to service discovery and load balancing.

Moving ahead, let us define an overlay network and make it attachable so that we can run a container and '*attach*' it to the overlay network.

```
docker network create --driver overlay --attachable my-overlay
```

Now, let us check our newly created network and also let us peek at the network namespace created post creation of our my-overlay network in the screenshot in *Figure 6.45*.

```
docker network ls
```

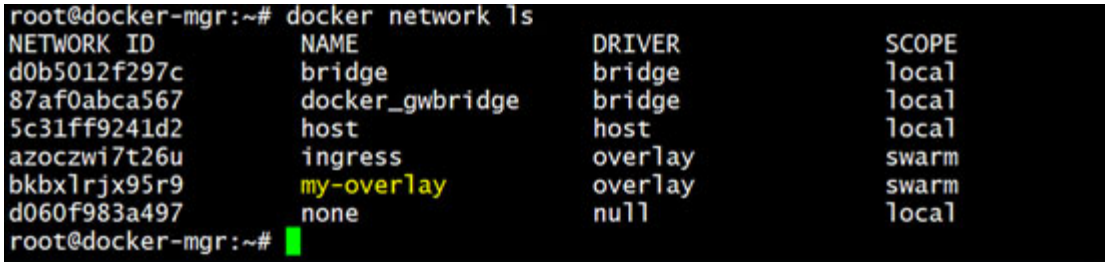



***Figure 6.45***

We try to list out our newly created namespace. See the screenshot in *Figure 6.45*.

```
ls -ltr /var/run/docker/netns
```

```
root@docker-mgr:~# ls -ltr /var/run/docker/netns
total 0
-r--r--r-- 1 root root 0 Apr 24 09:49 ingress_sbox
-r--r--r-- 1 root root 0 Apr 24 09:49 1-azoczwi7t2
```

***Figure 6.46***

Strangely we don't see any new network namespace having been created aligned to our network `my-overlay`. *This is the default behavior, and the namespace gets generated only when we create a container in that network*. This is shown in the following example. See the output in *Figure 6.47*:

```
docker run -dit --name test_container --network my-overlay
busybox:latest
ls -ltr /var/run/docker/netns
docker network ls
```

```
root@docker-mgr:~# docker run -dit --name test_container --network my-overlay busybox:latest
bccb4311355327bdef223b86bb1d3fe37d538dd8ea8152ad2872cc189d264f68
root@docker-mgr:~# ls -ltr /var/run/docker/netns
total 0
-r--r--r-- 1 root root 0 Apr 24 09:49 ingress_sbox
-r--r--r-- 1 root root 0 Apr 24 09:49 1-azoczwi7t2
-r--r--r-- 1 root root 0 Apr 25 07:30 lb_bkbxlrjx9
-r--r--r-- 1 root root 0 Apr 25 07:30 1-bkbxlrjx95
-r--r--r-- 1 root root 0 Apr 25 07:30 6ba280aebfb8
root@docker-mgr:~# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
d0b5012f297c        bridge              bridge              local
87af0abca567        docker_gwbridge     bridge              local
5c31ff9241d2        host                host                local
azoczwi7t26u        ingress             overlay             swarm
bkbxlrjx95r9        my-overlay          overlay             swarm
d060f983a497        none                null                local
root@docker-mgr:~#
```

***Figure 6.47***

So there we are. We can now see the new network namespace created as we have created a container and attached it to our new overlay network named my-overlay. Let's get inside the container and look around. Check out the output in *Figure 6.48*:

```
docker exec -it test_container sh
```


***Figure 6.48***

Interestingly, we notice that there are two interfaces attached to the container. `eth0@if25` and `eth1@if27`.

If we check for the interfaces, we will find *interface 27* on the host, as shown below in *Figure 6.49*. However, we don't see *interface 25*. Don't worry; we will learn and understand this interface later in the chapter.


***Figure 6.49***

One thing becomes clear. There is a VETH connectivity established between our container and `docker_gwbridge`. We can think of the `docker_gwbridge` as mimicking a Linux bridge.

Before we move ahead, let us try to look into `docker_gwbridge`.

If we match the interface seen from inside the container (i.e., `eth1@if27`), *interface 27* is plugged into a bridge, which is named `docker_gwbridge`. Let's peek into the `docker_gwbridge` network by running the following command:

```
docker inspect docker_gwbridge
root@docker-mgr:~# docker inspect docker_gwbridge
[
  {
    "Name": "docker_gwbridge",
```

```
“Id”:
“87af0abca56707273fef6eecc10ba4e38a3a43a75f815f671d6937ca73b1
96a1”,
“Created”: “2020-04-24T09:49:43.034481166Z”,
“Scope”: “local”,
“Driver”: “bridge”,
“EnableIPv6”: false,
“IPAM”: {
 “Driver”: “default”,
 “Options”: null,
 “Config”: [
  {
   “Subnet”: “172.18.0.0/16”,
   “Gateway”: “172.18.0.1”
  }
 ]
},
“Internal”: false,
“Attachable”: false,
“Ingress”: false,
“ConfigFrom”: {
 “Network”: “”
},
“ConfigOnly”: false,
“Containers”: {
 “bccb4311355327bdef223b86bb1d3fe37d538dd8ea8152ad2872cc189d
 264f68”: {
  “Name”: “gateway_6ba280aebfb8”,
  “EndpointID”:
  “a31e09d34cc43afc82c494f9553c2e37327058cb4f7bfdeec888ae0c3
  eab7ae6”,
  “MacAddress”: “02:42:ac:12:00:03”,
  “IPv4Address”: “172.18.0.3/16”,
  “IPv6Address”: “”
 },
 “ingress-sbox”: {
  “Name”: “gateway_ingress-sbox”,
```

```
      "EndpointID":
      "436d9ace432c4be9e8ed8f6e73ecf40c4194c160081013c7def6a4916
      2be3052",
      "MacAddress": "02:42:ac:12:00:02",
      "IPv4Address": "172.18.0.2/16",
      "IPv6Address": ""
     }
    },
    "Options": {
     "com.docker.network.bridge.enable_icc": "false",
     "com.docker.network.bridge.enable_ip_masquerade": "true",
     "com.docker.network.bridge.name": "docker_gwbridge"
    },
    "Labels": {}
  }
]
```

If we see carefully, we will notice that the `docker_gwbridge` has quite a bit of information about our container (IP address, MAC address, etc.), and if we scroll down, we will also see that the com.docker.network.bridge.enable_icc is set to "*false*".

This setting implies that it prevents containers on the same bridge from being able to communicate with each other. But why on earth would one want to do that? The fact is that since the containers are connected through the overlay network, the containers use this overlay network to communicate with one another.

But then this begs the question about the purpose of `docker_gwbridge` being there. The `docker_gwbridge` is there because of a very specific purpose. The containers on an overlay network can communicate with one another perfectly, but unfortunately, they are not able to talk to the outside world at all! And that is precisely where `docker_gwbridge` comes into the picture. Let's try to visualize the architecture, as shown in *Figure 6.50*:

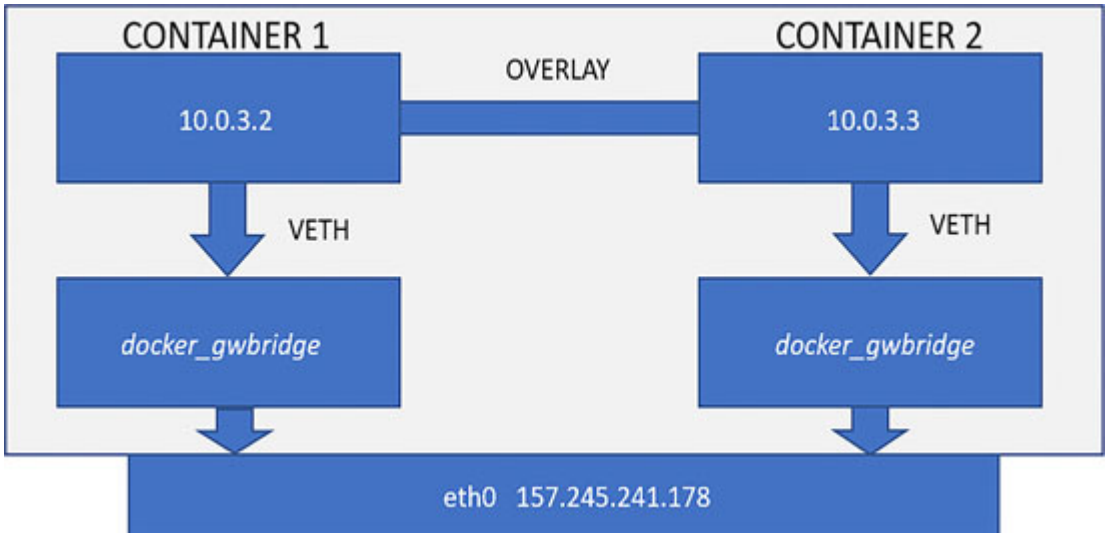



***Figure 6.50***

The two containers talk to each other using the overlay network, as shown in *Figure 6.50*. Each container has two interfaces-one for the overlay communication and the other for connecting to the `docker_gwbridge`. Although containers connect to the `docker_gwbridge` network, it is as if each container has its '*own slice*' of the `docker_gwbridge`, and the sole purpose of `docker_gwbridge` is to allow containers to communicate with the outside world. If we look closely at the output of the network inspect `docker_gwbridge` command, we will see that "`com.docker.network.bridge.enable_ip_masquerade`" is set to "*true*", meaning thereby the IP addresses of the containers can communicate with the outside world "*masquerading*" as the host in which the container resides.

Moving ahead, let us try to understand how the overlay network functions. What goes on under the hood?

The first thing we do is test whether the container on our host system 1 (which is the swarm manager) is ably to ping (i.e., communicate) with the container on host system 2 (which is a worker node) and vice-versa.

The setup on host system1 is shown in the following *Figure 6.51*:

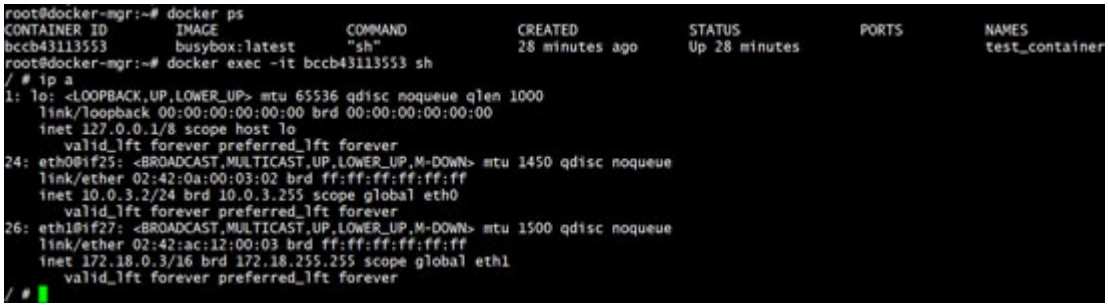



***Figure 6.51***

And the setup on host system 2 is shown in the following *Figure 6.52*:

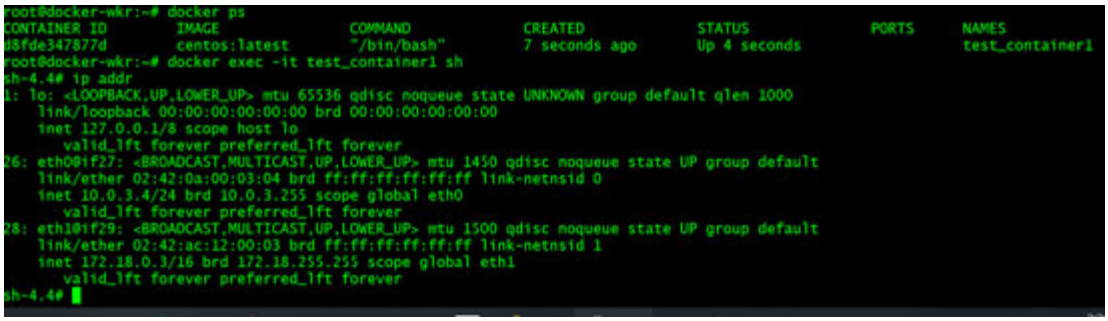



***Figure 6.52***

Let's attempt a ping from the host system 1 to the host system 2. The container on the host system 1 has an IP address of `10.0.3.2`, while that on the host system 2 has an IP address of `10.0.3.4`. See the output in the screenshots in *Figures 6.53* and *6.54*.

```
docker exec -it test_container sh
ping 10.0.3.4
```

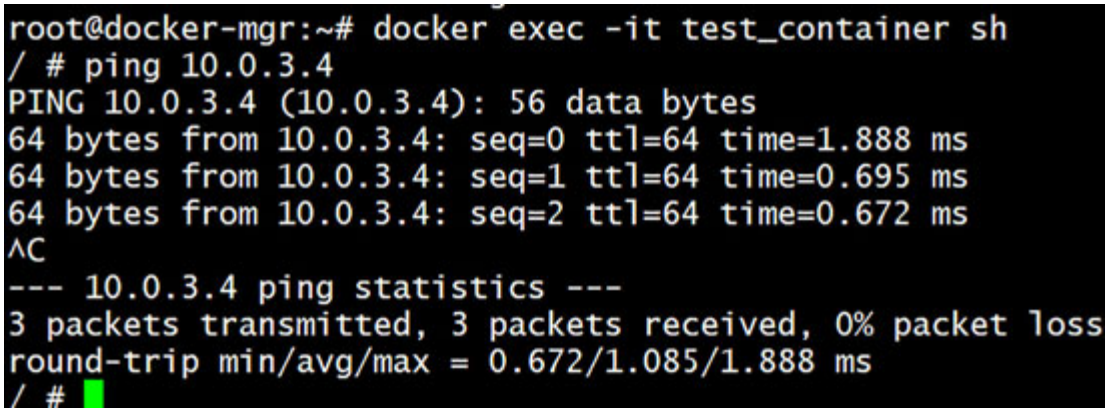



***Figure 6.53***

```
docker exec -it test_container1 sh
ping 10.0.3.2
```

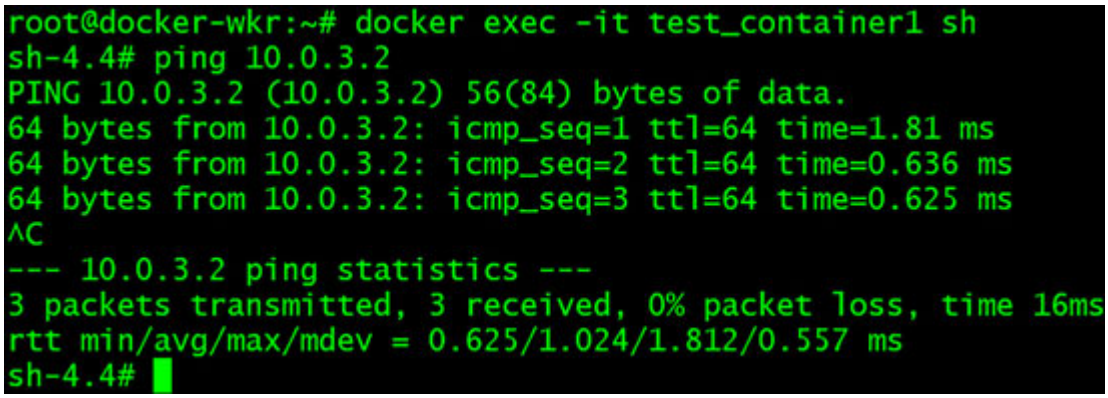



***Figure 6.54***

We can ping from one container to the other without any issues. That's good. But what goes on beneath the hood. How is overlay able to communicate between the containers?

Before we delve into those, let's take a step back to see where we stand.

When we looked inside the container, we saw we had two interfaces apart from the loopback. If we take host system 1 as an example, we will see:

1. `eth0` with an IP address of `10.0.3.2/24` which is on the overlay network, and
2. `eth1` with an IP address of `172.18.0.3/16` connected to the `docker_gwbridge`

And, as we discussed above, one end of the VETH connectivity in `eth1` is in the container, and the other end of it is in the `docker_gwbridge`.

So, let us now focus on the `eth0` interface. We find one end of the VETH in the container, as shown in the following *Figure 6.55*:

```
root@docker-mgr:~# docker exec -it test_container sh
/ # ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
24: eth0@if25: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdisc noqueue
    link/ether 02:42:0a:00:03:02 brd ff:ff:ff:ff:ff:ff
    inet 10.0.3.2/24 brd 10.0.3.255 scope global eth0
       valid_lft forever preferred_lft forever
26: eth1@if27: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:12:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.3/16 brd 172.18.255.255 scope global eth1
       valid_lft forever preferred_lft forever
/ #
```

***Figure 6.55***

But where is the other end of it? We are looking for interface 25. Let us check it in the host system. (see *Figure 6.56*). But we don't see it there either. So, where is it?

```
root@docker-mgr:~# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether d6:36:18:79:8b:11 brd ff:ff:ff:ff:ff:ff
    inet 167.71.113.156/20 brd 167.71.127.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet 10.46.0.6/16 brd 10.46.255.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::d436:18ff:fe79:8b11/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 2e:9b:57:c6:61:92 brd ff:ff:ff:ff:ff:ff
    inet 10.138.218.179/16 brd 10.138.255.255 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::2c9b:57ff:fec6:6192/64 scope link
       valid_lft forever preferred_lft forever
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:5f:a5:6d:e3 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:5fff:fea5:6de3/64 scope link
       valid_lft forever preferred_lft forever
9: docker_gwbridge: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:62:07:e9:62 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.1/16 brd 172.18.255.255 scope global docker_gwbridge
       valid_lft forever preferred_lft forever
    inet6 fe80::42:62ff:fe07:e962/64 scope link
       valid_lft forever preferred_lft forever
11: vethdd9fff20if10: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker_gwbridge state UP group default
    link/ether 22:87:98:22:16:b1 brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet6 fe80::2087:98ff:fe22:16b1/64 scope link
       valid_lft forever preferred_lft forever
27: veth645f9030if26: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker_gwbridge state UP group default
    link/ether fa:ae:cd:8f:cc:a6 brd ff:ff:ff:ff:ff:ff link-netnsid 4
    inet6 fe80::f8ae:cdff:fe8f:cca6/64 scope link
       valid_lft forever preferred_lft forever
root@docker-mgr:~#
```

***Figure 6.56***

The other end of the VETH lies in the overlay namespace and not in the host system or elsewhere. Earlier, we had checked the entries in our network namespace. Let us revisit it. Check the screenshot in *Figure 6.57*:

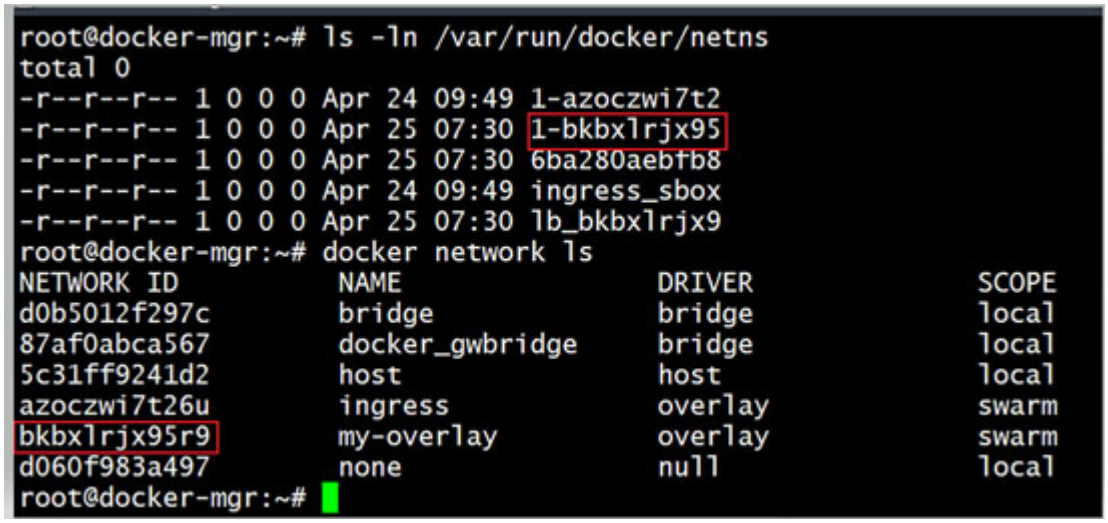



***Figure 6.57***

It is easy to see the match between our network `my-overlay` and the network namespace related to it. (Remember a network namespace will have a notation similar to 1-xxxxx).

Now, let us run the following two commands to check whether the other end of the *VETH* of `eth0` lies here or not. The first command associates a variable `overlay_namespace` with the network namespace of the overlay network we are interested in. The second command uses the `nsenter` command to check the interfaces in that namespace.

```
overlay_namespace=/var/run/docker/netns/1-bkbxlrjx95
nsenter --net=$overlay_namespace ip link show
```

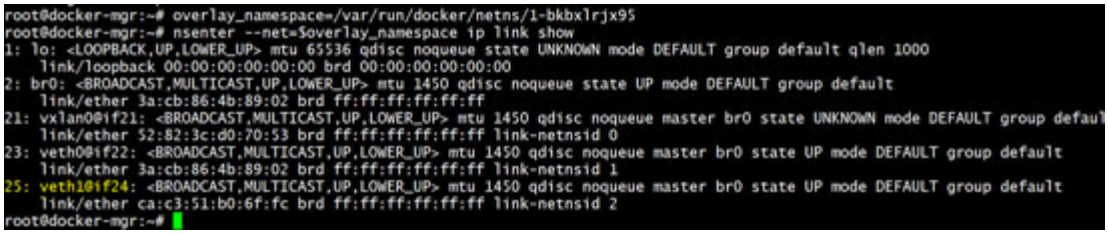



***Figure 6.58***

And sure enough, we see *interface 25* plugged into this namespace. Let us now visualize our understanding thus far.

The `eth1` interface is hooked into a `gwbridge` through a VETH connection, and the `gwbridge` routes traffic through the `eth0` interface

on the host and allows connectivity to the outside world. The `eth0` interface in the containers, on the other hand, has one end of a VETH connectivity inside the container while the other end is plugged into the namespace to which the containers are attached to. The following diagram demonstrates this..

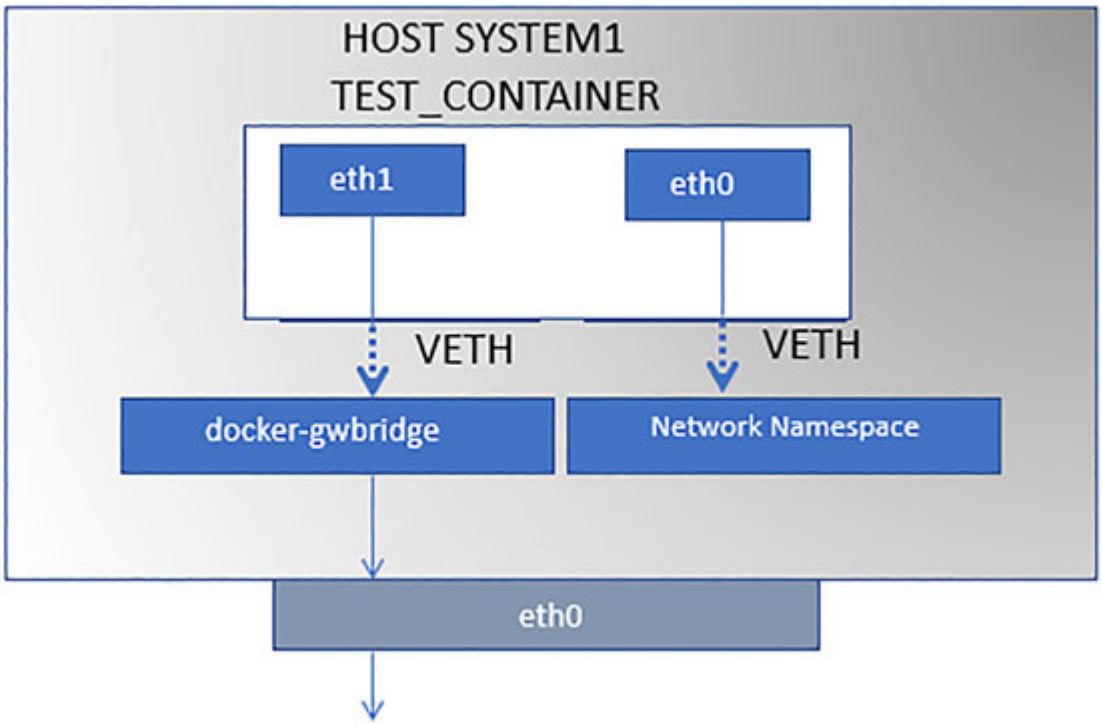



***Figure 6.59***

Okay, moving ahead, when we check the output of the command in *Figure 6.56*, we note an interface vxlan. And it is *VXLAN* that provides the bells and whistles for the container to container communication. So, what exactly is *VXLAN*?

It is a virtualization technology that boosts scalability and optimizes cloud deployments. *VXLAN* basically uses a tunneling technology to encapsulates Layer 2 Ethernet frames in Layer 3 UDP packets. Effectively, this provides us a mechanism to create virtualized Layer 2 subnets that ride on Layer 3 physical networks.

Each Layer 2 subnet is uniquely identified by a VXLAN network identifier (VNI) that segments traffic. The encapsulation and subsequent decapsulation are done by an entity known as VTEP(VXLAN Tunnel Endpoint).

*VXLAN* uses the MAC-in-UDP packet encapsulation mode. The VXLAN header, UDP header, outer IP header, and outer MAC header are added to the original data frame in the sequence.

Let's see how a *VXLAN* packet looks like. The actual payload is only the *Original L2* frame.


***Figure 6.60***

The outer *MAC Header* and *Outer IP Header* are used for the host to host communication. At the same time, the L2 frame is encapsulated in an *Outer UDP Header* with *VXLAN Header* holding key metadata information relating to *VXLAN port*, *UDP length*, and so on.

Now that we understand what a VXLAN packet looks like let us put things in perspective.

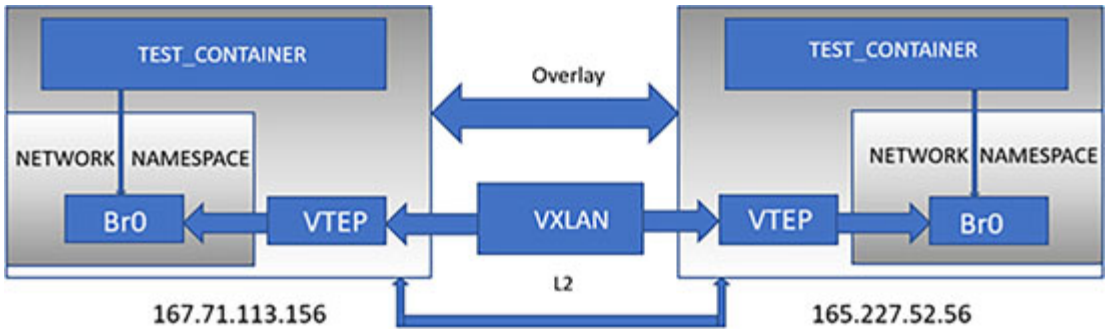



***Figure 6.61***

Each Node has its network namespace, and the container's `eth0` interface is hooked into the network namespace, or `Br0` interface inside the network namespace to be more precise. We can think of `Br0` as a virtual switch. One end of the *VTEP* is also hooked into Br0, while the other end is connected to the *VXLAN* tunnel. The *VXLAN* tunnel rides on the layer 3 undelay networks and is bound to IP addresses of the nodes and the *UDP port 4789*.

