docker-deep-dive

Contents

- Introduction
  - VMware
  - Containers
    - Linux containers
    - Windows containers
    - Mac containers

        * Windows vs Linux containers
- Docker
  - Docker, Inc
  - Docker runtime and orchestration
  - Docker project - MOBY
  - Container ecosystem
  - Open container initiative
  - Kernel
    * Control groups (cgroups)
    * Namespaces (namespaces)
    * Cgroup & Namespaces
  - Containerization
  - More kernel components
    * OverlayFS
    * Seccomp
    * Capabilities
    * AppArmor & SELinux
    * Capabilities & NNP
    * Device control
    * Network virtualization
    * Resource limits
- Components
  - Images
  - Containers
    * Attaching
    * Stopping
    * Removing
  - Dockerfile
  - Docker engine
- Deep-dive
  - Images
    * Definition
    * Pulling
    * Names
    * Tags
    * Layers
    * Digest
    * Architecture
    * Deleting
  - Containers
    * Definition
  - Containerizing application
  - Definition
    * Getting the code
    * Building the docker file
    * Containerize the app and build the image
    * Running the app
    * Test connectivity
  - Closer look
  - Moving to production
- Swarm mode

    - Definition
    - Swarm
        * Tasks
        * Enabling
        * Service
- Networking
    - Definition
- Security
    - Definition
        * Namespaces
        * Control groups
        * Capabilities
        * Mandatory access control systems
        * Seccomp
    - Conclusion
    - Swarm security
    - Swarm join tokens
    - TLS and mutual authentication
    - Cluster store
    - Signing images
    - Secrets

Introduction

Applications run businesses if applications break businesses suffer and sometimes go away, these statement get truer every day. Most applications run on servers. And in the past, we could only run one application per server, the open systems world of Windows and Linux just did not have the technologies to safely and securely run multiple applications on the same server. So the story usually went something like this - every time the business needed a new application IT department would go out and buy a new server, and most of the time nobody knew the performance requirements of the new application ! This meant IT had to make guesses when choosing the model and size of servers to buy.

As a result, IT did the only thing it could do - it bought big fast servers with lots of resiliency. After all ,the last thing anyone wanted - including the business was under powered servers, under powered servers might be unable to execute transactions which might result in lost customers and lost revenue. So IT usually bought bigger servers than were actually needed. This resulted in huge numbers of servers operating as low as 5-10% of their potential capacity. A tragic waste of company capital and resources.

VMware

Amid all of this, `vmware`, gave the world a gift - the virtual machine. And almost overnight the world changed into a much better place, We finally had a technology that would let us safely and securely run multiple business applications on a single server. This was a game changer. IT no longer needed to procure a brand new oversized server every time the business asked for a new application. More often than not they could run new apps on existing servers that were sitting around with spare capacity.

All of a sudden we could squeeze massive amounts of value out of existing corporate assets such as servers resulting in a lot more bang for the company's buck. However there is always a but ! As great as virtual machines are they are not perfect, the fact that every virtual machine requires it own dedicated OS is a major flaw, every OS consumes CPU, RAM and storage that could otherwise be used to power more applications, Every OS needs patching and monitoring. And in some cases every OS requires a license. All of this is a

waste of op-ex and cap-ex. The virtual machine model has other challenges too. Virtual machines are slow to boot and portability is not great - migrating and moving virtual machine workloads between `hypervisors` and cloud platforms is harder than it needs to be.

Containers

For a long time the big web-scale players like google have been using container technologies to address these shortcomings of the `VM` model. The container model the container is roughly analogous to the virtual machine. The major difference though, is that every container does not require a full blown OS. In fact all containers on a single host share a single OS. This frees up huge amounts of system resources such as CPU, RAM and storage. It also reduces potential licensing costs and reduces the overhead of OS patching and other maintenance. This results in savings on the `ap-ex` and `op-ex` fronts. Containers are also fast to start and ultra-portable. Moving container workloads from your laptop to the cloud and then to virtual machines or bare metal in your data center is a breeze.

Linux containers

Modern containers, started in the Linux world and are the product of an immense amount of work from a wide variety of people over a long period of time. Just as one example, Google has contributed many container related technologies to the Linux kernel. Without these and other contributions we would not have modern containers today. Some of the major technologies that enabled the massive growth of containers in recent years include `kernel namespaces` and `control groups` and of course - `Docker`. Despite all of this work, containers remained complex and outside of the reach of most organizations, it was not until docker came along that containers were effectively democratized and accessible to the masses.

Windows containers

Over the past few years, Microsoft, has worked extremely hard to bring docker and containers technologies to the windows platform. The core windows technologies required to implement containers are collectively referred to as Windows Containers. The user space tooling to work with these containers is docker.

Mac containers

There is currently no such thing as Mac containers, however one can use Linux containers on a mac using the Docker for mac product. This works by seamlessly running your containers inside of a `lightweight Linux VM running on the Mac`.

Windows vs Linux containers

It is vital to understand that a running container uses the kernel of the host machine it is running on. This means that a container designed to run on a host with a Windows kernel will not run on a Linux host. This means that one can think of it like this a at a high level - Windows containers require windows host, and Linux containers require a Linux host. It is however possible to run Linux containers on Windows machines, using the Docker for `Windows or WSL`. This is an area that is developing fast.

Docker

What is docker really, that could refer to three things:

- Docker, Inc - is the company behind the docker project
- Docker the container `runtime` and orchestration technology
- Docker the open source project - also called `Moby`

Docker, Inc

Docker is a software that runs on Linux and Windows. It creates, manages and orchestrates containers. The software is developed in the open part of the Moby open-source project on github. Docker, Inc is a company based out of San Fran, and is the overall maintainer of the project. There is also a commercial version Docker with more support and contracts. The company is founded by Solomon Hykes. Interestingly, Docker started its life as a platform as a service, provider called dotCloud. Behind the scenes, the dotCloud platform leveraged Linux containers. To help them create and manage these containers they built an internal tool that they named Docker

In 2013 the dotCloud PaaS business was struggling and the company needed a new lease of life. To help with this they hired Ben Gloub as new CEO, rebranded the company as Docker, Inc got rid of the dotCloud platform, and started a new journey with a mission to bring docker and containers to the world. Today Docker, Inc, is widely recognized as an innovative technology company with a market valuation said to be in the billions. Since becoming Docker, Inc they have made several small acquisitions for undisclosed fees, to help grow their portfolio of products and services.

Docker runtime and orchestration

When most technologists talk about Docker they are referring to the Docker engine, The docker engine is the infrastructure plumbing software that runs and orchestrates containers. If you are a VMware admin you can think of it as being similar to ESXi. In the same way that ESXi is the core hypervisor technology that runs virtual machines, the Docker Engine is the core container that runtime that runs containers. All other Docker, Inc, and 3rd party products plug into the Docker Engine and build around it.

Docker project - MOBY

The term docker is also used to refer to the open source Docker project. This is the set of tools that get combined into things like the docker deamon, and client you can download and install from the docker.com host. However the project was officially renamed as the Moby project at DockerCon in 2017. The goal of the Moby project is to break Docker down into more modular components and to do this, in the open. It is hosted on GitHub and you can see a list of the current sub-projects and tools included in the Moby repository. The core Docker Engine project is currently located at https://github.com/moby/moby. As an open-source project the source code is publicly available and you are free to download it, contribute to it, tweak it and use it, as long as you adhere to the license.

Looking at the commit history, one will immediately notice that some of the contributors are companies as RedHat, Microsoft and IBM, Cisco and HPE. Most of the project and its tools are written in golang, the relatively new system level programming language from Google also known as Go.

Container ecosystem

Once of the core philosophies at Docker is often referred to as Batteries included but removable. This is a way of saying you can swap out a lot of the native Docker stuff and replace it with stuff from 3rd parties. A good example of this is the networking stack. The core Docker product ships with built in networking, but the networking stack is pluggable meaning you can rip out the native Docker networking and replace it with something else from a 3rd party

In the early days it was common for 3rd party plugins to be better than the native offerings that shipped with docker. However this presented some business model challenges for Docker, After all, Docker has to turn a profit at some point to be a viable long term business. As a result, the batteries that are included are getting better and better.

Open container initiative

The OCI is a relatively new governance council responsible for standardizing the most fundamental components of container infrastructure such as image format and container runtime. It is also true that no discussion of the OCI is complete without mentioning a bit of history. And as with all accounts of history, the version you get depends on who is doing the talking.

From day one, use of Docker has grown like crazy. More and more people used it in more and more ways for more and more things, so it was inevitable that somebody was going to get frustrated, this is normal. The TLDR of this history according to Nigel is that a company called COREOS did not like the way docker did certain things. So they did something about it. They created a new open standard called APPC that defined things like image format and container runtime. They also created an implementation of the spec called rkt - pronounced rocket. This put the container ecosystem in an awkward position with two competing standards

This threatened to fracture the ecosystem and presented users and customers with a dilemma. While competition is usually a good thing, competing standards is not. This caused confusion and slowdown of user adoption. Not good for anybody, with this in mind everybody did their best to act like adults, and came together to form the OCI - a lightweight agile council to govern container standards. The OCI has published two specifications - image-spec and runtime-spec. An analogy that is often used when referring to these two standards is rail tracks. These two standards are like agreeing on standard sizes and properties of rail tracks. Leaving everyone else free to build better trains carriages better signaling systems better stations, all safe in the knowledge that they will work on the standardized tracks. Nobody wants two competing standards for rail track sizes

Kernel

Control groups and namespaces are foundational features in the Linux kernel that enable containerization by providing resource control and isolation. Together they form the building blocks for container runtimes like Docker, Podamn and Kubernetes. To appreciate their role it is essential to understand each in depth and how they synergize to create the container abstraction.

Control groups (cgroups)

Control groups are a Linux kernel feature that provides mechanisms for limiting prioritizing and monitoring the usage of system resource such as CPU, memory and disk I/O, and network bandwidth. Introduced in 2007, cgroups allow processes to be grouped hierarchically and to apply resource constraints or quotas at at the group level. Cgroups work by organizing processes into "control groups", which are hierarchical structures where resource limits and accounting are defined. This hierarchy is represented as virtual filesystem (often mounted at /sys/fs/cgroup), with directories corresponding to cgroups. Each cgroup can enforce specific limits to priorities for a particular type of resource. For instance:

1. CPU and CPU Shares: Cgroups allow limiting the amount of CPU time a process or group of processes can use. For example, if two containers are running on the same host, you can allocate more CPU time to one container over the other.

2. Memory Limits: By setting memory constraints, cgroups ensure that a process cannot exceed a specific memory threshold, thereby preventing memory exhaustion on the host system.

3. Block in/out network bandwidth: these can be throttled to ensure fair sharing among processes or to priorities critical workloads.

Cgroups are hierarchical, meaning resource limits propagate downward in the tree. This hierarchy enables complex configurations, such as allocating a fixed percentage of CPU to a parent cgroup and then subdividing that allocation among child cgroups.

Without cgroups processes running on the same host would contend for resources without constraints leading to potential resource starvation or overuse. In the context of containers, cgroups ensure that each container operates within its allocated resources, making them predictable, and performant even in multi tenant environment.

