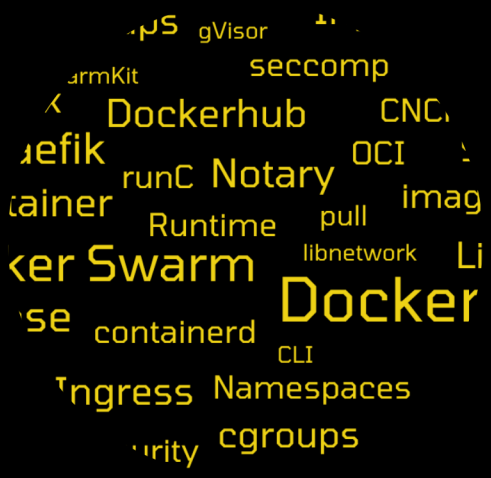
2024 Edition



Painless Docker

Unlock The Power Of Docker & Its Ecosystem

A practical guide to mastering Docker and its ecosystem with real-world examples.

Aymen EL Amri

www.faun.dev

Painless Docker

Unlock the Power of Docker and its Ecosystem

Aymen El Amri @eon01

This book is for sale at http://leanpub.com/painless-docker

This version was published on 2023-12-03

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Aymen El Amri @eon01 by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Looking forward to read ”Painless Docker: Unlock The Power Of Docker + Its Ecosystem”

The suggested hashtag for this book is #PainlessDocker.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#PainlessDocker

Also By Aymen El Amri @eon01

Saltstack For DevOps

OpenAI GPT For Python Developers

Cloud Native Microservices With Kubernetes

LLM Prompt Engineering For Developers

Contents

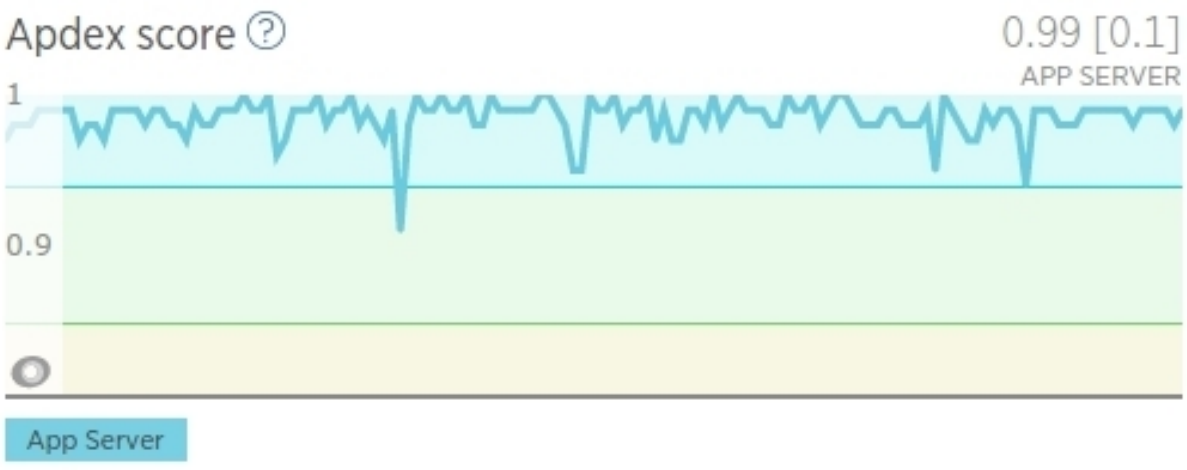
Preface

Docker is a powerful tool that can greatly enhance the management and optimization of your production servers. However, without a proper understanding of some key concepts, it can quickly become complex. In this guide, I will explain these concepts through theory and practical examples to help you effectively use Docker in your development, testing, and production environments.

The rapidly changing ecosystem of containers can present challenges and confusion for operations engineers and developers. As containerization technology continues to evolve, it is important to stay up-to-date and adapt to these changes. This technology is transforming the way system engineering, development, and release management have traditionally worked, making it a crucial aspect of modern IT.

Google, a leading tech company, relies heavily on containerization technology. According to Google, they use Linux containerization to run everything in their cloud, starting over two billion containers per week. This efficient technology allows for the sharing of parts of a single operating system among multiple isolated applications, making it ideal for massive distributed applications.

Initially, there was uncertainty surrounding the future of Docker. However, after testing and experimenting with Docker, I made the decision to use it in production. This choice proved to be invaluable. Recently, I created a self-service in my startup for developers - an internal scalable PaaS. Docker played a significant role in achieving a 14x improvement in production metrics and meeting my goal of a service with a 99% Appdex score and SLA.



Appdex score

Using Docker was not the sole factor in achieving success; it was part of a larger strategy. Implementing Docker allowed for smoother operations and transformation of the entire stack. It

facilitated continuous integration, automation of routine tasks, and provided a solid foundation for creating an internal PaaS.

Over time, computing has evolved from central processing units and mainframes to the emergence of virtual machines. Virtual machines allowed for the emulation of multiple machines within a single hardware environment. With the rise of cloud technologies, virtual machines transitioned to the cloud, eliminating the need for extensive physical infrastructure investments.

As software development and cloud infrastructures continue to grow, new challenges arise. Containers have gained significant traction as they offer solutions to these challenges. For example, maintaining consistent software environments across development, testing, and production is crucial. Containers provide a lightweight and portable solution, enabling the distribution of identical environments to R&D and QA teams.

Containers address the issue of unused libraries and unnecessary dependencies. By containing only the necessary OS libraries and application dependencies, containers significantly reduce the size of the application. A Node.js application that would take up 1.75 GB in a virtual machine can be reduced to a fraction of that size using optimized containers. This allows for more efficient resource utilization and the ability to run multiple applications per host.

Containers, particularly Docker, offer sophisticated solutions to modern IT challenges. Understanding and effectively utilizing Docker can revolutionize the way you manage and optimize your production servers.

To Whom Is This Guide Addressed?

To developers, system administrators, QA engineers, operation engineers, architects, and anyone faced to work in one of these environments in collaboration with the other or simply in an environment that requires knowledge in development, integration, and system administration.

Historically, the worlds of developers and sysadmins operated in parallel silos, each with its own distinct mindset and set of challenges.

Developers were often driven by the mandate to innovate and deliver new features. They perceived their role as one of creating code and applications tailored for optimal functionality. On the other hand, system administrators were the gatekeepers of stability and security. Their primary objective was to ensure that machines functioned efficiently and securely, a task that often required meticulous maintenance and optimization.

Yet, in many organizations, these differing priorities created an inherent tension:

From one side, sysadmins would sometimes view developers with skepticism, critiquing them for producing code that is resource-intensive, potentially insecure, or ill-suited for the existing hardware infrastructure.

Developers might feel stifled by system administrators, seeing them as overly cautious, resistant to change, and perhaps even out-of-step with the rapid pace of technological advancement.

These divides were further exacerbated by communication barriers and the costs associated with errors. In an environment without the cohesive tools and practices we see today, misunderstandings were widespread, and system outages could lead to a blame game. Was it the new code that caused the crash, or was it a misconfiguration on the system's end? Such questions often remained unanswered, fueling the divide.

In this landscape, the stage was set for the emergence of DevOps and containerization - a movement and technology that would seek to bridge these gaps and foster a new era of collaboration.

No more mutual accusations, now with the evolution of software development, infrastructure, and Agile engineering, the concept of DevOps was born.

DevOps is more a philosophy and a culture than a job (even if many of my previous roles were called "DevOps"). By admitting this, this role seeks closer collaboration and a combination of different roles involved in software development, such as the role of developer, responsible for operations, and responsible for quality assurance. The software must be produced at a frenetic pace while at the same time, the development in cascade seems to have reached its limits.

- If you are a fan of service-oriented architectures, automation, and the collaboration culture
- if you are a system engineer, a release manager or an IT administrator working on DevOps, SysOps or WebOps
- If you are a developer seeking to join the new movement

No matter what your Docker level is, through this guide, you will first learn the basics of Docker then move easily to more advanced concepts like Docker internals and real use cases.

I believe in learning led by practical, real-world examples, and you will be guided through all of this guide by tested examples.

How To Properly Enjoy This Guide

This guide provides technical explanations and practical use cases. Each case includes an example command or configuration to follow.

The explanations give you a general idea, and the accompanying code provides convenience and helps you practice what you're reading. It's recommended to refer to both parts for a better understanding.

When learning a new tool or programming language, it's normal to encounter difficulties and confusion, even after becoming more advanced. If you're not accustomed to learning new technologies, you may have a basic understanding while progressing through this guide. Don't worry, everyone has experienced this at least once.

To start, you can try skimming through the guide while focusing on the basic concepts. Then, attempt the first practical manipulation on your server or laptop. Occasionally, return to this guide for further reading on specific subjects or concepts.

This guide doesn't cover every aspect of Docker, but it does present the most important parts to learn and even master Docker and its rapidly expanding ecosystem. If you come across unfamiliar words or concepts, take your time and conduct your own online research.

Learning is usually a sequential process, where understanding one topic requires understanding another. Don't lose patience. You will encounter chapters with well-explained examples and practical use cases.

As you progress, try to demonstrate your understanding through the examples. And don't hesitate to revisit previous chapters if you're unsure or have doubts.

Lastly, be pragmatic and keep an open mind when facing problems. Finding a solution begins by asking the right questions.

Join the community

FAUN is a vibrant community of developers, architects, and software engineers who are passionate about learning and sharing their knowledge. If you are interested in joining FAUN, you can start by subscribing to our newsletter at faun.dev/join[1].

[1] https://faun.dev/join

The Missing Introduction to Containerization

In recent years, starting in 2015, cloud and distributed computing skills have become increasingly in demand. They have shifted from being niche skillsets to more prominent ones in the global workforce.

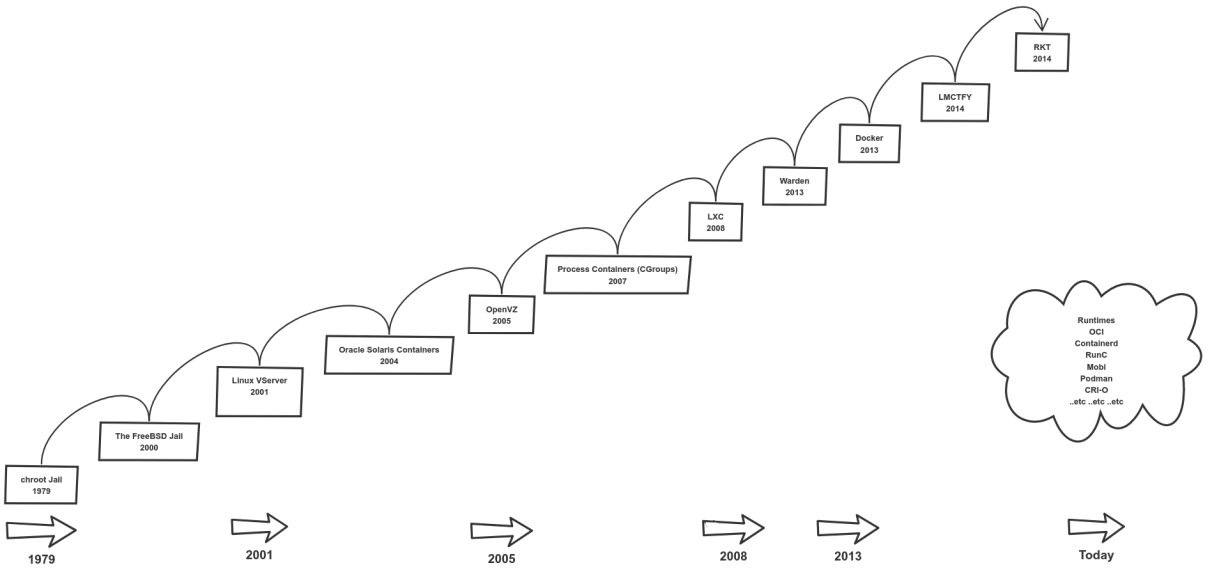
Containerization technologies have remained highly popular in the cloud economy and IT ecosystem.

The container ecosystem can be confusing at times. This guide aims to clarify some of the confusing concepts surrounding Docker and containers. We will also dive into the evolution of the containerization ecosystem and its current state.

We Are Made by History

Docker is currently one of the most well-known container platforms. Although it was released in 2013, the use of isolation and containerization started even before that.

Let's rewind to 1979 when we first began using the Chroot Jail and explore other well-known containerization technologies that emerged afterwards. This exploration will enhance our understanding of new concepts, not only in terms of their historical context but also in terms of technology.



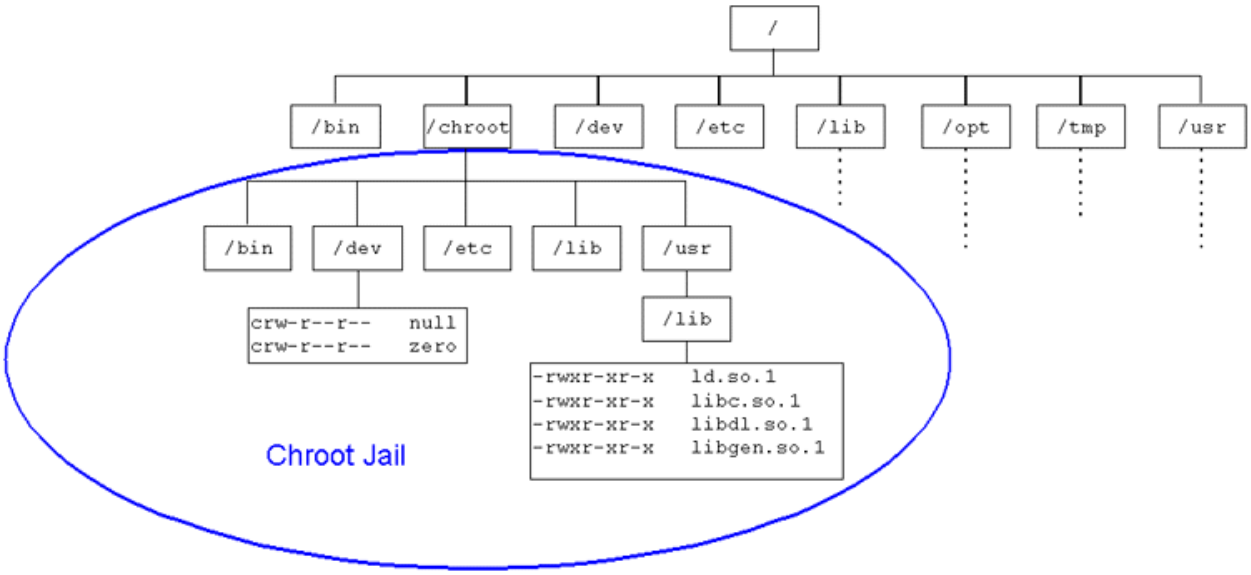
Evolution of containers and isolation systems

It all started with the Chroot Jail. The Chroot system calls were introduced during the development of Version 7 Unix[2] in 1979. The Chroot jail, short for "Change Root", is considered one of the first containerization technologies. It allows you to isolate a process and its children from the rest of the operating system.

The chroot mechanism changes the root directory of a process, creating an apparently isolated environment. However, it was not designed for security. Issues with chroot include the potential for root processes to escape, inadequate isolation, potential misuse of device files, maintenance complexities, and shared kernel vulnerabilities. While useful for specific tasks, it is not recommended as the primary method for securing untrusted processes. Modern containerization or virtualization solutions offer more robust security.

The FreeBSD Jail[3] was introduced in the FreeBSD operating system[4] in the year 2000, with the intention of enhancing the security of the simple Chroot file isolation. Unlike Chroot, the FreeBSD implementation also isolates processes and their activities to a specific view of the filesystem.

[2] https://en.wikipedia.org/wiki/Version_7_Unix
[3] https://wiki.freebsd.org/Jails
[4] https://www.freebsd.org/



Chroot jail

Linux VServer was introduced **in 2001** when operating system-level virtualization capabilities were added to the Linux kernel. It offers a more advanced virtualization solution by utilizing a combination of a chroot-like mechanism, "security contexts", and operating system-level virtualization (containerization). With Linux VServer, you can run multiple Linux distributions on a single distribution (VPS).

Initially, the project was developed by Jacques Gélinas. The implementation paper[5] abstract states:

> A soft partitioning concept based on Security Contexts which permits the creation of many independent Virtual Private Servers (VPS) that run simultaneously on a single physical server at full speed, efficiently sharing hardware resources. A VPS provides an almost identical operating environment as a conventional Linux Server. All services, such as ssh, mail, Web and databases, can be started on such a VPS, without (or in special cases with only minimal) modification, just like on any real server. Each virtual server has its own user account database and root password and is isolated from other virtual servers, except for the fact that they share the same hardware resources.

[5] http://linux-vserver.org/Paper

Linux VServer logo

> **i** This project is not related to the Linux Virtual Server[6] project, which implements network load balancing and failover of servers.

In **February 2004**, Sun (acquired later by Oracle) released **(Oracle) Solaris Containers**, an implementation of Linux-Vserver for X86 and SPARC processors.

> **i** SPARC is a RISC (reduced instruction set computing) architecture developed by Sun Microsystems.

[6] http://www.linuxvirtualserver.org/

Oracle Sloaris 11.3

Similar to Solaris Containers, the first version of **OpenVZ** was introduced **in 2005**.

OpenVZ, like Linux-VServer, utilizes OS-level virtualization. It gained popularity among hosting companies for isolating and selling VPSs. However, OS-level virtualization has limitations, as containers share the same architecture and kernel version. This becomes a disadvantage when guests require different kernel versions than the host.

Both Linux-VServer and OpenVZ require kernel patching to implement control mechanisms for creating isolated containers. However, OpenVZ patches were not integrated into the Kernel.

OpenVZ Logo

In 2007, Google released **cgroups**, a mechanism that limits and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes. Unlike OpenVZ Kernel[7], cgroups was integrated into the Linux kernel in the same year.

In 2008, the first version of **LXC (Linux Containers)** was released. LXC is similar to OpenVZ, Solaris Containers, and Linux-VServer, but it uses the existing cgroups mechanism implemented in the Linux Kernel.

A few years later, **in 2013**, CloudFoundry introduced **Warden**, an API for managing isolated, ephemeral, and resource-controlled environments. Initially, Warden[8] utilized LXC.

[7] https://wiki.openvz.org/Download/kernel
[8] https://github.com/cloudfoundry-attic/warden

LXC Logo

At the time, even with the introduction of LXC, container usage was not widespread. The main reason for this was the lack of a standard container format and runtime, as well as the absence of a developer-friendly tool that could be used by both developers and system administrators. This is where Docker came into the picture.

In 2013, the first version of **Docker** was introduced. Like OpenVZ and Solaris Containers, Docker performs OS-level virtualization.

In 2014, Google introduced **LMCTFY (Let me contain that for you)**, the open-source version of Google's container stack, which provides Linux application containers. Google engineers collaborated with Docker on libcontainer[9] and ported the core concepts and abstractions from LMCTFY to libcontainer.

As we are going to see later, libcontainer project evolved to become **runC**. The Open Container Initiative (OCI) was founded in **June 2015** to create open industry standards around container formats and runtimes by building on the contributions of Docker's container format and runtime. Among the projects donated to the OCI was runC, which was developed by Docker.

LMCTFY[10], on the other hand, runs applications in isolated environments on the same Kernel without patching it, using cgroups, Namespaces, and other Linux Kernel features. The project was abandoned in 2015.

[9] https://github.com/docker-archive/libcontainer
[10] https://github.com/google/lmctfy

Front entrance to Google's Headquarters in Dublin Ireland. Image courtesy of Outreach Pete, licensed under CC BY 2.0. via https://www.flickr.com/photos/182043990@N04/48077952638

It's worth saying that Google is a leader in the container industry. Everything at Google runs on containers. There are more than 2 billion containers[11] running on Google infrastructure every week.

In **December 2014**, CoreOS released and started to support **rkt (initially released as Rocket)** as an alternative to Docker but with a different approach. rkt is a container runtime for Linux and is designed to be secure, composable, and standards-based. However, rkt was abandoned[12].

[11] https://speakerdeck.com/jbeda/containers-at-scale
[12] https://github.com/rkt/rkt/issues/4024

CoreOs Logo

In **December 2015**, Docker Inc. introduced **containerd**, a daemon designed to control runC. This move was made as part of their effort to break Docker into smaller, reusable components. containerd[13], available on Linux and Windows, is an industry-standard container runtime that prioritizes simplicity, robustness, and portability. It serves as a daemon that manages the entire container lifecycle of its host system, including tasks such as image transfer and storage, container execution and supervision, and low-level storage and network attachments.

While containerd addressed many of the container runtime requirements with its robust and portable features, the expanding ecosystem saw the need for additional specialized runtimes tailored for specific orchestrators such as Kubernetes. That why in **2017**, **CRIO-O** was introduced as an alternative to Docker runtime for Kubernetes. As stated in the CRI-O repository[14]:

> CRI-O follows the Kubernetes release cycles with respect to its minor versions (1.x.y). Patch releases (1.x.z) for Kubernetes are not in sync with those from CRI-O, because they are scheduled for each month, whereas CRI-O provides them only if necessary. If a Kubernetes release goes End of Life[15], then the corresponding CRI-O version can be considered in the same way.

As the container landscape continued to evolve rapidly, catering to diverse use-cases and enhancing user flexibility became paramount. Red Hat developed a tool for managing OCI containers and pods called **Podman** (for Pod Manager). The tool was released in **2018**. Podman[16] is a daemonless container engine for developing, managing, and running OCI Containers on Linux System. Containers can either be run as root or in rootless mode. Simply put: `alias docker=podman`. Podman was mainly created to provide a Docker replacement for Linux users who want to avoid the daemon dependency of Docker and access more flexible container runtimes.

[13] https://github.com/containerd/containerd
[14] https://github.com/cri-o/cri-o
[15] https://kubernetes.io/releases/patch-releases/
[16] https://github.com/containers/podman

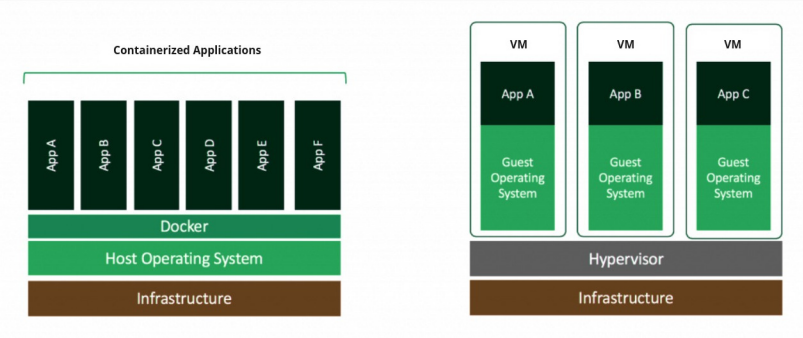
Jails, Virtual Private Servers, Zones, Containers, and VMs: What's the Difference Anyway?

Technologies like Jails, Zones, VMs, and Containers aim for system and resource isolation. Specifically:

While Jails provide advanced chroot environments in FreeBSD, Zones offer isolated OS environments in Solaris, VMs are emulated systems offering complete OS isolation and containers encapsulate application code in isolated packages on the same OS kernel.

Among all of these methods to provide resource isolation and control, containers and VMs are the most popular with VMs offering hardware-level virtualization and containers focusing on application-level isolation.

I'm sure you saw this image before many times, but it's worth mentioning again:



VM vs Containers

> **i** The Kernel is the core of the operating system, and it is responsible for managing the system resources. The OS is the Kernel and the user space programs that run on top of it. While containers such as Docker containers share the same kernel, VMs have their own kernel.

Here is a table that summarizes the differences between Jails, Zones, VMs, and Containers:

Technology	Description
Jails	Advanced chroot environments in FreeBSD
Zones	Isolated OS environments in Solaris
VMs	Emulated systems offering complete OS isolation
Containers	Encapsulate application code in isolated packages, sharing the same OS kernel

Diving deeper into the world of virtual machines, it's worth noting that VMs can be classified into two primary categories:

- System Virtual Machines

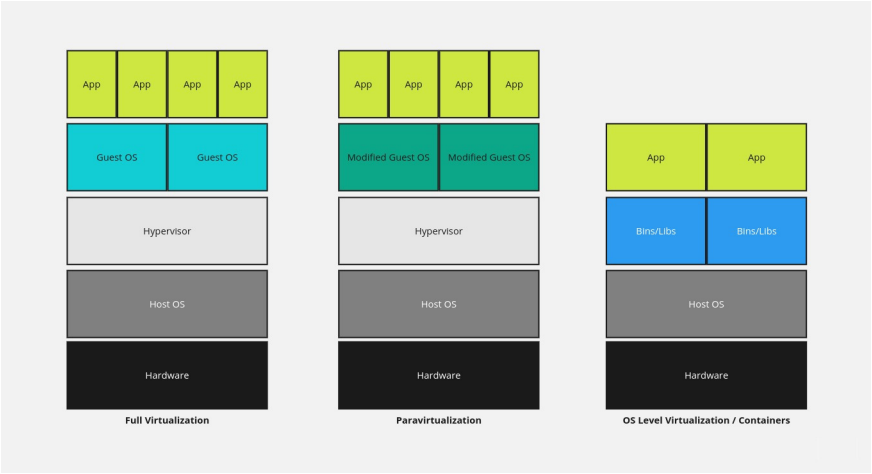
- Process Virtual Machines

When we refer to VMs in common parlance, we're typically alluding to "System Virtual Machines". These simulate the host hardware, allowing for the emulation of an entire operating system. On the other hand, "Process Virtual Machines", also known as "Application Virtual Machines", emulate a specific programming environment well-cut for executing singular processes. A classic example of this is the Java Virtual Machine (JVM).

Furthermore, there's a another specific type of virtualization called OS-level virtualization. This is also known as containerization. Technologies such as Linux-VServer and OpenVZ can run multiple (instances) operating systems while sharing the same architecture and kernel version and are prime examples of this.

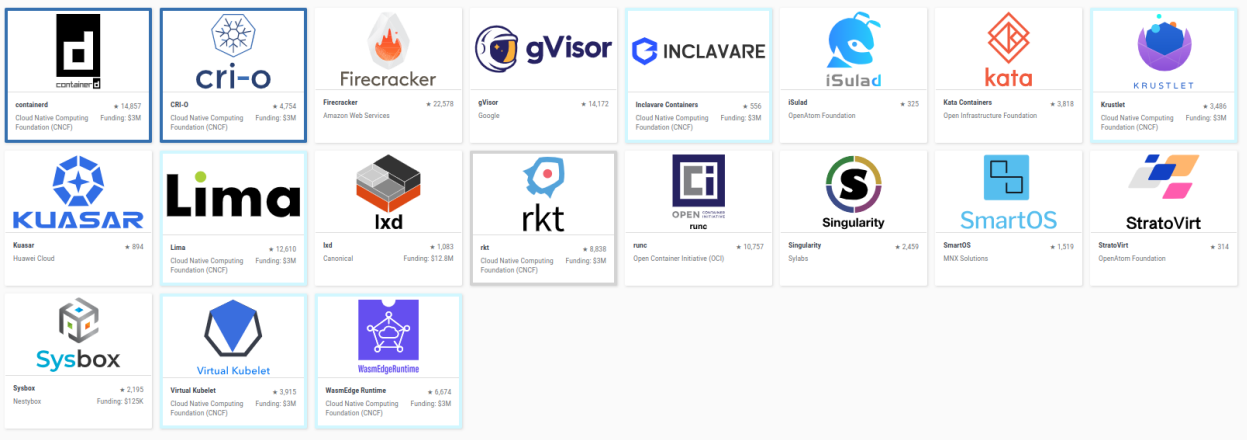
These instances coexist, leveraging the same underlying architecture and kernel version.

Leveraging a shared architecture and kernel presents certain limitations, especially when there's a need for guest instances to operate on different kernel versions than the host. However, it's worth considering: how often would you encounter this specific use case in your operations?



Different Types of Virtualization

This brings us to the modern landscape of OS-level virtualization, which has seen a surge in popularity and utilization. Docker, Podman, and a set of other containerization technologies have become the de facto standard for containerization especially with the work done by the Open Container Initiative (OCI), the Cloud Native Computing Foundation (CNCF), and the Linux Foundation.



CNCF container runtime landscape

Under the CNCF umbrella, notable container runtime projects include runC, containerd, CRI-O, gVisor, Kata and Firecracker with a collective funding of $15.9M at the time of writing.

OS Containers vs. App Containers

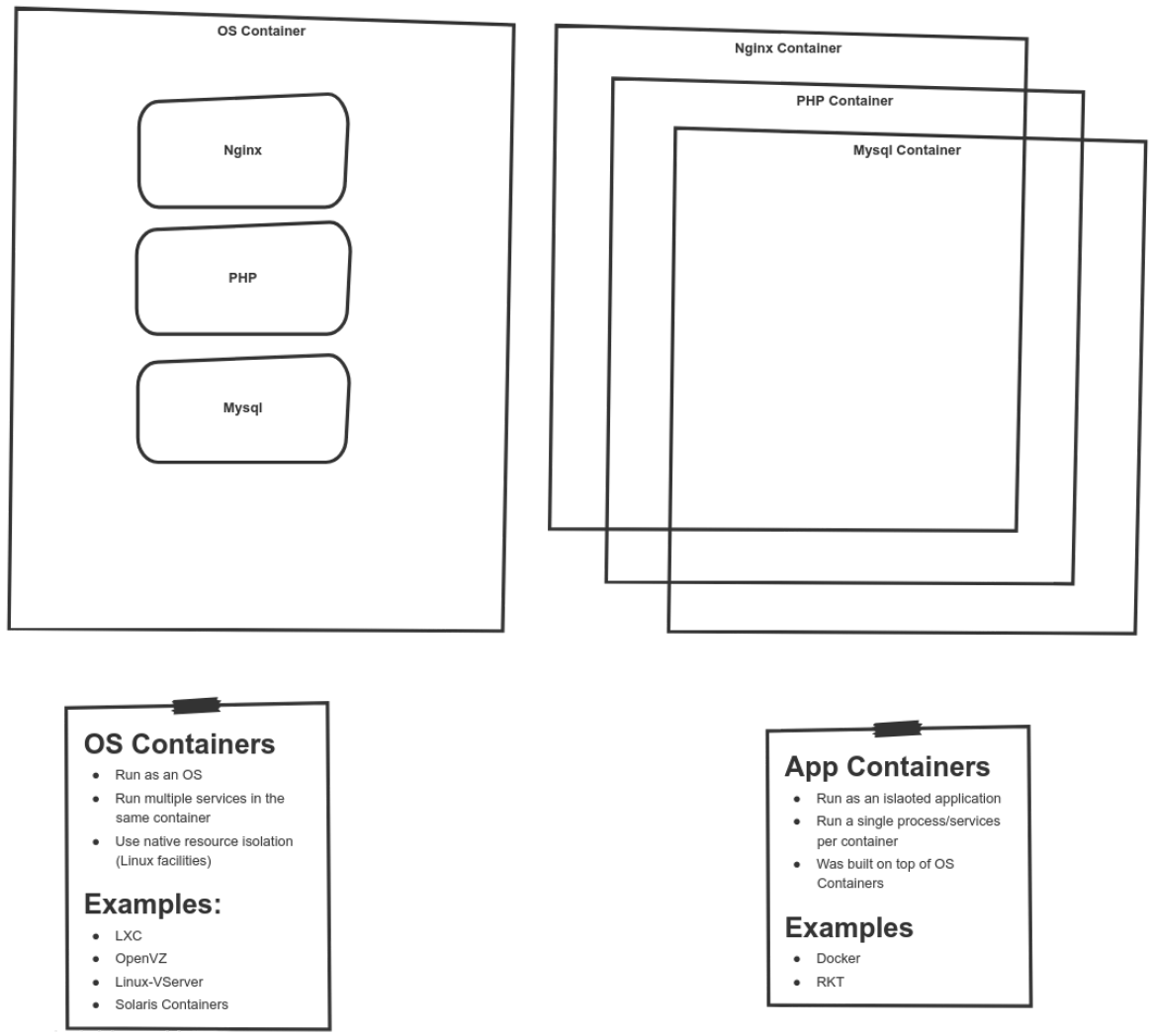
OS-level virtualization, as seen, enables us to create containers that share the same architecture and kernel version. Technologies such as OpenVZ and Docker use this type of isolation. In this context, we have two subcategories of containers:

- **OS Containers** package the entire application stack along with the operating system. Technologies such as LXC, Linux VServer, and OpenVZ are examples of suitable technologies for creating OS containers. Usually, OS containers run multiple applications and processes per container. A LEMP stack (Linux, NGINX, MySQL, and PHP) is an example of an OS container.
- **Application Containers** typically run a single process per container. The application is packaged with its dependencies and libraries. Docker is an example of a technology that can be used to create application containers.

For the creation of a LEMP stack using App Containers while following the best practices, we need to create three containers:

- PHP server (or PHP FPM).
- Web server (NGINX).
- MySQL.

Each container will run a single process.



OS Containers vs App Containers

Docker: Container or Platform?

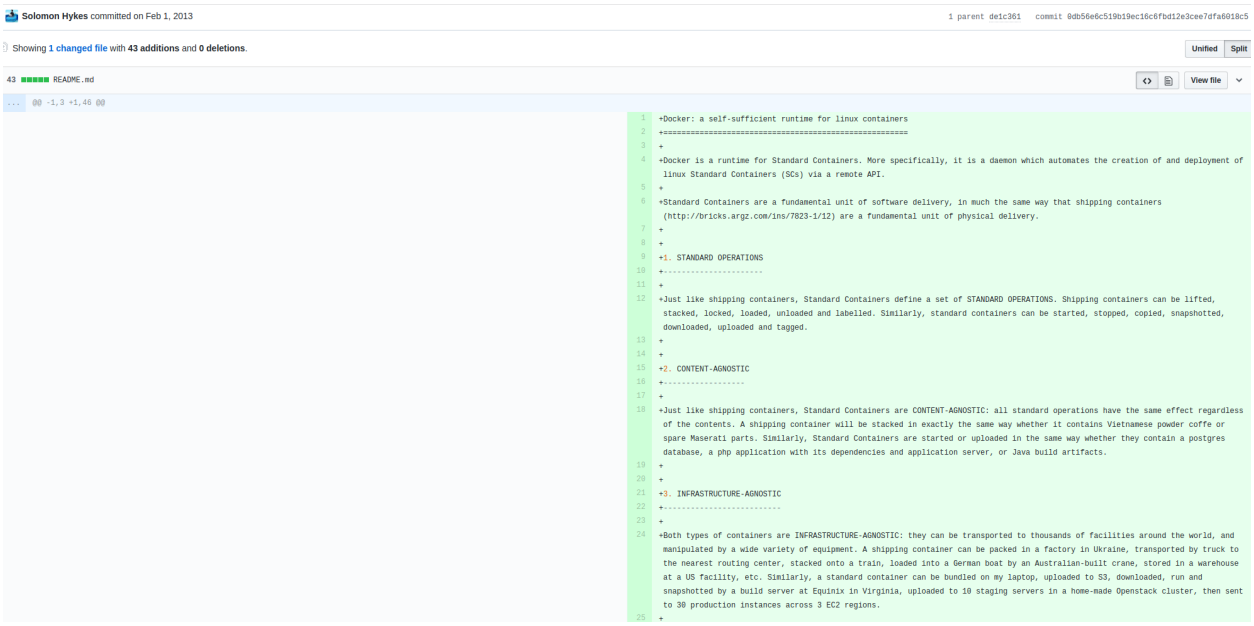
There are many misconceptions about Docker. Some people believe that Docker is solely a container runtime, while others consider it to be a container platform.

However, Docker can be seen as both a container and a platform, depending on the context.

When Docker was initially launched, it used LXC as its container runtime. The main idea behind Docker was to create an API for managing the container runtime, isolating single processes running applications, and supervising the container's lifecycle and resource usage.

In early 2013, the Docker project aimed to establish a "standard container", as outlined in this manifesto[17].

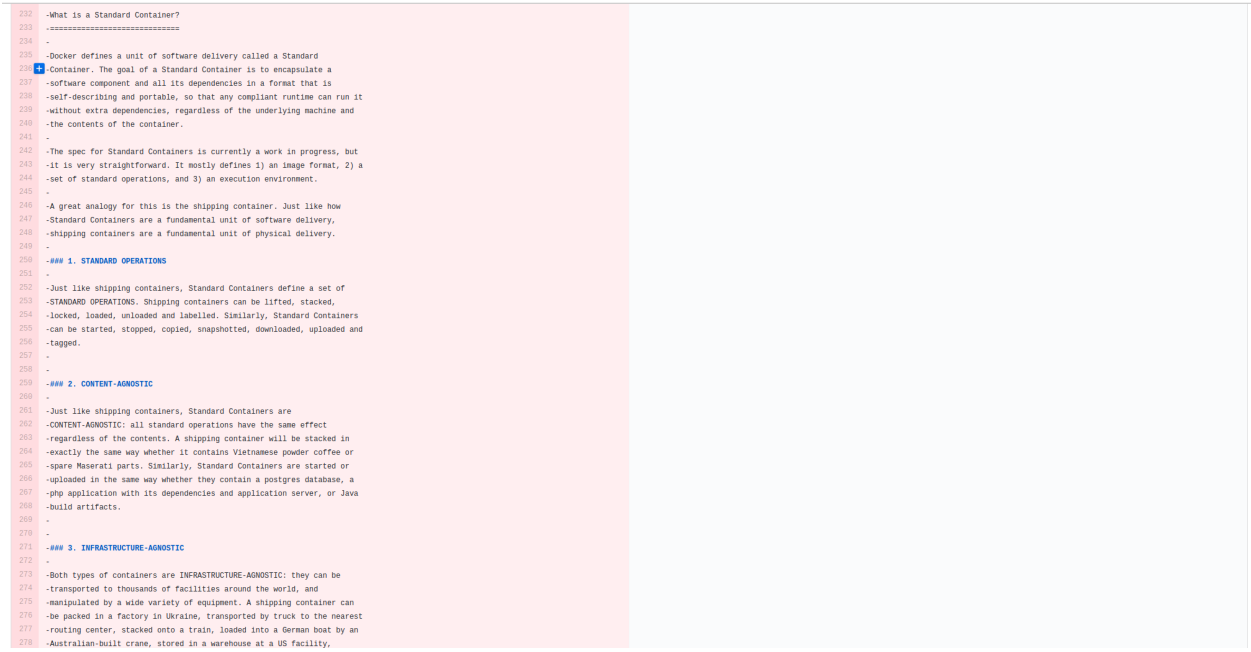
At its core, Docker introduced a technology to create and run containers as standalone, executable packages that includes everything needed to run a piece of software, including the code, runtime, system tools, and libraries.



Docker code (github.com)

The standard container manifesto was removed[18] a few months later.

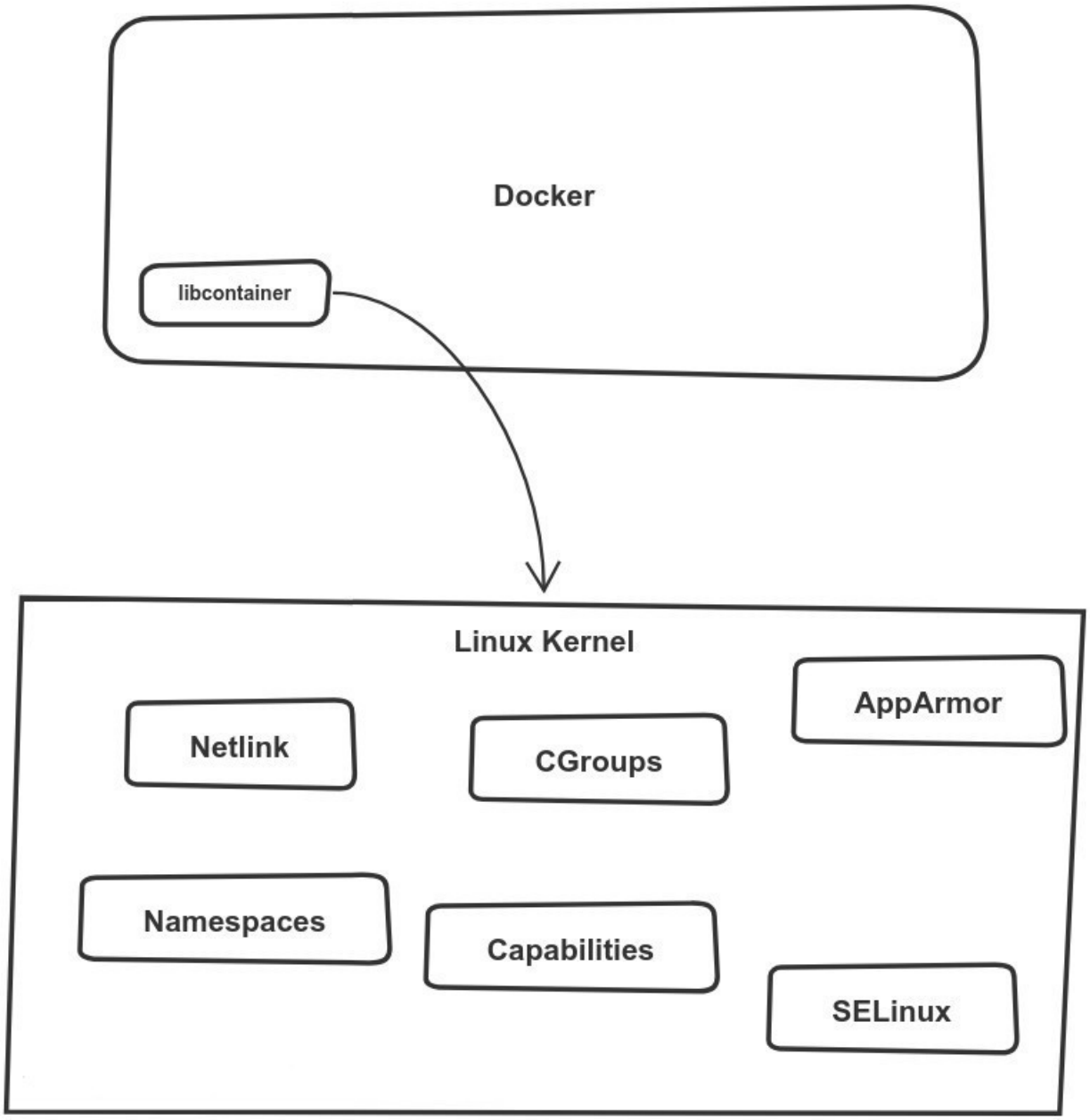
[17] https://github.com/moby/moby/commit/0db56e6c519b19ec16c6fbd12e3cee7dfa6018c5
[18] https://github.com/docker/docker/commit/eed00a4afd1e8e8e35f8ca640c94d9c9e9babaf7



```
-What is a Standard Container?
-==============================
-
-Docker defines a unit of software delivery called a Standard
+Container. The goal of a Standard Container is to encapsulate a
-software component and all its dependencies in a format that is
-self-describing and portable, so that any compliant runtime can run it
-without extra dependencies, regardless of the underlying machine and
-the contents of the container.
-
-The spec for Standard Containers is currently a work in progress, but
-it is very straightforward. It mostly defines 1) an image format, 2) a
-set of standard operations, and 3) an execution environment.
-
-A great analogy for this is the shipping container. Just like how
-Standard Containers are a fundamental unit of software delivery,
-shipping containers are a fundamental unit of physical delivery.
-
-### 1. STANDARD OPERATIONS
-
-Just like shipping containers, Standard Containers define a set of
-STANDARD OPERATIONS. Shipping containers can be lifted, stacked,
-locked, loaded, unloaded and labelled. Similarly, Standard Containers
-can be started, stopped, copied, snapshotted, downloaded, uploaded and
-tagged.
-
-
-### 2. CONTENT-AGNOSTIC
-
-Just like shipping containers, Standard Containers are
-CONTENT-AGNOSTIC: all standard operations have the same effect
-regardless of the contents. A shipping container will be stacked in
-exactly the same way whether it contains Vietnamese powder coffee or
-spare Maserati parts. Similarly, Standard Containers are started or
-uploaded in the same way whether they contain a postgres database, a
-php application with its dependencies and application server, or Java
-build artifacts.
-
-
-### 3. INFRASTRUCTURE-AGNOSTIC
-
-Both types of containers are INFRASTRUCTURE-AGNOSTIC: they can be
-transported to thousands of facilities around the world, and
-manipulated by a wide variety of equipment. A shipping container can
-be packed in a factory in Ukraine, transported by truck to the nearest
-routing center, stacked onto a train, loaded into a German boat by an
-Australian-built crane, stored in a warehouse at a US facility,
```

Docker code (github.com)

Docker started to use its own container runtime called libcontainer to interface with Linux kernel facilities like Control Groups and Namespaces.



Docker, Libcontainer and Linux Kernel Facilities

Docker started building a monolithic application with multiple features, from launching Cloud servers to building and running images/containers. The goal was providing a comprehensive ecosystem and platform that includes tools for building, storing, and managing container images (like Docker Hub), orchestrating containers (like Docker Swarm), and a user-friendly CLI and GUI (Docker Desktop) for interacting with containers.

Using Namespaces and cgroups to Isolate Containers in Practice

History aside, let's dive into some introductory yet practical aspects of containerization. We will begin by exploring the isolation mechanisms that Docker uses to create containers.

The objective here is to establish an isolated environment using namespaces and cgroups.

Make sure that cgoups v2 is enabled:

```
ls /sys/fs/cgroup/cgroup.controllers
```

If you don't see any output, then you need to enable cgroups v2:

```
sudo mount -t cgroup2 none /sys/fs/cgroup
```

Next, create a new execution context using the following command:

```
sudo unshare --fork --pid --mount-proc bash
```

> **i** The execution context is the environment in which a process runs. It includes the process itself and resources such as memory, file handles, and so on.

The unshare[19] command disassociates parts of the process execution context

```
unshare() allows a process (or thread) to disassociate parts of its execution contex\
t that are currently being shared with other processes (or threads).  Part of the ex\
ecution context, such as the mount namespace, is shared implicitly when a new proces\
s is created using fork(2) or vfork(2), while other parts, such as virtual memory, m\
ay be shared by explicit request when creating a process or thread using clone(2).
```

You can view the list of processes running in the current namespace using the following command:

```
ps aux
```

You will notice that the process list is different from the one on the host machine.

Now, using `cgcreate` we can create a Control Groups and define two controllers, one on memory and the other on CPU:

[19] http://man7.org/linux/man-pages/man2/unshare.2.html

```
sudo cgcreate -a $USER:$USER -t $USER:$USER -g memory,cpu:mygroup
```

The `-a` option is used to define the user and group that will be used to own the created cgroup and the `-t` option is used to define the user and group that will be used to own the tasks file in the created cgroup.

You can view the created cgroup using the following command:

```
ls /sys/fs/cgroup/mygroup/
```

The next step is defining a limit for the memory and a limit for the CPU:

```
# We want to set a very low memory limit, for instance, 50M.
echo "50M" | sudo tee /sys/fs/cgroup/mygroup/memory.max
# We want to set a very low CPU weight, for instance, 50.
echo "50" | sudo tee /sys/fs/cgroup/mygroup/cpu.weight
```

> **i** CPU limits in cgroup v2 are set differently compared to cgroup v1. Instead of setting explicit usage percentages, you set a "weight". The default weight is 100, and the range is from 1 to 10000.

Note that setting very low limits, especially for memory, can cause processes within a cgroup to be terminated if they exceed the limit. In our case, we are experimenting with the limits and we are not creating a production environment.

Let's disable the swap memory:

```
sudo swapoff -a
```

Now let's stress the isolated namespace we created with memory limits. We are going to use the stress utility to stress the memory with 60M of memory for 10 seconds:

```
sudo apt install stress -y && stress --vm 1 --vm-bytes 60M --timeout 10s
```

We can notice that the execution is failed; you should see something like this:

```
stress: info: [232] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
stress: FAIL: [232] (416) <-- worker 233 got signal 9
stress: WARN: [232] (418) now reaping child worker processes
stress: FAIL: [232] (452) failed run completed in 0s
```

Therefore, we know that the memory limit is working fine.

If you launch the same test on the host without the namespace, you will notice that the execution is successful.

```
stress: info: [37596] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
stress: info: [37596] successful run completed in 10s
```

Following these steps will help in understanding how Linux facilities like **CGroups** and other resource control features can create and manage isolated environments in Linux systems.

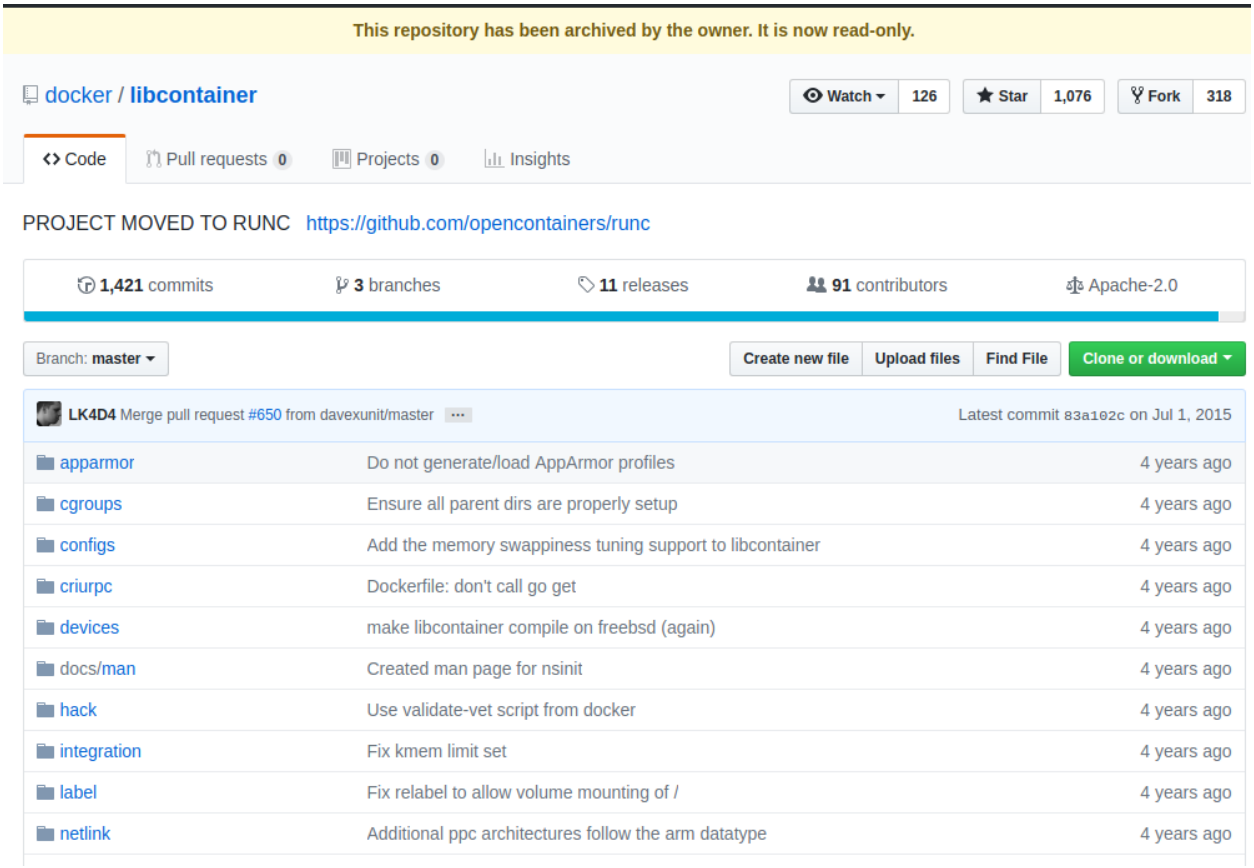
libcontainer interfaces with these facilities to manage and run Docker containers.

The Open Container Initiative: What is a Standard Container?

runC is basically a little command-line tool to leverage libcontainer directly, without going through the Docker Engine.

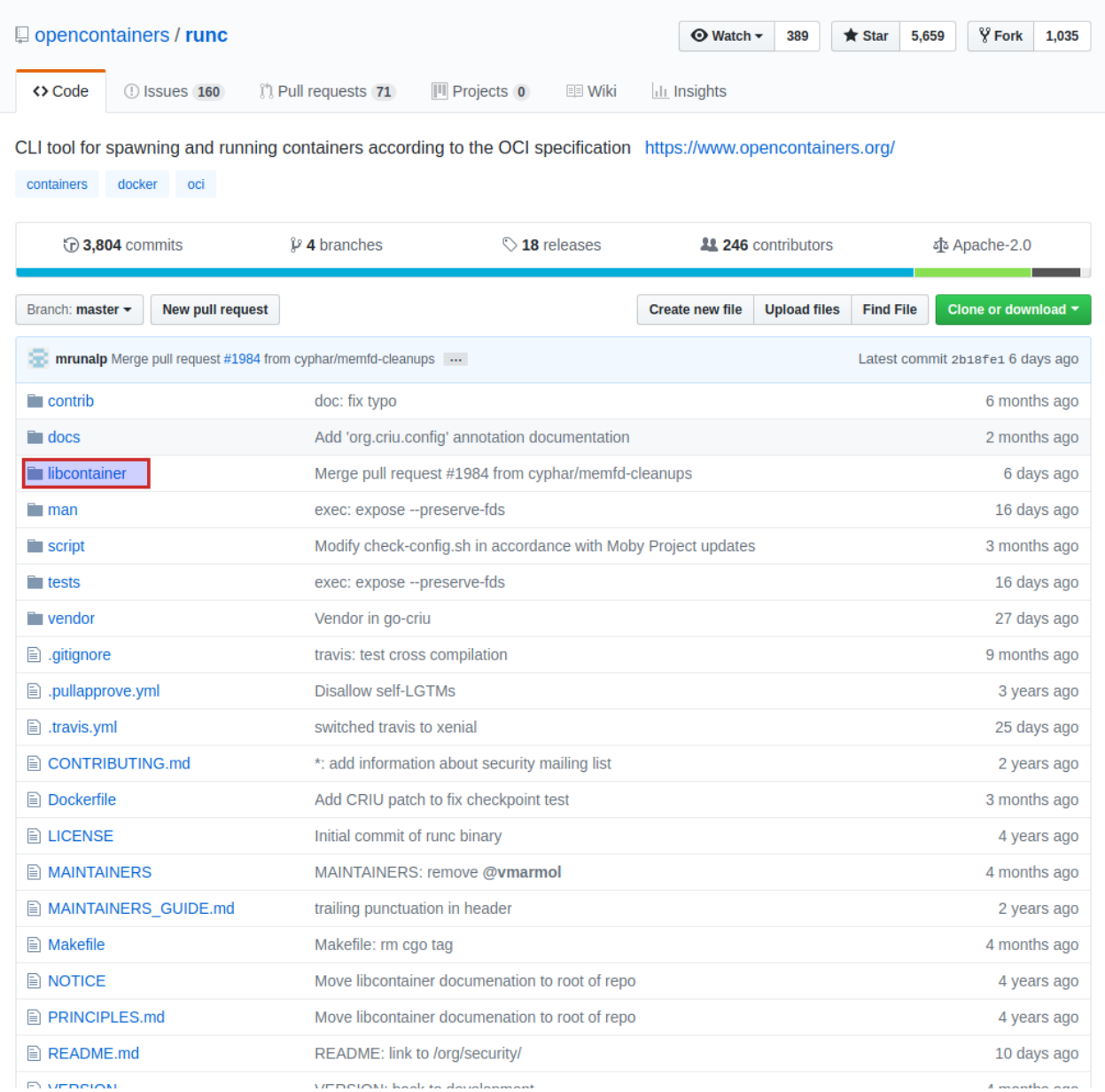
runC logo

The goal behind runC is to make standard containers available everywhere. This project was donated to the Open Container Initiative (OCI) and the libcontainer repository has been archived. The fact that Docker donated runC to the Open Container Initiative (OCI) solidified Docker's commitment to open standards and its willingness to collaborate with other industry players.



libcontainer repository

In reality, libcontainer was not abandoned, but it was moved to runC repository. We can understand that the goal finally was allowing libcontainer (or a modified version of it) to be used without Docker (through runC).



runC repository

But what is the Open Container Initiative (OCI)?