To test whether *VXLAN* is being used or not, let us ping container `test_container1` in host system 2 from the host system 1, and on host system 2 let us capture some `tcpdump` data and analyze it. As

we are aware, docker swarm uses *port 4789* for UDP communication for overlay network traffic; we will capture the *tcpdump* for interface *eth0* and *port 4789*. Check out the screenshots in *Figures 6.60* and *6.61*:

```
From Host System 1
ping 10.0.3.4

From Host System 2
tcpdump -i eth0 port 4789
```

Host System 1

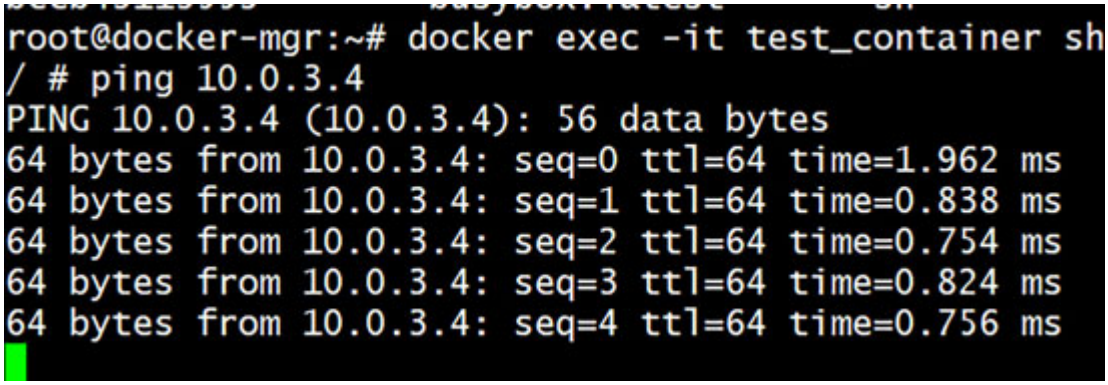



***Figure 6.62***

Host System 2


***Figure 6.63***

It is clear that VXLAN is being used to route the traffic. Additionally, we also see that traffic initiated from IP `167.71.113.156` towards the host `docker-wkr`, and it was being routed through *port 4789*, enabling container `10.0.3.2` to communicate with container `10.0.3.4`.

At this point, we still don't know how containers on each host can map the IP addresses to the MAC address and forward it to the correct server. So, let us check how this mapping takes place. For this, let us run the `nsenter` command and use appropriate flags in the `tcpdump` to check the output. We ping host system 2 from the host system 1 and then capture output from the host system 2. See the screenshot in the following *Figure 6.62*:

```
nsenter --net=$overlay_namespace tcpdump -vv -epni any arp
```

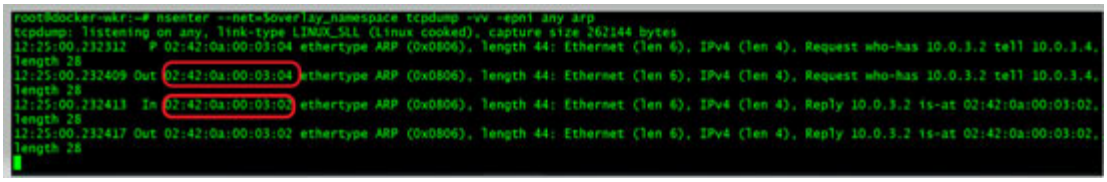



***Figure 6.64***

Without losing our way in the maze of details thrown up in the output of the above command, let us just concentrate on the two MAC addresses that we can see here. If we check in our containers, we will find these MAC addresses (shown below). So, it is now evident to us that the MAC addresses get added in the overlay namespace, and the VXLAN acts as a proxy for answering the ARP related queries. Re: *Figures 6.63* and *6.64*

```
docker exec -it test_container sh
```

```
root@docker-mgr:~# docker exec -it test_container sh
/ # ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
24: eth0@if25: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdisc noqueue
    link/ether 02:42:0a:00:03:02 brd ff:ff:ff:ff:ff:ff
    inet 10.0.3.2/24 brd 10.0.3.255 scope global eth0
       valid_lft forever preferred_lft forever
26: eth1@if27: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:12:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.3/16 brd 172.18.255.255 scope global eth1
       valid_lft forever preferred_lft forever
/ #
```

***Figure 6.65***

`docker exec -it test_container1 sh`

```
root@docker-wkr:~# docker exec -it test_container1 sh
sh-4.4# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
26: eth0@if27: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether 02:42:0a:00:03:04 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.0.3.4/24 brd 10.0.3.255 scope global eth0
       valid_lft forever preferred_lft forever
28: eth1@if29: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:12:00:03 brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet 172.18.0.3/16 brd 172.18.255.255 scope global eth1
       valid_lft forever preferred_lft forever
sh-4.4#
```

***Figure 6.66***

Let us look at one last thing before we wrap up. We would like to see whether the MAC address is hardcoded in the overlay namespace when we create a container. For that, let us run the following on host system 1 and check the output in *Figure 6.65*:

`nsenter --net=$overlay_namespace ip neighbor show`

```
root@docker-mgr:~# overlay_namespace=/var/run/docker/netns/1-bkbxlrjx95
root@docker-mgr:~# nsenter --net=$overlay_namespace ip neighbor show
10.0.3.5 dev vxlan0 lladdr 02:42:0a:00:03:05 PERMANENT
10.0.3.4 dev vxlan0 lladdr 02:42:0a:00:03:04 PERMANENT
root@docker-mgr:~#
root@docker-mgr:~#
```

***Figure 6.67***

We see a couple of entries here. Now, let us create a container and see if we find an entry added to the overlay namespace. From host

system 2, let us run the following, creating a container, as shown in *Figure 6.66*.

```
docker run -dit --name test_container2 --net my-overlay
redis:latest
```

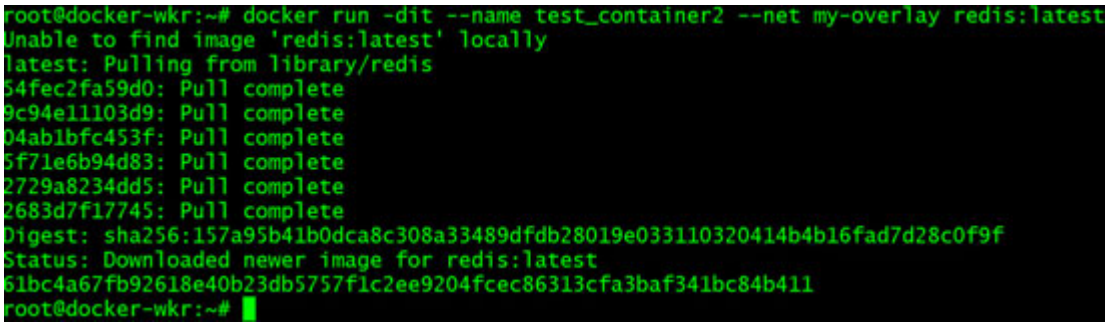



***Figure 6.68***

Let us check from the host system 1 once again. And sure enough, we see a new entry having been added as can be evidenced by the output of the screenshot in *Figure 6.67*:

```
nsenter --net=$overlay_namespace ip neighbor show
```

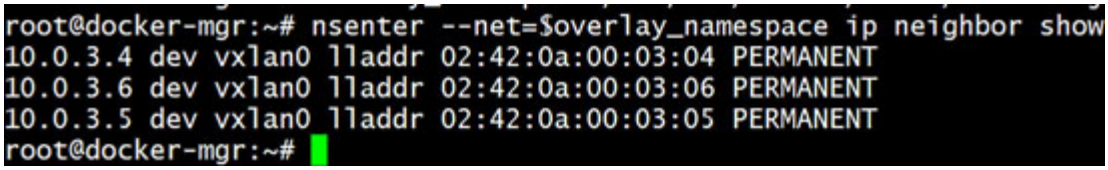



***Figure 6.69***

There we see the additional entry. IP address `10.0.3.6` has been added with the MAC address `02:42:0a:00:03:06`. We can check the particulars by getting into the newly created container and running the following set of commands and checking the output in *Figure 6.68*:

```
docker exec -it test_container sh
ifconfig -a
```

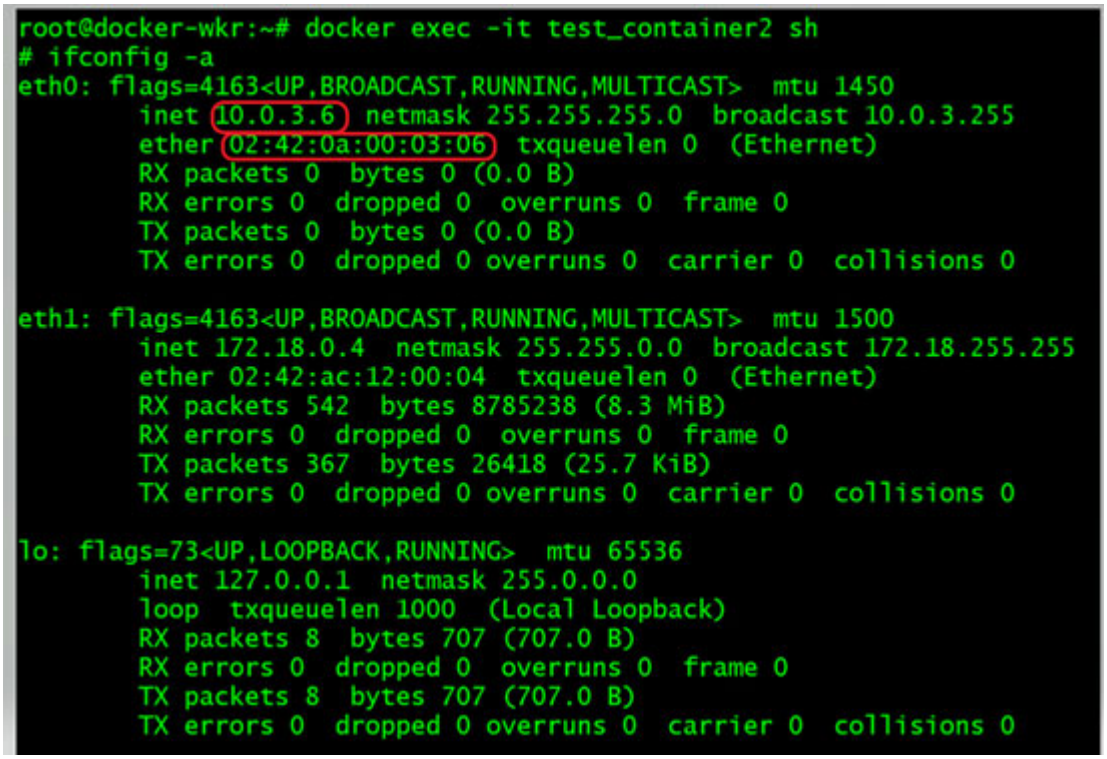



***Figure 6.70***

Well, so that kind of validates what we already guessed. For every container we create, there will be an entry in the overlay network namespace that will be used to map an IP address to the MAC address, and that is how everything works out.

## Points to Remember

- A docker installation comes with three default networks: *Bridge*, *Host,* and *None*.
- If we install a docker swarm, we get two new networks: an overlay network named `ingress` and a bridge network named `docker_gwbridge`.
- The default docker bridge network is built on top of a Linux bridge and is named `docker0`.
- A container can communicate directly with the outside world, but for the outside world to communicate with Docker, we have to go through the host server interface-usually `eth0`.

- We can easily map ports from the server to the container. However, the same port on the host cannot be mapped again to another container.
- We can map ports using either tcp or udp.
- If we use the Host network, then a container is mapped to the network used by the host and doesn't have its own network.
- When we want a container to be completely insulated from the outside world, we use the *None* network.
- Sometimes there may be a requirement for applications to be connected to the physical network, because such applications may require a physical network to run. The *MACVLAN* network is used in such situations.
- A docker swarm uses the overlay network named ingress to communicate with other containers, while it uses the `docker_gwbridge` to talk to the outside world.

## Multiple Choice Questions

**Choose the most appropriate answer:**

1. The Docker Container has a docker0 network of the type.
   a. Bridge
   b. None
   c. Overlay
   d. None of the above
2. In VXLAN
   a. Layer 2 subnets ride on Layer 3 physical networks
   b. Layer 3 subnets ride on Layer 2 physical networks
   c. VXLAN is a Layer 7 configuration
   d. None of the above
3. The Docker docker0 bridge
   a. Allows container to container communication using either the IP addresses or the names of the containers

b. Allows container to container communication using only names of the containers
c. Allows container to container communication using only the IP addresses of the containers
d. None of the above

4. The VXLAN tunnel operates on
   a. tcp port 2377
   b. udp port 2377
   c. tcp port 4789
   d. udp port 4789
5. Docker creates the network namespace in the location
   a. /var/run/docker/
   b. /var/run/netns
   c. /var/run/docker/netns
   d. None of the above
6. For a MACVLAN network
   a. The host NIC has to be in a non-promiscuous mode
   b. The host NIC has to be in promiscuous mode
   c. It doesn't matter in which mode the NIC is
   d. None of the above
7. While using an existing container's namespace
   a. The new container will not have its own filesystem
   b. The new container will neither have its own network nor its own filesystem.
   c. The two containers can have separate ports
   d. The new container will have its own filesystem but will share the network stack with the existing container
8. The -P flag maps
   a. A port on the container with a random port on the host

b. A port on the host with a random port on the container
c. Random ports on both container and host will be mapped to each other
d. None of the above is true

9. The MAC address
    a. Is hardcoded in the host system when a container is created or added under a swarm configuration
    b. Is hardcoded in the overlay namespace when a container is created or added under a swarm configuration
    c. There is no MAC address configuration in the overlay namespace.
    d. The MAC address is stored as an environment variable in the overlay network namespace.
10. Br0 may be a considered
    a. A physical switch
    b. A physical interface
    c. A virtual namespace
    d. A virtual switch

## Answers

1. a
2. a
3. c
4. d
5. c
6. b
7. d
8. a
9. b
10. d

## Questions

1. Name the different types of networks created through a default docker installation.
2. Explain the concept of a network namespace from the perspective of a container.
3. Which are the new networks created when the containers are put in Docker Swarm mode.
4. Explain the concept of the docker0 bridge.
5. Explain in detail how port-mapping can be done.
6. What do you understand by a MACVALAN network? What are the two modes of the MACVALAN network?
7. Explain the concept of VETH from the perspective of Linux namespace.
8. Explain in detail how the overlay network works.
9. What do you understand by VXLAN?
10. How can we set up the MACVLAN network so that we can ping the network from the host?

## Key Terms

- Bridge network
- Host network
- None network
- MACVLAN network
- Overlay network
- Docker_gwbridge
- Ethernet interface
- VETH
- Docker Swarm
- Port Mapping
- Network Namespaces

- 802.1q Trunk Bridge Network
- Br0 virtual switch
- VXLAN
- VTEP

# CHAPTER 7

# Docker Security-1

## Introduction

Docker security can be divided into two broad areas: Security provided by Linux (as this book deals with Docker on Linux, we take a deep look into the options available for security from the operating system level for Docker), and the native security for Docker containers leveraging the features provided by *Docker Enterprise Edition*. In this section, we will look at the security features provided by Linux.

## Structure

- Kernel Namespaces
- Control Groups
- Capabilities
- Mandatory Access Control-SELinux and AppArmor
- Seccomp

## Objective

After studying this chapter, you ought to able to understand deeply the security features inherently available in Linux and how it is leveraged by Docker to ensure that the container ecosystem is, in general, safe. You will be able to make informed choices on what security configurations to use to secure your containers and how to ensure that you can take advantage of the native security of the Linux operating system.

## Kernel Namespaces

Let's start with understanding the concept of a namespace. A namespace is simply a mechanism for isolation. The isolation is provided by organizing objects in such a way that objects or groups of objects are kept together and identified by a name. The namespace makes sure that the identifiers placed in it are unique and can be identified as such.

Kernel namespaces are perhaps the most fundamental of security features that are there in Docker. We can visualize kernel namespaces as allowing each Docker container to have its own '*share of the operating system*'. This literally lets us run hundreds of containers on the same host without having any sort of '*conflict'* at the operating system level. For example, we can have containers running on the same port or use the same set of configuration files without any problems whatsoever. This is all possible because of the isolation provided by different namespaces. Docker on Linux currently uses the following set of kernel namespaces:

- Process ID (pid)
- Network (net)
- Filesystem/Mount (mnt)
- Inter-Process Communication (ipc)
- User (usr)
- UTS (uts)

**Process ID**: Process ID stands for process identifier. An identification number is automatically generated for a process when it starts running on the Linux system. It is unique. Docker uses the PID namespace to create a separate process tree for each container.

Let's create a couple of containers and see how it work. We will create containers for the Couchbase database, which is a NOSQL database run for business-critical applications. Then, we will get inside the containers and have a look at the processes running. Okay, let's first create the containers and see the output in *Figure 7.1*:

```
docker run -dit --name test couchbase
```

```
docker run -dit --name test1 couchbase
docker container ls -a
```

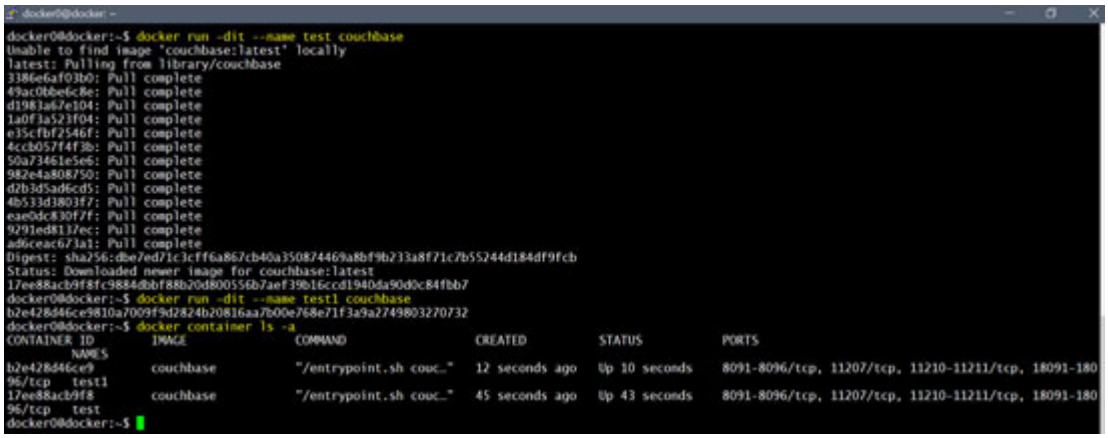



***Figure 7.1***

Next, let us get inside the containers and run a few commands. See the following commands and the output in *Figure 7.2*:

```
docker exec -it test sh
ps
docker exec -it test1 sh
ps
```

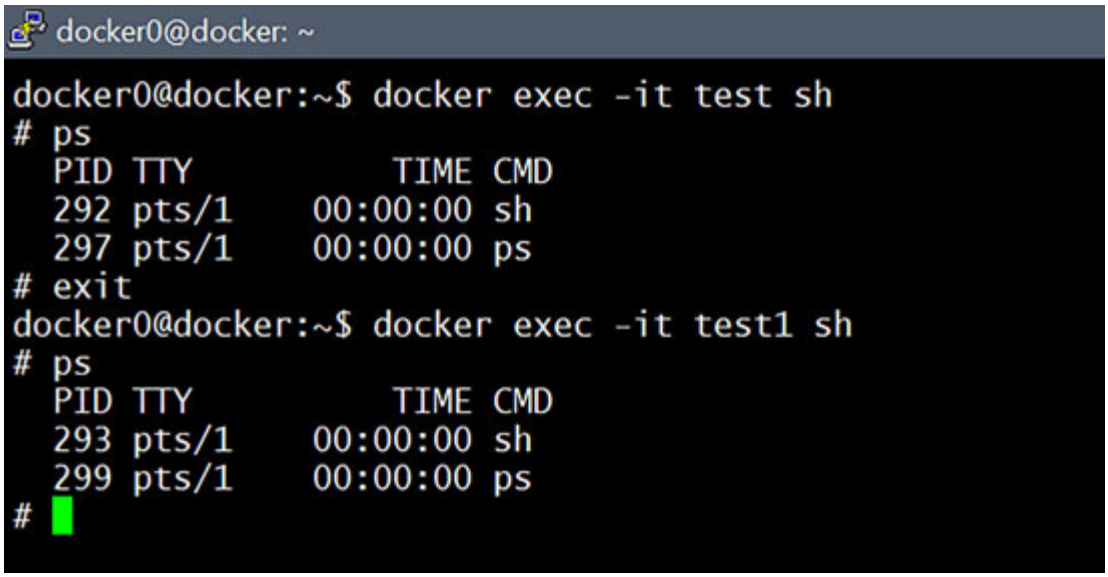



***Figure 7.2***

We see the exact same processes running for each of the containers, but they are running with different PIDs. This is the

isolation being provided to ensure that the processes running in one container do not in any way interfere with the processes running in the other container.

**Network**: The network stack of one container is again completely isolated from the other container. Meaning thereby, a container doesn't get any sort of privileged access to the sockets or interfaces of any other container. However, that does not mean that if the containers need to interact with each other at a network level, they cannot do so. The containers can do so through their respective network interfaces — just like they can interact with external hosts. When we specify public ports for our containers or use links, then IP traffic is allowed between containers.

Because of the namespace isolation provided by the network, we can map more than one container to the same port at the container level, provided they are using different ports at the host level. Let us create a couple of containers and test it out. This time let us use *Busybox* containers to do our test. *BusyBox* is a software suite that provides several Unix utilities in a single executable file. Check out the screenshot in *Figure 7.3*:

```
docker run -dit --name test_nw -p 80:4000 busybox:latest
docker run -dit --name test1_nw -p 81:4000 busybox:latest
```

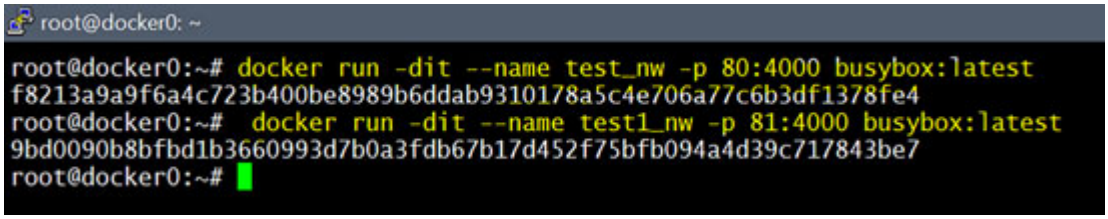



***Figure 7.3***

So, we see that we can map port 4000 on each of the containers to different ports on the host. We also see that each container gets its own `eth0` interface with its IP address and range of ports. Check out the screenshot in *Figure 7.4*:

```
docker exec -it test_nw sh
ifconfig
docker exec -it test1_nw sh
```

```
ifconfig
```

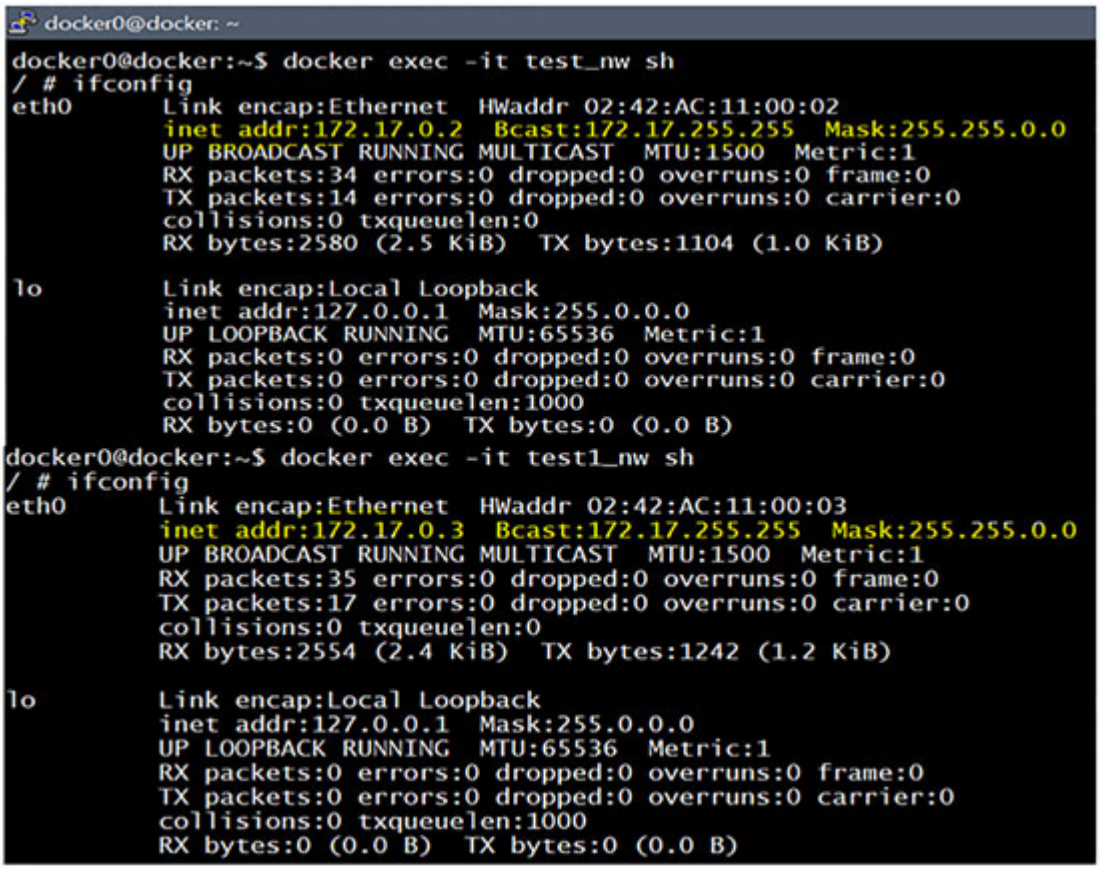



***Figure 7.4***

**Filesystem/Mount**: Each container gets its own filesystem to work with. To elucidate this, we can think of each container having its own insulated operating system. The host, of course, has its own full-fledged operating system. Containers can only see inside of their mount namespaces, and there is no way they can access the mount namespaces of other containers. Let us have a quick look at the filesystem of our *Busybox* containers to see how it is structured. See the screenshot in *Figure 7.5*:

```
docker exec -it test_nw sh
ls
docker exec -it test1_nw sh
ls
```

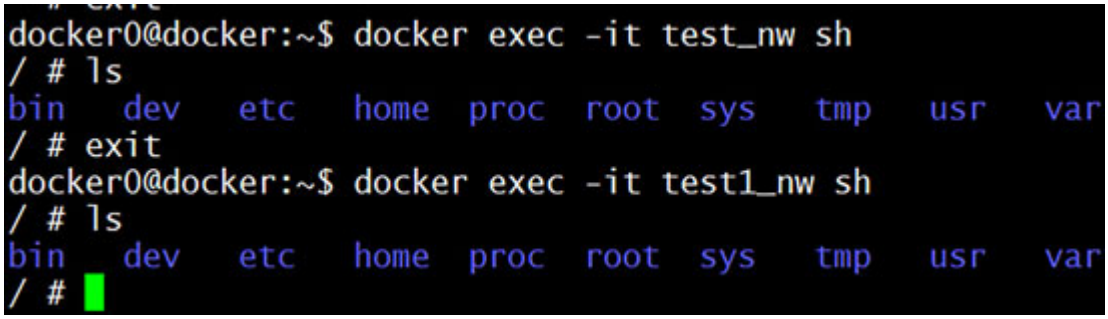



***Figure 7.5***

It is quite clear that each of the containers has similar filesystems and will be able to use them howsoever they want to, with absolutely no impact on any other filesystem attached to any other container.

**Inter-Process Communication** (**IPC**): Linux has a mechanism for sharing memory between processes. Shared memory segments are used to accelerate inter-process communication at memory speed, rather than through pipes or the network stack. This is known as inter-process communication. In large-scale software architecture, IPC binds the processes together. Containers have their own IPC namespace, but it can be configured for shared memory access as well. By default, however, one container cannot access the memory of another container or the memory of the host. The following are two examples of creating containers with shareable memory access. Check out the screenshot in *Figure 7.6*:

```
docker run -dit --name shared --ipc shareable centos
docker run -dit --name shared1 --ipc container:shared centos
```


***Figure 7.6***

The first command with the flag ipc shareable has its own private IPC namespace, but crucially, with a possibility to share it with other containers. On the other hand, the second command shares its IPC

namespace with the container named shared created by the first command, as shown in *Figure 7.6*.