Namespaces (namespaces)

Namespaces, another Linux kernel feature isolate global system resources for groups of processes, by scoping resources, namespaces provide the illusion that a process or group of processes is running on its own system, separate from the others. This isolation is the backbone of containerization, as it ensures that processes inside a container are unaware of and unaffected by the processes and resources outside their namespace.

There are several types of namespaces, each targeting a specific system resource or function:

1. PID Namespace: Provides isolation for process ID. Processes inside a PID namespace see a separate process tree starting from PID 1, often the init system within the container. This ensures that processes in one container cannot see or signal processes in another container or the host
2. Mount namespace: isolates filesystem mount points. Each container can have its own root filesystem and mount points separate from the host or other containers. This is critical for providing a private file system view for containers.
3. UTS namespace: isolates system identifiers such as host name domain name. This allows containers to have their own hostname decoupled from the host system's identity
4. Network namespace: isolates network resources, including IP addresses routing tables and sockets. Containers can have their own virtual network interfaces and IP address, managed independently of the host or other containers.
5. `IPC` namespace: isolates inter process communication resources, such as shared memory and semaphores ensuring that contains cannot inadvertently interfere with one another's IPC mechanisms
6. User namespace: provides user and group ID isolation , allowing process inside a container to have different user ID from the host.

By leveraging namespaces each container can function as if it is running on a standalone system with a private set of resources. However, the host kernel orchestrates and manages these namespaces allowing containers to coexist on a shared kernel.

Cgroup & Namespaces

Cgroups and namespace complement each other to create the container abstraction:

1. Isolation - namespaces ensure that a container processes , network, filesystem and other system resources are isolated from the host and from the other containers. This isolation is crucial for security and the lightweight vitalization illusion that containers offer.
2. Resource control - control groups regulate how much of the resources isolated by the namespaces a container can consume, this combination prevents resource contention ensuring fair and predictable usage.

For example a container running a web server might operate within its own network namespace, with its own virtual NIC and IP address, mount namespace, with a private filesystem view, and PID namespace, with its own process tree. Simultaneously, cgroups ensure that the container cannot exceed 512MB of RAM and 50% of the CPU usage.

Containerization

Containers are not independent operating systems like virtual machines, they are processes running on the host operating system, constrained and isolated by cgroups and namespaces. These technologies allow containers to share the host kernel while appearing isolated, enabling the following key benefits

1. Lightweight - since containers share the host kernel, the require far fewer resources than VM which emulate hardware and run separate OS instances.
2. Fast start and shutdown - containers start quickly because they do not boot an entire operating system. They are essentially just processes with extra isolation, running on the host, just any other user level user process
3. Scalability - with cgroups enforcing resource limits, many containers can run concurrently on the same host without over provisioning

Modern container runtimes like Docker encapsulate these capabilities by automating the creation and management of namespaces and cgroups for each container, Tools like Kubernetes further build on this foundation to orchestrate containers across distributes systems, ensuring high availability and scalability.

In conclusion cgroups and namespaces by isolating and managing resources, transform the Linux kernel into a powerful platform for containerization. They encapsulate processes in a lightweight portable and secure manner, paving the way for the cloud native ecosystem that underpins much of today's software industry and infrastructure

More kernel components

Beyond the already mentioned components the Linux kernel contains a lot of other components which facilitate the deployment and containerization of applications as we know them nowadays. Here are some of them

OverlayFS

That is a union filesystem that allows multiple layers of filesystems to be stacked. It plays a key role in containerization by enabling the creation of lightweight writable layers on top of read-only image layers. The way it works is that when a container is created, a writable layer is added on top of a resad only base image. Any modification made by the container are written to this top layer, while the base image remains unchanged. This enables:

- Efficient sharing of base images between containers.
- Minimal disk usage since only changes are stored.
- Fast container creation by simply stacking layers

In containers the `OverlayFS` underpins the storage driver mechanism in Docker and other container runtimes, making image building sharing and running highly efficient.

Seccomp

Seccomp is a Linux kernel security feature that restricts the system calls (syscalls) a process can make. Containers use seccomp to reduce the attack vector / surface by preventing them from invoking potentially dangerous syscalls. The way this works is that seccomp operates as syscall filter. A process or container is provided with a list of allowed syscalls, any syscall not on the list is blocked. This is typically configured using a seccomp profile.

In containerization this play key role in protecting the host from container exploits by limiting access to sensitive syscalls, enables fine grained control over what containers can and cannot do or execute

Capabilities

Linux capabilities break down the all powerful root privileges into discrete units of authority. Containers often run as root within their own namespaces but they are stripped of unnecessary capabilities to mitigate risks. The way this works is by having the kernel allowing processes to drop or retain specific capabilities, like CAP_NET_ADMIN, for network management. For example a container may have the ability to bind to privileged ports but not modify kernel parameters

In containerization this enhances security by limiting what containers can do, even when running as root, reduces the risk of privilege escalation.

AppArmor & SELinux

These are two kernel frameworks for enforcing mandatory access control (MAC) policies. These frameworks restrict how applications or containers interact with the system. The way they work is that AppArmor defines file and process level access policies for containers, it is path-based meaning access is controlled based on file paths. SELinux uses labels to define granular access policies for processes, files and other resources.

In containerization ensures that containers can only access files, directories and resources explicitly allowed by their profiles or labels, also protects the host system from compromised or malicious containers.

Capabilities & NNP

Beyond capabilities the "No New Privileges" is a kernel flag feature that ensures a process cannot gain additional privileges through mechanism like `setuid` binaries. The way this works is that when `NNP` is enabled, even if a containerized process runs a binary with `setuid` permissions, it cannot escalate its privileges.

```
setuid - short for set user identity allow users to run an executable with the file system
 permissions of the executable's owner, and to change behavior in directories. They are
often used to allow users on a computer system to run programs with a temporarily elevated
 privileges to perform a specific task.
```

Device control

The Linux kernel provides mechanism for controlling access to hardware devices through the device cgroup and the `mknod` syscall. The way this works is that the device cgroup allows fine grained control over which devices a container can access - block devices, character devices etc. Containers can also be restricted from creating new device nodes using `mknod`

In containerization prevents unauthorized access to host devices like storage or network interfaces, limits the potential for containers to interact directly with sensitive hardware

Network virtualization

Linux networking stack provides features that are critical for container networking, including virtual Ethernet (`veth`) pairs, network bridges and like `iptables`. The way this works containers are typically connected to the host network through virtual Ethernet pairs, where one end resides in the container's network namespace and the other in the host's namespace or bridge. Network bridges like `docker0` or `CNI` plugins create virtual networks to interconnect containers

Resource limits

Resource limits - `rlimits` provide per-process controls over resources like file descriptors, stack size, and CPU time. While not as flexible as cgroups, `rlimits` play a complementary role in containerized environments.

The way this works is that `rlimits` are set using the `setrlimit` syscall, constraining individual process behavior.

In containerization it is used to impose additional resource restrictions at the process level, prevents runaway resource consumption within a container by a rogue process

Components

The idea of this chapter is to give you a quick big picture of what Docker is all about before we dive in deeper in later chapters.

Images

A good way to think of a Docker image is an object that contains a `filesystem` and an application. If you work in operations, it is like a virtual machine template. Another analogy that could be made is that the image is like a class definition, while the container is like an instance of a class. Getting images onto your Docker host is called pulling.

```
# to pull the target image locally
docker image pull ubuntu:latest

# to display all the images locally
docker image ls
```

There are other interesting images such as the `microsoft/powershell` image, which contains the `windows nano server` with a `powershell`. Or the `microsfot/iis` image which in turn contains and image with an `IIS`

Containers

After an image has been pulled locally on the docker host daemon, the `docker container run` command can be used to launch an instance of this container.

```
docker container run -it microsoft/powershell:nanoserver PowerShell.exe
```

If one looks closely at the output from the command above, notice that the shell prompt has changed. This is because your shell is now attached to the shell of the new container. The -it flag tells the daemon to make the container interactive and to attach our current terminal to the shell of the container. It also tells the container to start the `PowerShell` executable before attaching and after starting the container. To list the containers currently created and/or running, one can use the following command

```
# this command will list all containers ever crated, running or stopped,
   along with the container id and container name,
# status and the image the container is started with
docker container ls
```

Attaching

Once a container is running, one can also attach to it, with the docker exec command, as the container from the previous steps is still running, one can attach to it with `docker container exec -it <container-id | container-name> <command-name>`

```
# the id here is an example id of a running container, this id can be
   obtained from the docker container ls command
docker container exec -it e2b69eeb5cb bash
```

The <command-name> above tells the container which process to start within the container when attaching to the container.

Stopping

To stop the container one can do that using the `docker container stop <container-id | container-name>`, this will stop the container but it will still remain ready to be started again with `docker container start` however one can also remove the container using `docker container rm` instead

```
docker container stop e2b69eeb5cb
```

Removing

To remove a container, removing and deleting all resources associated with it, one should use the `docker container rm <container-id | container-name>` command instead, this will completely remove all container resources, , however the image the container was created with will not be removed, since it is not part of the container's resources

```
docker container rm e2b69eeb5cb
```

Dockerfile

Usually some user applications have an additional file which is used to build on top of existing docker images, those are called `Dockerfile`, actually, the base images are also `Dockerfiles` as well, in essence.

```
FROM alpine
LABEL maintainer="yourname@hotmail.com"
RUN apk add --update nodejs nodejs-npm
COPY . /src
WORKDIR /src
RUN npm install
EXPOSE 8080
ENTRYPOINT ["node", "./app.js"]
```

The `Dockerfile` represents an image, just like the base images, the difference is that they usually build on top of some base image, custom user rules. In the example above, the `Dockerfile` creates an image which is based on the alpine base image, however it enhances it by installing node, the base alpine version has only the regular `coreutil` applications

```
# execute this command from a directory which contains a Dockerfile
docker image build -t test:latest .
```

To create an image from a custom `Dockerfile`, one can use the following command above, this will create an image called test, with the version of latest, the -t argument refers to the process of tagging an image, or in other words putting an identifier on the image

`Note that each command/line above in the Dockerfile creates what is called a layer, the layers is what docker deamon uses internally to build the images, each layer is built independently, meaning that if one layer changes that is the only layer that is going to be re-build, which makes modifying images very quick and efficient.`

Docker engine

The docker engine is the core software that runs and manages the containers, we often refer to it as simply as Docker, or the docker platform. If you know a thing or two about VMware it might be useful to think of it as being like ESXi in the VMware world. The Docker engine is modular in design with many swappable components, where possible these are based on open standards outlined by the OCI.

In many ways the docker engine is like a car engine - both are modular and created by connecting many small specialized parts. The major components that make up the docker engine are - Docker cilent, Docker daemon, containerd and runc. Together they create and run containers.

When docker was first released, the docker engine had two major components - the `deamon` and `LXC`. The daemon was a monolithic binary, It contains all of the code for the docker client, the docker api, the container `runtime` image builds and much more. The `LXC` component provided the `deamon` with access to the fundamental building blocks of containers such as kernel `namespaces` and control groups. The docker daemon was using the `LXC` to interact with the host kernel.