The OCI is a lightweight, open governance structure launched on **2015** by Docker, CoreOS, and other leaders in the container industry.

The Open Container Initiative Logo

The Open Container Initiative aims to establish common standards for software containers in order to avoid potential fragmentation and divisions inside the container ecosystem.

The main difference between libcontainer and runC is that libcontainer is a Go library, while runC is a CLI tool. The latter wraps the former and provides a CLI interface to it. The CLI was built while taking into consideration the OCI specifications.

These specifications are the result of the collaboration between Docker and other leaders in the container industry. The result of this collaboration aims to establish common standards for software containers. In other words, the OCI specifications answer the question: "What is a **standard** container?"

You can find the specifications in four main repositories:

- runtime-spec[20]: The Open Container Initiative develops specifications for standards on Operating System process and application containers.
- image-spec[21]: The OCI Image Format project creates and maintains the software shipping container image format spec (OCI Image Format).
- distribution-spec[22]: The OCI Distribution Spec project defines an API protocol to facilitate and standardize the distribution of content.

A Deep Dive into Container Prototyping with runC

Let's now dive into the runC project and see how it works.

Start by installing runC runtime:

```
1 sudo apt install runc -y
```

Let's create a directory (/mycontainer) where we are going to export the content of the image Busybox[23].

[20] https://github.com/opencontainers/runtime-spec
[21] https://github.com/opencontainers/image-spec
[22] https://github.com/opencontainers/distribution-spec
[23] https://hub.docker.com/_/busybox

i At a disk size ranging from 1 to 5 Mb based on its version, BusyBox[24] serves as an excellent component for creating compact distributions. BusyBox amalgamates miniaturized versions of numerous standard UNIX tools into one compact executable. It offers alternatives to many utilities typically found in GNU fileutils, shellutils, and the like. While the utilities within BusyBox might offer fewer features than their complete GNU versions, the provided features maintain their expected operations and closely mirror their GNU analogs. For smaller or embedded systems, BusyBox ensures a relatively comprehensive environment.

```
# Change to root user
sudo su
# Create the directory
mkdir -p /mycontainer && cd /mycontainer
# Create a rootfs directory
mkdir rootfs
# Export the busybox image
docker export $(docker create busybox) | tar -C rootfs -xvf -
ls -l rootfs
```

Create a spec file (config.json) that will be used by runC to run the container:

```
cd /mycontainer
runc spec
```

i The spec file is a JSON file that contains the configuration of the container. It contains the rootfs path, the hostname, the mounts, the namespaces, the process, and more.

Using runC command, we can run the busybox container that uses the extracted image and a spec file (config.json).

```
runc run busybox
```

`runc spec` command initially creates this JSON file:

[24] https://www.busybox.net/

```
{
        "ociVersion": "1.0.2-dev",
        "process": {
                "terminal": true,
                "user": {
                        "uid": 0,
                        "gid": 0
                },
                "args": [
                        "sh"
                ],
                "env": [
                        "PATH=..",
                        "TERM=xterm"
                ],
                "cwd": "/",
                "capabilities": {
                        "bounding": [
                                "CAP_AUDIT_WRITE",
                                "CAP_KILL",
                                "CAP_NET_BIND_SERVICE"
                        ],
                        "effective": [
                                "CAP_AUDIT_WRITE",
                                "CAP_KILL",
                                "CAP_NET_BIND_SERVICE"
                        ],
                        "permitted": [
                                "CAP_AUDIT_WRITE",
                                "CAP_KILL",
                                "CAP_NET_BIND_SERVICE"
                        ],
                        "ambient": [
                                "CAP_AUDIT_WRITE",
                                "CAP_KILL",
                                "CAP_NET_BIND_SERVICE"
                        ]
                },
                "rlimits": [
                        {
                                "type": "RLIMIT_NOFILE",
                                "hard": 1024,
                                "soft": 1024
```

```
                        }
                ],
                "noNewPrivileges": true
        },
        "root": {
                "path": "rootfs",
                "readonly": true
        },
        "hostname": "runc",
        "mounts": [
                {
                        "destination": "/proc",
                        "type": "proc",
                        "source": "proc"
                },
                {
                        "destination": "/dev",
                        "type": "tmpfs",
                        "source": "tmpfs",
                        "options": [
                                "nosuid",
                                "strictatime",
                                "mode=755",
                                "size=65536k"
                        ]
                },
                {
                        "destination": "/dev/pts",
                        "type": "devpts",
                        "source": "devpts",
                        "options": [
                                "nosuid",
                                "noexec",
                                "newinstance",
                                "ptmxmode=0666",
                                "mode=0620",
                                "gid=5"
                        ]
                },
                {
                        "destination": "/dev/shm",
                        "type": "tmpfs",
                        "source": "shm",
```

```
87                          "options": [
88                                  "nosuid",
89                                  "noexec",
90                                  "nodev",
91                                  "mode=1777",
92                                  "size=65536k"
93                          ]
94                  },
95                  {
96                          "destination": "/dev/mqueue",
97                          "type": "mqueue",
98                          "source": "mqueue",
99                          "options": [
100                                 "nosuid",
101                                 "noexec",
102                                 "nodev"
103                         ]
104                 },
105                 {
106                         "destination": "/sys",
107                         "type": "sysfs",
108                         "source": "sysfs",
109                         "options": [
110                                 "nosuid",
111                                 "noexec",
112                                 "nodev",
113                                 "ro"
114                         ]
115                 },
116                 {
117                         "destination": "/sys/fs/cgroup",
118                         "type": "cgroup",
119                         "source": "cgroup",
120                         "options": [
121                                 "nosuid",
122                                 "noexec",
123                                 "nodev",
124                                 "relatime",
125                                 "ro"
126                         ]
127                 }
128         ],
129         "linux": {
```

```
130                 "resources": {
131                         "devices": [
132                                 {
133                                         "allow": false,
134                                         "access": "rwm"
135                                 }
136                         ]
137                 },
138                 "namespaces": [
139                         {
140                                 "type": "pid"
141                         },
142                         {
143                                 "type": "network"
144                         },
145                         {
146                                 "type": "ipc"
147                         },
148                         {
149                                 "type": "uts"
150                         },
151                         {
152                                 "type": "mount"
153                         },
154                         {
155                                 "type": "cgroup"
156                         }
157                 ],
158                 "maskedPaths": [
159                         "/proc/acpi",
160                         "/proc/asound",
161                         "/proc/kcore",
162                         "/proc/keys",
163                         "/proc/latency_stats",
164                         "/proc/timer_list",
165                         "/proc/timer_stats",
166                         "/proc/sched_debug",
167                         "/sys/firmware",
168                         "/proc/scsi"
169                 ],
170                 "readonlyPaths": [
171                         "/proc/bus",
172                         "/proc/fs",
```

```
                        "/proc/irq",
                        "/proc/sys",
                        "/proc/sysrq-trigger"
                ]
        }
}
```

By using the generated specification JSON file, you have the ability to customize the runtime of the container. For instance, you can modify the argument for the application to be executed.

Let's compare the original config.json file with the updated version:

Differences between the original and the updated config.json

- Changed `"terminal": true` to `"terminal": false` to run the container in the background.
- Changed `"args": ["sh"]` to `"args": ["sh", "-c", "i=1; while [ $i -le 10 ]; do echo $i; i=$((i+1)); sleep 1; done"]` to run a command that prints the numbers from 1 to 10 and sleeps for 1 second between each number.

You can perform this by running the following command:

```
cd mycontainer/
# install jq: https://jqlang.github.io/jq/download/
sudo apt install jq -y
echo $(jq '.process.args=["sh", "-c", "i=1; while [ $i -le 10 ]; do echo $i; i=$((i+\
1)); sleep 1; done"]' config.json) > config.json
```

You can view the new JSON using:

```
cd /mycontainer
jq . config.json
```

Now, let's run the container again and observe how it sleeps for 10 seconds before exiting.

```
cd /mycontainer
runc run busybox
```

Another option for generating a customized spec config is to use oci-runtime-tool[25]. The `oci-runtime-tool generate` sub-command offers numerous options for customization.

Industry Standard Container Runtimes

Docker, known for its extensive set of features including image construction and management, as well as a powerful API, has consistently been a leader in container technology. Beyond these well-known capabilities, Docker also provides foundational tools that are suitable for reuse. One notable feature is the ability to fetch images and store layers using a user-selected union file system, as demonstrated in containerd.

> **i** Docker uses a union file system[26] to stack and manage changes in containers, with each change saved as a separate layer; "containerd" demonstrates this layering technique.

containerd serves as a core daemon responsible for supervising containers and images, excluding their construction. This unique role classifies containerd as a "container runtime," while runC is referred to as a "base-level container runtime." The release of Docker Engine 1.11[27] was particularly significant as it was built upon the foundations of runC and containerd. This release solidified Docker's position as a pioneer in deploying a runtime based on the Open Container Initiative (OCI) technology. It highlighted Docker's progress, especially since **mid-2015** when its standardized container format and runtime were entrusted to the Linux Foundation.

[25] https://github.com/opencontainers/runtime-tools
[26] https://en.wikipedia.org/wiki/UnionFS
[27] https://www.docker.com/blog/docker-engine-1-11-runc/

The journey towards this integration began effectively in **December 2015** when Docker introduced containerd. This daemon was specifically designed to manage runC and represented a significant step towards Docker's vision of breaking down its extensive architecture into more modular and reusable units. The foundation of Docker Engine 1.11 on containerd signifies that every interaction with Docker is now an interaction with the OCI framework. This commitment to expanding the scope of the OCI, in collaboration with over 40 members, emphasizes the collective effort to achieve uniformity in container technology.

Since containers gained mainstream popularity, there has been a collective effort by various stakeholders to establish standardization in this field. Standardization is essential as it allows for automation and ensures the consistent application of best practices.

Data supports the positive trajectory of Docker's adoption, highlighted by the move towards standardization, seamless integration with platforms like Kubernetes, and the introduction of modern technologies such as containerd. This upward trend is further illustrated by the following data:

Year	Pulls from Docker Hub	Introduced Technologies
2014 -> 2015	1M -> 1B	Introduction of libcontainer
2015 -> 2016	1B -> 6B	Notary, runC, libnetwork
2016 -> 2017	6B -> 12B	Hyperkit, VPNKit, DataKit, SwarmKit, InfraKit, containerd, Linuxkit

While some might perceive these technological nuances as captivating juicy bit of news for casual banter, the implications are profound. So, what does this mean for the end-user?

Immediately, the impact might seem negligible. However, the significance of this development is undeniable. With Docker Engine undergoing one of its most rigorous technical metamorphoses, the primary intent for the 1.11 version was a flawless amalgamation without perturbing the command-line interface or the API. Yet, this transition is a sign of forthcoming user-centric enhancements that hold immense promise.

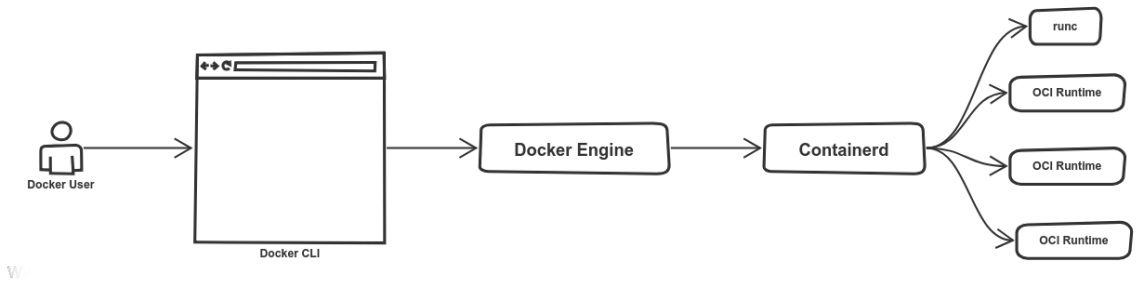
This change made it easier and more standard for everyone to use and improve containers, paving the way for new tools and ideas in the containerization landscape.

While donating the runC project to the OCI, Docker started using **containerd** in **2016**, as a container runtime that interface with the underlying low-level runtime runC.

There are actually different runtimes; each one acts at a different level.

```
1 docker info | grep -i runtime
```

containerd has full support for starting OCI bundles and managing their lifecycle. Containerd (as well as other runtimes like cri-o) uses runC to run containers but also implements other high-level features like image management and high-level APIs.



containerd integration with Docker & OCI runtimes

containerd, shim and runC: How Everything Works Together

When you launch a container using Docker, you are actually utilizing several components. Containerd is responsible for running the container, runC manages the container's lifecycle and resources, and containerd-shim keeps the file descriptors open.

runC is built on libcontainer, which was previously the container library used by Docker engine.

Before version 1.11, Docker engine handled the management of volumes, networks, containers, and images. However, libcontainer was integrated directly into Docker before being split out as runC.

Now, the Docker architecture consists of four components:

- Docker engine
- containerd
- containerd-shim
- runC

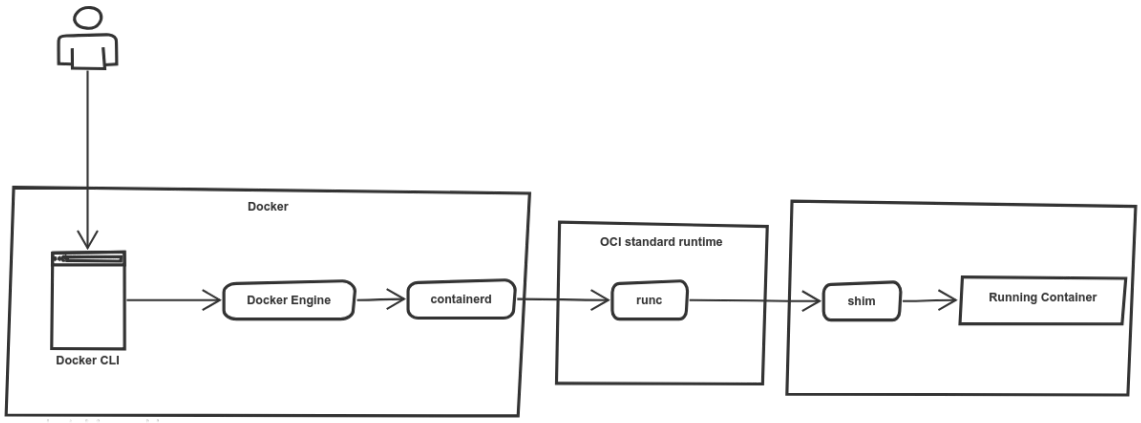
The corresponding binaries are **docker**, **docker-containerd**, **docker-containerd-shim**, and **docker-runc**.

Now let's go through the steps to run a container using the new architecture:

1. The user requests Docker to create a container using Docker CLI, which utilizes the Docker API.
2. The Docker engine, listening to the API, creates the container from an image and passes it to containerd.
3. containerd calls containerd-shim.
4. containerd-shim uses runC to run the container.
5. containerd-shim allows the runtime (runC in this case) to exit after starting the container.

This new architecture provides three advantages:

1. runC can exit once the container is started, eliminating the need to keep the entire runtime processes running. In other words, individual containers can run without the main runtime process persisting in memory.
2. containerd-shim keeps the file descriptors (stdin, stdout, stderr) open even if Docker and/or containerd terminates.
3. This architecture allows for more modularity, enabling other projects and tools to interact with or replace parts of the Docker ecosystem.



"Docker Architecture and OCI Runtimes"

Adding a New Runtime to Docker

Once you explore why Docker incorporated runC and containerd into its architecture, you'll understand that both serve as runtimes, although with different functionalities. So, why use two runtimes? This question commonly arises when discussing the Docker architecture.

If you've been following along, you may have noticed references to both high-level and low-level runtimes. This is the practical distinction between the two. While both can be considered "runtimes," they serve different purposes and offer unique features.

To maintain standardization in the containers ecosystem, low-level containers runtime only allows the execution of containers. The low-level runtime (like runC) should be lightweight, fast, and compatible with other higher levels of container management.

When you create a Docker container, it is actually managed by both runtimes: containerd and runC.

There are multiple container runtimes available, some of which are OCI standardized while others are not. Some are low-level runtimes, while others are high-level runtimes. Some go beyond basic runtimes and include additional tools for managing the container lifecycle, such as:

- Image transfer and storage
- Container execution and supervision
- Low-level storage
- Network attachments

The following table summarizes some of the most popular container runtimes and their levels:

Technology	Level
LXC	Low-level
runC	Low-level
lmctfy	Low-level
CRI-O	High-level
containerd	High-level
rkt	High-level

To add a new runtime, you should start by installing it. For instance, if you want to install the nvidia runtime, you can use the following command:

```
# Import the NVIDIA GPG key
curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey | \
  sudo gpg --dearmor -o /usr/share/keyrings/nvidia-container-toolkit-keyring.gpg

# Update the repository list with signing
curl -s -L https://nvidia.github.io/libnvidia-container/stable/deb/nvidia-container-\
toolkit.list | \
  sed 's#deb https://#deb [signed-by=/usr/share/keyrings/nvidia-container-toolkit-ke\
yring.gpg] https://#g' | \
  sudo tee /etc/apt/sources.list.d/nvidia-container-toolkit.list

# Update packages and install the NVIDIA container toolkit
sudo apt-get update && sudo apt-get install -y nvidia-container-toolkit
```

Then, you can add it to the configuration file `/etc/docker/daemon.json`:

```
# export the runtime name
export RUNTIME="nvidia"
# export the runtime path
export RUNTIME_PATH="/usr/bin/nvidia-container-runtime"
# Update the daemon.json file or create it if it doesn't exist
sudo tee /etc/docker/daemon.json <<EOF
{
  "runtimes": {
        "$RUNTIME": {
```

```
            "path": "$RUNTIME_PATH"
        }
    }
}
EOF
# Restart Docker
sudo systemctl restart docker
```

You can also use a command-line option to add a runtime. Here is an example:

```
export RUNTIME="nvidia"
export RUNTIME_PATH="/usr/bin/nvidia-container-runtime"
sudo dockerd --add-runtime=$RUNTIME=$RUNTIME_PATH
sudo systemctl restart docker
```

Check that the runtime is added:

```
docker info | grep -i runtime
```

The installation and configuration options for a runtime are usually provided by the runtime documentation. In this example, we used the NVIDIA container toolkit[28].

Does CRI Mean the Death of Docker?

You can't run a standalone container in production. You need an orchestrator like Kubernetes or Docker Swarm to manage the container lifecycle and resources.

Kubernetes is the most popular orchestration systems. With the evolving number of containers runtime, kubernetes aims to be more extensible and interface with more containers runtimes other than Docker. Originally, Kubernetes used Docker runtime to run containers, and it was still the default runtime until Kubernetes version 1.20[29].

"Don't panic"[30]", this was the title of a blog post published by the Kubernetes team to explain the deprecation of Docker as a container runtime in Kubernetes. The authors of the blog post stated:

> Docker as an underlying runtime is being deprecated in favor of runtimes that use the Container Runtime Interface (CRI) created for Kubernetes. Docker-produced images will continue to work in your cluster with all runtimes, as they always have. If you're an end-user of Kubernetes, not a whole lot will be changing for you. This doesn't mean the death of Docker, and it doesn't mean you can't, or shouldn't, use Docker as a development tool anymore. Docker is still a useful tool for building containers, and the images that result from running docker build can still run in your Kubernetes cluster.

[28] https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html
[29] https://kubernetes.io/blog/2020/12/08/kubernetes-1-20-release-announcement/
[30] https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/

Is Docker dead? Absolutely not. Docker is still a great tool for building containers, and the images that result from running docker build can still run in your Kubernetes cluster.

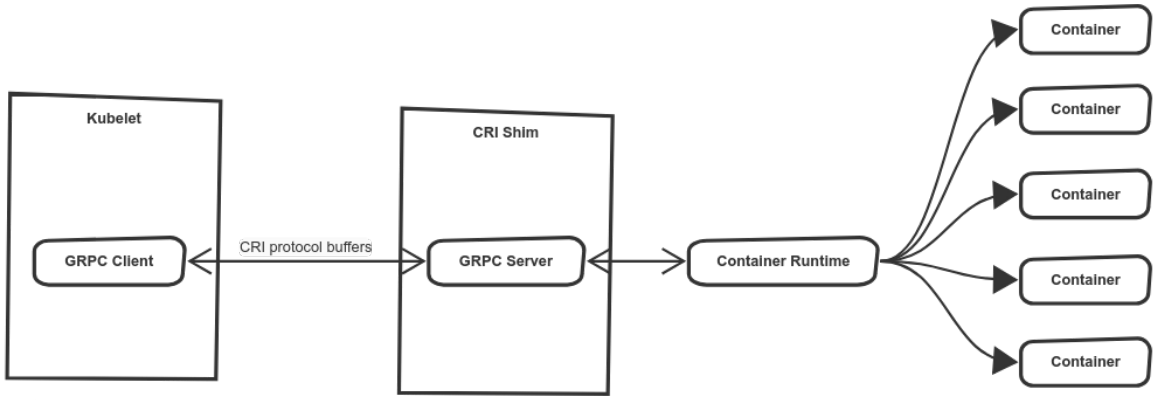
The same blog post explained the reasons behind this decision:

> You see, the thing we call "Docker" isn't actually one thing—it's an entire tech stack, and one part of it is a thing called "containerd," which is a high-level container runtime by itself. Docker is cool and useful because it has a lot of UX enhancements that make it really easy for humans to interact with while we're doing development work, but those UX enhancements aren't necessary for Kubernetes, because it isn't a human.
>
> As a result of this human-friendly abstraction layer, your Kubernetes cluster has to use another tool called Dockershim to get at what it really needs, which is containerd. That's not great, because it gives us another thing that has to be maintained and can possibly break. What's actually happening here is that Dockershim is being removed from Kubelet as early as v1.23 release, which removes support for Docker as a container runtime as a result. You might be thinking to yourself, but if containerd is included in the Docker stack, why does Kubernetes need the Dockershim?
>
> Docker isn't compliant with CRI, the Container Runtime Interface. If it were, we wouldn't need the shim, and this wouldn't be a thing. But it's not the end of the world, and you don't need to panic—you just need to change your container runtime from Docker to another supported container runtime.

Essentially, instead of changing the kubernetes code base each time and creating a new Kubernetes distribution when adding a new container runtime, Kubernetes upstream decided to create Container Runtime Interface (CRI)[31], which is a set of APIs and libraries that allows running different containers runtime in Kubernetes.

As a result, any interaction between kubernetes core and a supported runtime is performed through the CRI API.

[31] https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/



Docker and Kubernetes

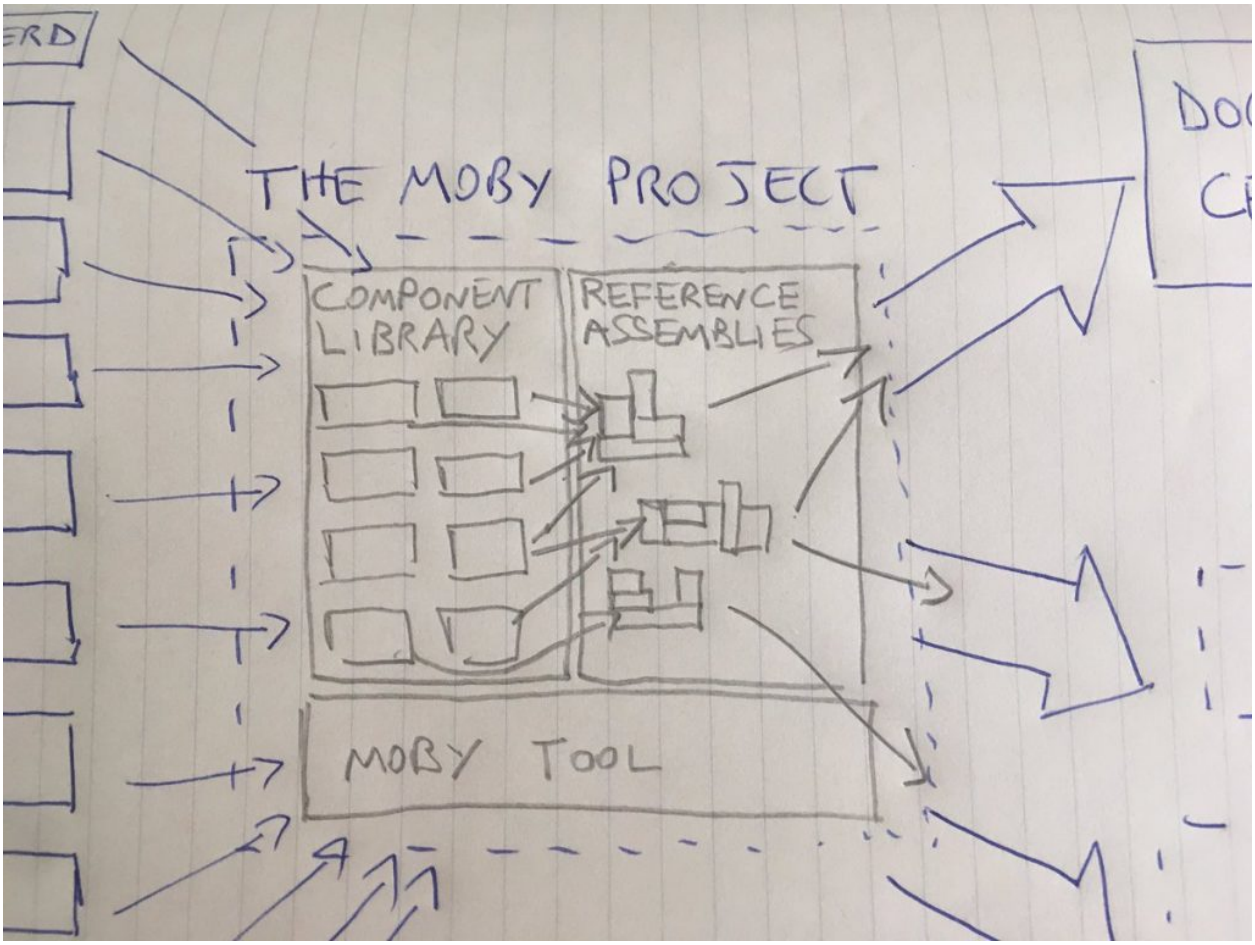
It is worth noting that CRI only supports OCI-compliant runtimes. These are some of the implementations of the Container Runtime Interface (CRI) that are OCI-compliant:

- **CRI-O**: The first container runtime created for the Kubernetes CRI interface. CRI-O is not meant to replace Docker, but it can be used as an alternative runtime specifically for Kubernetes.
- **CRI Containerd**: With cri-containerd, users can run Kubernetes clusters using containerd as the underlying runtime, without relying on Docker.
- **gVisor CRI**: A project developed by Google, which implements around 200 Linux system calls in userspace to provide enhanced security compared to Docker containers running directly on the Linux kernel with namespaces. The gVisor runtime integrates with Docker and Kubernetes, making it easy to run sandboxed containers.
- **CRI-O Kata Containers**: Kata Containers is an open-source project that builds lightweight virtual machines that integrate with the container ecosystem. CRI-O Kata Containers allows running Kata Containers on Kubernetes instead of using the default Docker runtime.

Runtimes and Kubernetes

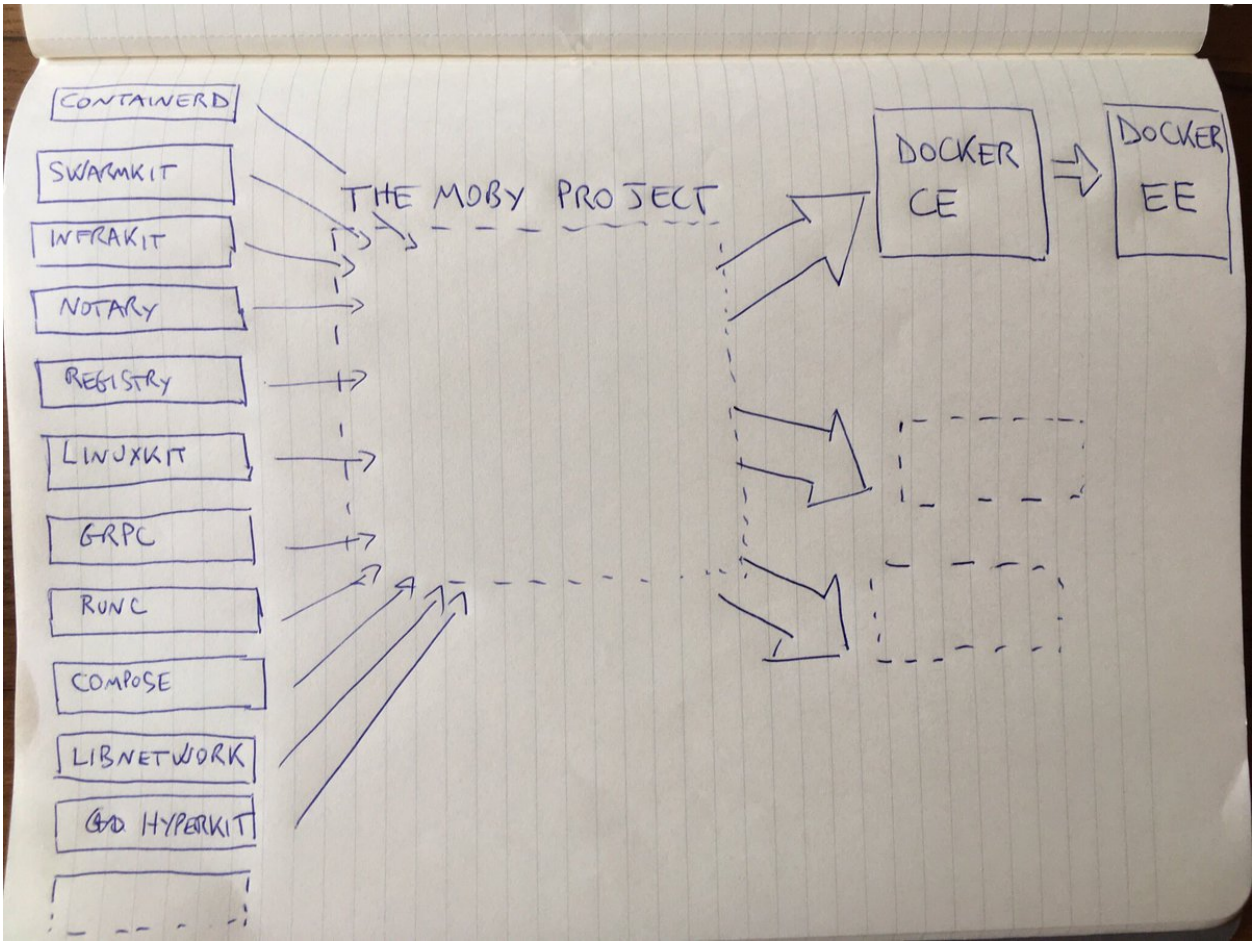
The Moby Project

When DotCloud launched Docker in **2013**, it was a single monolithic project. The goal was to create a single tool that can build, ship, and run containers. The project and the idea of building a single monolithic Docker platform were abandoned and gave birth to the Moby project, where Docker is now just one of the many components built on top of the Moby open-source framework,



The Moby Project in a nutshell: inside and outside by Solomon Hykes @solomonstre

Moby is a project that aims to organize and modularize the development of Docker. It provides an ecosystem for both development and production purposes but regular Docker users will not notice any difference.



The Moby Project in a nutshell: inside and outside by Solomon Hykes @solomonstre

This project provides a collection of components and a framework for building customized container-based systems. It is useful for developing and running Docker CE and EE, as Moby serves as Docker's upstream. Additionally, it can be used to create development and production environments for other runtimes and platforms.

Let's take a look at some of the components of the Moby project:

- **containerd**: This is the industry-standard core container runtime for Docker.
- **Linuxkit**: A tool used to build secure, portable, and lightweight operating systems for containers. It currently has support for local hypervisors like Hyper-V and VMware, as well as cloud-based platforms such as AWS, GCP, and Azure. It also works on bare metal with packet.net[32].
- **InfraKit**: A toolkit designed for creating and managing declarative, immutable, and self-healing infrastructures. InfraKit automates the setup and management of infrastructure to support distributed systems and higher-level container orchestration systems. It is particularly

[32] http://packet.net/

useful for scenarios like bootstrapping orchestration tools like Docker Swarm and Kubernetes, or creating autoscaling clusters across public clouds like AWS using autoscaling groups.

- **libnetwork**: A native Go implementation for connecting containers. It enables the development of network drivers and plugins, satisfying the need for "composable" networking in containers.
- **Notary**: A tool for publishing and managing trusted collections of content. It provides a mechanism for signing and verifying content using digital signatures. It is particularly useful for ensuring the integrity and authenticity of container images.
- **SwarmKit**: A native clustering system for Docker. It provides a native clustering solution for Docker, which is a lightweight alternative to Kubernetes.

Installing and Using Docker

Installing Docker

Docker Engine is available on various Linux distros, macOS, and Windows 10 through Docker Desktop, as well as through a static binary installation.

Docker is offered in three tiers:

- Docker Engine - Community: This tier includes the core container engine, along with built-in orchestration, networking, and security features.
- Docker Engine - Enterprise: It consists of certified infrastructure, plugins, and ISV containers.
- Docker Enterprise: This tier includes image management, container app management, and image security scanning features.

In this course, we will be installing and using the community edition of Docker.

The installation process is well-explained in the official Docker documentation[33]. There are different channels available, namely stable, test, and nightly. We will be installing and using the stable build.

For this guide, we will be using Ubuntu 22.04 LTS (Focal Fossa) as our host OS. If you are using another distribution or another operating system, adapting commands should not be difficult.

Mac Users

If you are a Mac user, you can download Docker Desktop for Mac from Docker official website[34].

If your Mac has an Intel CPU, you need to have at least 4 GB of RAM and install VirtualBox prior to version 4.3.30 must not be installed as it is not compatible with Docker Desktop.

If your Mac comes with an Apple silicon CPU, you need to install Rosetta[35] 2:

```
softwareupdate --install-rosetta
```

[33] https://docs.docker.com/install/
[34] https://docs.docker.com/desktop/install/mac-install/
[35] https://en.wikipedia.org/wiki/Rosetta_(software)

Windows Users

If you are a Windows user, you can download Docker Desktop for Windows from Docker official website[36].

You will need either WSL 2 (Windows Subsystem for Linux) or Hyper-V to run Docker Desktop for Windows. In both cases, you will need to enable virtualization[37] on your machine.

Docker for Linux

It depends on your distribution, so if you are going to install Docker on a CentOS machine, follow these instructions[38].

Check the requirements first. You need CentOS 7 or later like CentOS 8 (stream) or CentOS 9 (stream). Archived versions aren't supported or tested.

The `centos-extras` repository must be enabled. This repository is enabled by default, but if you have disabled it, you need to re-enable it[39]. The `overlay2` storage driver is recommended.

Now, start by uninstalling any older version of Docker:

```
sudo yum remove docker \
docker-client \
docker-client-latest \
docker-common \
docker-latest \
docker-latest-logrotate \
docker-logrotate \
docker-engine
```

Then set up the repository:

```
sudo yum install -y yum-utils device-mapper-persistent-data lvm2
sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-c\
e.repo
```

Now install Docker:

```
sudo yum install docker-ce docker-ce-cli containerd.io
```

[36] https://docs.docker.com/desktop/install/windows-install/
[37] https://docs.docker.com/desktop/troubleshoot/topics/#virtualization
[38] https://docs.docker.com/install/linux/docker-ce/centos/
[39] https://wiki.centos.org/AdditionalResources/Repositories

If you are using Ubuntu, follow these instructions[40].

You need the 64-bit version of one of Ubuntu Focal 20.04 (LTS) or later.

Uninstall all conflicting packages:

```
for pkg in docker.io docker-doc docker-compose docker-compose-v2 podman-docker conta\
inerd runc; do sudo apt-get remove $pkg; done
```

Install using APT:

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc\
/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg

# Add the repository to Apt sources:
echo \
  "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.gpg] \
https://download.docker.com/linux/ubuntu \
  "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update

# Install Docker Engine:
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin dock\
er-compose-plugin
```

After installing Docker in any of the above ways, you need to start the Docker service:

```
sudo systemctl start docker
```

Verify the installation by running a container:

```
sudo docker run hello-world
```

Linux: Alternative Installation Method

Docker provides a script to install the latest version of Docker and it can faster than the first way:

[40] https://docs.docker.com/install/linux/docker-ce/ubuntu/

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh
```

Linux: Using Docker as a Non-root User

Don't forget to add your current user to the Docker users group to avoid using `sudo` each time you want to run a Docker command:

```
# Create the docker group:
sudo groupadd docker
# Add your user to the docker group:
sudo usermod -aG docker $USER
# Log out and log back in so that your group
# membership is re-evaluated (or run `newgrp docker`):
newgrp docker
# Verify that you can run docker commands without sudo:
docker run hello-world
```

Docker CLI

When we tested the Docker installation, we used the `docker run hello-world` command.

```
sudo docker run hello-world
```

Running the above commands will pull the hello-world image from the Docker Hub and run it in a container.

These are the steps that Docker followed to run the hello-world container (you can see them in the output of the command):

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

In essence, this succinctly captures Docker's container creation process. Diving into a detailed reiteration would be redundant, especially considering the energy[41] consumption Docker incurs performing this operation.

[41] https://arxiv.org/pdf/1705.01176.pdf

General Information About Your Docker Installation

Docker is a highly active project, and its code undergoes frequent changes. To gain a better understanding of this technology, I have been following the project on Github[42] and referring to its issues whenever I encountered problems or discovered bugs.

It is crucial to know the version of Docker you are using on your production servers. You can use the command `docker -v` to check the version and build number.

```
docker -v
```

For more general information about the server/client version, architecture, Go version, and more, you can use the command `docker version`.

Example:

```
docker version
```

Docker Help

You can view the Docker help using the command line:

```
docker --help
```

You will see different lists of commands and options:

Common commands:

```
run         Create and run a new container from an image
exec        Execute a command in a running container
ps          List containers
build       Build an image from a Dockerfile
pull        Download an image from a registry
push        Upload an image to a registry
images      List images
login       Log in to a registry
logout      Log out from a registry
search      Search Docker Hub for images
version     Show the Docker version information
info        Display system-wide information
```

Management commands:

[42] https://github.com/moby/moby/issues

```
builder     Manage builds
buildx      Docker Buildx
compose     Docker Compose
container   Manage containers
context     Manage contexts
image       Manage images
manifest    Manage Docker image manifests and manifest lists
network     Manage networks
plugin      Manage plugins
system      Manage Docker
trust       Manage trust on Docker images
volume      Manage volumes
```

Swarm commands:

```
swarm       Manage Swarm
```

And other commands:

```
attach      Attach local standard input, output, and error streams to a running cont\
ainer
commit      Create a new image from a container's changes
cp          Copy files/folders between a container and the local filesystem
create      Create a new container
diff        Inspect changes to files or directories on a container's filesystem
events      Get real time events from the server
export      Export a container's filesystem as a tar archive
history     Show the history of an image
import      Import the contents from a tarball to create a filesystem image
inspect     Return low-level information on Docker objects
kill        Kill one or more running containers
load        Load an image from a tar archive or STDIN
logs        Fetch the logs of a container
pause       Pause all processes within one or more containers
port        List port mappings or a specific mapping for the container
rename      Rename a container
restart     Restart one or more containers
rm          Remove one or more containers
rmi         Remove one or more images
save        Save one or more images to a tar archive (streamed to STDOUT by default)
start       Start one or more stopped containers
stats       Display a live stream of container(s) resource usage statistics
```

```
stop        Stop one or more running containers
tag         Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
top         Display the running processes of a container
unpause     Unpause all processes within one or more containers
update      Update configuration of one or more containers
wait        Block until one or more containers stop, then print their exit codes
```

If you need more help about a specific command like `cp` or `rmi` , you need to type:

```
docker cp --help
docker rmi --help
```

In some cases, you have a 3rd level of help:

```
docker swarm init --help
```

Docker Events

To start this section, let's run a MariaDB container and list Docker Events . For this manipulation, you can use terminator[43] to split your screen into two and notice at the same time the event's output while typing the following command:

```
docker run --name mariadb -e MYSQL_ROOT_PASSWORD=password -v /data/db:/var/lib/mysql\
 -d mariadb
```

The events launched by the last command can be seen using:

```
docker events
```

You will see a stream of events that are related to the image pulling and the container creation.

```
image pull <IMAGE_NAME>:<TAG> (name=<NAME>, org.opencontainers.image.authors=<AUTHOR\
S>, org.opencontainers.image.base.name=<BASE_IMAGE_NAME>, org.opencontainers.image.d\
escription=<DESCRIPTION>, org.opencontainers.image.documentation=<DOCUMENTATION_URL>\
, org.opencontainers.image.licenses=<LICENSE>, org.opencontainers.image.ref.name=<RE\
F_NAME>, org.opencontainers.image.source=<SOURCE_URL>, org.opencontainers.image.titl\
e=<TITLE>, org.opencontainers.image.url=<URL>, org.opencontainers.image.vendor=<VEND\
OR>, org.opencontainers.image.version=<VERSION>)

container create <CONTAINER_ID> (image=<IMAGE_NAME>, name=<CONTAINER_NAME>, org.open\
containers.image.authors=<IMAGE_AUTHORS>, org.opencontainers.image.base.name=<BASE_I\
MAGE_NAME>, org.opencontainers.image.description=<IMAGE_DESCRIPTION>, org.opencontai\
ners.image.documentation=<IMAGE_DOC_URL>, org.opencontainers.image.licenses=<IMAGE_L\
ICENSES>, org.opencontainers.image.ref.name=<IMAGE_REF_NAME>, org.opencontainers.ima\
ge.source=<IMAGE_SOURCE_URL>, org.opencontainers.image.title=<IMAGE_TITLE>, org.open\
containers.image.url=<IMAGE_URL>, org.opencontainers.image.vendor=<IMAGE_VENDOR>, or\
g.opencontainers.image.version=<IMAGE_VERSION>)

network connect <NETWORK_ID> (container=<CONTAINER_ID>, name=<NETWORK_NAME>, type=<N\
ETWORK_TYPE>)

container start <CONTAINER_ID> (image=<IMAGE_NAME>, name=<CONTAINER_NAME>, org.openc\
ontainers.image.authors=<IMAGE_AUTHORS>, org.opencontainers.image.base.name=<BASE_IM\
```

[43] https://terminator-gtk3.readthedocs.io/en/latest/

```
AGE_NAME>, org.opencontainers.image.description=<IMAGE_DESCRIPTION>, org.opencontain\
ers.image.documentation=<IMAGE_DOC_URL>, org.opencontainers.image.licenses=<IMAGE_LI\
CENSES>, org.opencontainers.image.ref.name=<IMAGE_REF_NAME>, org.opencontainers.imag\
e.source=<IMAGE_SOURCE_URL>, org.opencontainers.image.title=<IMAGE_TITLE>, org.openc\
ontainers.image.url=<IMAGE_URL>, org.opencontainers.image.vendor=<IMAGE_VENDOR>, org\
.opencontainers.image.version=<IMAGE_VERSION>)
```

To understand line by line the event stream output:

- `image pull`: The image is pulled from the public repository by it identifier *Mariadb* .
- `container create`: The container is created from the pulled image and it was given a Docker identifier (`CONTAINER_ID`).
- `network connect`: The container is attached to a network called `bridge` having a unique identifier. At this stage the container is not running yet.
- `container start`: The container at this stage is running on your host system with the same identifier (`CONTAINER_ID`).

In total, there are over 50 distinct events, varying based on the specific object in question. By 'objects,' we're referring to elements such as containers, images, plugins, volumes, networks, daemons, services, nodes, secrets, and configs. It's worth noting that while many of these events are tied to the Docker engine itself, others are specifically associated with Docker's orchestration solution, Docker Swarm. Following is a list of the most common events:

- Docker containers report the following events:
  - `attach`
  - `commit`
  - `copy`
  - `create`
  - `destroy`
  - `detach`
  - `die`
  - `exec_create`
  - `exec_detach`
  - `exec_die`
  - `exec_start`
  - `export`
  - `health_status`
  - `kill`
  - `oom`
  - `pause`

    - `rename`
    - `resize`
    - `restart`
    - `start`
    - `stop`
    - `top`
    - `unpause`
    - `update`
- Docker images report the following events:
    - `delete`
    - `import`
    - `load`
    - `pull`
    - `push`
    - `save`
    - `tag`
    - `untag`
- Docker plugins report the following events:
    - `enable`
    - `disable`
    - `install`
    - `remove`
- Docker volumes report the following events:
    - `create`
    - `destroy`
    - `mount`
    - `unmount`
- Docker networks report the following events:
    - `create`
    - `connect`
    - `destroy`
    - `disconnect`
    - `remove`
- Docker daemons report the following events:
    - `reload`
- Docker services report the following events:
    - `create`
    - `remove`
    - `update`

- Docker nodes report the following events:
  - `create`
  - `remove`
  - `update`
- Docker secrets report the following events:
  - `create`
  - `remove`
  - `update`
- Docker configs report the following events:
  - `create`
  - `remove`
  - `update`

It is possible to filter the events by type using the `--filter` flag:

```
# List all events related to containers:
docker events --filter type=container
# List all events related to a specific container:
docker events --filter container=<CONTAINER_ID>
# List all events called "stop":
docker events --filter event=stop
# Using multiple filters:
docker events --filter event=stop --filter container=<CONTAINER_ID>
```

Using Docker API To List Events

One of the most important features of Docker is its API that makes it possible to interact with Docker using HTTP requests. You can, in fact, use the same API to see Docker Events while using the command line.

Start by installing curl[44] if it is not already installed on your system:

```
sudo apt install curl
```

Type the following command to see the Docker Events stream:

```
curl --unix-socket /var/run/docker.sock http://localhost/events
```

Open another terminal window (or a new tab) and type:

[44] https://curl.haxx.se/

```
docker pull mariadb
docker rmi -f mariadb
docker pull mariadb
```

These are the 4 events reported by the 3 commands typed above:

- Pulling an image that already exists in the host:


```
{
   "status":"pull",
   "id":"mariadb:latest",
   "Type":"image",
   "Action":"pull",
   "Actor":{
      "ID":"mariadb:latest",
      "Attributes":{
         "name":"mariadb",
         "org.opencontainers.image.authors":"MariaDB Community",
         "org.opencontainers.image.base.name":"docker.io/library/ubuntu:jammy",
         "org.opencontainers.image.description":"MariaDB Database for relational SQL\
",
         "org.opencontainers.image.documentation":"https://hub.docker.com/_/mariadb/\
",
         "org.opencontainers.image.licenses":"GPL-2.0",
         "org.opencontainers.image.ref.name":"ubuntu",
         "org.opencontainers.image.source":"https://github.com/MariaDB/mariadb-docke\
r",
         "org.opencontainers.image.title":"MariaDB Database",
         "org.opencontainers.image.url":"https://github.com/MariaDB/mariadb-docker",
         "org.opencontainers.image.vendor":"MariaDB Community",
         "org.opencontainers.image.version":"11.1.2"
      }
   },
   "scope":"local",
   "time":1698933573,
   "timeNano":1698933573154066725
}
```


- Untagging and deleting the image:

```
{
   "status":"untag",
   "id":"sha256:f35870862d64d0e29598fba1d7f75cfefeb3f891cb22b3e2d4459c903e666554",
   "Type":"image",
   "Action":"untag",
   "Actor":{
      "ID":"sha256:f35870862d64d0e29598fba1d7f75cfefeb3f891cb22b3e2d4459c903e666554",
      "Attributes":{
         "name":"sha256:f35870862d64d0e29598fba1d7f75cfefeb3f891cb22b3e2d4459c903e66\
6554",
         "org.opencontainers.image.authors":"MariaDB Community",
         "org.opencontainers.image.base.name":"docker.io/library/ubuntu:jammy",
         "org.opencontainers.image.description":"MariaDB Database for relational SQL\
",
         "org.opencontainers.image.documentation":"https://hub.docker.com/_/mariadb/\
",
         "org.opencontainers.image.licenses":"GPL-2.0",
         "org.opencontainers.image.ref.name":"ubuntu",
         "org.opencontainers.image.source":"https://github.com/MariaDB/mariadb-docke\
r",
         "org.opencontainers.image.title":"MariaDB Database",
         "org.opencontainers.image.url":"https://github.com/MariaDB/mariadb-docker",
         "org.opencontainers.image.vendor":"MariaDB Community",
         "org.opencontainers.image.version":"11.1.2"
      }
   },
   "scope":"local",
   "time":1698933584,
   "timeNano":1698933584450460765
}

{
   "status":"delete",
   "id":"sha256:f35870862d64d0e29598fba1d7f75cfefeb3f891cb22b3e2d4459c903e666554",
   "Type":"image",
   "Action":"delete",
   "Actor":{
      "ID":"sha256:f35870862d64d0e29598fba1d7f75cfefeb3f891cb22b3e2d4459c903e666554",
      "Attributes":{
         "name":"sha256:f35870862d64d0e29598fba1d7f75cfefeb3f891cb22b3e2d4459c903e66\
6554"
      }
   },
```

```
    "scope":"local",
    "time":1698933584,
    "timeNano":1698933584569420538
}
```


- Pulling the image


```
{
    "status":"pull",
    "id":"mariadb:latest",
    "Type":"image",
    "Action":"pull",
    "Actor":{
        "ID":"mariadb:latest",
        "Attributes":{
            "name":"mariadb",
            "org.opencontainers.image.authors":"MariaDB Community",
            "org.opencontainers.image.base.name":"docker.io/library/ubuntu:jammy",
            "org.opencontainers.image.description":"MariaDB Database for relational SQL\
",
            "org.opencontainers.image.documentation":"https://hub.docker.com/_/mariadb/\
",
            "org.opencontainers.image.licenses":"GPL-2.0",
            "org.opencontainers.image.ref.name":"ubuntu",
            "org.opencontainers.image.source":"https://github.com/MariaDB/mariadb-docke\
r",
            "org.opencontainers.image.title":"MariaDB Database",
            "org.opencontainers.image.url":"https://github.com/MariaDB/mariadb-docker",
            "org.opencontainers.image.vendor":"MariaDB Community",
            "org.opencontainers.image.version":"11.1.2"
        }
    },
    "scope":"local",
    "time":1698933681,
    "timeNano":1698933681839994581
}
```


Each event has a status (e.g.: pull, untag), a resource identifier (id) and a type (e.g.: image, container, network), and other information like:

- Actor: The actor is the object that is the source of the event. It can be an image, a container, a network, a volume, a plugin, a daemon, a service, a node, a secret, or a config.

- Scope: The scope is the type of the object that is the target of the event. It can be a local or a swarm.
- Time: The time is the time when the event was generated.

Docker Containers

Docker is a popular platform that effectively addresses various issues in modern IT but in its core, it is a containerization platform. Docker helps users manage volumes, images and networks and all of this is done to create and run containers. So containers are the core of Docker and in this section, we will learn how to create and manage containers.

A container is essentially an image that is running.

To create a container, you can use the `docker create` command but additional commands are used to start, pause, stop, run, and delete containers.

A container has a lifecycle with various phases that correspond to defined events:

- `attach`
- `commit`
- `copy`
- `create`
- `destroy`
- `detach`
- `die`
- `exec_create`
- `exec_detach`
- `exec_die`
- `exec_start`
- `export`
- `health_status`
- `kill`
- `oom`
- `pause`
- `rename`
- `resize`
- `restart`
- `start`
- `stop`
- `top`
- `unpause`
- `update`

We will explore these commands and most of the lifecycle phases in this section.

Creating Containers

To create a container, simply run a Docker image using the `docker create` command.

```
docker create [OPTIONS] IMAGE [COMMAND] [ARG...]
```

A simple example to start using this command is running:

```
docker create hello-world
```

Verify that the container was created by typing:

```
docker ps -a
```

or

```
docker ps --all
```

Docker would pick a random name for your container if you did not explicitly specify a name but you can set a name:

```
docker create --name hello-world-container hello-world
```

Verify the creation of the container:

```
docker ps -a
```

The `docker create` command uses the specified image and add a new writable layer to create the container and waits for the specified command to run inside the created container.

The container is created but not running yet. You can start it using the `docker start` command:

```
docker start web_server
```

In practice, the `docker create` command is not used as much as the `docker run` command which is a combination of `docker create` and `docker start` commands.

In reality the `docker run` command is an alias for `docker container run` command which is an alias for `docker container create && docker container start` commands.

Running Containers

The `docker run` command is used to create and start a container in one command.

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

The `docker run` command is used to create and start a container in one command.

```
docker run hello-world
```

Instead of using the following commands:

```
docker create --name web_server -it nginx
docker start web_server
```

You can use the `docker run` command:

```
# Remove the old container if it exists: docker rm web_server
docker run --name web_server nginx
```

Now if you want to run an interactive container, you can use the `-it` flag:

```
docker run -it ubuntu bash
# or
# docker run --interactive --tty ubuntu bash
```

By "interactive mode", we mean that you can interact with the container using the terminal. Therefore, you need to attach a terminal to the container. The `-it` flag is a combination of two flags:

- `i` or `interactive`: Keep STDIN open even if not attached
- `t` or `tty`: Allocate a pseudo-TTY

By default, Docker launches a container and assumes that you want to attach to its STDIN, STDOUT, and STDERR. If you want to run a container in the background, you can use the `-d` flag.