**User Namespace** (**USR**): Docker has a user namespace inside the container. This usr namespace is isolated from other users in other containers as well as users on the host. However, the USR namespace is not activated by default. This implies that if the UID in the container matches the UID in the host, then it will have the same permissions as the host user, which could be disastrous in certain cases, especially so where volumes have been created and shared between the container(s) and the host. The way out is to use Docker's `userns-remap` option to remap container user ids to Linux user ids, which have much fewer privileges and permissions. For example, root in the container which runs with the user id 0 can be mapped to the Linux host to run with a different id, thus eliminating the danger of misuse of root privileges by accident or by design from inside the container.

On a different note, typically, most containers run as the root user, with almost unfettered access to the state of the container. This might not be the most desirable way of doing things as any process running as that user inherits all of those privileges and permissions of the root user, and this, by implication, makes the container vulnerable. One way of handling this could be by creating another user inside the container with limited privileges and letting that newly created user work inside the container. Let's do an example to understand this.

Let's create a container using the ubuntu image. But before doing so, let's confirm that ubuntu, by default, run as root. Check out the screenshot in *Figure 7.7*:

```
docker run --rm ubuntu whoami
docker run -dit --name test ubuntu
```

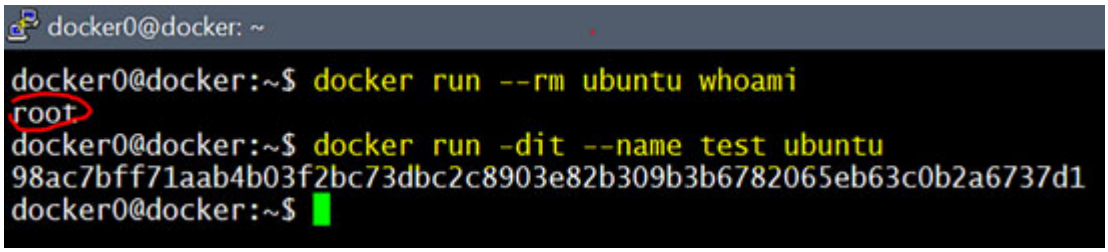


***Figure 7.7***

Now, let us get inside the container and create a new user. Check out the screenshot in *Figure 7.8*:

```
docker exec -it test sh
ls
```

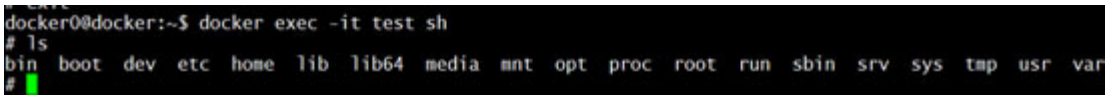



***Figure 7.8***

Next, we add a user named `test_user`, as shown in the screenshot in *Figure 7.9*.

```
adduser test_user
cat /etc/passwd
```

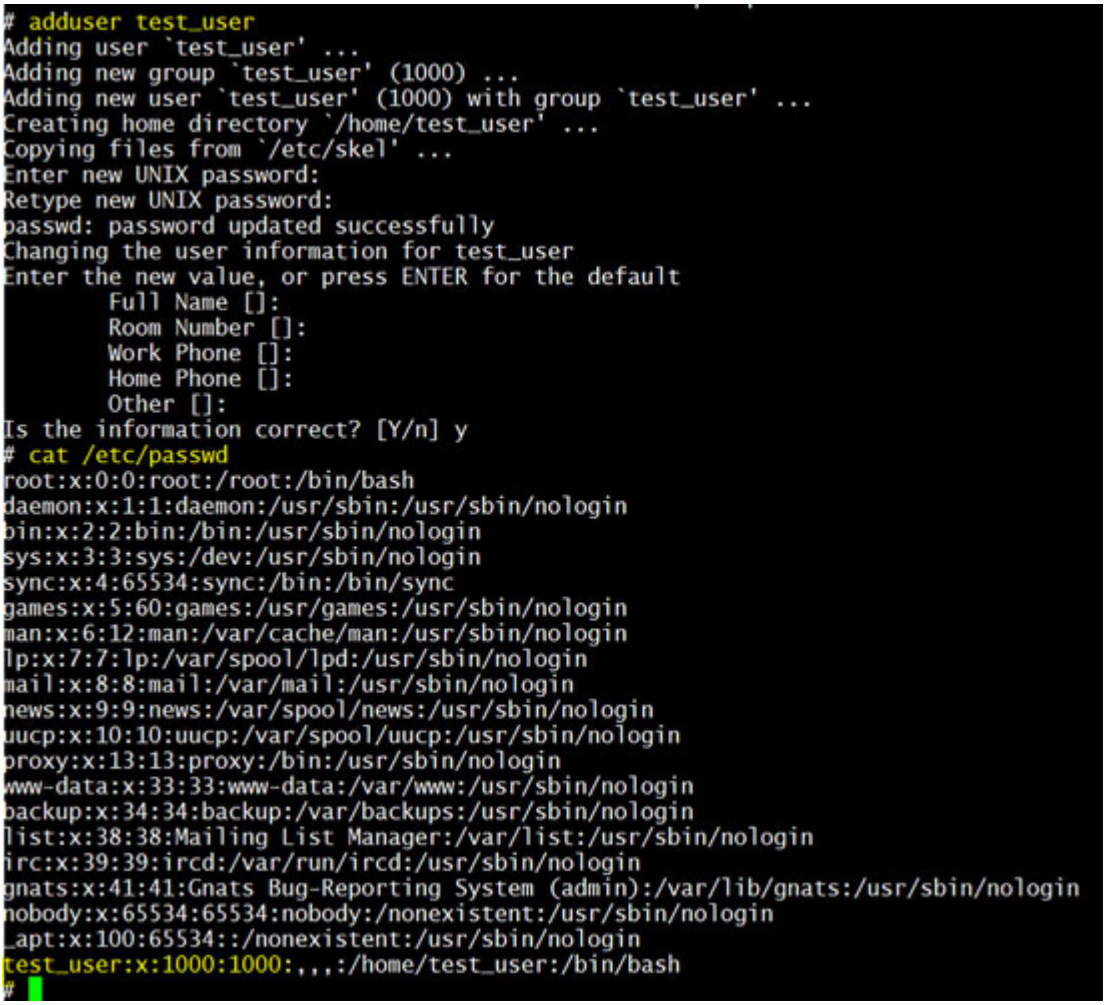


***Figure 7.9***

Now, with the user created, it is very simple to have this new user work inside the container following the principle of providing the least privilege to the user to do a task, thereby ensuring that we have done the diligence of '*hardening'* the container.

**Unix Time Sharing** (**UTS**): The UTS namespace is all about isolating the hostname. It ensures that isolation is guaranteed to two specific elements in the system that relates to the `uname` system call. In other words, it simply means that the process has a separate copy of the hostname and the NIS domain name, so it can set it to something else without affecting the rest of the system.

**Network Information System (NIS) is a directory service and was created by Sun Microsystems. It was discontinued once Oracle took over Sun Microsystems. NIS is now rarely used.**

## Control Groups

**Control Groups** (**cgroups**) are a mechanism to group processes together based on their resource consumption, prioritization, stability, and to an extent, their security. It can so happen that a bug or a piece of bad code finds its way into our system and almost takes down the entire system. Think of some users running an infinite loop on our system by design or otherwise and crippling the entire setup.

To prevent this from happening, control groups or `cgroups`, which is a Linux kernel feature, allow us to restrict usage of the resources on a system. This is done by:

1. *Putting a limit on resource consumption*: Groups may have a cap on usage of memory, CPU, network, and other resources.
2. *Prioritizing resource consumption*: When there is a resource crunch on the system, one group will be prioritized over the other in the context of resource allocation.
3. *Accounting*: A group's resource usage is measured and monitored.
4. *Controlling resource consumption*: Some groups may be frozen or temporarily stopped and restarted.

So, just to be clear, a `cgroup` is just a set of processes that have the same limits, and/or a common set of priorities set for it. `cgroups` can also be set up as a hierarchical tree structure, which means that a sub-group under the parent group will inherit the limits set for the parent group.

From a docker perspective, there are no resource limits that we start with. So, by default, a container has access to as much resource as the host's kernel scheduler will allow. That said, docker does allow us to control the memory and CPU usage of a container by providing flags that we can set during runtime configurations of the docker command.

## Memory

Obviously, there are some very serious consequences of running out of memory. If the host system runs out of memory, the kernel will throw an **Out Of Memory Exception** (**OOME**). This will result in the kernel randomly starting to kill processes to ensure that it can free up some memory. And this can result in docker itself going down-a very unattractive proposition, to say the least!

So, Docker partially resolves this by prioritizing the OOME, so that other processes on the host system are more likely to be killed first before the Docker daemon is touched. The implication of this is that it is more likely that a container may be killed, rather than the Docker daemon process or other system processes.

To mitigate the OOM exception, we can also enforce hard memory limits through Docker that will prevent any single container from using up all the memory. There are also mechanisms to set soft limits that set certain conditions to be fulfilled for it to kick in.

Let us check some of the more important memory limits.

The flag `--memory` or `-m` sets an upper limit on the memory that can be used by a container. However, if we are using this flag, the minimum memory that we need to set is 4 megabytes. We cannot go below that number.

The flag `--memory-reservation` is a soft limit for memory. This will always be lower than the `--memory` setting. The `--memory-reservation`

simply implies that during a memory crunch, the memory allocation may go down to the `--memory-reservation` value. This value is the bare minimum required to run the container. But when the memory situation improves, the container can access memory up to the level set by the `--memory` flag.

Let's create a couple of containers using these flags, as shown in the following screenshot:

```
docker run -dit –name test --memory 200m --memory-reservation
150m busybox:latest
```


***Figure 7.10***

We can create the container without any issues, but we see a warning near the bottom, which says *WARNING*: *Your kernel does not support swap limit capabilities or the cgroup is not mounted. Memory limited without swap*.

As per the Docker documentation, we get these messages on Ubuntu or Debian systems, but not on RPM-based systems, where these capabilities are enabled by default. Re: *https://docs.docker.com/install/linux/linux-postinstall/*

On my cloud-based Ubuntu system, I used the following set of commands to enable these capabilities.

I logged into the host as the root did the following:

1. Edited the `/etc/default/grub.d/50-cloudimg-settings.cfg` and added the line `GRUB_CMDLINE_LINUX_DEFAULT="console=ttyS0 cgroup_enable=memory swapaccount=1"`
2. Saved the file.
3. Updated the GRUB: `update-grub`.
4. Rebooted the system: reboot

Next, we can create another container with memory limits, but we don't see the warning anymore. Check out the screenshot in *Figure*

*7.11*:

```
docker run -dit --name test1 --memory 512m --memory-reservation
450m busybox:latest
```

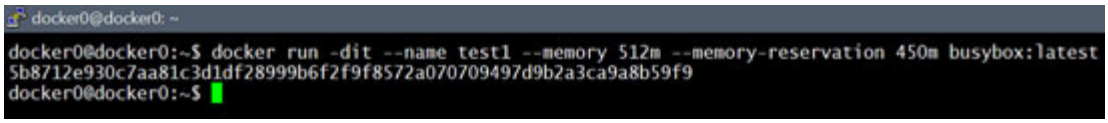



***Figure 7.11***

`--memory-swap` is another flag that can be used to control memory usage. This flag is a very useful one since it allows any excess memory requirements to be written to the disk. Of course, there will be a performance penalty for this, but at least it allows the container to function without throwing an error.

The settings can seem a bit complicated, but we will simplify it.

1. If the flag `--memory-swap` is set to a positive integer, then the `--memory` flag setting is also mandatory.
2. The `--memory-swap` value has to be greater than the `--memory` setting value, because the `--memory-swap` value is inclusive of the `--memory` setting value. For example, if we set `--memory-swap` to 1024m and `--memory` to 768m, then this means that the container can use 768m of memory and 256m of the swap, that is, 1024m-768m.
3. If we purposely set `--memory-swap` to 0, then this is tantamount to unsetting the `--memory-swap` flag.
4. If the `--memory` flag is set to a positive integer, and the flags `--memory-swap` and `--memory` are set to the same value, say both `--memory` and `--memory-swap` is set to 512m, then it is as if there is no swap set. The container will not have access to swap.
5. If only the `--memory` flag is set, and the `--memory-swap` is unset, then the container has access to as much swap as the `--memory` setting. For example, if we have set `--memory` at 512m, then by default, we have access to 512m of the swap, provided that much of swap is available on the system.

6. If we set `--memory-swap` explicitly to -1, then this will allow our container to access an unlimited amount of swap-as much as exists on the host system.

**Inside the container, tools like 'free' report the host's available swap, and not what is available inside the container. We should not rely on the output of free or similar tools to determine the swap inside the container.**

Before finishing this section, let us now create a container using the memory parameters we discussed above. Check out the screenshot in the following *Figure 7.12*:

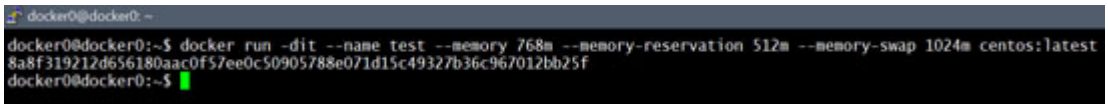



***Figure 7.12***

So, we see that we can create a container with several flags set for memory control.

## CPU

By default, containers may access unlimited CPU cycles on the host machine. Obviously, in a real-world situation, this cannot be an unfettered privilege, because just like memory, a host machine will have a finite amount of CPU available. No single container ought to be able to access all the CPU, because in the worst case scenario, it might lead to a massive CPU bottleneck on the host machine, with all of the consequences indigenous to such a situation.

There are several runtime flags that can be set to control a container's access to the CPU resources. When we set these flags, Docker modifies the `cgroups` settings on the host machine.

There are three commonly used flags we will discuss here. They are as follows:

1. `--cpus`: This is a explicit setting that specifies how much of the available CPU resources a container will use. We can also use decimals while specifying the maximum CPU allocation for a

container. For example, for a four CPU host machine, 2.6 or 1.2 CPUs is a valid specification.

2. `--cpuset-cpus`: This flag is even more granular. We can use a comma-separated or hyphenated list to specify the CPUs. For example, we can specify 0-2, which would mean the CPUs numbered 0, 1, and 2; or we can specify 0,4, which would mean the container would use the first and the fifth CPU.
3. `--cpu-shares`: This is really a soft limit. Setting this flag to a value greater than its default of 1024 would provide the container with a priority for getting CPU cycles whenever there is a CPU crunch on the system. Setting it to a value less than 1024 would naturally scale down proportionately its access to the host CPU cycles.

Let us try out a few examples.

First, let us find out the number of CPUs on our system:

```
lscpu | grep CPU
```

Then let us run the following and see the output in *Figure 7.13*:

```
docker run -dit --name test --cpus 2.4 busybox:latest
docker run -dit --name test1 --cpuset-cpus 0,3 busybox:latest
docker run -dit --name test2 --cpu-shares 1096 busybox:latest
```

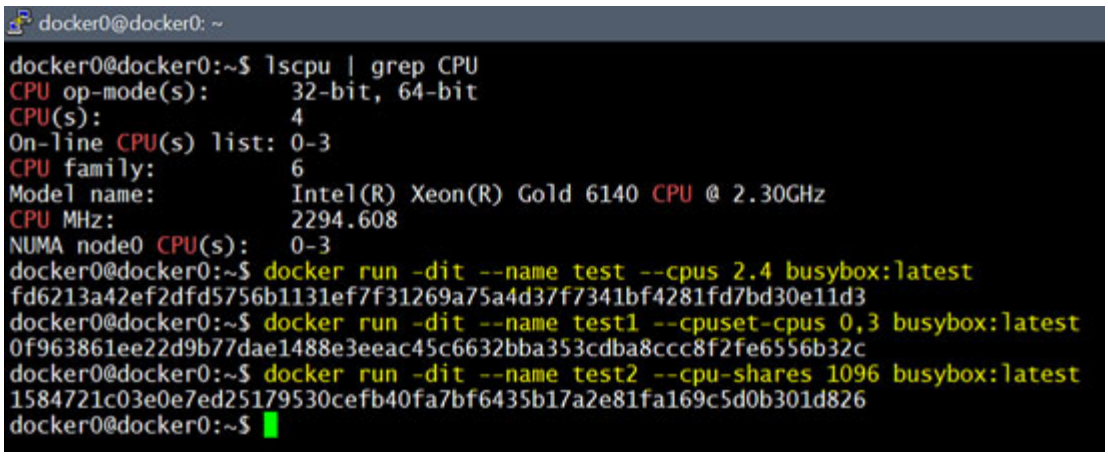



***Figure 7.13***

# Capabilities

On traditional Unix systems, there are two categories of processes:

**Privileged processes,** that is, processes that will run without any permission checks. Typically, these are processes in the super-user category or is the root process and has the `UID 0`.

**Non-Privileged processes** that do not have the `UID 0` and have to go through the full gamut of permission checks before they run.

Starting with the Linux kernel *version 2.2*, the privileges and permissions associated with a super-user or root have been granularized into a separate set of capabilities that can be independently enabled or disabled on a per-thread basis.

Let us run a few commands on our *Ubuntu 18.04.1* system and check it out. We run the `cat /proc/sys/kernel/cap_last_cap` command to see the number of capabilities on our system. And then we use the `capsh --print` command to list them out. Check out the screenshot in *Figure 7.14*:

```
cat /proc/sys/kernel/cap_last_cap
capsh --print
```

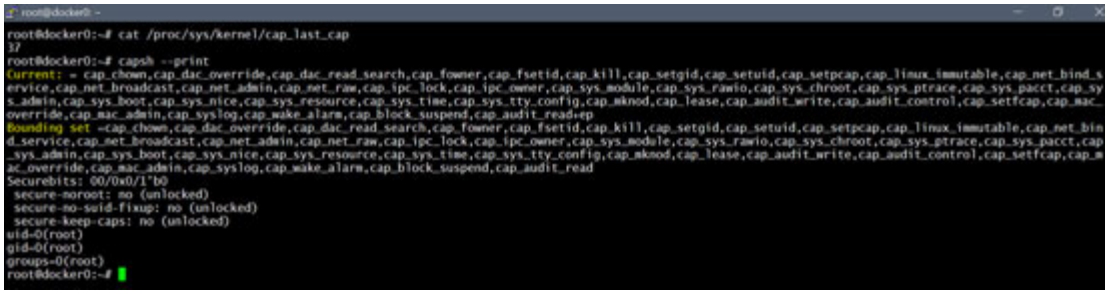



***Figure 7.14***

So, we see that there are thirty-seven distinct capabilities on our system, which are listed out by the next command. If we look carefully at the highlighted parts, there are two sets of capabilities listed out: Current and Bounding set. The Current set is the current set of capabilities, and the capability bounding gates the permitted capabilities that may be granted by an executable file. The bounding set is roughly intended to control which capabilities are available within a process tree.

When we create a container, most of the capabilities are unavailable by default except a few explicit ones which are required to run our applications. Let us do a quick example to understand this. Let us create a container and check what are the capabilities available by default. Check out the screenshot in *Figure 7.15*:

```
docker run --rm centos:latest sh -c "capsh --print"
```

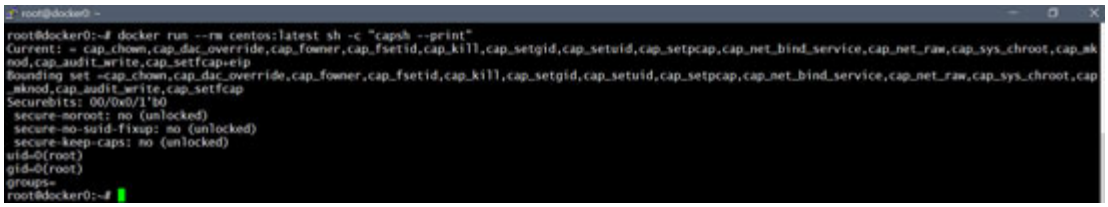



***Figure 7.15***

We see a much smaller set of capabilities available within the docker container by default. This is the built-in security. But the set of capabilities such as they are, are enough to run most applications, and not only that, if there are any additional capabilities required in a specific situation, Docker provides us the ability to add those capabilities into our container. On the flip side, if we run into some security concerns, we can also delete any available capability from the container. We just need to use the `--cap-add` or `--cap-delete` flags. Again, let us test it out to understand it better.

In the first example, we are going to check whether we can use the `chown` command to change the UID of a file inside the container, and then we will drop the capability `cap_chown` and see its effect. Let's run the following set of commands and see the output in *Figure 7.16*:

```
docker run -dit --name test centos:latest
docker exec -it test sh
touch myfile
ls
ls -l myfile
chown 5000 myfile
ls -l myfile
exit
docker run -dit --name test1 --cap-drop cap_chown centos:latest
docker exec -it test1 sh
```

```
touch myfile
ls
ls -l myfile
chown 5000 myfile
```

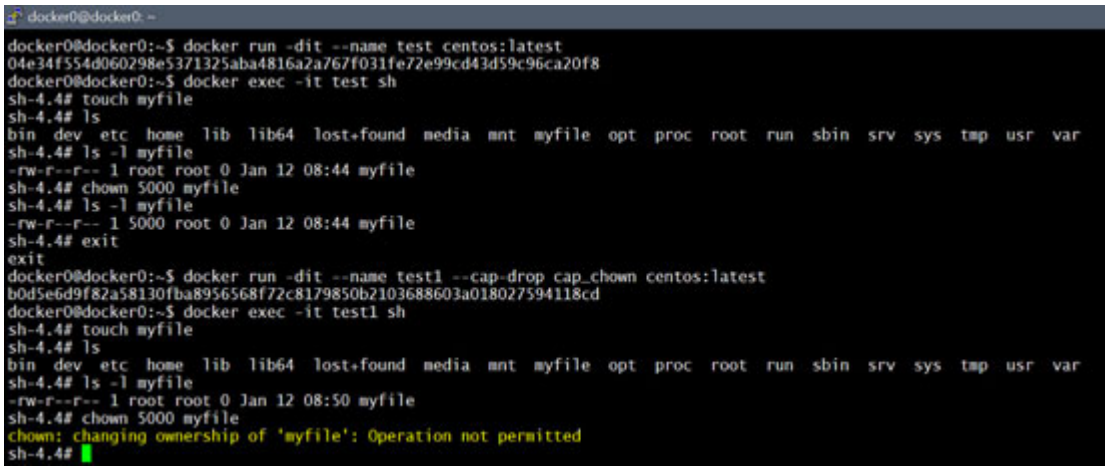



***Figure 7.16***

So, as expected, we see that the operation is not permitted, and we cannot change the ownership.

Now, let us add a capability and check it out. We will add a capability not available in the container by default. Let us choose `linux_immutable` capability. This capability allows us to make a file immutable, meaning thereby that not even the user root can delete the file. This capability is very helpful to ensure that no one accidentally removes a file.

We will create a container adding the `linux_immutable` capability. Then we will go inside the container, create a file, and change its attributes to immutable using the (change attribute) `chattr` command. We will then try to remove the file. We don't expect the operation to succeed because we have made the file immutable. Next, we remove the immutable tag using the `chattr` command again and then try to remove the file again, after which we ought to succeed. Check out the screenshot in *Figure 7.17*:

```
docker run -dit --name test --cap-add linux_immutable
ubuntu:latest
docker exec -it test sh
```

```
touch myfile
ls
chattr +i myfile
rm myfile
chattr -i myfile
rm myfile
ls
exit
```

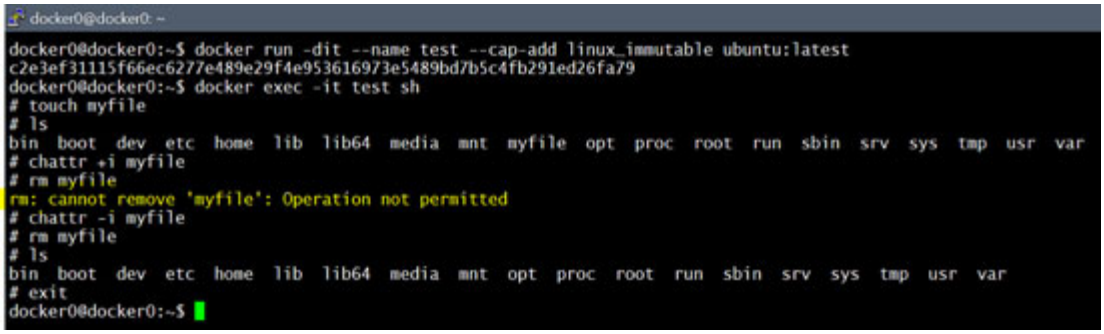



***Figure 7.17***

## Mandatory Access Control

Mandatory access control, often known as **Simplified Mandatory Access Control Kernel** (**SMACK**), is a set of mandatory access control rules at the operating system level that protects data and processes from any person or program trying to maliciously access it and/or manipulate it. *Since Linux 2.6.25, mandatory access control is merged into it*.

On systems that are enabled for MAC, system administrators can implement an organization-wide security policy that cannot be overridden by anyone, unlike **discretionary access policies** (**DAC**), that may be overridden.

Typically for Linux, we use either *AppArmor* or *SELinux* to enforce mandatory access control. Docker expects the *AppArmor* policy to be loaded and enforced.

Policies are enforced at the container level rather than the daemon level. Docker has a default *AppArmor* profile known as the '*default-profile*' that is silently attached to any container that we create. This

profile enforces a set of moderately strong access control rules. On Docker versions earlier than 1.13, the profile was generated in `/etc/apparmor.d/docker`, but for Docker versions 1.13 and later, the Docker binary generates the profile in `tmps` and then loads it into the kernel.

But before we go deeper into *AppArmor* or *SELinux* for containers, let's take a quick look at how *AppArmor* and *SELinux* works.

Let us get started with *AppArmor*. It is basically a Linux kernel security module that allows us to restrict the capability of a user or a program to access resources on the system. *This is done by creating a set of profiles that control the users' or programs' capability to access and work with the resources*. Capabilities like network access, raw socket access, and the permission to read, write, or execute files are controlled through these profiles.