To get rid of the `LXC` was an issue, first it is Linux specific, this was a problem for a project that had aspirations of being cross platform. Second up being reliant on an external tool for something so core to the project was a huge risk that could hinder development, as a result the Docker, Inc developed their own tool called `libcontainer` as a replacement for `LXC`. The goal of `libcontainer` was to be a platform agnostic tool that provided Docker with access to the fundamental container building blocks that exist inside the OS.

To get rid of the monolithic `deamon`, the aim was to break out as much of the functionality as possible from the `deamon`, and re-implement it in smaller specialized tools. These specialized tools can be swapped out as well as easily used by third parties to build other tools. This plan followed the UNIX philosophies of building small specialized tools that can be pieced together into large tools. This breaking down process is still ongoing, however it has already seen all of the container execution and container runtime code entirely removed from the daemon and refactored into small specialized tools such as runc (container runtime) and containerd (container supervisor)

- runc - as already mentioned `runc` is the reference implementation of the OCI container runtime spec, Docker, Inc was heavily involved in defining the spec and developing runc. Runc is small, it is effectively a lightweight CLI that wraps around `libcontainer`, it has a single purpose in life - to create containers.
- containerd - in order to use `runc`, the Docker engine needed something to act as a bridge between the `deamon` and `runc`. This is where `containerd` comes into the picture. `Containerd` implements the execution logic that was pulled out of the Docker daemon, this logic was obviously refactored and tuned when it was written as `containerd`. `Containerd` is a container supervisor - it is responsible for container `lifecycle` operations such as starting and stopping containers, pausing and un-pausing them and destroying them. Like `runc containerd` is small lightweight and designed for a single task in life - only interested in container `lifecycle` operations.

The most common way of starting containers is using the Docker CLI. The following docker container run command will start a simple new container based on the alpine:latest image

```
docker container run --name ctr1 -it alpine:latest sh
```

When this command is typed int the docker `CLI`, the docker client converts them into the appropriate API payload and POSTS them to the correct API endpoint. This API is implemented in the daemon, it is the same rich versioned, REST API that has become a hallmark of Docker and is accepted in the industry as a de-facto container API. Once the `deamon` receives the command to create a new container it makes a call to `containerd`. The daemon communicates with `containerd` via a CRUD like API over `gRPC`. Despite its name `containerd` cannot actually create containers, it uses `runc` to do that. It converts the required Docker image into an `OCI` bundle and tells `runc` to use this to create a new container. `Runc` interfaces with the OS kernel

to pull together all of the constructs necessary to create a container, in Linux these include `namespaces` and `cgroups`. The container process is started as a child process of `runc`, and as soon as it is started `runc` will exit. The benefit of this approach is that one can perform updates to the Docker daemon or the `containerd` without affecting, the containers.

There is a special component between the `containerd` and `runc`, called `shim`. The shim is integral to the implementation of the `deamonless` containers, i.e decoupling running a container from the daemon for things like upgrading the `deamon` without killing containers. As already shown, containerd uses `runc` to create a new container, in fact it forks a new instance of runc for every container it creates. However once each container is created its parent `runc` process exits. This means we can run hundreds of containers without having to run hundreds of runc instances. Once a container's parent `runc` terminated, the associated containers-shim process becomes the container's parent process. Some of the responsibility of the shim performs a container's parent include

- keeping any stdin and stdout streams open so that when the `deamon` is restarted the container does not terminate due to pipes being closed.
- reports the container's exit status back to the daemon.

On a Linux system the components we have discussed are implemented as separate binaries as follows:

- dockerd (`deamon`)
- docker-containerd (containerd)
- docker-containerd-shim (shim)
- docker-runc (runc)

All of those can be seen running if one runs the `ps` command in the host. Some of them will be present when the system has running containers only, like the shim, or when a container is starting, like runc.

Deep-dive

Images

Once can think as images as being like VM templates. A VM template is like a stopped VM - a Docker image is like a stopped container. The images are usually pulled from a registry. The most popular registry is the Docker Hub, but other do exist. The pull operation downloads the image to the host machine, where it can be used to start one or more docker containers. Images are made up of multiple layers that get stacked on top of each other and represented as single object. Inside of the image is a cut-down operating system and all of the files and dependencies required to run an application. Because containers are intended to be fast and lightweight images tend to be small.

Definition

As mentioned a couple of times already that images are like stopped containers (or classes, coming from the programming world). In fact one can stop a container and create a new image from it. With this in mind, images are considered a build time constructs where as containers are run-time constructs. One can think of images as a snapshot of a container in a given

The whole purpose of a container is to run an application or service, this means that the image a container is created from must contain all OS and application files required to run the service. However, containers are all about bein fast and lightweight. This means that the images they are build from are usually small and stripped of all non-essential parts. For example Docker images do not sip with 6 different shells, they do not contain a kernel, all containers running on a docker host share access to the host's kernel. For these reasons we sometimes say images contain just enough operating system - usually OS related files and filesystem objects.

The official Alpine Linux docker image is about `4mb`, in size and is an extreme example of how small Docker images can be. The official Ubuntu Docker image which is currently about 120MB. Windows based images tend to be bigger than Linux ones, because of the way that the Windows OS works, For example the latest Microsoft .NET image is over 2GB when pulled and uncompressed, the windows server nano (2016) is slightly over 1GB.

Pulling

Pulling images is the process of getting images onto a Docker host. These images are usually pulled from what is called image registries. Docker images are stored in these registries. The most common registry is the Docker Hub. Other registries exist, including 3rd party registries and secure on-premise registries. However the docker client is opinionated and defaults to the Docker Hub. Image registries contain multiple image repositories, in turn these repositories contain multiple images. Let us take the Ubuntu image as example - it is one of many repositories in the registry, the Ubuntu repository in turn has many versions of the Ubuntu image. The Docker hub has a concept of official repositories and unofficial ones. As the name suggests official repositories contain images that have been vetted by Docker. This means they should contain up-to-date high quality code that is secure well-documented and in line with best practices, Unofficial repositories can be like the wild west - you should not expect them to be safe, well documented, or even work. That is not saying everything in unofficial repositories is bad. Most of the populate operating systems and applications have their own official repositories. The are easy to spot as they live at the top level of the Docker hub `namespace`. These lie under the following URL formats - `https://hub.docker.com/_/<image>` While personal images are behind URL formats such as `https://hub.docker.com/r/<user>/image`

- `nginx` - https://hub.docker.com/_/nginx
- `busybox`- https://hub.docker.com/_/busybox
- `redis`- https://hub.docker.com/_/redis
- `mongo` - https://hub.docker.com/_/mongo

Names

Addressing images from the official repositories is as simple as giving the repository name and tag separated by a colon (:). The format for docker image pull when working with an image from an official repository is - `docker image pull <repository>:<tag>`. For example one can easily pull a given version of an image using the following commang

```
# This will pull the image tagged as 3.3.11, from the official mongo
   repository
docker image pull mongo:3.3.11
# This will pull the image tagged as latest, from the official redis
   repository
docker image pull redis:latest
# This will pull the image tagged as latest, from the official alpine
   repository
docker image pull alpine
```

Note that even though the alpine example above does not specify any version, by default the latest one is assumed. Another point, the latest tag does not have any magical powers, just because an image is tagged as latest, does not guarantee it is the most recent image in the repository. For example the most recent image in the alpine repository is usually tagged a edge. Latest in this case refers to the latest stable version, but not the actual latest latest available

Pulling image from an unofficial repository is essentially the same - the only difference is that the name of the repository is `prepended in front`, using a forward slash as a separator between the image name and the

repository name.

```
# Here the image name is tu-demo, however the source repository is
   nigelpoulton
docker image pull nigelpoulton/tu-demo:v2
```

Tags

The other important images with multiple tags, A single image can have as many tags as want wishes, This is becaue tags are arbitrary alpha-numeric values that are stored a metadata alongside the image, To pull all images in a repository add the -a flag to the docker image pull command.

```
# This will all images, that are tagged in the repository nigelpoulton
docker image pull -a nigelpoulton/tu-demo
```

Now since one image can have multiple tags, it is possible that the command above pulls images where the latest image is actually such that it already exists with another tag - say the `tu-demo` image has three versions, `v1 and v2`, and latest, however it is possible that the latest is actually also the one tagged as v2, in the end the image pull command will simply pull two images - `v1 and v2`

Layers

Docker image is just a bunch of loosely connected read only layers. Docker takes care of stacking these layers and representing them as a single unified object. There are few ways to see and inspect the layers that make up an image, and we have already seen one of them.

```
docker image pull ubuntu:latest
latest: Pulling from library/ubuntu
952132ac251a: Pull complete
82659f8f1b76: Pull complete
c19118ca682d: Pull complete
8296858250fe: Pull complete
24e0251a0e2c: Pull complete
Digest: sha256:f4691c96e6bbaa99d...28ae95a60369c506dd6e6f6ab
Status: Downloaded newer image for ubuntu:latest
```

Taking a look at the output of the pull command, one can see that each line in the output above that ends with Pull complete, represents a layer in the image, that was pulled. So the example image above has 5 layers. Another way to see the layers of an image is to inspect the image with the docker image inspect command, the example below inspects the same image, `ubuntu:latest`

```
docker image inspect ubuntu:latest

"Id": "sha256:bd3d4369ae.......fa2645f5699037d7d8c6b415a10",
"RepoTags": [
    "ubuntu:latest"
    "RootFS": {
        "Type": "layers",
        "Layers": [
            "sha256:c8a75145fc...894129005e461a43875a094b93412",
            "sha256:c6f2b330b6...7214ed6aac305dd03f70b95cdc610",
            "sha256:055757a193...3a9565d78962c7f368d5ac5984998",
            "sha256:4837348061...12695f548406ea77feb5074e195e3",
            "sha256:0cad5e07ba...4bae4cfc66b376265e16c32a0aae9"
```

```
        ]
    }
]
```

The trimmed output shows that the image has 5 layers, Only this time they are shown using their `SHA256` hashes. However both commands show that the image has 5 layers.

`Note the docker history command shows the build history of an image and is not a strict lit of layers in the image, for example some Dockerfile instructions used to build an image do not result in layers being created. These include MAINTAINER, ENV, EXPOSE, ENTRYPOINT`

All Docker images start with a base layer and as changes are made and new content is added, new layers are added on top. Further more a newer layer, can override, or obscure another older layer, meaning that if a layer adds in a file in certain system directory, which is then added again in the same location, under the same name, in another layer, it is overridden, or stacked. Meaning that that allows one to build upon base image layers, without having to modify the actual base layer itself.

```
docker image pull -a nigelpoulton/tu-demo

latest: Pulling from nigelpoulton/tu-demo
237d5fcd25cf: Pull complete
a3ed95caeb02: Pull complete
<Snip>
Digest: sha256:42e34e546cee61adb100...a0c5b53f324a9e1c1aae451e9
v1: Pulling from nigelpoulton/tu-demo
237d5fcd25cf: Already exists
a3ed95caeb02: Already exists
<Snip>
Digest: sha256:9ccc0c67e5c5eaae4beb...24c1d5c80f2c9623cbcc9b59a
v2: Pulling from nigelpoulton/tu-demo
237d5fcd25cf: Already exists
a3ed95caeb02: Already exists
<Snip>
eab5aaac65de: Pull complete
Digest: sha256:d3c0d8c9d5719d31b79c...fef58a7e038cf0ef2ba5eb74c
Status: Downloaded newer image for nigelpoulton/tu-demo
$$
docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
nigelpoulton/tu-demo v2 6ac...ead 4 months ago 211.6 MB
nigelpoulton/tu-demo latest 9b9...e29 4 months ago 211.6 MB
nigelpoulton/tu-demo v1 9b9...e29 4 months ago 211.6 MB
```

Sharing layers is also possible, between different images. This leads to efficiencies in space and performance. For example when Docker pulls images, with `docker image pull` command, Docker is smart enough to recognize when it is being asked to pull an image layer that is already has ac copy of. In this example the Docker pulled the image tagged as latest first. Then when it went to pull v1 and v2 images it noticed that it already had some of the layers that make up those images. This happens because the three images in the repository are almost identical, and therefore share many layers.