Additionally, we usually run containers in background mode using the `-d` flag:

```
docker run -d --name web_server nginx
```

The `-d` flag is short for `--detach` which means that the container will run in the background.

Restarting Containers

Let's start this container:

```
# Remove the old container if it exists: docker rm web_server
docker run -d --name web_server nginx
```

This will print an ID of the container.

Example:

```
5a9c835bb0ba1c3e3f61d351f7899622af51d169f4367c7dd65d58c04746366b
```

This is the long ID of the container. You can use the first 3 or 4 characters of the ID to reference the container and restart it using the `docker restart` command:

```
docker restart 5a9
```

Or the short ID that you can see when you type `docker ps`. You can also use the name of the container:

```
docker restart web_server
```

Referencing a container using the short/long ID, the first few characters of the ID, or the name of the container is the same and it applies to all Docker commands that accept a container as an argument.

Pausing and Unpausing Containers

To pause/unpause the Nginx container you can use the following commands:

```
docker pause web_server
docker unpause web_server
```

You can verify the STATUS of the container using `docker ps`.

```
docker ps --filter "name=web_server" --format "{{.ID}}: {{.Status}}"
```

Stopping Containers

You can use the `stop` command in the following way to stop one or multiple containers:

```
docker stop [OPTIONS] CONTAINER [CONTAINER...]
```

Example:

```
docker stop web_server
```

We can also stop a container after a specific number of seconds:

```
docker stop --time 20 web_server
```

The last command will wait 20 seconds before killing the container. The default number of seconds is 10, so executing `docker stop my_container` will wait 10 seconds.

Executing the `docker stop` command asks nicely to stop the container..

Stopping a container is sending a `SIGTERM` signal to the root process (PID 1) in the container, but if the process and if it has not exited within the timeout period (10 seconds), a `SIGKILL` signal will be sent.

You can customize the signal to send using the `--signal` flag:

```
docker stop --signal=SIGKILL web_server
```

Killing Containers

The `docker kill` command is used to terminate containers and force them to exit. By default, the command sends a `SIGKILL` signal to the containers, but this can be changed by using the `--signal` option.

The signal can be specified either as `SIG[NAME]` or as a numeric value from the kernel's syscall table. It's important to note that certain signals, like SIGHUP, may not necessarily stop the container depending on its main process.

It's worth mentioning that syscall numbers can differ across different architectures. For the x86_64 architecture, you can refer to syscall_64.tbl[45], and for the x86 architecture, you can refer to syscall_-32.tbl[46].

Here is an example of how to use the `docker kill` command:

```
docker kill [OPTIONS] CONTAINER [CONTAINER...]
```

Docker's kill command does not gracefully terminate the container process. By default, it sends a `SIGKILL` signal, which is equivalent to using the kill -9 command in Linux. Alternatively, you can use the `SIGINT` or `SIGTERM` signals to terminate the container process.

[45] https://github.com/torvalds/linux/blob/v4.17/arch/x86/entry/syscalls/syscall_64.tbl#L11
[46] https://github.com/torvalds/linux/blob/v4.17/arch/x86/entry/syscalls/syscall_32.tbl#L17

```
docker kill --signal=SIGINT web_server
docker kill --signal=SIGTERM web_server
```

A common example is using `SIGHUP` to reload the configuration of a process. For instance, if you have an NGINX server running in a container and you want to reload its configuration, you can use the following command:

```
docker kill --signal=SIGHUP web_server
```

There are three different approaches to reload the configuration of a process running in a container:

- The first approach is to send a signal to the container's PID.
- The second option is to execute the kill command inside the container.
- The third way to do it is by using the `docker kill` command.

```
kill -SIGHUP [PID]
docker exec [CONTAINER] kill -SIGHUP 1 # The main process PID is 1
docker kill --signal=HUP [CONTAINER]
```

Removing Containers

To remove one or multiple containers, you can use the `docker rm` command.

```
docker rm [OPTIONS] CONTAINER [CONTAINER...]
```

To see this in action, run and remove a container:

```
docker run -it -d --name my_container nginx
container_id=$(docker ps -a --filter "name=my_container" --format "{{.ID}}")
docker rm $container_id
# or docker rm my_container
```

If the container is stopped, it will be removed. However, you cannot remove a running container. To force the removal of a running container, you should add the `-f` or `--force` option:

```
docker rm -f $container_id
# or docker rm -f my_container
```

Adding the `--force` command is an alternative to executing two commands:

```
docker stop my_container
docker rm my_container
```

Removal not only stops a running container but also erases all of its traces.

Container Lifecycle

A container may be in one of the following states:

- Created: The image is pulled, and the container is configured to run but not running yet.
- Running: Docker creates a new RW layer at the top of the pulled image and starts running the container.
- Paused: Docker container is paused but ready to run again without any restart, just run the `unpause` command.
- Stopped: The container is stopped and not running. You should start another new container if you want to run the same application.
- Removed: A container could not be removed unless it is stopped. In this status, there is no container at all (neither in the disk nor in the memory of the host machine).

To see the status of a container, you can use the `docker ps` command:

```
docker ps --format "{{.Names}} is {{.Status}}"
```

All of these states are part of the container lifecycle and can be achieved using the commands we have learned so far:

- `docker create`
- `docker start`
- `docker run`
- `docker pause`
- `docker unpause`
- `docker restart`
- `docker stop`
- `docker kill`
- `docker rm`

Starting Containers Automatically

You have the option to automatically restart Docker containers in the following scenarios: when the container stops, when it exits with an error code, or when the Docker daemon restarts (unless the container was manually stopped). Alternatively, you can choose to deactivate automated restarts altogether.

This table from the official documentation explains the different flags used to implement all of the mentioned policies:

Flag	Description
`no`	Do not automatically restart the container. (the default)
`on-failure[:max-retries]`	Restart the container if it exits due to an error, which manifests as a non-zero exit code. Optionally, limit the number of times the Docker daemon attempts to restart the container using the `:max-retries` option. The `on-failure` policy only prompts a restart if the container exits with a failure. It doesn't restart the container if the daemon restarts.
`always`	Always restart the container if it stops. If it is manually stopped, it is restarted only when Docker daemon restarts, or the container itself is manually restarted. (See the second bullet listed in restart policy details[47])
`unless-stopped`	Similar to `always`, except that when the container is stopped (manually or otherwise), it is not restarted even after Docker daemon restarts.

These are the various policies available for automatically restarting containers. You can use the `--restart=[POLICY]` flag to specify the restart policy when creating or running a container.

```
# Always restart the container regardless of the exit status
docker run --restart=always nginx
# Restart the container unless it is explicitly stopped
docker run --restart=unless-stopped nginx
# Restart the container when there's an error
docker run --restart=on-failure nginx
# Restart the container when there's an error with a maximum of 10 restart attempts
docker run --restart=on-failure:10 nginx
```

Accessing Containers Ports

When you run a container, you can expose ports using the `-p` flag. This flag takes two arguments: the port on the host machine and the port inside the container. For example, if you want to expose port 80 on the host machine and port 80 inside the container, you can use the following command:

```
docker run -p 80:80 nginx
```

This means that any request to port 80 on the host machine will be forwarded to port 80 inside the container.

However, three conditions must be met for this to work:

[47] https://docs.docker.com/config/containers/start-containers-automatically/#restart-policy-details

- The Dockerfile of the image powering the container (Nginx image in our case) should expose the port that you want to forward traffic to (port 80 in our case).
- The application running inside the container should be listening on the port that is exposed (Nginx listens on port 80 by default).
- The port on the host machine should not be used by another application.

Keep in mind that these 3 conditions should be met for the port forwarding to work. Later in the following sections, we will explore these concepts in more detail.

It's worth mentioning that:

- You can expose multiple ports using the `-p` flag. For example, if you want to expose ports 80 and 443 on the host machine and ports 80 and 443 inside the container, you can use the following command:

```
docker run -d -p 80:80 -p 443:443 --name nginx_two_ports nginx
```

- You don't have the have the same port on the host machine and inside the container. For example, if you want to use port 8080 on the host machine and port 80 inside the container, you can use the following command:

```
docker run -d -p 8080:80 --name nginx_different_ports nginx
```

- If you don't specify the port in the host, Docker will pick a random port and forward traffic to the specified port inside the container. For example, if you want to use port 80 inside the container, you can use the following command:

```
docker run -d -p 80 --name nginx_random_port nginx
```

You can get the host port using:

```
docker port nginx_random_port
```

Running Docker In Docker

Suppose you are running your CI/CD pipeline using Jenkins. In this scenario, your Jenkins instance is containerized using Docker. Now, within this Jenkins container, you want to execute Docker commands. This situation is commonly referred to as Docker-in-Docker (DinD).

There are various methods to achieve Docker-in-Docker, but the most optimal approach is to bind-mount the Docker socket into the currently running Jenkins container. This technique is different from running a Docker container inside another Docker container, but it achieves the same result.

Let's create a Dockerfile for our Jenkins container:

```
cat <<EOF > Dockerfile
FROM jenkins/jenkins:lts
USER root
RUN curl -fsSL https://get.docker.com -o get-docker.sh && sh get-docker.sh
EOF
```

Build and run the container:

```
docker build -t jenkins-docker .
docker run -d -p 8080:8080 -p 50000:50000 --name jenkins -v /var/run/docker.sock:/va\
r/run/docker.sock jenkins-docker
```

Now, go inside the container and execute the `docker ps` command:

```
docker exec -it jenkins bash
docker ps
```

To view the list of containers running on the host machine, you can use the Docker dashboard. If you need to launch Docker containers for testing purposes, CI/CD, etc., you can do so through the Jenkins dashboard.

In his blog post "Using Docker-in-Docker for your CI or testing environment? Think twice"[48], Jérôme Petazzoni suggests different methods for running Docker in Docker. He also mentions that the most effective approach is to utilize the Docker API or mount the Docker socket.

[48] https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/

Managing Containers Resources

When you create a container, it is associated with specific cgroups (Control Groups) that manage its resources. cgroups allow you to set limits on resources such as memory, CPU, and block I/O for the container, ensuring it does not consume more than its allocated resources.

Control Groups act as controllers that monitor and limit the resource usage of a process or group of processes. If the container exceeds the allocated resources, the cgroup will terminate the process(es).

By default, when you start a container, it has no limitations. It can utilize as much CPU and memory as permitted by the host's kernel scheduler. You can restrict the resources available to a container by specifying the maximum amount of memory and CPU it can use.

Similarly, you can allocate a reserved amount of memory and CPU for a container. This reserved amount of resources is guaranteed to be available to the container.

Setting limits or reservations on available resources for a container is important. Not setting them can result in a container using all available resources on the host or not having enough resources to run.

By setting these limits and reservations, you can:

- Prevent other containers on the same host from being starved of resources.
- Predict the performance of a container.
- Enhance security by minimizing the impact of a compromised container.
- Manage costs by restricting the resources available to a container.
- Ensure a minimum level of resources for a container, preventing it from being terminated by the kernel during memory pressure on the host.
- Avoid unexpected failures due to insufficient resources.
- Isolate containers from resource-intensive neighbors.

When using an orchestrator like Kubernetes, it is often unnecessary to set limits or reservations on container resources as the orchestrator handles this for you. However, when running standalone containers, it is good practice to set them for each container.

In the following sections, we will explore all of this in more detail.

Memory Usage Reservations and Limits

We are going to see the available options to manage memory usage for a container by following some practical examples.

Note that in the following examples, you can use `b`, `k`, `m`, `g`, to indicate bytes, kilobytes, megabytes, or gigabytes.

Setting the maximum amount of memory available to a container:

```
docker run -d --memory="512m" nginx
```

In the above example, we are setting the maximum amount of memory available to the container to 512 MB. If the container exceeds this limit, it will be terminated by the kernel. The minimum value you can set is 6 MB.

Setting the amount of memory the container is allowed to swap to disk:

```
docker run -d  --memory="512m" --memory-swap="1000m" nginx
```

In the above example, we are setting the amount of memory the container is allowed to swap to disk to 1000 MB. The `--memory` must be set for the `--memory-swap` to work.

The configured swap represents the total amount of memory and swap that can be used by the container. If the memory is set to 512 MB and the swap is set to 1000 MB, the container can use 512m from memory and 488m from swap (1000m - 512m). Therefore, the amount of swap should be always greater than the amount of memory. Otherwise, the swap will be ignored.

If you want to use unlimited swap, you can set the `--memory-swap` to `-1`.

```
docker run -d  --memory="512m" --memory-swap="-1" nginx
```

Setting the memory swappiness for a container:

```
docker run -d --memory-swappiness="10" nginx
```

Swappiness is a property for the Linux kernel that changes the balance between swapping out runtime memory and it could be a number between 0 and 100. The lower the value, the less the kernel will swap memory pages. The higher the value, the more the kernel will swap memory pages. If you don't set `--memory-swappiness`, the value is inherited from the host machine.

By setting the `--memory-swappiness` value to a lower number, like 10, for a specific Docker container, you instruct the kernel to be less aggressive about swapping out the memory pages of that container. This ensures that the container's memory pages are swapped out only when necessary which improves performance.

It's worth noting that memory has a faster access time than swap. Therefore, it is better to keep the memory pages in memory as much as possible.

Allocating a reserved amount of memory for a container:

```
docker run -d --memory-reservation="256m" nginx
```

The flag `--memory-reservation` is used to set the amount of memory that is guaranteed to be available to a container. If the host is under memory pressure, the kernel will not reclaim the memory reserved for the container. We call this a soft limit and it should be set lower than the `--memory` flag.

Setting the kernel memory limit for a container:

```
docker run -d --kernel-memory="256m" nginx
```

The flag `--kernel-memory` is used to set the maximum amount of kernel memory that can be used by the container. This memory is used by the kernel for TCP/IP networking, the VFS cache, and other kernel allocations.

The Kernel memory limits are expressed in terms of the memory allowed for the container. If you start a container with a total memory limit of 1GB and set the kernel memory limit to 200MB, the container has up to 1GB of total memory to use. Out of this, a maximum of 200MB can be used for kernel-related tasks, and the remaining 800MB is available for user-space tasks.

These are some scenarios to better understand how the kernel memory limit works:

- Unlimited memory, unlimited kernel memory: The default setting.
- Unlimited memory, limited kernel memory: When there's more total memory demand than what's available, you can limit only the memory used by the system tasks in containers. This ensures system operations don't exceed the host's memory. If a container needs more memory, it must wait.
- Limited memory, unlimited kernel memory: Only user memory is capped.
- Limited memory, limited kernel memory: Helps in troubleshooting memory issues by constraining both user and kernel memory.

Disabling the OOM Killer for a container:

```
docker run -d --oom-kill-disable nginx
```

When using the `--oom-kill-disable` flag, the kernel will not kill the container if it runs out of memory. Instead, it will return an out-of-memory (OOM) error. This flag should only be used in combination with the `--memory` option to avoid potential system-wide memory issues.

CPU Usage Reservations and Limits

We are going to see the available options to manage CPU usage for a container by following some practical examples.

Specifying how much of the available CPU resources a container can use:

Use `--cpus="1.5"` to let a container use up to one and a half CPUs. For example:

```
docker run -d --cpus="1.5" nginx
```

Specifying the CPU CFS scheduler period and quota for a container:

CFS (Completely Fair Scheduler) is a process scheduler that selects which process to run next in the Linux kernel. It is the default scheduler for Linux and is used on all Linux systems. The CFS scheduler period is the amount of time in microseconds that a container can use the CPU before it is throttled. The default value is 100000 microseconds (100 milliseconds).

While most users stick to `--cpus`, you can also use `--cpu-period` and `--cpu-quota`.

```
docker run -d --cpu-quota=50000 --cpu-period=100000 nginx
```

The CPU quote refers to the maximum allowed CPU time in microseconds during the period specified by `--cpu-period`.

If you set `--cpu-quota=50000` and `--cpu-period=100000`, the container can get up to 50,000 microseconds (or 50 milliseconds) of CPU time every 100,000 microseconds (or 100 milliseconds). This equates to 50% of a single CPU core.

Limiting the specific CPUs or cores a container can use:

Decide which CPUs your container runs on. For instance, use 0-3 for the first four CPUs or 1,3 for the second and fourth CPUs. For example:

```
docker run -d --cpuset-cpus="1,3" nginx
```

Setting the CPU shares for a container:

Alter the `--cpu-shares` value (default is 1024) to prioritize CPU usage between containers. This doesn't guarantee fixed CPU amounts but prioritizes access. For example:

```
docker run -d --cpu-shares="2048" nginx
```

If there's competition for CPU, this container gets a higher preference because of the "2048" value set for `--cpu-shares` and because the default value is 1024. This value is relative to the CPU shares of other containers. If you want to give less priority to a container, you can set the `--cpu-shares` value to a lower number.

```
docker run -d --cpu-shares="512" nginx
```

Docker Images

What is an Image?

Without images, we cannot run containers. Images are the building blocks of containers.

Imagine you're in the kitchen, and you want to make a dish. You'd refer to a recipe book, right? In the world of Docker, an image is like that recipe book. It has all the instructions (the "recipe") for a specific software or application (the "dish"). When you want to execute the software, you "cook" or "prepare" it using the recipe, much like you would spin up a container from a Docker image.

But where do you get these recipe books? Well, there's a grand library (the Docker registry) where all these recipe books (Docker images) are stored. Some of these recipe books are publicly available for anyone to use (public registry like Docker Hub), while some are private collections only accessible to certain individuals or groups (private registry).

Now, in this grand library, you can "borrow" a recipe book (pulling an image), add your own twist to the recipe and then "return" the updated version for others to use (pushing an image), or simply keep your special recipe to yourself. And just like in a real library where books might undergo revisions or updates, in the Docker world, images too can be tagged, updated, or even deleted.

Images are Layers

Imagine a Docker image as a multi-layered cake, where each layer represents a distinct piece of the application or its environment. These layers come together to form the final product, but they remain independent and can be reused across different cakes (or images).

In practice, each layer corresponds to a change in the filesystem. For example, if you install a package, the layer will contain the filesystem changes made during the package installation. Similarly, if you remove a file, the layer will capture the filesystem changes made by the file removal.

Every Docker image starts with a base layer, typically an operating system like Ubuntu or Alpine. This foundational layer serves as the foundation for everything else. As we add instructions in our Dockerfile, such as installing software packages or copying files, new layers are created. Each layer records the changes or additions made by a specific instruction.

> **i** Dockerfile is a text file that contains all the commands a user could call on the command line to assemble an image.

An important characteristic of these layers is that once created, they are immutable, meaning they cannot be altered. However, when changes occur, Docker simply adds a new layer on top, preserving the integrity and history of the previous layers.

When you run a container, Docker creates a new layer on top of the image layers. This layer is known as the container layer. The container layer is temporary, meaning it will be deleted when the container is deleted.

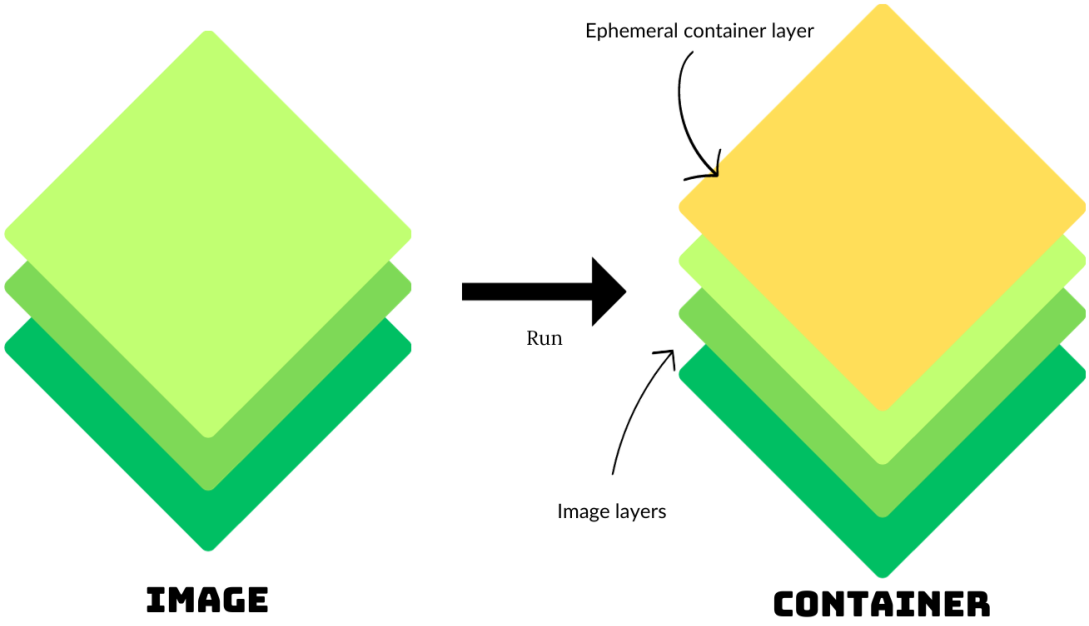


Image layers

Docker is an intelligent tool that optimizes the build process and storage usage when creating images. It achieves this by checking if it has previously encountered a layer (a specific instruction and its output) during image building. If it has, Docker reuses the existing layer instead of recreating it. This not only speeds up the build process but also saves storage space.

Whenever a particular layer changes, all subsequent layers are rebuilt to ensure that the final image remains up-to-date with the instructions in the Dockerfile.

Docker utilizes a special file system called UnionFS[49] to store and manage images. UnionFS combines multiple folders into a single view, allowing Docker to layer different images together. One of the key features of this system is the Copy-on-Write (CoW) approach. Rather than creating a new copy of data for each container, Docker enables multiple containers to share the same data. Only

[49] https://unionfs.filesystems.org/

when a container needs to modify the data does Docker create a separate copy exclusively for that container. This approach saves space and improves performance. When it comes to deleting files, Docker doesn't actually remove them from the base layer. Instead, it adds a special "whiteout" file in the top layer, instructing Docker to ignore the original file while it still exists in the base layer.

Images, Intermediate Images & Dangling Images

If you type `docker images`, you will see that you have two images (or more if you have already used Docker before):

- `hello-world`: This is the image that we pulled and ran in the previous section.
- `mariadb`: This is the image that we pulled and ran in the previous section.

```
docker images
# or docker image ls
```

You should see something like this:

```
REPOSITORY  TAG     IMAGE ID        CREATED     SIZE
<none>      none    57ce8f25dd63    2 days ago  229.7 MB
scratch     latest  f02aa3980a99    2 days ago  0 B
xenial      latest  7a409243b212    2 days ago  229.7 MB
<none>      none    149b13361203    2 days ago  12.96 MB
```

Images called `<none>` are intermediate images. They are created during the build process. They are not tagged and are not used by any container. They are just intermediate images that are used to build the final image. To list all of them, type:

```
docker images --filter "dangling=true"
```

Other filters may be used:

```
- label=<key> or label=<key>=<value>
- before=(<image-name>[:tag]|<image-id>|<image@digest>)
- since=(<image-name>[:tag]|<image-id>|<image@digest>)
```

An image has a tag (e.g.: `latest`), an id (e.g.: `92495405fc36`), a creation date (e.g.: `12 days ago`), and a size (e.g.: `356 MB`).

You can get more information about an image by using the `docker image inspect` command:

```
docker image inspect mariadb
# or docker image inspect mariadb:latest
```

You can also print a custom output where you choose to view the ID and the size of the image:

```
docker images --format "{{.ID}}: {{.Size}}"
```

To view the repository, type:

```
docker images --format "{{.ID}}: {{.Repository}}"
```

In some cases, we just need to get IDs:

```
docker images -q
```

If you want to list all of them, then type:

```
docker images -a
```

or

```
docker images --all
```

In this list, you will see all of the images, even the intermediate ones.

```
REPOSITORY  TAG     IMAGE ID        SIZE
my_app      latest  7f49abaf7a69    1.093 MB
<none>      <none>  afe4509e17bc    225.6 MB
```

All of the `<none>:<none>` are intermediate images. These images will grow with the numbers of images you download.

As you may know, each docker image is composed of different layers with a parent-child hierarchical relationship. These intermediate layers are a result of caching build operations, which decrease disk usage and speed up builds. Every build step is cached; that's why you may experience some disk space problems after using Docker for a while.

All docker layers are stored in `/var/lib/docker/overlay2` by default.

```
sudo ls -l /var/lib/docker/overlay2
```

You should see a list of directories with long identifiers. Each directory is a layer.

Searching for Images

If you type:

```
docker search busybox
```

You will get a list of images called Ubuntu that people shared publicly in the Docker Hub (hub.docker.com[50]).

```
NAME                                DESCRIPTION                                     \
STARS     OFFICIAL   AUTOMATED
busybox                             Busybox base image.                             \
3122      [OK]
rancher/busybox                                                                     \
0
openebs/busybox-client                                                              \
1
antrea/busybox                                                                      \
1
hugegraph/busybox                   test image                                      \
2
privatebin/chown                     Docker image providing busybox' chown, stat…   \
1
yauritux/busybox-curl               Busybox with CURL                               \
23
radial/busyboxplus                  Full-chain, Internet enabled, busybox made f…   \
54                  [OK]
vukomir/busybox                     busybox and curl                                \
1
arm64v8/busybox                     Busybox base image.                             \
5
odise/busybox-curl                                                                  \
4                   [OK]
amd64/busybox                       Busybox base image.                             \
1
busybox42/zimbra-docker-centos      A Zimbra Docker image, based in ZCS 8.8.9 an…   \
2                   [OK]
joeshaw/busybox-nonroot             Busybox container with non-root user nobody     \
2
p7ppc64/busybox                     Busybox base image for ppc64.                   \
2
ppc64le/busybox                     Busybox base image.                             \
1
s390x/busybox                       Busybox base image.                             \
3
```

[50] http://hub.docker.com/

```
busybox42/alpine-pod                                                    \
0
arm32v7/busybox                     Busybox base image.                 \
10
i386/busybox                        Busybox base image.                 \
3
prom/busybox                        Prometheus Busybox Docker base images \
2                   [OK]
spotify/busybox                     Spotify fork of https://hub.docker.com/_/bus… \
1
arm32v5/busybox                     Busybox base image.                 \
0
busybox42/nginx_php-docker-centos   This is a nginx/php-fpm server running on Ce… \
1                   [OK]
concourse/busyboxplus                                                   \
0
```

As you can notice, there are images having automated builds, while others don't have this feature activated.

An automated build allows your image to be up-to-date with changes on your private and public git (e.g.: Github, Bitbucket) code.

Notice that if you type the `search` command, you will get only a few images, and if you want more, you could use the `--limit` option:

```
docker search --limit 100 busybox
```

To refine your search, you can filter it using the `--filter` option.

Let's search for the best Mongodb images according to the community (images with more than 5 stars):

```
docker search --filter=stars=5  mongo
```

Images could be either official or unofficial. Just like any Open Source project, Docker public images are made by anyone who may have access to the Dockerhub so consider double-checking images before using them in your production servers.

Official images could be filtered in this way:

```
docker search --filter=is-official=true  mongo
```

Pulling Images and the Latest Tag

If you want to pull `latest` tag of an image, you can simply type:

```
docker pull <image-name>
```

Or:

```
docker pull <image-name>:latest
```

In the following example, we will pull `ubuntu:latest` tag:

```
docker pull ubuntu
# or docker pull ubuntu:latest
```

The `latest` tag is the default tag, so if you don't specify a tag, the `latest` tag will be pulled. However, the `latest` tag is not always the latest version of the image.

Example: I can create two images:

- `image:v1`
- `image:v2`

If I pull `image` using `docker pull image`, I should get `image:latest` since I didn't specify a tag. However, even if `image:v2` is the latest version, I will get an error if I try to pull `image:latest` because I didn't explicitly tag the v2 image as `latest`.

In essence, if I want `image:v2` to be the default when pulling the image, I need to tag it as `image:latest`. Otherwise, simply pulling the image will fetch whichever version is tagged as latest. If no version is tagged as latest, the pull request for the image would result in an error.

Additionally, I can also tag `image:v1` as `image:latest`. In this case, when users pull `image`, they will get `image:v1` because it is tagged as `latest` and not `image:v2`.

As a conclusion, the `latest` tag is not always the latest version of the image, don't rely on it.

Removing Images

To remove an image, you can use the `rmi` command:

```
docker rmi <image-name>
```

Or

```
docker image rm <image-name>
```

If you want to remove all images (except the ones that are used by running containers), you can type:

```
docker rmi $(docker images -q)
```

We can also safely remove only dangling images by typing:

```
docker rmi $(docker images -f "dangling=true" -q)
```

The Dockerfile and its Instructions

The Dockerfile is a text file that contains various instructive commands and arguments. These commands and arguments describe how your base image will look at the end of the build process.

While it is possible to run an image directly without building it, such as a public or private image pulled from Docker Hub or another registry, creating a Dockerfile for your application is a good starting point if you want to customize and build your own images. It also allows you to organize your deployments and distribute the same image to different teams.

The first rule is that a Dockerfile (should) always start with the FROM instruction.

Creating a Dockerfile follows a simple and explicit syntax that must be followed exactly. For example, if you need to execute commands beyond what the Docker instructions permit, you can use the `RUN` command or use `CMD` and `ENTRYPOINT` to run scripts, executables, binaries, or any other command.

We will examine the different instructions and their variations in detail. By using these instructions, you should be able to create a Dockerfile.

This is a basic example of a container that launches an Apache web server with a simple "Hello, World!" message.

Start by creating the Dockerfile:

```
cat <<EOF > Dockerfile
FROM ubuntu:latest
MAINTAINER Aymen EL Amri - @eon01
RUN apt-get update && apt-get install -y apache2 && apt-get clean
RUN echo "Hello, World!" > /var/www/index.html
EXPOSE 80
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
EOF
```

Build the Docker image:

```
docker build -t my-apache2 .
```

Run the container:

```
docker run -d -p 8000:80 my-apache2
```

Test the container:

```
curl localhost:8000
```

Later, we will explore more examples, including creating custom images but for now, let's focus on the instructions that we can use in a Dockerfile to enrich and customize our images.

In the following examples, you will have to make some abstractions since there are many concepts that we haven't covered yet. We will cover them in the next sections.

FROM

In a Dockerfile, the foundational step begins with the `FROM` instruction, specifying which base image you want to use.

If you're aiming to create a multi-stage build[51], where one image is used as a base for another, you can actually incorporate multiple `FROM` instructions within a single Dockerfile.

```
FROM <image>:<tag>
```

Example:

```
FROM ubuntu:14.04
```

Should you omit the tag, Docker will default to fetching the image tagged as `latest`.

MAINTAINER

The `MAINTAINER` line in a Dockerfile isn't a functional instruction in the same sense as others. Instead, it provides metadata about the individual or organization responsible for the image. It can include the name, email, or even a website.

```
MAINTAINER <name>
```

Example:

```
MAINTAINER Aymen EL Amri - @eon01
```

RUN

In a Dockerfile, the `RUN` instruction allows you to execute commands, just as you would on the command line. There are two syntax forms for the `RUN` instruction: the **shell form** and the **exec form**.

Shell Form:

[51] https://docs.docker.com/build/building/multi-stage/

```
RUN <command>
```

For Linux-based containers, this command runs with the `/bin/sh -c` shell, and for Windows, it uses the `cmd /S /C` shell.

Example:

```
RUN ls -l
```

When building the Docker image, you'll witness the output of this command:

```
Step 4 : RUN ls -l
 ---> Running in b3e87d26c09a
total 64
... [trimmed output]
drwxr-xr-x  13 root root 4096 Oct 13 21:13 var
drwxr-xr-x   2 root root 4096 Oct 13 21:13 sbin
drwxr-xr-x   2 root root 4096 Oct 13 21:13 media
... [trimmed output]
```

Exec Form:

```
RUN ["<executable>", "<param>", "<param1>", ... ,"<paramN>"]
```

In this form, the command and its parameters are specified as a JSON array.

Example:

```
RUN ["/bin/sh", "-c", "ls", "-l"]
```

The shell form is more commonly used due to its simplicity, while the exec form is more versatile and is often chosen for more complex scenarios.

CMD

The `CMD` instruction in a Dockerfile specifies what command you want to run when the container starts. Unlike the `RUN` instruction which executes commands during the image build process, `CMD` sets the command for the running container.

There are three forms to use the `CMD` instruction:

Shell Form:

```
CMD command param1 param2 ... paramN
```

In this form, the command runs in a shell, which by default is `/bin/sh -c` on Linux.

Example:

```
CMD echo "Hello, World!"
```

Exec Form:

```
CMD ["executable", "param1", "param2", ..., "paramN"]
```

This form is recommended for CMD because it avoids shell string munging, and allows for the container to run an executable directly.

Example:

```
CMD ["ls", "-l"]
```

As Default Parameters to ENTRYPOINT:

```
CMD ["param1", "param2", ... , "paramN"]
```

In this case, `CMD` provides default parameters which can be overridden by the command line arguments when the container runs. This form is often used in combination with the `ENTRYPOINT` instruction.

Example:

If your Dockerfile has:

```
ENTRYPOINT ["echo"]
CMD ["Hello, World!"]
```

Running the container without any arguments will print "Hello, World!". But if you run the container with a different argument, like `docker run <image> Hi`, it will print "Hi".

LABEL

The `LABEL` instruction allows you to attach metadata to your Docker image. This metadata is represented as key-value pairs. This can be beneficial for storing additional information about the image, such as version numbers, contact details, or licensing information.

Syntax:

```
LABEL key1=value1 key2=value2 ... keyN=valueN
```

Example:

```
LABEL version="1.0" maintainer="John Doe <johndoe@example.com>"
```

It's important to note that not only Docker images can utilize labels. In the Docker ecosystem, labels can be attached to:

- **Docker Containers**: Running instances of a Docker image
- **Docker Daemons**: The background service running on the host that manages building, running, and managing Docker containers
- **Docker Volumes**: The storage volumes associated with containers
- **Docker Networks**: Networks that connect containers, aiding in communication
- **Docker Swarm Nodes**: Individual machines, VMs, or physical computers that are members of a Swarm
- **Docker Swarm Services**: A Swarm-specific term representing a group of tasks (or containers) to ensure designated workloads run in a specified state

Using labels, you can organize, manage, and track resources better in a Docker environment.

EXPOSE

When running an application or service inside a Docker container, there's often a need for the container to communicate with the outside world. For this, we expose and publish ports.

Consider you have a PHP/MySQL web application on a host. You've created two containers:

- A MySQL container
- An Apache (webserver) container

Currently, neither the database (DB) server can communicate with the webserver, nor can the webserver query the database. Moreover, both servers are not accessible from outside the host.

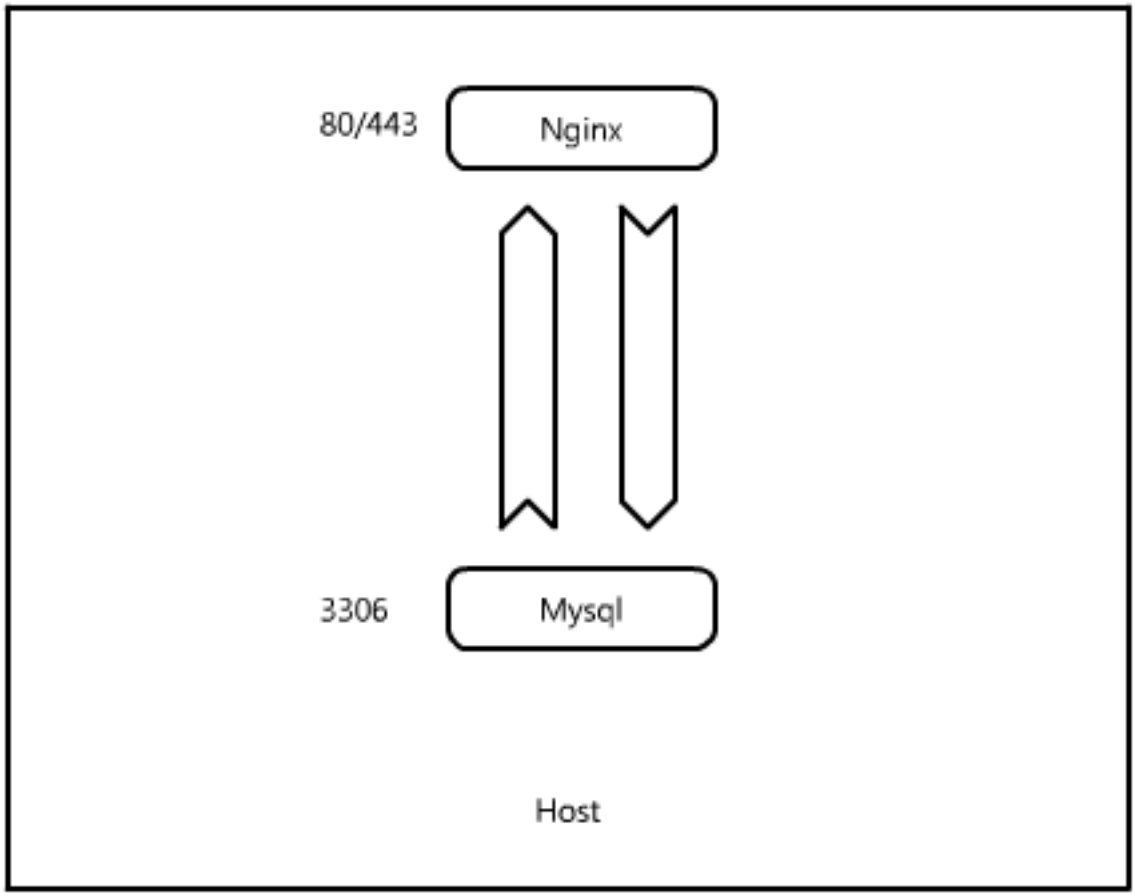
To allow communication, you first need to expose the necessary ports in each container's Dockerfile:

MySQL Dockerfile:

```
EXPOSE 3306
```

Apache Dockerfile:

```
EXPOSE 80 443
```



Inter-communication between containers

Although the ports are exposed, only the webserver port should be made publicly accessible. The database container should remain accessible solely from the webserver.

The `EXPOSE` instruction lets the web server communicate with the DB server when they're on the same Docker network.

Exposing ports in a Dockerfile doesn't make them accessible from the host. To do that, you need to map the exposed ports to the host's ports using the Docker CLI. You can use the `-p` or `-P` flags:

- `-p`: Maps a container port to a host port
- `-P`: Maps all exposed ports to random ports on the host

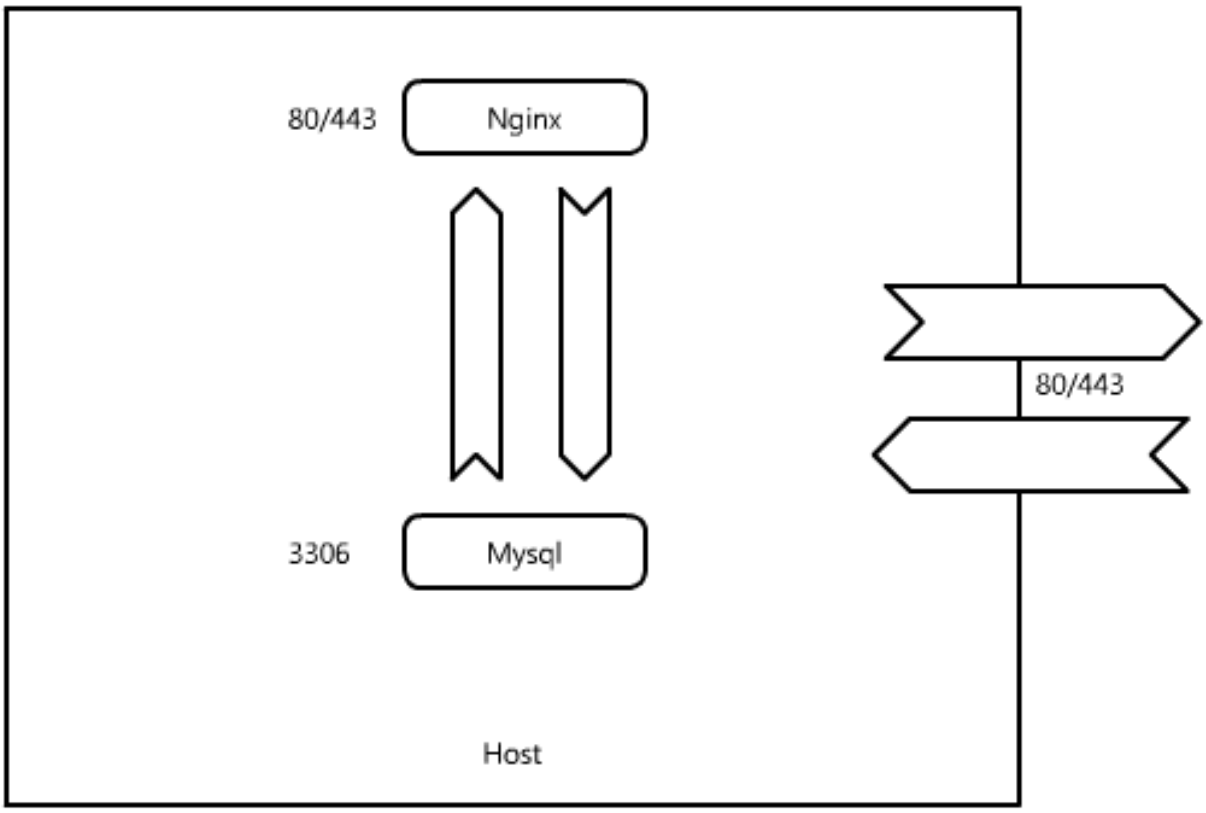
For better control, it's recommended to use `-p` to map ports individually.

Example :

```
docker run -it -p 80:80 nginx
```

If needed, you can map the container port to a different host port:

```
docker run -it -p 8000:80 nginx
```



Mapping hosts to containers

To expose a range of ports, use the following format:

```
EXPOSE 3000-4000
```

ENV

The `ENV` instruction in a Dockerfile is used to set environment variables. It's akin to the `export` command in Linux. It uses a `<key>/<value>` pair format. You can set environment variables in two ways:

One variable per line:

```
ENV variable1 value1
ENV variable2 value2
```

Multiple variables in a single line:

```
ENV variable1="value1" variable2="value2"
```

If you're familiar with building Docker images, you might have encountered this line that tells the container usign Debian-based distributions to run in non-interactive mode (i.e., no prompts):

```
ENV DEBIAN_FRONTEND noninteractive
```

However, setting `DEBIAN_FRONTEND` with the `ENV` instruction is not recommended because this environment variable will persist beyond the build, which might not be desirable in many cases.

A better approach is to use the `ARG` instruction:

```
ARG DEBIAN_FRONTEND=noninteractive
```

By using `ARG`, the environment variable won't persist after the build process.

ARG

The `ARG` instruction in a Dockerfile allows you to define variables that can be used during the build process. These variables don't persist after the build, making them suitable for temporary configurations or values you don't want to embed in the final image.

The general syntax for `ARG` is:

```
ARG <argument_name>[=<default_value>]
```

You can declare the argument without a default value:

```
ARG time
```

Or, you can specify a default value directly in the Dockerfile:

```
ARG time=3s
```

Variables declared with `ARG` can be overridden during the build process. You can do this using the `docker build` command and the `--build-arg` flag. We will dive into this in more detail in a subsequent section.

WORKDIR

The `WORKDIR` instruction is used to set the working directory for any subsequent commands in the Dockerfile. This becomes especially handy to streamline and simplify the paths used in subsequent instructions like `RUN`, `CMD`, `ENTRYPOINT`, `COPY`, and `ADD`.

Here's the general way to use the `WORKDIR` instruction:

```
WORKDIR <path>
```

Consider a scenario where you want to copy an `index.html` file into the `/var/www` directory. Instead of specifying the full path in the `ADD` instruction like:

```
ADD index.html /var/www
```

You can set the working directory first using `WORKDIR` and then simplify the `ADD` command:

```
WORKDIR /var/www
ADD index.html .
```

If the directory specified in the `WORKDIR` instruction doesn't already exist, Docker will create it for you. This ensures that subsequent instructions always have a valid directory to work within.

ADD

The `ADD` instruction is primarily used to copy files or directories from the host system to the container's filesystem. It offers some added functionality beyond the capabilities of the `COPY` instruction, such as remote file retrieval and on-the-fly tar extraction.

The `ADD` instruction can be written in two forms:

The most common form:

```
ADD <src>... <dest>
```

If your path contains spaces, use the following format:

```
ADD ["<src>",... "<dest>"]
```

The `ADD` instruction supports wildcard characters like `*` and `?`:

- `*` matches all files in a directory.

```
# Copy all files from /var/www/ to /var/www/
ADD /var/www/* /var/www/
```

- `?` matches any single character.

```
# Copy files named index.html and index.htm from /var/www/ to /var/www/
ADD /var/www/index.htm? /var/www/
```

You can also use `ADD` to directly fetch files from remote URLs:

```
ADD https://github.com/twbs/bootstrap/raw/main/dist/js/bootstrap.js /var/www/static/\
js/
```

The above command downloads the specified file and copies it into the container's filesystem.

`ADD` is intuitive when it comes to compressed files. If a recognized archive format (e.g., gzip, bzip2, or xz) is provided as a source, Docker will automatically unpack it into the specified directory:

```
ADD source_data.tar.gz /destination_directory/
```

For the above command, the `source_data.tar.gz` archive is extracted to `/destination_directory/` within the container.

COPY

The `COPY` instruction is an essential tool in Dockerfiles, allowing users to copy files or directories from a source on the host to the container's filesystem. While similar to the `ADD` instruction, `COPY` is simpler and doesn't handle archive extraction or URL fetching.

Standard format:

```
COPY <src>... <dest>
```

Format for paths containing spaces:

```
COPY ["<src>",... "<dest>"]
```

For instance:

```
COPY ["/home/eon01/Painless Docker.html", "/var/www/index.html"]
```

- The * wildcard is useful to match all files in a directory:

```
COPY /var/www/* /var/www/
```

- The ? wildcard matches any single character. Given a set of files named `chapter1`, `chapter2`, ..., `chapter9`, you can copy them all with:

```
COPY /home/eon01/painlessdocker/chapter? /var/www
```

If the `WORKDIR` instruction sets the working directory to `/var/www/`, you can shorten your paths:

Instead of:

```
COPY index.html /var/www/
```

You can use:

```
WORKDIR /var/www/
COPY index.html .
```

However, take note that relative paths that move up a directory (like `../index.html`) are not supported.

As a reminder:

1. `COPY` does not handle archive extraction.
2. `COPY` cannot fetch files from URLs.

ENTRYPOINT

The `ENTRYPOINT` instruction in Docker answers the question: "What should happen when a container starts?" It specifies a command that will always be executed when the container is run.

Suppose you want to launch a Python or Node.js server. You could use commands like:

```
python -m SimpleHTTPServer
```

or

```
node app.js
```

Without a defined `ENTRYPOINT`, the container might start and then immediately stop because it lacks an ongoing task. Essentially, a Docker container without a running process will exit quickly.

To ensure that the container stays up and running, you can use the `ENTRYPOINT` instruction to specify a command that will run when the container starts. There are two forms of the `ENTRYPOINT` instruction:

Exec form (recommended):

```
ENTRYPOINT ["<executable>", "<param1>", "<param2>", ... "<paramN>"]
```

Shell form:

```
ENTRYPOINT <command> <param1> <param2> ... <paramN>
```

Example:

```
ENTRYPOINT ["node", "app.js"]
```

Consider a simple Python script that prints "Hello World":

```
print("Hello World")
```

You'd want this script to run when the container starts. Using the instructions `FROM`, `COPY`, and `ENTRYPOINT`, you can craft the following Dockerfile:

```
FROM python:3.7
COPY app.py .
ENTRYPOINT ["python", "app.py"]
```

This Dockerfile communicates:

- The base image `python:3.7` is sourced from Docker Hub
- The script "app.py" is copied into the container
- When the container starts, it will execute `python app.py`

To get this container running build the Docker image!

```
docker build -t <image-name> <path-to-dockerfile>
```

Then, run the container:

```
docker run <image-name>
```

Crucially, notince that the `ENTRYPOINT` is executed only during the container's runtime, not during its build.

VOLUME

The `VOLUME` instruction designates a specified directory in the container as a mount point and potentially mounts an external volume to it. Another Docker container can use any external volume mounted using this instruction.

The syntax for the `VOLUME` instruction comes in several forms:

Single Directory:

```
VOLUME ["/path/to/directory"]
```

Multiple Directories:

```
VOLUME /path/to/directory1 /path/to/directory2 ... /path/to/directoryN
```

JSON Array (similar to the first form):

```
VOLUME ["/path/to/directory1", "/path/to/directory2", ... "/path/to/directoryN"]
```

Volumes have several use cases and benefits:

- **Persistence**: Docker containers are inherently ephemeral. Volumes ensure data persistence across container restarts, stops, or terminations
- **Host Access**: Containers, by default, don't expose their data to the host. Volumes bridge this gap, permitting the host system to access container data
- **Inter-container Communication**: Containers, even if they reside on the same host, don't naturally share data. Docker volumes facilitate data sharing between containers, allowing multiple containers to access shared files.

It's crucial to establish the permissions or ownership of a volume before introducing the `VOLUME` instruction in a Dockerfile.

The following sequence is incorrect:

```
VOLUME /app
ADD app.py /app/
RUN chown -R foo:foo /app
```

Instead, permissions should be set as demonstrated in this corrected Dockerfile:

```
RUN mkdir /app
ADD app.py /app/
RUN chown -R foo:foo /app
VOLUME /app
```

This ensures that the volume ownership is correctly set before designating `/app` as a volume.

USER

When executing commands inside a container using the `RUN` instruction, there may be instances where you want to run the command as a different user instead of the default user (which is `root`). In such cases, the `USER` instruction comes into play.

The `USER` instruction is used in the following manner:

```
USER <username>
```

By default, Docker operates as the `root` user and possesses complete access to the host system. Therefore, utilizing the `USER` instruction can also be seen as a security measure.

For security reasons, it's a best practice to avoid running containers as the `root` user, as this could introduce potential vulnerabilities if the container is compromised. Instead, you should create a new user and run the container as that user.

For example:

```
USER my_user
```

However, in situations where you need to execute a specific command as `root`, you can temporarily switch users, run the command, and then revert back:

```
USER root
RUN <command to be run as root>
USER my_user
```

The `USER` instruction influences subsequent `RUN`, `CMD`, and `ENTRYPOINT` instructions. This means any command executed by these instructions will be associated with the specified user

ONBUILD

To facilitate the automatic building of applications, Docker introduced the `ONBUILD` instruction. This special instruction acts as a trigger that only executes when the current image is used as the base for another image.

The `ONBUILD` instruction is written in the following format:

```
ONBUILD <Docker Instruction>
```

The trigger activates during the downstream build, behaving as if it was placed immediately following the `FROM` instruction in the new image's Dockerfile.

The `ONBUILD` instruction has been available in Docker since version 0.8.

To clarify its functionality, let's define a child image:

Consider an image A that has an `ONBUILD` instruction. If another image B is built using image A as its base (to add more instructions and layers), then image B becomes a child image of image A.

Here's the Dockerfile for image A:

```
FROM ubuntu:18.04
ONBUILD RUN echo "This will be executed automatically in the child image."
```

Now, the Dockerfile for image B:

```
FROM imageA
```

When you build image A, the `ONBUILD` instruction remains dormant - the `RUN echo "..."` command doesn't execute. However, during the construction of image B, this command is invoked right after the `FROM` instruction.

Here's a sample output from building image B, demonstrating the automatic execution of the `ONBUILD` trigger:

```
Uploading context  4.51 kB
Uploading context
Step 0 : FROM imageA

# Executing 1 build triggers
Trigger 0, onbuild-0 : RUN echo "This will be executed automatically in the child im\
age."

 ---> Running in acefe7b39c5
This will be executed automatically in the child image.
```

STOPSIGNAL

The `STOPSIGNAL` instruction allows you to specify the system call signal that will be dispatched to the container to initiate its termination.

The `STOPSIGNAL` instruction is written in the following format:

```
STOPSIGNAL <signal>
```

The provided signal can either be a valid unsigned number, such as 9, or one of the established signal names like SIGTERM, SIGKILL, SIGINT, etc.

By default, Docker utilizes `SIGTERM`. This is functionally analogous to executing the `kill <pid>` command.

To illustrate, consider the following example where we set the stop signal to `SIGINT` (the same signal dispatched when pressing Ctrl-C):

```
FROM ubuntu:18.04
STOPSIGNAL SIGINT
```

HEALTHCHECK

Introduced in Docker version 1.12, the `HEALTHCHECK` instruction provides the capability to monitor the health status of containers.

The instruction comes in two primary forms:

To ascertain the health of a container by executing a specified command within it:

```
HEALTHCHECK [OPTIONS] CMD <command>
```

To disable any existing health checks (this includes any checks inherited from a base image):

```
HEALTHCHECK NONE
```

The `HEALTHCHECK` instruction offers three configurable options before the `CMD` command:

- `--interval=<interval duration>`:

This specifies the frequency at which the health check will be conducted. By default, this interval is set to 30 seconds.

```
--interval=1m
```

- `--timeout=<timeout duration>`:

If the health check does not complete within this duration, it is deemed to have failed. The default timeout is 30 seconds.

```
--timeout=3s
```

- `--retries=N`:

Defines the number of consecutive failures that are tolerated before considering the container as unhealthy. By default, Docker allows 3 consecutive failures.

```
--retries=3
```

To demonstrate, consider the following example where a health check is initiated every 1 minute. The check has a 3-second timeout, and by not explicitly defining the retry count, it defaults to 3. If the health check command fails 3 times in a row, the container is marked as unhealthy:

```
HEALTHCHECK --interval=1m --timeout=3s CMD curl -f http://localhost/ || exit 1
```

SHELL

In Docker, the default shell that interprets and executes commands is `/bin/sh -c`. As a result, when the `CMD ls -l` instruction is given, it is effectively executed within the container as:

```
/bin/sh -c ls -l
```

For Windows containers, the default shell is:

```
cmd /S /C
```

To customize the shell, you can use the `SHELL` instruction in the Dockerfile. However, this instruction must be specified in JSON array format:

```
SHELL ["<executable>", "<parameters>"]
```

This customization becomes particularly handy for Windows users who might want to toggle between `cmd` and `powershell`.

For instance:

```
SHELL ["powershell", "-command"]
SHELL ["cmd", "/S", "/C"]
```

In Unix-based systems, other shells such as `bash`, `zsh`, `csh`, and `tcsh` are available. For example, to switch to `bash`:

```
SHELL ["/bin/bash", "-c"]
```

ENTRYPOINT VS CMD

The CMD and ENTRYPOINT instructions in Docker serve to specify a command that will be executed when a container is started. While they share similarities, their use cases and combinations offer flexibility.