**AppArmor is available for Debian-based distributions like Ubuntu, Astra Linux, Kali Linux etc. While SELinux is available on Fedora, RHEL, CentOS, Oracle Linux, and other rpm based systems.**

Let us see the default profiles loaded on our *Ubuntu* system. But before that, we have to make sure we have the *AppArmor* utilities installed.

```
apt install apparmor-utils
```

Next, let us run the command `apparmor_status` to see the profiles loaded by default on the system. We see we have 16 profiles running on the system as can be seen from the screenshot in *Figure 7.18*:

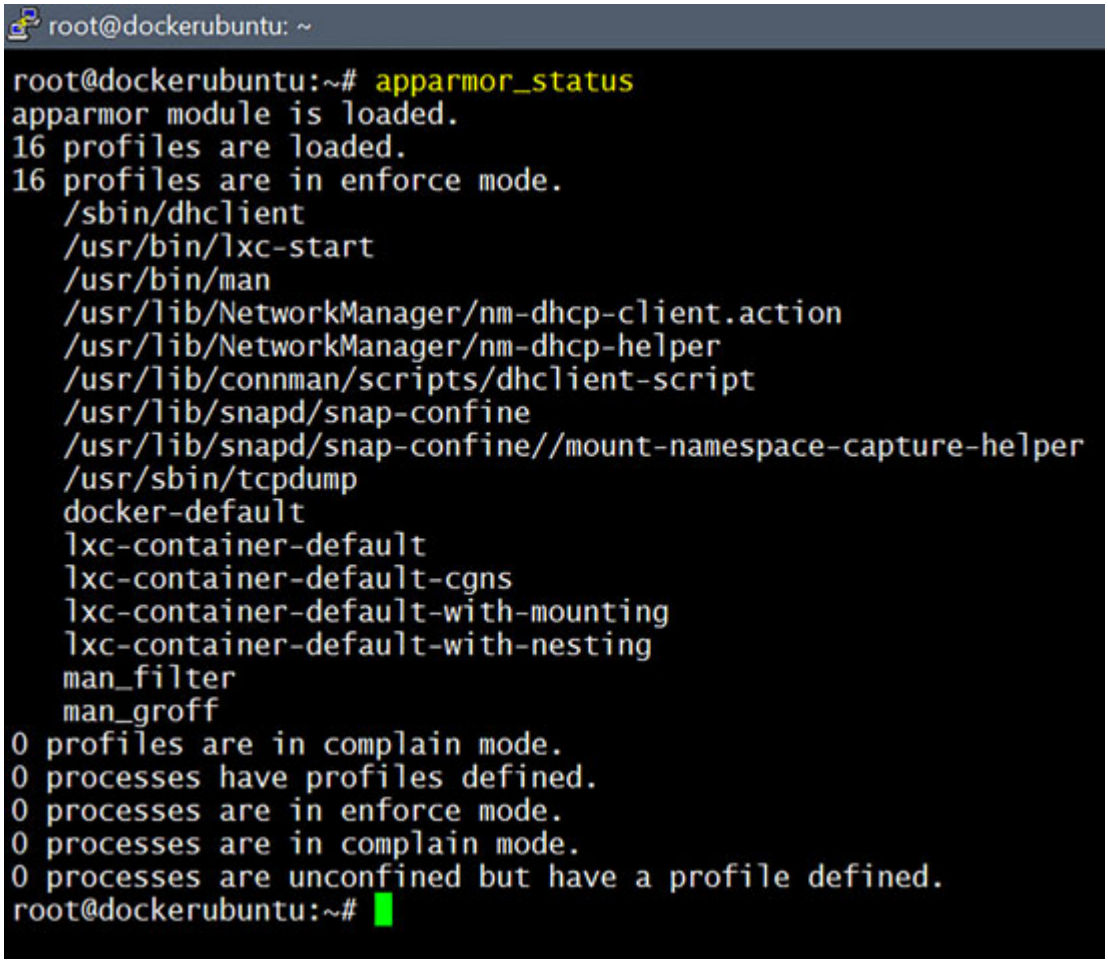



***Figure 7.18***

These profiles are usually found under the `/etc/apparmor.d` directory. Let us have a quick look at it. Check out the screenshot in *Figure 7.19*:

```
cd /etc/apparmor.d
ls
```

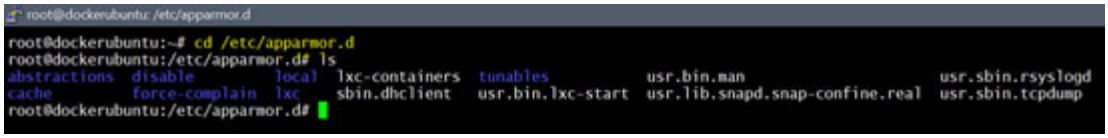



***Figure 7.19***

The default profiles provide us with a built-in security apparatus whereby notwithstanding the fact whether we are a superuser or even root, we have to go through the whole mechanism of

mandatory access control checks to get access to the resource. And if we don't pass the access control checks, the resource is out of bounds for us.

Let us test this out. Let us take a simple Unix command like `useradd` and build a profile for it. Then we try creating a user using the `useradd` command as root and see what happens.

However, before we do that, let's familiarize ourselves with a few *AppArmor* related commands. By using `aa-` followed by pressing the tab key twice, we get a list of executables available under *AppArmor*. We can use these executables to set up, enable, disable, and in general, configure the *AppArmor* profile. Let us take a look at the screenshot in *Figure 7.20*:


***Figure 7.20***

So, now let's go ahead with creating a profile for the `useradd` command. But, just to be sure, we will first create a user by using the `useradd` command to ensure the command is working for the root user. Check out the screenshot in *Figure 7.21*:

```
useradd test_aa
cat /etc/passwd |grep test_aa
```

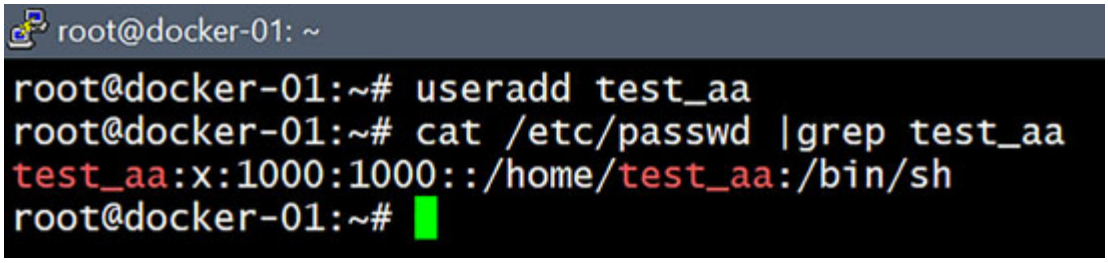



***Figure 7.21***

Sure enough, the command `useradd` works just fine. So now, what we will do is we will use the available executable `aa-autodep` to set a profile for the `useradd` command.

`aa-autodep` is used to generate a very minimal *AppArmor* profile for a set of executables. Re: *Figure 7.22*:

```
aa-autodep useradd
```

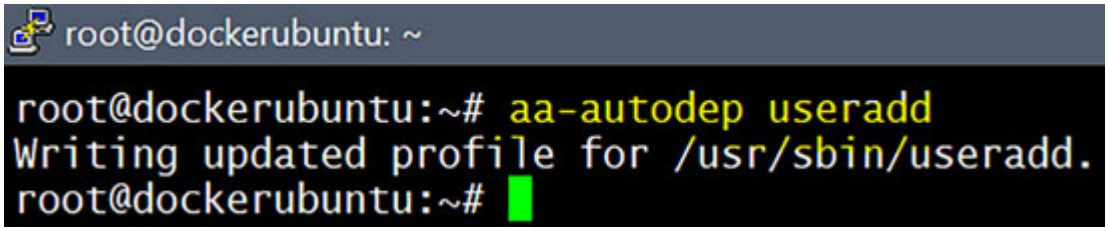



***Figure 7.22***

Running the above command creates an entry in `/etc/apparmor.d` directory`-usr.sbin.useradd`, as shown in the screenshot in *Figure 7.23*. It also adds another profile to the sixteen profiles we already had on our system. Re: *Figure 7.24*:

```
cd /etc/apparmor.d
ls
cd ~
aa-status
```

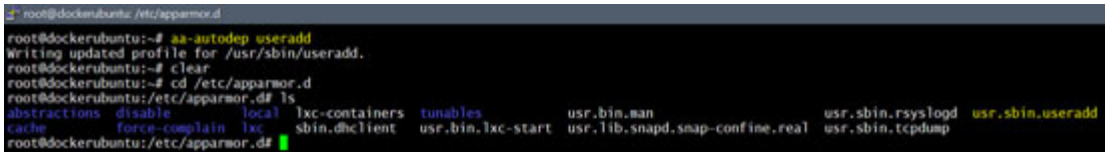



***Figure 7.23***

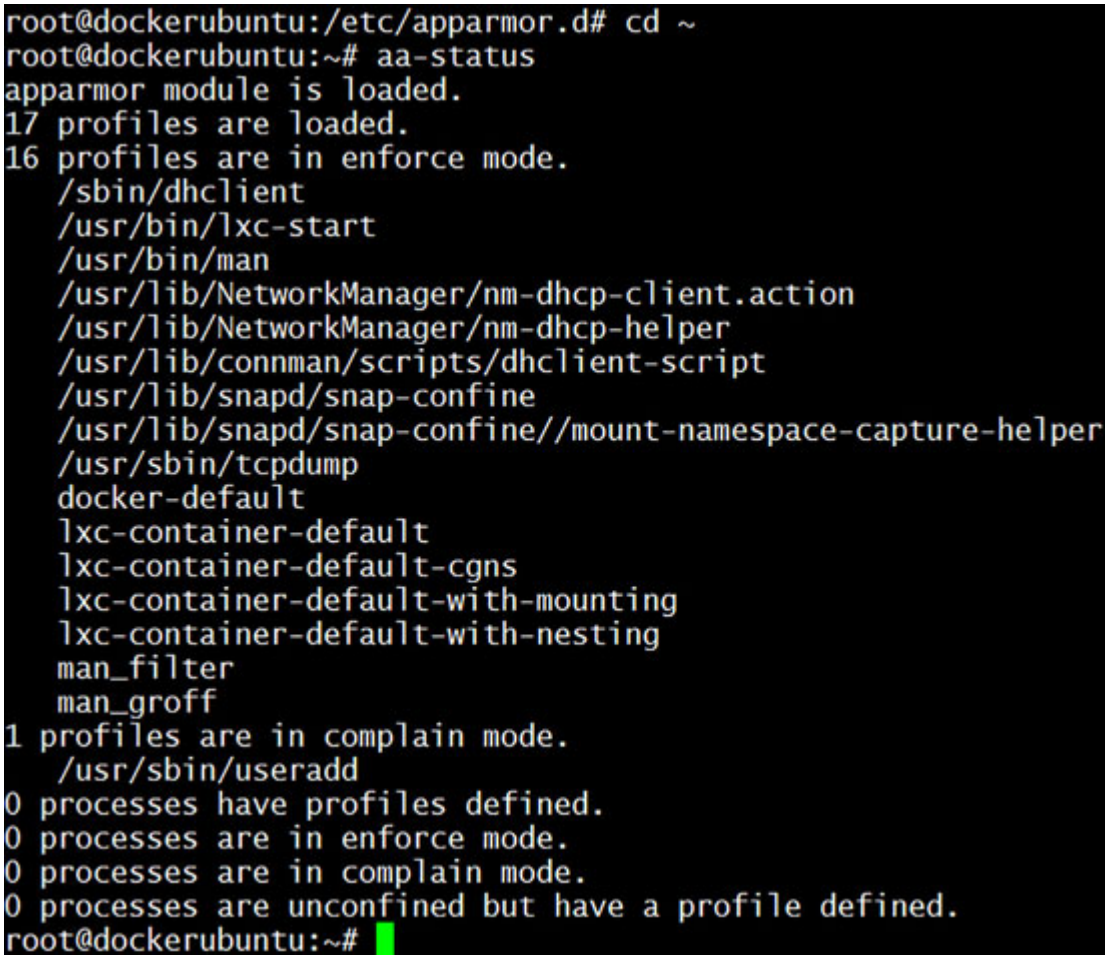



***Figure 7.24***

If we look closely, while the number of profiles has indeed increased to seventeen, one profile is in '*complain mode*,' and that is the profile we just created. Having a profile in complain mode means that while there will be no access restrictions on the resource, but the system will '*complain'* and log any '*access violations*' made. So, effectively we are still not prevented from creating users on our system; only our actions will be logged.

Let us go ahead and try creating another user and see if we can do it. Check out the screenshot in *Figure 7.25*:

```
useradd test_aa1
cat /etc/passwd | grep test_aa1
```

```
root@docker-01:~# useradd test_aa1
root@docker-01:~# cat /etc/passwd |grep test_aa1
test_aa1:x:1001:1001::/home/test_aa1:/bin/sh
root@docker-01:~#
```

***Figure 7.25***

Again, no problems at all. So, now let us use `aa-enforce` to enforce the profile and check what happens. Check out the screenshot in *Figure 7.26*:

```
aa-enforce useradd
aa-status
```

```
root@dockerubuntu:/etc/apparmor.d# aa-enforce useradd
Setting /usr/sbin/useradd to enforce mode.
root@dockerubuntu:/etc/apparmor.d# aa-status
apparmor module is loaded.
17 profiles are loaded.
17 profiles are in enforce mode.
   /sbin/dhclient
   /usr/bin/lxc-start
   /usr/bin/man
   /usr/lib/NetworkManager/nm-dhcp-client.action
   /usr/lib/NetworkManager/nm-dhcp-helper
   /usr/lib/connman/scripts/dhclient-script
   /usr/lib/snapd/snap-confine
   /usr/lib/snapd/snap-confine//mount-namespace-capture-helper
   /usr/sbin/tcpdump
   /usr/sbin/useradd
   docker-default
   lxc-container-default
   lxc-container-default-cgns
   lxc-container-default-with-mounting
   lxc-container-default-with-nesting
   man_filter
   man_groff
0 profiles are in complain mode.
0 processes have profiles defined.
0 processes are in enforce mode.
0 processes are in complain mode.
0 processes are unconfined but have a profile defined.
root@dockerubuntu:/etc/apparmor.d#
```

***Figure 7.26***

Now, all the seventeen profiles stand enforced, and no profile is in complain mode. So, we can guess now the root user now should not be able to create a user. Let us check it out. After all, seeing is believing! Check out the screenshot in *Figure 7.27*:

```
useradd test-aa2
```

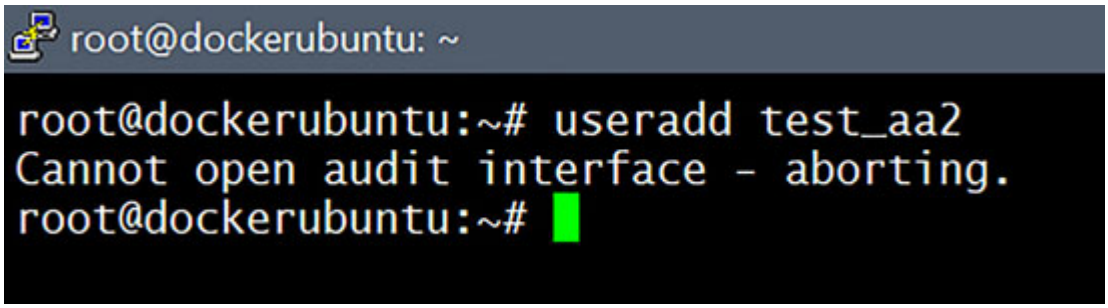



***Figure 7.27***

So, there we are. Even root cannot create a user.

We can use `aa-disable` to unload the profile from the kernel and prevent the profile from being loaded on *AppArmor* startup, or we might want to use `aa-logprof` to scan the profile and then allow only the root user the privilege to create users. Once we *(A)llow* that and *(S)ave* the changes, root can go ahead and create users as is shown in the screenshot *Figure 7.28*:

```
aa-logprof
(A)llow and (S)ave the changes
```

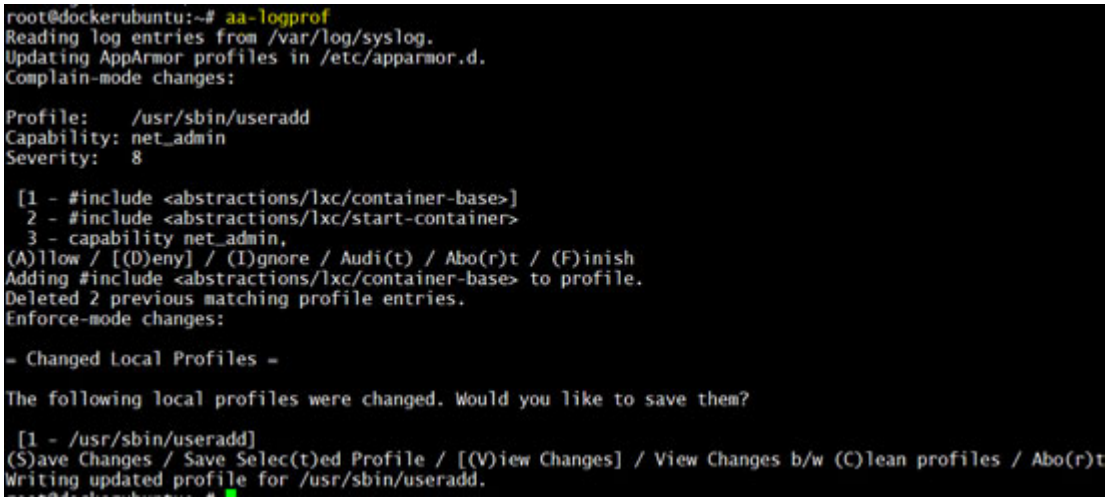



***Figure 7.28***

Next, we try adding a user and see the output in *Figure 7.29*:

```
useradd test_aa2
```

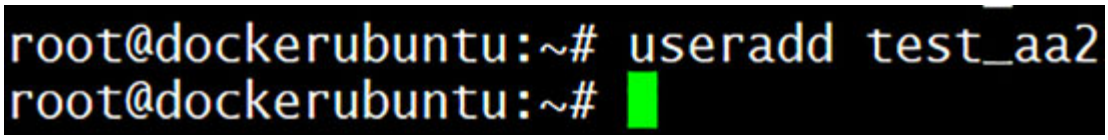



***Figure 7.29***

## Docker and AppArmor

For every Docker container we create, a default *AppArmor* profile gets attached to the container. On Docker versions before 1.13.0, a default profile for containers was created under `/etc/apparmor.d/docker`. In *versions 1.13.0* and later, the Docker binary generates this profile in `tmps`, and then it gets loaded into the kernel.

The default profile for docker is known as the docker-default, and it is moderately protective and, in general, ought to be good enough for running most applications. The docker-default is loaded into the kernel along with other profiles. To start with, let us see the profiles which are loaded by default. Check out the screenshot in *Figure 7.30*:

```
aa-status
```

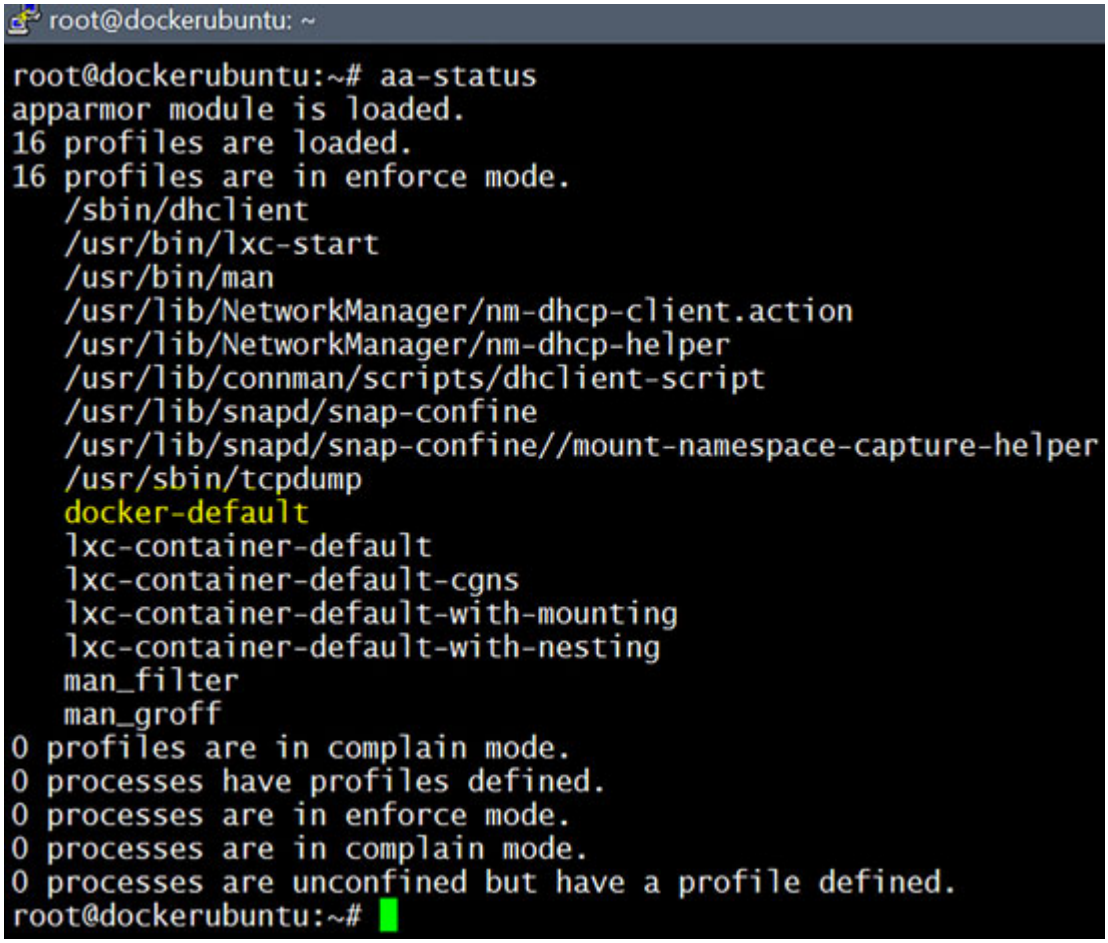



***Figure 7.30***

There we go-the `docker-default` is there as one of the sixteen loaded profiles. Let us create a container and run a few commands to see how things look. Check out the screenshot in *Figure 7.31*:

```
docker run -dit --name test_profile busybox:latest
docker ps
aa-status
```

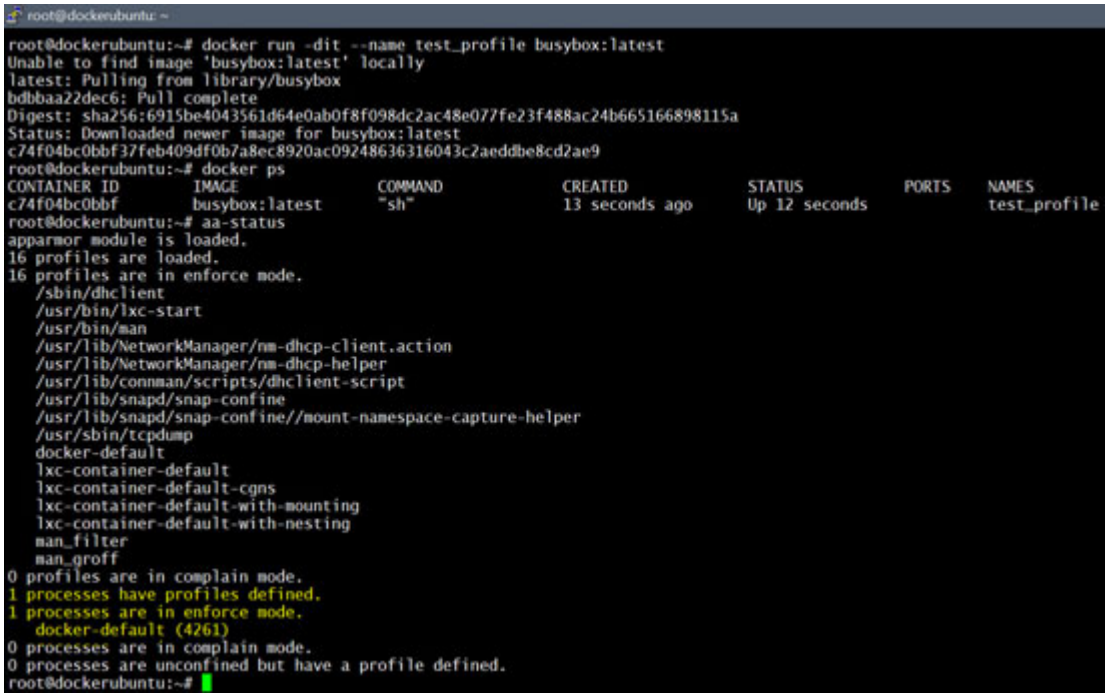



***Figure 7.31***

We see that one process (our container) has a profile defined. And the docker-default with PID 4261 is also running. Let us create another container, explicitly specifying the `docker-default` profile using the flag `--security-opt`.

```
docker run -dit --name test_profile1 --security-opt
apparmor=docker-default ubuntu:latest
```

And, now let us check the status of profiles once again. Check out the following screenshot:

```
aa-status
```

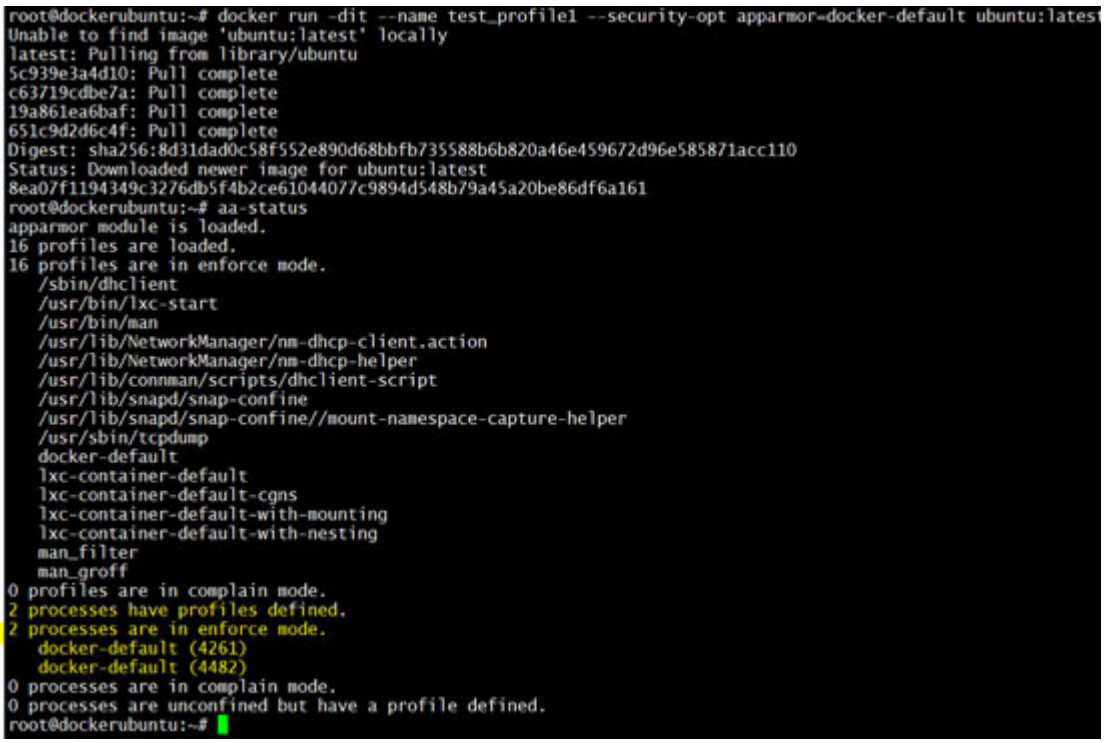



***Figure 7.32***

We see that now there are two `docker-default` profiles running in enforced mode.