Digest

So far we have shown how to pull images by tag, and this is by far the most common way. But it has a problem - tags are mutable. This means it is possible to accidentally tag an image with an incorrect tag. Sometimes it is even possible to tag an image with the same tag as an existing but different image, this can cause problems

As an example imagine that you have got an image called `golftrack:1.5` and it has known bug, you pull the image apply a fix and push the updated image back to its repository with the same tag. Take a second to understand what just happened there. So what just happened is that we have an image called `golftrack:1.5` that has a bug. That image is being used in the production environment, the image is pulled and fix is applied. Then comes the mistake the image is pushed back with the fixed state, however under the same tag as the vulnerable image. `How are you going to know which of your production systems are running the vulnerable image and which are running the patched image, both images have the same tag !`

This is where image digests come to the rescue. Docker introduced a new content addressable storage model. As part of this model, all images now get a `cryptographic` content hash. For the purposes of this discussion we will refer to this hash as the digest. Because the digest is a hash of the contents of the image, it is not possible to change the contents of the image without the digest also changing. This means digests are immutable, this helps to avoid the problem just mentioned

Every time you pull an image the docker image pull command will include the image's digest as part of the return code. You can also view the digests of the images in your Docker host's lock repository by adding the –digests flag to the docker image ls command, these are both shown in the following example below.

```
docker image pull alpine
Using default tag: latest
latest: Pulling from library/alpine
e110a4a17941: Pull complete
Digest: sha256:3dcdb92d7432d56604d...6d99b889d0626de158f73a
Status: Downloaded newer image for alpine:latest

docker image ls --digests alpine
REPOSITORY TAG    DIGEST              IMAGE        ID CREATED   SIZE
alpine     latest sha256:3dcd...f73a 4e38e38c8ce0 10 weeks-ago 4.8 MB
```

There is more to digests, since they represent the content of the image, which is simply but the stack of layers which make up this image, what happens when an image is uploaded, the layers are compressed to save bandwidth, as well as space in the registry's blob store. Cool but compression a layer changes its content, this means that it's content hash will no longer match after the push or pull operations. This is a problem. For example when you push an image layer to Docker Hub, Docker hub will attempt to verify that the image arrived without being tampered with en-route. To do this it turns a hash against the layer and checks to see if it matches the hash that was sent with the layer. Because the layer was compressed (changed) the has verification will fail. To get around this each layer also gets something called a distribution hash, this is a hash of the compressed version of the layer, when a layer is pushed and pulled from the registry, its distribution hash is included and this is what is used to verify that the layer arrived without being tempered with. This content-addressable storage model vastly improves security by giving us a way to verify image and layer data after push and pull operations.

Architecture

Docker now includes support for multi platform and multi Architecture images. This means a single image repository and tag to have an image for Linux, Arm, PowerPc and so on. Other examples exist. To enable

this the Registry API supports a fat manifest as well as an image manifest. Fat manifests list the architectures supported by a particular image, whereas image manifests list the layers that make up a particular image.

What happens is that when an image is pulled from the Docker hub, the Docker client makes the relevant API requests to the registry, if a fat manifest exists for that image, it will be parsed to see if an entry exists for the current local host's arch (assume as pull the image on a local host which is an x86-64, Linux machine). It is exists the image manifest for that image is retrieved and parsed for for the actual layers that make up the image, the layers ae identified by their crypto ID and are pulled from the registry's blob store.

Deleting

When an image is no longer needed you can delete it from your docker host, with the `docker image rm command`, rm is short for remove. Images can be deleted by an ID or by the name of the image along with the version, if a version is not provided, latest is assumed. If an image is in use by a running container you will not be able to delete it. A cool handy trick to remove all downloaded images which are downloaded locally, along with their layers is to use the `docker image rm` combined with the `docker image ls -q` commands

```
# This will list all image ids on new lines such as, handy way to avoid using
    something like xargs, with docker rm
docker image ls -q
bd3d4369aebc
4e38e38c8ce0

# This will remove all images, which are currently not used by containers and
    also locally downloaded
docker image rm $(docker image ls -q) -f
Untagged: ubuntu:latest
Untagged: ubuntu@sha256:f4691c9...2128ae95a60369c506dd6e6f6ab
Deleted: sha256:bd3d4369aebc494...fa2645f5699037d7d8c6b415a10
Deleted: sha256:cd10a3b73e247dd...c3a71fcf5b6c2bb28d4f2e5360b
Deleted: sha256:4d4de39110cd250...28bfe816393d0f2e0dae82c363a
Deleted: sha256:6a89826eba8d895...cb0d7dba1ef62409f037c6e608b
Deleted: sha256:33efada9158c32d...195aa12859239d35e7fe9566056
Deleted: sha256:c8a75145fcc4e1a...4129005e461a43875a094b93412
Untagged: alpine:latest
Untagged: alpine@sha256:3dcdb92...313626d99b889d0626de158f73a
Deleted: sha256:4e38e38c8ce0b8d...6225e13b0bfe8cfa2321aec4bba
Deleted: sha256:4fe15f8d0ae69e1...eeeeebb265cd2e328e15c6a869f
```

Containers

The container is the runtime instance of an image. In the same way that we can start a virtual machine from a virtual machine template we start one or more containers from a single image. The big difference between a VM and a container is that containers are faster and more lightweight instead of running a full-blown OS like a VM, containers share the OS/kernel with the host they are running on

The simplest way to start a container is with the docker container run command. The command can take a lot of arguments, but in its most basic form you tell it an image to use and a command to run `docker container run <image> <command>` - e.g. `docker container run -it ubuntu/bin/bash`, or `docker container run -it microsfot/powershell:nanoserver`

Containers run until the program they are executing exists. In the two examples above, the Linux container will exit when the bash shell exits, and Windows container will exit when the PowerShell process terminates.

A really simple way to demonstrate this is to start a new container and tell it to run the sleep command for 10 seconds. The container will start, run for 10 seconds and exit - `docker container run alpine:latest sleep 10.`

To manually stop a running container, one can use the docker container stop, which takes in the container id, then the container can be started with the docker container start, again taking the container id.

To remove a container or delete it, one can use the docker container rm, again taking the container id as an argument, however one can not delete a container that is running, it has to be stopped first

Definition

Assume for the example above, a single physical server, which is required to run 4 different applications, using either the virtual machine or container approach

VM model In the VM model, the physical server is powered on and the hypervisor boots, we are skipping the BIOS and bootloader code. Once the hypervisor boots it lays claim to all physical resources on the system, such as CPU, RAM, storage and NIC. The hypervisor then carves these hardware resources into virtual versions that look smell and feel exactly like the real thing. It then packages them into a software construct called a virtual machine. We then take those virtual machines and install an operating system and application on each one. We said we had a single physical server and needed to run 4 applications, so we have created 4 virtual machines, install 4 operating systems and then install the 4 applications.

Container model Things are a bit different in the container model. When the server is powered on your chosen OS boots, in the docker world this can be Linux or a modern version of Windows that has support for the container primitives in its kernel. As per the VM model, the OS claims all hardware resources. On top of the OS we install a container engine such as Docker. The container engine then takes OS resources such as the process tree, the filesystem and the network stack, and carves them up into secure isolated constructs called containers. Each container looks smells and feels like a real operating system. Inside of each container we can run an application. Like before we are assuming a single physical server with 4 applications. Therefore we would carve out 4 containers and run a single application inside of each.

Comparison At a high level we can say that the hypervisors perform hardware virtualization the carve up physical hardware resources into virtual versions. On the other hand the containers perform OS virtualization - they carve up OS resources into virtual versions. In the virtual machine model Every OS, consumes a slice of the processor, a slice of the memory, a slice of the storage and so on. Most need their own licenses as well as well as people and infrastructure to patch and upgrade them. Each OS also presents a sizable attack surface. This is often called the operating system tax, or the virtual machine tax, where every operating system you install consumes way too many resources. The container models has a single kernel running in the host operating system. It is possible to run tens or hundreds of containers on a single host with every container sharing that single kernel. That means a single operating system, consuming the processor, memory, storage and other resources, a single operating system that needs licensing, a single operating system that need upgrading and patching, and a single operating system, presenting an attack surface, all of that is a SINGLE operating system tax bill !

Lifecycle It is a common myth that containers can not persist data, they certainly can. A big part of the reason people think containers are not good for persistent workloads or persisting data, is because they are so good at non persistence stuff. But begin good at one thing does not meant you can not do the other well. Let us

```
# start the container, name it percy and use the latest ubuntu image
docker container run --name percy -t ubuntu:latest /bin/bash
```

```
# add some persistent data to the container, create a new file
cd tmp
ls -l
echo "test" > newfile
ls -l

# press Ctrl-PQ to exit the container without killing it, using Ctrl-D will
   kill the process, in this case bash, and
# will also kill the container, and exit

# stop the container
docker container stop percy

# list all containers (use the -a flag, which is similar to the unix ls -a,
   to list all files)
docker container ls -a

# start the container back up
docker container start percy

# start a new bash process and attach to the container, one can also use the
   attach instead of exec to attach to an
# already running process
docker container exec -it percy bash

# attach to the process running in the container, that would by default
   attach to stdout, and only to stdin if the
# process was started interactively, with the -i flag
docker container attach

cd temp
ls -la
# the `newfile` which was created above should be visible in the listing
```

Now what is going on, when the container is first created with docker container run the very first command it is started with will become the new entrypoint, in this case this is /bin/bash. This is also started with the `-t flag`, which creates a `pseudo tty, which forces the process to not terminate`, in this case we have a container running a persistent bash process, now calling `docker container stop will stop the container`, but the command with which it was started is remembered, so the subsequent `docker container start will start the same process` with a pseudo `tty`, then the `docker container exec -it percy` bash will actually start another bash process and attach to it. In the end one will end up with two bash processes, running in this container, can be verified using the docker top command. Note that `docker attach` will work for `stdout` just fine, but if one wishes to attach to a running process' `stdin` that process would have to have been started with the `-i flag`

Cleaning up There is a quick and dirty way to remove all containers on the host system, just to make sure that there are no left over resources, similarly to how we did for the images, one can also destroy and remove all containers, however that is not terminating them safely, meaning that it should be used with caution. The following command can be used to kill all containers

```
# remove all containers, listed with ls -aq, the -f flag stands for force,
   which means it will kill all running
# containers, regardless of their state
docker container rm $(docker container ls -aq) -f
```

Containerizing application

The process of taking an application and configuring i t to run as a container is called containerizing, or dockerizing. The process is somewhat simple

1. start with the application code
2. create a docker file that describes your app, its dependencies and how to run it
3. feed this docker file into the docker image build command.
4. sit back while docker build your application into a docker image.

