Contents

Chapter 1: Microservices

Microservice is a small, loosely coupled distributed service. Microservice architecture evolved as a solution to the scalability, independently deployable, and innovation challenges with Monolithic architecture (Monolithic applications are typically huge – more than 100,000 lines of code). It allows you to take a large application and decompose or break it into easily manageable small components with narrowly defined responsibilities. It is considered the building block of modern applications. Microservices can be written in a variety of programming languages, and frameworks, and each service act as a mini-application on its own. Microservice can be considered as the subset of SOA (Service Oriented Architecture).

1.1 Reasons for Using Microservice

In monolithic applications, there are a few challenges:

- For a large application, it is difficult to understand the complexity and make code changes fast and correctly, sometimes it becomes hard to manage the code
- Applications need extensive manual testing to ensure the impact of changes
- An application typically shares a common relational database to support the whole application
- For small changes, the whole application needs to be built and deployed
- The heavy application slows down start-up time

1.2 Benefits of Microservices

1. **Small Modules –** The application is broken into smaller modules that are easy for developers to code and maintain.
2. **Easier Process Adaption –** By using microservices, new Technology & Process Adaption becomes easier. You can try new technologies with the newer microservices that we use.
3. **Independent scaling –** Each microservice can scale independently via X-axis scaling (cloning with more CPU or memory) and Z-axis scaling (sharding), and Y-axis scaling (functional decomposition) based on their needs.
4. **Removes dependency –** Microservice eliminates long-term commitment to any single technology stack.
5. **Unaffected –** Large applications remain largely unaffected by the failure of a single module.
6. **DURS –** Each service can be independently DURS (deployed, updated, replaced, and scaled).
7. **Increased Security: –**Microservices enable data separation. Each service has its own database, making it harder for hackers to compromise your application.
8. **Open Standards: –**APIs enable developers to build their microservices using the programming language and technology they prefer.

1.3 Characteristics of Microservices

- The application will be divided into micro-components
- Each service has a separate database layer, independent codebase, and CI/CD tooling sets
- Can also use different languages, frameworks, and technologies
- Well-understood Distribution Transaction Management
- Presents API and is a decentralized app
- Easy routing process
- Robust and failure-resistant
- Decentralized operations
- Stateless and stateful services
- Designed for business
- Each service is independent
- Autonomous and specialized
- Each Service can implement an independent security mechanism

1.4 Microservice Pros and Cons

Pros

- Can independently develop and deploy services
- Dynamically scalable and quickly functioning
- Integration with third-party dependencies
- Has an independent manageable deployment module
- Different services may use different languages

Cons

- Multiple services mean multiple resources (difficult to handle)
- Different services using different languages makes testing difficult
- Debugging issues
- Increase in effort while handling it
- Challenges in deployment
- Communication between services is not easy

Chapter 2: Introduction to Dockers Containers

Containers have a long and storied history in computing. Unlike hypervisor virtualization, where one or more independent machines run virtually on physical hardware via an intermediation layer, containers instead run in user space on top of an operating system's kernel. As a result, container virtualization is often called operating system-level virtualization. Container technology allows multiple isolated user space instances to be run on a single host. As a result of their status as guests of the operating system, containers are some-times seen as less flexible: they can generally only run the same or a similar guest operating system as the underlying host.

For example, you can run Red Hat Enterprise Linux on an Ubuntu server, but you cannot run Microsoft Windows on top of an Ubuntu server. Containers have also been seen as less secure than the full isolation of hypervisor virtualization. Countering this argument is that lightweight containers lack the larger attack surface of the full operating system needed by a virtual machine combined with the potential exposures of the hypervisor layer itself. Despite these limitations, containers have been deployed in a variety of use cases. They are popular for hyperscale deployments of multi-tenant services, for lightweight sandboxing, and, despite concerns about their security, as process isolation environments. Indeed, one of the more common examples of a container is a chroot jail, which creates an isolated directory environment for running processes.