Recalling our discussion on the CMD instruction, when used in the following format:

```
CMD ["<param1>", "<param2>", ..., "<paramN>"]
```

it provides default parameters for the ENTRYPOINT instruction.

Consider this simple Dockerfile:

```
FROM python:2.7
COPY app.py .
ENTRYPOINT ["python", "app.py"]
```

One might think that an equivalent Dockerfile could be:

```
FROM python:2.7
COPY app.py .
ENTRYPOINT ["python"]
CMD ["app.py"]
```

However, when you attempt to run the container using the above Dockerfile, it will produce an error. To fix it, the ENTRYPOINT should have the full path to the `python` executable:

```
FROM python:2.7
COPY app.py .
ENTRYPOINT ["/usr/bin/python"]
CMD ["app.py"]
```

To summarize:

```
ENTRYPOINT ["python", "app.py"]
```

can also be represented as:

```
ENTRYPOINT ["/usr/bin/python"]
CMD ["app.py"]
```

Both CMD and ENTRYPOINT can be used independently or in combination, but it's important to include at least one of them. Failing to use either CMD or ENTRYPOINT will result in the container's execution failing.

You can find the same table in the official Docker documentation[52], and this is the best way to understand all of the possibilities:

	No ENTRYPOINT	**ENTRYPOINT exec_entry p1_entry**	**ENTRYPOINT [“exec_entry”, “p1_entry”]**
No CMD	*error, not allowed*	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD [“exec_cmd”, “p1_cmd”]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd
CMD [“p1_cmd”, “p2_cmd”]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry p1_cmd p2_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

The Base Image

If a Dockerfile has a single instruction, it would most likely be the `FROM` instruction. This instruction specifies the base image from which a new image is derived. Every valid Dockerfile must start with (or at least include) a `FROM` instruction.

[52] https://docs.docker.com/engine/reference/builder/#understand-how-cmd-and-entrypoint-interact

```
FROM <BASE_IMAGE>
```

Docker requires a base image to run. Without an image, there can be no container. The base image serves as the foundation upon which you add additional layers to create a new image that contains your application. Each layer contributes to the overall image, and the `FROM <BASE_IMAGE>` instruction is essential for creating a child image.

Here is an example of a Dockerfile that utilizes the `FROM` instruction exclusively:

```
cat <<EOF > Dockerfile
FROM tutum/hello-world
EOF
```

Build and run the container:

```
docker build -t my-hello-world .
docker run -p 8001:80 my-hello-world
```

Test the container:

```
curl localhost:8001
# or visit http://localhost:8001
```

Extending the Base Image

Let's consider this example where we use the `FROM` instruction with `golang:1.21.3-alpine3.17` as the base image:

```
# Use the specified image as the base
FROM golang:1.21.3-alpine3.17
```

We also have our Go application in the same directory as the Dockerfile:

```
cat <<EOF > main.go
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
EOF
```

We want to copy the Go application into the container and build it. To do this, we can use the `COPY` instruction:

```
# Set the working directory inside the container
WORKDIR /app
# Copy the local package files to the container's workspace
ADD . /app
```

Then we need to build the application:

```
# Build the Go app
RUN go mod init my-golang-app
RUN go build -o main .
```

Finally, we can run the application:

```
# Run the binary program produced by `go build`
CMD ["/app/main"]
```

The final Dockerfile can be created using the following command:

```
cat << 'EOF' > Dockerfile
# Use the specified image as the base
FROM golang:1.21.3-alpine3.17
# Set the working directory inside the container
WORKDIR /app
# Copy the local package files to the container's workspace
ADD . /app
# Build the Go app
RUN go mod init my-golang-app
RUN go build -o main .
# Run the binary program produced by `go build`
CMD ["/app/main"]
EOF
```

Build and run the container:

```
# Build using the image name and tag
docker build -t my-golang-app:v1 .
# Run the container
docker run my-golang-app:v1
```

You should see the following output:

```
Hello, World!
```

Congratulations! You have successfully created a Docker image that builds and runs a Go application.

Exploring Images' Layers

Using dive[53], we can visualize the layers of an image:

```
# Install dive (adapt the command for your OS)
export DIVE_VERSION=$(curl -sL "https://api.github.com/repos/wagoodman/dive/releases\
/latest" | grep '"tag_name":' | sed -E 's/.*"v([^"]+)".*/\1/')
curl -OL https://github.com/wagoodman/dive/releases/download/v${DIVE_VERSION}/dive_$\
{DIVE_VERSION}_linux_amd64.deb
sudo apt install ./dive_${DIVE_VERSION}_linux_amd64.deb
# Show the layers of the image
dive <IMAGE_NAME>
```

Let's take the example of tutum/hello-world that you can find on the Docker Hub website just by concatenating:

```
base_url="https://hub.docker.com/r"
image_name="tutum/hello-world"
echo "${base_url}/${image_name}"
```

The Dockerfile of this image[54] is the following:

[53] https://github.com/wagoodman/dive
[54] https://hub.docker.com/r/tutum/hello-world/Dockerfile

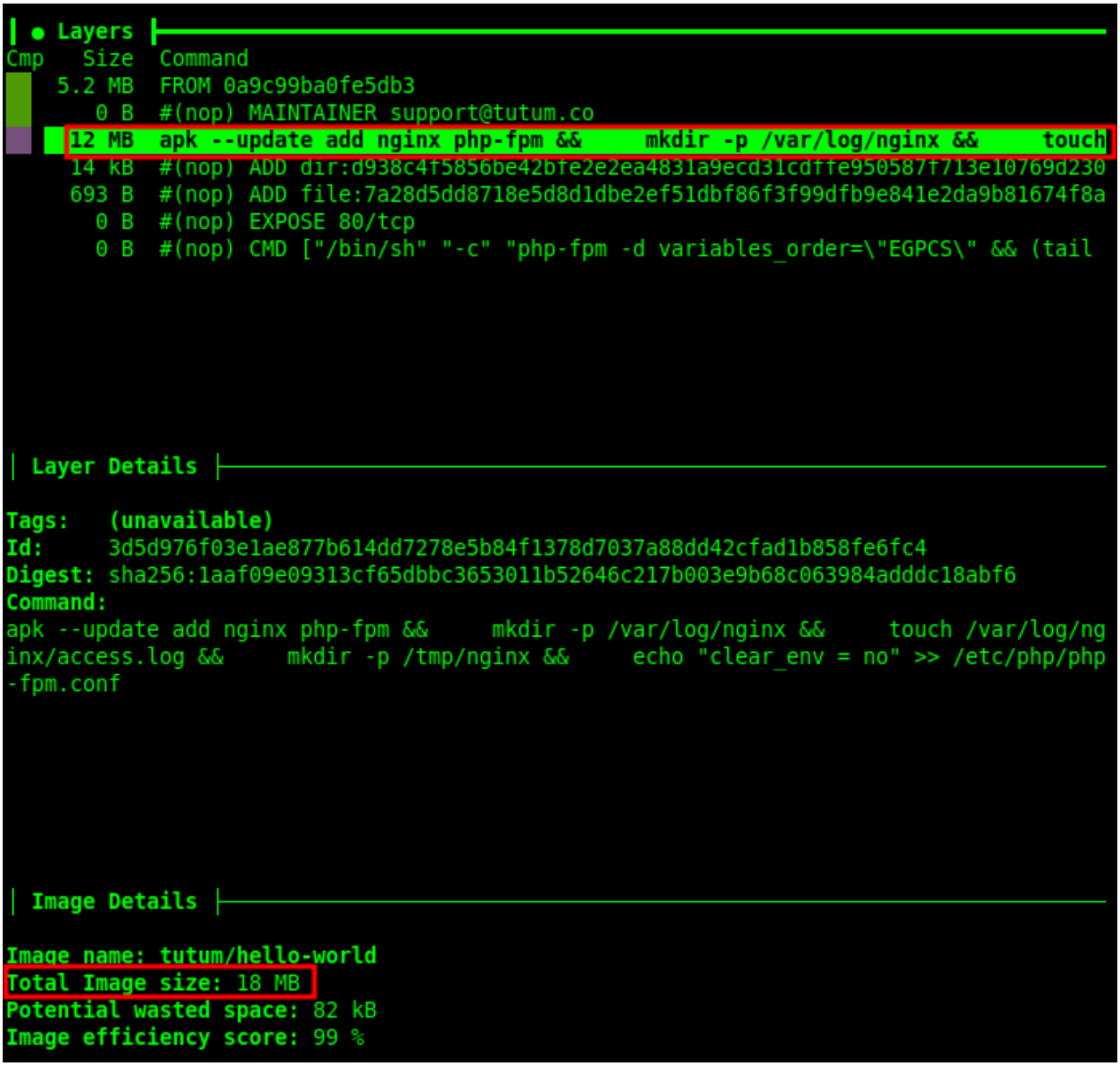
```
FROM alpine
MAINTAINER support@tutum.co
RUN apk --update add nginx php-fpm && \
    mkdir -p /var/log/nginx && \
    touch /var/log/nginx/access.log && \
    mkdir -p /tmp/nginx && \
    echo "clear_env = no" >> /etc/php/php-fpm.conf
ADD www /www
ADD nginx.conf /etc/nginx/
EXPOSE 80
CMD php-fpm -d variables_order="EGPCS" && (tail -F /var/log/nginx/access.log &) && e\
xec nginx -g "daemon off;"
```

To visualize the layers of this image, you can use `dive tutum/hello-world`.

The base image is Alpine[55], a lightweight Linux distribution based on musl[56] and Busybox[57]. This image is only 5 MB in size and consists of 5 layers.

This image, which is 18 MiB in size, has 7 unique layers. The largest layer in this image is 12 MB.

[55] https://hub.docker.com/_/alpine
[56] https://musl.libc.org/
[57] https://busybox.net/



Images layers

Let's take another image but this time, we are going to use the `history` command which is a Docker command that shows you the history of an image and its different layers.

```
1 docker pull nginx
2 docker history nginx
```

You can see more information with human-readable output about an image by using :

```
docker history --no-trunc -H nginx
```

You can also customize the display using the `--format` option to specify which fields you're interested in:

```
docker history --no-trunc -H --format "{{.CreatedBy}}" nginx
```

Building an Image Using a Dockerfile

We have learned about various instructions that can assist us in creating a Dockerfile. To have a complete image, we need to build it using the `docker build` command.

Example:

```
# Create a directory for the Dockerfile
mkdir -p nginx
cd nginx
echo "Hello, World!" > index.html
# Create the Dockerfile
cat <<EOF > Dockerfile
FROM nginx
COPY index.html /var/www/index.html
EOF
# Build the image
docker build .
```

The `.` specifies the build context, which is the path to a directory containing all the files and folders that should be sent to the Docker daemon in order to construct the Docker image. In our case, the build context is the current directory that contains the index.html file.

```
docker build -f /path/to/the/Dockerfile .
```

If your Dockerfile is not in the same directory where you are executing the build command, you can use the `f` option to specify the path to the Dockerfile.

```
docker build -f /path/to/the/Dockerfile/TheDockerfile /path/to/the/context/
```

Example:

If your Dockerfile is located in the "/tmp" directory and your configuration and code files are in the "/app" directory, the command should be:

```
docker build -f /tmp/Dockerfile /app
```

Creating Images out of Containers

Docker images are comprised of multiple layers, each representing a set of file changes or instructions. Image layers are instantiated to create containers, turning the static image into a runnable environment.

So the Dockerfile is the representation of the image, and the image is the representation of the container.

A running container when restarted (for a reason or another) will always start from the same state, the same image, and the same layers. Therefore, any state saved in the container will be lost.

However, using the `docker commit` command, you can save the state of a container as a new image. This new image will be based on the previous one, and it will contain all the changes that happened in the container.

To see this in action, let's create this Dockerfile:

```
cat <<EOF > Dockerfile
FROM alpine
RUN apk update && apk add nginx
EOF
```

Build and run the container in the background:

```
docker build -t my-nginx .
docker run -it -d --name my-nginx my-nginx
```

Execute a command inside the container:

```
docker exec -it my-nginx /bin/sh -c "echo 'Hello World' > /var/www/index.html"
```

We created a file called "index.html" in the "/var/www" directory. This file will be lost if we remove the container.

```
# remove the container
docker rm -f my-nginx
# start a new container
docker run -it -d --name my-nginx my-nginx
# check if the file exists
docker exec -it my-nginx ls /var/www/index.html > /dev/null 2>&1 && echo "File exist\
s" || echo "File does not exist"
```

Recreate the file:

```
docker exec -it my-nginx /bin/sh -c "echo 'Hello World' > /var/www/index.html"
```

We can save the state of the container as a new image using the `docker commit` command:

```
docker commit my-nginx my-nginx:v1
```

Now if you remove the container and start a new one, the file will still exist:

```
# remove the container
docker rm -f my-nginx
# start a new container using the new image
docker run -it -d --name my-nginx my-nginx:v1
# check if the file exists
docker exec -it my-nginx ls /var/www/index.html > /dev/null 2>&1 && echo "File exist\
s" || echo "File does not exist"
```

The file has the same content that we created in the previous container:

```
docker exec -it my-nginx cat /var/www/index.html
```

Migrating a VM to a Docker Image

The most common way of creating Docker images is by using a Dockerfile. However, you can also create an image from an existing container like we did in the previous section.

Another less common way is to create an image from an existing VM. This is useful if you have a VM that you want to migrate to a Docker image. It is not a best practice since you will end up with a large image, but it is possible. We are going to discover how through some examples.

Creating a Docker Image from an ISO File

ISO or optical disc image, from the ISO 9660[58] file system used with CD-ROM media, is a disk image that contains everything that would be written to an optical disc, disk sector by disc sector, including the optical disc file system.

This format is typically used as virtual disks for virtual machines like VirtualBox[59] or VMware[60].

First, we need to create a directory named iso in the home directory and navigate into it.

```
mkdir -p iso && cd iso
```

We'll download the ISO file for Ubuntu 23.10 using the `wget` command.

```
wget https://releases.ubuntu.com//mantic/ubuntu-23.10-live-server-amd64.iso
```

Next, let's create two directories, `rootfs` and `unsquashfs`. The `rootfs` directory will be used for mounting the ISO, and `unsquashfs` will be used to extract the filesystem.

```
mkdir -p rootfs unsquashfs
```

Now, we'll mount the downloaded ISO into the `rootfs` directory.

```
sudo mount -o loop ubuntu-23.10-live-server-amd64.iso rootfs
```

We need to find the `ubuntu-server-minimal.squashfs` file inside the mounted ISO.

> **i** Squashfs is a highly compressed read-only filesystem for Linux.

```
squashfs_file=$(sudo find rootfs -name ubuntu-server-minimal.squashfs)
```

Using the `unsquashfs` command, we'll extract the files from the squashfs filesystem into the `unsquashfs` directory.

```
sudo unsquashfs -f -d unsquashfs $squashfs_file
```

We'll compress the `unsquashfs` directory and then import it into Docker to create an image labeled `my_iso_image:v1`.

[58] https://en.wikipedia.org/wiki/ISO_9660
[59] https://www.virtualbox.org/
[60] https://www.vmware.com/

```
sudo tar -C unsquashfs -c . | docker import - my_iso_image:v1
```

To ensure that our image was created correctly, we'll run a test command (`ls`) inside a container spawned from the `my_iso_image:v1` image.

```
docker run --rm -it my_iso_image:v1 ls
```

After testing, we'll unmount the ISO from `rootfs` and then remove the `rootfs` and `unsquashfs` directories, as well as the downloaded ISO file.

```
sudo umount rootfs
sudo rm -rf rootfs unsquashfs ubuntu-23.10-live-server-amd64.iso
```

Finally, let's list all the Docker images present on our system to verify our new image's presence.

```
docker image ls | grep my_iso_image
```

Creating a Docker Image from a VMDK File

VMDK (Virtual Machine Disk) is a file format that describes containers for virtual hard disk drives to be used in virtual machines like VMware Workstation[61] or VirtualBox[62].

In this example, we'll use a VMDK file that contains a virtual machine with Ubuntu 23.10 installed.

Set up a directory for VMDK operations:

```
mkdir -p vmdk && cd vmdk
```

Download the VMDK file from SourceForge:

```
wget https://sourceforge.net/projects/osboxes/files/v/vm/55-U--u/23.04/64bit.7z/down\
load
```

Install the necessary tools to extract 7z archives and extract the downloaded file:

```
sudo apt install -y p7zip-full
7z x download
```

Install required tools to convert VMDK files into Docker images:

[61] https://www.vmware.com/products/workstation-pro.html
[62] https://www.virtualbox.org/

```
apt install -y libguestfs-tools qemu qemu-kvm libvirt-clients libvirt-daemon-system \
virtinst bridge-utils
```

Move and rename the VMDK image to a more accessible name:

```
mv "64bit/64bit/Ubuntu 23.04 64bit.vmdk" image.vmdk
```

Convert the VMDK image into a tar archive:

```
virt-tar-out -a image.vmdk / my_vmdk_archive.tar
```

Import the tar archive as a Docker image:

```
docker import my_vmdk_archive.tar my_vmdk_image:v1
```

Test the newly created Docker image:

```
docker run --rm -it my_vmdk_image:v1 ls
```

Clean up by removing the downloaded and intermediate files:

```
rm -rf image.vmdk download my_vmdk_archive.tar 64bit
```

List the available Docker images to confirm the operation:

```
docker image ls
```

Creating a Docker Image Using Debian's Debootstrap

Debootstrap is a tool that allows you to install a Debian-based Linux distribution into a subdirectory of another, already installed Linux distribution. It can be used to create a Debian base system from scratch, without requiring the availability of dpkg or apt.

Let's see how to use it to create a Docker image.

Start by creating a directory for the Docker image:

```
mkdir -p debootstrap && cd debootstrap
```

Install the debootstrap package:

```
sudo apt install -y debootstrap
```

Create a temporary directory for the Debian installation:

```
sudo debootstrap --variant=minbase bookworm ./bookworm
```

Create a tar archive from the temporary directory:

```
sudo tar -C bookworm/ -c . | docker import - my_debootstrap_image:v1
```

Test the newly created Docker image:

```
# List the files in the image
docker run --rm -it my_debootstrap_image:v1 ls
# Get the OS version
docker run --rm -it my_debootstrap_image:v1 /bin/bash -c 'cat /etc/os-release'
```

You can see that the image was created and you can find it using the `docker image ls` command.

```
docker image ls my_debootstrap_image
```

Clean up by removing the temporary directory:

```
sudo rm -rf bookworm
```

Docker Images out of VMs and the 6 R's of Migration

As you may have noticed, the previous two examples are very similar. The only difference is the format of the image. In the first example, we used an ISO file, and in the second example, we used a VMDK file. In both cases, we always end up with a tar archive that we import into Docker to create an image.

In other words, converting a VM image to a Docker image is fundamentally about extracting the filesystem from the VM and packaging it in a way that Docker understands (tar archive). The process can vary depending on the type of VM image format you're starting with (ISO, VMDK, QCOW2, etc.), but the general approach remains the same.

It is worth mentioning that VMs are full-blown virtual machines with their kernels, while containers share the host's kernel. Some software configured to run on a VM might not run out of the box in a container environment. Proper testing and validation are crucial after the conversion.

If you're moving to the cloud or an orchestrator, you are most likely going to use one of the 6 R's of migration:

- Rehosting (Lift and Shift): This strategy involves moving applications without any modification, often to an Infrastructure as a Service (IaaS) platform. It's essentially moving from an on-premises environment to a cloud environment.
- Redeployment: Migrating to a cloud provider's infrastructure, potentially leveraging cloud-native features but not necessarily altering the application code itself.
- Repackaging: Taking existing applications and placing them into containers or into a PaaS (Platform as a Service) environment. The application might undergo some modification to fit into the new environment, but not a complete overhaul.
- Refactoring (Re-architecting): This involves reimagining how the application is architected and developed, using cloud-native features. It's the most involved and requires changes to the application code.
- Repurchasing: Switching to a different product altogether. This might involve moving from a traditional CRM to a cloud solution like Salesforce.
- Retiring: Deciding that certain parts of the IT portfolio are no longer necessary and can be shut down, saving costs.
- Retaining: Keeping certain components of the IT portfolio in their current state. This might be used for legacy systems that are critical to business operations but not suitable for cloud migration.

So, when you're moving a VM to a container, you're essentially taking an existing application, potentially making some minor adjustments, and then "packaging" it in a new format suitable for container orchestration platforms like Kubernetes or Docker Swarm. This approach is consistent with the repackaging strategy.

Creating and Understanding the Scratch Image

In Docker, there's a special image called the scratch image. It essentially represents a blank image with no layers. Though you typically don't need to create a scratch image yourself (because it is already available in Docker), it's useful to understand how it works.

A scratch image is conceptually equivalent to an image built using an empty tar archive.

To illustrate, you can effectively "create" a scratch image like this:

```
tar cv --files-from /dev/null | docker import - scratch
```

However, again, it's worth noting that Docker provides scratch by default, so you rarely, if ever, need to execute the above command.

Here's a practical example of how you might use the scratch image in a Dockerfile, as you'd see in Docker's official repositories:

```
FROM scratch
ADD hello /
CMD ["/hello"]
```

Scratch image is useful when you want to create a statically compiled binary. This is a binary that is compiled to run on a specific operating system and CPU architecture. It does not depend on any shared libraries.

When you run a container, you are using an image that is based on a base image. This base image is based on another base image, and so on. But when does it stop? It stops when you reach the scratch image. This image is the base image of all base images.

Docker Hub and Docker Registry

Docker Hub, Public and Private Registries

When you run an Nginx container, for example, the Docker engine will look for the image locally, and if it doesn't find it, it will pull it from a remote registry.

Let's try this:

```
# Remove any existing Nginx image
docker rmi nginx:latest
# Run Nginx
docker run --name nginx -d nginx
```

You should see something like this:

```
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
578acb154839: Pull complete
e398db710407: Pull complete
85c41ebe6d66: Pull complete
7170a263b582: Pull complete
8f28d06e2e2e: Pull complete
6f837de2f887: Pull complete
c1dfc7e1671e: Pull complete
Digest: sha256:86e53c4c16a6a276b204b0fd3a8143d86547c967dc8258b3d47c3a21bb68d3c6
Status: Downloaded newer image for nginx:latest
fa097d42c051a4b1eef64ab3f062578ace630b433acb6e4202bd3d90aeaccbad
```

The Docker engine is informing us that it couldn't find the image `nginx:latest` locally, so it fetched it from `library/nginx`.

In reality, what the `docker run` command did was:

- `docker pull nginx:latest` to fetch the image from the remote registry
- `docker create` to create a container based on the image
- `docker start` to start the container

The actual pull command is `docker pull nginx:latest`, but behind the scenes, it executes the full command `docker pull docker.io/library/nginx:latest`.

The `docker.io` part represents the default registry that Docker uses when a specific registry is not specified.

The default remote registry is known as Docker Hub[63]. This is where official images can be found. Docker Hub is a public registry that is accessible to everyone. It offers both free and paid accounts. The free account allows you to create public repositories, while the paid account allows you to create private repositories.

However, Docker Hub is not the only available registry. You can also use other registries such as Quay[64] or Google Container Registry[65], or even host your own private registry.

In general, think of Docker Hub and Docker registry as a GIT repository for Docker images. It allows you to push, pull, store, share, and tag images based on your requirements.

Docker Hub: The Official Docker Registry

Docker Hub is a cloud registry service provided by Docker Inc. It allows you to store and share Docker images.

Docker enables you to package code, artifacts, and configurations into a single image. These images can be reused by you, your colleagues, your customers, or anyone else who has access to your registry.

When you want to share your code, you typically use a git repository like GitHub or Bitbucket. Similarly, when you want to share your Docker images, you can use Docker Hub.

Docker Hub is a public Docker repository, and it also offers a paid version that provides private repositories and additional features such as security scanning.

Docker Hub offers the following capabilities:

- Access to community, official, and private image libraries
- Public or paid private image repositories where you can push and pull your images to/from your servers
- Creation and building of new images with different tags when the source code inside your container changes
- Creation and configuration of webhooks to trigger actions after a successful push to a repository
- Integration with GitHub and Bitbucket

You can also:

[63] https://hub.docker.com/
[64] https://quay.io/
[65] https://cloud.google.com/container-registry/

- Create and manage teams and organizations
- Create a company
- Enforce sign-in
- Set up SSO and SCIM
- Use Group mapping
- Carry out domain audits
- Turn on Registry Access Management

Automated builds of images from GitHub or Bitbucket repositories, automated security scanning, and triggering of automatic actions using Docker Hub webhooks can be used in a CI/CD pipeline to automate the deployment of your applications.

To use Docker Hub, visit hub.docker.com[66] and create an account. Then, using the CLI, you can authenticate to it using the following command:

```
1 docker login
```

In reality, the command `docker login` is just a shortcut for `docker login docker.io` where `docker.io` is the address of the default registry (Docker Hub).

Using Docker Hub

We will be tagging an image and pushing it to your Docker Hub account. Follow these steps:

- Create an account on Docker Hub.
- Authenticate to Docker Hub on your local machine by executing the command `docker login`.

Next, let's pull the MariaDB image from Docker Hub and run it:

```
1 docker run --name mariadb -e MYSQL_ROOT_PASSWORD=password -d mariadb
```

To see the list of images, type `docker images`:

```
1 docker images ls
```

You should see the MariaDB image:

Example:

[66] https://hub.docker.com/

```
mariadb                latest    f35870862d64   3 weeks ago     404MB
```

To push the same image to your repository, tag it using the following command:

```
export DOCKER_ID="YOUR_DOCKER_ID"
export IMAGE_ID=docker.io/$DOCKER_ID/mariadb:latest
docker tag mariadb $IMAGE_ID
```

Replace `YOUR_DOCKER_ID` with your Docker ID. For example, if your Docker ID is `foo`, the final command should be:

```
docker tag mariadb docker.io/foo/mariadb:latest
```

After listing the images again using `docker images`, you will notice a new image in the list:

```
foo/mariadb         latest    f35870862d64   3 weeks ago     404MB
mariadb             latest    f35870862d64   3 weeks ago     404MB
```

Both images have the same ID because they are the same image with different tags.

Now you can push it:

```
docker push $IMAGE_ID
```

Visit this URL:

```
echo https://hub.docker.com/r/$DOCKER_ID/mariadb
```

You can also tag the image in a different way (commonly used):

```
export DOCKER_ID="YOUR_DOCKER_ID"
export IMAGE_ID=$DOCKER_ID/mariadb:latest
docker tag mariadb $IMAGE_ID
```

The difference is that we didn't specify the registry. In this case, the default registry, which is Docker Hub, is used.

The final command, if your Docker ID is `foo`, should be:

```
# Instead of docker.io/foo/mariadb:latest use:
docker tag mariadb foo/mariadb:latest
```

The same way of tagging images destined for Docker Hub is used when you build images from a Dockerfile.

Example:

```
export DOCKER_ID="YOUR_DOCKER_ID"
docker build -t $DOCKER_ID/mariadb:latest .
docker push $DOCKER_ID/mariadb:latest
```

DockerHub Alternatives

Docker Hub is not the only registry available. You can use other registries or you can host your own private registry.

A private Docker Registry is a server-side application conceived to be an "on-premise Docker Hub".

Just like Docker Hub, it helps you push, pull, and distribute your images publicly and privately.

Docker has developed an open-source under-Apache-license registry called Distrubution[67] (formerly known as Docker Registry). It is a highly scalable server-side application that stores and lets you distribute Docker images. Docker Registry could also be a cloud-based solution.

Other alternatives are:

- Quay[68]: A registry for storing and building container images as well as distributing other OCI artifacts.
- Google Artifact Registry[69]: Artifact Registry provides a single location for storing and managing your packages and Docker container images. You can: Integrate Artifact Registry with Google Cloud CI/CD services or your existing CI/CD tools. Store artifacts from Cloud Build.
- Amazon Elastic Container Registry[70]: An AWS managed container image registry service that is secure, scalable, and reliable.
- Azure Container Registry[71]: A registry of Docker and Open Container Initiative (OCI) images, with support for all OCI artifacts.
- JFrog Artifactory[72]: A repository manager that supports all available software package types, enabling automated continous integration and delivery.
- Harbor[73]: An open source trusted cloud native registry project that stores, signs, and scans content.

[67](https://hub.docker.com/_/registry)
[68]https://quay.io
[69]https://cloud.google.com/artifact-registry
[70]https://aws.amazon.com/ecr/
[71]https://azure.microsoft.com/en-us/services/container-registry/
[72]https://jfrog.com/artifactory/
[73]https://goharbor.io/

Creating a Private Docker Registry

Docker Registry can be run in a Docker container and deployed on a server or cluster managed by Kubernetes, Docker Swarm, or any other container orchestration tool. The Docker image for Docker Registry is available here[74].

To create a registry, simply pull and run the image using the following command:

```
docker run -d -p 5000:5000 --restart=always --name registry registry:2.7.1
```

Let's test the registry by pulling an image from Docker Hub, tagging it, and pushing it to our own registry:

```
docker pull eon01/infinite
```

Tag the image:

```
docker tag eon01/infinite:latest localhost:5000/infinite:latest
```

Push it to the local repository:

```
docker push localhost:5000/infinite:latest
```

Now re-pull it:

```
docker pull localhost:5000/infinite:latest
```

As you can see, we have successfully pushed and pulled an image from our local registry. The registry is running on port 5000 on the same machine where Docker is running, which is why we used `localhost:5000`. When tagging an image, make sure to include the registry host name or IP address:

```
docker tag [IMAGE_ID] [REGISTRY_HOST]:[REGISTRY_PORT]/[IMAGE_NAME]:[TAG]
```

When you build an image from a Dockerfile, you should also follow the same rule:

```
docker build -t [REGISTRY_HOST]:[REGISTRY_PORT]/[IMAGE_NAME]:[TAG] .
```

Persisting the Registry Data

Since Docker images are stored inside the registry under "/var/lib/registry", you can mount a volume to this directory to persist the images.

[74] https://hub.docker.com/_/registry/

```
docker run -d -p 5000:5000 --restart=always --name registry -v /data/registry:/var/l\
ib/registry registry:2.7.1
```

Configuring the Registry

If you want to setup an authentication mechanism, you can use basic authentication, token authentication or other options documented here[75].

Let's see how to use basic authentication. First, create a password file using "htpasswd":

```
# Remove the old registry
docker rm -f registry
# Install htpasswd
apt-get install -y apache2-utils
# Create the password file
cd $HOME
mkdir /root/auth
export USERNAME=admin
export PASSWORD=admin
# Use -B for bcrypt,  -n for no newline and -b for basic auth
htpasswd -Bbn $USERNAME $PASSWORD > /root/auth/htpasswd
```

Make sure to change the username and password to your own values. Now launch the registry using the following command:

```
docker run -d \
  -p 5000:5000 \
  --restart=always \
  --name registry \
  -v /root/auth:/root/auth \
  -e "REGISTRY_AUTH=htpasswd" \
  -e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm" \
  -e REGISTRY_AUTH_HTPASSWD_PATH=/root/auth/htpasswd \
  registry:2.7.1
```

To test it, tag an image and push it to the registry but before that, you need to login:

[75] https://distribution.github.io/distribution/

```
docker login localhost:5000
docker tag eon01/infinite:latest localhost:5000/infinite:latest
docker push localhost:5000/infinite:latest
```

You can check the repositories and tags using the following command:

```
# Check the repositories
curl -u $USERNAME:$PASSWORD -X GET http://localhost:5000/v2/_catalog
# Check the tags
curl -u $USERNAME:$PASSWORD -X GET http://localhost:5000/v2/infinite/tags/list
```

Other Options

There are many other options that you can use to configure the registry, for example, you can enable TLS, configure the storage driver, and more. You can find them in the official documentation here[76].

[76] https://distribution.github.io/distribution/

Optimizing Docker Images

You can find many public images of the same application, but not all of them are optimized.

- Some images can be quite large, leading to longer build and network transfer times. These heavy images can also increase deployment time.
- Other images may have many layers, which can lead to slower build times.
- Other images may contain unnecessary tools and packages that are not required to run the application. This can lead to security issues and increase the attack surface.
- etc.

Optimizing Docker images involves more than just reducing their size. It also includes managing how layers are built, managing dependencies, using multi-stage builds, and selecting the appropriate base image.

The main reasons for optimizing Docker images are:

- Faster build, deployment, and startup times
- Faster network transfer
- Reduced disk, memory, and network usage
- Lower CPU usage

All of the above contribute to faster development and deployment cycles, cost reduction, and ensuring quality - the ultimate goal of any DevOps engineer.

In the following sections, we will discuss some best practices for optimizing Docker images.

Less Layers = Faster Builds?

The build time depends on the number of layers in the image. The more layers there are, the longer it takes to build the image. This is because Docker needs to create a container for each layer and then copy the files from the previous layer to the current layer. Theoritically and logically, this is true but let's test and see if this is true in practice.

Let's test this by creating two Dockerfiles. The first one consists of 5 lines:

```
cat <<EOF > Dockerfile
FROM ubuntu
RUN apt-get update -y
RUN apt-get install python3 -y
RUN apt-get install python3-pip -y
RUN pip3 install flask
EOF
```

Build the image while measuring the build time:

```
time docker build -t my-python-app:v1 --no-cache .
```

```
docker images my-python-app:v1 --format "{{.Repository}} {{.Tag}} {{.Size}}"
```

This one consists of only two lines:

```
cat <<EOF > Dockerfile
FROM ubuntu
RUN apt-get update -y && \
    apt-get install python3 -y && \
    apt-get install python3-pip -y && \
    pip3 install flask
EOF
```

Build the image and measure the build time:

```
time docker build -t my-python-app:v2 --no-cache .
```

My results are:

```
# The first Dockerfile
real    0m25,260s                                                    \

user    0m0,157s
sys     0m0,071s
# The second Dockerfile
real    0m25,160s                                                    \

user    0m0,113s
sys     0m0,088s
```

Based on the results, the second Dockerfile is faster. The "real" time (wall-clock time) for the second Dockerfile is indeed slightly shorter (by 0.1 seconds) than the first Dockerfile. The difference in execution time is minimal (only a tenth of a second), so in practical terms, the performance difference might not be significant for many use cases.

Both images are installing Python and using the Ubuntu base image. They may have the same size (or a slightly different size), but the second image is built faster because it has fewer layers.

```
# Check that the size of the images is the same
docker images my-python-app --format "{{.Repository}} {{.Tag}} {{.Size}}"
```

Let's try another example. This time, we will compare two Dockerfiles:

First Dockerfile:

```
FROM busybox
RUN echo This is the 1st command > 1 && rm -f 1
RUN echo This is the 2nd command > 2 && rm -f 2
RUN echo This is the 3rd command > 3 && rm -f 3
RUN echo This is the 4th command > 4 && rm -f 4
[...]
RUN echo This is the 20th command > 20 && rm -f 20
```

Second Dockerfile:

```
RUN echo This is the 1st command > 1 && \
    rm -f 1 && \
    echo This is the 2nd command > 2 && \
    rm -f 2 && \
    echo This is the 3rd command > 3 && \
    rm -f 3 && \
    echo This is the 4th command > 4 && \
    rm -f 4 && \
    [...]
    rm -f 19 && \
    echo This is the 20th command > 20 && \
    rm -f 20
```

This script will create two Dockerfiles, build and measure the build time. Create it using the following command:

```
cat << 'REALEND' > build_and_measure.sh
#!/bin/bash

# Generate the first Dockerfile
cat > Dockerfile_1 <<EOF
FROM busybox
EOF
for i in $(seq 1 20); do
    echo "RUN echo This is the ${i}th command > ${i} && rm -f ${i}" >> Dockerfile_1
done

# Start generating the Dockerfile_2
echo "FROM busybox" > Dockerfile_2
# Append the RUN command with automation for the repeated pattern
echo "RUN \\" >> Dockerfile_2
for i in $(seq 1 19); do # Loop until the 19th command
    echo "    echo This is the ${i}th command > ${i} && \\" >> Dockerfile_2
    echo "    rm -f ${i} && \\" >> Dockerfile_2
done
# Append the 20th command without a trailing `&& \`
echo "    echo This is the 20th command > 20 && \\" >> Dockerfile_2
echo "    rm -f 20" >> Dockerfile_2

# Build the first Dockerfile and measure the time
echo "Building the first image..."
start_time1=$(date +%s%N)
docker build -t image_1 -f Dockerfile_1 --no-cache .
end_time1=$(date +%s%N)
elapsed_time1=$((($end_time1 - $start_time1)/1000000))

# Build the second Dockerfile and measure the time
echo "Building the second image..."
start_time2=$(date +%s%N)
docker build -t image_2 -f Dockerfile_2 --no-cache .
end_time2=$(date +%s%N)
elapsed_time2=$((($end_time2 - $start_time2)/1000000))

# Print the results
echo "First image build time: $elapsed_time1 milliseconds"
echo "Second image build time: $elapsed_time2 milliseconds"
REALEND
```

Run the script:

```
chmod +x build_and_measure.sh && ./build_and_measure.sh
```

This is the output I got:

```
First image build time: 7798 milliseconds
Second image build time: 540 milliseconds
```

There's a difference of 7.258 seconds between the two images which is significant.

Let's check the size of the images:

```
docker images image_* --format "{{.Repository}} {{.Tag}} {{.Size}}"
```

It should be the same - there's no difference in size when using the `RUN` command in a single line or multiple lines.

```
image_1 latest 4.26MB
image_2 latest 4.26MB
```

So what should we understand from this? Should we always try to reduce the number of layers? The answer, as you can see, is not always. The more layers there are, the longer it takes to build the image. However, the difference in execution time might not be significant when the number of layers is small.

The final answer is: it depends. But I'd say that you should try to reduce the number of layers as much as possible. It's a good practice to follow.

The number of layers is not the only factor that may affect the build time. Other factors should be considered when building Docker images. We are going to discuss some of them in the following sections.

Is There a Maximum Number of Layers?

The maximum number of layers that can exist in an image's history (127 layers) is not documented[77], but this seems to be a limitation imposed by AUFS[78].

Linux has a restriction that does not accept more than 127 layers, which can be raised in modern kernels but is not the default.

> **i** AUFS is a union filesystem. The aufs storage driver was previously the default storage driver used for managing images and layers on Docker for Ubuntu, and for Debian versions prior to Stretch. It was deprecated in favor of overlay2.

[77] https://github.com/docker/docs/issues/8230
[78] https://en.wikipedia.org/wiki/Aufs

The overlay2 storage driver, which is the default storage drivers used by Docker now, has also limitation on the number of layers it supports. With the overlay2 driver, there's a maximum of 128 lower OverlayFS layers. However, this limit doesn't count the upper writable layer, so you can think of it as having a total of 129 layers, including the upper writable layer.

Let's test this limitation by creating a Dockerfile:

```
# Define the base image
echo "FROM busybox" > Dockerfile_test

# We add 128 layers here.
for i in $(seq 1 128); do
    echo "RUN touch /file${i} && rm /file${i}" >> Dockerfile_test
done

# Add one more to exceed the limit of 128 layers.
echo "RUN touch /limit_exceeded && rm /limit_exceeded" >> Dockerfile_test
```

It it worth noting that the effective limit is 129 layers, including the upper writable layer (the layer where the container's filesystem is mounted). The base image is not counted as a layer in the limit.

Build the image:

```
docker build -t test-overlay2-layers -f Dockerfile_test .
```

The build should fail with a message similar to this:

```
 => ERROR [130/130] RUN touch /limit_exceeded && rm /limit_exceeded 0.0s
------
 > [130/130] RUN touch /limit_exceeded && rm /limit_exceeded:
------
Dockerfile_test:130
--------------------
 128 |     RUN touch /file127 && rm /file127
 129 |     RUN touch /file128 && rm /file128
 130 | >>> RUN touch /limit_exceeded && rm /limit_exceeded
 131 |
--------------------
ERROR: failed to solve: failed to prepare vbpizlnaw1zjd46noaaivf6o2 as m8yxsz8tkmygp\
7vth18esd2ru: max depth exceeded
```

Optimizing Dockerfile Layer Caching for Dependency Management

Usually, when running an application, you will need to install some dependencies. Here are some examples:

- When using Python, you can install libraries using `pip install`
- When using Node.js, you can install libraries using `npm install`
- When using Java, you can install libraries using `mvn install`

Let's assume we need to install libraries using `pip install` from a `requirements.txt` file.

```
cat <<EOF > Dockerfile
FROM alpine
# Copy the requirements file
COPY requirements.txt .
# Install Python and the dependencies
RUN apk add python3 py3-pip && pip3 install -r requirements.txt
# Run the application
CMD ["python3", "app.py"]
EOF
```

In the above example, we followed the best practice of combining the `apk add` and `pip install` commands into a single `RUN` instruction. This reduces the number of layers in the image. However, in this particular example, the "requirements.txt" file may change while the `apk add` command does not. If the requirements file is modified, all layers will be rebuilt, and the `RUN` instruction will be executed again due to this change. This means that both the `apk add` and `pip install` commands will be executed again, even though we only need to execute the `pip install` command to update the Python dependencies. This is not optimal.

To avoid this, we can optimize the Dockerfile by restructuring our instructions. One common approach is to separate the installation of system packages (which typically change less frequently) from the installation of Python packages.

```
cat <<EOF > Dockerfile
FROM alpine
# Install system dependencies (not subject to change frequently)
RUN apk add python3 py3-pip
# Copy the requirements file and install Python dependencies (subject to change freq\
uently)
COPY requirements.txt .
RUN pip3 install -r requirements.txt
# Run the application
CMD ["python3", "app.py"]
EOF
```

Splitting the commands allows Docker to use the cached layer for system packages installation (`apk add python3 py3-pip`) if the "requirements.txt" file remains unchanged. It will only rebuild the layer where Python packages are installed (`pip3 install -r requirements.txt`) when there are changes in Python dependencies. This approach speeds up the build process when only the Python dependencies are modified.

The Multi-Stage Build

Let's reconsider this Dockerfile as an example:

```
# Use the specified image as the base
FROM golang:1.21.3-alpine3.17
# Set the working directory inside the container
WORKDIR /app
# Copy the local package files to the container's workspace
ADD . /app
# Build the Go app
RUN go mod init my-golang-app
RUN go build -o main .
# Run the binary program produced by `go build`
CMD ["/app/main"]
```

This Dockerfile is not optimized. It creates a large image that includes the Go compiler and other unnecessary tools. To optimize it, we can use a multi-stage build.

In the first stage, we build the application using the `golang:1.21.3-alpine3.17` image, which contains the Go compiler and other tools that are not required to run the application. We name this stage `builder`.

```
# Build stage
FROM golang:1.21.3-alpine3.17 AS builder

# Set the working directory inside the container
WORKDIR /app

# Copy the local package files to the container's workspace
COPY . .

# Initialize a module and build the Go app
RUN go mod init my-golang-app
RUN go build -o main .
```

In the second stage, we copy the binary from the first stage to a smaller base image (`alpine:3.17`) to run the application.

```
# Final stage
FROM alpine:3.17

# Set the working directory
WORKDIR /app

# Copy the binary from the build stage
COPY --from=builder /app/main .

# Run the binary program produced by `go build`
CMD ["./main"]
```

To create the final Dockerfile, you can use the following script:

```
cat <<'EOF' > Dockerfile
# Build stage
FROM golang:1.21.3-alpine3.17 AS builder
# Set the working directory inside the container
WORKDIR /app
# Copy the local package files to the container's workspace
COPY . .
# Initialize a module and build the Go app
RUN go mod init my-golang-app
RUN go build -o main .
##############################################
# Final stage
```

```
FROM alpine:3.17
# Set the working directory
WORKDIR /app
# Copy the binary from the build stage
COPY --from=builder /app/main .
# Run the binary program produced by `go build`
CMD ["./main"]
EOF
```

To build the image, run the following command:

```
docker build -t my-golang-app:v2 .
```

You can compare the size of the final image (`my-golang-app:v2`) with the previous one (`my-golang-app:v1`) using the `docker images` command:

```
docker images my-golang-app --format "{{.Repository}} {{.Tag}} {{.Size}}"
```

Using numbers instead of stage names is also possible. Here's an alternative Dockerfile using numbers:

```
cat <<'EOF' > Dockerfile
FROM golang:1.21.3-alpine3.17
WORKDIR /app
COPY . .
RUN go mod init my-golang-app
RUN go build -o main .
##############################################
FROM alpine:3.17
WORKDIR /app
COPY --from=0 /app/main .
CMD ["./main"]
EOF
```

In this case, we use `COPY --from=0 /app/main .` instead of `COPY --from=builder /app/main .`. The first stage is `0`, and the second stage is `1`.

Additionally, we can further optimize the Dockerfile by using a smaller base image called Scratch[79]. The Scratch image is empty, containing no files or packages, and is only 0.5 MB in size.

Here's the final Dockerfile using the Scratch image:

[79] https://hub.docker.com/_/scratch

```
cat <<'EOF' > Dockerfile
# Build stage
FROM golang:1.21.3-alpine3.17 AS builder
# Set the working directory inside the container
WORKDIR /app
# Copy the local package files to the container's workspace
COPY . .
# Initialize a module and build the Go app
RUN go mod init my-golang-app
RUN go build -o main .
##############################################
# Final stage
FROM scratch
WORKDIR /app
COPY --from=builder /app/main .
CMD ["./main"]
EOF
```

To build the image:

```
docker build -t my-golang-app:v4 .
```

You can compare the size of the final image (`my-golang-app:v4`) with the previous ones (`my-golang-app:v1`, `my-golang-app:v2`, and `my-golang-app:v3`) using the `docker images` command:

```
docker images my-golang-app --format "{{.Repository}} {{.Tag}} {{.Size}}"
```

my-golang-app:v1 is 248 MB in size, my-golang-app:v2 and my-golang-app:v3 are the same image and are 8.86 MB in size, and my-golang-app:v4 is only 1.82 MB in size.

```
my-golang-app v4 1.8MB
my-golang-app v2 8.86MB
my-golang-app v3 8.86MB
my-golang-app v1 248MB
```

Smaller Images

Many Docker images weigh more than 1GB. How do they become so heavy? Do they really need to be this large? Can we make them smaller without sacrificing functionality? The answer is yes.

Let's reconsider this Dockerfile as an example:

```
FROM ubuntu
RUN apt-get update -y && \
    apt-get install python3 -y && \
    apt-get install python3-pip -y && \
    pip3 install flask
```

Instead of using Ubuntu here, you can use smaller base images like Alpine[80]. The Alpine image is only 5 MB in size while Ubuntu is 77.8 MB.

Let's create a simple Dockerfile that installs Python on Alpine.

```
cat <<EOF > Dockerfile
FROM alpine
RUN apk add python3 py3-pip && pip3 install flask
EOF
```

Build the image:

```
docker build -t my-python-app:alpine .
```

Check the size of the image:

```
docker images my-python-app:alpine --format "{{.Repository}} {{.Tag}} {{.Size}}"
```

Compared to the Ubuntu-based image, the Alpine-based image is much smaller:

```
my-python-app   alpine  77.6MB
my-python-app   v2      476MB
my-python-app   v1      477MB
```

Next, we are going to explore a list of images that are small and that you can use as a base image for your Docker images.

Scratch

The smallest and initial image you can use is the "scratch" image. This image is not derived from any other image; it is essentially empty. It is particularly useful for creating base images (like debian and busybox) or extremely minimal images (containing only a single binary and its dependencies, like hello-world). It is small, fast, secure, and free of bugs. In essence, it is a completely empty image.

[80] https://hub.docker.com/_/alpine

BusyBox

BusyBox is a software that runs in various POSIX environments such as Linux, Android, and FreeBSD. Although it is designed to work with interfaces provided by the Linux kernel, it can be utilized in other environments as well. It was specifically developed for embedded operating systems that have limited resources. The authors often refer to it as "The Swiss Army knife of Embedded Linux" because its single executable replaces the functionality of more than 300 common commands. BusyBox is released under the GNU General Public License v2, making it free software.

In essence, BusyBox provides a collection of stripped-down Unix tools packaged in a single executable file. For example, the `ls` command can be executed using BusyBox.

```
/bin/busybox ls
```

BusyBox is the winner for having the smallest images (somewhere between 1 and 5 Mb in on-disk size) that can be used with Docker. Executing a docker pull busybox command will take approximately 1 second.

```
time docker pull busybox
Using default tag: latest
latest: Pulling from library/busybox
Digest: sha256:3fbc632167424a6d997e74f52b878d7cc478225cffac6bc977eedfe51c7f4e79
Status: Image is up to date for busybox:latest
docker.io/library/busybox:latest

real    0m1,086s
user    0m0,009s
sys     0m0,012s
```

While there are advantages to the minimal size of BusyBox, there are also some drawbacks. BusyBox comes with a minimalistic package manager called opkg. However, the version of BusyBox commonly found in Docker images may not have opkg installed by default, as BusyBox is often configured to be as lightweight as possible.

To view the list of binaries included in BusyBox, you can run the `busybox` executable from Docker.

```
docker run busybox busybox
```

Unfortunately, this tiny distribution does not have a proper package manager, which can be quite inconvenient. However, Alpine is an excellent alternative if you require a package manager.

Alpine Linux

Alpine Linux is a lightweight Linux distribution that prioritizes security. It is based on "musl libc" and BusyBox. Alpine Linux is highly favored for Docker images due to its small size of only 5.5MB. It also includes a package manager called apk[81].

Alpine has also one of the fastest boot times of any operating system.

Considering its features, Alpine Linux is an excellent choice as a base image for your Docker images and is widely used in embedded devices and routers.

Phusion Baseimage / Baseimage-docker / Passenger-docker

Phusion Baseimage, also known as Baseimage-docker, is a minimal Ubuntu base image that has been modified to be Docker-friendly.

According to the authors, it consumes only 8.3 MB of RAM and is more powerful than Busybox or Alpine.

In addition to being Ubuntu, it includes:

- Modifications specifically designed for Docker-friendliness.
- Administration tools that are particularly useful in the context of Docker.
- Mechanisms for easily running multiple processes without conflicting with the Docker philosophy.

You can find more detailed information in the official GitHub repository[82].

The team behind this project has also created a base image for running Ruby, Python, Node.js, and Meteor web apps called passenger-docker[83].

> Passenger-docker is a set of Docker images meant to serve as good bases for Ruby, Python, Node.js and Meteor web app images. In line with Phusion Passenger's goal, passenger-docker's goal is to make Docker image building for web apps much easier and faster. (source: phusion/passenger-docker[84].)

Other Techniques: Squashing, Distroless, etc

There are other techniques that can be used to reduce the size of Docker images. For example, you can use tools like docker-squash[85] to reduce the size of Docker images by removing intermediate layers. You can also use tools like docker-slim[86] to optimize and secure your Docker containers.

[81] https://wiki.alpinelinux.org/wiki/Alpine_Package_Keeper
[82] https://github.com/phusion/baseimage-docker
[83] https://github.com/phusion/passenger-docker
[84] https://github.com/phusion/passenger-docker
[85] https://github.com/goldmann/docker-squash
[86] https://github.com/slimtoolkit/slim

There's also a technique called Distroless[87] that can be used to create container images that contain only your application and its runtime dependencies. This technique is recommended by Google.

Another technique is reducing the size of your dependencies. For example, if you are developing a Python application and don't need the full Python runtime, you can use PyInstaller[88] to create a single executable file that contains your application and its dependencies. This will reduce the size of your Docker image.

If you're building a Python container for production, you can use Pex[89] to create a single executable file that contains your application and its dependencies. PEX files have been used by Twitter to deploy Python applications to production since 2011 and this helped them reduce the size of their images.

While we are talking about Python, you can also use MicroPython[90] when needed. MicroPython is a lean and efficient implementation of the Python 3 programming language that includes a small subset of the Python standard library and is optimised to run on microcontrollers and in constrained environments.

I mentioned Python here as an example, but if you are using another programming language, you can search for similar tools and techniques to reduce the size of your Docker images and adapt them to your needs.

[87] https://github.com/GoogleContainerTools/distroless
[88] https://www.pyinstaller.org/
[89] https://pex.readthedocs.io/en/stable/
[90] https://micropython.org/

Docker Volumes

What is a Docker Volume?

If we run the following container:

```
docker run --name nginx -d nginx
```

We are able to add a file inside the container:

```
docker exec -it nginx bash -c "echo 'Hello World' > /usr/share/nginx/html/index.html"
```

We can check that the file is inside the container:

```
docker exec -it nginx bash -c "cat /usr/share/nginx/html/index.html"
```

When we remove the container, relaunch it, and check the file, we see that the file is not there anymore:

```
docker rm -f nginx
docker run --name nginx -d nginx
docker exec -it nginx bash -c "cat /usr/share/nginx/html/index.html"
```

A container is a process that runs in isolation. When we remove the container, we remove the process and all the data that was inside the container.

Is there a solution to keep the data of a container when we remove it? Yes, we can use Docker volumes.

Docker volumes are directories (or files) that are outside the default Union File System and exist as normal directories and files on the host filesystem. This is why they are not removed when the container is removed and this is why they are a solution to the problem we just saw (ephemeral containers).

Volumes are also used to share data between containers but most commonly they are used in stateful applications (databases, key-value stores, etc.).

Docker volumes report the following events: create, mount, unmount, destroy.

We can use the `docker events` command to see the events related to volumes when we create, delete, or mount them:

```
docker events --filter 'type=volume'
```

Creating and Using Docker Volumes

Volumes are usually created and managed by Docker (but you can also create them manually). When you create a volume, Docker creates a directory under `/var/lib/docker/volumes/` on the host machine and mounts it inside the container under the mount point:

```
/var/lib/docker/volumes/<volume-name-or-id>/_data
```

Let's see how to create a volume:

```
docker volume create my-vol
```

This command will create a volume called `my-vol` and mount it inside the container under `/var/lib/docker/volumes/my-vol/_data`.

In order to use the volumes in a real application, we need to mount them inside the container. We can do this using the `-v` flag:

```
docker run --name nginx -v my-vol:/usr/share/nginx/html -d nginx
```

When we add a file inside the container, we can see it on the host:

```
docker exec -it nginx bash -c "echo 'Hello World' > /usr/share/nginx/html/index.html"
sudo ls -l /var/lib/docker/volumes/my-vol/_data
```

When we remove the container, the volume is not removed:

```
docker rm -f nginx
sudo ls -l /var/lib/docker/volumes/my-vol/_data
```

Listing and Inspecting Docker Volumes

We can list the volumes using the `docker volume ls` command:

```
docker volume ls
```

We can use filters such as `dangling` to list only dangling volumes:

```
docker volume ls -f dangling=true
```

We are going to understand what dangling volumes are in the next section.