We can also create our very own profile and attach it to the containers we create. Let us do a quick and dirty test case. We create a profile by the name of `myprofile` under `/etc/apparmor.d` that basically denies read, write and execute privilege under the `/etc directory`. We save the file.

```
cd /etc/apparmor.d
vi myprofile
profile myprofile1 flags=(attach_disconnected,mediate_deleted)
{
  #include <abstractions/base>
  file,
  umount,
deny /etc/** wrx,
}
```

Next, we load the profile and check it out whether we now have the seventeenth profile loaded. Check out the screenshot in *Figure 7.33*:

```
apparmor_parser -r -W /etc/apparmor.d/myprofile
aa-status
```

```
root@docker-01:/etc/apparmor.d# apparmor_parser -r -W /etc/apparmor.d/myprofile
root@docker-01:/etc/apparmor.d# aa-status
apparmor module is loaded.
17 profiles are loaded.
17 profiles are in enforce mode.
   /sbin/dhclient
   /usr/bin/lxc-start
   /usr/bin/man
   /usr/lib/NetworkManager/nm-dhcp-client.action
   /usr/lib/NetworkManager/nm-dhcp-helper
   /usr/lib/connman/scripts/dhclient-script
   /usr/lib/snapd/snap-confine
   /usr/lib/snapd/snap-confine//mount-namespace-capture-helper
   /usr/sbin/tcpdump
   docker-default
   lxc-container-default
   lxc-container-default-cgns
   lxc-container-default-with-mounting
   lxc-container-default-with-nesting
   man_filter
   man_groff
   myprofile
0 profiles are in complain mode.
1 processes have profiles defined.
1 processes are in enforce mode.
   docker-default (4019)
0 processes are in complain mode.
0 processes are unconfined but have a profile defined.
```

***Figure 7.33***

Indeed, we see all seventeen profiles loaded, and we are good to go. Let us create a container attaching our profile to it as shown in the following screenshot:

```
docker run -dit --name test_myprofile --security-opt
"apparmor=myprofile" alpine:latest
```

```
root@docker-01:/etc/apparmor.d# docker run -dit --name test_myprofile --security-opt "apparmor=myprofile" alpine:latest
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
c9b1b535fdd9: Pull complete
Digest: sha256:ab00606a42621fb68f2ed6ad3c88be54397f981a7b70a79db3d1172b11c4367d
Status: Downloaded newer image for alpine:latest
683b487da0e57c54e1dedbf53cc293701b41b7e7588e75fee21ab0f0d8069b4a
root@docker-01:/etc/apparmor.d#
```

***Figure 7.34***

And when we now run `aa-status`, we see two containers running with the docker-default profile, while the third container is running with the customized profile-`myprofile`. The screenshot in the following *Figure 7.35* is self-explanatory:

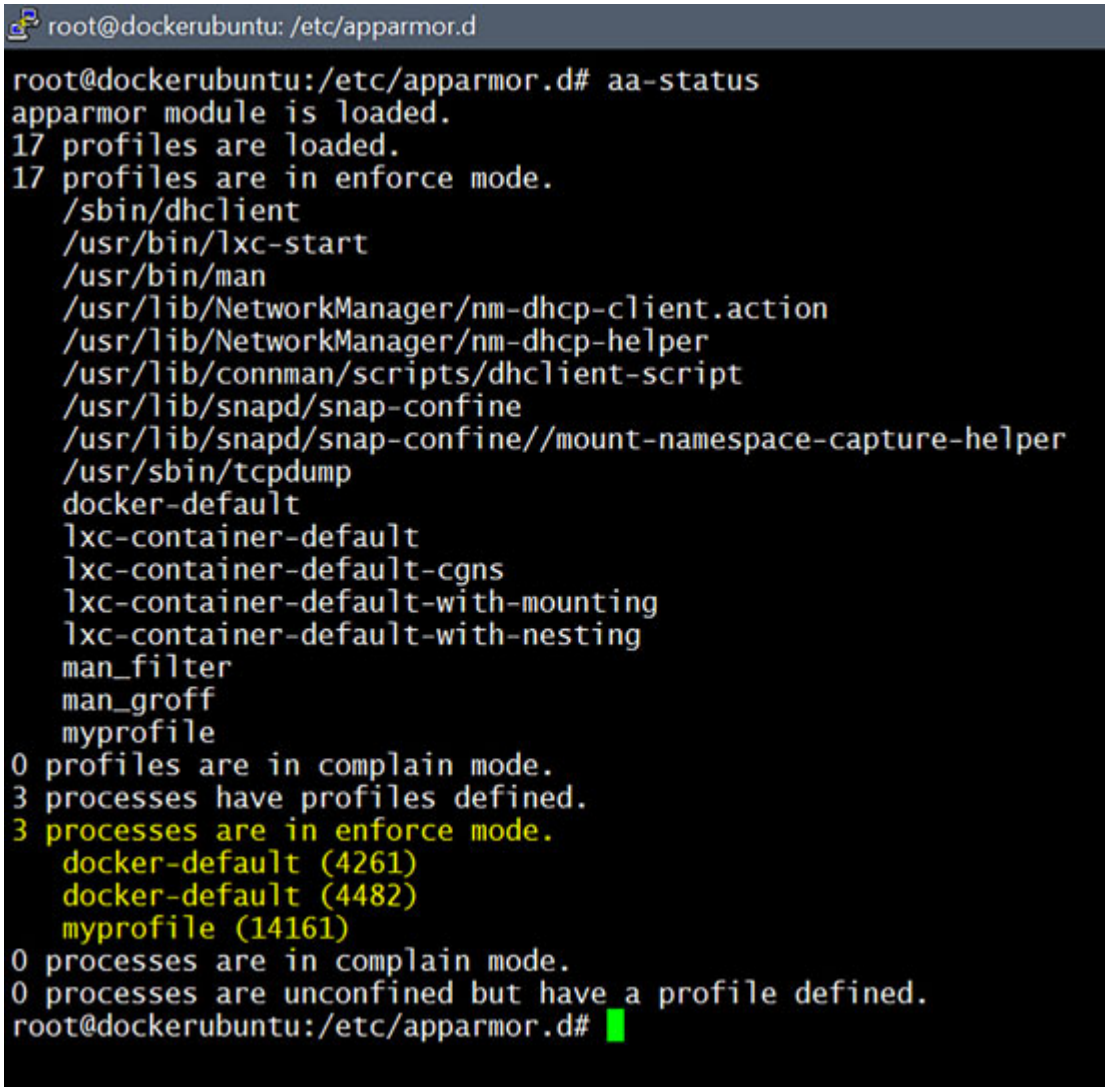



***Figure 7.35***

Okay, so far, so good. Let us get inside our container and see if the `myprofile` profile is actually working. Check out the screenshot in *Figure 7.36*:

```
docker ps
docker exec -it test_myprofile sh
ls
cd /etc
mkdir test
cd ~
cd /dev
mkdir test
```

```
ls
```

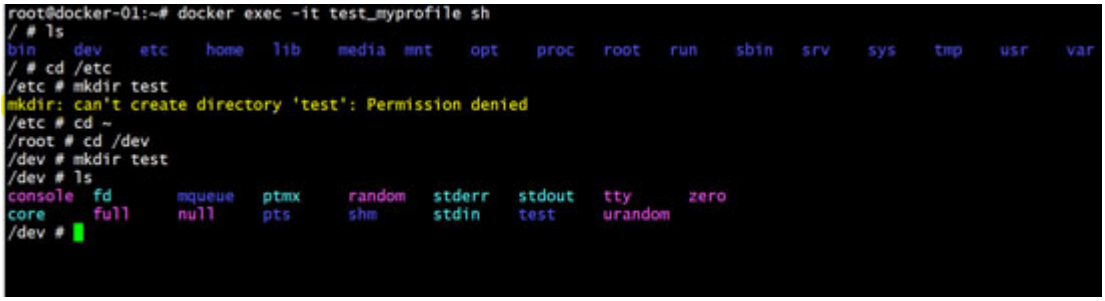



***Figure 7.36***

And as we can see in the diagram above, we cannot create a directory in the `/etc` directory inside the container. We get a 'Permission denied' message based on the fact that we created a custom *AppArmor* profile that denied *read/write/execute* privilege in `/etc` directory. However, when we try creating a directory inside `/dev` (or for that matter in any of the other directories) inside the container, we can easily do so. The reader can test it out.

## Docker and SELinux

While *AppArmor* is one way of securing our Docker containers, *SELinux* is another mechanism available and is mostly used for *RHEL*, *CentOS*, *Fedora*, *Oracle Linux*, and other such systems.

*SELinux* labels every file, directory, process, network, ports, just about anything. Rules are written to control the access of a labeled process to a labeled object like a file. *We call this policy. The kernel enforces the rules*.

*SELinux* needs to be enforced at the daemon level. Linux kernel is the central component of Linux operating systems. It is responsible for managing the system's resources, the communication between hardware and software, and security. The kernel plays a critical role in supporting security at higher levels.

The labeling system of *SELinux* controls the access granted to individual processes by the kernel. *As pointed out earlier, the rules define the access allowed to a process based on the '**labeled type**' of the process accessing the '**labeled type**' of the object*.

*SELinux* caches decisions are allowing access or denying it so that in future, policy rules can be checked more optimally, thereby making the system more performant. The name of the cache is **Access Vector Cache** (**AVC**).

**SELinux doesn't kick in if access is already denied under the Discretionary Access Control (DAC) policy.**

Okay, let us first login to our Linux system and check the status of *SELinux*. We can use either the `getenforce` or `sestatus` command to do so. Check out the screenshot in *Figure 7.37*:

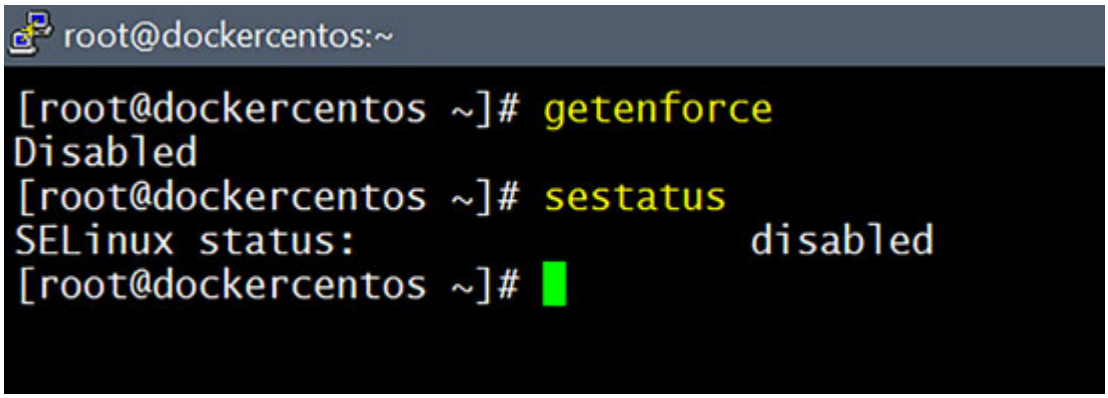



***Figure 7.37***

We see that *SELinux* is disabled. So, let us go ahead and enable it on our system. We need to edit the config file under `/etc/selinux` and add the word 'enforcing' to the 'SELinux=' line. The other two options are 'permissive' and 'disabled'.

```
vi /etc/selinux/config
```

We then need to reboot the system for the change to take effect. After the reboot, we see that *SELinux* is now enabled on the system. Check out the screenshot in *Figure 7.38*:

```
reboot
sestatus
```

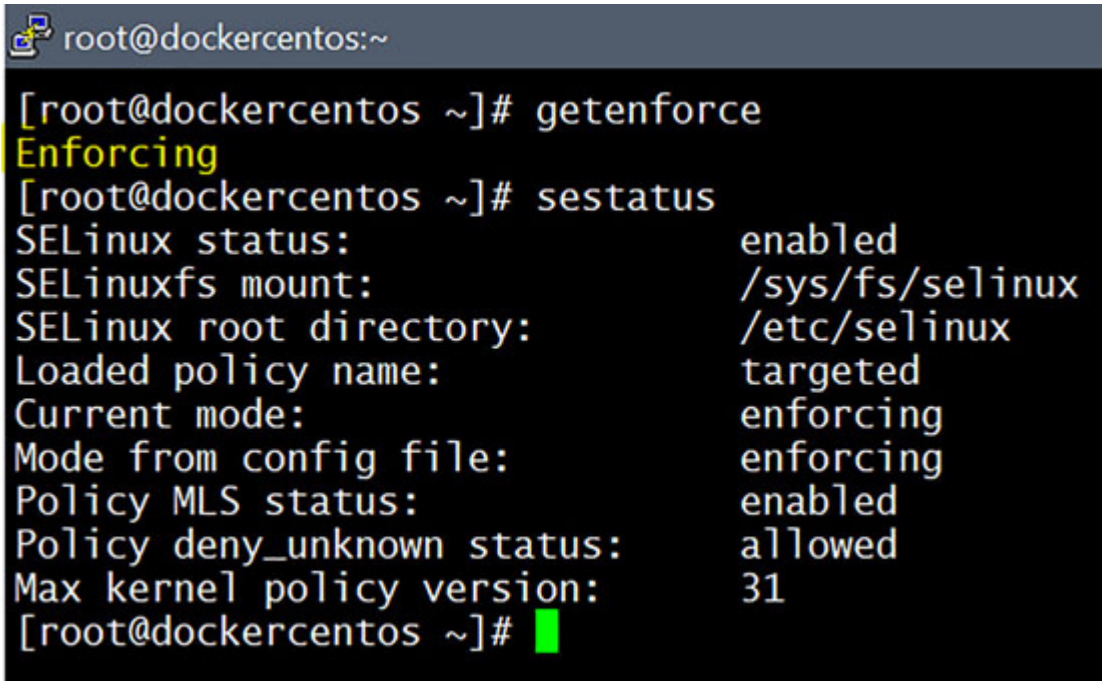



***Figure 7.38***

However, after enabling *SELinux* on our system, we see that we need to do a `systemctl restart docker` to get docker up and running because the docker daemon does not start by itself after an *SELinux* configuration change.

Running the `docker info` command could show the following warning. It is nothing to worried about and can be easily fixed, as shown in *Figure 7.39*:

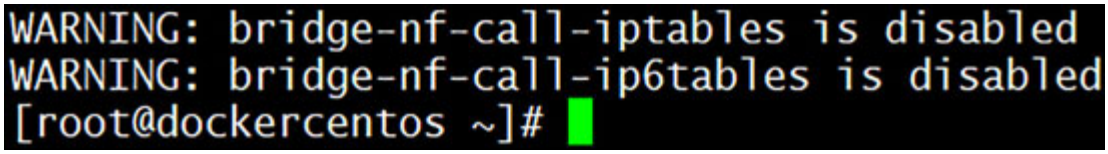



***Figure 7.39***

We just need to modify the `/etc/sysctl.conf` file and add the following lines:

```
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
```

Once the file has been modified, we need to run `sysctl -p` to ensure the changes take effect. Now the docker info command produces

output without any warning. *However, SELinux is not enabled by default in the Docker daemon*. We need to do so explicitly. If we check the docker on our system now, we will see that *SELinux* is not enabled. There is no entry for *SELinux*. Check out the screenshot:

```
docker info --format '{{.SecurityOptions}}'
```

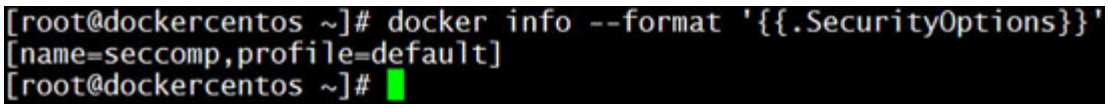



***Figure 7.40***

At this point, if we check the '*Labels*' section of the `docker info` command, we will find that there are no labels created either. This is a significant observation at this stage because we will run the same exact command after enabling *SELinux* on Docker. Additionally, let us create a container and inspect the labeling on it. Let us look at the screenshot in *Figure 7.41* carefully:

```
docker run -dit --name test ubuntu:latest
docker info | grep "Labels"
docker inspect <container id> | grep "Labels"
```


***Figure 7.41***

We notice that the container has no labels. *And as we are aware, labeling is a key component of SELinux*.

Okay, let us go ahead and enable *SELinux* on Docker by modifying the `/etc/docker/daemon.json` file and adding the following lines, or in case the `daemon.json` file is not there, creating the `daemon.json` file and putting in the following:

```
{
 "selinux-enabled": true
```

```
}
```

After making the changes, we need to restart docker by running `systemctl restart docker` for the change to take effect. We now see that '`name=selinux`' has also been added to the security options. Check out the screenshot in *Figure 7.42*:

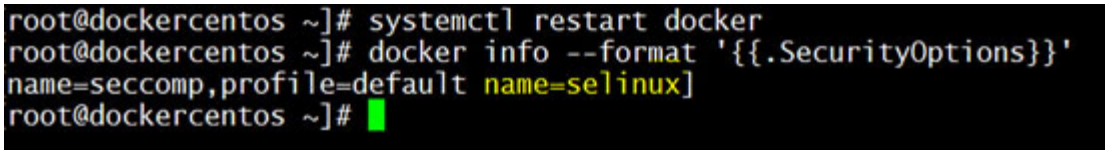



***Figure 7.42***

However, if we do `docker info | grep "Label"`, we won't see anything populated there as we can see in the following screenshot:

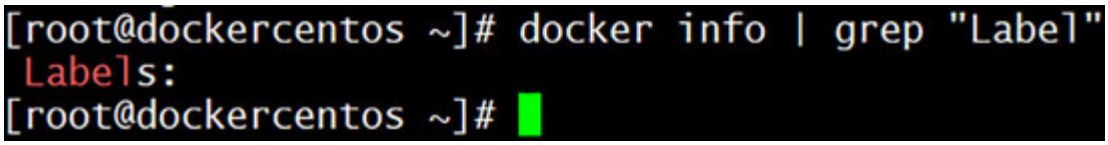



***Figure 7.43***

Now, let us create a container and see how enabling *SELinux* impacts it. Check out the screenshot in *Figure 7.44:*

```
docker run -dit --name test ubuntu:latest
docker container ls -a
docker inspect <container id> | grep "Label"
```

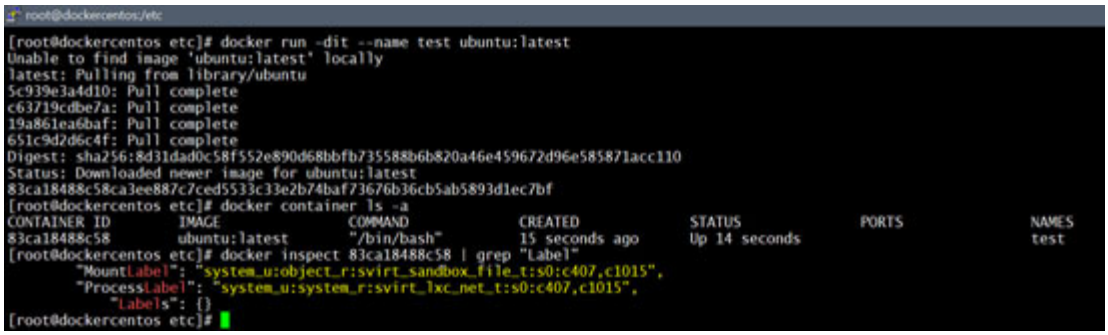



***Figure 7.44***

We observe that the container has got labeling based on the fact that *SELinux* is now enabled on the system, and Docker is implementing it. If we analyze the label, we see that it is a colon-separated list

written in the format SELinux user: SELinux role: Type: Sensitivity level: Unique category.

Let us try to understand this a little better. When we create a container with SELinux enabled, it simply means that SELinux will label the container with its version of the user, role, type, sensitivity level, as well as pick a set of two random numbers between 0 and 1023 that have to be unique. These numbers are prefixed with a c for the category, and SELinux also uses the sensitivity level s0. This is technically known as *Multi-Category Security*. The following table will give us a better understanding of the *Multi-Category Security* translation table for SELinux.

```
cat /etc/selinux/targeted/setrans.conf
```

```
#
# Multi-Category Security translation table for SELinux
#
# Uncomment the following to disable translation libary
# disable=1
#
# Objects can be categorized with 0-1023 categories defined by the admin.
# Objects can be in more than one category at a time.
# Categories are stored in the system as c0-c1023.  Users can use this
# table to translate the categories into a more meaningful output.
# Examples:
# s0:c0=CompanyConfidential
# s0:c1=PatientRecord
# s0:c2=Unclassified
# s0:c3=TopSecret
# s0:c1,c3=CompanyConfidentialRedHat
s0=SystemLow
s0-s0:c0.c1023=SystemLow-SystemHigh
```

***Figure 7.45***

From the above table, we can glean information about meaningfully using *Multi-Category Security*. But generally speaking, we can use any categorization for objects, as shown below. Check out the screenshot in *Figure 7.46*:

```
docker run -dit --name max --security-opt label=level:s0:c0,c3
busybox
docker container ls -a
docker inspect <container id> | grep "Label"
```


***Figure 7.46***

All this is done basically to ensure the compartmentalization of the containers to improve security. This compartmentalization done by *SELinux* prevents, amongst other things, one container process from attacking another container's process and contents.

The '*type*' defines the permissions and privileges the user has. For example, if we look at the "*ProcessLabel*", then the type defined there is `svirt_lxc_net_t`. It will have a set of permissions attached to it. Those are the permissions Docker uses because Docker is connected to the type `svirt_lxc_net_t`.

As part of the set of privileges, Docker gains access to `/usr/var/` and some other locations, as well as complete access to things that are labeled with `svirt_sandbox_file_t`. In other words, this means `svirt_lxc_net_t` can write to files having the label `svirt_sandbox_file_t`.

To conclude, *SELinux* is a great mechanism to enforce **Mandatory Access Control** (**MAC**) by providing isolation for container runtimes and ensuring that the containers remain secure. Even in cases where SELinux may cause issues, it is very easy to switch to Permissive mode, at which point violations are only logged, and nothing is blocked.

## Seccomp

**Secure Computing Mode** (**Seccomp**) is a feature of the kernel that allows us to restrict system calls from the container. This feature is available if the following conditions are fulfilled:

1. If Docker has seccomp inbuilt, and
2. The kernel is configured with `CONFIG_SECCOMP` enabled.

We can easily check if `CONFIG_SECCOMP` is enabled by running the following command and checking the screenshot in *Figure 7.47*:

```
grep CONFIG_SECCOMP= /boot/config-$(uname -r)
```

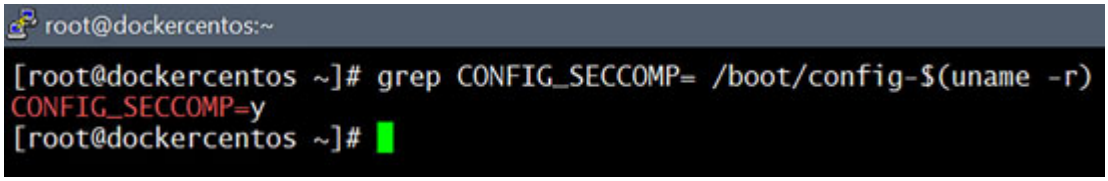



***Figure 7.47***

The default profile for Docker blocks 44 system calls out of more than 300, and while not being stringently protective, it has wide application compatibility. *https://docs.docker.com/engine/security/seccomp/* provides a table of the calls that are blocked by the default profile.

## Conclusion

In this chapter, we looked into various security features that are part of the Linux kernel and can be leveraged for docker. We also had a look at the Linux Security Modules-*AppArmor* and *SELinux*, which provide Mandatory Access Control. We dealt at length with Kernel Namespaces, cgroups, Capabilities, and Seccomp as well. We saw how these features could be leveraged to secure our containers and have the necessary isolation so that we can configure safe and secure containers.

## Points to Remember

Let us ensure we remember the following:

- Linux provides several features that help us secure our containers.
- Namespaces isolate a '*share of the operating system*' for each container.
- Control Groups (cgroups) are a mechanism to group processes together based on their resource consumption, prioritization, stability, and to an extent, their security.

- Starting with the *Linux kernel version 2.2*, the privileges and permissions associated with a super-user or root have been granularized into a separate set of capabilities that can be independently enabled or disabled on a per-thread basis.
- Linux Security Modules-*AppArmor* and *SELinux*, which act as an intermediate layer between the operating system and the security provider, helps to make the containers more secure.
- **Secure Computing Mode** (**Seccomp**) is a feature of the kernel that allows us to restrict system calls from the container.

## Multiple Choice Questions

**Choose the most appropriate answer:**

1. Docker on Linux currently uses the following number of kernel namespaces:

   a. 4

   b. 5

   c. 6

   d. 7

2. Because of the namespace isolation provided, we can map more than one container to the same port at the container level, provided they are using different ports at the host level. Which namespace are we talking about?

   a. PID

   b. Network

   c. Kernel

   d. Inter-Process Communication

3. AppArmor is most commonly used for :

   a. RHEL

   b. CentOS

   c. Fedora

   d. Ubuntu

4. SELinux is loaded at the:
    a. Daemon level
    b. Container level
    c. Swarm level
    d. Host level
5. When we create a container, most on the in-built Linux capabilities are:
    a. Modified
    b. Unavailable
    c. Available
    d. Restricted
6. SELinux is a mechanism to:
    a. Make the system more performant
    b. Restrict system calls
    c. Restrict user calls
    d. Label objects, files, networks, and so on, on the system
7. Discretionary Access Control (DAC) denial is prioritized over:
    a. AppArmor
    b. Root user
    c. SELinux
    d. Privileged user
8. Docker allows us to control the usage of:
    a. Networks and CPU
    b. Memory and CPU
    c. Networks and Memory
    d. Memory and Users

## Answer

1. c
2. b
3. d
4. a
5. b
6. d
7. c
8. b

## Questions

1. What are the options available for security from the operating system level for Docker?
2. What are kernel namespaces?
3. Explain in detail the concept of a control group (cgroup).
4. What is **mandatory access control** (**MAC**), and how is it different from **discretionary access control** (**DAC**)?
5. What is seccomp?
6. Explain the concept of capabilities.
7. How do Docker containers deal with capabilities?
8. What benefits does SELinux provide to docker containers?
9. How can you configure a non-default AppArmor profile for your container?
10. How can we control the usage of memory and CPU by a container?

## Key Terms

- Kernel Namespaces
- Control Groups
- Mandatory Access Control
- Discretionary Access Control

- Capabilities
- Seccomp
- AppArmor
- SELinux
- Isolation
- Compartmentalization

# CHAPTER 8

# Docker Security-II

## Introduction

In this chapter, we will see how to leverage the native security features available in Docker Enterprise Edition for securing our docker containers. This chapter goes deep into the security features available off the shelf in *Docker Enterprise Edition* and how we may use them to secure our containers in a better way.

## Structure

- Docker Enterprise Edition
- Installing Docker Enterprise Edition
- Security Scanning
- Role-based access control

## Objective

After studying this chapter, you ought to able to install Docker Enterprise Edition, along with *Universal Control Plane* and *Docker Trusted Registry*, and understand clearly how to leverage the security features indigenous to Docker Enterprise Edition such as security scanning as well as leverage the functionalities offered by role-based access control.

## Docker Enterprise Edition

It is available for running mission-critical applications and enterprise-grade development. The enterprise edition comes with *Docker Engine-Enterprise*, *Docker Desktop Enterprise*, a secure image registry, and an advanced management control plane.

**Mirantis Inc. has acquired the Docker Enterprise Platform business effective Nov. 13, 2019, including its products, customers, and employees. How this will affect the existing and potential customers' business is documented in *https://www.docker.com/faq-for-docker-enterprise-customers-and-partners***

Docker Enterprise has three major components:

1. Docker Engine-Enterprise
2. Docker Trusted Registry
3. Universal Control Plane

Docker Enterprise Engine is the heart of the software-it creates images and runs the containers for us.

The **Universal Control Plane** (**UCP**) manages the orchestrators like *Docker Swarm* and *Kubernetes* and helps in the deployment of the applications.

**Docker Trusted Registry** (**DTR**): This is an enterprise-grade registry for storing our images.

## Installing Docker Enterprise Edition

To install Docker Enterprise Edition, we will need a license. We can go to *https://hub.docker.com/editions/enterprise/docker-ee-trial/trial* for a trial license. As of the time of writing this book, we can get a one-month trial license and get started. We will use Ubuntu as our operating system. Before starting, we need to download the license key and save it in a secure location.

Because Enterprise Edition works in swarm mode, we need to have at least two Linux servers to ensure that we have a manager node and a worker node. We need to install the Docker Enterprise Edition is both the servers, and then we need to install the UCP on one of the servers. DTR then needs to be installed on the other server.