Once the image is created, i.e the app is containerized, you are ready to ship and run it as a container.

Definition

The process below explains in more detail and walks through all the processes described above, for containerizing a Linux based nodejs web app. The process is the same for Windows. The process includes, getting the app code, inspect the docker file, containerize the app, run the app, test the app.

Getting the code

Let us assume that we already have the code, or we can clone a repository which contain the code. Assume that this is a regular nodejs project, which contains all the source files inside a folder called src, located inside the top level project folder.

Building the docker file

To build the docker file for our application, note that this docker file does a few interesting things, one of them being that the code of the application is copied inside the container, that is done during the image build phase, meaning that the code of the app will be built into the image. It is a common practice to keep the `Dockerfile` at the root of the project's folder, this way it is easier to refer to resources from the project, in this case the resource that we are interested in is the src folder, which contains the source of the project.

```
FROM alpine
LABEL maintainer="nigelpoulton@hotmail.com"
RUN apk add --update nodejs nodejs-npm
COPY . /src
WORKDIR /src
RUN npm install
EXPOSE 8080
ENTRYPOINT ["node", "./app.js"]
```

The `Dockerfile` also has a very important role which is to bridge the gap between the dev and the ops, which means that it can often be used to document the way a given application is supposed to be bundled and distributed, built and tested. Which is quite important.

`Dockerfile` starts with the FROM instruction. This will be the base layer of the image, and the rest of the app will be added on top as additional layers, this particular app is a Linux app, so it is important that the

FROM instruction refers to a Linux based image. If you are containerizing a Windows app you will need to specify the appropriate Windows base image.

Next is the LABEL and it specifies the maintainer of the image, Labels are simple key value pairs and are an excellent way of adding custom metatadata to an image, It is considered a best practice to list a maintainer of an image so that other people and potential users have a point of contact when working with it.

The RUN layer instructs the package manager to install nodejs and nodejs-npm into the image. The RUN instruction installs. The instruction installs these packages as a new image layer on top of the alpine base image created by the FROM alpine instruction

The COPY instruction copies in the app files from the build context the run instruction copies these files into the image as a new layer, the image now has three layers

Next the `Dockerfile` uses the `WORKDIR` instruction to set the working directory for the rest of the instructions in the file. This directory is relative to the image, and the information is added as metadata to the image config and not as a new layer.

Then the RUN npm install instruction uses npm to install application dependencies listed in npm package.json. It runs within the context of the `WORKDIR` set in the previous instruction and installs the dependencies as a new layer in the image.

The image now has four layers

- RUN npm install
- COPY . /src
- RUN apk add npm
- FROM alpine

The application exposes a web service on TCP port 8080 so the `Dockerfile` documents this with the EXPOSE 8080 instruction. This is added as image metadata and not an image layer. Finally the `ENTRYPOINT` instruction is used to set the main application that the image (container) should run. This is also added as metadata and not an image layer.

Containerize the app and build the image

Now that the application code and the `Dockerfiles` are ready the image is ready to be built. The following command will make sure to build a new image called web:latest. The period at the end of he command tells Docker to use the shell's current working directory as the build context, in other words it tells the docker CLI where to find the `Dockerfile` to use to build the image, in this case it is run from the current working directory where the `Dockerfile` is already located, therefore it is reasonable to use the `cwd - current working directory`

```
docker image build -t web:latest .

Sending build context to Docker daemon 74.75kB
Step 1/8 : FROM alpine
latest: Pulling from library/alpine
88286f41530e: Pull complete
Digest: sha256:f006ecbb8...d935c0c103f4820a417d
Status: Downloaded newer image for alpine:latest
---> 76da55c8019d
<Snip>
Step 8/8 : ENTRYPOINT node ./app.js
---> Running in c576be4427a7
---> e33cdd8266d0
```

```
Removing intermediate container c576be4427a7
Successfully built e33cdd8266d0
Successfully tagged web:latest
```

After the command finishes one can also verify that the build has been successfully by running the docker image ls command to verify that the new image is added to the local registry as well.

Running the app

The example application that we have containerized is a simple web server that listens on TCP port 8080. The following command will start a new container called `c1` based on the web:latest image we just created. It maps port 80 on the Docker host to port 8080 inside the container. This mean that you will be able to point a web browser at the DNS name or IP address of the Docker host and access the app.

```
# run the container in the background with -d, give it a simple name and
   expose the port 8080 from the container to port 80 on the host.
docker container run -d --name c1 -p 80:8080 web:latest

# check what is the status of the container after the command above has
   finished
docker container ls
```

Test connectivity

To test if everything is working fine, open up a web browser and navigate to and pint it to the DNS name or IP address of the host that the container is running on, this is usually `localhost or 127.0.0.1`

Closer look

Now that the application is containerized let us take a closer look at how some the machinery works, comment lines in a `Dockerfile` start with # character. All non comment lines are Instructions, Instructions take the format INSTRUCTION argument. Instruction names are not case sensitive but it is normal practice to write them in UPPERCASE. This makes reading the `Dockerfile` easier. The docker image build command parses the `Dockerfile` one line at a time starting from the top. Some instructions create new layers, whereas others just add metadata to the image. Examples of instructions that create new layers are FROM, RUN and COPY. Examples of instructions that create metadata are `EXPOSEEXPOSE`, `WORKDIR`, `ENV` and `ENTRYPOINT`. The basic premise is that - if an instruction is adding content such as files and programs to the image, it will create a new layer, if it is adding instructions on how to build the image and run the application it will create metadata. You can view the instructions that were used to build the image with the docker image history command.

```
docker image history web:latest

IMAGE     CREATED BY                                      SIZE
e33..6d0  /bin/sh -c #(nop) ENTRYPOINT ["node"./a...      0B
d38..20c  /bin/sh -c #(nop) EXPOSE     8080/tcp           0B
e2a..0b6  /bin/sh -c npm    install                       18.7MB
a8e..50e  /bin/sh -c #(nop) WORKDIR    /src               0B
23b..b58  /bin/sh -c #(nop) COPY       dir:03b6808e26dacac... 22kB
fda..b35  /bin/sh -c apk add --update nodejs  nodejs-npm  32.9MB
8d3..501  /bin/sh -c #(nop) LABEL      maintainer=nigelp...  0B
76d..19d  /bin/sh -c #(nop) CMD        ["/bin/sh"]        0B
<missing> /bin/sh -c #(nop) ADD        file:4583e12bf5caec4... 3.97MB
```

Two things from this output above are worth noting. First each line in the output corresponds to an instruction in the `Dockerfile`, the created by column even lists the exact instruction that was executed, second only 4 of the image layers are displayed in the output contain any data (the ones with non-zero values in the size columns) These correspond to the from, run and copy instructions in the `Dockerfile`. Although the other instructions look like they create layers, they only create metadata, but are still listed.

Use the docker image inspect command to confirm that only 4 layers were created.

```
docker image inspect web:latest
<Snip>
    "RootFS": {
    "Type": "layers",
    "Layers": [
        "sha256:5bef08...00324f75e56f589aedb0",
        "sha256:03f8d2...f7061341ab09fab9d2d5",
        "sha256:7bb5e2...5718961a7a706c5d0085",
        "sha256:110b48...541f301505b0da017b34"
    ]
   }
<Snip>
```

It is considered a good practice to use images from the official repositories with the from instruction. This is because they tend to follow best practices and be relatively free from known vulnerabilities. It is also a good idea to start with a FROM small images as this reduces the potential attack surface.

Moving to production

When it comes to docker images, big is bad, big means slow, big means hard to work with and big means a large attach surface, for these reasons, Docker images should be small, the aim of the game is to only ship production images containing the stuff needed to run your app in production, The problem is keeping images small was hard work. For example the way you write your `Dockerfiles` has a huge impact on the size of the image. A common example is that every RUN instruction adds a new layer. As a result it is usually considered a best practice to include multiple commands as part of a single RUN instruction - all glued together with double ampersands (&&) and backslash for mutliline command break.

Another issues is that we do not clean up after ourselves, we RUN a command against an image that pulls some build time tools and we leave those tools in the image when we ship it to production - not good. There are ways around this, most notably the builder pattern, but most require a discipline and added complexity.

The builder pattern required you to have at least two `Dockerfiles` one for development and one for production, you'd write your `Dockerfile.dev` to start from a large base image, pull in any additional build tools required and build your app, you'd then build an image from the `Dockerfile.dev` and create a container from it. You'd then use your `Dockerfile.prod` to build a new image from a smaller base image and copy over the application from the container

This approach was doable but at the expense of complexity, multi stage builds to the rescue. Multi-stage builds are all about optimizing builds without adding complexity. With multi stage builds we have a single `Dockerfile` containing multiple FROM instructions. Each FROM instruction is a new build stage that can easily copy artifacts from previous stages.

```
FROM node:latest AS storefront
WORKDIR /usr/src/atsea/app/react-app
COPY react-app .
RUN npm install
```

```
RUN npm run build

FROM maven:latest AS appserver
WORKDIR /usr/src/atsea
COPY pom.xml .
RUN mvn -B -f pom.xml -s /usr/share/maven/ref/settings-docker.xml dependency\
:resolve
COPY . .
RUN mvn -B -s /usr/share/maven/ref/settings-docker.xml package -DskipTests

FROM java:8-jdk-alpine AS production
RUN adduser -Dh /home/gordon gordon
WORKDIR /static
COPY --from=storefront /usr/src/atsea/app/react-app/build/ .
WORKDIR /app
COPY --from=appserver /usr/src/atsea/target/AtSea-0.0.1-SNAPSHOT.jar .
ENTRYPOINT ["java", "-jar", "/app/AtSea-0.0.1-SNAPSHOT.jar"]
CMD ["--spring.profiles.active=postgres"]
```

The first thing to note is that the `Dockerfile` above has three FROM instructions, each of these constitutes a distinct build stage, internally they are numbered from the top starting at 0, however we have also given each stage a friendly name. Stage 0 is called `storefront`, stage 1 is called `appserver`, stage 2 is called `production`

The storefront stage pulls the node:latest image which is over `600mb` in size. It sets the working directory copies in some app code and uses two RUN instructions to perform some npm magic. This adds three layers and considerable size. The result is an even bigger image containing lots of build stuff and not very much app code.

The `appserver` stage pulls the maven latest image which is over `700mb` in size, It adds four layers of content via two copy instructions and two run instructions. This produces another very large image with lots of build tools and very little actual production code.

The production stage starts by pulling the java:8-jdk, This image is approximately `150mb` - considerably smaller than the other two stages. It adds in a user, sets the working directory and copies in some app code from the image produced by the storefront stage. After that it sets a different working directory and copies in the application code from the image produced by the `appserver` stage. Finally it sets the main application for the image to run when it is started as a container.