In computing, a container is an encapsulated process that includes the required runtime dependencies for the program to run. In a container, application-specific libraries are independent of the host operating system libraries. Libraries and functions that are not specific to the containerized applications are provided by the operating system and kernel. The provided libraries and functions help to ensure that the container remains compact, and that it can quickly execute and stop as needed.

A container engine creates a union file system by merging container image layers. Because container image layers are immutable, a container engine further adds a thin writable layer for runtime file modifications. Containers are ephemeral by default, which means that the container engine removes the writable layer when you remove the container

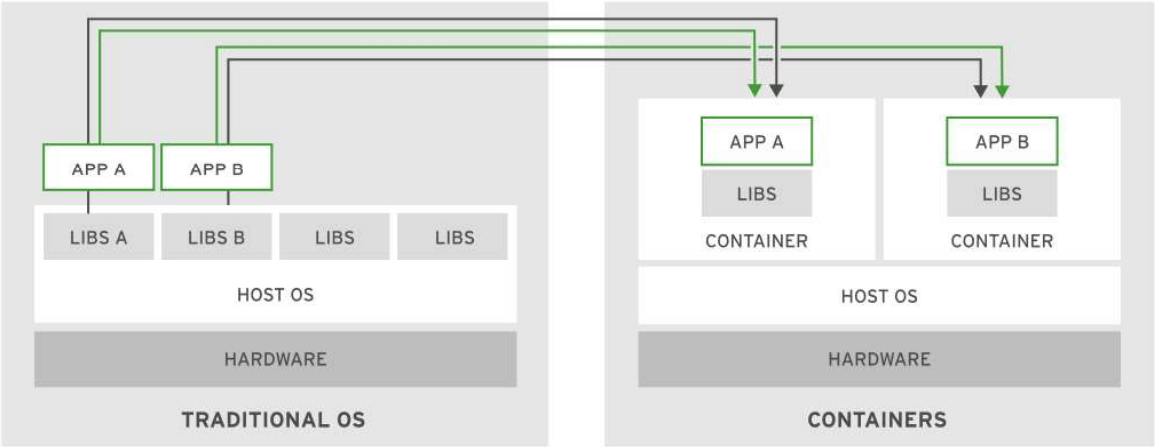


FIGURE:1 TRADITIONAL OS v/s CONTAINERS

Attackers, if they breach the running process in the jail, then find themselves trapped in this environment and unable to further compromise a host. More recent container technologies have included OpenVZ, Solaris Zones, and Linux containers like lxc. Using these more recent

technologies, containers can now look like full-blown hosts in their own right rather than just execution environments.

In Docker's case, having modern Linux kernel features, such as control groups and namespaces, means that containers can have strong isolation, their own network and storage stacks, as well as resource management capabilities to allow friendly co-existence of multiple containers on a host. Containers are generally considered lean technology because they require limited overhead. Unlike traditional virtualization or paravirtualization technologies, they do not require an emulation layer or a hypervisor layer to run and instead use the operating system's normal system call interface. This reduces the overhead required to run containers and can allow a greater density of containers to run on a host. Despite their history containers haven't achieved large-scale adoption. A large part of this can be laid at the feet of their complexity: containers can be complex, hard to set up, and difficult to manage and automate. Docker aims to change that.

2.1 Images versus Instances:

Containers can be split into two similar but distinct ideas: container images and container instances. A container image contains effectively immutable data that defines an application and its libraries. Container images can be used to create container instances, which are executable versions of the image that include references to networking, disks, and other runtime necessities. A single container image can be used multiple times to create many distinct container instances. These replicas can even be split across multiple hosts. The application within a container is independent of the host environment.

2.2 Virtual Machines Vs Containers:

Virtual machines and containers use different software for management and functionality. Hypervisors, such as KVM, Xen, VMware, and Hyper-V, are applications that provide the virtualization functionality for VMs. The container equivalent of a hypervisor is a container engine, such as Docker.