We start by going to the hub.docker.com and copying the URL to download our edition as shown in the following screenshot:

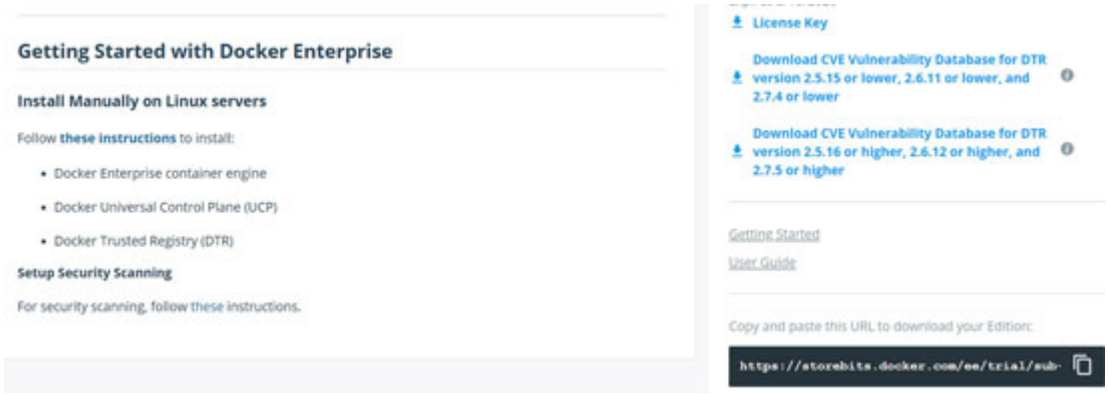



***Figure 8.1***

Next, we paste the URL, as shown in the following screenshot:


***Figure 8.2***

Then we need to go through the following steps:

If older versions of Docker called by names such as docker, *docker.io*, or *docker-engine* are running on the system, then we need to remove them before we do anything else as shown in the following command:

```
apt-get remove docker docker-engine docker-ce docker-ce-cli
docker.io
apt-get update
```

We now go ahead and install the packages to allow apt to use a repository over HTTPS.

```
apt-get install apt-transport-https ca-certificates\
curl software-properties-common -y
```

We set up the environment as follows:

```
DOCKER_EE_URL=https://storebits.docker.com/ee/ubuntu/sub-
1c10b54e-3021-46e6-a0c0-2ce2519b572a
DOCKER_EE_VERSION=19.03
```

We now go ahead and add Docker's official GPG key by running the following and checking the screenshot in *Figure 8.3*:

```
curl -fsSL "${DOCKER_EE_URL}/ubuntu/gpg" | sudo apt-key add -
```


***Figure 8.3***

We now verify that we have the appropriate key by searching for the last 8 characters of the fingerprint `DD91 1E99 5A64 A202 E859 07D6 BC14 F10B 6D08 5F96` as shown in the following command:

```
apt-key fingerprint 6D085F96
```

The following command sets up a stable repository. A stable repository is required for us to install the Docker Enterprise Edition.

```
add-apt-repository \
 "deb [arch=$(dpkg --print-architecture)] $DOCKER_EE_URL/ubuntu
 \
 $(lsb_release -cs) \
 stable-$DOCKER_EE_VERSION"
```

Check out the following screenshot:

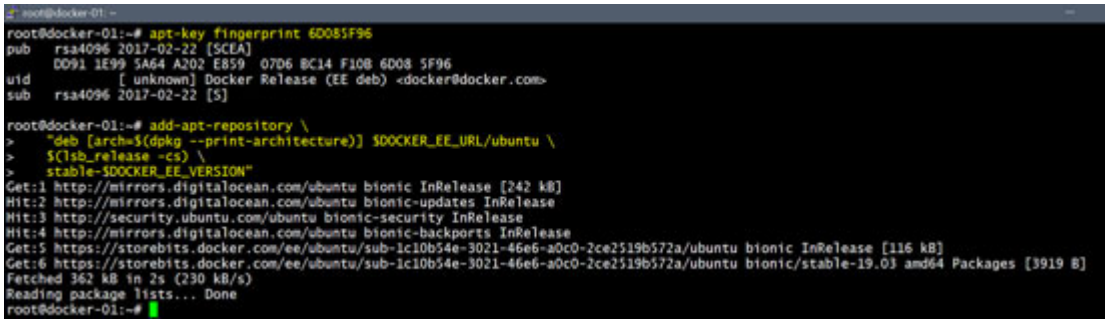



***Figure 8.4***

Next, we need to run the following set of commands:

To start with, we update the apt package index by running the following:

```
apt-get update
```

Once this is done, the next step is to install the latest version of the Docker Engine - Enterprise and containerd:

```
apt-get install docker-ee docker-ee-cli containerd.io -y
```

Now, we list the available versions in our repo by running the following:

```
apt-cache madison docker-ee
```

From the list, we now choose to install a specific version, as shown in *Figure 8.5*:

```
apt-get install docker-ee=5:19.03.5~3-0~ubuntu-bionic\
docker-ee-cli=5:19.03.5~3-0~ubuntu-bionic containerd.io
```

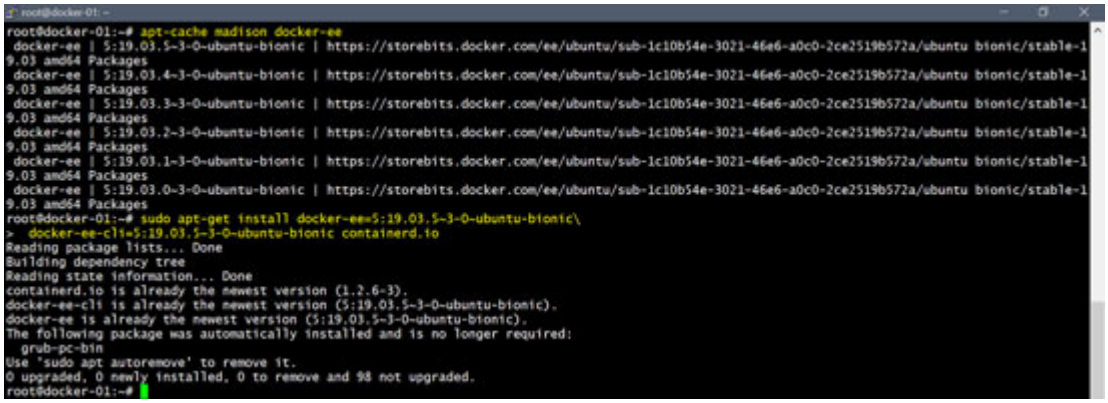



***Figure 8.5***

Finally, to check everything is fine, we run the following and check the output in *Figure 8.6*:

```
docker run hello-world
```

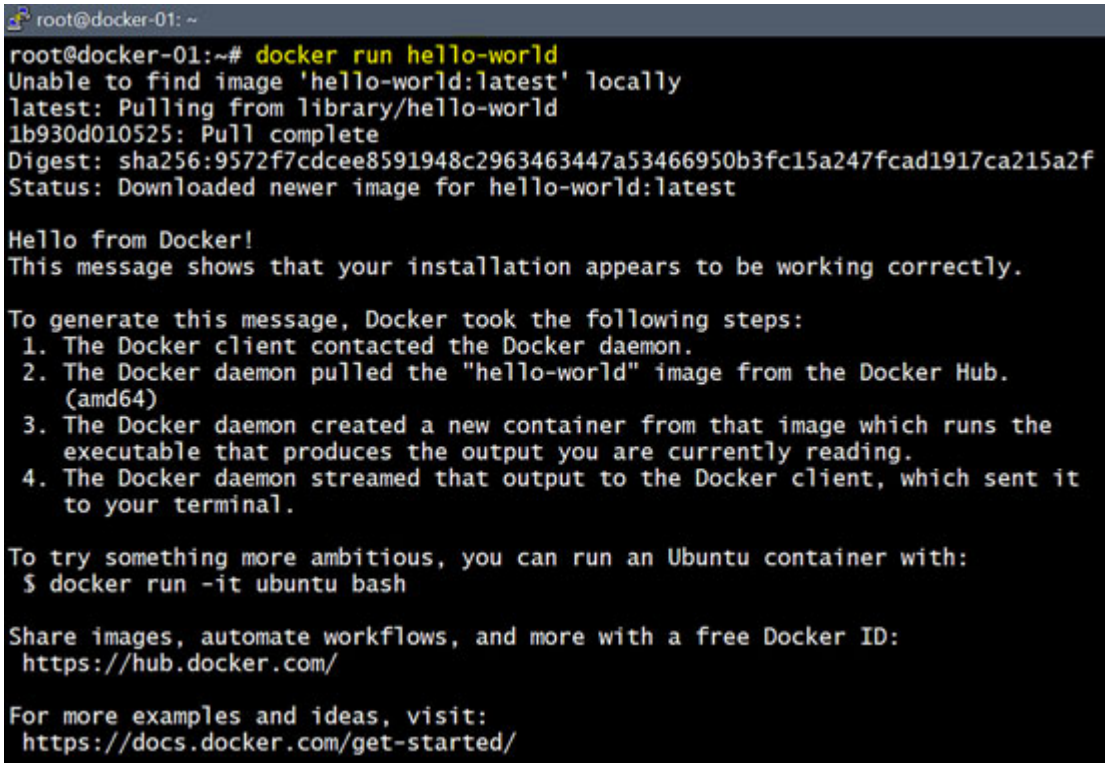



***Figure 8.6***

And to check the docker version, we run the following and check the output in the following screenshot:

```
docker version
```

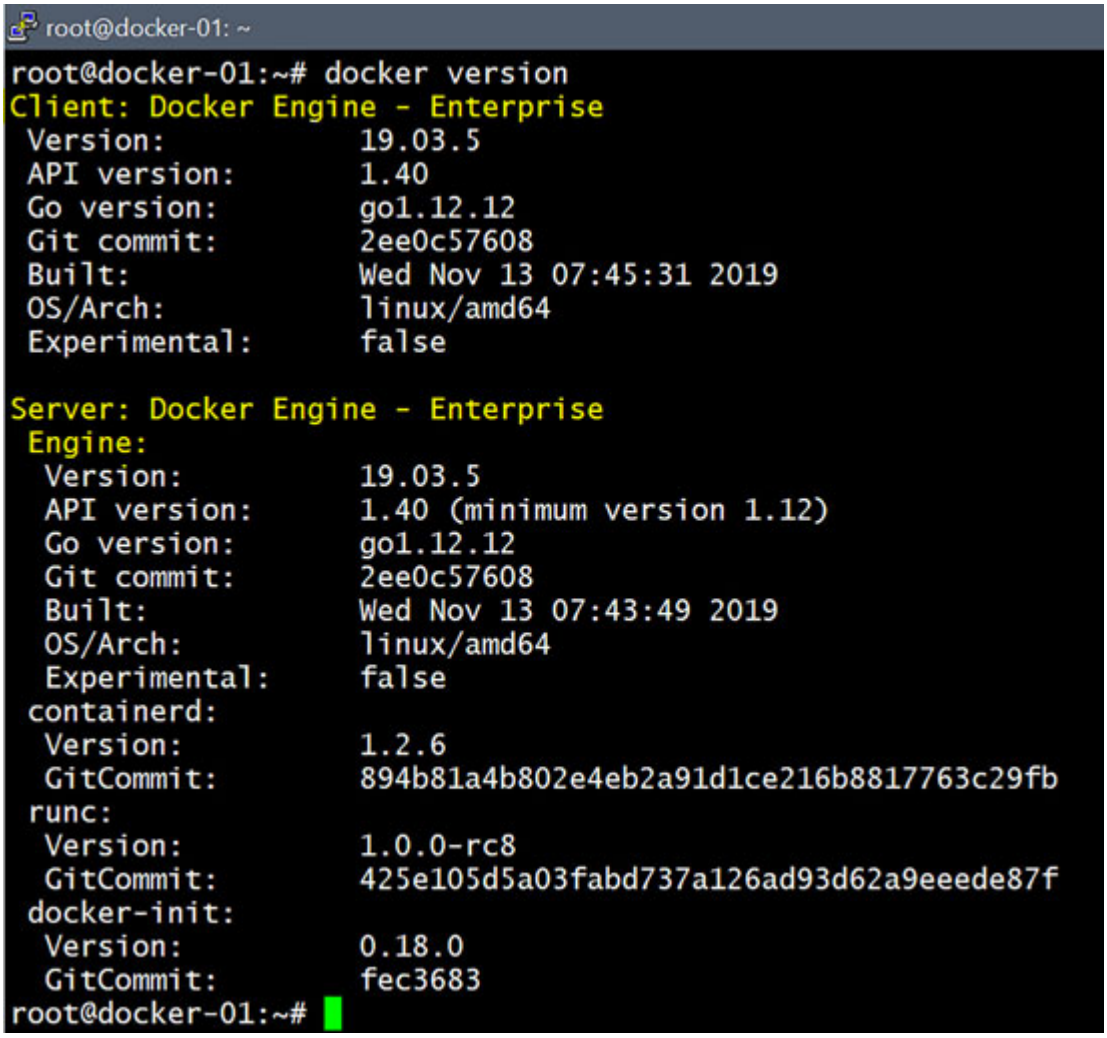



***Figure 8.7***

So, there we are! Docker Enterprise Edition is installed on our first server, which we will treat as the manager node. We will now go ahead and install Docker Enterprise Edition on the other node as well. We will follow the same steps to install the software on the other node.

Once we are done installing Docker Enterprise Edition on both the nodes, we will install the UCP on the first node, then join the two nodes into a swarm cluster and then install DTR on the second node.

## Installing Universal Control Plane

Docker **Universal Control Plane** (**UCP**) is an enterprise-grade cluster management solution that helps us manage both our cluster and applications from a single interface. With UCP installed, it becomes very easy to manage everything from a centralized place.

To install and run UCP, typically, we would require 16GB of RAM, *4vCPUs*, and 25-100GB of free disk space.

Also, the *Docker Engine - Enterprise* version should be *19.03*, and the Linux kernel ought to be *3.10* or higher. It is also expected that the node will have a static IP address for us to get going.

Let us check our IP address by running the following command and checking the output in the screenshot shown in *Figure 8.8*:

```
ifconfig -a eth0
```

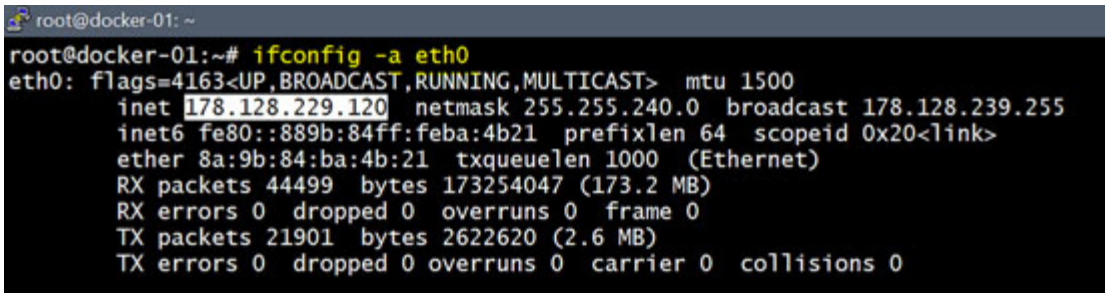



***Figure 8.8***

Next, we need to pull the latest version of UCP and install it on the system.

We pull the latest version of UCP by running the following:

```
docker image pull docker/ucp:latest
```

Install UCP-

```
docker container run --rm -it --name ucp \
 -v /var/run/docker.sock:/var/run/docker.sock \
 docker/ucp:latest install \
 --host-address 178.128.229.120 \
 --interactive
```

During the installation, we will be asked for the admin username and password. In our case, we just put ‘admin’ as the username. We also did not enter any SAN value and just pressed enter to move on.

And the installation goes on for a few minutes and through thirty-five distinct steps before it completes, as can be envisaged from the snapshot of the last few steps shown in the screenshot in *Figure 8.9*:

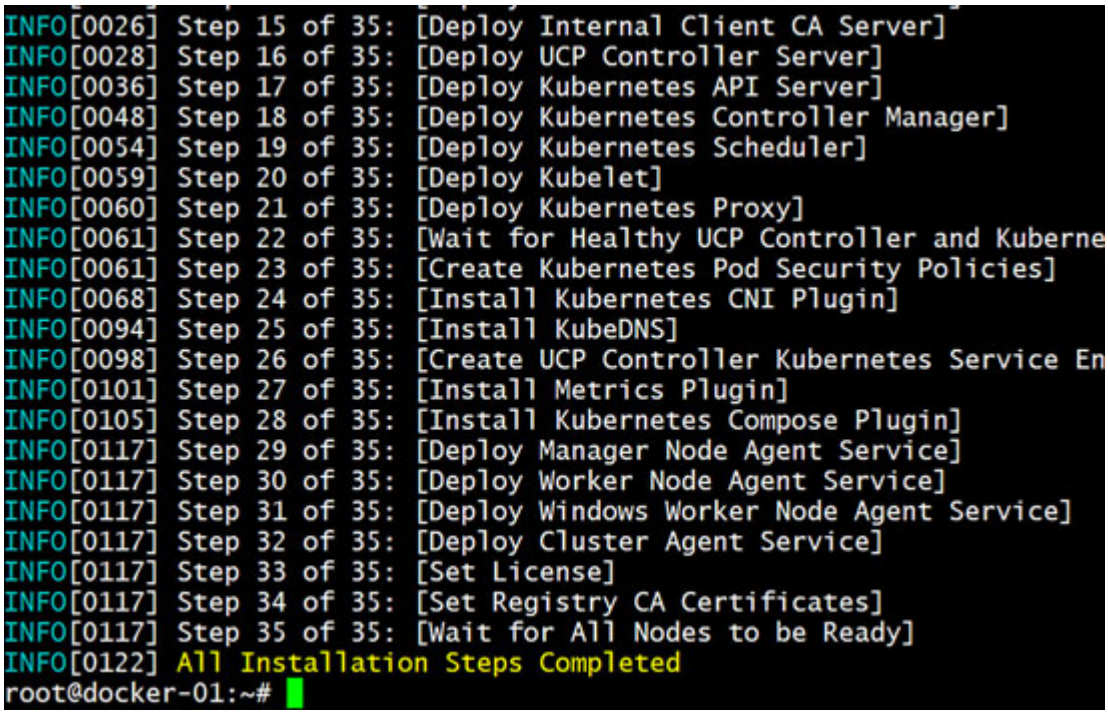



***Figure 8.9***

**UCP will automatically install Project Calico for inter-container communication for Kubernetes. We can just as easily choose to install an alternative CNI plugin, such as Weave or Flannel.**

So, with UCP installed, let us try to access it through our browser of choice at `https://178.128.229.120`. It is the public IP of the node on which we have installed UCP. We can then login with our administrator credentials and upload our license.

However, when we try to login, we will get a warning, as shown in the following screenshot. We don't have to worry about the warning though, we simply click on **Advanced** and follow the steps to go ahead and provide our administrator credentials and upload the license; after that, we should get connected. See the following screenshot:

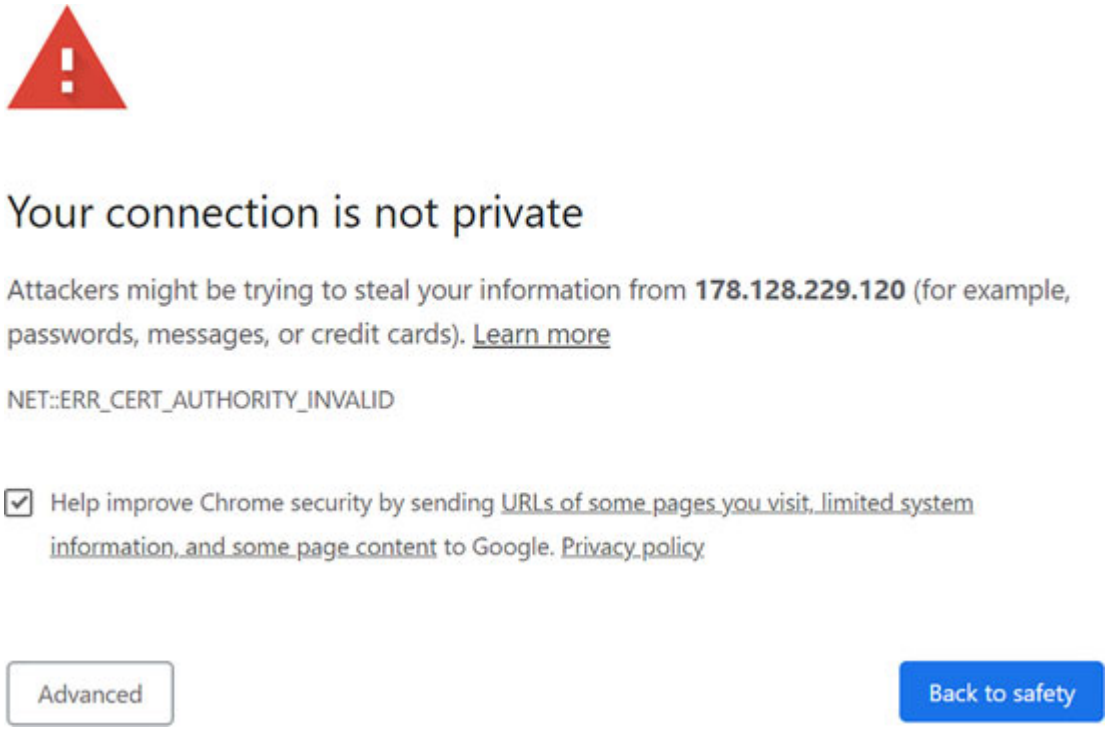



***Figure 8.10***

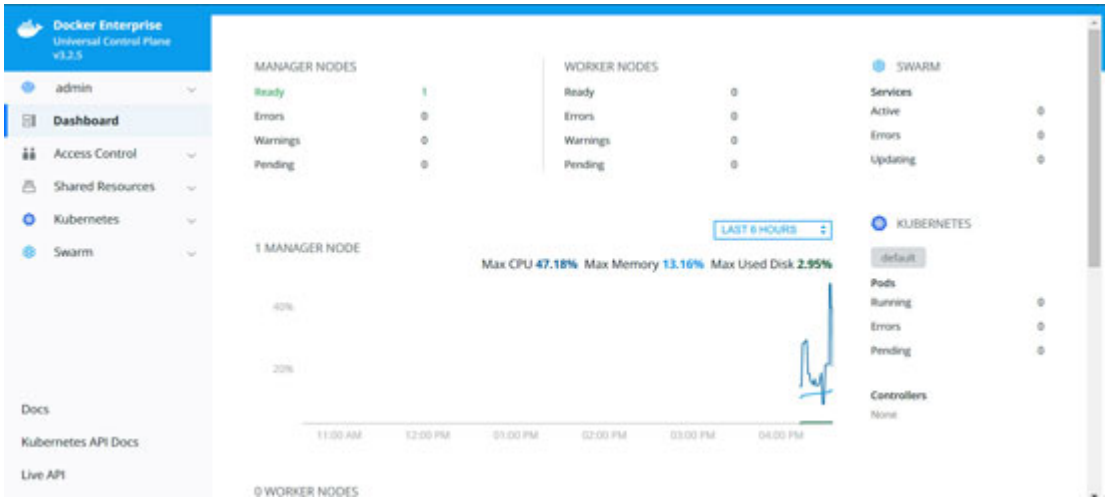



***Figure 8.11***

So, we are on the UCP home page, and our next target is to use the UCP UI to get the second node connected to the swarm cluster as a worker node.

We go to **Shared Resources|Nodes|Add Nodes**. We should get a page similar to the one shown in the following screenshot:


***Figure 8.12***

We need to copy the '*swarm join*' command and run it on the worker node to join it to the cluster.

In the following screenshot, we see the second node in our setup has joined as a worker node to the cluster:

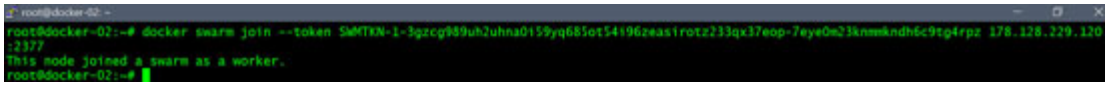



***Figure 8.13***

## Installing Docker Trusted Registry

To install **Docker Trusted Registry** (**DTR**), we need to go to **UCP|Admin Settings|Docker Trusted Registry**. Here we need to fill in certain details. In our case, we want to keep it very simple, so we aren't going with a Load Balancer, (we just put in the public IP of the second node in the box for DTR External URL) and we don't want TLS certificate verification either, so we have checked that box as can be seen from the screenshot below. Once we have made our choices, we should have a screen similar to the following screenshot:

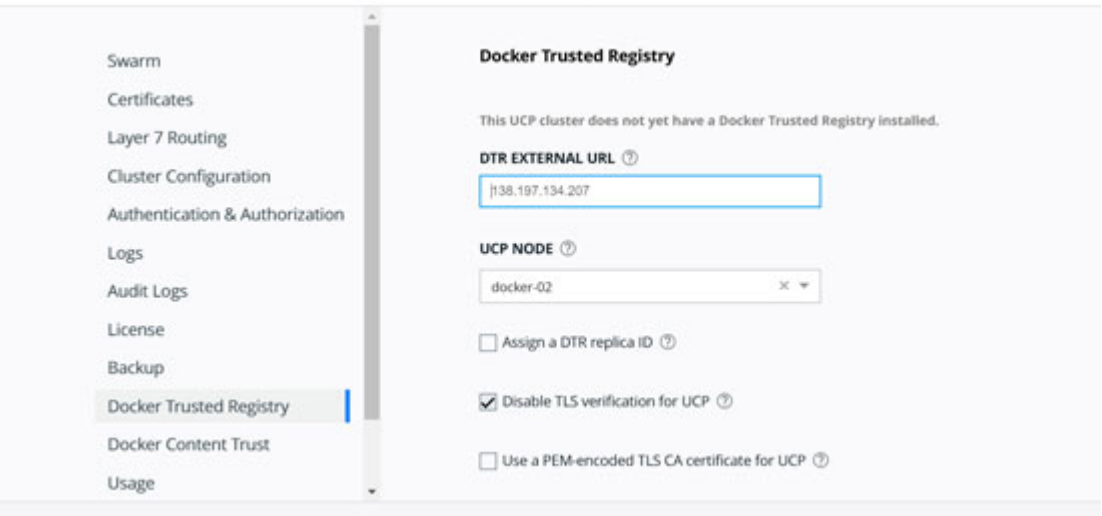



***Figure 8.14***

And if we scroll down, we will see that the command has been generated for installing DTR. See the screenshot in *Figure 8.15*:


***Figure 8.15***

We copy the command and run it from the first node. This will install DTR and register it with the UCP. Now it is time to test it out.

We open a browser of our choice and use `https://<Public_IP_of the_second_node>`, in our case, it is: `138.197.134.207`. So using `https:// 138.197.134.207` we should be able to login and land on the page shown in the following screenshot:

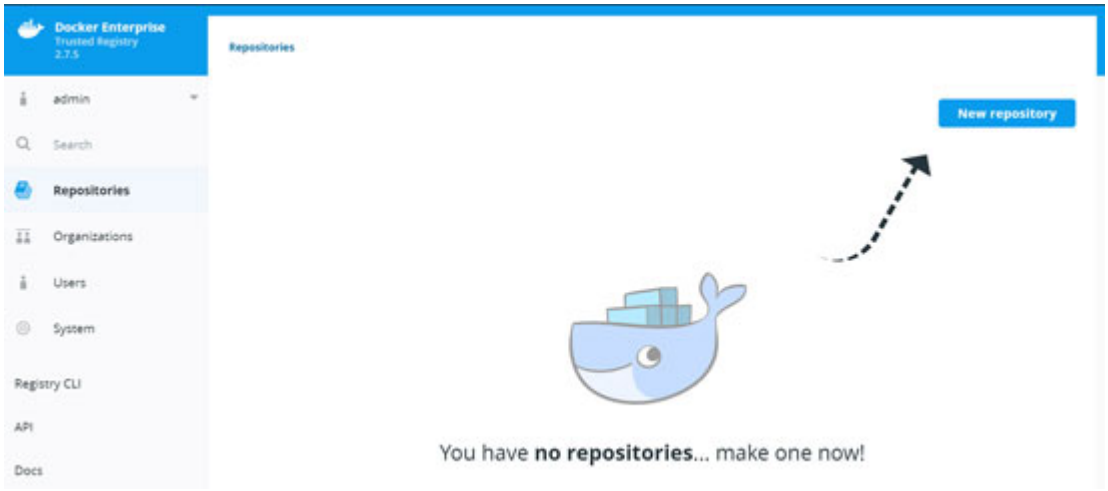



***Figure 8.16***

So there we are! We have installed UCP and DTR, and we are good to go. Let us create our first repository here-we will be using it for our experiments later.