The important things to note are that he `COPY --from` instructions only copy production related application binaries from the images built by the previous stages. They do not copy any other auxiliary resources used to build the application itself, including the code since those are not needed. Also note that the previous FROM/stages instructions create images as well, which can be listed with `docker image ls -a`

Swarm mode

At a high level orchestration is all about automating and simplifying the management of containerized applications at a scale. Things like automatically rescheduling containers when nodes break, scaling things up when demand increases and smoothly pushing updates and fixes into production environments. For the longest time orchestration like this was hard, tools like `Docker Swarm` and `Kubernetes` were available but they were complicated, then along came Docker 1.12 and the new native swarm mode, and overnight things changed. All the orchestration stuff got a whole lot easier.

Definition

Swarm mode brought a load of changes and improvements to the way we manage containers at scale. At the heart of those changes is native clustering of Docker hosts that is deeply integrated into the Docker platform. We are not talking about something like `Kubernetes` that is separate too requiring a highly skilled specialist to configure it on top of existing Docker infrastructure. The clustering here is about first class citizen in the Docker technology stack. And it is simple.

By default a standard installation of Docker will default to running in single engine mode, ensuring 100% backward compatibility with previous version of Docker. Putting Docker Engine into swarm mode gives you all of the latest orchestration goodness it just comes in at the price of some backward compatibility

Swarm

A swarm consist of one or more nodes. These can be physical servers, virtual machines or cloud instances, the only requirement is that all nodes in a swarm can communicate with each other over reliable networks. Nodes are then configured as managers or workers. Mangers look after the state of the cluster and are in charge of dispatching tasks/containers to workers. Workers accept tasks from mangers and execute them. Swarm nodes also heavily use and rely on `TLS` to encrypt communications, authenticate nodes and authorize roles, Automatic key rotation is also thrown in.

Locally when running docker swarm the node behavior of standalone physical servers or virtual machine is simulated by the docker engine. Therefore locally there is no real benefit of working with docker swarm, since the power of the swarm is in the distributed nature of it across many physical machines - each of which is a node. Another important thing to note is that a single node can run many tasks or containers inside of it, each node has its own docker engine and docker daemon and so on.

Tasks

Tasks in the context of a swarm we mean containers, so when we say managers dispatch tasks to workers we are saying they dispatch container workloads, you might also hear them referred to as replicas this might be confusing at this point so try and remember that tasks and replicas are words that mean containers.

Enabling

To enable docker swarm, one needs to run the following command on the docker host - `docker swarm init`. Docker host in single-engine will now convert and switch to swarm mode. It will also make the node the first manager of the Swarm. Additional nodes can then be joined to the swarm as workers and managers using the `docker swarm join` command.

```
# to switch the docker daemon host to a swarm mode instead of the default
   single node mode
docker swarm init
```

The following steps will put `mgr1` into swarm mode and initialize a new swarm. It will then join `wrk1` and `wrk2` and `wrk3` as worker nodes - automatically putting them int swarm mode. Finally it will add `mgr2` and `mgr3` as additional managers and switch them into swarm mode. At the end of the procedure all 6 nodes will be part of the same swarm and will all be operating in swarm mode.

The current single local host becomes the cluster's manager and is responsible for maintaining the swarm state. While you only have one physical machine, docker uses its own daemon and networking layers to act as if you are running separate nodes.

```
# to initialize mgr1 into swarm mode and create a new swarm, with explicit IP
    addresses
docker swarm init --advertise-addr 10.0.0.1:2377 --listen-addr 10.0.1:2377
Swarm initialized: current node (d21lyz...c79qzkx) is now a manager.
```

This command can be broken down as follows

- `docker swarm init` tells Docker daemon to initialize a new swarm and make this node the first manager. It also enables the swarm mode on the node.
- `advertise-addr` is the IP and port that other nodes should use to connect to this manager. The flag is optional but it gives you control over which IP gets used on nodes with multiple IPs. It also gives you the chance to specify an IP address that does not exit on the node such as a load balancing IP address.
- `listen-addr` lets you specify which IP and port you want to listen on for swarm traffic. This will usually match the address provided in `advertise-addr`, but is useful in situations where you want to restrict swarm to a particular IP on a system

```
docker swarm join-token worker
To add a manager to this swarm, run the following command: docker swarm join
   --token SWMTKN-1-0uahebax...c87tu8dx2c 10.0.0.1:2377

docker swarm join-token manager
To add a manager to this swarm, run the following command: docker swarm join
   --token SWMTKN-1-0uahebax...ue4hv6ps3p 10.0.0.1:2377
```

Notice that the commands to join a worker and a manager are identical apart from the join tokens (`SWMTKN` ending in either `c87tu8dx2c` or `ue4hv6ps3p`). This means that whether a node joins as a worker or a manager depends entirely on which token you use when joining it. These tokens should be protected, as these are all that is required to join a node to a swarm.

```
docker swarm join --token SWMTKN-1-0uahebax...c87tu8dx2c 10.0.0.1:2377
```

To add `wrk1` and join it to the swarm using the docker swarm join command with the correct token for a worker node (`c87tu8dx2c`). Note that one can keep retrieving tokens for a manager with the docker swarm join-token worker, this would produce a new unique token which can be used to add a new worker to the swarm

```
docker swarm join --token SWMTKN-1-0uahebax...ue4hv6ps3p 10.0.0.1:2377
```

To add `mgr1` and join in to the swarm using the docker swarm join command with the correct token for a manager (`ue4hv6ps3p`). Note that one can keep retrieving tokens for a manager with the docker swarm join-token manager, this would produce a new unique token which can be used to add a new manager to the swarm

```
# to list the nodes in the swarm
docker node ls
ID          HOSTNAME STATUS AVAILABILITY MANAGER STATUS
d21...qzkx mgr1     Ready  Active       Leader
d21...qzkx wrk1     Ready  Active
```

This command will show that `mgr1` is currently the only manager node in the swarm and the `wrk1` as the only worker the swarm, but that depends on how many of the swarm join-token worker/manager we run.

Looking at the manager status column in the list output, note that only one manager can be `Leader`, the other managers will be marked as `Reachable`. The worker nodes have no manager status which should be obvious as to why.

High availability Swarm managers have native support for high availability (H/A). This means that one or more can fail and the survivors will keep the swarm running. Technically speaking, swarm mode implements a form of active passive multi manager high availability. This means that although you might and should have multiple managers, only one of them is ever considered active. We call this active manager the leader. And the leader is the only one that will ever issues live commands against the swarm such as changing the configuration of the swarm or issuing tasks to workers. If a non active manager receives commands for the swarm it will proxy them across to the leader. The swarm uses and implementation of the Raft consensus algorithm to power the high availability and the following two best practices apply

1. Deploy an odd number of managers to avoid consensus collisions
2. Do not deploy too many managers (at best 3 or 5 is recommended)

Having an odd number of managers increases the change of reaching quorum and avoiding a split-brain. Fr example if you had 4 managers and the network partitioned you could be left with two mangers on each side of the partition. This is known as a split brain - each side knows there used to be 4 but can now only see 2. Neither side has any way of knowing if the two it can no longer see are still alive and which side holds the majority share (quorum). However if you had 3 or 5 managers and the same network partition occurred it would be impossible to have the same number of managers on both sides of the split. This means that one side would have a far better chance of knowing if it had more or less than the other side and achieving quorum. As with all consensus algorithms more participants means more time required to achieve consensus, it is like deciding where to eat - it is quicker and easier for 3 people to decide than it is for 33.

A final word of caution regarding manager HA. While it is obviously a good practice to spread your managers across availability zones within your network you need to make sure that the networks connecting them are reliable. Network partitions can be a pain. This means hosting your active production applications and infrastructure across multiple cloud providers such as AWS or Azure is a bit of a daydream.

Service

Services, at the highest level services are the way to run tasks on a swarm to run a task/container on a Swarm we wrap it in a service and deploy that service. Beneath the hood service are declarative way of setting the desired state on the cluster. For example

- set the number tasks/containers in the service
- set the image the containers in the service will use
- set the procedure for updating to newer version of the image

Services let us declare the desired state for an application service and feed that to Docker. For example assume that you have got an app that has a web front end. You have an image for the web service and

testing has shown that you will need 5 instance of the web service. You would translate the requirement into a service declaring the image the containers should use, and that service should always have 5 running tasks. To create a service use the `docker service create` command

```
docker service create --name web-fe -p 8080:8080 --replicas 5 nigelpoulton/
   pluralsight-docker-ci
```

The name flag tells docker what name to assign to the service, also docker is told to map port 8080 on every node in the swarm to 8080 inside of each container (task) in the service. Next the replicas flag to tell Docker that there should always be 5 tasks or containers running in the service. Finally Docker is told which image to use in this case it is an example custom image representing the app - `nigelpoulton/pluralsight/docker-ci`.

After executing the command the manager acting as a leader instantiated 5 tasks across the swarm, remember that managers also act as workers. Each worker or manager then pulled the image and started a container from it running on port 8080. The swarm leader also ensured a copy of the service desired state was replicated to every manager in the swarm.

All services are constantly monitored by the swarm, the swarm runs a reconciliation loop the constantly compares the actual state of the service to the desired state, if the two states match the world is a happy place, and no further actions is needed, if they do not match the swarm takes actions so that they do. Or in other words the swarm is constantly making sure that the actual state matches the desired state, on a constant loop.

As an example if one of the workers hosting one of the 5 containers tasks fails the actual state for the `web-fe` service will drop from 5 running tasks to 4, this will no longer match the desired state of 5, so Docker will start a new `web-fe` task to bring actual state back in line with the desired state. This behavior is very powerful and allows the service to self-heal in the event of node failures and the likes.

Listing service can be done with the command `docker service ls`. This will show all services, and their details

```
docker service ls
ID        NAME   MODE        REPLICAS IMAGE              PORTS
z7o...uw web-fe replicated 5/5      nigel...ci:latest *:8080->8080
```

The output shows a single running service as well as some basic information about the state. Among other things we can see that the name of the service and that 5 out of the 5 desired tasks/replicas/container instances are in running state, If one runs the command soon after deploying the service it might not show all tasks or replicas as running, this is probably because of the time it takes to pull the image on each node.

The `docker service ps` command can be used to see a list of tasks in a service and their state, this is similar to the PS command on Linux or Unix which shows information about the current processes running on the system.

```
docker service ps <service-name>

ID         NAME     IMAGE             NODE DESIRED CURRENT
817...f6z web-fe.1 nigelpoulton/... mgr2 Running Running 2 mins
a1d...mzn web-fe.2 nigelpoulton/... wrk1 Running Running 2 mins
cc0...ar0 web-fe.3 nigelpoulton/... wrk2 Running Running 2 mins
6f0...azu web-fe.4 nigelpoulton/... mgr3 Running Running 2 mins
dyl...p3e web-fe.5 nigelpoulton/... mgr1 Running Running 2 mins
```

For a more detailed information about a service use the docker service inspect command. This will show a more concise stack of information

```
docker servie inspect --pretty <service-name>

ID: z7ovearqmruwk0u2vc5o7ql0p
Name: web-fe
Service Mode: Replicated
Replicas: 5
Placement:
UpdateConfig:
Parallelism: 1
On failure: pause
Monitoring Period: 5s
Max failure ratio: 0
Update order: stop-first
RollbackConfig:
Parallelism: 1
On failure: pause
Monitoring Period: 5s
Max failure ratio: 0
Rollback order: stop-first
ContainerSpec:
Image: nigelpoulton/pluralsight-docker-ci:latest@sha256:7a6b01...d8d3d
Resources:
Endpoint Mode: vip
Ports:
PublishedPort = 8080
Protocol = tcp
TargetPort = 8080
PublishMode = ingress
```

The example above uses the –pretty flag to limit the output to the most interesting items printed in an easy to read format. Leaving off the –pretty flag will give a more verbose output.