	Virtual machines	Containers
Machine-level functionality	Hypervisor	Container engine
Management	VM management interface	Container engine or orchestration software
Virtualization level	Fully virtualized environment	Only relevant parts
Size	Measured in gigabytes	Measured in megabytes
Portability	Generally only same hypervisor	Any OCI-compliant engine

FIGURE2: Virtual Machines v/s Containers

2.3 Deployment at Scale:

Both containers and VMs can work well at various scales. Because a container requires considerably fewer resources than a VM, containers have performance and resource benefits at a

larger scale. A common method in large-scale environments is to use containers that run inside VMs. This configuration takes advantage of the strong points in each technology.

2.4 Development for Containers:

Containerization provides many advantages for the development process, such as easing testing and deployment, providing additional tools for stability and security, and creating flexibility with multi-container functionality.

2.5 Testing and Workflows:

One of the greatest advantages for developers in using containerization is the ability to scale. A developer can write software and test locally, and then deploy the finished application to a cloud server or a dedicated cluster with few or no changes. This workflow is especially useful when creating microservices, which are small and ephemeral containers that are designed to spin up and down as needed. Additionally, developers that use containers can take advantage of Continuous Integration/Continuous Development (CI/CD) pipelines to deploy containers to various environments. RHOCP offers various integration features with CI/CD pipelines and workflows in mind

2.6 Stability:

As mentioned earlier, container images are a stable target for developers. Software applications require specific versions of libraries to be available for deployment, which can result in dependency issues or specific OS requirements. Because libraries are included in the container image, a developer can be confident of no dependency issues in a deployment. Having the libraries integrated within the container removes variability between testing and production environments. For example, a container with a specific version of Python ensures that the same version of Python is used in every testing or deployment environment.

Chapter 3: Dockers Containers:

Docker is an open-source engine that automates the deployment of applications into containers. It was written by the team at Docker, Inc (formerly dotCloud Inc, an early player in the Platform-

as-a-Service (PAAS) market), and released by them under the Apache 2.0 license. So, what is special about Docker? Docker adds an application deployment engine on top of a virtualized container execution environment. It is designed to provide a lightweight and fast environment in which to run your code as well as an efficient workflow to get that code from your laptop to your test environment and then into production. Docker is incredibly simple. Indeed, you can get started with Docker on a minimal host running nothing but a compatible Linux kernel and a Docker binary. Docker's mission is to provide:

3.1 An easy and lightweight way to model reality:

Docker is fast. You can Dockerize your application in minutes. Docker relies on a copy-on-write model so that making changes to your application is also incredibly fast: only what you want to change gets changed. You can then create containers running your applications.

Most Docker containers take less thana second to launch Removing the overhead of the hypervisor also means containers are highly performant and you can pack more of them into your hosts and make the best possible use of your resources.

3.2 Fast, efficient development life cycle:

Docker aims to reduce the cycle time between code being written and code be ingested, deployed, and used. It aims to make your applications portable, easy to build, and easy to collaborate on.

3.3 Docker components:

Let us look at the core components that compose Docker:

• The Docker client and server, also called the Docker Engine.

• Docker Images

• Registries

• Docker Containers

3.4 Docker client and server:

Docker is a client-server application. The Docker client talks to the Docker server or daemon, which, in turn, does all the work. You will also sometime see the Docker daemon called the Docker Engine. Dockerships with a command line client binary, docker, as well as a full RESTful API to interact with the daemon. You can run the Docker daemon and client on the same host or connect your local Docker client to a remote daemon running on another host. You can see Docker's architecture depicted here.

3.5 Docker images:

Images are the building blocks of the Docker world. You launch your containers from images. Images are the "build" part of Docker's life cycle. They are a layered format, using Union file systems, that are built step-by-step using a series of instructions.

For example:

* Add a file

* Run a command

* Open a port

You can consider images to be the" source code" for your containers. They are highly portable and can be shared, stored, and updated. In the book, we will learn how to use existing images as well as build our own images.