To format the output, we can use the `--format` flag:

```
docker volume ls --format "{{.Name}}"
# or
docker volume ls --format "{{.Name}} {{.Driver}}"
# or
docker volume ls --format "table {{.Name}}\t{{.Driver}}\t{{.Mountpoint}}"
```

To have more information about a volume, we can use the `docker volume inspect` command:

```
docker volume inspect my-vol
```

The output is a JSON containing information about the volume:

- `CreatedAt`: The date and time the volume was created.
- `Driver`: The driver used by the volume.
- `Labels`: The labels associated with the volume.
- `Mountpoint`: The mount point of the volume on the host.
- `Name`: The name of the volume.
- `Options`: The driver specific options used when creating the volume.
- `Scope`: The scope of the volume.

Named Volumes vs Anonymous Volumes

When we created the volume `my-vol`, we used the command:

```
docker volume create my-vol
```

This is called a named volume. We can also create anonymous volumes. Anonymous volumes are volumes that are not given a specific name. They are created when we use the `-v` flag without specifying a name:

```
docker run --name nginx_with_anonymous_volume -v /usr/share/nginx/html -d nginx
```

If you inspect the container to get the volume name, you will see that it is a random string:

```
# use the following command to get the volume name
docker inspect nginx_with_anonymous_volume --format "{{range .Mounts}}{{.Name}}{{end\
}}"
```

Note that a container may have multiple volumes, in this case you will get multiple volume names when you run the above command.

The expected output is the identifier of the volume and not a real name. This is because anonymous volumes are not given a name but a random identifier.

You can inspect the volume to get more information about it like the mount point (where it is stores files on the host):

```
vol_id=$(docker inspect nginx_with_anonymous_volume --format "{{range .Mounts}}{{.Na\
me}}{{end}}")
docker volume inspect $vol_id
```

Bind Mounts

A bind mount is a file or directory located anywhere on the host filesystem that is mounted into a container. It is defined by its source path and the container path to which it is mounted. However, bind mounts are not the ideal solution for creating volumes because they are not portable. This lack of portability arises from their reliance on the host filesystem. Additionally, bind mounts are not managed by Docker itself, but rather by the user.

This is an example:

```
# start by creating the folder files
mkdir files
# create a file inside the folder files
echo "Hello World" > files/index.html
# run a container and mount the folder files inside the container
docker run -d --name nginx_with_bind_mount -v "$(pwd)"/files:/usr/share/nginx/html n\
ginx
# test that the file is inside the container
docker exec -it nginx_with_bind_mount bash -c "cat /usr/share/nginx/html/index.html"
```

Data Propagation

When you attach an empty volume to a container directory that already has files, those files are copied into the volume. If you start a container and specify a volume that hasn't been created, Docker will initialize an empty one for you. This can be useful for sharing data between containers.

Let's say we have a Docker image with a directory `/app/data` that contains a file named "sample.txt".

Now, when we run a container from this image and mount an empty volume to `/app/data`, the "sample.txt" will be copied to the volume.

```
# Run a container with an empty volume attached to /app/data
docker run -d -v myvolume:/app/data my_image
```

If you inspect the contents of myvolume, you'll find the "sample.txt" file in it.

What if the volume already has content? Let's see what happens when we mount a volume that already has content to a container directory that also has files.

```
# Create a volume and add some content to it
docker volume create my_test_volume
echo "Test from the volume" > /var/lib/docker/volumes/my_test_volume/_data/index.html
# Create a Dockerfile that updates the index.html file
cat > Dockerfile <<EOF
FROM nginx
RUN echo "Test from the container" > /usr/share/nginx/html/index.html
EOF
# Build the image and run a container from it that uses the volume
docker build -t my_image .
docker run --name container -d -v my_test_volume:/usr/share/nginx/html my_image
# Inspect the contents of the volume
docker exec -it $(docker ps -lq) bash -c "cat /usr/share/nginx/html/index.html"
```

The last command should show:

```
Test from the volume
```

This means that the volume's content "overshadowed the container's original files. Think of it like putting a sticker on a printed page. Even if the page has text or images on it, the sticker will cover and hide whatever's beneath it.

The same thing happens when you use bind mounts instead of volumes.

To summarize, this is what happens when you mount a volume to a container:

Volume Status	Container Status	Result
Empty	Has Data	Data from the container is copied into the volume.
Empty	Empty	Volume remains empty.
Has Data	Empty	Container directory displays the data from the volume.
Has Data	Has Data	Container's original data is overshadowed by the volume data.

Another scenario that could be intresting to explore is when a container with a volume is running, then we update the volume from the host. What happens to the container? Will the container see the changes? The answer is no, unless we restart the container.

Let's see this in action:

```
# Create a volume
docker volume create my_volume_1
# Create a container that uses the volume
docker run --name my_nginx_1 -v my_volume_1:/usr/share/nginx/html -d nginx
# Update the volume from the host
echo "Hello World" > /var/lib/docker/volumes/my_volume_1/_data/index.html
# Checl if the container sees the changes
docker exec -it my_nginx_1 bash -c "cat /usr/share/nginx/html/index.html"
```

You will see that the container does not see the changes. To make the container see the changes, we need to restart it:

```
docker restart my_nginx_1
docker exec -it my_nginx_1 bash -c "cat /usr/share/nginx/html/index.html"
```

What should we understand from this? Volumes are not updated automatically when we update them from the host. We need to restart the container to make it see the changes. Changes will not propagate from the host to the container automatically which is a good thing because we don't want to break our application by updating the volume from the host. However, if you use bind mounts instead of volumes, changes will propagate from the host to the container automatically.

```
# Remove the container
docker rm -f my_nginx_1
# Run the container again but this time use a bind mount instead of a volume
docker run --name my_nginx_1 -v /nginx/:/usr/share/nginx/html -d nginx
# Update the volume from the host
echo "Hello World!" > /nginx/index.html
# Check if the container sees the changes
docker exec -it my_nginx_1 bash -c "cat /usr/share/nginx/html/index.html"
```

You will see that the container sees the changes without the need to restart it.

Dangling Volumes

A dangling volume is a volume that is not associated with any container. This happens when we remove a container that uses a volume without removing the volume. The volume will still exist on the host but it will not be associated with any container.

To show dangling volumes, we can use the following command:

```
docker volume ls -f dangling=true
```

Dangling volumes are not removed automatically but we can clean them and this is what we are going to see in one of the next sections. However, for now, let's see how to calculate the size of a dangling volume. We can use the following script:

```
#!/bin/bash

total_size=0

# Loop through each dangling volume
for volume in $(docker volume ls -qf dangling=true); do
    # Get the mountpoint of the volume
    mountpoint=$(docker volume inspect $volume --format '{{ .Mountpoint }}')
    # Calculate the size of the volume using du
    size_with_unit=$(du -sh $mountpoint | cut -f1)
    size=$(echo $size_with_unit | awk '{print int($1)}')
    unit=${size_with_unit: -1}

    echo "Volume: $volume Size: $size_with_unit"

    # Convert all sizes to KB for aggregation
    case $unit in
```

```
        K)
            total_size=$((total_size + size))
            ;;
        M)
            total_size=$((total_size + size*1024))
            ;;
        G)
            total_size=$((total_size + size*1024*1024))
            ;;
        *)
            echo "Unknown unit: $unit. Skipping volume $volume in aggregation."
            ;;
    esac
done

echo "Total size of all dangling volumes in KB: $total_size KB"
```

TMPFS Mounts

If you need to persist data between the host machine and containers, you should use Docker volumes or bind mounts. However, in cases where you don't need to write files to the container's writable layers, TMPFS mounts can be a suitable option. TMPFS mounts store data in the host's memory, and when the container stops, the TMPFS data is completely removed. This approach is useful for scenarios like passing sensitive files to containers where persistence isn't required, due to the ephemeral nature of these files. Let's explore creating a TMPFS volume for an Nginx container. There are two different methods to achieve this.

1) Use the `--mount` flag.

```
docker run -d \
  -it \
  --name nginx_with_tmpfs_volume_1 \
  --mount type=tmpfs,destination=/secrets \
  nginx
```

2) Use the `--tmpfs` flag.

```
docker run -d \
  -it \
  --name nginx_with_tmpfs_volume_2 \
  --tmpfs /secrets \
  nginx
```

When creating a TMPFS, you can specify options like `tmpfs-size` and `tmpfs-mode`.

Option	Description
`tmpfs-size`	Size of the tmpfs mount in bytes. Unlimited by default.
`tmpfs-mode`	File mode of the tmpfs in octal. For instance, `700` or `0770`. Defaults to `1777` or world-writable.

This is an example:

```
docker run -d \
  -it \
  --name tmpfs_volume \
  --mount type=tmpfs,destination=/secrets,tmpfs-mode=700 \
  nginx
```

A real-world example where using a TMPFS mount is particularly useful is for a web application that handles sensitive data, such as session tokens or temporary encryption keys, which should not be persisted on disk or exposed outside the container's lifecycle.

In this scenario, you can use a TMPFS mount to create a temporary in-memory storage for these sensitive files. The data stored in this TMPFS mount will be lost when the container is stopped, thereby ensuring that sensitive information does not persist on the host system.

Here's an example using a Docker container that might be running a web application:

```
docker run -d \
  -it \
  --name secure_web_app \
  --mount type=tmpfs,destination=/app/secrets \
  my_web_app_image
```

Your application should write these tokens or data to "/app/secrets".

This ensures that:

- The sensitive data is kept away from potentially persistent storage layers.
- The data is automatically cleaned up and not left behind after the container's lifecycle ends.

Using a TMPFS mount in such scenarios enhances the security of the application.

Docker Volume From Containers

The `--volumes-from` flag in Docker is used to mount volumes from one container into another container. This allows you to share data between containers without binding the data to the host machine. When you use `--volumes-from`, the new container gains access to all the volumes defined in the source container (the one you're referencing).

This is an example:

```
docker run -d --name container1 -v /data some-image
docker run -d --name container2 --volumes-from container1 another-image
```

In this example, container2 will have access to the "/data" volume created in container1.

This flag is useful when you want to share data between containers without binding the data to the host machine. It's also useful if you want perform backups of your data by mounting a volume from a container that has the data you want to backup.

Let's see a real-world example. Let's say we have a database container that stores data in a volume. We want to backup this data to another container. We can do this using the `--volumes-from` flag.

This is the first container:

```
docker run -d \
  --name original_mysql \
  -e MYSQL_ROOT_PASSWORD=my-secret-pw \
  -v mysql_data:/var/lib/mysql \
  mysql:5.7
```

`mysql_data` is the name of the Docker volume where the MySQL data is stored in the example above.

Next, create a backup of your MySQL data. You'll use a new container and mount the volume from the original MySQL container.

```
docker run --rm \
  --volumes-from original_mysql \
  -v $(pwd):/backup \
  ubuntu \
  tar cvf /backup/mysql_backup.tar /var/lib/mysql
```

If you type `ls` in the current directory, you'll see the "mysql_backup.tar" file.

Now, to start a new MySQL container using the same volume therefor the same data, you can use the following command:

```
docker run -d \
  --name new_mysql \
  -e MYSQL_ROOT_PASSWORD=my-secret-pw \
  -v mysql_data:/var/lib/mysql \
  mysql:5.7
```

The `new_mysql` container uses the same named volume `mysql_data` to ensure it has access to the same data as the original container.

Docker Logging

How Docker Logs Work

When we say "Docker logs", we are referring to the logs that Docker "generates for containers". In reality, Docker does not generate logs for containers. Instead, it captures the standard output (stdout) and standard error (stderr) streams generated by a container and writes them to a file. By default, Docker writes the logs to the JSON file `/var/lib/docker/containers/<container_id>/<container_id>-json.log`.

Let's see an example. First start a container from the `nginx` image and map port 8080 on the host to port 80 on the container:

```
docker run --name my_container -d -p 8080:80 nginx
```

Then send multiple requests to the container:

```
for i in {1..10}; do curl --silent localhost:8080; done
```

Get the container ID (long version):

```
container_id=$(docker inspect --format='{{.Id}}' my_container)
```

Check the file where Docker writes the logs:

```
tail /var/lib/docker/containers/$container_id/$container_id-json.log
```

In practice, instead of checking the file where Docker writes the logs, you can use the `docker logs` command to view the logs:

```
docker logs my_container
```

You can use the `--tail` option to limit the number of lines returned:

```
docker logs --tail 5 my_container
```

You can use the `--since` option to limit the number of lines returned to those that have been generated since a given timestamp:

```
docker logs --since 2023-08-01T00:00:00 my_container
```

You can use the `--until` option to limit the number of lines returned to those that have been generated until a given timestamp:

```
docker logs --until 2023-08-01T00:00:00 my_container
```

You can use the `--follow` option to follow the logs:

```
docker logs --follow my_container
# Or use the short version
docker logs -f my_container
```

You can use the `--timestamps` option to include timestamps in the logs:

```
docker logs --timestamps my_container
```

You can also use the `--details` option to include extra details in the logs:

```
docker logs --details my_container
```

Finally, you can combine multiple options to get the desired output.

Logging Best Practices and Recommendations

Docker containers are ephemeral, does not store logs inside the container, and should not.

When you create an image for a container, you should not include a logging system inside the container and you should not store the logs inside the container. You don't want to make the image bigger than it needs to be and your image should be portable. Therefore, you should always send the logs to the standard output (stdout) and standard error (stderr) streams.

Let's see an example. Create a Dockerfile with the following content:

```
cat > Dockerfile <<EOF
FROM ubuntu:latest
RUN apt-get update && apt-get install -y nginx
CMD ["nginx", "-g", "daemon off;"]
EOF
```

Build and run the image:

```
docker build -t my_image .
# Remove the old container if it already exists
docker rm -f my_container &>/dev/null || true
docker run --name my_container -d -p 8080:80 my_image
```

Make some curl tests:

```
for i in {1..10}; do curl --silent localhost:8080; done
```

Check the logs:

```
docker logs my_container
```

If you are expecting to see the access logs, you will be disappointed. The access logs are being written to the regular files inside the container, however, they since they are not being redirected to the standard output (stdout) or standard error (stderr) streams, Docker does not capture them.

You can check the files using the following command:

```
docker exec my_container cat  /var/log/nginx/access.log
```

In order to help Docker capture Nginx logs, we should redirect the access log to the standard output (stdout) stream and the error log to the standard error (stderr) stream. We can do that by updating the configuration of Nginx but a simpler and more common way is to add the following lines to the Dockerfile:

```
RUN ln -sf /dev/stdout /var/log/nginx/access.log \
    && ln -sf /dev/stderr /var/log/nginx/error.log
```

This is a common practice and you will find it in many Dockerfiles.

Let's recreate the Dockerfile, build, run, make some tests and see if logs are being captured:

```
# Create the new Dockerfile
cat > Dockerfile <<EOF
FROM ubuntu:latest
RUN apt-get update && apt-get install -y nginx
RUN ln -sf /dev/stdout /var/log/nginx/access.log \
    && ln -sf /dev/stderr /var/log/nginx/error.log
CMD ["nginx", "-g", "daemon off;"]
EOF
# Remove the old container, build and run
docker rm -f my_container &>/dev/null || true
```

```
docker build -t my_image .
docker run --name my_container -d -p 8080:80 my_image
# Make some curl tests
for i in {1..10}; do curl --silent localhost:8080; done
# Check the logs
docker logs my_container
```

You should see the access logs now.

What we saw here is specific to Nginx, however, the same applies to other applications. You should always find the best way for you to redirect the logs to the standard output (stdout) and standard error (stderr) streams.

So, at this point, we have a container that is streaming logs outside the container and every line of logs is stored in "/var/lib/docker/containers/<container_id>/<container_id>-json.log".

```
# Get the container ID (long version)
container_id=$(docker inspect --format='{{.Id}}' my_container)
# Check the file where Docker writes the logs
tail /var/lib/docker/containers/$container_id/$container_id-json.log
```

When we remove the container, the folder inside "/var/lib/docker/containers" that contains the logs for that container is removed.

```
docker rm -f my_container
ls /var/lib/docker/containers/$container_id
```

If you think that storing logs on the host is a good idea, you are wrong. Unless you don't need logs, you should always send them to a centralized logging system. We will see more about this in the next section.

Logging Drivers

Logging drivers are tools and mechanisms that make the collection of log data from running containers and services easy. Each Docker daemon already have a default logging driver, typically the "json-file" driver, which stores logs as JSON. Containers use this default unless configured otherwise, and users have the option to install and use other logging tools and drivers or implement custom logging driver plugins.

You can check the logging driver used by the Docker daemon using the following command:

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

If you have not changed the default logging driver, you should see "json-file". If you want to configure this driver, you should create a file called "daemon.json" in "/etc/docker" and add your configuration there. For example, if you want to limit the size of the log files to 100MB and keep only 10 files, you can add the following lines to "daemon.json":

```
cat >> /etc/docker/daemon.json <<EOF
{
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m",
    "max-file": "10"
  }
}
EOF
```

To start a container with a different logging driver, you can use the `--log-driver` option. Here, for example, we are starting a container with the "none" logging driver which disables any logging:

```
# Remove the old container if it already exists
docker rm -f my_container &>/dev/null || true
docker run --name my_container -p 8080:80 -d --log-driver none nginx
```

You can check the logging driver used by a container using the following command:

```
docker inspect --format='{{.HostConfig.LogConfig.Type}}' my_container
```

Until now, we have seen two logging drivers: "json-file" and "none". There are many other logging drivers available. Here is a list of the most common and also natively supported logging drivers:

Name	Description	Documentation
`none`	Disables any logging.	
`json-file`	Writes JSON messages to file.	json-file[91]
`local`	Writes messages to local storage as they are.	local[92]
`syslog`	Writes messages to the syslog facility.	syslog[93]
`journald`	Writes messages to the systemd journal.	journald[94]

[91] https://docs.docker.com/config/containers/logging/json-file/
[92] https://docs.docker.com/config/containers/logging/local/
[93] https://docs.docker.com/config/containers/logging/syslog/
[94] https://docs.docker.com/config/containers/logging/journald/

Name	Description	Documentation
`gelf`	Writes messages to a GELF endpoint like Graylog or Logstash.	gelf[95]
`fluentd`	Writes messages to a Fluentd endpoint.	fluentd[96]
`awslogs`	Writes messages to Amazon CloudWatch Logs.	awslogs[97]
`splunk`	Writes messages to Splunk using the HTTP Event Collector.	splunk[98]
`etwlogs`	Writes messages as ETW events.	etwlogs[99]
`gcplogs`	Writes messages to Google Cloud Logging.	gcplogs[100]
`logentries`	Writes messages to Rapid7 Logentries.	logentries[101]

Logging Using Loki and Grafana

There are also many third-party logging drivers available. For example, if you are a Grafana user, you can use Loki[102].

This is how to do it, first, install the Loki driver:

```
1 docker plugin install grafana/loki-docker-driver:2.9.1 --alias loki --grant-all-perm\
2 issions
```

Check the installed plugins:

```
1 docker plugin ls
```

Now let's start a Grafana instance using this Docker Compose file[103]:

[95] https://docs.docker.com/config/containers/logging/gelf/
[96] https://docs.docker.com/config/containers/logging/fluentd/
[97] https://docs.docker.com/config/containers/logging/awslogs/
[98] https://docs.docker.com/config/containers/logging/splunk/
[99] https://docs.docker.com/config/containers/logging/etwlogs/
[100] https://docs.docker.com/config/containers/logging/gcplogs/
[101] https://docs.docker.com/config/containers/logging/logentries/
[102] https://grafana.com/oss/loki/
[103] https://github.com/grafana/loki/blob/main/production/docker-compose.yaml

```
# Create a new folder
mkdir -p ~/loki
# Create the docker-compose.yml file
cat > ~/loki/docker-compose.yml <<EOF
version: "3"

networks:
  loki:

services:
  loki:
    image: grafana/loki:2.9.2
    ports:
      - "3100:3100"
    command: -config.file=/etc/loki/local-config.yaml
    networks:
      - loki

  promtail:
    image: grafana/promtail:2.9.2
    volumes:
      - /var/log:/var/log
    command: -config.file=/etc/promtail/config.yml
    networks:
      - loki

  grafana:
    environment:
      - GF_PATHS_PROVISIONING=/etc/grafana/provisioning
      - GF_AUTH_ANONYMOUS_ENABLED=true
      - GF_AUTH_ANONYMOUS_ORG_ROLE=Admin
    entrypoint:
      - sh
      - -euc
      - |
        mkdir -p /etc/grafana/provisioning/datasources
        cat <<EOF > /etc/grafana/provisioning/datasources/ds.yaml
        apiVersion: 1
        datasources:
        - name: Loki
          type: loki
          access: proxy
          orgId: 1
```

```
          url: http://loki:3100
          basicAuth: false
          isDefault: true
          version: 1
          editable: false
        EOF
        /run.sh
    image: grafana/grafana:latest
    ports:
      - "3000:3000"
    networks:
      - loki
EOF
```

Start the Grafana instance:

```
cd ~/loki
docker-compose up -d
```

Visit your machine IP on the port 3000 and login to Grafana using the default credentials (username: admin, password: admin). Let's keep the default login and password as they are for now.

Now, let's configure the driver to send the logs to Loki:

```
# Export the environment variables
export LOKI_INSTANCE_HOST=localhost
export LOKI_INSTANCE_PORT=3100
export LOKI_INSTANCE_USERNAME=admin
export LOKI_INSTANCE_PASSWORD=admin
# Create the daemon.json file
cat > /etc/docker/daemon.json <<EOF
{
  "log-driver": "loki",
  "log-opts": {
    "loki-url": "http://$LOKI_INSTANCE_USERNAME:$LOKI_INSTANCE_PASSWORD@$LOKI_INSTAN\
CE_HOST:$LOKI_INSTANCE_PORT/loki/api/v1/push"
  }
}
EOF
```

If your Loki instance is running on a different machine, a different port, or you have changed the default credentials, you should export the correct values for the environment variables. Here we

are using the login “admin” and the password “admin” because we have not changed the default credentials.

Restart the Docker daemon to apply the changes:

```
systemctl restart docker
```

Start a new Nginx container

```
# Remove the old container if it already exists
docker rm -f my_container &>/dev/null || true
# Start a new container with the Loki driver
docker run --name my_container -p 8080:80 -d --log-driver loki nginx
```

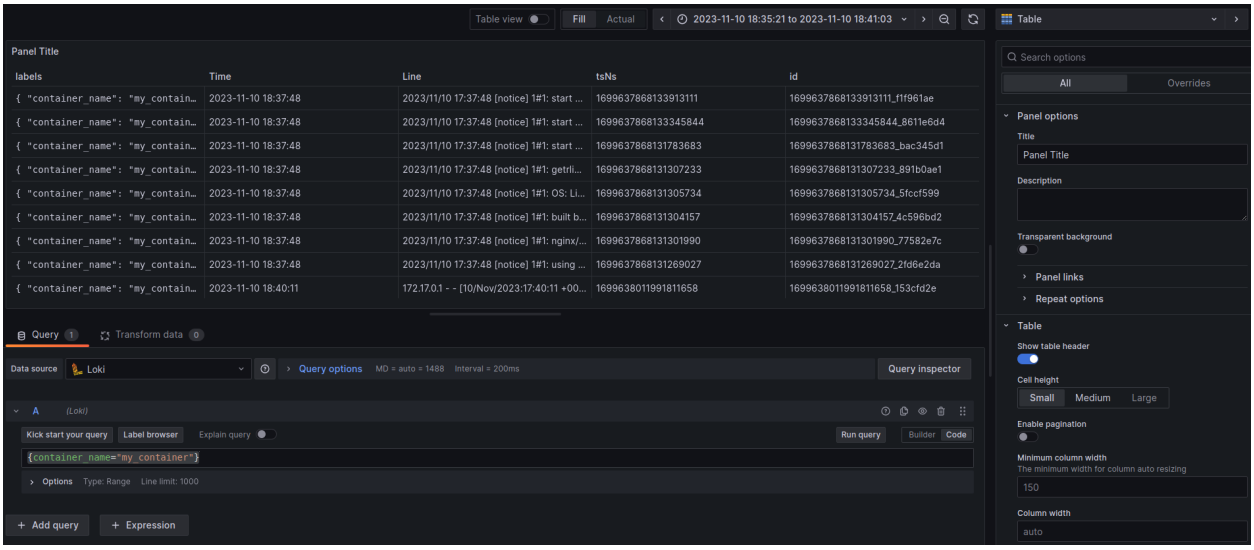
Now, let’s make some curl tests:

```
for i in {1..10}; do curl --silent localhost:8080; done
```

Visit your Grafana instance, create a new dashboard, configure the source to be “Loki”, and add a new query with the following content:

```
{container_name="my_container"}
```

Click on “Run query” and you should see the logs. Select the Table as the visualization type and you should see something like this:



Loki

Logging Using AWS CloudWatch

If you are an AWS user, then you CloudWatch may be a good option for you.

Amazon CloudWatch, as described by AWS, is a service that monitors applications, responds to performance changes, optimizes resource use, and provides insights into operational health. By collecting data across AWS resources, CloudWatch gives visibility into system-wide performance and allows users to set alarms, automatically react to changes, and gain a unified view of operational health.

CloudWatch Logo

In our context, AWS CloudWatch can be used to collect Docker logs and centralize them in one place.

In order to use CloudWatch, you should first create a new IAM user with the following permissions:

- `logs:CreateLogGroup`
- `logs:CreateLogStream`
- `logs:PutLogEvents`
- `logs:DescribeLogStreams`

This is an example of a policy that you can use:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams"
    ],
      "Resource": [
        "arn:aws:logs:*:*:*"
    ]
  }
 ]
}
```

Go to AWS console / CloudWatch, create a log group and call it `my-group`. Click on the created group and create a new stream, call it `my-stream`.

You should create a new file where you store your AWS credentials:

```
mkdir -p /etc/systemd/system/docker.service.d/
touch /etc/systemd/system/docker.service.d/aws-credentials.conf
```

Export your AWS credentials:

```
export AWS_ACCESS_KEY_ID=[YOUR_AWS_ACCESS_KEY_ID]
export AWS_SECRET_ACCESS_KEY=[YOUR_AWS_SECRET_ACCESS_KEY]
```

Now add these lines:

```
cat <<EOF > /etc/systemd/system/docker.service.d/aws-credentials.conf
[Service]
Environment="AWS_ACCESS_KEY_ID=$AWS_ACCESS_KEY_ID"
Environment="AWS_SECRET_ACCESS_KEY=$AWS_SECRET_ACCESS_KEY"
EOF
```

Now restart Docker:

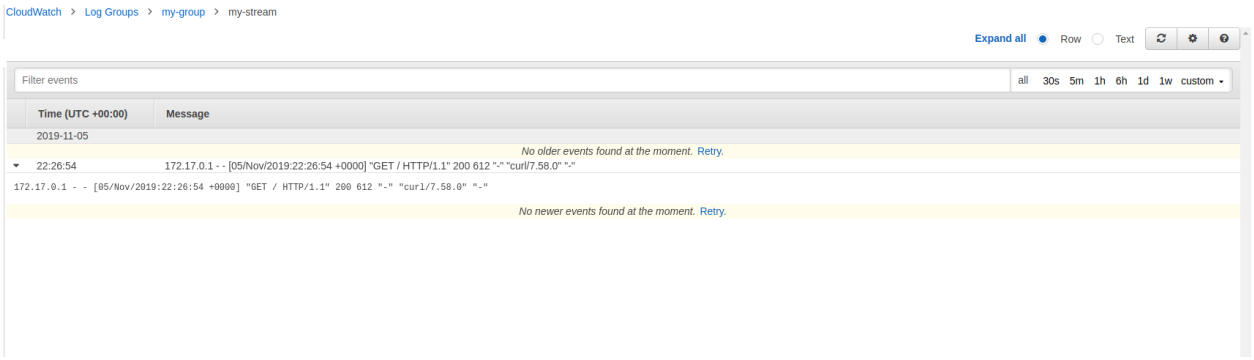
```
sudo systemctl daemon-reload && sudo service docker restart
```

Let's create a new container and use the AWS logging driver:

```
# Remove the old container if it already exists
docker rm -f webserver;

# Create a new container with the AWS logging driver
docker run -it --log-driver="awslogs" \
--log-opt awslogs-region="eu-central-1" \
--log-opt awslogs-group="my-group" \
--log-opt awslogs-stream="my-stream" \
-p 8000:80 -d --name webserver nginx
```

Execute `curl http://0.0.0.0:8000` and go back to the AWS console; you should see a new logline:



"Container logs on AWS CloudWatch"

If you want to use AWS logging driver with all of the other containers, you should start the Docker daemon using `--log-driver=awslogs`:

```
dockerd --log-driver=awslogs
```

You can also use awslogs driver or any other log driver in a Docker Compose file:

```yaml
version: "3.9"
services:
  web:
    image: nginx
    ports:
      - "8000:80"
    logging:
      driver: "awslogs"
      options:
        awslogs-region: "eu-central-1"
        awslogs-group: "my-group"
        awslogs-stream: "my-stream"
```

Docker Daemon Logging

The Docker daemon logs information about its operations. These logs are essential for troubleshooting, monitoring, and ensuring the smooth running of Docker containers and services. The nature and location of these logs can vary based on the operating system and Docker configurations. The following shows the default log locations for the Docker daemon on different operating systems:

Linux: Use `journalctl -xu docker.service`. Alternatively, check the following directories depending on the Linux distribution:

- `/var/log/syslog`
- `/var/log/daemon.log -/var/log/messages`

macOS (dockerd logs): `~/Library/Containers/com.docker.docker/Data/log/vm/dockerd.log`

macOS (containerd logs): ‘~/Library/Containers/com.docker.docker/Data/log/vm/containerd.log

Windows (WSL2) (dockerd logs): `%LOCALAPPDATA%\Docker\log\vm\dockerd.log`

Windows (WSL2) (containerd logs): `%LOCALAPPDATA%\Docker\log\vm\containerd.log`

Windows (Windows containers): Logs are available in the Windows Event Log.

We are using Ubuntu in this guide, so we can check the logs using the following command:

```
journalctl -xu docker.service
```

Sometimes, you may want to increase the log level of the Docker daemon. You can do that by updating the “daemon.json” file. For example, if you want to set the log level to “debug”, you can add the following lines to “daemon.json”:

```
cat > /etc/docker/daemon.json <<EOF
{
  "log-level": "debug"
}
EOF
```

Reload the Docker daemon to apply the changes:

```
systemctl reload docker
# or sudo kill -SIGHUP $(pidof dockerd)
```

Debugging can be very useful, however, you should not keep the log level to “debug” all the time. It will generate a lot of logs and it may slow down the Docker daemon. You should only increase the log level when you need to debug something and then set it back to “info” or “warn”.

Docker Networks

Like virtual machines (VMs), a Docker container can be attached to one or more networks. This allows containers to communicate with each other, as well as with external systems.

After installing Docker, you might notice a new network interface on your host machine. You can view this using the ifconfig (or ip a on newer systems) command:

```
ifconfig
```

You should see a new interface named docker0. This is the default bridge network interface that Docker creates when you install it.

```
docker0   Link encap:Ethernet  HWaddr 02:42:ef:e0:98:84
          inet addr:172.17.0.1  Bcast:0.0.0.0  Mask:255.255.0.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:5 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:356 (356.0 B)  TX bytes:0 (0.0 B)
```

By default, Docker provides three network types: bridge, host, and none. You can list these using the docker network ls command:

```
docker network ls
```

Docker's network system generates several events which can be useful for monitoring or troubleshooting. These events include `create`, `connect`, `destroy`, `disconnect`, and `remove`.

Docker Networks Types

When you type the following command, you will see the list of networks that Docker has created on your system:

```
docker network ls
```

You should see the following output (the output may vary depending on your system but the network names will be the same):

```
NETWORK ID          NAME                DRIVER              SCOPE
62a68d83996b        bridge              bridge              local
16edb70ec4c9        host                host                local
4f24136a4691        none                null                local
```

The above 3 networks are the default networks that Docker creates when you install it.

In the next sections, we will see what each network type is and how to use it.

The (System) Bridge Network

The bridge network, as said, is created by default when you install Docker. It is a private internal network created by Docker on the host and is used for communication between containers on the same host.

When you run a container, it is attached to the bridge network by default:

```
# Run a container
docker run -d --name nginx_server nginx
# Check the networks that the container is attached to
docker inspect nginx_server -f "{{json .NetworkSettings.Networks }}" | jq
```

We used `jq` command to format the output. If you don't have `jq` installed, you can remove it from the command or install it[104].

The output should show the list of networks that the container is attached to and here we should see one network that is the bridge network.

This is an example:

```
{
  "bridge": {
    "IPAMConfig": null,
    "Links": null,
    "Aliases": null,
    "NetworkID": "7af5d7cbd0c102cb55f7f0c62dca694ccfc6ab1f2c7602177119a17eb3b6b1ab",
    "EndpointID": "7a9187f8a5feaf19aa46544d4ed3fe189e4475341e74f2d5b92846e09e52d828",
    "Gateway": "172.17.0.1",
    "IPAddress": "172.17.0.2",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
    "GlobalIPv6Address": "",
```

[104] https://jqlang.github.io/jq/download/

```
    "GlobalIPv6PrefixLen": 0,
    "MacAddress": "02:42:ac:11:00:02",
    "DriverOpts": null
  }
}
```

The default bridge mode allows containers to communicate with each other using IP addresses.

```
# create two containers busybox1 and busybox2 that are attached to the default bridg\
e network
docker run -it -d --name busybox1 busybox
docker run -it -d --name busybox2 busybox
# Get the IP of busybox1 and busybox2
IP_BUSYBOX1=$(docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{\
end}}' busybox1)
IP_BUSYBOX2=$(docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{\
end}}' busybox2)
# Ping busybox1 from busybox2 and vice versa using the IP address
docker exec -it busybox2 ping -c1 $IP_BUSYBOX1
docker exec -it busybox1 ping -c1 $IP_BUSYBOX2
# Ping busybox1 from busybox2 and vice versa using the container name
docker exec -it busybox1 ping -c1 busybox2
docker exec -it busybox2 ping -c1 busybox1
```

Pinging using the address should work but pinging using the container name should fail.

The (User) Bridge Network

In addition to the default bridge network that I called the system bridge network, you can create your own bridge network and attach containers to it (user bridge network).

To create a bridge network, you can use the following command:

```
docker network create --driver bridge my_bridge_network
# or docker network create -d bridge my_bridge_network
# or docker network create my_bridge_network (bridge is the default driver)
```

Make sure the network name is unique. You can list the networks using the `docker network ls` command.

Now update the busybox1 and busybox2 containers to attach them to the new bridge network:

```
docker network connect my_bridge_network busybox1
docker network connect my_bridge_network busybox2
```

Try to ping the containers using the container name:

```
docker exec -it busybox1 ping -c1 busybox2
docker exec -it busybox2 ping -c1 busybox1
```

You should see that the ping command works fine. What should we conclude from this?

The user bridge network allows containers to communicate with each other using the container name as well as the IP address (you can test this) unlike the system bridge network that allows containers to communicate with each other using the IP address only.

To get more information about the user bridge network, you can use the `docker network inspect` command:

```
docker network inspect my_bridge_network
```

This is an example of an output:

```
[
    {
        "Name": "my_bridge_network",
        "Id": "93b80b38b3e749d7aa448d60661c9ce3e21be2413b5ebe30231eb3f8e5e9fdf6",
        "Created": "2023-11-08T08:49:56.110636436Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
```

```
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

You can see different information about this bridge network like:

- The IP address range that the bridge network uses: `172.18.0.0/16`. This subnet notation means that the network can accommodate IP addresses from 172.18.0.0 to 172.18.255.255.
- The default gateway for the containers connected to this bridge network: `172.18.0.1`.

You can also check the same interface using the `ifconfig` command:

```
ifconfig
```

You will see one or more network interfaces named `br-<network_id>` (e.g., `br-93b80b38b3e7`). This interface acts as the default gateway for the containers connected to that bridge network. In other words, containers use this gateway IP to communicate with the outside world. When a container wants to send traffic outside its own network (e.g., to the internet or another network), it sends the traffic to this gateway.

Docker allows customizing the IP address range and the gateway IP address when creating a bridge network. For example, you can create a bridge network with the following command:

```
docker network create --driver bridge --subnet=192.168.1.0/24 --gateway=192.168.1.1 \
my_custom_network
```

Create the network, then create this container using the image `tutum/dnsutils`:

```
docker run -it -d --network my_custom_network --name test_container tutum/dnsutils
```

From the container, try to check the IP address and the default gateway:

```
docker exec -it test_container ifconfig
docker exec -it test_container ip route
```

You will find that the IP address of the container is in the range `192.168.1.0/24` and the default gateway is `192.168.1.1`.

The Host Network

When you use the host network, any container attached to it will use the host's network directly. This means that the container will not have its own IP address and it will use the host's IP address directly. If there's a port exposed on the container, it will be opened on the host as well.

Let's see this through an example. First, create a container attached to the host network:

```
docker run -it -d --name nginx_host_network --network host nginx
```

Notice that we didn't specify the port mapping. This is because the container will use the host's network directly.

Now, check the ports that are opened on the host:

```
sudo netstat -tulpn | grep "/nginx:"
```

You should see an output similar to this:

```
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      1292\
4/nginx: master
tcp6       0      0 :::80                   :::*                    LISTEN      1292\
4/nginx: master
```

Or this if you don't have IPv6 enabled:

```
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      1292\
4/nginx: master
```

Test the connection to the container:

```
curl localhost:80
```

You should see the default nginx page.

The None Network

Some containers don't need to be attached to any network since they are, for instance, used to run a script or a command, process data or some background tasks, etc.

In this specific case, it is better to attach the container to the none network. This will prevent the container from accessing the network and it will be isolated from other containers and the host.

This is an example of a container that runs a script to print the size of the storage used by dangling volumes:

```
docker run \
--network none \
--rm \
-v /var/run/docker.sock:/var/run/docker.sock \
-v /var/lib/docker/volumes/:/var/lib/docker/volumes/ \
eon01/dvsc:latest
```

The executed script has no need to access other containers, the host network or the internet, it has no need to be accessible from other containers, the host or the internet. Therefore, it is better to attach it to the none network.

The Macvlan Network

Docker's Macvlan network allows a Docker container to have its own MAC address and IP address on your network, just as if it were a physical device plugged into your network. This makes the container appear and function as a regular device on the network, separate from the host.

Imagine you have a physical room with various devices: computers, printers, etc. Each device has its own unique identity (MAC address) and can be given its own address (IP address) in the room. Now, think of the Macvlan network as giving a Docker container its very own identity and address in that room, even though it's technically running inside one of the computers. The other devices in the room see this container just like any other device.

Let's see this with an example. We want to create a Docker Macvlan network so that our containers can get their own IP addresses from our local network and appear just like any other physical device on the network.

To do this, start by identifying the physical network interface on your Docker host through which the traffic will pass. This is often "eth0", but it might differ depending on your setup. You can use the `ifconfig` command to list your network interfaces.

Now create a Macvlan network:

```
docker network create \
    --driver macvlan \
    --subnet=192.168.3.0/24 \
    --gateway=192.168.3.1 \
    -o parent=eth0 \
    pub_net
```

- The `--subnet=192.168.3.0/24` specifies the IP range that the containers will use. It should be aligned with your local network range.

- The `--gateway=192.168.3.1` is the gateway that containers will use to access external networks. Typically, this would be your router's IP address.
- The `-o parent=eth0` is the network interface on the Docker host that this Macvlan network will use to send and receive packets. If your host uses a different interface (like "ens33" or "wlp0s20f3"), replace "eth0" with that.
- `pub_net` is the name of the network.

Now you can run a container attached to the Macvlan network:

```
docker run --network=pub_net -it ubuntu /bin/bash
```

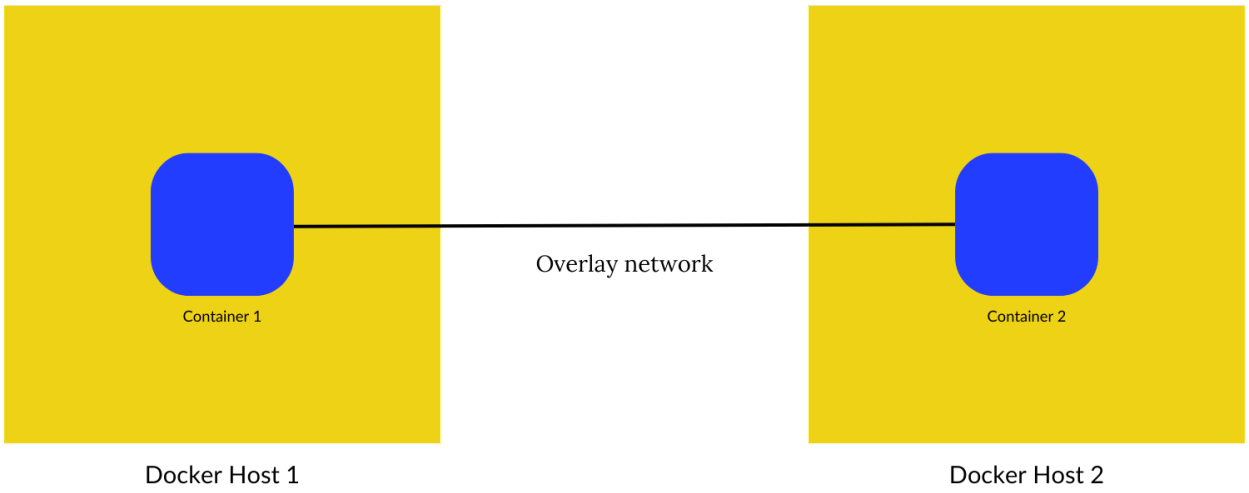
Inside the container, you can use the `ifconfig` command to see the IP address that the container has received from the local network:

```
apt update
apt install net-tools
ifconfig
```

The Overlay Network

Overlay networks are used to connect multiple Docker daemons together. Two standalone containers on different Docker daemons (usually different hosts) can be connected using an overlay network. This network type uses VXLAN technology to encapsulate OSI layer 2 Ethernet frames within layer 4 UDP datagrams, allowing the creation of layer 2 overlay networks across multiple physical hosts.

This strategy removes the need to do OS-level routing between these containers.



overlay network

It is usually used with Docker Swarm to enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container.

We have seen Docker Swarm yet, but we will see it in one of the next sections. In the example we'll use Swarm to create an overlay network and attach containers to it.

You need to have two machines (at least) to create an overlay network:

- machine1
- machine2

You can use two virtual machines.

Docker should be installed on both machines.

On the fist machine, initialize the swarm:

```
1 docker swarm init --advertise-addr [MACHINE1_IP]
```

Change `[MACHINE1_IP]` with the IP address of the first machine. If both machines are on the same network (which is the recommended case), you can use the local IP address instead of the public IP address.

You should see an output similar to this:

```
docker swarm join --token [TOKEN] [MACHINE1_IP]:2377
```

Copy the output and run it on the second machine to join the swarm.

Go back to the first machine and create an overlay network:

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

> **i** Usually, when we run a Swarm cluster, we run services and not containers. However, in our case, we will run standalone containers to see how overlay networks work. That's why we created an attachable overlay network (`--attachable`). In this case, we can attach standalone containers that are managed by us and not by Swarm to the overlay network that is managed by Swarm.

Now, run containers attached to the overlay network:

```
# Run a container attached to the overlay network
docker run -it -d --name busybox1 --network my_overlay_network busybox
docker run -it -d --name busybox2 --network my_overlay_network busybox
# Ping busybox1 from busybox2 and vice versa using the container name
docker exec -it busybox1 ping -c1 busybox2
docker exec -it busybox2 ping -c1 busybox1
```

You should see that the ping command works fine.

Now, on machine2, run the following command to create a third container attached to the overlay network:

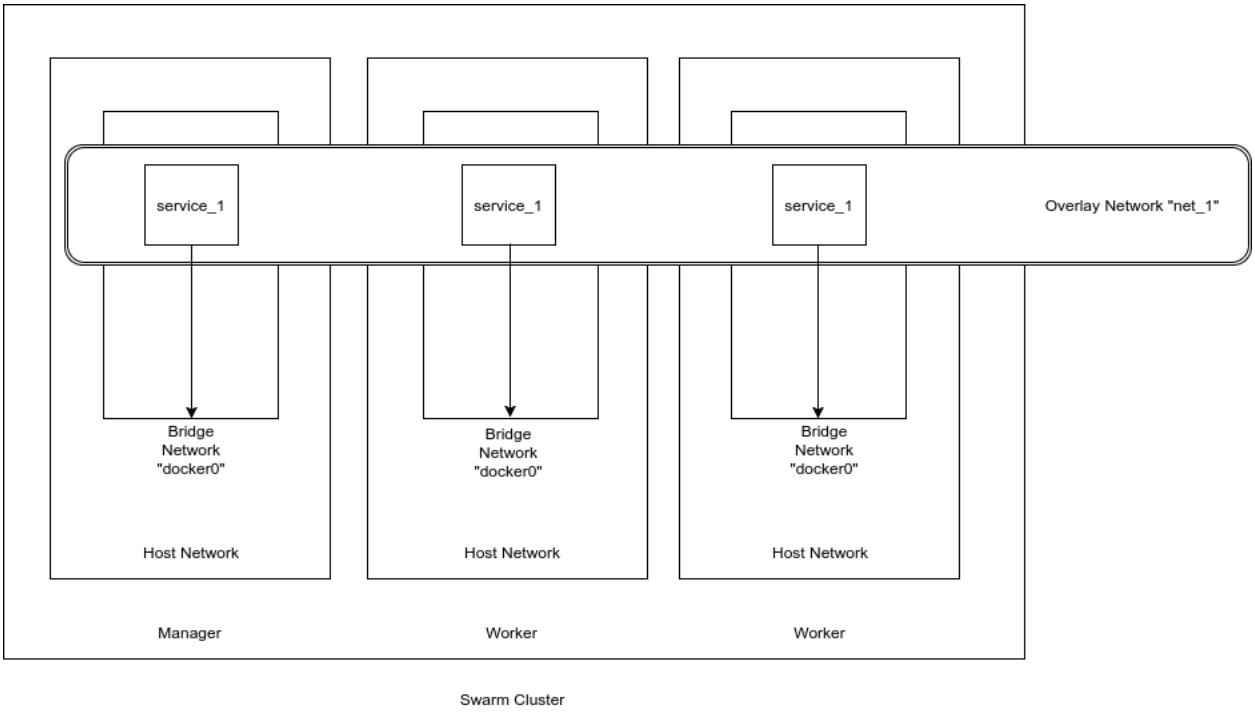
```
docker run -it -d --name busybox3 --network my_overlay_network busybox
```

Ping busybox1 and busybox2 from busybox3 and vice versa:

```
# On machine2
docker exec -it busybox3 ping -c1 busybox1
docker exec -it busybox3 ping -c1 busybox2
# On machine1
docker exec -it busybox1 ping -c1 busybox3
docker exec -it busybox2 ping -c1 busybox3
```

You should see that the ping command works fine even of the containers are on different machines. Through the overlay network, the containers can communicate with each other as if they were on the same machine and this is the main purpose of the overlay network.

Usually, we use overlay networks with Docker Swarm services instead of standalone containers. But the example above shows how overlay networks work.

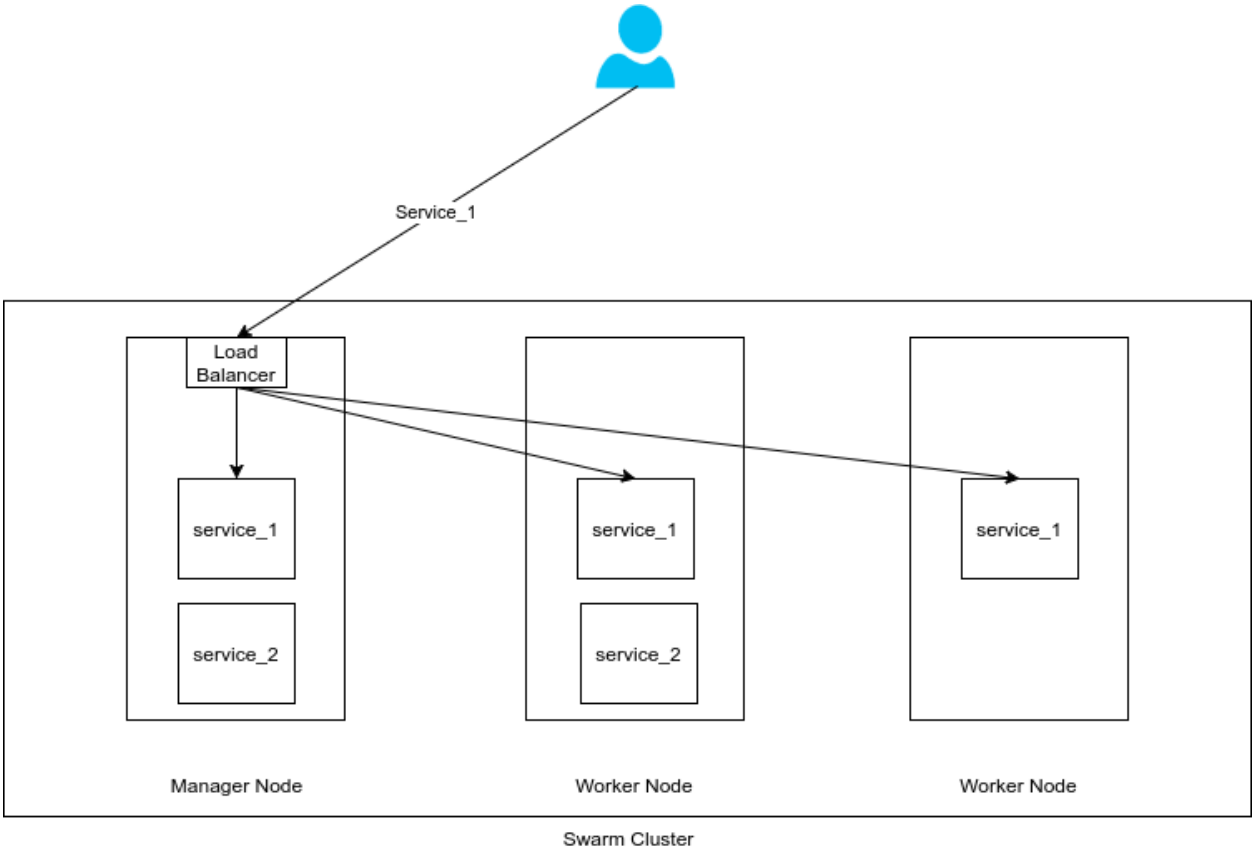


Swarm Service Networking

The Ingress Network

The ingress network is a special network that is automatically created when you initialize a swarm. However, there's a difference between the ingress network and the other overlay networks that you create.

The ingress handles the routing of incoming traffic to the appropriate service within the swarm cluster.



Load Balancing In Swarm Cluster

One of the key features of the ingress network is the routing mesh. With this, if a service's port is published (like port 80), you can make a request to any node in the swarm on that port, and Docker will route the request to an active container of the service, even if that container is not on the node you initially connected to. This provides a level of abstraction and ensures high availability.

These concepts will be clearer when we see Docker Swarm.

Docker Links

Docker links are a legacy feature of Docker. They are used to connect two containers together. However, they are not recommended to be used anymore. Instead, you should use user-defined networks.

Let's see an example of how to use Docker links.

First, create a container named `alpine1`:

```
1 docker run -it -d --name alpine1 alpine
```

Now, create a second container named `alpine2` and link it to `alpine1`:

```
docker run -it -d --name alpine2 --link alpine1 alpine
```

Now, check the environment variables of `alpine2`:

```
docker exec -it alpine2 env
```

You should see an output similar to this:

```
[...]
ALPINE1_NAME=/alpine2/alpine1
[...]
```

The `ALPINE1_NAME` environment variable is created automatically by Docker and it contains the name of the linked container.

Now, try to ping `alpine1` from `alpine2`:

```
docker exec -it alpine2 ping -c1 alpine1
```

You should see that the ping command works fine even if the containers are not attached to a user-defined bridge network.

As a reminder, containers by default are attached to the default bridge network which doesn't allow containers to communicate with each other using the container name. However, Docker links are a workaround to this limitation.

Docker Compose

What is Docker Compose and Why Should I Care?

Running containers individually has its merits, but consider a scenario where we want to orchestrate a LAMP or LEMP stack. Instead of manually starting a PHP container and then initializing the webserver container to link them, there's a more streamlined approach.

Enter Docker Compose: a tool that allows us to define and run multi-container applications. By defining a declarative YAML file, commonly referred to as a Compose file, we can specify our entire stack configuration. With just one command, Docker Compose enables us to effortlessly spin up multiple interconnected containers, making the deployment of linked services like a LAMP or LEMP stack easier than ever.

In addition to being a mini orchestration tool, Docker Compose also provides a convenient environment for local development. By defining our application's configuration in a Compose file, a developer can spin up a local environment that matches the production stack with a single command. This allows us to test our application in a production-like environment without having to deploy it to a production environment.

When you are developing an application in a Docker container, apart from the fact that you need to test it in an environment that is as close as possible to the production environment, you need to have an auto-reload feature that will reload the application when a change is detected. This is where Docker Compose comes in handy.

Imagine you are developing a PHP application running inside an Apache container, you need to have a way to reload the application when a change is detected. You may think of creating a volume that will be mounted to the container, then on the host, you will open your IDE and start editing the files, but this is not the best way to do it. The problem with this approach is that you need to restart the container every time you make a change, and this is not very practical.

Therefore, you need to have a way to automatically reload the application when a change is detected without restarting the container. This is where Docker Compose comes in handy.

Additionally, Docker Compose offers several benefits. It enables you to version and share your application, collaborate with other developers, run it on various environments, and distribute it.

To summarize, Docker Compose is a user-friendly tool that allows you to define your application using a YAML file. With a single command, you can create and start all the services based on your configuration. It also provides an auto-reload feature, making it convenient for running your application in a development environment.

Installing Docker Compose

In the official documentation, Docker recommends installing Docker Compose by installing Docker Desktop. This tool is available for Windows, Mac, and Linux.

However, there's an alternative way to install Docker Compose on Linux systems (docker-compose-plugin). This is a plugin that allows you to install Docker Compose as a Docker plugin.

Example, if you are using Ubuntu or Debian, you can install Docker Compose using the following command:

```
apt-get install docker-compose-plugin
```

Other distributions are supported, you can check the official documentation[105] for more details.

In the next sections, we are going to learn Docker Compose by example.