Scaling Another powerful feature of services is the ability to easily scale them up and down. Let us assume business is booming and we are seeing double the amount of anticipated traffic hitting the `webfront` end. Fortunately scaling the service is as simple as running the docker service scale command

```
# scale the service from 5 to 10 replicas
docker service scale <service-name>=10
<service-name> scaled to 10
```

Running a docker service `ps` command will now show that the tasks i the service are balanced across all nodes in the swarm as evenly as possible. That means that given the fact that we have less worker and manager nodes, there is no way to run one replica per worker or manager, meaning that they will be distributed across the workers and mangers, therefore one worker might get 2 or even 3 tasks running on them, depending on how the docker engine and the swarm splits them up, it is not something that we have control over generally.

Behind the scenes swarm mode runs a scheduling algorithm that defaults to trying to balance tasks as evenly as possible across the nodes in the swarm. This amounts to running an equal number of tasks on each node without taking into consideration things like CPU load etc.

To reduce the number of replicas, one can run the same command again, doing `docker service scale <service-name>=5`, which will scale down the service back to 5, the initial desired state, this will remove

running tasks from the worker and managers, again no control over which exact instances are killed, which is fine since they are all clones of each other anyway

Removing To remove or delete a service is simple, as simple as running docker service rm . This will delete the service we have already deployed earlier. However a word of caution when using the docker service rm command as is, it deletes all tasks in the service without asking for confirmation, meaning that along with the service, all running tasks will be killed/stopped from running on the nodes in the swarm

Updating Pushing updates to deployed application is a fact of life. And for the longest time it has been really painful, we have lost more than enough weekends to major application updates. To demonstrate how this works, the example below is using the rolling update method, but any number of other methods can be employed, however docker does provide a rolling update method out of the box, `with the use of update-parallelism & update-delay`

```
# create an overlay network, an overlay network essentially creates a new
   layer 2 network that can be used to place
# containers on and these containers can all communicate with each other,
   this works even if the Docker hosts they are
# running on are on different underlying networks
docker network create -d overlay uber-net
```

```
# to list the available networks on the docker host, note that some of these
   might look familiar like the bridge, host,
# and none, also the last column shows what is the type of the network, where
    does it run - local implies that it is
# running on the local docker engine host, while swarm implies it is part of
   the swarm
$ docker network ls
NETWORK      ID              NAME    DRIVER SCOPE
490e2496e06b bridge          bridge  local
a0559dd7bb08 docker_gwbridge bridge  local
a856a8ad9930 host            host    local
1ailuc6rgcnr ingress         overlay swarm
be581cd6de9b none            null    local
43wfp6pzea47 uber-net        overlay swarm
```

```
# create the new service which is going to be used for demo purposes for
   rolling updates on the app
docker service create --name uber-svc --network uber-net -p 80:80 --replicas
   12 nigelpoulton/tu-demo:v1
```

```
# list the current services, which are active
docker service ls
ID           NAME        REPLICAS            IMAGE
dhbtgvqrg2q4 uber-svc    12/12               nigelpoulton/tu-demo:v1

# list all the tasks/containers/replicas for the service
docker service ps uber-svc
ID           NAME        IMAGE               NODE DESIRED CURRENT STATE
0v...7e5     uber-svc.1  nigelpoulton/...:v1 wrk3 Running Running 1 min
bh...wa0     uber-svc.2  nigelpoulton/...:v1 wrk2 Running Running 1 min
23...u97     uber-svc.3  nigelpoulton/...:v1 wrk2 Running Running 1 min
```

```
82...5y1     uber-svc.4  nigelpoulton/...:v1 mgr2 Running Running 1 min
c3...gny     uber-svc.5  nigelpoulton/...:v1 wrk3 Running Running 1 min
e6...3u0     uber-svc.6  nigelpoulton/...:v1 wrk1 Running Running 1 min
78...r7z     uber-svc.7  nigelpoulton/...:v1 wrk1 Running Running 1 min
2m...kdz     uber-svc.8  nigelpoulton/...:v1 mgr3 Running Running 1 min
b9...k7w     uber-svc.9  nigelpoulton/...:v1 mgr3 Running Running 1 min
ag...v16     uber-svc.10 nigelpoulton/...:v1 mgr2 Running Running 1 min
e6...dfk     uber-svc.11 nigelpoulton/...:v1 mgr1 Running Running 1 min
e2...k1j     uber-svc.12 nigelpoulton/...:v1 mgr1 Running Running 1 min
```

```
# update the service with a new image, with a 20 second delay at a time, and
   2 containers/tasks per update
docker service update --image nigelpoulton/tu-demo:v2 --update-parallelism 2
   --update-delay 20s uber-svc
```

What does this command really do ? Well the docker service update lets us make updates to a running service, by updating the service's desired state, this time we gave it a new image tag `v2`, instead of `v1`. And used the update-parallelism, and the update-delay. What this does is that the new image was pushed to `2 tasks at a time with a 20 second cool off` period in between each pair. Running the `docker service ps` against the service some of the tasks in the service are at `v2` image, while some at `v1`.

```
docker service ps uber-svc
ID        NAME          IMAGE       NODE DESIRED  CURRENT  STATE
7z...nys uber-svc.1    nigel...v2 mgr2 Running  Running  13 secs
0v...7e5 \_uber-svc.1 nigel...v1 wrk3 Shutdown Shutdown 13 secs
bh...wa0 uber-svc.2    nigel...v1 wrk2 Running  Running  1  min
e3...gr2 uber-svc.3    nigel...v2 wrk2 Running  Running  13 secs
23...u97 \_uber-svc.3 nigel...v1 wrk2 Shutdown Shutdown 13 secs
82...5y1 uber-svc.4    nigel...v1 mgr2 Running  Running  1  min
c3...gny uber-svc.5    nigel...v1 wrk3 Running  Running  1  min
e6...3u0 uber-svc.6    nigel...v1 wrk1 Running  Running  1  min
78...r7z uber-svc.7    nigel...v1 wrk1 Running  Running  1  min
2m...kdz uber-svc.8    nigel...v1 mgr3 Running  Running  1  min
b9...k7w uber-svc.9    nigel...v1 mgr3 Running  Running  1  min
ag...v16 uber-svc.10   nigel...v1 mgr2 Running  Running  1  min
e6...dfk uber-svc.11   nigel...v1 mgr1 Running  Running  1  min
e2...k1j uber-svc.12   nigel...v1 mgr1 Running  Running  1  min
```

Strategies As we have seen above, this is one of the many deployment strategies which we can take in order to update our service, however there are a few other. Not all deployment strategies are listed below, however some of the more well known ones are, along with their pros and cons depending on the circumstances

- Recreate Target - stops the old version entirely before deploying the new one, all instances of the app are replaced simultaneously. This is simple to implement, and guarantees no coexistence of old and new versions, no resource hogging. However there is a downtime during deployment process, clients experience a complete service outage. This is suitable for non critical apps where downtime is acceptable
- Rolling Update - Gradually replaces old instances with new ones, one or a few at a time, continues until all instances are updated. It has minimal downtime, allows monitoring of new instances as they are deployed and put into action, reduces deployment risk compared to an all-at-once strategy. However old and new versions coexist during the deployment which may cause inconsistencies, it also requires careful planning for compatibility between versions. Common for stateless or backward compatible apps

- Blue-Green - Deploys the new version (green) alongside the current version (blue) in parallel, traffic is switched to the new version once it is verified to be working, the old version remains available as a fallback. The cool thing here is that there is no downtime, seamless transition for users, easy and fast rollback, since the old version is still active and running. However resource intensive since both the versions are working or active at the same time, complex traffic routing needs to be setup in order to make this viable and transparent to the end user.
- Canary - deploys the new version to a small subset of users or servers first, gradually increases the percentage of traffic routed to the new version, full roll-out occurs after successful validation. Allows controlled testing in production with minimal risk, issues can be caught early and affect fewer users, rollbacks are easier during initial stages. Requires dynamic traffic routing and monitoring systems, and takes longer to fully deploy compared to the other strategies
- Feature toggle - deploys the new version with features disabled, features are enabled incrementally via configuration toggles without redeploying the code. There is a minimal risk, features can be turned off instantly if issues occur, allows gradual roll-out of new functionality, supports A/B testing and phased roll-outs. However in increases code complexity, as well as there is a risk of stale toggles cluttering the codebase i.e obsolete features.
- Shadow - deploys the new version alongside the old version but does not expose it to users, the new version processes mirrored traffic for testing purposes. No impact on live users, allows the real-world testing of the new version, High resource usage, in effect you have two production environments, and does not test user interactions with the new versions, rather testers or QA do that, which is not really the same
- A/B Testing - deploys multiple versions at the same time - the old and the new. Routes a subset of users to the new version while the rest continue using the old version. Collects metrics to compare performance and user experience. Provides direct insights into user preferences, and performance of different versions, allows for controlled exposure to the new version. However it requires robust traffic routing and user segmentation, risk of user confusion if versions behave differently. Requires a robust traffic routing and user segmentation. Risk of user confusion if version behave differently

Networking

In the real world, containers have to be able to communicate with each other reliably and securely even when they are on different hosts, on different networks. This is where overlay networking comes in to play. It allows you to create a flat secure layer 2 network spanning multiple hosts, that containers can connect to. Containers on this network can then communicate directly. Behind the scenes the docker networking stack is comprised of `libnetwork` and drivers. `Libnetwork` is the canonical implementation of the container network model (`CNM`) and drivers are pluggable components that implement different networking technologies and topologies.

Definition

In 2025 Docker, Inc acquired container networking startup Socket Plane. Two of the reasons behind the acquisition were to bring real networking to Docker and to make container networking simple.

Security

Good security is all about the layers, and docker has lots of layers, it supports all the major Linux security technologies as well as having a lot of its own, and most them are simple and easy to configure. Docker on Linux leverages most of the common Linux security technologies, these include `namespaces, control groups, capabilities, mandatory access control and seccomp`. For each docker implements sensible

defaults for a seamless and moderately secure out of the box experience. However it also allows you to customize each one to your liking.

The Docker platform itself offers some excellent native security technologies, and one the best things about these is that they are amazingly simple to use.