For creating a repository, all we need to do is to click on **New Repository**, provide a name, and then click on **Create**. That's all there is to it. See the screenshot in *Figure 8.17*:

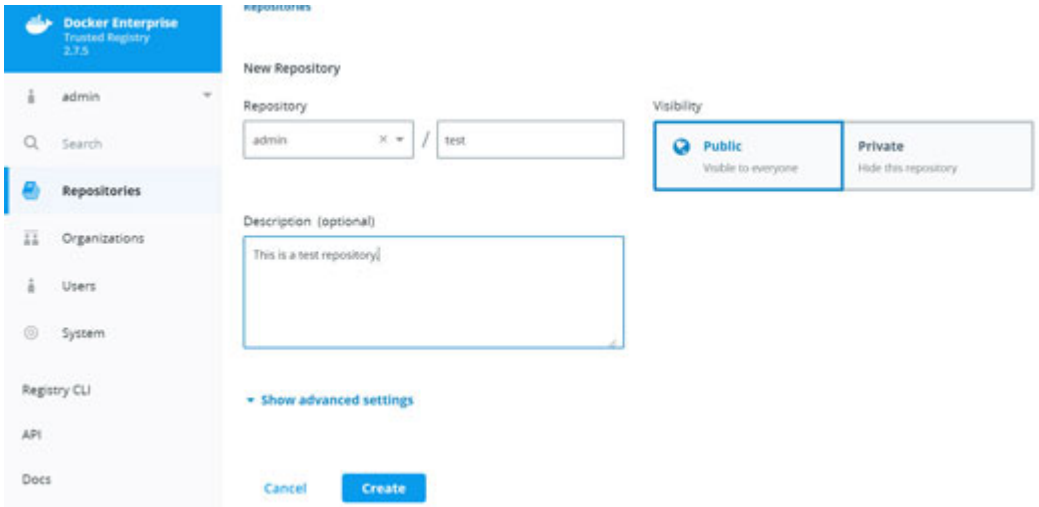



***Figure 8.17***

## Downloading and Installing the Client Bundle

Docker UCP secures our cluster with role-based access control (which we will see later in the chapter), which puts in a mechanism

whereby only authorized users can perform a change to the cluster.

So, when we run a docker command on a UCP node, that command will throw an error unless our request is authenticated by the use of a valid client certificate. That is the reason we need to generate the client certificate and download it and install it on our system.

For downloading the client certificate bundle, we have to log into UCP, and navigate to **My Profile** page as shown in the following screenshot:


***Figure 8.18***

The next step is to generate a new client bundle and copy the bundle to our node, unzip it, and run the `env.sh` to start using the certificates. This is shown in the screenshots in *Figures 8.19* and *8.20*.

```
ls -ltr
```

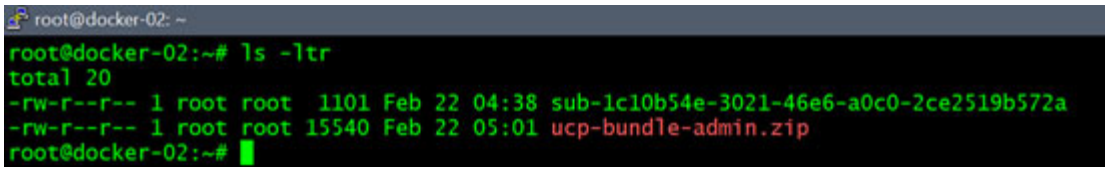



***Figure 8.19***

```
unzip ucp-bundle-admin.zip
```

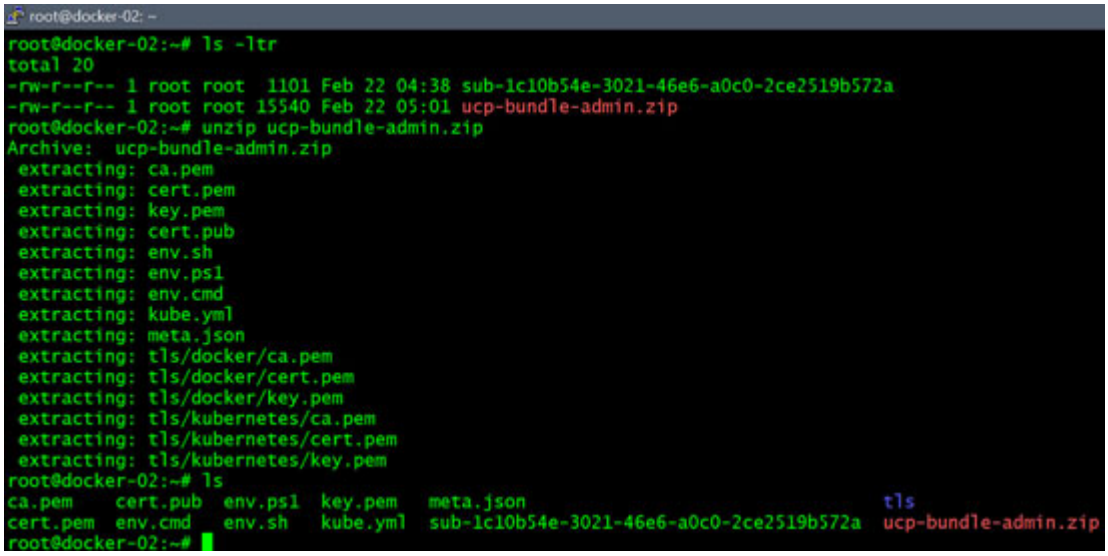



***Figure 8.20***

And now, we need to run the `env.sh` script, which will successfully install the UCP client certificate. This can easily be verified by running:

```
docker version --format '{{.Server.Version}}'
```

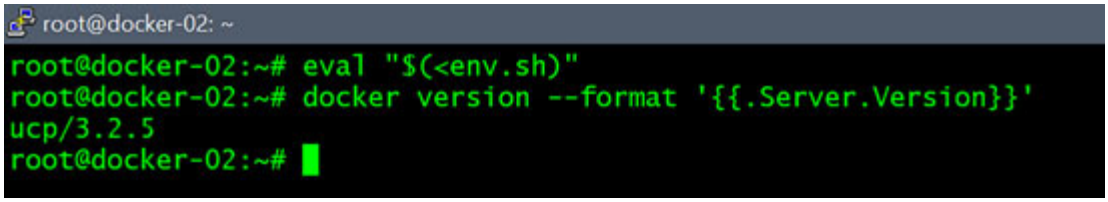



***Figure 8.21***

A client bundle generates a key pair-a private key and a public key that authorizes requests on the UCP. Remember, we will no longer interact with the Docker Engine directly, but only through the UCP. The client also contains a set of utility scripts that configures our *docker* and *kubectl* client tools so that they can interact with our UCP.

The Client Bundle utility silently updates the environment variables pertaining to the `DOCKER_HOST`, `DOCKER_CERT_PATH`, and `DOCKER_TLS_VERIFY`.

The environment variable `DOCKER_HOST` sets up the URL of the docker daemon, while the variable `DOCKER_CERT_PATH` points to the path

where the `ca.pem`, `cert.pem`, and `key.pem` files used for TLS verification, are stored. The environment variable `DOCKER_TLS_VERIFY`, if updated to any value other than an empty string, kicks off TLS verification with the docker daemon. Thus, any communication with the docker daemon through UCP will make use of the certificates available in the client bundle.

We can easily check these by running the commands on the node on which we just set up the client bundle. See the following commands and the output in the screenshot in *Figure 8.22*:

```
echo $DOCKER_HOST
echo $ DOCKER_CERT_PATH
echo $ DOCKER_TLS_VERIFY
```

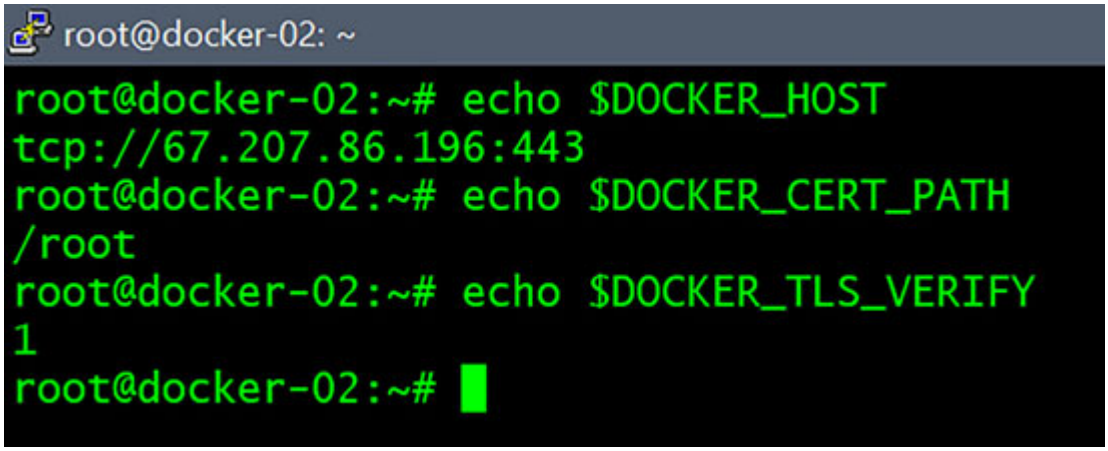



***Figure 8.22***

Similarly, we can set up things for the other node(s) also.

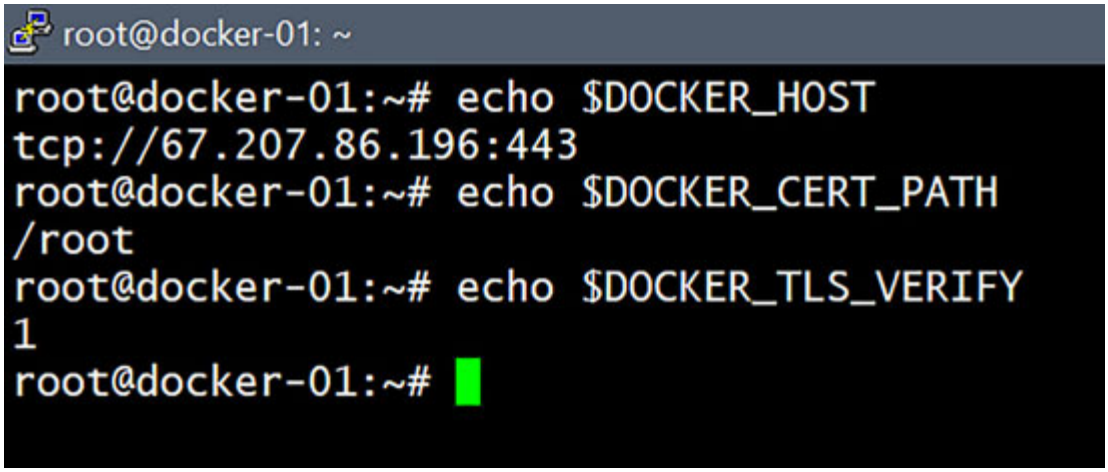


*Figure 8.23*

## Using the Security Features of Docker EE

One of the strongest reasons for running the Enterprise version is to leverage the security features available therein. And one of my personal favorites is: Docker Image Security Scanning.

Let us see how this works. To keep things simple and easy to understand, we will just pull the latest busybox image from Docker Hub, rename it and push it to the **Docker Trusted Registry** (**DTR**). We will then use the Docker Image Security Scanning feature mentioned above to scan the image for vulnerabilities. Normally, we shouldn't be seeing any vulnerabilities, as the image is the latest image of busybox and has probably been tested thousands of times.

In our second example, we will pull a very old busybox image (in which we expect to see security vulnerabilities), rename it, and push it to the DTR. This time when we run the scan, we may end up with a bunch of vulnerabilities. Let us check it out.

Before getting started, there is one more thing that we need to be aware of, and that is the **Common Vulnerabilities and Exposure Database** (**CVE Database**), which is a database of all the known vulnerabilities. This database gets updated from time to time as DTR performs checks for new vulnerabilities and downloads them and applies them without interrupting any scans in progress.

However, if we don't have access to the internet, then to update the CVE Database, we will need to download and install a tar file that contains the database updates. So, let us do a quick demo to understand this. We run the following sequence of commands:

```
docker pull busybox:latest
docker tag busybox:latest
<DTR_IP_ADDRESS>/<username>/<repositoty_name>: busybox
docker login --username <username> < DTR_IP_ADDRESS>
docker push <DTR_IP_ADDRESS>/<username>/<repositoty_name>:
busybox
```

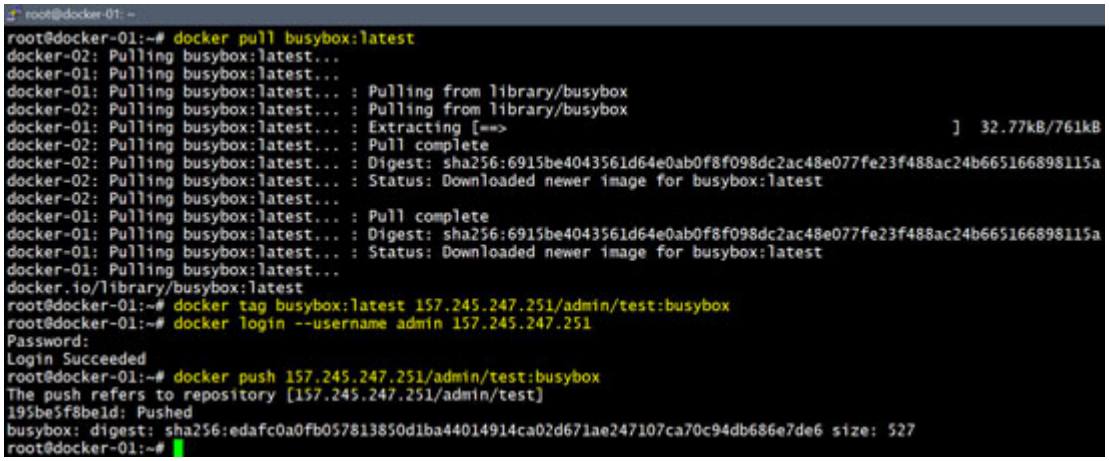



***Figure 8.24***

On the DTR page, if we go **Repositories|admin/test|Tags**, we should see a screen similar to the one shown below. Our image now tagged as '*busybox*' is ensconced therein, as shown in the screenshot in *Figure 8.25*:

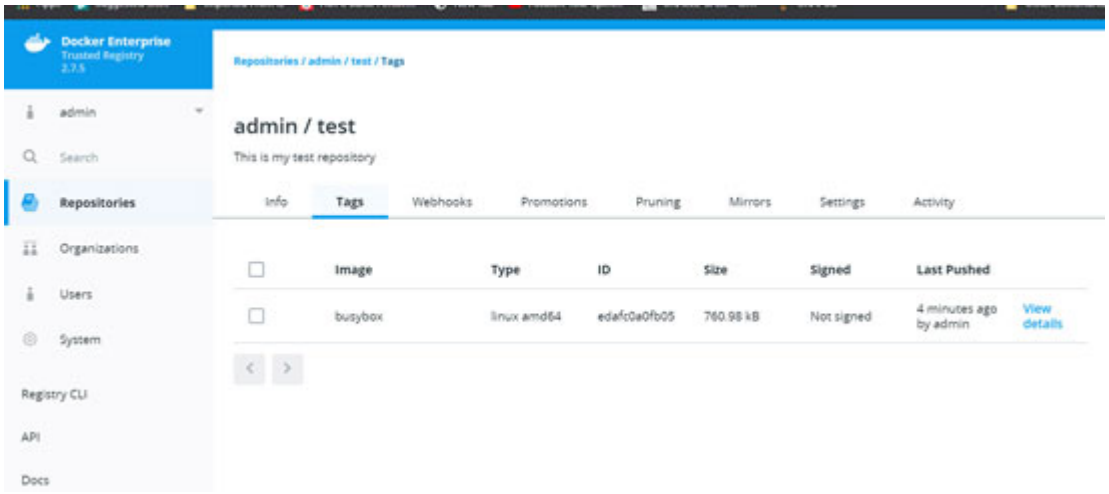



***Figure 8.25***

If we go to the **System|Security** page, we will see that scanning is not enabled at this point. We will now enable scanning and also click on **Enable Online Syncing**, which will allow for the online updating of the CVE database. See the following *Figure 8.26*:

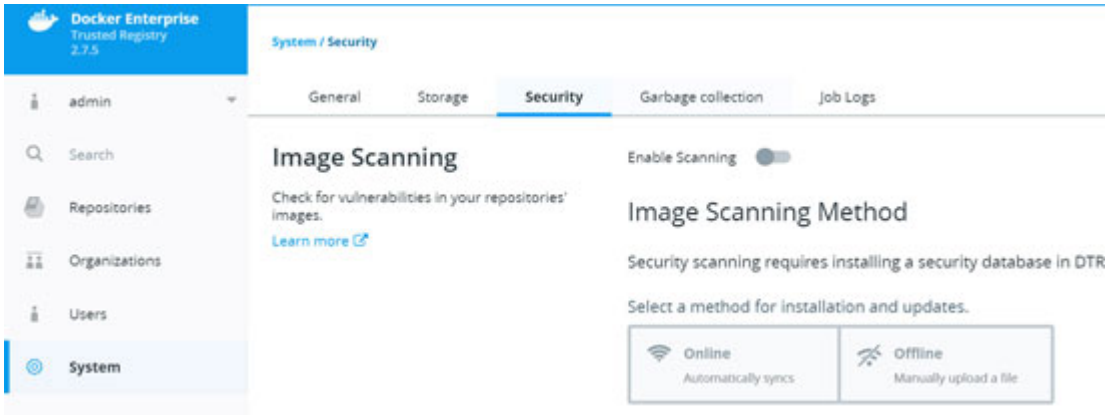



***Figure 8.26***

Now we can start a scan as shown in the following screenshot:

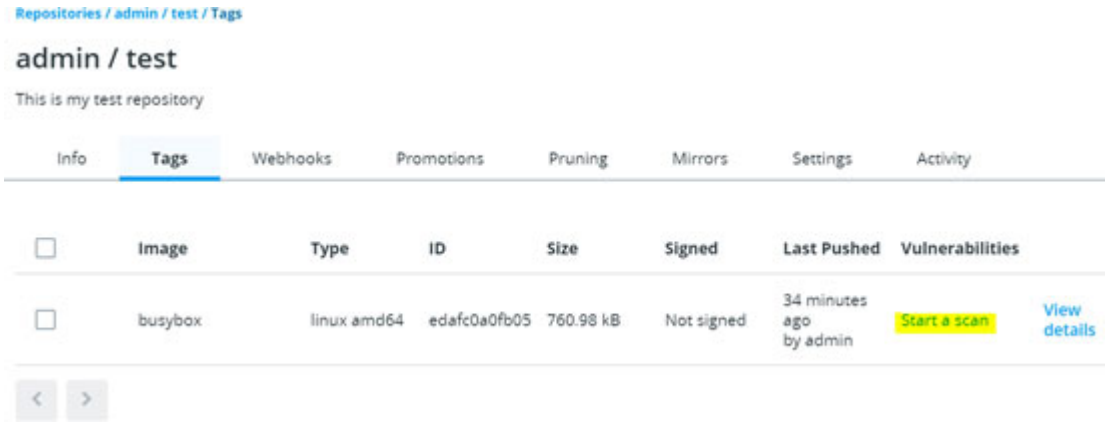



***Figure 8.27***

As expected, once the scan completes, we see that the image is just fine, and there are no security vulnerabilities discovered during the scan. Re: *Figure 8.28*.

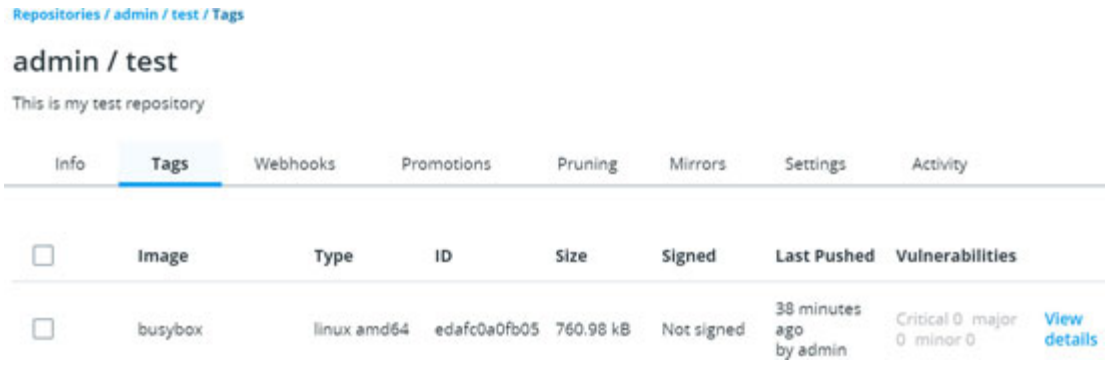



***Figure 8.28***

Now, as per our plan, let us pull an old image of the busybox and check whether the security scan uncovers any vulnerabilities for us. See the screenshot in *Figure 8.29*:

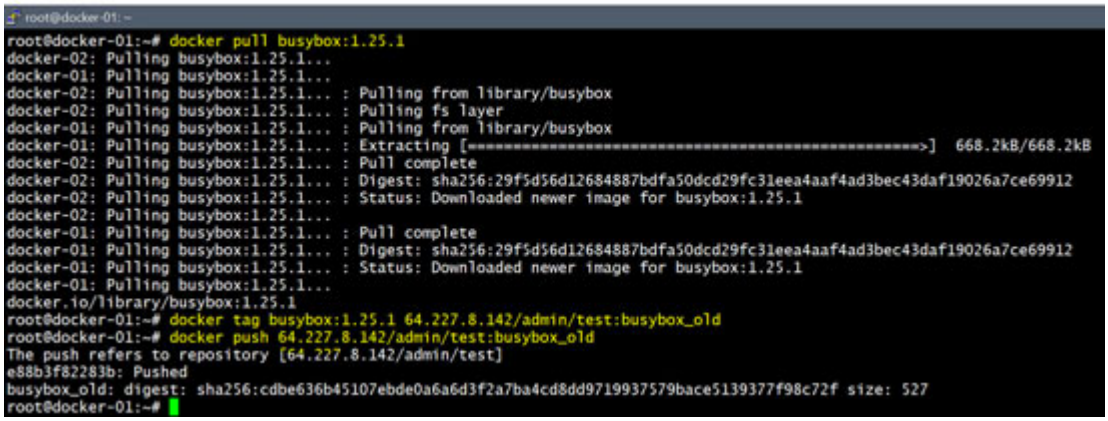



***Figure 8.29***

As expected, we now see the image `busybox_old` available in our DTR as evidenced by the following screenshot:

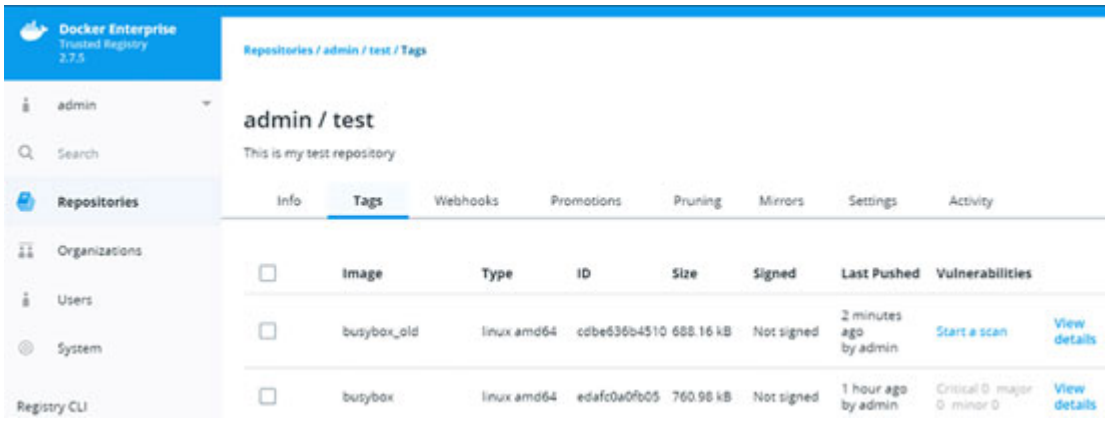



***Figure 8.30***

Let us start the scan now and see what happens. Check the screenshot in *Figure 8.31*.

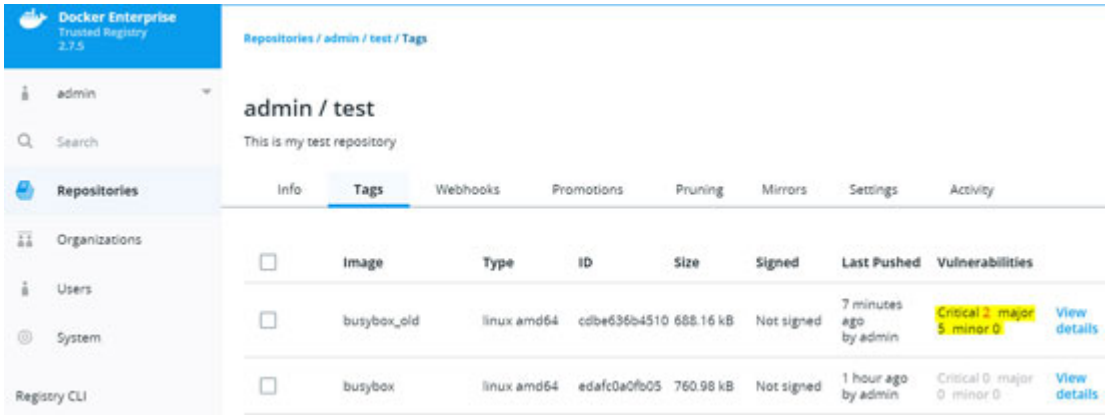



***Figure 8.31***

It is clear that there are two major and five minor security vulnerabilities, and if we drill down into those, we will see the details, and as we know, the devil lies in the detail. Re: *Figure 8.32*:

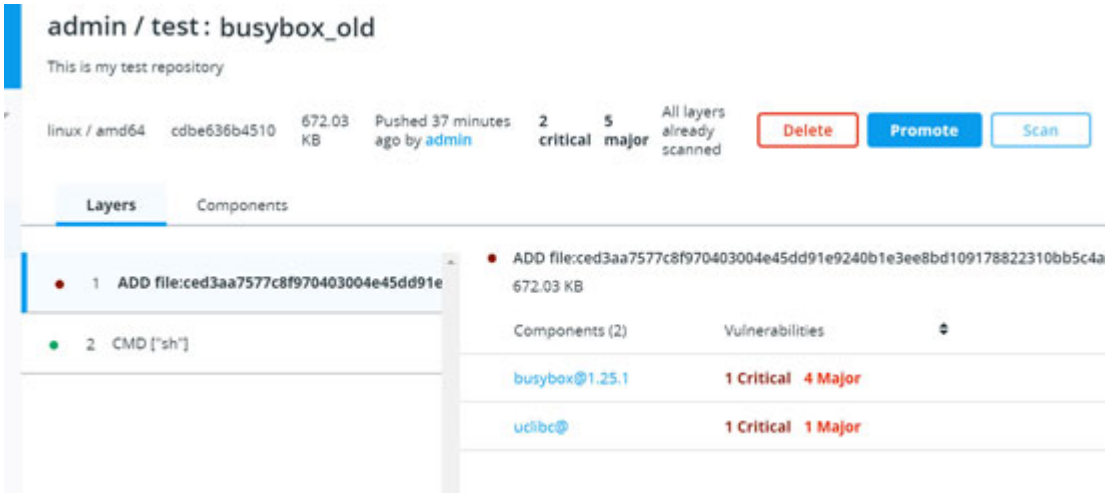



***Figure 8.32***

The following screenshot shows exactly in which layer the vulnerabilities are found.