Before proceeding, make sure that you quit the Swarm mode (activated in one of the previous chapters). Let's also remove all the containers to start with a clean environment:

```
# Remove all containers
docker rm -f $(docker ps -qa)
# Quit swarm mode
docker swarm leave --force
```

Understanding Docker Compose and How it Works

Let's start by creating a new folder and a new file called `docker-compose.yml`:

```
# Create a new folder
mkdir wordpress && cd wordpress
# Create a new file called docker-compose.yml
cat << EOF > docker-compose.yml
version: '3.9'

services:
   db:
     image: mysql:5.7
     volumes:
       - db_data:/var/lib/mysql
     restart: always
```

[105] https://docs.docker.com/compose/install/linux/#install-using-the-repository

```
13      environment:
14        MYSQL_ROOT_PASSWORD: mypassword
15        MYSQL_DATABASE: wordpress
16        MYSQL_USER: user
17        MYSQL_PASSWORD: mypassword
18
19    wordpress:
20      image: wordpress:latest
21      ports:
22        - "8000:80"
23      restart: always
24      environment:
25        WORDPRESS_DB_HOST: db:3306
26        WORDPRESS_DB_USER: user
27        WORDPRESS_DB_PASSWORD: mypassword
28  volumes:
29      db_data:
30  EOF
```

Notice that we are using the version 3.9 of Docker Compose specification (the latest version at the time of writing this article). There is a difference between Docker Compose versions which is the version of the package[106] used, the version of the specification which refers to the syntax of the `docker-compose.yml` file and the Docker Engine version. You can check the compatibility matrix[107] for more details.

Now, let's start the application:

```
1  # Make sure you are in the wordpress folder
2  docker-compose up
```

You should start seeing the logs of the containers:

- db
- wordpress

The two containers are linked together, so the `wordpress` container can access the `db` container using the hostname `db`.

In reality, `db` and `wordpress` are the names of the services, not the containers. If you run `docker ps`, you will see that the names of the containers are:

- wordpress_db_1

[106] https://github.com/docker/compose/releases
[107] https://docs.docker.com/compose/compose-file/compose-versioning/#compatibility-matrix

- wordpress_wordpress_1

This is the format: `foldername_servicename_number`. The same format is used for any other resources created by Docker Compose (networks, volumes, ...etc).

When we use Docker Compose, we are no longer using just containers, we are using services. A service is a set of 1 or more containers that are managed by Docker Compose. A service can have multiple containers, but in our case, we have one container per service.

Note that the file defining the services is called `docker-compose.yml`. This is the default name, but you can use any name you want. If you want to use a different name, you can use the `-f` option:

```
1 docker-compose -f mydocker-compose.yml up
```

If you want to pause the stack, use `docker-compose pause`. This will pause all the containers in the stack. If you want to pause a specific service, use `docker-compose pause servicename`.

If you want to unpause the stack, use `docker-compose unpause` or `docker-compose start`. This will unpause all the containers in the stack. If you want to unpause a specific service, use `docker-compose unpause servicename` or `docker-compose start servicename`.

We can also use `docker-compose stop` to stop the stack. This will stop all the containers in the stack. If you want to stop a specific service, use `docker-compose stop servicename`.

Finally, we can remove everything using `docker-compose down`. This will stop and remove all the containers in the stack and delete their data.

Docker Compose Dependencies

In the previous example, we have two services: `db` and `wordpress`. The `wordpress` service, in reality, depends on the `db` service. This means that the `wordpress` service cannot start until the `db` service is up and running. If we start the `wordpress` service without starting the `db` service, we will get an error.

Therefor, we need to start the `db` service first, then start the `wordpress` service. In Docker Compose, we can do this using the `depends_on` option:

```
cat << EOF > docker-compose.yml
version: '3.9'

services:
   db:
     image: mysql:5.7
     volumes:
       - db_data:/var/lib/mysql
     restart: always
     environment:
       MYSQL_ROOT_PASSWORD: mypassword
       MYSQL_DATABASE: wordpress
       MYSQL_USER: user
       MYSQL_PASSWORD: mypassword

   wordpress:
     depends_on:
       - db
     image: wordpress:latest
     ports:
       - "8000:80"
     restart: always
     environment:
       WORDPRESS_DB_HOST: db:3306
       WORDPRESS_DB_USER: user
       WORDPRESS_DB_PASSWORD: mypassword
volumes:
    db_data:
EOF
```

Creating Portable Docker Compose Stacks

Docker Compose allows you to create portable stacks. This means that you can create a stack on your local machine and run it on another machine without any changes. However, the words "portable" and "without any changes" are not 100% accurate. There are some things that you need to consider when creating a portable stack.

Consider a Django application that needs two configurations:

- Database configuration: this includes the database host, the database name, the database user, and the database password..etc.
- Allowed hosts: this is a list of hosts that are allowed to access the application.

These configurations are usually defined in a file called `settings.py`. These configurations can be different from one environment to another (development, staging, production, ...etc). In order to make the application portable, we need to make sure that these configurations are defined in the `docker-compose.yml` file and not in the `settings.py` file.

This is how to do it:

- Use environment variables to define the database configuration. This is what we did in the previous example.
- Use environment variables to define the allowed hosts.
- Consume these environment variables in the `settings.py` file.

If you have other configurations of the same type, you need to do the same thing.

This is an example:

Start by creating a new folder, install the required packages and create a new Django project:

```
mkdir django && cd django && apt install python3-pip -y && pip install Django==3.2.4\
 && django-admin startproject mysite && cd mysite && cat << EOF > requirements.txt
Django==3.2.4
psycopg2==2.8.6
EOF
```

Create the Dockerfile:

```
cat << EOF > Dockerfile
FROM python:3.9
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1
WORKDIR /code
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
EOF
```

Then the Docker Compose file:

```
cat << EOF > docker-compose.yml
version: '3.9'

services:
    web:
        build: .
        ports:
            - "8000:8000"
        volumes:
            - .:/code
        environment:
            - POSTGRES_DB=postgres
            - POSTGRES_USER=postgres
            - POSTGRES_PASSWORD=postgres
            - ALLOWED_HOSTS=*
        depends_on:
            - db
    db:
        image: postgres
        environment:
            - POSTGRES_DB=postgres
            - POSTGRES_USER=postgres
            - POSTGRES_PASSWORD=postgres
EOF
```

Add the following content at the end of the `mysite/settings.py` file:

```
cat << EOF >> mysite/settings.py
import os
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.environ.get('POSTGRES_DB'),
        'USER': os.environ.get('POSTGRES_USER'),
        'PASSWORD': os.environ.get('POSTGRES_PASSWORD'),
        'HOST': 'db',
        'PORT': 5432,
    }
}
ALLOWED_HOSTS = os.environ.get('ALLOWED_HOSTS').split(' ')
EOF
```

Start the application:

```
docker compose up --build
```

Whenever you need to change the allowed hosts or a database configuration, all you need to do is to change the `docker-compose.yml` file and restart the application.

In summary, environment variables can be used to make the application portable. However, there are other ways to make the application portable. For example, you can use a configuration file that will be mounted to the container. This file can be different from one environment to another. You can also use a secret management tool such as HashiCorp Vault.

Docker Compose Logging

Docker containers tend to redirect their logs to the standard output (stdout) and standard error (stderr). This is the same behavior for Docker Compose. If you run `docker-compose up`, you will see the logs of the containers in the terminal.

However, we usually need to run Docker Compose services in the background (detached mode). In this case, we need to use the `-d` option:

```
docker compose up -d
```

In this case, we will not see the logs of the containers. To see the logs, we need to use the `logs` command:

```
# Check the logs of all the containers
docker compose logs
# Follow the logs of all the containers
docker compose logs -f
# Check the logs of a specific container
docker compose logs db
# Check the logs of a specific container and follow the logs
docker compose logs -f db
# Show the last 10 lines of the logs of a specific container
docker compose logs --tail=10 db
```

Understanding Docker Compose Syntax

If we want to run a Wordpress site using Docker, we need to run two containers: one for the database and one for the Wordpress application. We can do this using the following commands:

```
# Create a network for the containers
docker network create wordpress

# Create a volume for the database
docker volume create db_data

# Create a container for the database
docker run -d \
  --name db \
  -v db_data:/var/lib/mysql \
  -e MYSQL_ROOT_PASSWORD=mypassword \
  -e MYSQL_DATABASE=wordpress \
  -e MYSQL_USER=user \
  -e MYSQL_PASSWORD=mypassword \
  --restart always \
  --network wordpress \
  mysql:5.7

# Create a container for the Wordpress application
docker run -d \
  --name wordpress \
  -p 8000:80 \
  -e WORDPRESS_DB_HOST=db:3306 \
  -e WORDPRESS_DB_USER=user \
  -e WORDPRESS_DB_PASSWORD=mypassword \
  --restart always \
  --network wordpress \
  wordpress:latest
```

As you can see, we need to run 4 commands to create the two containers. The idea of is to simplify this process by allowing us to define the configuration of the containers in a YAML file.

What a user should do here is "translate" the commands above to a YAML file.

Let's start by the first command:

```
docker network create wordpress
```

Here, we are creating a network called `wordpress`. In Docker Compose, there is no need to define a network, it will be created automatically. All services in a single Compose file will be connected to the same network that will be created automatically.

Now, let's move to the second command:

```
docker volume create db_data
```

Here, we are creating a volume called `db_data`. In Docker Compose, to define a volume, we need to use `volumes:`

```
volumes:
    db_data:
```

Under the name of the volume, we can add other options such as the driver, the driver options, ...etc. In our case, we are using the default driver, so we don't need to specify it.

Now, let's move to the third and fourth commands. Both create containers, so we need to use `services:`

```
services:

   # Database service definition
   db:
     # Use MySQL version 5.7 as the base image
     image: mysql:5.7

     # Mount the named volume 'db_data' to persist the database data
     volumes:
       - db_data:/var/lib/mysql

     # Always restart the container if it stops
     restart: always

     # Set environment variables used by the MySQL image
     environment:
       # Password for the root user
       MYSQL_ROOT_PASSWORD: mypassword
       # Name of the default database to be created
       MYSQL_DATABASE: wordpress
       # Create a new user with the name 'user'
       MYSQL_USER: user
       # Password for the newly created 'user'
       MYSQL_PASSWORD: mypassword

   # WordPress service definition
   wordpress:

     # This service depends on the db service
```

```
30      depends_on:
31        - db
32
33      # Use the latest version of WordPress as the base image
34      image: wordpress:latest
35
36      # Map port 8000 on the host to port 80 inside the container
37      ports:
38        - "8000:80"
39
40      # Always restart the container if it stops
41      restart: always
42
43      # Set environment variables used by the WordPress image
44      environment:
45        # Database host. The name corresponds to the db service and port 3306.
46        WORDPRESS_DB_HOST: db:3306
47        # Database user. This should match the user created in the db service.
48        WORDPRESS_DB_USER: user
49        # Database password. This should match the password set for the user in the d\
50  b service.
51        WORDPRESS_DB_PASSWORD: mypassword
```

Now, we have a `docker-compose.yml` file that defines the same services as the commands above.

The configurations used in Docker Compose do not exactly look the options used when creating and running standalone containers but they are not very different. For example, when we create a container, we use the `-p` option to map a port on the host to a port inside the container. In Docker Compose, we use the `ports` option. The same applies to other options such as `-e` and `environment`, `-v` and `volumes`, ...etc.

We are going to see more examples in the next sections and understand the syntax of Docker Compose.

Using Dockerfile with Docker Compose

In the previous example, we used the `wordpress` image to create the `wordpress` service. However, we can use a Dockerfile to build a custom image and use it in Docker Compose.

Let's see an example. Start by creating a folder called `apache_1` and a new file called `Dockerfile`:

```
# Create a new folder
mkdir apache_1 && cd apache_1
# Create a new file called Dockerfile
cat << EOF > Dockerfile
FROM ubuntu:20.04
ARG DEBIAN_FRONTEND=noninteractive
RUN apt-get update && apt-get install -y apache2
RUN echo "Hello World. I'm Apache" > /var/www/html/index.html
EXPOSE 80
CMD apachectl -D FOREGROUND
EOF
```

Create the Docker Compose file:

```
cat << EOF > app12/docker-compose.yml
version: '3.9'

services:
  web:
    build: .
    ports:
      - "80:80"
EOF
```

Ad you can see, instead of using the `image` option, we are using the `build` option. This option takes the path to the Dockerfile as a value.

To run the application, use the following command:

```
docker compose build
```

Or, build without using the cache:

```
docker compose --build --no-cache
```

Now, start the application:

```
docker compose up -d
```

We could also use the following command to start the application and build the image in the same command:

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

To force the full recreation of the containers, use the `--force-recreate` option:

```
docker compose up -d --force-recreate
```

Docker Compose with Bind Mounts

It is common to use bind mounts when developing an application. This is an example:

```
version: '3.9'

services:
  web:
    build: .
    ports:
      - "80:80"
    volumes:
      - ./html:/var/www/html
```

The developer will create a folder called `html` and put the files of the application inside it. When the container starts, the files will be copied to the `/var/www/html` folder inside the container. There is no need to edit files inside the container, the developer will edit the files on the host and the changes will be reflected inside the container automatically.

Creating Custom Networks

By default a network is created for all the services in a single Compose file. However, we can create a custom network and use it in the Compose file.

```
version: '3.9'

services:
    frony:
        image: front-image
        networks:
            - mynet
networks:
    my_net_1:
```

Adding networks created by users can be useful when you want to connect services from different Compose files. For example, you can create a network in one Compose file and use it in another Compose file.

Let's say you have 2 apps: FRONT and API. Both are launched using a Docker Compose file.

We will use the first example:

```
version: '3.9'

services:
    frony:
        image: front-image
        networks:
            - frontend
networks:
    frontend:
```

The second stack will look like this:

```
version: '3.9'

services:
    api:
        image: api-image
        networks:
            - mynet
    db:
        image: postgres
        networks:
            - backend
networks:
    backend:
```

We want to connect FRONT to API knowing that both are running in different Docker Compose stacks.

To connect the two stacks, we need to create a third network and use it in both stacks. These are the new stacks with the third network:

```yaml
version: '3.9'

services:
    frony:
        image: front-image
        networks:
            - frontend
            - my_net_external
networks:
    frontend:
    my_net_external:
        external: true
```

The second stack:

```yaml
version: '3.9'

services:
    api:
        image: api-image
        networks:
            - backend
            - my_net_external
    db:
        image: postgres
        networks:
            - backend
networks:
    backend:
    my_net_external:
        external: true
```

Docker Compose Secrets

In one of the previous examples, we used environment variables to define the database configuration. This is not the best way to do it. The problem with this approach is that the environment variables are stored in plain text in the `docker-compose.yml` file. This is not very secure.

```
version: '3.9'

services:
   db:
     image: mysql:5.7
     volumes:
       - db_data:/var/lib/mysql
     restart: always
     environment:
       MYSQL_ROOT_PASSWORD: mypassword
       MYSQL_DATABASE: wordpress
       MYSQL_USER: user
       MYSQL_PASSWORD: mypassword

   wordpress:
     depends_on:
       - db
     image: wordpress:latest
     ports:
       - "8000:80"
     restart: always
     environment:
       WORDPRESS_DB_HOST: db:3306
       WORDPRESS_DB_USER: user
       WORDPRESS_DB_PASSWORD: mypassword
volumes:
    db_data:
```

`WORDPRESS_DB_PASSWORD` is stored in plain text in the `docker-compose.yml` file. Docker Compose provides a better way to define secrets. This is how to do it:

```
cat << EOF > docker-compose.yml
version: '3.9'

services:
   db:
     image: mysql:5.7
     volumes:
       - db_data:/var/lib/mysql
     restart: always
     secrets:
       - db_root_password
       - db_user
```

```
      - db_password
    environment:
      MYSQL_ROOT_PASSWORD_FILE: /run/secrets/db_root_password
      MYSQL_DATABASE: wordpress
      MYSQL_USER_FILE: /run/secrets/db_user
      MYSQL_PASSWORD_FILE: /run/secrets/db_password

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    ports:
      - "8000:80"
    restart: always
    secrets:
      - db_user
      - db_password
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER_FILE: /run/secrets/db_user
      WORDPRESS_DB_PASSWORD_FILE: /run/secrets/db_password

secrets:
    db_root_password:
        file: ./db_root_password.txt
    db_user:
        file: ./db_user.txt
    db_password:
        file: ./db_password.txt

volumes:
    db_data:
EOF
```

As you can see, we are using the `secrets` option to define the secrets. Then, we are using the `_FILE` suffix to define the path to the secret file.

The `_FILE` environment variable is used by the Docker image to read the secret from the file. For example, the `MYSQL_ROOT_PASSWORD_FILE` environment variable is used by the MySQL image to read the root password from the file `/run/secrets/db_root_password`. This is a convention used by Docker images to read secrets from files. Some images may not apply this convention, so you need to check the documentation of the image to see how to read secrets from files.

What we need to do now is to create the secret files. We can do this using the `echo` command:

```
echo mypassword > ./db_root_password.txt
echo user > ./db_user.txt
echo mypassword > ./db_password.txt
```

The MySQL image supports reading the password from a file using the `MYSQL_ROOT_PASSWORD_FILE` environment variable. However, not all images support this. If you want to use a secret with an image that doesn't support reading the password from a file, you can use a script at the entrypoint of the container to read the secret from the file and set it as an environment variable.

```
cat << EOF > entrypoint.sh
#!/bin/bash

# Define the target directory where secrets are stored
secrets_dir="/run/secrets"

# List all secret files in the secrets directory
secret_files=$(ls -1 "$secrets_dir")

# Loop through each secret file and export it as an environment variable
for secret_file in $secret_files; do
    secret_name=$(basename "$secret_file")
    secret_value=$(cat "$secrets_dir/$secret_file")
    export "$secret_name=$secret_value"
done
EOF
```

The secret name will be the name of the file and the secret value will be the content of the file. So if you want to use `MYSQL_ROOT_PASSWORD` instead of `MYSQL_ROOT_PASSWORD_FILE`, you should create a file called `MYSQL_ROOT_PASSWORD` and put the password in it.

```
echo "secret" > MYSQL_ROOT_PASSWORD
```

Now, you need to create a Dockerfile that uses this script as the entrypoint:

```
FROM mysql:5.7
COPY entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]
CMD ["mysqld"]
```

Scaling Docker Compose Services

Docker Compose provides a seamless way to orchestrate multi-container applications. A powerful feature it offers is the ability to scale services. Let's take an example: imagine you're running an application that uses PHP and you've anticipated higher traffic. Instead of running a single PHP container, you want to run 5 of them behind an NGINX web server. Docker Compose lets you do this with a simple command: `docker compose alpha scale php=5`. By executing this, you'd have a total of 5 PHP containers up and running. And because NGINX is linked to this PHP service, it can seamlessly load-balance between these PHP instances.

However, a challenge arises when you consider scaling services like NGINX in this setup. If you recall, the NGINX service maps its internal port 80 to the host's port 8000. This is a 1-to-1 mapping. So, if you tried to scale NGINX, the newly spawned container would also attempt to bind to the host's port 8000, resulting in a conflict. This is because a host port can only be bound to by one service at a time. As a result, in Docker Compose, scaling a service that uses port mapping directly to the host can introduce issues.

Now, keeping this in mind, let's discuss the following Docker Compose setup:

```
version: '3.9'

services:
   db:
     image: mysql:5.7
     volumes:
       - db_data:/var/lib/mysql
     restart: always
     environment:
       MYSQL_ROOT_PASSWORD: mypassword
       MYSQL_DATABASE: wordpress
       MYSQL_USER: user
       MYSQL_PASSWORD: mypassword

   wordpress:
     depends_on:
       - db
     image: wordpress:latest
     ports:
       - "8000:80"
     restart: always
     environment:
       WORDPRESS_DB_HOST: db:3306
       WORDPRESS_DB_USER: user
```

```
25        WORDPRESS_DB_PASSWORD: mypassword
26
27  volumes:
28      db_data:
```

In this setup, we're running a WordPress service that depends on a MySQL database. If you wanted to scale the WordPress service, you'd face the same challenge as discussed earlier with NGINX. The WordPress service is mapping its internal port 80 to the host's port 8000. If you tried to scale this service, you'd encounter a port conflict on the host. Therefore, when designing applications using Docker Compose, it's essential to consider these constraints, especially when planning to scale services with port mappings.

Note that the scaling feature is only available in the alpha version of Docker Compose:

```
1  # Download the install script
2  wget https://get.docker.com/ -O install-docker.sh
3  # Install from the test channel (alpha)
4  sudo sh install-docker.sh --channel test
```

Essentially, scaling up and down is a production feature and Docker Compose is not meant to be used in production. In reality, if you are running a small application that is not mission-critical, you can use Docker Compose in production - even if it is not really recommended - alternatively, you should use other tools such as Docker Swarm or Kubernetes.

Cleaning Docker

With time, Docker will accumulate unused images, containers, volumes, and networks. Here are some commands to clean up Docker.

Delete Volumes

Delete all unused volumes:

```
docker volume prune
```

Delete specific volumes:

```
docker volume rm <volume_name> <volume_name>
```

Delete dangling volumes:

```
docker volume ls -qf dangling=true
```

Delete dangling volumes (alternative):

```
docker volume rm $(docker volume ls -qf dangling=true)
docker volume ls -qf dangling=true | xargs -r docker volume rm
```

Delete Networks

Delete all unused networks:

```
docker network prune
```

Delete specific networks:

```
docker network rm <network_name> <network_name>
```

Delete all networks:

```
docker network rm $(docker network ls -q)
```

Delete all networks from a specific driver:

```
export DOCKER_NETWORK_DRIVER=bridge
docker network ls | grep "$DOCKER_NETWORK_DRIVER" | awk '/ / { print $1 }' | xargs -\
r docker network rm
```

Delete Images

Delete all unused images:

```
docker image prune
```

Delete specific images:

```
docker image rm <image_name> <image_name>
```

Delete dangling images:

```
docker image ls -qf dangling=true
```

Delete dangling images (alternative):

```
docker image rm $(docker image ls -qf dangling=true)
docker image ls -qf dangling=true | xargs -r docker image rm
```

Delete all images:

```
docker image rm $(docker image ls -q)
```

Remove Docker Containers

Remove a container:

```
docker rm <container_name>
```

Force remove a container:

```
docker rm -f <container_name>
```

Remove all exited containers:

```
docker rm $(docker ps -qa --no-trunc --filter "status=exited")
```

Cleaning Up Everything

Delete all unused data:

```
docker system prune
```

Remove all unused images not just dangling ones:

```
docker system prune -a
```

Prune volumes:

```
docker system prune --volumes
```

Docker Plugins

Docker plugins are an exciting feature because they allow Docker to be extended with third-party plugins, such as those for networking or storage. These plugins operate independently of Docker processes and provide webhook-like functionality that the Docker daemon uses to send HTTP POST requests.

There are three types of plugins:

- Network plugins like Weave Network Plugin[108]
- Volume plugins like docker-volume-netshare[109]
- Authorization plugins like docker-casbin-plugin[110]

These plugins are maintained by specific vendors and/or a developers community, however with the rise of orchestration tools like Kubernetes, Docker Swarm, and Mesos, the need for plugins is decreasing and many of these plugins are no longer maintained.

This is a non-exhaustive overview of available plugins that you can also find in the official documentation:

- Contiv Networking[111]: An open-source network plugin to provide infrastructure and security policies for a multi-tenant microservices deployment while providing integration to physical network for the non-container workload. Contiv Networking implements the remote driver and IPAM API s available in Docker 1.9 onwards.
- Kuryr Network Plugin[112]: A network plugin is developed as part of the OpenStack Kuryr project and implements the Docker networking (libnetwork) remote driver API by utilizing Neutron, the OpenStack networking service. It includes an IPAM driver as well.
- Weave Network Plugin[113]: A network plugin that creates a virtual network that connects your Docker containers - across multiple hosts or clouds and enables automatic discovery of applications. Weave networks are resilient, partition tolerant, secure and work in partially connected networks, and other adverse environments - all configured with delightful simplicity.
- Azure File Storage plugin[114]: Lets you mount Microsoft Azure File Storage[115]shares to Docker containers as volumes using the SMB 3.0 protocol. Learn more[116].

[108] https://www.weave.works/docs/net/latest/introducing-weave/
[109] https://github.com/ContainX/docker-volume-netshare
[110] https://github.com/casbin/docker-casbin-plugin
[111] https://github.com/contiv/netplugin
[112] https://github.com/openstack/kuryr
[113] https://www.weave.works/docs/net/latest/introducing-weave/
[114] https://github.com/Azure/azurefile-dockervolumedriver
[115] https://azure.microsoft.com/blog/azure-file-storage-now-generally-available/
[116] https://azure.microsoft.com/blog/persistent-docker-volumes-with-azure-file-storage/

- Blockbridge plugin[117]: A volume plugin that provides access to an extensible set of container-based persistent storage options. It supports single and multi-host Docker environments with features that include tenant isolation, automated provisioning, encryption, secure deletion, snapshots, and QoS.
- Contiv Volume Plugin[118]: An open-source volume plugin that provides multi-tenant, persistent, distributed storage with intent-based consumption. It has support for Ceph and NFS.
- Convoy plugin[119]: A volume plugin for a variety of storage back-ends, including device-mapper and NFS. It's a simple standalone executable written in Go and provides the framework to support vendor-specific extensions such as snapshots, backups, and restore.
- DRBD plugin[120]: A volume plugin that provides highly available storage replicated by
- DRBD[121]. Data written to the docker volume is replicated in a cluster of DRBD nodes.
- Flocker plugin[122]: A volume plugin that provides multi-host portable volumes for Docker, enabling you to run databases and other stateful containers and move them around across a cluster of machines.
- gce-docker plugin[123]: A volume plugin able to attach, format and mount Google Compute persistent-disks[124].
- GlusterFS plugin[125]: A volume plugin that provides multi-host volumes management for Docker using GlusterFS.
- Horcrux Volume Plugin[126]: A volume plugin that allows on-demand, version-controlled access to your data. Horcrux is an open-source plugin, written in Go, and supports SCP, Minio[127] and Amazon S3.
- HPE 3Par Volume Plugin[128]: A volume plugin that supports HPE 3Par and StoreVirtual iSCSI storage arrays.
- IPFS Volume Plugin[129]: An open-source volume plugin that allows using an ipfs[130]filesystem as a volume.
- Keywhiz plugin[131]: A plugin that provides credentials and secret management using Keywhiz as a central repository.
- Local Persist Plugin[132]: A volume plugin that extends the default `local` driver's functionality by allowing you specify a mount-point anywhere on the host, which enables the files to always persist , even if the volume is removed via `docker volume rm`.

[117] https://github.com/blockbridge/blockbridge-docker-volume
[118] https://github.com/contiv/volplugin
[119] https://github.com/rancher/convoy
[120] https://www.drbd.org/en/supported-projects/docker
[121] https://www.drbd.org/
[122] https://clusterhq.com/docker-plugin/
[123] https://github.com/mcuadros/gce-docker
[124] https://cloud.google.com/compute/docs/disks/persistent-disks
[125] https://github.com/calavera/docker-volume-glusterfs
[126] https://github.com/muthu-r/horcrux
[127] https://www.minio.io/
[128] https://github.com/hpe-storage/python-hpedockerplugin/
[129] http://github.com/vdemeester/docker-volume-ipfs
[130] https://ipfs.io/
[131] https://github.com/calavera/docker-volume-keywhiz
[132] https://github.com/CWSpear/local-persist

- NetApp Plugin[133](nDVP): A volume plugin that provides direct integration with the Docker ecosystem for the NetApp storage portfolio. The nDVP package supports the provisioning and management of storage resources from the storage platform to Docker hosts, with a robust framework for adding additional platforms in the future.
- Netshare plugin[134]: A volume plugin that provides volume management for NFS 3/4 , AWS EFS and CIFS file systems.
- OpenStorage Plugin[135]: A cluster-aware volume plugin that provides volume management for file and block storage solutions. It implements a vendor-neutral specification for implementing extensions such as CoS , encryption, and snapshots. It has example drivers based on FUSE , NFS, NBD, and EBS, to name a few.
- Portworx Volume Plugin[136]: A volume plugin that turns any server into a scale-out converged compute/storage node, providing container granular storage and highly available volumes across any node, using a shared-nothing storage backend that works with any docker scheduler.
- Quobyte Volume Plugin[137]: A volume plugin that connects Docker to Quobyte[138]'s data center file system, a general-purpose scalable and fault-tolerant storage platform.
- REX-Ray plugin[139]: A volume plugin which is written in Go and provides advanced storage functionality for many platforms including VirtualBox, EC2, Google Compute Engine, OpenStack, and EMC .
- Virtuozzo Storage and Ploop plugin[140]: A volume plugin with support for Virtuozzo Storage distributed cloud file system as well as ploop devices.
- VMware vSphere Storage Plugin[141]: Docker Volume Driver for vSphere enables customers to address persistent storage requirements for Docker containers in vSphere environments.
- Twistlock AuthZ Broker[142]: A basic extendable authorization plugin that runs directly on the host or inside a container. This plugin allows you to define user policies that it evaluates during authorization. Basic authorization is provided if Docker daemon is started with the –tlsverify flag (username is extracted from the certificate common name).

Docker Engine's plugin system allows you to install, start, stop, and remove plugins using Docker Engine.

For example, to install a plugin, you can use the following command:

```
1 docker plugin install [PLUGIN_NAME]
```

To list all installed plugins, you can use the following command:

[133]https://github.com/NetApp/netappdvp
[134]https://github.com/ContainX/docker-volume-netshare
[135]https://github.com/libopenstorage/openstorage
[136]https://github.com/portworx/px-dev
[137]https://github.com/quobyte/docker-volume
[138]http://www.quobyte.com/containers
[139]https://github.com/emccode/rexray
[140]https://github.com/virtuozzo/docker-volume-ploop
[141]https://github.com/vmware/docker-volume-vsphere
[142]https://github.com/twistlock/authz

```
docker plugin ls
```

This is an example:

```
docker plugin install weaveworks/net-plugin:latest_release
```

Configure the plugin based on the prompts. The plugin is installed and enabled.

To enable a plugin, you can use the following command:

```
docker plugin enable [PLUGIN_NAME]
```

Example:

```
# Plugin already enabled but this is how you would enable it
docker plugin enable weaveworks/net-plugin:latest_release
```

To upgrade it, you can use the following commands:

```
# Disable the plugin first
docker plugin disable [PLUGIN_NAME]
# Upgrade the plugin
docker plugin upgrade [PLUGIN_NAME]
```

Example:

```
docker plugin disable weaveworks/net-plugin:latest_release
docker plugin upgrade weaveworks/net-plugin:latest_release
```

Once installed and enabled, you can create networks that use the Weave plugin.

```
# Restart the docker daemon
sudo systemctl restart docker
# Re-enable the plugin if needed
docker plugin enable weaveworks/net-plugin:latest_release
# Create a network using the plugin
docker network create --driver=weaveworks/net-plugin:latest_release my_weave_network
```

You can inspect the plugin using the following command:

```
docker plugin inspect [PLUGIN_NAME]
```

Example:

```
docker plugin inspect weaveworks/net-plugin:latest_release
```

The output for this plugin includes the following information:

- Plugin configuration
- Environment variables
- Interface information
- Linux capabilities
- Mount information
- Network information
- Plugin status
- Plugin settings

The `rootfs` section and SHA digest (`diff_ids`) represent the plugin's root filesystem layers, which are used by Docker to manage the plugin's images.

The output confirms that the plugin is properly configured, active, and ready to be used with its default settings. If you need to customize any configurations, such as adjusting the `LOG_LEVEL` or providing a `WEAVE_PASSWORD` for encryption, you can use the `docker plugin set` command to modify these environment variables accordingly.

```
# Remove the network
docker network rm my_weave_network
# Disable the plugin
docker plugin disable weaveworks/net-plugin:latest_release
# Change the configuration. Example: Set the log level to debug
docker plugin set weaveworks/net-plugin:latest_release LOG_LEVEL=debug
# Enable the plugin
docker plugin enable weaveworks/net-plugin:latest_release
# Recreate a network using the plugin
docker network create --driver=weaveworks/net-plugin:latest_release my_weave_network
```

Orchestration - Docker Swarm

What is Docker Swarm?

Docker Swarm is the container orchestration platform developed and maintained by Docker, Inc.

It is specifically designed to simplify the deployment, management, and scalability of containerized applications at scale. Swarm allows users to create and manage a cluster of Docker hosts as a single virtual system - a feature referred to as container orchestration. This makes it easier to distribute containerized applications across multiple nodes.

Docker Swarm comes with important key features such as service discovery, load balancing, scaling, rolling updates, high availability, security, integration with Docker, and more.

Service Abstraction

Docker Swarm introduced the concept of services, which represent a group of containers that perform the same task.

When you work with a standalone container, you can use the `docker run` command to start it. However, when you work with a service, you use the `docker service create` command to specify the desired state of the service. This transition from transactional to declarative commands is a key feature of Docker Swarm.

With a service, you can define the desired number of replicas, and Docker Swarm ensures that the specified number of containers are running across the cluster. If one of the containers in a service fails, Docker Swarm automatically restarts it to match the desired state.

Load Balancing

Swarm includes an integrated load balancer that distributes incoming requests across containers within a service. This load balancer runs on every node in the cluster and ensures that requests are routed to the appropriate containers, regardless of which node they are running on.

Scalability

You can easily adjust the number of replicas of a service to accommodate changes in load or application requirements. However, to automatically scale a service based on CPU usage, you would need to utilize third-party tools such as Prometheus or cAdvisor, along with custom scripts or other tools.

High Availability

Swarm provides high availability by automatically rescheduling containers in the event of node failures. This ensures that services remain accessible and reliable.

Let's consider a scenario where you have 5 containers forming a service and initially, you have 3 nodes in your cluster. When you deploy the service, Swarm will distribute the containers across the nodes, running the service on all 3 nodes.

If one of the nodes fails due to system, network, or hardware failure, Swarm will automatically reschedule the containers on the remaining nodes. As a result, the service will continue to run smoothly without any downtime.

Rolling Updates

Swarm supports rolling updates, which enable you to update a service without causing any downtime. It gradually replaces old containers with new ones to minimize any impact on the service.

Security

Docker Swarm provides security features such as mutual TLS (Transport Layer Security) to encrypt communication between nodes, role-based access control (RBAC), and secrets management.

Integration with Docker

Since Docker Swarm is part of the Docker ecosystem, it integrates with Docker Compose. First, create a docker-compose.yml file that defines the services comprising your application. Then, use the `docker stack deploy` command to deploy the stack to the Swarm cluster.

Creating a Swarm Cluster

In this section, we will be using 3 machines:

- The "manager" machine: This machine will serve as the manager of the Swarm cluster. Its responsibilities include managing the cluster and scheduling containers on the worker nodes.
- The "worker01" machine: This machine will function as a worker node within the Swarm cluster. It will be responsible for running containers.
- The "worker02" machine: Similar to "worker01", this machine will also serve as a worker node within the Swarm cluster and run containers.

To ensure proper functionality, the following ports should be open on all machines:

- Port 2377 TCP: This port is used for communication between manager nodes.
- Port 7946 TCP/UDP: It facilitates overlay network node discovery.
- Port 4789 UDP: This port is essential for overlay network traffic.

Additionally, Docker must be installed on all machines. Since this tutorial focuses on Ubuntu 22.04, you can install Docker using the following commands:

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh
```

After installing Docker, the Docker service should be running. You can check its status using the following command:

```
sudo systemctl status docker
```

If the service is not running, you can start it using the following commands:

```
sudo systemctl enable docker
sudo systemctl start docker
```

Our cluster is not ready yet. We need to initialize the Swarm cluster and add the worker nodes to it.

Initializing the Swarm

On the manager node, we need to create and initialize the Swarm using a command similar to the following:

```
docker swarm init --advertise-addr <MANAGER-IP>
```

The manager IP refers to the IP address of the manager node. It is crucial to use the private IP of the manager node, as it will be used for communication between the manager node and the worker nodes.

To execute the command, run:

```
export MANAGER_IP=<MANAGER-IP>
docker swarm init --advertise-addr $MANAGER_IP
```

This will show the output similar to the following:

```
Swarm initialized: current node (9i91nnzwypfbqfzjamtyucndy) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-2o2julopgcjvmgt95p3eaqwp7evyy6xsqgj9fplqqdd6v\
332e0-77zt8cv7ea22m9igrl2tkuhp8 10.135.0.6:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the\
 instructions.
```

The output displays the command that needs to be executed on the worker nodes in order to add them to the cluster.

Copy the command and navigate to each of your worker nodes to run it. The command should be similar to the following:

```
docker swarm join --token <Token> <Manager-IP>:2377
```

You should see the following output on each worker node:

```
This node joined a swarm as a worker.
```

Run the following command on the manager node to see the nodes in the cluster:

```
docker node ls
```

You should see the following output:

```
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   E\
NGINE VERSION
9i91nnzwypfbqfzjamtyucndy *   manager    Ready     Active         Leader           2\
4.0.6
dxs15h0utix51cpeam9nboa0x     worker01   Ready     Active                          2\
4.0.6
rk06f3dtfo55i8yqvu56zt0fl     worker02   Ready     Active                          2\
4.0.6
```

If you see this output, it means that the Swarm cluster is ready.

Installing Docker Swarm Visualizer

Docker Swarm Visualizer is a tool that visually represents the Swarm cluster. It displays the nodes within the cluster and the services running on each node.

To install Docker Swarm Visualizer, execute the following command on the manager node:

```
docker service create \
  --name=viz \
  --publish=8080:8080/tcp \
  --constraint=node.role==manager \
  --mount=type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock \
  dockersamples/visualizer
```

You can now access the visualizer using the following URL:

```
echo "http://$MANAGER_IP:8080"
```

You can use this tool to visualize the Swarm nodes and services.

Adding New Nodes to the Cluster

To add new nodes to the cluster, we need to run the following command on the manager node:

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

This will show the command that we need to run on the new node to add it to the cluster.

Swarm Services and Tasks

In this section, we are going to create a service that runs a web server. We will use the `nginx` image to create the service.

To create the service, run the following command on the manager node:

```
docker service create --name webserver -p 80:80 nginx
```

This will create a service called `webserver` that runs the `nginx` image. It will also expose port 80 on the host machine.

To see the services running on the cluster, run the following command on the manager node:

```
docker service ls
```

To see the tasks running on the cluster, run the following command on the manager node:

```
docker service ps webserver
```

A task is a running container that is part of a service.

To see the logs of a task, run the following command on the manager node:

```
docker service logs webserver
```

Scaling a Service

To scale a service, we need to run the following command on the manager node:

```
docker service scale webserver=3
```

This will start three containers that are part of the `webserver` service. Alternatively, we could have specified the number of replicas when creating the service using the `--replicas` option:

```
docker service create --name webserver -p 80:80 --replicas 3 nginx
```

Networking in Docker Swarm

When initially creating a Swarm cluster, Docker automatically creates a network called `ingress`. This network is utilized for inter-node communication and functions as the load balancer for distributing incoming requests among containers within a service.

When creating a service, you have the option to specify which network it will use. If no network is specified, the service will default to using the `ingress` network.

To view all the default networks on the master node, use the following command:

```
docker network ls
```

You will see the following 4 default networks:

- bridge: This is the default network used when you run a container without specifying a network.
- docker_gwbridge: Connects the individual Docker daemon to other daemons participating in the swarm.
- host: This is the network used when you run a container in host mode.
- none: This is the network used when you run a container without a network.

When you run a service, you can specify the network it will use using the `--network` option. For example, let's create a service called `webserver` that runs the `nginx` image and uses the `bridge` network.

To begin, create the network:

```
docker network create --driver overlay webserver-network
```

Now, you have two choices:

- Remove the service and recreate it using the `-network` option.
- Update the service using the `-network` option.

Let's proceed with removing the service and recreating it using the `--network` option.

```
docker service rm webserver
docker service create --name webserver -p 80:80 --network webserver-network nginx
```

The second option can be done using the following command:

```
docker service update --network-add webserver-network webserver
```

Now you can inspect the service and see that it is using the `webserver-network` network:

```
docker service inspect webserver --pretty | grep -i network
```

You can also inspect the network and see that the service is using it. Run this on each node to see how the network is distributed across the cluster:

```
docker network inspect webserver-network
```

You will notice that there are two containers running within the network:

- Replica Containers: These containers are responsible for executing the tasks specified by the service. In our case, this refers to the Nginx container that serves your web application.
- Endpoint Containers: These containers serve as the network endpoint for the service. Their role is to direct incoming network traffic to the replica containers. The endpoint container acts as a load balancer, distributing requests among the individual replicas.

Performing Operations on Nodes

Sometimes, you may need to perform certain operations on a node, such as:

- Pausing a node
- Draining a node
- Removing a node
- Promoting a node
- Demoting a node

Let's explore how these operations can be performed and what they entail.

Pausing a Node

When a node is paused, Docker Swarm will halt the scheduling of new tasks on that node and instead reschedule any future tasks on other nodes.

To pause a node, execute the following command on the manager node:

```
export NODE_HOSTNAME=worker01
docker node update --availability pause $NODE_HOSTNAME
```

Then launch a new service and see that the tasks are not scheduled on the paused node.

```
docker service create --name webserver-test --replicas 5 -p 8081:80 nginx
```

Check where the tasks are running using the following command:

```
docker service ps webserver-test
```

To check the status of the node, run the following command on the manager node:

```
docker node ls
```

You should see that the node is paused (`AVAILABILITY` column).

To resume the node, run the following command on the manager node:

```
docker node update --availability active $NODE_HOSTNAME
```

Try scaling the service and see that the tasks are scheduled on the node again.

```
docker service scale webserver-test=10
docker service ps webserver-test
```

Draining a Node

Draining a node in Docker Swarm means that new tasks will no longer be scheduled on the node. Additionally, any existing tasks on the node will be rescheduled to other nodes.

To drain a node, you need to run the following command on the manager node:

```
export NODE_HOSTNAME=worker01
docker node update --availability drain $NODE_HOSTNAME
```

Now you can check the status of the node using the following command:

```
docker node ls
```

Check that all the tasks are running on all nodes except the drained node:

```
docker service ps --filter desired-state=Running $(docker service ls -q)
```

To resume the node, run the following command on the manager node:

```
docker node update --availability active $NODE_HOSTNAME
```

Scale any service to refresh the tasks and see that the tasks are scheduled on the node again.

```
docker service scale webserver-test=15
docker service ps --filter desired-state=Running $(docker service ls -q)
```

Removing a Node

To remove a node from the cluster, we need to run the following command on the manager node:

```
export NODE_HOSTNAME=worker01
docker node update --availability drain $NODE_HOSTNAME
docker node rm $NODE_HOSTNAME --force
```

To complete this operation, we need to run the following command on the removed node:

```
docker swarm leave
```

If you want to add the node back to the cluster, you need to run the following command on the manager node:

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

This will show the command that we need to run on the removed node to make it join the cluster again.

Promoting and Demoting Nodes

Promoting a node means that Docker Swarm will designate it as a manager node.

In our cluster, we have 3 nodes. One of them is already a manager node (named "manager"), while the other two are worker nodes (named "worker01" and "worker02"). If you want to promote one of the worker nodes to a manager node, you need to execute the following command on the current manager node:

```
export NODE_HOSTNAME=worker01
docker node promote $NODE_HOSTNAME
```

We currently have two manager nodes. To list the manager nodes, use the following command:

```
docker node ls --filter role=manager
```

To demote a manager node, we need to run the following command on the manager node:

```
export NODE_HOSTNAME=worker01
docker node demote $NODE_HOSTNAME
```

Multi-manager Docker Swarm

We previously created a Swarm cluster with one manager node. Then, we promoted one of the worker nodes to a manager node. But what is the difference between these two types of nodes? Additionally, what is the difference between a single-manager cluster and a multi-manager cluster? This section will provide the answers.

Firstly, there are two types of nodes in a Swarm cluster:

- Manager nodes: These nodes are responsible for managing the cluster, including scheduling containers on worker nodes and maintaining the cluster state. Only manager nodes can run the `docker swarm` commands.
- Worker nodes: These nodes are responsible solely for running containers and do not participate in managing the cluster.

When you start a service, the manager node will schedule containers on all nodes, including the manager nodes themselves. This means that manager nodes can also run containers. Therefore, there are no differences between manager nodes and worker nodes except that manager nodes have the additional responsibility of managing the entire cluster.

If desired, you can prevent the scheduling of containers on manager nodes. This is useful when you want to dedicate manager nodes solely to managing the cluster and not running containers. There are two options to achieve this:

Option 1: You can simply drain the manager nodes like we did in the previous section:

```
export NODE_HOSTNAME=manager
docker node update --availability drain $NODE_HOSTNAME
```

Option 2: You can use the `constraint` option when you create the service to specify that the service should not run on the manager nodes:

This is an example:

```
docker service create --name webserver-another-test -p 8002:80 --constraint node.rol\
e==worker --replicas 10 nginx
```

Now check where the new service is running:

```
docker service ps webserver-another-test --format "table {{.Name}}\t{{.Node}}"
```

You should see that the service is not running on the manager nodes.

Let's now understand the difference between a single-manager cluster and a multi-manager cluster.

In a single-manager cluster, if the manager node fails, the entire cluster will fail. This is because the manager node is responsible for managing the cluster. As a result, a single-manager cluster lacks high availability. However, if you have multiple manager nodes, the cluster is more likely to survive the failure of one of the manager nodes. In practice, the more manager nodes you have, the more resilient your cluster will be. However, it's important to note that you need to have an odd number of manager nodes.

It is recommended to have an odd number of manager nodes for fault tolerance, with a maximum of 7 (as recommended by Docker documentation). However, it's important to understand that adding more managers doesn't increase scalability or performance. Stability is different from scalability.

In Docker swarm mode, manager nodes use the Raft Consensus Algorithm to ensure consistent cluster states, enabling any manager to take over in case of failure. Raft tolerates failures but requires a majority of nodes to agree on values, which can impact task scheduling when a node is unavailable. This implementation in swarm mode aligns with key properties of distributed systems, such as fault tolerance, mutual exclusion through leader election, cluster membership management, and globally consistent object sequencing. Simply put, manager nodes use a voting system to reach agreements, which can impact workload assignment if some managers are down. It's like a team where most members need to agree before taking action. This teamwork ensures smooth operation and alignment among all managers.

Imagine you have a team of managers in charge of a big project. They need to ensure everyone is informed about the project's progress and that it continues to run smoothly, even if one of them becomes ill or is unable to work. They use a special voting system called Raft to achieve this. This system ensures that all managers agree on the project's status and tasks. If one manager is unable to continue, another manager can step in because they all have the same information. However, there's

a condition: to make a decision, they need the majority of the managers to agree. It's like requiring at least half of them to say "yes" before proceeding.

Similarly, Raft is effective at handling a certain number of unavailable managers. However, if too many managers are out of commission, important decisions cannot be made and workload balancing becomes ineffective. Specifically, Raft can tolerate up to (N-1)/2 failures and requires a majority of (N/2)+1 members to agree on values. For example, if you have three managers, you can tolerate one failure. If you have five managers, you can tolerate two failures. If you have seven managers, you can tolerate three failures. And so on.

The following table shows the number of managers that can fail in a cluster of different sizes:

Number of Managers	Number of Failures
1	0
3	1
5	2
7	3

Docker Swarm Environment Variables and Secrets

Before we start, let's set back the cluster to its initial state. On the manager node, run:

```
docker node demote worker01
docker node update --availability active manager
```

Let's also remove all the previous services we created:

```
docker service rm $(docker service ls -q)
```

If you want to run MySQL as a standalone container using the official MySQL image[143], you can use the following command:

```
docker run --name mysql -e MYSQL_DATABASE=wordpress -e MYSQL_USER=wordpress -e MYSQL\
_ROOT_PASSWORD=secret -d mysql:5.7
```

This will create a MySQL container with the following configuration read from environment variables:

- MYSQL_DATABASE: This is the name of the database that will be created.
- MYSQL_USER: This is the username of the database user.
- MYSQL_PASSWORD: This is the password of the database user.

To do the same thing in a Docker Swarm cluster, we need to create a service using the following command:

[143] https://hub.docker.com/_/mysql

```
docker service create --name mysql -e MYSQL_DATABASE=wordpress -e MYSQL_USER=wordpre\
ss -e MYSQL_ROOT_PASSWORD=secret -d mysql:5.7
```

The problem here is that the password is stored in plain text in the command. You can see it if you run the following command:

```
docker service inspect mysql --format "{{json .Spec.TaskTemplate.ContainerSpec.Env}}"
```

Or:

```
docker service inspect mysql --pretty | grep Env
```

A good practice is to avoid storing sensitive information in plain text. This is where Docker secrets come in.

Docker Secrets

Docker secrets are used to store sensitive information such as passwords, SSH keys, API keys, authentication tokens, and more. They are securely stored in an encrypted format on disk and in memory, and are only accessible to the services that require them.

To create a secret, we need to run the following command on the manager node:

```
echo "secret" | docker secret create mysql_root_password -
```

You can also store the secret in a file and create the secret using the following command:

```
echo "secret" > mysql_root_password.txt
docker secret create mysql_root_password mysql_root_password.txt
```

To show the secret, run the following commands on the manager node:

```
docker secret ls
docker secret inspect mysql_root_password
```

To use the secret in a service, we need to run the following command:

```
# remove the previous service
docker service rm mysql
# create the new service with the secret
docker service create --name mysql \
--secret mysql_root_password \
-e MYSQL_ROOT_PASSWORD_FILE=/run/secrets/mysql_root_password \
-e MYSQL_DATABASE=wordpress \
-e MYSQL_USER=wordpress \
-d mysql:5.7
```

When you create a secret, it is stored in the in-memory filesystem of the container. The path to the secret is `/run/secrets/[SECRET_NAME]`. To check the content of the secret on the manager node, use the following command:

```
docker exec -it $(docker ps -q -f name=mysql) cat /run/secrets/mysql_root_password
```

Only MySQL containers have access to the secret. If you attempt to access the secret from another container, you will receive an error.

```
docker service create --name webserver -p 80:80
docker exec -it $(docker ps -q -f name=webserver) cat /run/secrets/mysql_root_passwo\
rd
```

You should see the following error:

```
cat: /run/secrets/mysql_root_password: No such file or directory
```

The MySQL image allows you to read the password from a file by utilizing the `MYSQL_ROOT_PASSWORD_FILE` environment variable. It's important to note that not all images support this feature. In case you need to use a secret with an image that doesn't support reading the password from a file, you can employ a script at the container's entrypoint. This script can read the secret from the file and set it as an environment variable.

```bash
#!/bin/bash

# Define the target directory where secrets are stored
secrets_dir="/run/secrets"

# List all secret files in the secrets directory
secret_files=$(ls -1 "$secrets_dir")

# Loop through each secret file and export it as an environment variable
for secret_file in $secret_files; do
    secret_name=$(basename "$secret_file")
    secret_value=$(cat "$secrets_dir/$secret_file")
    export "$secret_name=$secret_value"
done
```

The secret name will be the name of the file, and the secret value will be the content of the file. So, if you want to use `MYSQL_ROOT_PASSWORD` instead of `MYSQL_ROOT_PASSWORD_FILE`, you should create a file called `MYSQL_ROOT_PASSWORD` and put the password in it.