- Docker Swarm mode is secure by default, you get all the following with zero configuration required - cryptographic node ID, mutual authentication, automatic CA configuration, automatic certificate rotation, encrypted cluster store, encrypted networks.
- Docker content trust lets yo sign your images and verify the integrity and publisher of images you pull
- Docker Security Scanning analyses Docker images, detects known vulnerabilities and provides you with a detailed report.
- Docker secrets - makes secrets first class citizens in the Docker ecosystem, they get stored in the encrypted cluster store, encrypted in flight when delivered to containers, and stored in in-memory filesystems when in use

Definition

Linux security technologies, what are those ? All good container platforms should use `namespaces` and cgroups to build containers. The best container platforms will also integrate with other Linux security technologies such as `capabilities, mandatory access control systems and seccomp`

Namespaces

Kernel `namespaces` are at the very heart of containers, they let us slice up an operating system so that it looks and feels like multiple isolate operating systems, this lets us do really cool things like run multiple web servers on the same OS without having port conflicts, it also lets us run multiple apps on the same OS without them fighting over shared configuration files and shared libraries.

- You can run multiple web servers, each requiring port 443 on a single OS. To do this you jut run each web server app inside its own network `namespace`.This works because each network `namespace` gets its own IP address and full range of ports.
- You can run multiple apps each requiring their own particular version of a shared library or configuration file. To do this you run each app inside of its own mount namespace. This works because each mount `namespace` can have its own isolated copy of any directory on the system (/etc, /var, /dev etc.)

Docker on Linux currently utilizes the following kernel types of `namespaces`:

- Process id (pid)
- Network (net)
- Filesystem mount (mnt)
- Inter-process communication (ipc)
- User (user)
- UTS

`A docker container is an organized collection of namespaces.` For example every container is made up of its own `pid, net, mnt, ipc, uts and potentially user namespaces`. The organized collection of these is what we call a container.

- PID - `namespace` - docker uses the pid `namespace` to provide isolated process trees for each container, every container gets it own process tree meaning that every container can have its own PID 1, PID `namespaces` also mean that a container can not have see or access to the process tree of other containers or host processes it's running on

- Network - docker uses the net `namespace` to provide each container its own isolated network stack, this stack includes interfaces, IP addresses, port ranges, and routing tables, for example every container gets its own `eth0` interface with its own unique IP and range of ports
- Mount - every container gets its own unique isolate `root/filetsystem`. This means that every container can have its own `/etc, /var, /dev etc`. Processes inside of a container cannot access the mount `namesapce` of the Linux host, or other containers, they can only see and access their own isolate mount `namespace`.
- Inter process communication - docker uses the IPC `namespace` for shared memory access within a container. It also isolates the container from shared memory outside of the container.
- User - docker lets you use the user `namespace` to map users inside of a container to a different user on the Linux host, a common example would be mapping the root user of a container to a non-root user on the Linux host, user `namespaces` are quite new to Docker and are currently optional
- UTS - docker uses the UTS `namespace` to provide each container with its own `hostname`.

Control groups

If `namespaces` are about isolation, the control groups are about setting limits, think of containers as similar to rooms in a hotel, yes each room is isolated but each room also shares a common set of resources, things like water supply electricity supply, shared swimming pool shared gym shared breakfast bar etc. Control groups let us set limits on containers so that no single container can use all of the resources on the host

In the real world, containers are isolated from each other but all share a common set of resources - things like the host processor, ram and disk. Control groups let us set limits on each of these so that a single container can not take ownership of the whole host system resources

Capabilities

It is a bad idea to run containers as root - root is all powerful and therefore very dangerous but it is a pain in the backside running containers as non-root, non root is so powerless it is practically useless. What we need is a technology that lets us pick and choose which root powers our containers need in in order to run. Under the hood the Linux root account is made up of a long list of capabilities, some these include

- `CAP_SHOWN` - lets you change file ownership
- `CAP_NET_BIND_SERVICE` - lets you bind a socket to low numbered network ports
- `CAP_SETUPID` - lets you elevate the privilege level of a process
- `CAP_SYS_BOOT` - lets you reboot the system.

Docker works with capabilities so that you can run containers as root, but strip out the root capabilities that you do not need. For example if the only root privilege your container needs is the ability to bind to low numbered network ports yo should start a container and drop all root capabilities then add back the `CAP_NET_BIND_SERVICE` capability. Docker also imposes restrictions so that containers cannot re-add the removed capabilities back.

Mandatory access control systems

Docker works with a major Linux MAC technologies such as `AppArmor` and `SELinux`. Depending on your Linux distribution, Docker applies a default `AppArmor` profile to all new containers, According to the Docker documentation, this default profile is moderately protective while providing wide application compatibility, Docker also lets you start containers without a policy applied as well as giving you the ability to customize policies to meet your specific requirements.

Seccomp

Docker uses `seccomp`, if filter mode, to limit the `syscalls` a container can make to the host kernel. As per the docker security philosophy all new containers get a default `seccomp` profile configured with sensible defaults, this is intended to provide a moderate security without impacting application compatibility. As always you can customize `seccomp` profiles and you can pass a flag to docker so that containers can be started without a `seccomp` profile,

Conclusion

```
Docker supports most of the important Linux security technologies and ships, with sensible
 defaults that add security but are not too restrictive. Some of these technologies can be
 complicated to customize as they can require deep dive knowledge of how they work and how
 the Linux kernel works, Hopefully they will get simpler to configure in the future but
for now the default configurations that ship with Docker are a good place to start.
```

Swarm security

Swarm mod is the future of Docker, since it lets you cluster multiple docker hosts (nodes) and deploy your app in a declarative way, Every swarm is comprised of managers and workers, that can be Linux or Windows, Managers make up the control plane of the cluster and are responsible for configuring the cluster and dispatching work to it. Workers are the nodes that run your application code as containers. As expected, swarm mode includes many security features that are enabled out of the box with a sensible defaults. These include

- Cryptographic node IDs
- mutual authentication via TLS
- secure join tokens
- CA configuration with automatic certificate rotation
- Encrypted cluster store (config DB)
- Encrypted networks

The moment the `docker swarm init` command is executed, the default security configurations take place out of the box, with sensible defaults as already mentioned, The swarm has been given a cryptographic ID, and `mgr1` has issues itself with a client certificate that identifies it as a manager in the Swarm. Certificate rotation has been configured with the default value of 90 days and a cluster configuration database has been configured and encrypted. A set of secure tokens have also been created so that new managers and new workers can be joined to the Swarm, and all of this with a single command.

Swarm join tokens

The only thing that is needed to join managers and workers to an existing swarm is the relevant join token. For this reason it is vital that you keep your tokens safe, no posting them on public repositories. Every swarm maintains two distinct join tokens - one for joining new managers and one for joining new workers. It is worth understanding the format of the Swarm join token, every join token is comprised of 4 distinct fields separated by dashes, the format looks like that - `PREFIX - VERSION - SWARM ID - TOKEN`.

- The prefix is always - `SWMTKN`.
- The version fields indicates the version of the Swarm
- The Swarm ID field is a hash of the swarm's certificate.
- The token portion is the part that determines if the token can be used to join node as manager or worker.

If you suspect that either of your join tokens has been compromised you can revoke them and issue new ones with a single command, the following example revokes the existing manager join token and issues a new one, `docker swarm join-token --rotate manager`. Notice that the only difference between the old and new join tokens is going to be in the last field, the swarm id remains the same. Join tokens are stored in the cluster config database, which is encrypted by default.

TLS and mutual authentication

Every manager and worker that joins a Swarm is issued a client certificate, this certificate is used for mutual authentication, it identifies the node, which Swarm the node is a member of, and role the node performs in the Swarm (manger or worker). On a Linux host, one can inspect a node's client certificate with the following command - `sudo openssl x509 -in /var/lib/docker/swarm/certificates/swarm-node.crt -text`. That will decrypt the certificate, and display the contents of it in human readable format.

```
Certificate:
Data:
Version: 3 (0x2)
Serial Number:
80:2c:a7:b1:28...a8:af:89:a1:2a:51:89
Signature Algorithm: ecdsa-with-SHA256
Issuer: CN=swarm-ca
Validity
Not Before: Jul 19 07:56:00 2017 GMT
Not After : Oct 17 08:56:00 2017 GMT
Subject: O=mfbkgjm2tlametbnfqt2zid8x, OU=swarm-manager,
CN=7xamk8w3hz9q5kgr7xyge662z
Subject Public Key Info:
<SNIP>
```

The Subject data in the output above uses the standard `O, OU, and CN` fields to specify the Swarm ID, the node's role and the node ID.

- The organization `O` field stores the Swarm ID
- The organizational unit `OU` field stores the nodes role in the Swarm
- The canonical name `CN` field stores the nodes crypto ID

Some of the certificate properties can be configured, like for example the rotation period with the following command `docker warm update --cert-expiry 720h`, this will set the expiration of the certificate to 30 days instead of the default 90. Swarm allows nodes to renew certificates early, before they expire, so that not all nodes in the Swarm try and update their certificates at the same time. You can configure an external CA (signing authority) when creating a Swarm by passing the `--external-ca` flag to the `docker swarm init` command. The new docker swarm ca sub-command can also be used to manage the CA related configuration. Run the command with the `--help` flag to see the list of things it can do

Cluster store

The cluster store is the brains of a Swarm and is the place where cluster configuration and state are stored, the store is currently based on an implementation of etcd and is automatically configured to replicate itself to all managers in the Swarm, it is also encrypted by default. The cluster store is becoming a critical component of many docker platform technologies - for example Docker networking and Docker secrets both use the cluster store. This is one of the reasons that

Signing images

Docker content trust, makes it simple and easy to verify the integrity and the publisher of images that you download, This is especially important when pulling images over untrusted networks such as the internet. At a high level the docker content trust allows developers to sign their images when they are pushed to docker hub or docker trusted registry, ti will also automatically verify images when they are pulled. Docker content trust can also provide important context, this includes things like whether an image has been superseded by a newer version and is therefore stale.

Secrets

Many applications need secrets, things like passwords, certificates, ssh keys and more. Docker introduced something called docker secrets, effectively making secrets first class citizens in the docker ecosystem. For example there is a whole new docker secret sub command dedicated to managing secrets. There is also page for creating and managing secrets in the `docker universal control plane ui`. Behind the scenes secrets are encrypted at rest, encrypted in-flight, mounted in memory `filesystems` and only available to services/containers that have been explicitly granted access to them, it is quite a comprehensive end to end solution

So how does this process work, imagine we have 3 worker, each of which is running two different images - red and blue. The red and the blue each have two replicas/tasks/containers active/running in the service. One of the workers is running one instance of the red and one instance of the blue (due to the fact that we have 3 workers, but 4 tasks in total, one of the workers is bound to have 2 tasks/containers running on it, in this case one of each kind - red and blue). Now the challenge here is how to distribute the secret to the workers, without leaking the information to tasks/containers that must not see it.

1. The secret is created and posted to the Swarm
2. It gets stored in the encrypted cluster store
3. The blue service is created and the secret is attached to it
4. The secret is encrypted in flight while it is delivered to the containers in the blue service
5. The secret is mounted into the containers of the blue service as an unencrypted file /run/secrets/. This is an in memory tmpfs filesystem.
6. Once the container service task completes the in memory filesystem is torn down
7. The red containers service cannot access the secret.