And as we drill down deeper, we can see a lot of what is not right with this image. Refer *Figure 8.33*:

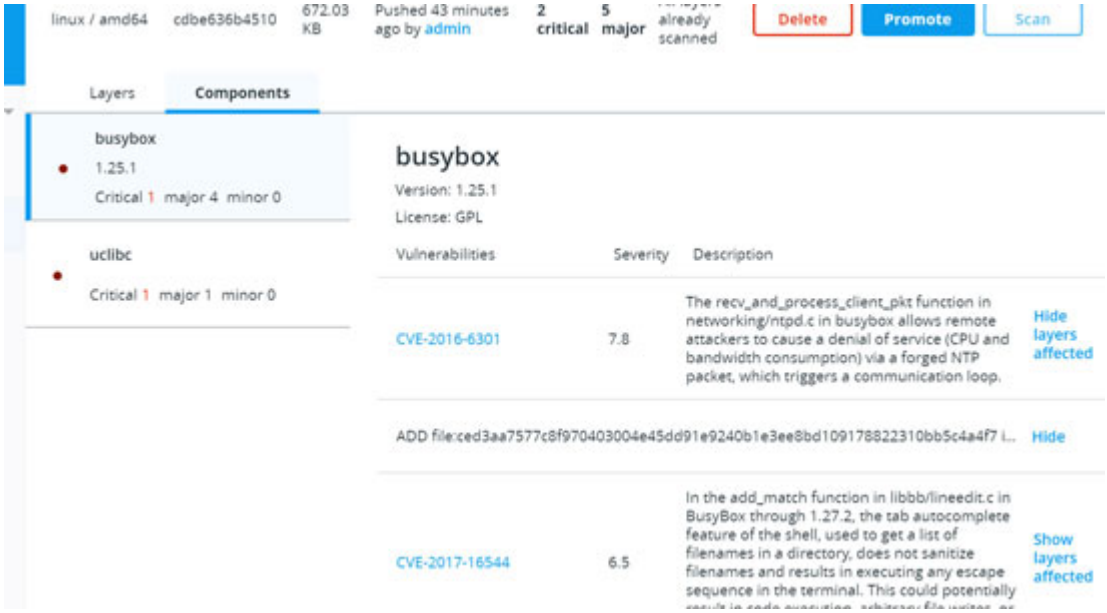



***Figure 8.33***

So, now we ought to be clear with the mechanism of Docker Image Security Scanning.

It is possible to automate to a security scan whenever we push an image by going to **Repositories|<Repository_name>|Settings** and enabling the scan on push, as shown in the following *Figure 8.34*:

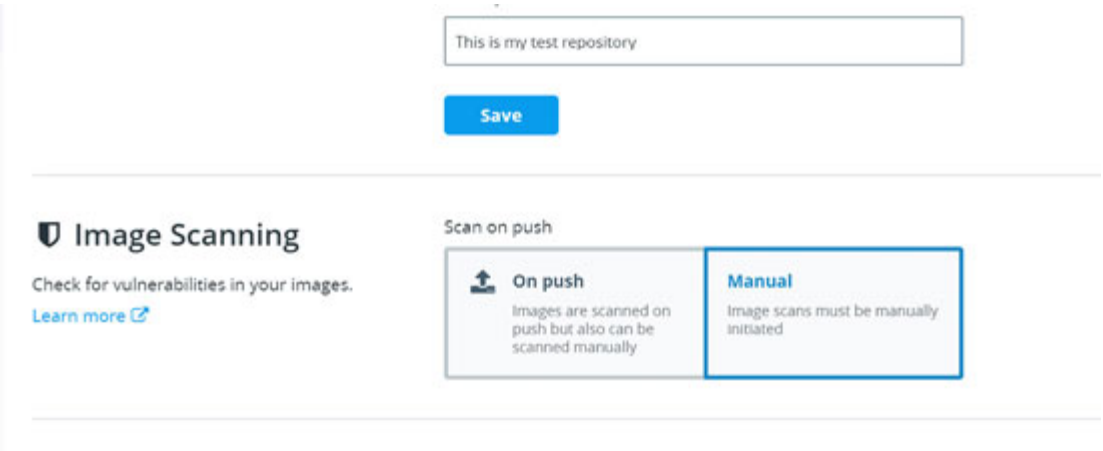



***Figure 8.34***

## Role-Based Access Control

The beauty of the Docker Universal control plane is that it allows us the privilege to create users with different levels of permission on the

resources of the swarm-like images, services, networks, volumes, and so on.

Typically, we can control who (subject) has how much and what kind of access (role) to a set of resources (collection). So, we can create individual users or a group of users and give them a set of permissions, thereby exercising fine-grained access control.

Again, typically we will have an administrator doing all of these tweakings for user and group related access. Getting into the nitty-gritties, subjects are divided into:

- Users
- Organizations
- Teams

A user is someone using the UCP and can be assigned to teams in an organization or more than one organization.

An organization is a group of users sharing the same set of permissions and privileges.

A team is also a group of users sharing the same set of permissions and privileges, but a team can be considered as a subset of the organization. So, an organization can consist of several teams, and a team can belong to more than one organization.

There is a default permission level for each user to access the swarm. Creating a user is very easy. We go to the UCP UI and navigate to the Users page and click on create to create a user. In this case, we don’t want to create this user as an admin. Check out the screenshot in *Figure 8.35*:

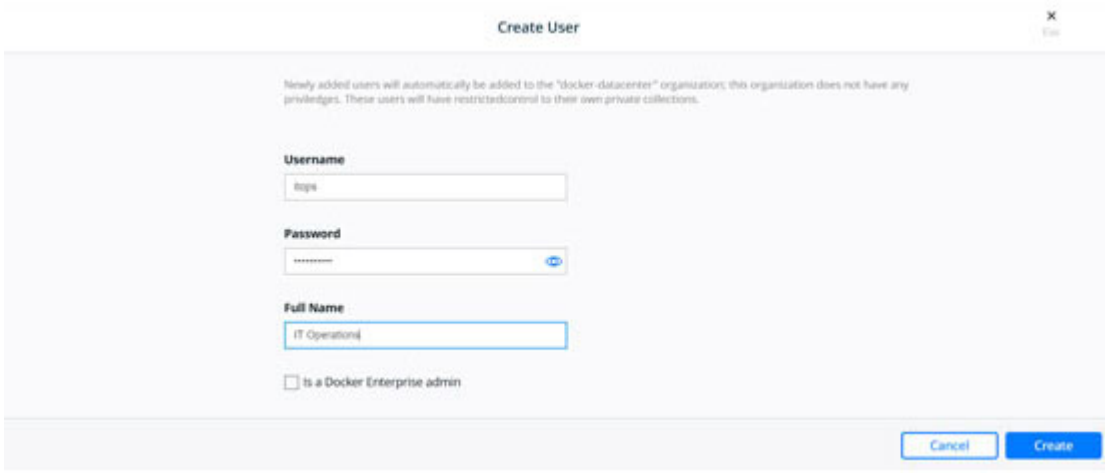



***Figure 8.35***

The new user is created and is currently active and can be seen in the screenshot in *Figure 8.36*:

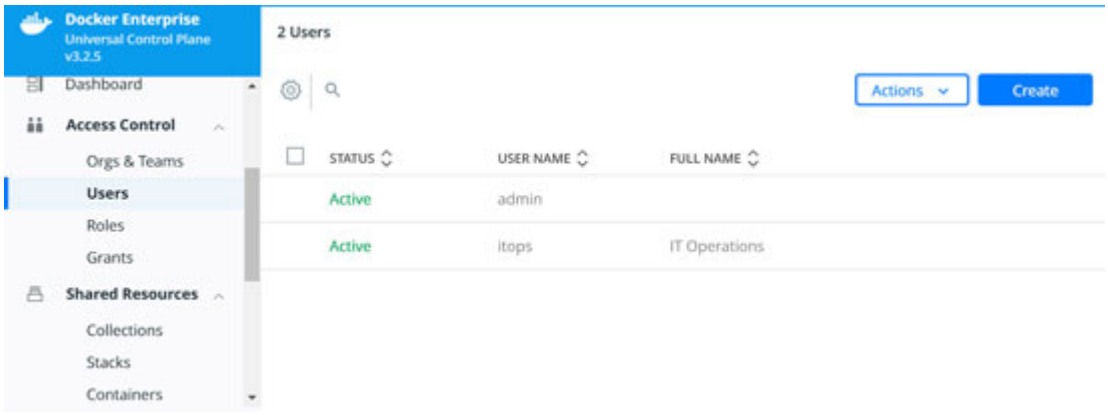



***Figure 8.36***

So far, so good.

We can just as easily create an organization and add a set of users to it. Let us create an organization by the name of operations, and we plan to add a user to it. See the screenshot in *Figure 8.37*:

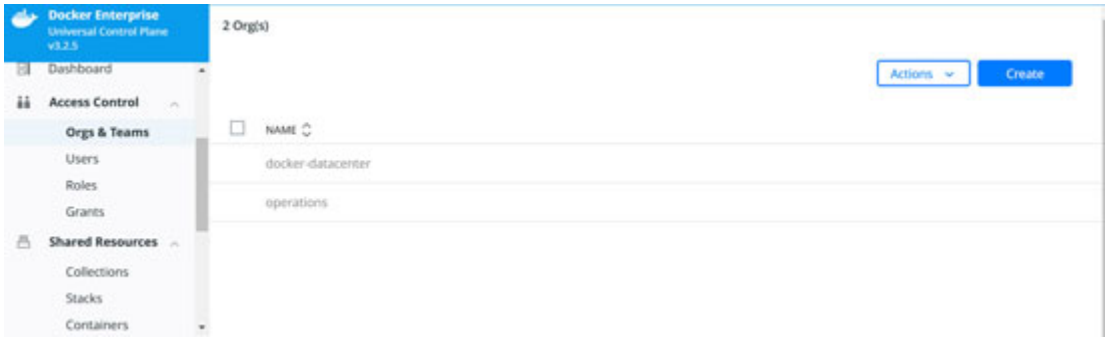



***Figure 8.37***

But to do so, as a prerequisite, we need to create a team mapped to the organization operations. See the following screenshot:

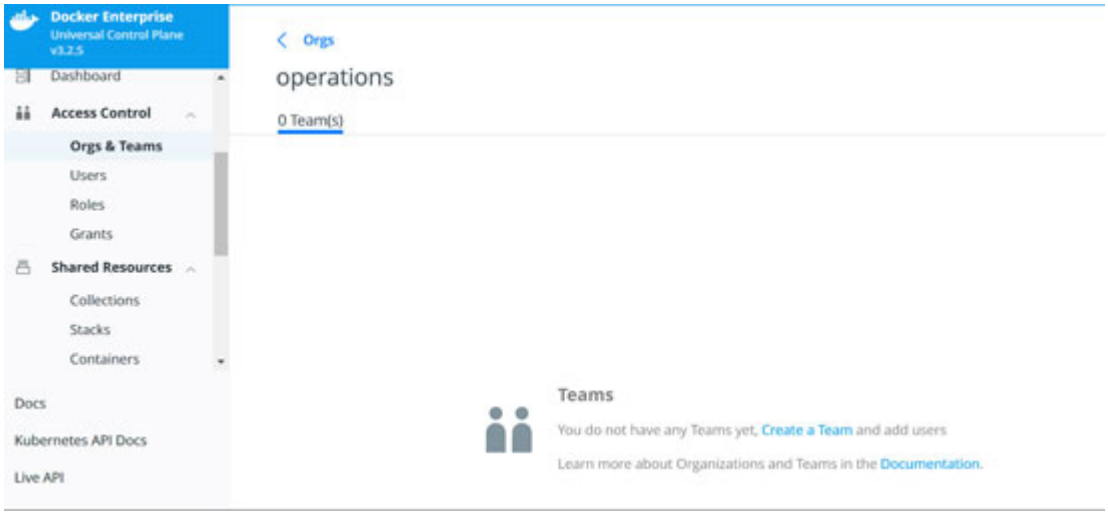



***Figure 8.38***

After creating the team named `operations_team`, we are going to add a user. Refer to the screenshot in *Figure 8.39*, where we add the team named `operations_team`.

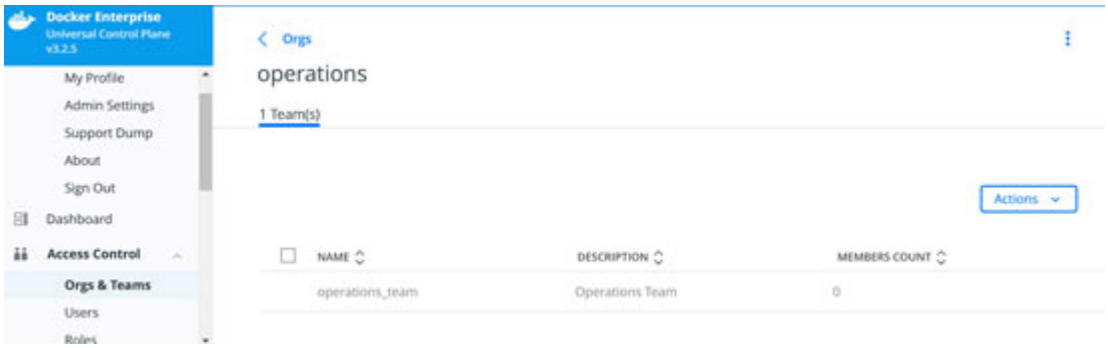



***Figure 8.39***

Refer to the following screenshot where we add the user `itops`.

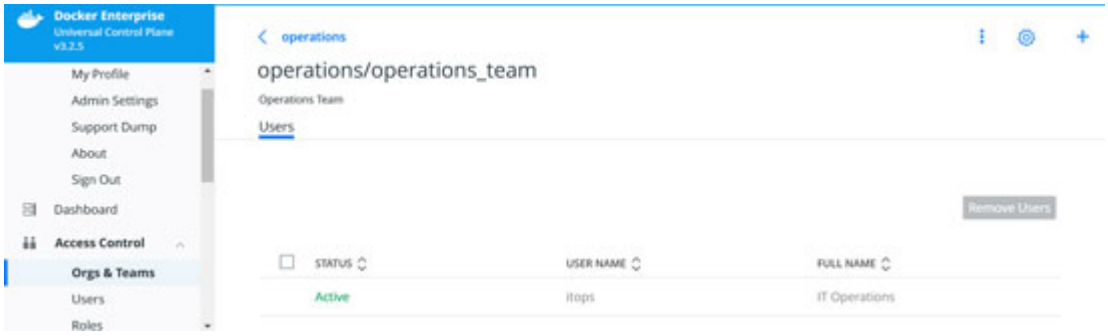



***Figure 8.40***

So let us take a step back and see where we are.

We created an organization named operations, a team under that organization named `operations_team`, and then added a user named `itops` to it.

Now that we have caught our breath let us move ahead.

As we have mentioned briefly earlier, roles are a set of privileges and permissions to access a resource. Both *Docker* and *Kubernetes* have several predefined roles available. The predefined roles for Docker Swarm are shown in the following screenshot:

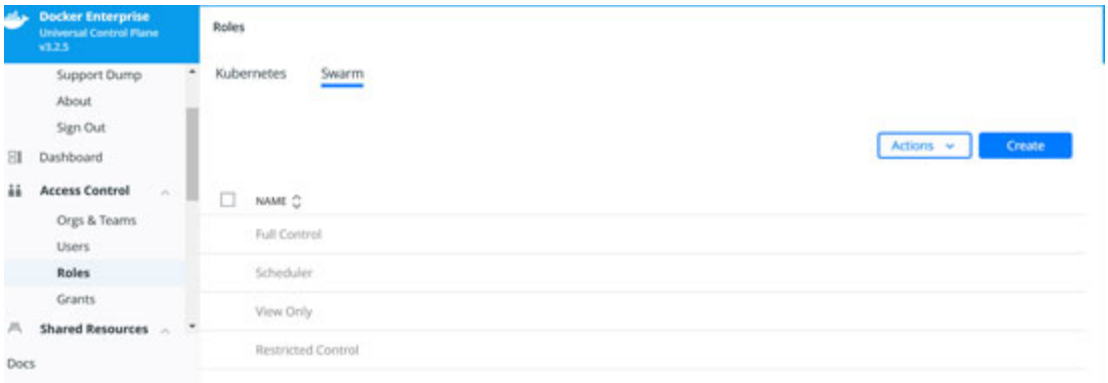



***Figure 8.41***

The following table explains the predefined roles:

| Role Name | Description | Remarks |
|---|---|---|
| View Only | Users can only view a resource but cannot create a resource. | This is very limited permission, which allows a user only to |

| | | |
|---|---|---|
| | | view a resource and nothing else. |
| Restricted Control | Permission is granted to view and edit resources, but there is a restriction on running a service or container impacting the node on which it is running. For example, the user is prevented from running a container in privileged mode or with additional kernel capabilities | While this role is much more permissive than the View-Only role, it still limits the way the users can use the resources. For example, a user cannot exec into a container or mount a node directory in the container. So, there are obvious limitations to this role. |
| Scheduler | The users can schedule workloads as well as view the nodes. | While users can schedule workloads, they cannot view them. For viewing them, they need separate permission like Container View. |
| Full Control | This is the most unfettered role. The user can view and edit all the granted resources. | This role allows for the creation of containers but does not allow a view of containers created by others. |

***Table 8.1***

We had also earlier very briefly mentioned Collections. Collections are the resources on which the subject (i.e., users) will work, and they will have a set of permissions on that collection. For example, we may have three resources (collections), namely, dev, testing, and production, and the user `itops` may have different types of roles for each of the resources.

To understand it better, let us create a custom collection and a custom role and then create a grant which grants the custom collection and custom role to the user itops.

### Step 1) We create a custom collection

We go to **Collections/Swarm/Shared/Private|Create Collection**. We name the collection `My_collection`, and it is created successfully as it is evidenced by the following screenshot:

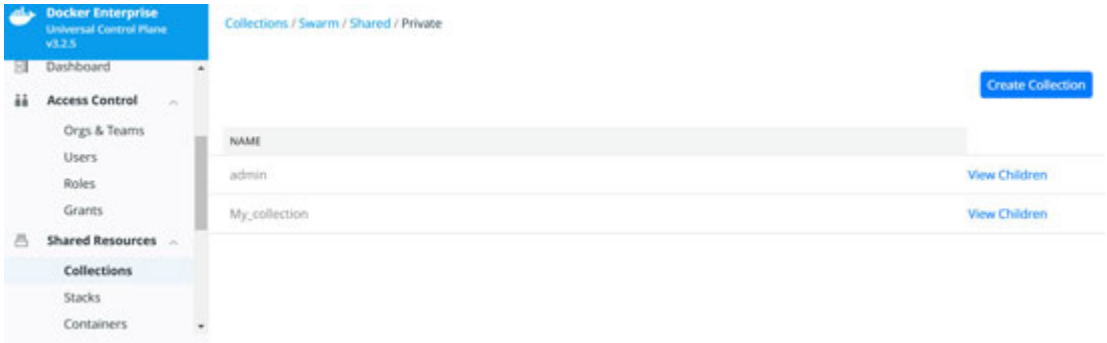



*Figure 8.42*

**Step 2) We create a custom role.**

We go to **Roles|Swarm|Create|Names|Operations**, and then we put a checkmark against all the boxes, thereby granting this role all the available operations and then click on **Create**, and that is it.

We now have a role by the name of `My_role` created as we can see in the following screenshot:

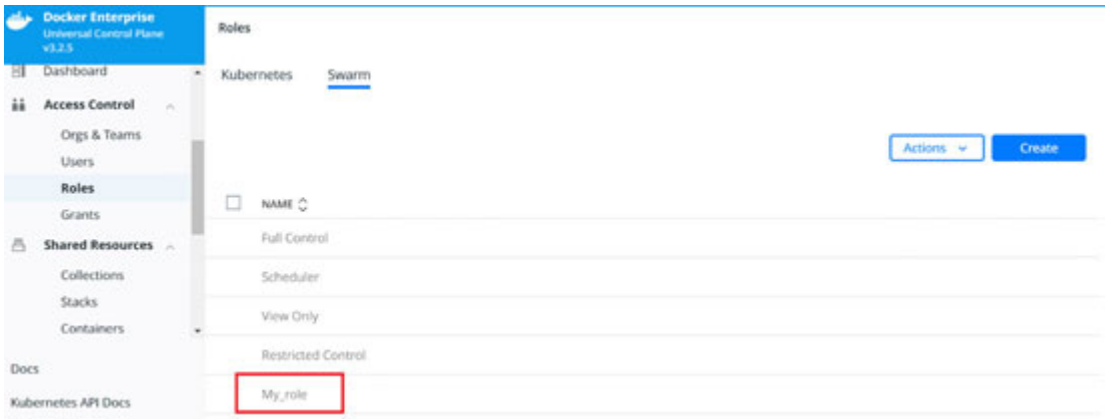



*Figure 8.43*

**Step 3) Now, let us go ahead and create the grant.**

**Grant|Swarm|Subject** (we choose **IT Operations** from the drop-down list)|**Next|Resource** set (navigate to `My_collection`)|**Role** (select `My_role` from the drop-down list **Create**)

This will create the grant, and we check it out from the following screenshot:

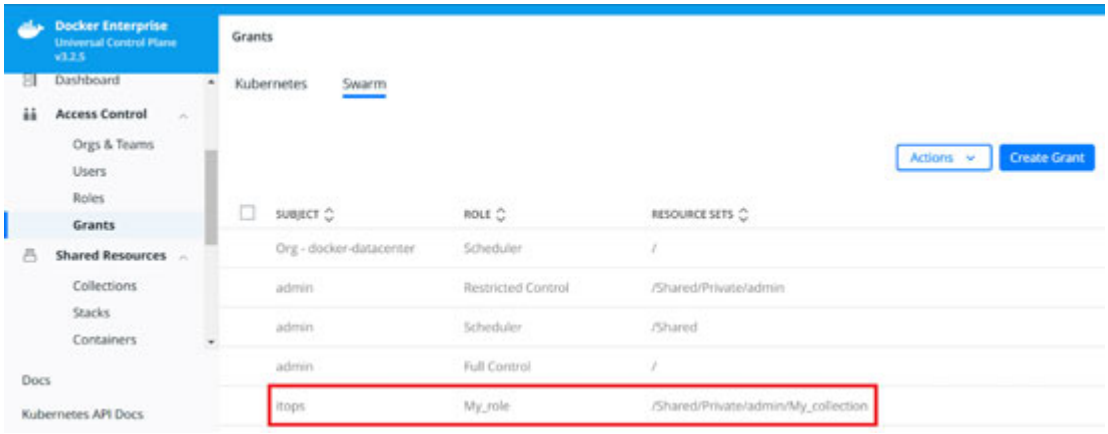



***Figure 8.44***

So, what we have effectively done is that we have granted the user `itops` a custom role called `My_role` on a resource set named `My_collection`, which we created.

Thus, we see that we can exercise control over who can access what resource and how that resource may be used by the user. This is the crux of role-based access control.

Before we move ahead, we ought to be aware that docker has some built-in collections available which may be used instead of creating our collections:

| Default Collection | Description | Remarks |
|---|---|---|
| / | This is a kind of catch-all path to all resources in a swarm cluster. Resources that are not in any other collection are put in here. | A residual place-holder for all resources which are not part of any other collection. |
| /System | This points to the UCP manager(s) and DTR system services | By default, admins have this access. |
| /Shared | This points to all worker nodes for scheduling. | The worker nodes can also be moved and isolated. |
| /Shared/Private/ | This points to a user's private collection(s) | Private collections are not initialized until the user logs in to the node the first time. |

| | | |
|---|---|---|
| /Shared/Legacy/ | This points to collections of legacy versions | UCP 2.1 and lower. |

***Table 8.2***

## Conclusion

In this chapter, we learned about the Docker Enterprise Edition from the ground up; went through all of the steps for installing it on our Ubuntu operating system. We also installed the **Universal Control Plane** (**UCP**), and the **Docker Trusted Registry** (**DTR**) as well. Additionally, we went through the process of downloading the client bundle and installing it on our system. We saw some practical examples of security scanning and how it can benefit us from the perspective of validating an image that we are pulling in from a registry. We also walked through an example of another security mechanism known as role-based access control. Overall, this chapter provided us a strong grounding on the use of Docker Enterprise Edition and its various features, especially from the perspective of establishing security controls on our system.

## Points to Remember

- Docker Enterprise Edition consists of the Docker Enterprise Container Engine, Docker Universal Control Plane, and Docker Trusted Registry.
- Installing the Docker Enterprise Edition requires a license, a trial version for one month, is available for download at the time of writing of this book.
- Docker Enterprise Edition needs swarm mode to run, so we need to have at least two servers available.
- If older versions of Docker called by names such as docker, docker.io, or docker-engine are running on the system, then we need to remove them before we install Docker Enterprise Edition.
- Docker Universal Control Plane is an enterprise-grade cluster management solution that helps us manage both our cluster

and application from a single interface. With UCP installed, it becomes very easy to manage everything from a centralized place.

- Docker Trusted Registry is an enterprise-grade registry for storing our images.
- A client certificate is required to be generated, downloaded, and installed on our system, because otherwise, when we run a command on a UCP node, the command will throw an error.
- The Client Bundle utility silently updates the environment variables pertaining to the `DOCKER HOST`, `DOCKER_CERT_PATH`, and `DOCKER_TLS_VERIFY`.
- Security scanning is a nifty feature of the Docker Enterprise Edition, and this is used to validate an image that is being pulled from a registry.
- Universal control Plane allows us the privilege to create users with different levels of permission on the resources of the swarm-like images, services, networks, volumes, and so on.
- Role-based access control provides us with a mechanism to control who (subject) has how much and what kind of access (role) to a set of resources (collection).
- We can create individual users or a group of users and give them a set of permissions, thereby exercising fine-grained access control.

## Multiple Choice Questions

**Choose the most appropriate answer:**

1. The CVE database is a database

   a. Of all security patches available for docker

   b. Of the latest security patches available for docker

   c. Of all the fixes for security breaches in docker

   d. A dynamic database that gets updated from time to time with all the known vulnerabilities

2. Which of the following is true for a Universal Control Plane?
    a. Universal Control Plane makes the system more performant
    b. Universal Control Plane allows for security patches to be applied dynamically
    c. With Universal Control Plane installed, it becomes very easy to manage everything from a centralized place.
    d. Universal Control Plane builds more stability into a system
3. Which of the following is true for a Docker Trusted Registry?
    a. Docker Trusted Registry is available with Docker Community Edition
    b. **Docker Trusted Registry** (**DTR**) is the enterprise-grade image storage solution from Docker
    c. Docker Trusted Registry makes the system more performant
    d. Docker Trusted Registry is not supposed to give you an insight into exposure to security threats
4. Which of the following statements is true?
    a. Docker and Kubernetes have no predefined roles available
    b. Docker has predefined roles available, but not Kubernetes.
    c. Both Docker and Kubernetes have predefined roles available
    d. Kubernetes has predefined roles available, but not Docker
5. Which of the following is true about an organization?
    a. An organization is a group of users sharing the same set of permissions and privileges
    b. An organization is a group of users sharing the same set of permissions and privileges but needs to belong to a team
    c. An organization is a group of users not necessarily sharing the same set of permissions and privileges but needs to belong to a team
    d. None of the above

6. The Client Bundle utility silently updates the environment variables pertaining to:
    a. docker_host, docker_certification_path, docker_tls_verify
    b. docker_host_path, docker_cert_path, docker_tls_verify
    c. docker_host, docker_cert_path, docker_tls_verify
    d. docker_host, docker_cert_path, docker_tls_verification
7. If you don't have access to the internet, which of the following can be used to update the CVE database?
    a. You download and install a jar file that contains the database updates
    b. You download and install a script file that contains the database updates
    c. You download and install a tar file that contains the database updates
    d. You download and install a zip file that contains the database updates
8. Which of the following is not a default collection?
    a. /Shared/Legacy/
    b. /Shared/Private/
    c. /System
    d. /System/Shared/
9. Which of the following is not a default role?
    a. Role only
    b. View only
    c. Full control
    d. Restricted control
10. Which of the following is a correct statement?
    a. Basically, a client bundle generates a key pair-a private key and a public key that authorizes requests on the UCP

b. Basically, a client bundle generates a key pair-a private key and a secondary key that authorizes requests on the UCP.
c. Basically, a client bundle generates a key pair-a private key and a public key that authorizes requests on the DTR
d. Basically, a client bundle generates a key pair-a private key and a public key that authorizes requests on the Docker Engine

## Answers

1. d
2. c
3. b
4. c
5. a
6. c
7. c
8. d
9. a
10. a

## Questions

1. What are the main components of the Docker Enterprise Edition?
2. What is the main function of the Universal Control Plane?
3. What is the main function of the Docker Trusted Registry?
4. What is the purpose of the Client Bundle?
5. What are the environment variables that are updated by the Client Bundle?
6. What is the role of a CVE?
7. What do you understand by Role-Based Access Control?
8. What do you understand by users, organizations, and teams?

9. What is the role?
10. What is a collection?

## Key Terms

- Docker Enterprise Edition
- Universal Control Plane
- Docker Trusted Registry
- Client Bundle
- Repository
- Docker Swarm
- CVE Database
- Security Scanning
- Roles
- Teams
- Collections
- Organizations