```
echo "secret" > MYSQL_ROOT_PASSWORD
```

Now, you need to create a Dockerfile that uses this script as the entrypoint:

```
FROM mysql:5.7
COPY entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]
CMD ["mysqld"]
```

You can also use a simpler script:

```bash
#!/bin/bash

# Read the secret from a file and export it as an environment variable
export SECRET_VALUE=$(cat /run/secrets/my_secret)

# Execute the main command of the container
exec "$@"
```

It is important to note that exporting sensitive information as environment variables is not a recommended practice. This is because environment variables can be accessed by any process running on the system.

The use of Docker secrets and storing them in files instead of environment variables is not about making the container completely secure, but rather about reducing the exposure of secrets in less secure components of the Docker ecosystem.

In Docker Swarm, secrets are securely encrypted both during transmission and while at rest, ensuring that they can only be accessed by explicitly authorized services. These secrets are only accessible to designated services and only for the duration of their active tasks.

There are alternative methods to use secrets with images that do not support reading passwords from files, such as modifying your application's code to directly read the secret from the file located at `/run/secrets/`.

If your company uses a secret management system like HashiCorp Vault, AWS Secret Manager, or Azure Key Vault, you are not limited to using Docker secrets. You can continue using these tools without needing the Swarm secret management system.

Docker Configs

To make a Docker image as generic as possible, you can utilize Docker configs to store configuration files. Docker configs serve as an alternative to setting up environment variables. While a Docker config is similar to a secret, it is not encrypted at rest and is directly mounted into the container's filesystem without utilizing RAM disks.

There are differences between environment variables and configs, but the primary advantage of using configs is that managing them is more practical compared to managing environment variables, as they can be managed using the Docker CLI.

Furthermore, configs are immutable, meaning they cannot be modified once created. This is beneficial for security purposes, as it prevents unauthorized changes to your application's configuration.

Config files can have restricted read permissions, ensuring only the application user has access to them. In contrast, environment variables may potentially be accessed by subprocesses or anyone accessing your container.

Let's start an Nginx server with a custom "index.html" file. First, create the "index.html" file:

```
1 echo "Hello World" > index.html
```

Then create the config:

```
1 docker config create nginx_index index.html
```

Check the config:

```
docker config ls
docker config inspect nginx_index
```

Now you can use the config in a service:

```
docker service create \
--name webserver \
-p 8004:80 \
--config source=nginx_index,target=/usr/share/nginx/html/index.html \
nginx
```

Find out where the Nginx container is running:

```
docker service ps webserver
```

Then check the content of the index.html file from the right worker:

```
docker exec -it $(docker ps -q -f name=webserver) cat /usr/share/nginx/html/index.ht\
ml
```

You should be able to see that the content of the file is `Hello World`.

Note that we can also set the file permissions when we create the config. For example, we can set the file permissions to 0440 using the following command:

```
docker service create \
--name webserver \
-p 8004:80 \
--config source=nginx_index,target=/usr/share/nginx/html/index.html,mode=0440 \
nginx
```

The permissions are set to 0440, which means that only the owner and the group can read the file and not the others.

We can also rotate the config using the following command:

```
# create a new index.html file
echo "Hello World v2" > index.html
# create a new config
docker config create nginx_index_v2 index.html
# update the service to use the new config and remove the old one
docker service update \
--config-rm nginx_index \
--config-add source=nginx_index_v2,target=/usr/share/nginx/html/index.html \
webserver
```

Now check the content of the index.html file from the right worker:

```
docker exec -it $(docker ps -q -f name=webserver) cat /usr/share/nginx/html/index.ht\
ml
```

You should be able to see that the content of the file is `Hello World v2`.

Docker Swarm Volumes

Since containers are ephemeral, the data stored inside them can be lost when the container is removed. The same principle applies to services in Docker Swarm, as they are composed of containers.

When running stateful applications such as databases, it is important to persist the data. This is where Docker volumes come into play.

Now, let's remove the previous MySQL service:

```
docker service rm mysql
```

To create a volume, we need to run the following command on the manager node:

```
docker volume create mysql_data
```

To use the volume in a service, we need to run the following command:

```
docker service create --name mysql \
--secret mysql_root_password \
-e MYSQL_ROOT_PASSWORD_FILE=/run/secrets/mysql_root_password \
-e MYSQL_DATABASE=wordpress \
-e MYSQL_USER=wordpress \
--mount type=volume,source=mysql_data,target=/var/lib/mysql \
-d mysql:5.7
```

When you run a service with a volume, the volume is created on the node where the container is running. You can check the volume using the following command:

```
docker volume ls
```

Since we have launched a single instance of MySQL, the command above should be executed on the node where the container is running. To determine the location of the container, run the following command:

```
docker service ps mysql
```

If the MySQL container is moved to another node, the volume will not be moved along with it. This is because the volume is created on the node where the container is currently running.

To overcome this challenge, we can use constraints to ensure that the container always runs on the same node. To do this, execute the following command:

```
# choose the node where you want to run the MySQL service
export NODE_HOSTNAME=worker01
# remove the old service and the old volume
docker service rm mysql
docker volume rm mysql_data
# create the new volume on the right node (e.g: worker01)
docker volume create mysql_data
# create the new service with the constraint
docker service create --name mysql \
--secret mysql_root_password \
-e MYSQL_ROOT_PASSWORD_FILE=/run/secrets/mysql_root_password \
-e MYSQL_DATABASE=wordpress \
-e MYSQL_USER=wordpress \
--mount type=volume,source=mysql_data,target=/var/lib/mysql \
--constraint node.hostname==$NODE_HOSTNAME \
-d mysql:5.7
```

Now check the volume using the following command:

```
docker volume ls
```

MySQL data is now stored on the host machine (worker01 in our case) and not on the container. You can find the data in the following directory:

```
ls /var/lib/docker/volumes/mysql_data/_data/
```

The decision to force MySQL to run on a specific node resolved the issue of data persistence, but it introduced another problem: what happens if the node fails? In that case, the service will become unavailable.

Additionally, if we want to scale up our database or set up a high availability (HA) MySQL system with masters and replicas, it is not possible with a single MySQL instance.

Swarm Mode itself does not handle the scalability of stateful applications like databases, so it is the responsibility of the user to address this. To tackle this challenge, we can utilize a distributed file system such as NFS or GlusterFS to store the data, or opt for a managed service like Amazon EFS.

Deploying a WordPress Application on Docker Swarm

Let's utilize what we have learned so far to deploy a WordPress application on Docker Swarm.

Before moving forward, let's delete all the previous services we created:

```
docker service rm $(docker service ls -q)
```

The Wordpress application will utilize a MySQL database, so we need to create a MySQL service as well. Both services should be in the same network to enable communication between them.

Let's begin by creating the network:

```
export network_name=wordpress-network
docker network create --driver overlay $network_name
```

Create Docker secrets for MySQL and WordPress:

```
echo "your_db_root_password" | docker secret create db_root_password -
echo "your_db_wordpress_password" | docker secret create db_wordpress_password -
```

Create the MySQL service:

```
docker service create --name mysql \
  --network $network_name \
  --secret db_root_password \
  --secret db_wordpress_password \
  -e MYSQL_ROOT_PASSWORD_FILE=/run/secrets/db_root_password \
  -e MYSQL_USER=wordpress \
  -e MYSQL_PASSWORD_FILE=/run/secrets/db_wordpress_password \
  -e MYSQL_DATABASE=wordpress \
  mysql:5.7
```

Create a WordPress service:

```
docker service create --name wordpress \
  --network $network_name \
  --secret db_wordpress_password \
  -e WORDPRESS_DB_HOST=mysql \
  -e WORDPRESS_DB_USER=wordpress \
  -e WORDPRESS_DB_PASSWORD_FILE=/run/secrets/db_wordpress_password \
  -e WORDPRESS_DB_NAME=wordpress \
  -p 80:80 \
  --replicas 3 \
  wordpress:latest
```

Now visit one of the nodes public IPs on port 80 and you should see the WordPress installation page.

Docker Swarm Global Services

A global service is a service that runs one task on each node in the cluster. This mode is also referred to as the global mode.

Let's say you want to run a logging agent on every node in the cluster to collect logs from all the containers. To achieve this, you need to ensure two things:

- The logging agent should run on all nodes.
- Only one instance of the logging agent should be running on each node.

In the following example, we will be using Fluentd as the logging agent. To create a global service, run the following command on the manager node and include the `--mode global` option:

```
docker service create \
--name logging-agent \
--mode global \
fluent/fluentd:v1.3-debian-1
```

You should be able to see that the service is running on all nodes and that the total number of tasks is equal to the number of nodes in the cluster:

```
docker service ps logging-agent
```

Docker Swarm Resouce Management

Like with standalone containers, we can configure services with resource reservations and limits. This is useful if you want to control the amount of resources that a service can use.

Let's start with this example:

```
# remove old services
docker service rm $(docker service ls -q)
# create the service
docker service create \
--name webserver \
--replicas 3 \
--reserve-memory 100m \
--limit-memory 300m \
--reserve-cpu 0.5 \
--limit-cpu 1 \
-p 80:80 \
nginx
```

As you can see we used the following options:

- `--reserve-memory`: This option is used to reserve memory for the service. In our example, we reserved 100MiB of memory for the service (MiB stands for Mebibyte).
- `--limit-memory`: This option is used to limit the memory that the service can use. In our example, we limited the memory to 300MiB.
- `--reserve-cpu`: This option is used to reserve CPU for the service. In our example, we reserved 0.5 CPU for the service.
- `--limit-cpu`: This option is used to limit the CPU that the service can use. In our example, we limited the CPU to 1.

You can see how Docker converts the values to the format it uses internally using the following command:

```
docker service inspect --format "{{ json .Spec.TaskTemplate.Resources }}" webserver
```

You should see this:

```
{"Limits":{"NanoCPUs":1000000000,"MemoryBytes":314572800},"Reservations":{"NanoCPUs"\
:500000000,"MemoryBytes":104857600}}
```

> **i** Docker uses different units for CPU and memory. Here is how Docker converts the values:
>
> A CPU core is equal to 1e9 (1,000,000,000) nanocores (or nanocpus). So 0.5 CPU is equal to 500,000,000 nanocores and 1 CPU is equal to 1,000,000,000 nanocores.
>
> 1 Mebibyte is equal to 1.04858 Megabytes which is equal to 1,048,576 bytes. So 100MiB is equal to 104,857,600 bytes and 300MiB is equal to 314,572,800 bytes.

Note that each container in the service will have the same resource reservations and limits defined in the service. You can check the resources of a container using the following command:

```
docker inspect --format "{{ json .HostConfig.Resources }}" $(docker ps -q -f name=we\
bserver)
```

Docker Swarm Stacks

A stack is a group of interrelated services that share dependencies and can be orchestrated and scaled together. Here are some examples:

- A WordPress stack that includes a WordPress service, Apache, and a MySQL service.
- A web application stack that includes a web application service, a caching service like Redis, and a database service like PostgreSQL.
- A monitoring stack that includes Prometheus, Loki, and Grafana services.
- ...and so on.

One of the main advantages of Docker Swarm is its ability to deploy using a docker-compose file. In the context of Docker Swarm, this file is referred to as a stack.

Let's start with a basic example. Create a file named `docker-compose.yml` for a WordPress stack:

```
cat <<EOF > docker-compose.yml
version: '3.9'

services:
   db:
     image: mysql:latest
     volumes:
       - db_data:/var/lib/mysql
     environment:
       MYSQL_ROOT_PASSWORD_FILE: /run/secrets/db_root_password
       MYSQL_DATABASE: wordpress
       MYSQL_USER: wordpress
       MYSQL_PASSWORD_FILE: /run/secrets/db_password
     secrets:
       - db_root_password
       - db_password
     networks:
       - wordpress-network

   wordpress:
     depends_on:
       - db
     image: wordpress:latest
     ports:
       - "8000:80"
     environment:
       WORDPRESS_DB_HOST: db:3306
       WORDPRESS_DB_USER: wordpress
       WORDPRESS_DB_PASSWORD_FILE: /run/secrets/db_password
     secrets:
       - db_password
     networks:
       - wordpress-network

networks:
   wordpress-network:
     driver: overlay

secrets:
   db_password:
     file: .password.txt
   db_root_password:
     file: .root_password.txt
```

```
44
45 volumes:
46     db_data:
47 EOF
```

Remove all the previous services to free up resources and ports:

```
1 docker service rm $(docker service ls -q)
```

Create the secrets:

```
1 echo "your_db_root_password" > .root_password.txt
2 echo "your_db_wordpress_password" > .password.txt
```

Now, if you were working in your development environment, you could run the following command to start the stack:

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

But since our goal is to deploy the stack on a Docker Swarm cluster, we need to run the following command:

```
1 docker stack deploy -c docker-compose.yml my-stack
```

Now, we can check the services:

```
1 docker service ls
```

You should see that both my-stack_db and my-stack_wordpress are running.

That's it! You have successfully deployed your first stack on Docker Swarm.

Now, let's learn how to add constraints to the services. For instance, if we want to deploy the WordPress service on the worker nodes and the MySQL service on the manager node, we need to add the following constraints to the services:

- WordPress: `node.role==worker`
- MySQL: `node.role==manager`

Here is the updated `docker-compose.yml` file:

```
cat <<EOF > docker-compose.yml
version: '3.9'

services:
   db:
     image: mysql:latest
     volumes:
       - db_data:/var/lib/mysql
     environment:
       MYSQL_ROOT_PASSWORD_FILE: /run/secrets/db_root_password
       MYSQL_DATABASE: wordpress
       MYSQL_USER: wordpress
       MYSQL_PASSWORD_FILE: /run/secrets/db_password
     secrets:
       - db_root_password
       - db_password
     networks:
       - wordpress-network
     deploy:
       placement:
         constraints:
           - node.role==manager

   wordpress:
     depends_on:
       - db
     image: wordpress:latest
     ports:
       - "8000:80"
     environment:
       WORDPRESS_DB_HOST: db:3306
       WORDPRESS_DB_USER: wordpress
       WORDPRESS_DB_PASSWORD_FILE: /run/secrets/db_password
     secrets:
       - db_password
     networks:
       - wordpress-network
     deploy:
       placement:
         constraints:
           - node.role==worker

networks:
```

```
    wordpress-network:
      driver: overlay

secrets:
    db_password:
      file: .password.txt
    db_root_password:
      file: .root_password.txt

volumes:
    db_data:
EOF
```

To update the stack, use the following command:

```
docker stack deploy -c docker-compose.yml my-stack
```

Check where the services are running:

```
docker service ps  my-stack_db
docker service ps  my-stack_wordpress
```

You should verify that the WordPress service is running on the worker nodes as intended, and the MySQL service is running on the manager node.

Next, we need to add memory and CPU reservations and limits to the services, and run 3 replicas of the WordPress service. Here is the updated `docker-compose.yml` file:

```
cat <<EOF > docker-compose.yml
version: '3.9'

services:
   db:
     image: mysql:latest
     volumes:
       - db_data:/var/lib/mysql
     environment:
       MYSQL_ROOT_PASSWORD_FILE: /run/secrets/db_root_password
       MYSQL_DATABASE: wordpress
       MYSQL_USER: wordpress
       MYSQL_PASSWORD_FILE: /run/secrets/db_password
     secrets:
       - db_root_password
```

```
16        - db_password
17      networks:
18        - wordpress-network
19      deploy:
20        placement:
21          constraints:
22            - node.role==manager
23        resources:
24          limits:
25            cpus: '1'
26            memory: 800M
27          reservations:
28            cpus: '0.5'
29            memory: 500M
30
31    wordpress:
32      depends_on:
33        - db
34      image: wordpress:latest
35      ports:
36        - "8000:80"
37      environment:
38        WORDPRESS_DB_HOST: db:3306
39        WORDPRESS_DB_USER: wordpress
40        WORDPRESS_DB_PASSWORD_FILE: /run/secrets/db_password
41      secrets:
42        - db_password
43      networks:
44        - wordpress-network
45      deploy:
46        placement:
47          constraints:
48            - node.role==worker
49        replicas: 3
50        resources:
51          limits:
52            cpus: '1'
53            memory: 800M
54          reservations:
55            cpus: '0.5'
56            memory: 500M
57
58  networks:
```

```
    wordpress-network:
      driver: overlay

secrets:
    db_password:
      file: .password.txt
    db_root_password:
      file: .root_password.txt

volumes:
    db_data:
EOF
```

Each time we have a new version of the `docker-compose.yml` file, we need to update the stack using the following command:

```
docker stack deploy -c docker-compose.yml my-stack
```

So there is no need to remove the stack and create it again.

Docker Swarm Rolling Updates

In the previous example, we updated the stack by executing the `docker stack deploy` command. However, how can we achieve the same operation while minimizing downtime? This is where rolling updates come into play.

A rolling update is a deployment strategy that involves updating the services one by one.

Let's see how to accomplish this. First, we need to modify the `docker-compose.yml` file to use the rolling update strategy:

```
cat <<EOF > docker-compose.yml
version: '3.9'

services:
   db:
     image: mysql:latest
     volumes:
       - db_data:/var/lib/mysql
     environment:
       MYSQL_ROOT_PASSWORD_FILE: /run/secrets/db_root_password
       MYSQL_DATABASE: wordpress
```

```
12        MYSQL_USER: wordpress
13        MYSQL_PASSWORD_FILE: /run/secrets/db_password
14      secrets:
15        - db_root_password
16        - db_password
17      networks:
18        - wordpress-network
19      deploy:
20        placement:
21          constraints:
22            - node.role==manager
23        resources:
24          limits:
25            cpus: '1'
26            memory: 800M
27          reservations:
28            cpus: '0.5'
29            memory: 500M
30        update_config:
31          parallelism: 1
32          delay: 10s
33          order: start-first
34
35    wordpress:
36      depends_on:
37        - db
38      image: wordpress:latest
39      ports:
40        - "8000:80"
41      environment:
42        WORDPRESS_DB_HOST: db:3306
43        WORDPRESS_DB_USER: wordpress
44        WORDPRESS_DB_PASSWORD_FILE: /run/secrets/db_password
45      secrets:
46        - db_password
47      networks:
48        - wordpress-network
49      deploy:
50        placement:
51          constraints:
52            - node.role==worker
53        replicas: 3
54        resources:
```

```
          limits:
            cpus: '1'
            memory: 800M
          reservations:
            cpus: '0.5'
            memory: 500M
        update_config:
          parallelism: 1
          delay: 10s
          order: start-first

networks:
    wordpress-network:
      driver: overlay

secrets:
    db_password:
      file: .password.txt
    db_root_password:
      file: .root_password.txt

volumes:
    db_data:
EOF
```

This is what we added to the `docker-compose.yml` file:

- `update_config`: This is used to configure the rolling update strategy.
- `parallelism`: This is used to specify the number of containers that can be updated in parallel. In our example, we set it to 1.
- `delay`: This is used to specify the delay between updates. In our example, we set it to 10 seconds.
- `order`: This is used to specify the order of the updates. In our example, we set it to `start-first` which means that the new containers will be started before the old ones are stopped. The default value is `stop-first` which means that the old containers will be stopped before the new ones are started.

Now, we need to update the stack using the following command:

```
docker stack deploy -c docker-compose.yml my-stack
```

To make our deployment more stable, we can also add other configurations like the healthcheck, restart policy and the the failure action to the `docker-compose.yml` file:

```
cat <<EOF > docker-compose.yml
version: '3.9'

services:
  db:
    image: mysql:latest
    volumes:
      - db_data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD_FILE: /run/secrets/db_root_password
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD_FILE: /run/secrets/db_password
    secrets:
      - db_root_password
      - db_password
    networks:
      - wordpress-network
    deploy:
      placement:
        constraints:
          - node.role==manager
      resources:
        limits:
          cpus: '1'
          memory: 800M
        reservations:
          cpus: '0.5'
          memory: 500M
      update_config:
        parallelism: 1
        delay: 10s
        order: start-first
      restart_policy:
        condition: any
        delay: 5s
        max_attempts: 3
        window: 120s

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
```

```
    ports:
      - "8000:80"
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD_FILE: /run/secrets/db_password
    secrets:
      - db_password
    networks:
      - wordpress-network
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:80"]
      interval: 1m30s
      timeout: 10s
      retries: 3
    deploy:
      placement:
        constraints:
          - node.role==worker
      replicas: 3
      resources:
        limits:
          cpus: '1'
          memory: 800M
        reservations:
          cpus: '0.5'
          memory: 500M
      update_config:
        parallelism: 1
        delay: 10s
        order: start-first
        failure_action: rollback
      rollback_config:
        parallelism: 0
        order: stop-first
      restart_policy:
        condition: any
        delay: 5s
        max_attempts: 3
        window: 120s

networks:
  wordpress-network:
```

```
87      driver: overlay
88
89  secrets:
90    db_password:
91      file: .password.txt
92    db_root_password:
93      file: .root_password.txt
94
95  volumes:
96    db_data:
97  EOF
```

Here is what we added so far:

First, we added the `restart_policy`. This is used to configure the restart policy. In our example, we set the condition to `any` which means that the container will be restarted regardless of the exit code. We also set the delay to 5 seconds, the max attempts to 3 and the window to 120 seconds.

The `condition` can be set to one of the following values:

- `none`
- `on-failure`
- `any`: This is the default value.

The `delay` is the time to wait between restart attempts. The `max_attempts` is the maximum number of attempts to restart a container. The `window` is the time window to evaluate the restart policy.

The `window` specifies how long to wait before deciding if a restart has succeeded.

We also added the `rollback_config` which is used to configure the rollback strategy.

In our example, we set the parallelism to `0` which means that all containers will rollback simultaneously. We also set the order to `stop-first` which means that the old containers will be stopped before the new ones are started.

In the Wordpress service, we configured the healthcheck. In our example, we set the test to `["CMD", "curl", "-f", "http://localhost:80"]` which means that the healthcheck will run the command `curl -f http://localhost:80` every 1 minute and 30 seconds. If the command fails, the healthcheck will be retried 3 times with a timeout of 10 seconds. When a container is considered unhealthy, it will be stopped and replaced with a new one. During the deployment, an unhealthy container will not receive any traffic.

We also added the `failure_action` which is used to configure the behavior when a task fails to update. In our example, we set it to `rollback` which means that the deployment will be rolled back when a task fails to update.

Now, we need to update the stack using the following command:

```
docker stack deploy -c docker-compose.yml my-stack
```

Using an External Load Balancer with Docker Swarm

Usually, a cluster resides in a private network and is not accessible from the outside world. This is a common and good practice as it adds an extra layer of security. It is not recommended to expose all the nodes of your cluster to the outside world. Therefore, it is common to use a front-end load balancer.

For instance, you can use an AWS Application Load Balancer (ALB), Traefik, HAProxy, or Nginx as a front-end load balancer. In this guide, we will focus on setting up an HAProxy load balancer.

To begin, create a new machine outside the cluster that is part of the same local network as the cluster. We will refer to this machine as "loadbalancer" and use Ubuntu 22.04 as the operating system.

Next, you need to install HAProxy on the loadbalancer machine:

```
sudo apt update
sudo apt install -y haproxy
```

Now, let's configure HAProxy. We need to add the following configuration to the `/etc/haproxy/haproxy.cfg` file:

```
# export Wordpress service port
export PORT=8000
export MANAGER_IP=<MANAGER_IP>
export WORKER01_IP=<WORKER01_IP>
export WORKER02_IP=<WORKER02_IP>

# create HAProxy configuration file
cat <<EOF >> /etc/haproxy/haproxy.cfg
# Configure HAProxy to listen on port 80
frontend lb
   bind *:80
   stats uri /haproxy?stats
   default_backend wordpress

# Route to all nodes in the cluster on port 8000 to reach the wordpress application
backend wordpress
   balance roundrobin
   server manager  $MANAGER_IP:$PORT  check
   server worker01 $WORKER01_IP:$PORT check
   server worker02 $WORKER02_IP:$PORT check
EOF
```

Make sure to change the values of `MANAGER_IP`, `WORKER01_IP` and `WORKER02_IP` with the right values. Now, restart HAProxy:

```
sudo systemctl restart haproxy
```

Now, you should be able to access the WordPress application from the load balancer IP. It will be listening on port 80 and redirecting traffic to the WordPress service on port 8000 using the round robin algorithm. This ensures that traffic is evenly distributed among the nodes. If desired, you can change the algorithm to use another method, such as leastconn, where HAProxy sends traffic to the node with the fewest number of connections.

The HAProxy configuration in the provided example is static, meaning it needs to be manually updated for any changes in the Docker Swarm, such as adding or removing nodes or services. Not only is this error-prone in dynamic environments, but updating the HAProxy configuration file itself is also a manual process. To overcome this challenge, we can use an Ingress instead of a simple load balancer.

To continue using HAProxy, it is recommended to incorporate a service discovery tool like Consul. Consul is a service discovery tool that enables the discovery of services within a cluster. Additionally, it can automatically update the HAProxy configuration file. For more information, refer to the HAProxy and Consul with DNS for Service Discovery blog post[144] and the official Consul website[145].

There are also other alternatives like Traefik[146], which is a modern HTTP reverse proxy, load balancer, and service mesh. It supports multiple backends such as Docker Swarm, Kubernetes, Amazon ECS, Rancher, and more. Traefik natively integrates with Docker Swarm and can automatically discover services, react to changes in the Swarm, and adjust its routing rules dynamically without needing to restart or reconfigure. Traefik is particularly well-suited for Docker Swarm environments due to its seamless integration, automatic service discovery, and ease of configuration, especially in dynamic and frequently changing environments.

There are other alternatives like:

- Docker Flow Proxy: A project aimed at creating a reconfigurable proxy for Docker Swarm. It works in conjunction with Docker Flow Swarm Listener to automatically reconfigure itself when services are scaled up or down.
- Swarm-Router: 1 “zero config” ingress router for Docker swarm mode deployments, based on the mature and superior haproxy library and a little of golang offering unique advantages
- Envoy Proxy: Although not specifically designed for Docker Swarm, Envoy Proxy can be used within a Swarm setup. Envoy is a high-performance distributed proxy designed for cloud-native applications.
- Caddy-Docker-Proxy: A plugin that enables Caddy[147] to be used as a reverse proxy for Docker containers via labels.

[144] https://www.haproxy.com/blog/haproxy-and-consul-with-dns-for-service-discovery
[145] https://www.consul.io/
[146] https://traefik.io/
[147] https://caddyserver.com/

Let's see an example of how to use Traefik with Docker Swarm in the next section.

Using Traefik as a Front-End Load Balancer with Docker Swarm

Let's start with a basic example. Create a network and two stacks:

- A network called `whoami` that will be used by the whoami service and Traefik.
- A Traefik stack that contains a Traefik service.
- A sample stack that contains a whoami service.

Traefik has to access to the Docker Swarm API which is only available on manager nodes. This is why we need to deploy Traefik on a manager node.

When Traefik receives a request, it will check the routing rules and forward the request to the right service. In our example, we will use the following routing rule: `Host(whoami.$MANAGER_NODE_IP.nip.io)`. This means that Traefik will forward the request to the whoami service when the request is sent to `whoami.$MANAGER_NODE_IP.nip.io`.

nip.io[148] is a DNS service that provides wildcard DNS for any IP address. This will allow us to access the whoami service using the following URL: `whoami.$MANAGER_NODE_IP.nip.io`.

If we need to deploy more services and need to expose them to the outside world, we can use other routing rules: domains, subdomains, paths, etc.

How Traefik works

It is also worth noting that our setup is a cluster with 1 manager and 1 worker.

```
# create the traefik stack
cd $HOME && mkdir -p traefik && cd traefik && cat <<EOF > docker-compose.yml
# Docker Compose version
version: '3.9'
# Define services
services:
  # Reverse proxy service using Traefik
  reverse-proxy:
    # Use Traefik version 2.10 image
    image: traefik:v2.10
    # Traefik command-line options
```

[148] https://nip.io

```
    command:
      # Enable insecure API (for testing purposes)
      - "--api.insecure=true"
      # Enable Docker as a provider
      - "--providers.docker"
      # Enable Docker Swarm mode
      - "--providers.docker.swarmMode=true"
      # Enable access log
      - "--accesslog=true"
      # Set log level to DEBUG
      - "--log.level=DEBUG"
      # Set log format to JSON
      - "--log.format=json"
    # Expose ports for HTTP traffic and the Traefik dashboard
    ports:
      # Expose port 80 for HTTP traffic
      - "80:80"
      # Expose port 8080 for the Traefik dashboard
      - "8080:8080"
    # Mount the Docker socket for dynamic configuration
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    deploy:
      labels:
        # Disable Traefik for this service
        traefik.enable: "False"
      placement:
        constraints:
          # Deploy this service only on manager nodes in Docker Swarm
          - node.role == manager
    networks:
      # Connect this service to the 'whoami' external network
      - whoami
# Define external networks
networks:
  # External network named 'whoami'
  whoami:
    external: true
EOF

# make sure to change MANAGER_NODE_IP by the external IP of the manager node
export MANAGER_NODE_IP=<MANAGER_NODE_IP>

```

```
# create the whoami stack
cd $HOME && mkdir -p whoami && cd whoami && cat <<EOF > docker-compose.yml
version: '3.9'
services:
  # Service named 'whoami' using the containous/whoami image
  whoami:
    image: containous/whoami
    deploy:
      # Deploy 2 replicas of the 'whoami' service
      replicas: 2
      labels:
        # Enable Traefik for this service
        traefik.enable: "True"
        # Configure the Traefik routing rule
        # Change [MANAGER_NODE_IP] by the external IP of the manager node
        traefik.http.routers.whoami.rule: Host(\`whoami.$MANAGER_NODE_IP.nip.io\`)
        # Use the 'http' entrypoint for routing
        traefik.http.routers.whoami.entrypoints: http
        # Set the load balancer port to 80
        traefik.http.services.whoami.loadbalancer.server.port: 80
    networks:
      # Connect this service to the 'whoami' network
      - whoami
# Define external networks
networks:
  # External network named 'whoami'
  whoami:
    external: true
EOF
```

After creating the above resources, we need to execute the following commands:

```
# create the network
docker network create --driver overlay --attachable whoami
# create the traefik stack
cd $HOME && docker stack deploy -c traefik/docker-compose.yml traefik
# create the whoami stack
cd $HOME && docker stack deploy -c whoami/docker-compose.yml whoami
```

Now, you should be able to access the whoami service using the following URL: `whoami.<manager_node_ip>.nip.io`.

Now, if you want to do the same thing for the Wordpress stack that we created in the previous section, you need to adapt your compose file by adding the right labels to the WordPress service.

Let's see the full `docker-compose.yml` file:

```
# create the folder wordpress
cd $HOME && mkdir -p wordpress
# add the secrets
echo "your_db_root_password" > wordpress/.root_password.txt
echo "your_db_wordpress_password" > wordpress/.password.txt

# export MANAGER_NODE_IP
MANAGER_NODE_IP=<MANAGER_NODE_IP>

# create the network
docker network create --driver overlay --attachable wordpress

# create the yaml file
# make sure to change [MANAGER_NODE_IP] by the external IP of the manager node
cd $HOME && cat <<EOF > wordpress/docker-compose.yml
version: '3.9'
services:
  db:
    image: mysql:latest
    volumes:
      - db_data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD_FILE: /run/secrets/db_root_password
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD_FILE: /run/secrets/db_password
    secrets:
      - db_root_password
      - db_password
    networks:
      - internal
    deploy:
      placement:
        constraints:
          - node.role==manager
      resources:
        limits:
          cpus: '1'
          memory: 800M
        reservations:
          cpus: '0.5'
```

```
42            memory: 500M
43        update_config:
44          parallelism: 1
45          delay: 10s
46          order: start-first
47        restart_policy:
48          condition: any
49          delay: 5s
50          max_attempts: 3
51          window: 120s
52    wordpress:
53      depends_on:
54        - db
55      image: wordpress:latest
56      # comment the ports
57      # Traefik will handle the routing to the WordPress service on port 80.
58      #ports:
59      #  - "8000:80"
60      environment:
61        WORDPRESS_DB_HOST: db:3306
62        WORDPRESS_DB_USER: wordpress
63        WORDPRESS_DB_PASSWORD_FILE: /run/secrets/db_password
64      secrets:
65        - db_password
66      networks:
67        - wordpress
68        - internal
69      healthcheck:
70        test: ["CMD", "curl", "-f", "http://localhost:80"]
71        interval: 1m30s
72        timeout: 10s
73        retries: 3
74      deploy:
75        # add labels
76        labels:
77          # Enable Traefik for this service
78          traefik.enable: "True"
79          # Configure the Traefik routing rule
80          # Change [MANAGER_NODE_IP] by the external IP of the manager node
81          traefik.http.routers.wordpress.rule: Host(\`wordpress.$MANAGER_NODE_IP.nip.i\
82  o\`)
83          # Use the 'http' entrypoint for routing
84          traefik.http.routers.wordpress.entrypoints: http
```

```
        # Set the load balancer port to 80 of the WordPress service
        # By default, Traefik uses the first exposed port of a container unless you \
specify the port.
        traefik.http.services.wordpress.loadbalancer.server.port: 80
      placement:
        constraints:
          - node.role==worker
      replicas: 3
      resources:
        limits:
          cpus: '1'
          memory: 800M
        reservations:
          cpus: '0.5'
          memory: 500M
      update_config:
        parallelism: 1
        delay: 10s
        order: start-first
        failure_action: rollback
      rollback_config:
        parallelism: 0
        order: stop-first
      restart_policy:
        condition: any
        delay: 5s
        max_attempts: 3
        window: 120s
networks:
  wordpress:
    external: true
  internal:
    driver: overlay
secrets:
  db_password:
    file: .password.txt
  db_root_password:
    file: .root_password.txt
volumes:
  db_data:
EOF
```

You need to update the Traefik configuration to add the new network:

```
# create the traefik stack
cd $HOME && mkdir -p traefik && cat <<EOF > traefik/docker-compose.yml
# Docker Compose version
version: '3.9'
# Define services
services:
  # Reverse proxy service using Traefik
  reverse-proxy:
    # Use Traefik version 2.10 image
    image: traefik:v2.10
    # Traefik command-line options
    command:
      # Enable insecure API (for testing purposes)
      - "--api.insecure=true"
      # Enable Docker as a provider
      - "--providers.docker"
      # Enable Docker Swarm mode
      - "--providers.docker.swarmMode=true"
      # Enable access log
      - "--accesslog=true"
      # Set log level to DEBUG
      - "--log.level=DEBUG"
      # Set log format to JSON
      - "--log.format=json"
    # Expose ports for HTTP traffic and the Traefik dashboard
    ports:
      # Expose port 80 for HTTP traffic
      - "80:80"
      # Expose port 8080 for the Traefik dashboard
      - "8080:8080"
    # Mount the Docker socket for dynamic configuration
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    deploy:
      labels:
        # Disable Traefik for this service
        traefik.enable: "False"
      placement:
        constraints:
          # Deploy this service only on manager nodes in Docker Swarm
          - node.role == manager
    networks:
      - whoami
```

```
      - wordpress
# Define external networks
networks:
  # External network named 'whoami'
  whoami:
    external: true
  # External network named 'wordpress'
  wordpress:
    external: true
EOF
```

Deploy the wordpress stack and update the traefik one:

```
cd $HOME/wordpress &&  docker stack deploy -c docker-compose.yml wordpress
cd $HOME/traefik && docker stack deploy -c docker-compose.yml traefik
```

We have deployed the WordPress stack, the Whoami stack and the Traefik stack. Now, we should be able to:

- access the WordPress application using the following URL: `wordpress.<manager_node_ip>.nip.io`.
- access the Whoami application using the following URL: `whoami.<manager_node_ip>.nip.io`.

Here, we can change the nip.io service to a real domain name and use a DNS provider like Cloudflare to manage the DNS records. We could also use two subdomains for both services or use paths instead of subdomains.

Example:

```
rule = "Host(`example.com`) || (Host(`example.org`) && Path(`/wordpress`))"
```

Docker Swarm Logging

Docker Swarm has a built-in logging system that allows you to see logs from all the services in the cluster. However, logs are not stored and should be sent to an external logging system like ELK or Loki.

This is how to see the logs of a service:

```
export service_name=my-stack_wordpress
docker service logs $service_name
```

To follow the logs, use the following command:

```
docker service logs -f $service_name
# or docker service logs --follow $service_name
```

You can see logs since a specific time using the `--since` option:

```
docker service logs --since 10m $service_name
# or docker service logs --since=2023-10-01T00:00:00 $service_name
```

Other options are available:

- `--tail`: This is used to show the last N lines of the logs. The default value is `all`.
- `--details`: This is used to show extra details provided to logs.
- `--no-trunc`: This is used to show the full logs instead of truncating them.
- `--timestamps`: This is used to show timestamps.

Docker Swarm vs. Kubernetes

Although Kubernetes is the most popular container orchestration tool, Docker Swarm remains a viable choice for many use cases. Here are some advantages of Docker Swarm over Kubernetes:

- Docker Swarm is easier to set up and use compared to Kubernetes.
- Docker Swarm is more lightweight than Kubernetes.
- Docker Swarm is more accessible than Kubernetes.
- Docker Swarm is built-in to Docker Engine.
- Docker Swarm scales faster than Kubernetes.
- Docker Swarm provides automatic load balancing between containers of the same service.

Creating a Swarm cluster can be as simple as running the following command:

```
docker swarm init
```

Creating a Kubernetes cluster requires more steps and extensive configuration compared to Swarm. However, Swarm's simplicity comes with limitations, including the lack of support for multiple clusters (federation), multiple runtimes, automatic scaling, and fewer integrations compared to Kubernetes.

Despite these differences, there are many shared features between Swarm and Kubernetes. Both are open source, declarative, self-healing, highly available, scalable, portable, and extensible. Concepts like load balancing, ingress, and service discovery are also common to both platforms.

Docker Swarm is a suitable choice for small and medium projects, as well as projects that prioritize a simple setup and don't require advanced features like federation, multiple runtimes, or automatic scaling.

While Kubernetes is more widely adopted than Docker Swarm, it's worth noting that some users may be over-engineering their projects by using Kubernetes when it is not necessary. Kubernetes is better suited for larger projects, and some teams are even using both Swarm and Kubernetes to benefit from the advantages of both platforms.

Additionally, Swarm can serve as a starting point for projects that may be migrated to Kubernetes in the future.

Docker Desktop

What is Docker Desktop?

Docker Desktop is a user-friendly, one-click-install application designed for Mac, Linux, and Windows environments. With Docker Desktop, you can build, share, and run containerized applications and microservices.

This tool provides a Graphical User Interface that simplifies container management, application deployment, and image handling directly on your local machine. You have the flexibility to use Docker Desktop on its own or in conjunction with the command-line interface (CLI).

According to Docker inc., Docker Desktop reduces the time spent on complex setups so you can focus on writing code. It takes care of port mappings, file system concerns, and other default settings, and is regularly updated with bug fixes and security updates.

These are some of the features that Docker Desktop offers:

- Docker Engine: The core Docker technology that enables container creation and management
- Docker CLI client: The command-line interface that allows you to interact with Docker Engine
- Docker Scout (paid): A tool that helps you find and fix vulnerabilities in your Docker images
- Docker Buildx: A CLI plugin that extends the Docker CLI with the full support of the features provided by Moby BuildKit builder toolkit
- Docker Extensions: A set of tools that extend the functionality of Docker Desktop
- Docker Compose: A tool for defining and running multi-container Docker applications especially useful in development environments
- Docker Content Trust: A feature that provides the ability to use digital signatures for data sent to and received from remote Docker registries
- Kubernetes: An open-source container orchestration system for automating deployment, scaling, and management of containerized applications
- Credential Helper: A suite of programs to use native stores to keep Docker credentials safe.

How to Install Docker Desktop

Docker Desktop works on Mac, Linux, and Windows. The installation process is straightforward and can be completed in a few minutes.

The following sections provide instructions for installing Docker Desktop on Ubuntu 22.04. Mac users should follow the instructions in this page[149], and Windows users should follow the instructions in this page[150]. Other Linux distributions like Fedora, Debian-based and Arch-based distributions are

[149] https://docs.docker.com/desktop/install/mac-install/
[150] https://docs.docker.com/desktop/windows/install/

also supported. You can find the installation instructions for these distributions here[151].

For Ubuntu, you can install Docker Desktop using the following steps:

For non-Gnome Desktop environments, start by installing the Gnome terminal:

```
sudo apt install gnome-terminal
```

Uninstall any previous versions of Docker Desktop:

```
rm -r $HOME/.docker/desktop
sudo rm /usr/local/bin/com.docker.cli
sudo apt purge docker-desktop
```

Install Docker Desktop:

```
sudo apt-get update
# docker-desktop-4.24.2 is the latest version at the time of writing this guide
wget https://desktop.docker.com/linux/main/amd64/docker-desktop-4.24.2-amd64.deb
sudo apt install ./docker-desktop-4.24.2-amd64.deb
```

You can now start Docker Desktop from the Applications menu or by running the following command:

```
systemctl --user start docker-desktop
```

To enable Docker Desktop to start on login use the following command:

```
systemctl --user enable docker-desktop
```

To stop Docker Desktop use the following command:

```
systemctl --user stop docker-desktop
```

[151] https://docs.docker.com/desktop/install/linux-install/

Common Security Threats

Docker vs. VMs: Which is more secure?

Docker containers are often compared to virtual machines for pedagogical purposes. Many Docker introductions highlight the advantages of Docker containers, such as being lighter, faster, and not requiring a hypervisor or guest OS. However, it is important to consider security when comparing Docker containers to virtual machines.

Let's examine some common differences between Docker containers and virtual machines in terms of security.

Virtual Machines:

(-) Virtual machines run a full guest OS, which increases the attack surface.

(+) Virtual machines provide strong isolation by running each VM as a separate guest OS on top of the host OS, managed by the hypervisor. This hardware-level isolation can offer better security against certain types of attacks.

(+) Virtualization technology has been around for a long time and is well understood and mature, whereas Docker is still catching up.

Docker Containers:

(-) Containers share the same kernel as the host OS, meaning that a compromised container could potentially compromise the host OS.

(-) Containerization is a relatively new technology compared to virtualization, so it is not as mature in terms of security.

(+) Docker containers are more lightweight than VMs, resulting in a smaller attack surface.

(+) Containers are lighter and have less overhead, making it faster and easier to patch them compared to VMs.

(+) Docker containers are immutable, meaning they cannot be changed once built. This makes it easier to detect changes and potential security breaches.

Conclusion: It is important to note that there is no definitive conclusion when comparing VMs and Docker containers in terms of security. In most cases, Docker users do not use Docker containers as a replacement for VMs, but rather in conjunction with them. Docker containers are used for running applications, improving portability, breaking applications into microservices, creating development environments, etc., while VMs are used to run Docker containers. Therefore, both technologies are used together.

Docker is neither more nor less secure than a VM, and the VM vs. Docker security discussions are addressing the wrong question. Many problems, especially in the security field, are often PEBCAK problems (PEBCAK: Problem Exists Between Chair And Keyboard). Thus, a more constructive question would be: "How can I enhance the security of my Docker containers?".

Next, we will explore common security threats to be aware of when using Docker and introduce best practices to enhance the security of your Docker containers.

Kernel Panic & Exploits

A container is a process that runs on top of the host OS and has direct access to the Kernel. However, this direct access, which is one of Docker's strengths, can also lead to serious damages. If a container triggers a Kernel panic, it can potentially bring down both the host and other containers.

Consider a scenario where a Docker container is running an application that requires access to low-level system or kernel resources, such as network operations. If there is a bug in the application or the kernel module providing access to these resources, the container can crash the kernel and cause the entire host to go down.

Container Breakouts & Privilege Escalation

If you start the container "X" with the user "Y", container "X" will have the same privileges on the host system as the user "Y". This poses a risk when a process breaks out of the container. In such cases, if you were root in the container, you would also have root access on the host system.

Container breakout can lead to unauthorized access across containers, hosts, and even data centers.

The Dirty COW vulnerability (CVE-2016-5195)[152] provides a clear example of how kernel vulnerabilities can be exploited in container environments through privilege escalation.

Below is a basic example demonstrating how to gain access to the root user of the host system from within a container:

```
# Create a hidden file containing the line to add to /etc/passwd
# The output of the command is redirected to /.hidden
# The hidden file will contain something like: myroot:$1$mysalt$mypassword:0:0:root:\
/root:/bin/bash
echo "myroot:$(openssl passwd -1 -salt mysalt mypassword):0:0:root:/root:/bin/bash" \
 > /.hidden
# Start a container and add the hidden file content to /etc/passwd
docker run -ti -v /:/mnt/ --name container alpine sh -c "cat /mnt/.hidden >> /mnt/et\
c/passwd"
```

[152] https://github.com/scumjr/dirtycow-vdso

```
# Now you have access to the root user of the host system
su - myroot
whoami
```

Another example of a DoS attack is the CVE-2019-5736[153] vulnerability. It enables attackers to overwrite the host runc binary[154] and gain root-level code execution on the host. This vulnerability was discovered by Adam Iwaniuk and Borys Popławski and has been patched in runc v1.0.0-rc6.

Poisoned Images

It is possible that you download and use a Docker image that runs malware (e.g., scanning the network for sensitive data, downloading malware from a distant host, executing harmful actions, cryptojacking[155] .. etc.). An attacker can also get access to your data if you are using his poisoned image.

In 2019, just a few years after the release of the first Docker release, Unit 42 researchers[156] discovered the first-ever cryptojacking worm on Docker Hub. The worm, dubbed Graboid, was hidden in a Docker image that was downloaded more than 10000 times. The worm was able to spread to other containers and mine Monero cryptocurrency. This is how the worm worked:

- The attacker targets an unsecured Docker host and remotely commands it to download and deploy the malicious Docker image pocosow/centos:7.6.1810. This image contains a Docker client tool for communicating with other Docker hosts.
- The container's entry point script, "/var/sbin/bash", downloads four shell scripts from a Command and Control (C2) server: "live.sh", "worm.sh", "cleanxmr.sh", and "xmr.sh", executing them sequentially.
    1. "live.sh": Reports the number of available CPUs on the compromised host back to the C2 server.
    2. "worm.sh": Downloads a list of over 2000 IP addresses, which are hosts with unsecured Docker API endpoints. It then randomly selects an IP from this list and uses the Docker client tool to remotely deploy the pocosow/centos container on the chosen host.
    3. "cleanxmr.sh": Selects a vulnerable host from the IP list and stops cryptojacking containers running on it, including gakeaws/nginx and other containers based on xmrig.
    4. "xmr.sh": Picks a host from the IP list and deploys the gakeaws/nginx image on it. This image contains a disguised xmrig binary, masquerading as nginx, for cryptojacking.

Through this process, repeated over and over again on every host, Graboid spreads from host to host, stopping competitors' cryptojacking operations and initiating its own, leveraging the computational resources of the compromised hosts (CPUs and GPUs) for Monero mining.

[153] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5736
[154] https://asciinema.org/a/226969
[155] https://unit42.paloaltonetworks.com/graboid-first-ever-cryptojacking-worm-found-in-images-on-docker-hub/
[156] https://unit42.paloaltonetworks.com/

Denial-of-service Attacks

All containers on the same host share the same Kernel as the host system. Consequently, if a container monopolizes the Kernel resources, it can cause other containers to be deprived. These "poisoned" containers can consume CPU time and memory resources, potentially causing other containers and even the host system to crash.

Compromising secrets

A Docker container has the potential to contain or transmit sensitive data. This data can be vulnerable to being viewed and stolen by attackers during transit or when at rest. The same security threats apply to microservices architecture, but there are effective solutions available to mitigate these risks.

In 2023, researchers at RWTH Aachen University in Germany conducted a study where they analyzed 337,171 images from Docker Hub and private registries. The study revealed that 8.5% of these images contained sensitive information, such as private keys and API secrets. Alarmingly, many of these exposed keys were actively being used for malicious purposes. This resulted in the exposure of 52,107 valid private keys and 3,158 distinct API secrets across 28,621 Docker images on Docker Hub.

Application Level Threats

A container is primarily designed for building, shipping, and running applications. However, even if the host and container are secured, the application layer can still be vulnerable to attacks.

Some common threats at the application level include:

- XSS vulnerabilities and SQL injection
- Insecure deserialization
- Broken authentication and session management
- Exposure of sensitive data
- Broken access control
- Security misconfiguration
- Insecure direct object references
- Use of components with known vulnerabilities
- … and so on.

Host System Level Treats

If the host system is not up to date, you may have security threats such as Heartbleed, Shellshock (Bashdoor), and other vulnerabilities.

Docker Security Best Practices

Implement Security by Design

Designing a system to be secure from the start is a good practice, just like with any other security threat. The application layer, containerization layer, host system, software, and cloud architecture, among others, are all components of the same running stack. Any of these elements could potentially be the weakest link in your system.

Security should not be an afterthought. It should be incorporated into the development process right from the beginning. A common mistake is adding security measures at the end of the development process, which is not advisable. Instead, security should be integrated into the CI/CD pipeline using a DevSecOps approach.

By designing a system to be secure from the outset, you can mitigate the damage in the event of a container breakout.

setuid/setgid Binaries

In Linux, every file and program has permissions that determine who can read, write, or execute them. Normally, when you run a program, it operates with your user permissions. However, there are times when a program needs to perform actions that require higher permissions than those of a regular user. This is where `setuid` (Set User ID) and `setgid` (Set Group ID) come into play.

In simple terms, `setuid` and `setgid` are special settings that allow regular users to temporarily execute specific programs with elevated permissions. These settings utilize a specific bit in the file's permissions to grant temporary elevated privileges. With `setuid`, the user's privileges are elevated to those of the file's owner, while `setgid` elevates them to the group level.

Here's a common example using the passwd command, which typically has the `setuid` bit set:

```
ls -l /usr/bin/passwd
```

The output will be something like:

```
-rwsr-xr-x 1 root root 59976 Nov 24  2022 /usr/bin/passwd
```

Notice the "s" in the permissions part (`-rwsr-xr-x`). This "s" indicates that the `setuid` bit is set.

The `setuid` and `setgid` bits, if compromised by an attacker, can be used to gain elevated privileges.

We can find these binaries using the following command:

```
find / -perm +6000 -type f -exec ls -ld {} \; 2>/dev/null
```

We can also remove the `setuid` and `setgid` bits from all binaries using the following command inside the Dockerfile:

```
RUN find / -perm +6000 -type f -exec chmod a-s {} \; || true
```

This is a prevention from some privilege escalation threats and it will typically be applied to files like:

```
/sbin/unix_chkpwd
/usr/bin/chage
/usr/bin/passwd
/usr/bin/mail-touchlock
/usr/bin/mail-unlock
/usr/bin/gpasswd
/usr/bin/crontab
/usr/bin/chfn
/usr/bin/newgrp
/usr/bin/sudo
/usr/bin/wall
/usr/bin/mail-lock
/usr/bin/expiry
/usr/bin/dotlockfile
/usr/bin/chsh
/usr/lib/eject/dmcrypt-get-device
/usr/lib/pt_chown
/bin/ping6
/bin/su
/bin/ping
/bin/umount
/bin/mount
```

However, this approach is quite aggressive as it will remove the setuid and setgid bits from many system files. This could potentially break some applications. For instance, the “ping” command requires setuid to function properly, as it needs access to raw network sockets.

Control Resources

By default, when multiple containers are running on a Docker host without specific resource limits, they share the CPU and memory resources of the host in an unrestricted manner. Here’s a general overview of how this works:

- Docker uses completely fair scheduling (CFS) under the Linux kernel to distribute CPU time among containers. This means that if no specific CPU limits are set for containers, they can use the CPU as needed, competing equally for CPU time.
- Memory is also shared among containers. By default, there's no hard limit on how much memory a container can use, unless specified.

In environments where multiple containers are running, and if some of them are resource-intensive, this can lead to resource contention. Containers may compete for CPU and memory, potentially degrading performance. Therefore, it's important to control the resources used by each container. Docker provides flags to control the resources used by a container. This helps avoid DoS attacks and limit damage in the case of a container breakout.

Here's an example command that demonstrates how to set resource limits when running a Docker container:

```
1 docker run -d --name my_container --memory=512m --cpus=1.5 ubuntu:latest
```

- `--memory=512m`: Limits the container's memory usage to 512 megabytes. This prevents the container from using excessive memory, which could impact other containers or the host system.
- `--cpus=1.5`: Limits the container to use a maximum of 1.5 CPUs. This means the container can use one full CPU and half of another.

Use Notary to Verify Image Integrity

Be cautious when pulling or using images from public repositories. Always verify the integrity of the images you use. You can use the `docker trust inspect` command to check the image's integrity. This command verifies the image's signature and the publisher's identity.

To avoid security issues, only use images from trusted sources. You can also use the `docker trust sign` command to sign images you build yourself.

Additionally, use images from automated builds with linked source code, official images, and trusted sources. A more secure approach is to build your own base images from scratch[157].

Let's take a look at an example of how to verify the integrity of an image. We will use Notary, a tool that provides trust over any content. Notary is built on the TUF (The Update Framework) specification and is part of the CNCF (Cloud Native Computing Foundation).

The TUF specification is a framework that offers a flexible solution to software update security. It is designed to be integrated into existing software development and publishing workflows. TUF provides a set of libraries and tools to secure software update systems against various attacks. It

[157] https://hub.docker.com/_/scratch

ensures that the software running on your system is the expected software and has not been modified by a malicious attacker.

Here's how the TUF specification works:

- The client developer sets up a TUF server and publishes the public keys to the client.
- When there's a new update, the developer signs the files with their keys. This signature is then added to the metadata.
- The client downloads the metadata and verifies the signature. If the signature is valid, the client downloads the update.

One notable early adoption of the TUF specification in the open-source community was by Docker Content Trust, which is an implementation of the Notary project from Docker. Notary, built on the TUF framework, serves two purposes:

1. It helps Docker publishers sign their images and verify their identity.
2. It helps Docker users verify the integrity of the images they use.

When you pull a signed image, Docker CLI communicates with the Notary server to verify the image's signature and the publisher's identity. If the image is signed and the signature is valid, the image is pulled. Otherwise, the pull operation fails.

Now, let's see how to sign and verify an image using Notary in practice. Since this is a development environment, we will use a self-signed certificate, and both the client and the server are on the same machine.

First, install Notary on the client side (the host you use to pull the image):

```
wget --no-check-certificate -O /usr/local/bin/notary https://github.com/docker/notar\
y/releases/download/v0.6.1/notary-Linux-amd64 && chmod a+x /usr/local/bin/notary
```

Then, you need to create a Notary repository on the server side:

```
# clone the notary repository
cd /
git clone https://github.com/theupdateframework/notary.git
# generate own certificates
cd /notary/fixtures
./regenerateTestingCerts.sh
# build and start the notary server and the notary signer
cd /notary/
docker compose build
docker compose up -d
# if you are testing on your localhost, add the following line to /etc/hosts
echo "127.0.0.1 notaryserver" >> /etc/hosts
```

Now, test the Notary server:

```
# remove the default certificate file
rm -f /etc/ssl/certs/root-ca.pem
# make the notray server certificate known to the machine
cp /notary/fixtures/intermediate-ca.crt /usr/local/share/ca-certificates/intermediat\
e-ca.crt && update-ca-certificates
```

Create an alias for the Notary client:

```
mkdir -p ~/.docker/trust
alias notary="notary -s https://notaryserver:4443 -d ~/.docker/trust --tlscacert /us\
r/local/share/ca-certificates/intermediate-ca.crt"
```

On the client, generate a trusted user key:

```
# generate a trusted user key
export DOCKER_USER=admin
docker trust key generate $DOCKER_USER --dir ~/.docker/trust
```

Adapt the `DOCKER_USER` variable to your needs.

We are going to use DockerHub as a registry, so you need to login to DockerHub:

```
docker login
```

Now let's see how to sign an image:

```
# update this variable with your DockerHub username
export DOCKERHUB_USERNAME=[YOUR_DOCKERHUB_USERNAME]
# pull an image
docker pull alpine:latest
# tag the image
docker tag alpine:latest $DOCKERHUB_USERNAME/alpine:signtest
# switch to Docker trusted mode
export DOCKER_CONTENT_TRUST=1
export DOCKER_CONTENT_TRUST_SERVER=https://notaryserver:4443
# sign the image
docker trust sign $DOCKERHUB_USERNAME/alpine:signtest
# push the image
docker push $DOCKERHUB_USERNAME/alpine:signtest
```

The image is now signed and pushed to DockerHub. The metadata is also automatically uploaded to the Notary server at the same time. Let's see how to verify the image:

```
# remove the local image
docker rmi $DOCKERHUB_USERNAME/alpine:signtest
# pull the image
docker pull $DOCKERHUB_USERNAME/alpine:signtest
# check the signature
docker trust inspect --pretty $DOCKERHUB_USERNAME/alpine:signtest
```

If you want to add a second publisher, you can do it like this:

```
# generate a second trusted user key
export DOCKER_USER=another-admin
docker trust key generate $DOCKER_USER --dir ~/.docker/trust
# add the second publisher
docker trust signer add --key ~/.docker/trust/$DOCKER_USER.pub $DOCKER_USER $DOCKERH\
UB_USERNAME/alpine:signtest
# sign the image with the second publisher
docker trust sign $DOCKERHUB_USERNAME/alpine:signtest
# push the image
docker push $DOCKERHUB_USERNAME/alpine:signtest
```

Let's see the result of the verification:

```
# remove the local image
docker rmi $DOCKERHUB_USERNAME/alpine:signtest
# pull the image
docker pull $DOCKERHUB_USERNAME/alpine:signtest
# check the signature
docker trust inspect --pretty $DOCKERHUB_USERNAME/alpine:signtest
```

At this step, the client can only pull images signed by the first and second publishers. Only these users will be able to update the signed image.

If you have another Docker environment where you want to ensure that only trusted images are used, you need to enable Docker Content Trust. This can be done by setting an environment variable that instructs Docker to only pull signed images. Here's how to set it up:

```
export DOCKER_CONTENT_TRUST=1
```

Next, you need to inform Docker about the location of your Notary server. The Notary server stores the metadata and signatures for your trusted images. Share the URL of your Notary server with your Docker clients. For instance, if your Notary server's URL is notaryserver.com, you would use:

```
export DOCKER_CONTENT_TRUST_SERVER=https://notaryserver.com:4443
```

To establish trust between Docker and the Notary server, especially when using a self-signed certificate, you must install the server's certificate on each Docker client machine. Obtain the certificate file, usually located at "fixtures/intermediate-ca.crt" on the server, and transfer it to the client machine.

Then, add the certificate to the list of trusted certificates on the client machine:

```
# intermediate-ca.crt was copied from the server to the client
# copy the certificate to the trusted certificates directory
sudo cp intermediate-ca.crt /usr/local/share/ca-certificates/
# update the trusted certificates
sudo update-ca-certificates
```

Scan Images

Scanning images is a good practice to detect security problems, vulnerabilities, and best practice violations. There are many tools that can help you to scan your images. Some of them are:

- Docker Scout[158]: A tool by Docker to scan images for vulnerabilities and SBOM discovery.
- Clair[159]: An open-source project for the static analysis of vulnerabilities in application containers.
- Anchore[160]: A tool to protect cloud-native workloads with continuous vulnerability scanning for container images.
- Trivy[161]: A tool by Aqua to find vulnerabilities & IaC misconfigurations, SBOM discovery, Cloud scanning, Kubernetes security risks,and more.
- Dagda[162]: Atool to perform static analysis of known vulnerabilities, trojans, viruses, malware & other malicious threats in docker images/containers and to monitor the docker daemon and running docker containers for detecting anomalous activities
- Docker Bench for Security[163]: A script that checks for dozens of common best-practices around deploying Docker containers in production.

Set Container Filesystem to Read Only

Unless you need to modify files in your container, make its filesystem read-only.

[158] https://docs.docker.com/scout/
[159] https://github.com/quay/clair
[160] https://anchore.com/container-vulnerability-scanning/
[161] https://trivy.dev/
[162] https://github.com/eliasgranderubio/dagda
[163] https://github.com/docker/docker-bench-security

```
docker run --read-only alpine touch /tmp/killme
```

If a hacker manages to breach a container, their first objective is usually to modify the filesystem. By making the filesystem read-only, it becomes resistant to such modifications. However, this approach is not foolproof and comes with certain limitations. For instance, commands like `docker exec` for running commands within the container or `docker cp` for copying files to and from the container cannot be used. Additionally, if the application requires write access to the filesystem, it will be unable to do so.

Set Volumes to Read-Only

If you don't need to modify files in the attached volumes, make them read-only.

```
mkdir /folder
docker run -v /folder:/folder:ro --name container alpine touch /folder/killme
```

Do Not Use the Root User

By default, Docker containers run as the root user. Running a container as root means that it has the same privileges on the host system as the root user, increasing the risk of a container breakout.

If root privileges are not necessary, it is recommended to run the container as a non-root user. This helps mitigate the potential damage in the event of a container breakout.

Here is an example of a Dockerfile that runs the container as a non-root user:

```
# Use an official Python runtime as a parent image
FROM python:3.8-slim
# Set the working directory in the container
WORKDIR /app
# Copy the current directory contents into the container at /app
COPY . /app
# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt
# Create a user and group
RUN groupadd -r appuser && useradd -r -g appuser appuser
# Switch to the non-root user
USER appuser
# Make port 5000 available to the world outside this container
EXPOSE 5000
# Run app.py when the container launches
CMD ["python", "app.py"]
```

Adding `nologin` to the user in a Docker container is generally considered a good security practice. This approach restricts the user's ability to log into the container, which can help mitigate the risks if an attacker gains access to the container.

```
# Create a user and group with no login shell
RUN groupadd -r appuser && useradd -r -g appuser -s /usr/sbin/nologin appuser
```

Run the Docker Daemon in Rootless Mode

Rootless Docker allows running Docker containers without requiring root access on the Docker host. This approach enhances security, as any issues with a container are less likely to impact the entire Docker host.

To achieve this, Rootless Docker utilizes Linux user namespaces[164]. User namespaces alter user IDs in a manner that distinguishes the root user within Docker from the root user of the host system. Consequently, even if a container breaches its boundaries, it will not have root access to the host system.

```
# install uidmap
apt-get install uidmap -y
# install rootless docker
curl -fsSL https://get.docker.com/rootless | sh
# run the following command to add PATH and DOCKER_HOST to your .bashrc
cat <<EOT >> ~/.bashrc
export PATH=/home/$USER/bin:\$PATH
export DOCKER_HOST=unix:///run/user/$UID/docker.sock
EOT
# reload .bashrc
source ~/.bashrc
# start rootless Docker
systemctl --$USER start docker
# run the following command to check if rootless docker is working
docker run --rm hello-world
```

`USER` is the username of the user that you want to install Docker for. `UID` is its user ID.

Do Not Use Environment Variables For Sensitive Data

Sensitive data should not be shared using the `ENV` instruction. Doing so could expose the data to child processes, other linked containers, Docker inspection output, and other potential vulnerabilities.

[164] https://man7.org/linux/man-pages/man7/user_namespaces.7.html

Use Secret Management Tools

Kubernetes and Docker both provide their own secret management tools. If you need to store sensitive data or transfer it between services, it is recommended to use these tools. For example, Docker Secret is recommended for Swarm mode. Alternatively, you can utilize other tools such as Amazon SSM or HashiCorp Vault.

Do Not Run Containers in the Privileged Mode

When you run a container with the `--privileged` flag, it grants the container all capabilities, allowing it to access all devices on the host and configure AppArmor or SELinux settings. The container will have nearly the same level of access to the host as a regular process running directly on the host, without being confined within a container.

Here is an example of running a container in privileged mode, demonstrating how to gain access to the host's root filesystem:

```
# create a container running in the privileged mode
docker run --privileged -it -d --name evil ubuntu
# get access to the host's root filesystem
docker exec -it evil bash
# inside the container, mount the host's root filesystem
mount /dev/sda1 /mnt
# modify the host's root filesystem
echo 'malicious code' >> /mnt/etc/passwd # Modify system files
# change kernel modules
modprobe malicious_module
# disable apparmor
aa-disable /etc/apparmor.d/profile_name
# disable SELinux
setenforce 0
# ..etc
```

Turn Off Inter-Container Communication

By default, all containers in a host can communicate with each other using the docker0 bridge. If you do not require this functionality, you can disable it by setting the icc flag to false.

```
# start the docker daemon with the icc flag set to false
dockerd --icc=false
```

You can also set the icc flag to false in the Docker daemon configuration file `/etc/docker/daemon.json`:

```
cat <<EOF >> /etc/docker/daemon.json
{
  "icc": false
}
EOF
```

This will disable inter-container communication for all containers on the host. If you want to enable inter-container communication for a specific container, you can use Docker links.

```
docker run -d --name container1 ubuntu
docker run -d --name container2 --link container1:container1 ubuntu
```

Only Install Necessary Packages

Inside the container, only install the necessary packages and avoid installing unnecessary ones. This will help reduce the attack surface. To view the list of installed packages in a container, use the appropriate package manager and run the following command:

```
export CONTAINER_ID=[YOUR_CONTAINER_ID]
# Debian/Ubuntu
docker exec -it $CONTAINER_ID apt list --installed
# RedHat/CentOS
docker exec -it $CONTAINER_ID rpm -qa
# Alpine
docker exec -it $CONTAINER_ID apk info
# ..etc
```

Make Sure Docker is up to Date

Docker has a thriving community, and frequent security updates are released. From a security standpoint, it is advisable to always have the latest version of Docker in your production environments.

Properly Configure Your Docker Registry Access Control

Vine Docker images were hacked because their private registry was publicly accessible[165].

Security Through Obscurity

If `docker.vineapp.com` were `1xoajze313kjaz.vineapp.com`, it is likely that the hacker would have a harder time discovering the private registry, or it would at least take more time. While this may not be the best example to demonstrate security through obscurity, we should not underestimate the power of obscurity.

Some examples of obscurity include using uncommon names for images, containers, networks, and volumes, as well as running applications on non-standard ports.

```
1 docker run -p 7639:6793 --name x479x --network x479x image-xz12o
```

Use Limited Linux Capabilities

When limiting the Linux capabilities of a container, the host system is protected even if a hacker gains access to the container. Docker, by default, starts containers with a restricted set of capabilities.

Linux capabilities are a subset of the traditional superuser (root) privileges. They can be enabled or disabled independently for different processes. Managing capabilities is a security mechanism in the Linux kernel that helps ensure the confinement of execution for applications running on the system. This mechanism refines the application of the principle of least privilege. Capabilities are divided into distinct units, and here are some common ones:

- `CAP_CHOWN`: Allows making arbitrary changes to file UIDs and GIDs.
- `CAP_NET_BIND_SERVICE`: Enables binding a socket to Internet domain privileged ports (port numbers less than 1024).
- `CAP_DAC_OVERRIDE`: Permits bypassing file read, write, and execute permission checks.

You can view the full list by typing:

```
1 man capabilities
```

Capabilities can be assigned to executables using commands like setcap and can be queried using getcap. For example, to grant an executable the ability to bind to low-numbered ports without giving it full root privileges, you can use the following command:

[165] https://docker.vineapp.com:443/library/vinewww

```
setcap 'cap_net_bind_service=+ep' /usr/bin/python3.8
```

The `--privileged` flag grants all capabilities to the container. However, you can use the `--cap-add` and `--cap-drop` flags to add or drop specific capabilities.

Let's examine the default capabilities of a container:

```
docker run -it --rm --name test-capabilities ubuntu
# inside the container
docker exec -it test-capabilities bash
# install libcap2-bin
apt-get update && apt-get install libcap2-bin -y
# check the capabilities of the bash process
capsh --print
```

These are the default capabilities that a container has:

```
Current: cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_se\
tuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit\
_write,cap_setfcap=ep
```

Start a new container while dropping the `CAP_NET_RAW` capability:

```
docker run -it --rm --cap-drop=NET_RAW --name test-no-net-raw ubuntu
# inside the container
docker exec -it test-no-net-raw bash
# install libcap2-bin
apt-get update && apt-get install libcap2-bin -y
# check the capabilities of the bash process
capsh --print
```

As you can see, the `CAP_NET_RAW` capability is not present in the list of capabilities:

```
Current IAB: !cap_dac_read_search,!cap_linux_immutable,!cap_net_broadcast,!cap_net_a\
dmin,!cap_net_raw,!cap_ipc_lock,!cap_ipc_owner,!cap_sys_module,!cap_sys_rawio,!cap_s\
ys_ptrace,!cap_sys_pacct,!cap_sys_admin,!cap_sys_boot,!cap_sys_nice,!cap_sys_resourc\
e,!cap_sys_time,!cap_sys_tty_config,!cap_lease,!cap_audit_control,!cap_mac_override,\
!cap_mac_admin,!cap_syslog,!cap_wake_alarm,!cap_block_suspend,!cap_audit_read,!cap_p\
erfmon,!cap_bpf,!cap_checkpoint_restore
```

The `!` symbol indicates that a capability is dropped.

By dropping the `CAP_NET_RAW` capability, you prevent any process inside the container from using raw sockets. This limits the attacker's ability to perform certain network attacks, such as packet sniffing or crafting custom packets, if they compromise the container. However, ensure that you do not drop any capabilities that are required by your application.

Use Seccomp

Seccomp is a Linux kernel feature that allows you to restrict the system calls that a process can make. It is commonly used to restrict the actions available within a container. Here are some common system calls:

- `open()`: Opens a file.
- `read()`: Reads data from a file descriptor.
- `write()`: Writes data to a file descriptor.
- `_exit()`: Terminates the calling process.

By default, a container has around 44 disabled system calls[166] out of 300+. The remaining 270 calls that are still open may be susceptible to attacks. For a high level of security, you can set Seccomp profiles individually for containers. However, it is important to understand each system call and its impact on your application.

Here is an example of how to create a container with a Seccomp profile:

```
# create a Seccomp profile
cat <<EOF >> profile.json
{
  "defaultAction": "SCMP_ACT_ALLOW",
  "syscalls": [
    {
      "names": ["mkdir", "rmdir", "unlink", "unlinkat"],
      "action": "SCMP_ACT_ERRNO"
    }
  ]
}
EOF
# create a container with a Seccomp profile
docker run -d -it --security-opt seccomp=profile.json --name seccomp-container ubuntu
```

The above profile will block the `mkdir`, `rmdir`, `unlink`, and `unlinkat` calls. If any of these calls are made, the `SCMP_ACT_ERRNO` action will return an error message. You can test it using the following method:

[166] https://github.com/docker/docker/blob/master/profiles/seccomp/default.json

```
# start a bash session inside the container
docker exec -it seccomp-container bash
# run the mkdir command
mkdir test
```

You will get an error message:

```
mkdir: cannot create directory 'test': Operation not permitted
```

To obtain the comprehensive list of system calls that can be managed using seccomp in Linux, you can install and utilize auditd.

```
apt update && apt install auditd -y
ausyscall --dump
```

Use AppArmor

AppArmor is a Linux kernel security module that enables you to restrict the actions available within a container. It provides AppArmor profiles for numerous popular Linux applications, and you can also create custom profiles.

Here is a brief example of creating a container with an AppArmor profile:

```
# install apparmor-profiles
sudo apt install apparmor-profiles -y
# create a profile for the container
cat <<EOF >> /etc/apparmor.d/docker-seccomp
#include <tunables/global>
profile docker-nginx flags=(attach_disconnected,mediate_deleted) {
  #include <abstractions/base>
  network inet tcp,
  network inet udp,
  network inet icmp,
  deny network raw,
  deny network packet,
  file,
  umount,
  deny /bin/** wl,
  deny /boot/** wl,
  deny /dev/** wl,
  deny /etc/** wl,
```

```
19    deny /home/** wl,
20    deny /lib/** wl,
21    deny /lib64/** wl,
22    deny /media/** wl,
23    deny /mnt/** wl,
24    deny /opt/** wl,
25    deny /proc/** wl,
26    deny /root/** wl,
27    deny /sbin/** wl,
28    deny /srv/** wl,
29    deny /tmp/** wl,
30    deny /sys/** wl,
31    deny /usr/** wl,
32    audit /** w,
33    /var/run/nginx.pid w,
34    /usr/sbin/nginx ix,
35    deny /bin/dash mrwklx,
36    deny /bin/sh mrwklx,
37    deny /usr/bin/top mrwklx,
38    capability chown,
39    capability dac_override,
40    capability setuid,
41    capability setgid,
42    capability net_bind_service,
43    deny @{PROC}/* w,   # deny write for all files directly in /proc (not in a subdir)
44    # deny write to files not in /proc/<number>/** or /proc/sys/**
45    deny @{PROC}/{[^1-9],[^1-9][^0-9],[^1-9s][^0-9y][^0-9s],[^1-9][^0-9][^0-9][^0-9]*}\
46  /** w,
47    deny @{PROC}/sys/[^k]** w,  # deny /proc/sys except /proc/sys/k* (effectively /pro\
48  c/sys/kernel)
49    deny @{PROC}/sys/kernel/{?,??,[^s][^h][^m]**} w,  # deny everything except shm* in\
50   /proc/sys/kernel/
51    deny @{PROC}/sysrq-trigger rwklx,
52    deny @{PROC}/mem rwklx,
53    deny @{PROC}/kmem rwklx,
54    deny @{PROC}/kcore rwklx,
55    deny mount,
56    deny /sys/[^f]*/** wklx,
57    deny /sys/f[^s]*/** wklx,
58    deny /sys/fs/[^c]*/** wklx,
59    deny /sys/fs/c[^g]*/** wklx,
60    deny /sys/fs/cg[^r]*/** wklx,
61    deny /sys/firmware/** rwklx,
```

```
  deny /sys/kernel/security/** rwklx,
}
EOF
# reload apparmor
sudo apparmor_parser -r /etc/apparmor.d/docker-seccomp
# create a container with an AppArmor profile
docker run -d -it --security-opt apparmor=docker-seccomp --name apparmor-container n\
ginx
# test the profile
docker exec -it apparmor-container bash
# inside the container test the ping command
ping google.com
```

The ping command will not work because the profile blocks the `ping` command using the `deny network raw` rule.

Use SELinux

SELinux (Security Enhanced Linux) is a security module in Linux systems that supports access control security policies. It consists of kernel modifications and user-space tools that enforce various security policies on Linux systems.

The NSA, who initially developed SELinux, released the first version to the open source community under the GNU GPL on December 22, 2000.

Implementing and managing SELinux policies can be complex and requires a solid understanding of SELinux concepts and policy writing skills. However, there are numerous SELinux policies available for commonly used Linux applications.

Docker API

Docker SDKs

The Docker API is served by Docker Engine and provides full control over Docker. This is particularly useful when building applications that utilize Docker. To determine the version of the Engine API you are running, you can use the command `docker version | grep -i api`. The API often changes with each Docker release, so API calls are versioned to ensure compatibility. To interact with this API, you should use one of the available SDKs based on the programming language you are using.

Here is a list of some known SDKs to use Docker Engine API:

Language	Library	Official
Go	Moby[167]	Yes
Python	docker-py[168]	Yes
C	libdocker[169]	No
C#	Docker.DotNet[170]	No
C++	lasote/docker_client[171]	No
Clojure	clj-docker-client[172]	No
Clojure	contajners[173]	No
Dart	bwu_docker[174]	No
Erlang	erldocker[175]	No
Gradle	gradle-docker-plugin[176]	No
Groovy	docker-client[177]	No
Haskell	docker-hs[178]	No
HTML (Web Components)	docker-elements[179]	No
Java	docker-client[180]	No
Java	docker-java[181]	No
Java	docker-java-api[182]	No
Java	jocker[183]	No

[167] https://github.com/moby/moby/tree/master/client
[168] https://github.com/docker/docker-py
[169] https://github.com/danielsuo/libdocker
[170] https://github.com/ahmetalpbalkan/Docker.DotNet
[171] https://github.com/lasote/docker_client
[172] https://github.com/into-docker/clj-docker-client
[173] https://github.com/lispyclouds/contajners
[174] https://github.com/bwu-dart/bwu_docker
[175] https://github.com/proger/erldocker
[176] https://github.com/gesellix/gradle-docker-plugin
[177] https://github.com/gesellix/docker-client
[178] https://github.com/denibertovic/docker-hs
[179] https://github.com/kapalhq/docker-elements
[180] https://github.com/spotify/docker-client
[181] https://github.com/docker-java/docker-java
[182] https://github.com/amihaiemil/docker-java-api
[183] https://github.com/ndeloof/jocker

Language	Library	Official
NodeJS	dockerode[184]	No
NodeJS	harbor-master[185]	No
Perl	Eixo::Docker[186]	No
PHP	Docker-PHP[187]	No
Ruby	docker-api[188]	No
Rust	docker-rust[189]	No
Rust	shiplift[190]	No
Scala	tugboat[191]	No
Scala	reactive-docker[192]	No
Swift	docker-client-swift[193]	No

Docker API: Hello World

In the following section, we are going to see how to use the Docker API to create a container and run it. We are going to use the following methods:

- HTTP REST API
- Go SDK
- Python SDK

REST API

Using the REST API, we can create a container using the following command:

```
curl \
--unix-socket /var/run/docker.sock \
-H "Content-Type: application/json" \
-d '{"Image": "alpine", "Cmd": ["tail", "-f", "/dev/null"]}' \
-X POST http://localhost/v1.43/containers/create?name=test-container
```

This curl command creates a container named “test-container” using the Alpine image and runs the command `tail -f /dev/null` in the container. The command `tail -f /dev/null` is a way to keep the container running without doing anything. The container will be created in the stopped state. To start the container, we can use the following command:

[184] https://github.com/apocas/dockerode
[185] https://github.com/arhea/harbor-master
[186] https://github.com/alambike/eixo-docker
[187] https://github.com/docker-php/docker-php
[188] https://github.com/swipely/docker-api
[189] https://github.com/abh1nav/docker-rust
[190] https://github.com/softprops/shiplift
[191] https://github.com/softprops/tugboat
[192] https://github.com/almoehi/reactive-docker
[193] https://github.com/valeriomazzeo/docker-client-swift

```
curl \
--unix-socket /var/run/docker.sock \
-X POST http://localhost/v1.43/containers/test-container/start
```

To verify that the container is running, we can use the following command:

```
curl \
--unix-socket /var/run/docker.sock \
-X GET http://localhost/v1.43/containers/test-container/json
```

You should see that the container is in the "Running" state:

```
"status": "running",
```

Go SDK

Let's create the same container using Go SDK. Start by removing the container we created in the previous section:

```
docker rm -f test-container
```

First, we need to run the following commands:

```
mkdir -p container-test-go && cd container-test-go
# Initialize a Go module
go mod init container-test-go
# Install the Docker SDK for Go
go get github.com/docker/docker/api/types
go get github.com/docker/docker/api/types/container
go get github.com/docker/docker/client
```

Add the following code to "run.go":

```
cat <<EOF > run.go
package main

import (
    "context"
    "io"
    "os"
    "github.com/docker/docker/api/types"
    "github.com/docker/docker/api/types/container"
    "github.com/docker/docker/client"
)

func main() {
    // Create a background context for Docker operations
    ctx := context.Background()

    // Initialize Docker client
    cli, err := client.NewClientWithOpts(client.FromEnv, client.WithAPIVersionNegoti\
ation())
    if err != nil {
        panic(err)
    }
    defer cli.Close()

    // Pull the Alpine image from Docker Hub
    reader, err := cli.ImagePull(ctx, "docker.io/library/alpine", types.ImagePullOpt\
ions{})
    if err != nil {
        panic(err)
    }
    io.Copy(os.Stdout, reader)

    // Create a container from the Alpine image, executing the command "tail -f /dev\
/null"
    resp, err := cli.ContainerCreate(ctx, &container.Config{
        Image: "alpine",
        Cmd:   []string{"tail", "-f", "/dev/null"},
    }, nil, nil, nil, "test-container")  // Set container name to "test-container"
    if err != nil {
        panic(err)
    }

    // Start the created container
```

```
    if err := cli.ContainerStart(ctx, resp.ID, types.ContainerStartOptions{}); err !\
= nil {
        panic(err)
    }
}
EOF
```

Run the following command to build and run the program:

```
go run run.go
```

This will create a container named “test-container” and start it. To verify that the container is running, we can run this Go program:

```
cat <<EOF > list.go
package main

import (
    "context"
    "fmt"
    "time"

    "github.com/docker/docker/api/types"
    "github.com/docker/docker/client"
)

func main() {
    // Create a background context for Docker operations
    ctx := context.Background()

    // Initialize Docker client
    cli, err := client.NewClientWithOpts(client.FromEnv, client.WithAPIVersionNegoti\
ation())
    if err != nil {
        panic(err)
    }
    defer cli.Close() // Ensure the client is closed when the function returns

    // Retrieve a list of containers
    containers, err := cli.ContainerList(ctx, types.ContainerListOptions{})
    if err != nil {
        panic(err)
```

```
    }

    // Iterate through and print details of each container
    for _, container := range containers {
        fmt.Printf("ID: %s\n", container.ID)
        fmt.Printf("Names: %v\n", container.Names)
        fmt.Printf("Image: %s\n", container.Image)
        fmt.Printf("Command: %s\n", container.Command)
        fmt.Printf("Created: %s\n", time.Unix(container.Created, 0).Format(time.RFC1\
123))
        fmt.Printf("Status: %s\n", container.Status)
        fmt.Printf("Ports: %v\n\n", container.Ports)
    }
}
EOF
```

Run the following command to build and run the program:

```
go run list.go
```

Python SDK

Let's do the same thing using Python SDK. Start by removing the container we created in the previous section:

```
docker rm -f test-container
```

First, we need to run the following commands:

```
mkdir -p container-test-python && cd container-test-python
# Initialize a Python virtual environment
apt install python3-pip
pip3 install virtualenvwrapper
mkvirtualenv container-test-python
# Install the Docker SDK for Python
pip install docker
```

Add the following code to "run.py":

```
cat <<EOF > run.py
import docker

# Create a Docker client
client = docker.from_env()

# Pull the Alpine image (if not already present)
client.images.pull("alpine")

# Create a container with the specified name, image, and command
container = client.containers.create(
    image="alpine",
    command="tail -f /dev/null",
    name="test-container"
)

# Start the container
container.start()
EOF
```

In order to run the program, we need to activate the Python virtual environment:

```
workon container-test-python
python run.py
```

This will create a container named "test-container" and start it. To verify that the container is running, we can run this Python program:

```
cat <<EOF > list.py
import docker

# Create a Docker client
client = docker.from_env()

# List all containers (including non-running ones)
containers = client.containers.list(all=True)

# Iterate through and print details of each container
for container in containers:
    print(f"ID: {container.id}")
    print(f"Name: {container.name}")
    print(f"Image: {container.image.tags}")
```

```
    print(f"Status: {container.status}")
    print("-----")
EOF
```

Run the following command to build and run the program:

```
python list.py
```

Prototyping a Log Collector Service

Let's create a service that collects logs from all running containers. We are going to use Python SDK. This is the initial program:

```
import docker
import threading

def stream_container_logs(container):
    """Stream logs from a single container."""
    for log in container.logs(stream=True):
        print(f"Logs from {container.name}: {log.decode().strip()}")

def main():
    # Create a Docker client
    client = docker.from_env()

    # List all running containers
    containers = client.containers.list()

    # Create and start a thread for each container to stream logs
    threads = []
    for container in containers:
        thread = threading.Thread(target=stream_container_logs, args=(container,))
        thread.start()
        threads.append(thread)

    # Wait for all threads to complete (optional)
    for thread in threads:
        thread.join()

if __name__ == "__main__":
    main()
```

The idea is to create a thread for each container to stream logs. However, this program does not store logs yet. For the sake of simplicity, we are going to use SQLite to store logs. First, we need to run the following commands that creates the database and table:

```
cat <<EOF > init.py
import sqlite3

def create_db():
    conn = sqlite3.connect('container_logs.db')
    cursor = conn.cursor()
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS logs (
            timestamp DATETIME,
            host TEXT,
            container_id TEXT,
            log_line TEXT
        )
    ''')
    conn.commit()
    conn.close()

if __name__ == "__main__":
    create_db()
EOF
```

Run the script:

```
python init.py
```

Now, let's go back to the initial code and make it save logs to the database we just created:

```
cat <<EOF > main.py
import docker
import sqlite3
import datetime
import threading
import socket

def save_log_to_db(container_id, log_line):
    """Save log line to SQLite database."""
    conn = sqlite3.connect('container_logs.db')
    cursor = conn.cursor()
```

```
    timestamp = datetime.datetime.now()
    host = socket.gethostname()
    cursor.execute("INSERT INTO logs (timestamp, host, container_id, log_line) VALUE\
S (?, ?, ?, ?)",
                   (timestamp, host, container_id, log_line))
    conn.commit()
    conn.close()

def stream_container_logs(container):
    """Stream logs from a single container and save them to the database."""
    for log in container.logs(stream=True):
        save_log_to_db(container.id, log.decode().strip())

def main():
    # Create a Docker client
    client = docker.from_env()

    # List all running containers
    containers = client.containers.list()

    # Create and start a thread for each container to stream logs
    threads = []
    for container in containers:
        thread = threading.Thread(target=stream_container_logs, args=(container,))
        thread.start()
        threads.append(thread)

    # Wait for all threads to complete (optional)
    for thread in threads:
        thread.join()

if __name__ == "__main__":
    main()
EOF
```

Debugging And Troubleshooting

Docker Daemon Logs

When encountering a problem, one of the first steps for many is to check the Docker Daemon logs. The accessibility of Docker logs varies depending on your system:

Operating System	Log File Location or Command
OSX	`~/Library/Containers/*/log/docker.log`
Debian	`/var/log/daemon.log`
CentOS	Run `/var/log/daemon.log | grep docker`
CoreOS	Run `journalctl -u docker.service`
Ubuntu (upstart)	`/var/log/upstart/docker.log`
Ubuntu (systemd)	Run `journalctl -u docker.service`
Fedora	Run `journalctl -u docker.service`
Red Hat Enterprise Linux Server	Run `/var/log/messages | grep docker`
OpenSuSE	Run `journalctl -u docker.service`
Boot2Docker	`/var/log/docker.log`
Windows	`AppData\Local`

Another way to troubleshoot the daemon is by running it in the foreground.

```
dockerd
```

If you are already running Docker, you should stop it and start the daemon.

```
# stop
sudo service docker stop
# start
sudo dockerd
```

Activating Debug Mode

To activate the debug mode, you need to set the log level to debug. To do this, set the debug key to true in the “daemon.json” file. Normally, you can find this file under “/etc/docker”. If the file does not exist, you may need to create it.

```
{
  "debug": true
}
```

Possible values are `debug`, `info`, `warn`, `error`, `fatal`.

Now send a HUP signal to the daemon to make it reload its configuration: `sudo kill -SIGHUP $(pidof dockerd)` or execute `service docker stop && dockerd`:

You will be able to see all of the actions that Docker is doing. For example, when you run `docker run hello-world`, you will see something like:

```
DEBU[2023-11-24T08:13:40.601762558Z] form data: {"AttachStderr":true,"AttachStdin":f\
alse,"AttachStdout":true,"Cmd":null,"Domainname":"","Entrypoint":null,"Env":null,"Ho\
stConfig":{"AutoRemove":false,"Binds":null,"BlkioDeviceReadBps":[],"BlkioDeviceReadI\
Ops":[],"BlkioDeviceWriteBps":[],"BlkioDeviceWriteIOps":[],"BlkioWeight":0,"BlkioWei\
ghtDevice":[],"CapAdd":null,"CapDrop":null,"Cgroup":"","CgroupParent":"","CgroupnsMo\
de":"","ConsoleSize":[49,85],"ContainerIDFile":"","CpuCount":0,"CpuPercent":0,"CpuPe\
riod":0,"CpuQuota":0,"CpuRealtimePeriod":0,"CpuRealtimeRuntime":0,"CpuShares":0,"Cpu\
setCpus":"","CpusetMems":"","DeviceCgroupRules":null,"DeviceRequests":null,"Devices"\
:[],"Dns":[],"DnsOptions":[],"DnsSearch":[],"ExtraHosts":null,"GroupAdd":null,"IOMax\
imumBandwidth":0,"IOMaximumIOps":0,"IpcMode":"","Isolation":"","Links":null,"LogConf\
ig":{"Config":{},"Type":""},"MaskedPaths":null,"Memory":0,"MemoryReservation":0,"Mem\
orySwap":0,"MemorySwappiness":-1,"NanoCpus":0,"NetworkMode":"default","OomKillDisabl\
e":false,"OomScoreAdj":0,"PidMode":"","PidsLimit":0,"PortBindings":{},"Privileged":f\
alse,"PublishAllPorts":false,"ReadonlyPaths":null,"ReadonlyRootfs":false,"RestartPol\
icy":{"MaximumRetryCount":0,"Name":"no"},"SecurityOpt":null,"ShmSize":0,"UTSMode":""\
,"Ulimits":null,"UsernsMode":"","VolumeDriver":"","VolumesFrom":null},"Hostname":"",\
"Image":"hello-world","Labels":{},"NetworkingConfig":{"EndpointsConfig":{}},"OnBuild\
":null,"OpenStdin":false,"StdinOnce":false,"Tty":false,"User":"","Volumes":{},"Worki\
ngDir":""}
DEBU[2023-11-24T08:13:40.642530675Z] container mounted via layerStore: /var/lib/dock\
er/overlay2/29100a7dc2eb24f86aa2f80fa8ff55b5d92cc069221d1013297ef310b9e91b35/merged \
 container=0f40cbf4bbc319b530b9f1f7b4224e10a7a5166a38b7d9331bb8bf6042d462c7
DEBU[2023-11-24T08:13:40.661266118Z] Calling POST /v1.43/containers/0f40cbf4bbc319b5\
30b9f1f7b4224e10a7a5166a38b7d9331bb8bf6042d462c7/attach?stderr=1&stdout=1&stream=1
DEBU[2023-11-24T08:13:40.661717018Z] attach: stdout: begin
DEBU[2023-11-24T08:13:40.661773761Z] attach: stderr: begin
DEBU[2023-11-24T08:13:40.662228060Z] Calling POST /v1.43/containers/0f40cbf4bbc319b5\
30b9f1f7b4224e10a7a5166a38b7d9331bb8bf6042d462c7/wait?condition=next-exit
DEBU[2023-11-24T08:13:40.662965861Z] Calling POST /v1.43/containers/0f40cbf4bbc319b5\
30b9f1f7b4224e10a7a5166a38b7d9331bb8bf6042d462c7/start
```

Debugging Docker Objects

Whether you are using standalone containers or managed services with Docker Swarm, Docker provides the inspect command to retrieve information about the objects you are working with. The inspect command offers detailed information about Docker objects, including containers, images, volumes, networks, and services.

```
docker service inspect [OPTIONS] SERVICE [SERVICE...]
docker container inspect [OPTIONS] CONTAINER [CONTAINER...]
docker image inspect [OPTIONS] IMAGE [IMAGE...]
docker network inspect [OPTIONS] NETWORK [NETWORK...]
docker volume inspect [OPTIONS] VOLUME [VOLUME...]
```

As an example, let's create the following container:

```
docker run -it -p 8000:80 -d --name webserver nginx
```

Now, use the following command:

```
docker inspect webserver
```

The output will be a JSON object that contains all the information about the container. It is also possible to retrieve a single element, such as the IP address or port bindings.

```
# Get the IP address
docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' w\
ebserver
# Get the port bindings
docker inspect --format='{{range $p, $conf := .NetworkSettings.Ports}} {{$p}} -> {{(\
index $conf 0).HostPort}} {{end}}' webserver
```

Since you can access the container's host, you can also use regular commands to debug it:

```
docker exec -it webserver ps aux
docker exec -it webserver cat /etc/resolv.conf
```

Using `docker stats` and `docker events` commands could give you also some additional information useful for debugging.

Troubleshooting Docker Using Sysdig

There are several tools available for troubleshooting Docker. One of the most popular tools is Sysdig[194]. Sysdig is a Linux system exploration and troubleshooting tool that provides support for containers.

To install Sysdig, simply run the following command. This is the recommended installation method.

```
curl -s https://download.sysdig.com/stable/install-sysdig | sudo bash
```

You can also install sysdig using your Linux distribution package manager:

```
apt install sysdig
```

Or by running it in a container:

```
sudo docker run --rm -i -t --privileged --net=host \
    -v /var/run/docker.sock:/host/var/run/docker.sock \
    -v /dev:/host/dev \
    -v /proc:/host/proc:ro \
    -v /boot:/host/boot:ro \
    -v /src:/src \
    -v /lib/modules:/host/lib/modules:ro \
    -v /usr:/host/usr:ro \
    -v /etc:/host/etc:ro \
    docker.io/sysdig/sysdig
```

We are going to install using the script method. The script will install the Sysdig repository and the Sysdig package.

Then add your username to the same group as Sysdig:

```
groupadd sysdig
usermod -aG sysdig $USER
```

Use `visudo` to edit the "sudo-config" file and add the line `%sysdig ALL=/usr/bin/sysdig`. Save the changes.

These configurations will allow you to run Sysdig without using sudo.

Sysdig can be used to obtain information about:

[194] https://github.com/draios/sysdig

- Networking
- Containers
- Applications
- Disk I/O
- Processes and CPU usage
- Performance and Errors
- Security
- Tracing

Debugging containers is also debugging the host, so Sysdig can be used for general troubleshooting. However, in this section, we are specifically interested in the container-related commands. Let's take a look at some of them.

To list the running containers along with their resource usage:

```
csysdig -vcontainers
```

Listing all of the processes with container context can be done using:

```
csysdig -pc
```

To view the CPU usage of the processes running inside the "my_container" container, use:

```
sysdig -pc -c topprocs_cpu container.name=my_container
```

Bandwidth can be monitored using:

```
sysdig -pc -c topprocs_net container.name=my_container
```

Processes using most network bandwidth can be checked using:

```
sysdig -pc -c topprocs_net container.name=my_container
```

To view the top network connections:

```
sysdig -pc -c topconns container.name=my_container
```

Top used files consuming I/O bytes could be checked using:

```
sysdig -pc -c topfiles_bytes container.name=my_container
```

And to show all the interactive commands executed inside the my_container container, use:

```
sysdig -pc -c spy_users container.name=my_container
```

The Ultimate Docker Cheat Sheet

Installation

Linux

Here's a general command to install Docker on most Linux distributions:

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh
```

Notes:

- This script will attempt to install Docker from Docker's repositories. It's a convenient one-liner, but remember that running scripts downloaded from the internet can be risky. Always review scripts before executing them.
- The installation method may vary depending on your specific Linux distribution. It's recommended to follow the installation guide for your distribution from the Docker documentation.
- After installation, consider adding your user to the docker group to manage Docker as a non-root user:

```
sudo usermod -aG docker $USER
# You will need to log out and log back in for this to take effect.
```

- For production environments, it's advised to follow a more controlled installation process, like using your distribution's package manager to install Docker from official repositories.

For more information, see here[195]

[195] https://docs.docker.com/install/#server

Mac

Use one of the following links to download the Docker Desktop .dmg file for the stable version:

- Docker Desktop for Mac with Apple silicon: Download[196]
- Docker Desktop for Mac with Intel chip: Download[197]

Open the downloaded .dmg file. Follow the on-screen installation instructions, which typically involve dragging the Docker icon to the Applications folder.

Once installed, you can open Docker from your Applications folder. Verify the installation by running `docker --version` in the terminal.

Note: Docker for Mac requires Apple Mac with Intel's hardware support for Memory management unit (MMU) virtualization, including Extended Page Tables (EPT) and Unrestricted Mode. You can check your hardware specifications for compatibility.

For more information, see here[198]

Windows

Download Docker Desktop for Windows from here[199]

Open the downloaded .msi file. Follow the on-screen instructions to complete the installation process.

Verify Installation:

- After installation, you can open Docker from the Start menu.
- To verify the installation, open a command prompt or PowerShell and run `docker --version`.

Note: Docker Desktop for Windows requires Microsoft Windows 10 Pro or Enterprise 64-bit, or Windows 11 for Intel processors. For older versions of Windows, Docker Toolbox may be used. Docker Desktop for Windows comes with Docker Engine, Docker CLI client, Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper.

Ensure your system meets the necessary requirements and that virtualization is enabled in the BIOS settings. Docker Desktop for Windows uses Hyper-V for virtualization and also supports WSL 2 backend.

For more information, see here[200]

[196] https://desktop.docker.com/mac/main/arm64/Docker.dmg
[197] https://desktop.docker.com/mac/main/amd64/Docker.dmg
[198] https://docs.docker.com/docker-for-mac/install/
[199] https://desktop.docker.com/win/main/amd64/Docker%20Desktop%20Installer.exe
[200] https://docs.docker.com/docker-for-windows/install/

Docker Registries & Repositories

Login to a Registry

- Docker Hub

```
docker login
# or docker login docker.io
```

- Private Registry

```
docker login localhost:8080
```

Logout from a Registry

- Docker Hub

```
docker logout
# or docker logout docker.io
```

- Private Registry

```
docker logout localhost:8080
```

Searching an Image

```
docker search nginx
```

```
docker search --filter stars=3 --no-trunc nginx
```

Pulling an Image

```
docker image pull nginx
```

```
docker image pull eon01/nginx localhost:5000/myadmin/nginx
```

Pushing an Image

```
docker image push eon01/nginx
```

```
docker image push eon01/nginx localhost:5000/myadmin/nginx
```

Running Containers

Create and Run a Simple Container

- Start an ubuntu:latest[201] image
- Bind the port `80` from the **CONTAINER** to port `3000` on the **HOST**
- Mount the current directory to `/data` on the CONTAINER
- Note: on **windows** you have to change `-v ${PWD}:/data` to `-v "C:\Data":/data`

```
docker container run --name infinite -it -p 3000:80 -v ${PWD}:/data ubuntu:latest
```

Creating a Container

```
docker container create -t -i eon01/infinite --name infinite
```

Running a Container

```
docker container run -it --name infinite -d eon01/infinite
```

Renaming a Container

[201] https://hub.docker.com/_/ubuntu/

```
docker container rename infinite infinity
```

Removing a Container

```
docker container rm infinite
```

A container can be removed only after stopping it using `docker stop` command. To avoid this, add the `--rm` flag while running the container.

Updating a Container

```
docker container update --cpu-shares 512 -m 300M infinite
```

Running a command within a running container

```
docker exec -it infinite sh
```

Starting & Stopping Containers

Starting

```
docker container start nginx
```

Stopping

```
docker container stop nginx
```

Restarting

```
docker container restart nginx
```

Pausing

```
docker container pause nginx
```

Unpausing

```
docker container unpause nginx
```

Blocking a Container

```
docker container wait nginx
```

Sending a SIGKILL

```
docker container kill nginx
```

Sending another signal

```
docker container kill -s HUP nginx
```

Attaching to a Container

```
docker container attach nginx
```

Getting Information about Containers

From Running Containers

Short version:

```
docker ps
```

Alternative:

```
docker container ls
```

From All containers

```
docker ps -a
```

```
docker container ls -a
```

Container Logs

```
docker logs infinite
```

Tail Containers' Logs

```
docker container logs infinite -f
```

Inspecting Containers

```
docker container inspect infinite
```

```
docker container inspect --format '{{ .NetworkSettings.IPAddress }}' $(docker ps -q)
```

Containers Events

```
docker system events infinite
```

Public Ports

```
docker container port infinite
```

Running Processes

```
docker container top infinite
```

Container Resource Usage

```
docker container stats infinite
```

Inspecting changes to files or directories on a container's filesystem

```
docker container diff infinite
```

Managing Images

Listing Images

```
docker image ls
```

Building Images

From a Dockerfile in the Current Directory

```
docker build .
```

From a Remote GIT Repository

```
docker build github.com/creack/docker-firefox
```

Instead of Specifying a Context, You Can Pass a Single Dockerfile in the URL or Pipe the File in via STDIN

```
docker build - < Dockerfile
```

```
docker build - < context.tar.gz
```

Building and Tagging

```
docker build -t eon/infinite .
```

Building a Dockerfile while Specifying the Build Context

```
docker build -f myOtherDockerfile .
```

Building from a Remote Dockerfile URI

```
curl example.com/remote/Dockerfile | docker build -f - .
```

Removing an Image

```
docker image rm nginx
```

Loading a Tarred Repository from a File or the Standard Input Stream

```
docker image load < ubuntu.tar.gz
```

```
docker image load --input ubuntu.tar
```

Saving an Image to a Tar Archive

```
docker image save busybox > ubuntu.tar
```

Showing the History of an Image

```
docker image history
```

Creating an Image From a Container

```
docker container commit nginx
```

Tagging an Image

```
docker image tag nginx eon01/nginx
```

Networking

Creating Networks

Creating an Overlay Network

```
docker network create -d overlay MyOverlayNetwork
```

Creating a Bridge Network

```
docker network create -d bridge MyBridgeNetwork
```

Creating a Customized Overlay Network

```
docker network create -d overlay \
  --subnet=192.168.0.0/16 \
  --subnet=192.170.0.0/16 \
  --gateway=192.168.0.100 \
  --gateway=192.170.0.100 \
  --ip-range=192.168.1.0/24 \
  --aux-address="my-router=192.168.1.5" --aux-address="my-switch=192.168.1.6" \
  --aux-address="my-printer=192.170.1.5" --aux-address="my-nas=192.170.1.6" \
  MyOverlayNetwork
```

Removing a Network

```
docker network rm MyOverlayNetwork
```

Listing Networks

```
docker network ls
```

Getting Information About a Network

```
docker network inspect MyOverlayNetwork
```

Connecting a Running Container to a Network

```
docker network connect MyOverlayNetwork nginx
```

Connecting a Container to a Network When it Starts

```
docker container run -it -d --network=MyOverlayNetwork nginx
```

Disconnecting a Container from a Network

```
docker network disconnect MyOverlayNetwork nginx
```

Exposing Ports

Using Dockerfile, you can expose a port on the container using:

```
EXPOSE <port_number>
```

You can also map the container port to a host port using:

```
docker run -p $HOST_PORT:$CONTAINER_PORT --name <container_name> -t <image>
```

e.g.

```
docker run -p $HOST_PORT:$CONTAINER_PORT --name infinite -t infinite
```

Cleaning Docker

Removing a Running Container

```
docker container rm nginx
```

Removing a Container and its Volume

```
docker container rm -v nginx
```

Removing all Exited Containers

```
docker container rm $(docker container ls -a -f status=exited -q)
```

Removing all Stopped Containers

```
docker container rm `docker container ls -a -q`
```

Removing a Docker Image

```
docker image rm nginx
```

Removing Dangling Images

```
docker image rm $(docker image ls -f dangling=true -q)
```

Removing all Images

```
docker image rm $(docker image ls -a -q)
```

Removing all Untagged Images

```
docker image rm -f $(docker image ls | grep "^<none>" | awk "{print $3}")
```

Stopping and Removing all Containers

```
docker container stop $(docker container ls -a -q) && docker container rm $(docker c\
ontainer ls -a -q)
```

Removing Dangling Volumes

```
docker volume rm $(docker volume ls -f dangling=true -q)
```

Removing all Unused Resources: Containers, Networks, Images, and Volumes

```
docker system prune -a
```

Forcefully Removing all Unused Resources: Containers, Networks, Images, and Volumes

```
docker system prune -a --force
```

Docker Swarm

Installing Docker Swarm

```
curl -ssl https://get.docker.com | bash
```

Initializing the Swarm

```
docker swarm init --advertise-addr 192.168.10.1
```

Getting a Worker to Join the Swarm

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

Getting a Manager to Join the Swarm

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

Listing Services

```
docker service ls
```

Listing nodes

```
docker node ls
```

Creating a Service

```
docker service create --name vote -p 8080:80 instavote/vote
```

Listing Swarm Tasks

```
docker service ps
```

Scaling a Service

```
docker service scale vote=3
```

Updating a Service

```
docker service update --image instavote/vote:movies vote
```

```
docker service update --force --update-parallelism 1 --update-delay 30s nginx
```

```
docker service update --update-parallelism 5--update-delay 2s --image instavote/vote\
:indent vote
```

```
docker service update --limit-cpu 2 nginx
```

```
docker service update --replicas=5 nginx
```

Docker Scout Suite

Compare the Most Recently Built Image to a Reference

```
docker scout compare --to namespace/repo:latest
```

Compare an Image to the Latest Tag

```
docker scout compare --to namespace/repo:latest namespace/repo:v1.2.3-pre
```

Compare a Local Build to the Same Tag from the Registry

```
docker scout compare local://namespace/repo:v1.2.3 --to registry://namespace/repo:v1\
.2.3
```

Ignore Base Images

```
docker scout compare --ignore-base --to namespace/repo:latest namespace/repo:v1.2.3-\
pre
```

Generate a Markdown Output

```
docker scout compare --format markdown --to namespace/repo:latest namespace/repo:v1.\
2.3-pre
```

Only Compare Maven Packages and Only Display Critical Vulnerabilities for Maven Packages

```
docker scout compare --only-package-type maven --only-severity critical --to namespa\
ce/repo:latest namespace/repo:v1.2.3-pre
```

List Existing Environments

```
docker scout environment
```

List Images of an Environment

```
docker scout environment staging
```

Record an Image to an Environment, for a Specific Platform

```
docker scout environment staging namespace/repo:stage-latest --platform linux/amd64
```

Display Vulnerabilities For The Most Recently Built Image

```
docker scout cves
```

Display Vulnerabilities Grouped By Package

```
docker scout cves alpine
```

Display Vulnerabilities From A Docker Save Tarball

```
docker save alpine > alpine.tar
docker scout cves archive://alpine.tar
```

Display Vulnerabilities From An Oci Directory

```
skopeo copy --override-os linux docker://alpine oci:alpine
docker scout cves oci-dir://alpine
```

Display Vulnerabilities From The Current Directory

```
docker scout cves fs://.
```

Export Vulnerabilities To A Sarif Json File

```
docker scout cves --format sarif --output alpine.sarif.json alpine
```

Markdown Output, Including Html Tags. To Be Used In Pull Request Comments For Instance

```
docker scout cves --format markdown alpine
```

List All Go Packages Of The Image That Are Vulnerable

```
docker scout cves --format only-packages --only-package-type golang --only-vuln-pack\
ages golang:1.18.0
```

Display Base Image Update Recommendations Of The Most Recently Buit Image

```
docker scout recommendations
```

Only Display Base Image Refresh Recommendations

```
docker scout recommendations golang:1.19.4 --only-refresh
```

Only Display Base Image Update Recommendations

```
docker scout recommendations golang:1.19.4 --only-update
```

Evaluate Policies Against An Image

```
docker scout policy IMAGE
```

Evaluate Policies Against An Image For A Specific Organization

```
docker scout policy IMAGE --org ORG
```

Evaluate Policies Against An Image With A Specific Platform

```
docker scout policy IMAGE --platform PLATFORM
```

Compare Policy Results For A Repository In A Specific Environment

```
docker scout policy REPO --env ENV
```

Resources

Guidelines for Building Secure Images and Containers

- Use Minimal Base Images: Choose smaller and simpler base images to reduce attack surface.
- Non-root User: Run as a non-root user to minimize privileges.
- Sign and Verify Images: Implement image signing to prevent man-in-the-middle (MITM) attacks.
- Manage Vulnerabilities: Regularly scan for and address vulnerabilities in open-source components.
- Protect Sensitive Data: Avoid embedding secrets or sensitive data in image layers.
- Immutable Tags: Use specific version tags for reproducibility and stability.
- Prefer COPY Over ADD: Use COPY for file transfer to avoid unexpected behavior of ADD.
- Utilize Labels: Apply metadata labels for better image organization and management.
- Multi-Stage Builds: Leverage multi-stage builds for smaller, more secure final images.
- Use a Linter: Implement a Dockerfile linter for best practices and common mistakes.
- Regular Updates: Keep base images and dependencies up-to-date to patch vulnerabilities.
- Limit Build Context: Minimize the build context sent to the Docker daemon.
- Avoid Privilege Escalation: Set USER before CMD or ENTRYPOINT to enforce user context.
- Explicit Port Exposing: Only expose necessary ports to limit network access.
- Implement Health Checks: Include health checks for container self-assessment and recovery.
- Use Specific FROM Images: Avoid latest tag; use specific versions for base images.
- Control Resource Use: Implement resource limits (CPU, memory) to mitigate DoS attacks.
- Enable Docker Security Features: Utilize features like seccomp, AppArmor, and user namespaces.
- Review and Reduce Layers: Minimize layers and consolidate commands to reduce complexity.
- Use Trusted Registries: Pull base images and dependencies from trusted, official sources.

Newsletters, News and Blogs

- Docker Blog[202]
- Docker Security Blog[203]
- Kaptain Newsletter: Docker, Kubernetes and Distributed Systems[204]
- DevOpsLinks Newsltter: Cloud and DevOps[205]

[202] https://www.docker.com/blog/
[203] https://www.docker.com/blog/tag/docker-security/
[204] https://faun.dev/newsletter/kaptain
[205] https://faun.dev/newsletter/devopslinks

How to Start

- Docker Documentation[206]
- Dockerfile Reference[207]
- Play With Docker[208]

Security Resources

- 10 Docker Image Security Best Practices[209]
- Security best practices[210]
- Docker security[211]
- Docker Security: 5 Risks and 5 Best Practices for Securing Your Containers[212]
- Docker Security Cheat Sheet[213]
- Docker Security Best Practices: A Complete Guide[214]
- Top 20 Dockerfile best practices[215]
- Top 20 Docker Security Best Practices: Ultimate Guide[216]
- Docker Security Best Practices - cheat sheet included[217]
- Security Best Practices for Docker Containers[218]
- Docker Security: 14 Best Practices for Securing Docker Containers[219]

Dockerfile Best Practices

- Overview of best practices for writing Dockerfiles[220]
- Guidelines for Dockerfile Best Practices[221]
- Dockerfile Best Practices[222]
- Best Practices and Tips for Writing a Dockerfile[223]
- Best Practices for Docker[224]
- Best Practices in Writing Dockerfiles[225]

[206] https://docs.docker.com/
[207] https://docs.docker.com/engine/reference/builder/
[208] https://labs.play-with-docker.com/
[209] https://snyk.io/blog/10-docker-image-security-best-practices/
[210] https://docs.docker.com/develop/security-best-practices/
[211] https://docs.docker.com/engine/security/
[212] https://www.tigera.io/learn/guides/container-security-best-practices/docker-security/
[213] https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html
[214] https://anchore.com/blog/docker-security-best-practices-a-complete-guide/
[215] https://sysdig.com/blog/dockerfile-best-practices/
[216] https://blog.aquasec.com/docker-security-best-practices
[217] https://blog.gitguardian.com/how-to-improve-your-docker-containers-security-cheat-sheet/
[218] https://kinsta.com/blog/docker-security/
[219] https://www.bmc.com/blogs/docker-security-best-practices/
[220] https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
[221] https://docs.docker.com/develop/develop-images/guidelines/
[222] https://sysdig.com/blog/dockerfile-best-practices/
[223] https://www.qovery.com/blog/best-practices-and-tips-for-writing-a-dockerfile
[224] https://www.harness.io/blog/best-practices-for-docker
[225] https://www.divio.com/blog/best-practices-writing-dockerfiles/

Afterword

What's next?

Congratulations on completing "Painless Docker"! I trust that this deep dive into the world of Docker has enriched your understanding and equipped you with useful and especially actionable insights.

Throughout this guide, we have explained how to use Docker to build, ship, and run applications. We have also explored the Docker ecosystem and its various components, including Docker Compose and Docker Swarm.

By now, you should possess a robust grasp of the methodologies and best practices pivotal to harnessing the full capabilities of Docker.

As with any skill, mastery comes with practice. I urge you to apply the strategies and examples discussed in this guide, experimenting and iterating to refine your proficiency.

I hope this reading experience was both enlightening and enjoyable. Continue your journey of discovery and **never cease to be curious**!

Thank you

Thank you for accompanying me on this enlightening journey. I wish you success and innovation in all your future adventures and endeavors.

About the author

Aymen El Amri is a polymath software engineer, author, and entrepreneur. He is the founder of FAUN Developer Community[226], a platform dedicated to helping developers in their continuous learning journey. He has penned numerous guides on software development, AI, and cloud technologies. Connect with him on LinkedIn[227] and Twitter[228].

Join the community

If this guide resonated with you, consider joining the FAUN community[229]. Stay updated with upcoming free and premium guides, courses, and weekly newsletters.

[226] https://faun.dev
[227] https://www.linkedin.com/in/elamriaymen/
[228] https://twitter.com/eon01
[229] https://faun.dev/join

Feedback

Your insights and feedback are invaluable. They play a pivotal role in the ongoing enhancement and evolution of this guide.

If this work has struck a chord with you, I'd be honored to receive your testimonial. Please email me at aymen@faun.dev[230]. Your experiences can guide and inspire future readers, and I'd be thrilled to share your words with our broader community.

[230] mailto:aymen@faun.dev