3.6 Registries:

Docker stores the images you build in registries. There are two types of registries: public and private. Docker, Inc., operates the public registry for images, called the Docker Hub. You can create an account on the Docker Hub and use it to share and store your own images.

The Docker Hub also contains, at last count, over 10,000 images that other people have built and shared. Want a Docker image for a Nginx web server, the Asterisk open source PABX system, or a MySQL database? All of these are available, along with a whole lot more.

You can also store images that you want to keep private on the Docker Hub. These images might include source code or other proprietary information you want to keep secure or only share with other members of your team or organization. You can also run your own private registry. This allows you to store images behind your firewall, which may be a requirement for some organizations.

3.7 Containers:

Docker helps you build and deploy containers inside of which you can package your applications and services. As we have just learned, containers are launched from images and can contain one or more running processes. You can think about images as the building or packing aspect of Docker and the containers as the running or execution aspect of Docker.

A Docker container is:

- An image formats.
- A set of standard operations.
- An execution environment.

Docker borrows the concept of the standard shipping container, used to transport goods globally, as a model for its containers. But instead of shipping goods, Docker containers ship software. Each container contains a software image -- its 'cargo' -- and, like its physical counterpart, allows a set of operations to be performed. For example, it can be created, started, stopped, restarted, and destroyed.

Like a shipping container, Docker does not care about the contents of the container when performing these actions; for example, whether a container is a web server, a database, or an application server. Each container is loaded the same as any other container. Docker also does not care where you ship your container: you can build on your laptop, upload to a registry, then download to a physical or virtual server, test, deploy to a cluster of a dozen Amazon EC2 hosts,

and run like a normal shipping container, it is interchangeable, stackable, portable, and as generic as possible. With Docker, we can quickly build an application server, a message bus, a utility appliance, a CI test bed for an application, or one of a thousand other possible ap-plications, services, and tools. It can build local, self-contained test environments or replicate complex application stacks for production or development purposes. The possible use cases are endless.

3.8 What can you use Docker for?

So why should you care about Docker or containers in general? We have discussed briefly the isolation that containers provide; as a result, they make excellent sand-boxes for a variety of testing purposes.

Additionally, because of their 'standard' nature, they also make excellent building blocks for services. Some of the examples of Docker running out in the wild include:

Helping make your local development and build workflow faster, more efficient, and more lightweight. Local developers can build, run, and share Docker containers. Containers can be built in development and promoted to testing environments and, in turn, to production.

1. Running stand-alone services and applications consistently across multiple environments, a concept especially useful in service-oriented architectures and deployments that rely heavily on micro-services.
2. Using Docker to create isolated instances to run tests like, for example, those launched by a Continuous Integration (CI) suite like Jenkins CI.
3. Building and testing complex applications and architectures on a local host prior to deployment into a production environment.
4. Building a multi-user Platform-as-a-Service (PAAS) infrastructure.
5. Providing lightweight stand-alone sandbox environments for developing, testing, and teaching technologies, such as the Unix shell or a programming language.
6. Software as a Service applications.
7. Highly performant, hyperscale deployments of hosts.

3.9 Docker with configuration management:

Since Docker was announced, there have been a lot of discussions about where Docker it's with configuration management tools like Puppet and Chef. Docker includes an image-building and image-management solution. One of the drivers for modern configuration management tools was the response to the "golden im-age" model. With golden images, you end up with massive and unmanageable image sprawl: large numbers of (deployed) complex images in varying states of versioning. You create randomness and exacerbate entropy in your environment as your image use grows. Images also tend to be heavy and unwieldy. This often forces manual change or layers of deviation and unmanaged configuration on top of images because the underlying images lack appropriate flexibility.

Introduction includes an image-building and image-management solution. One of the drivers for modern configuration management tools was the response to the "golden im-age" model. With golden images, you end up with massive and unmanageable image sprawl: large numbers of

(deployed) complex images in varying states of versioning. You create randomness and exacerbate entropy in your environment as your image use grows.

Images also tend to be heavy and unwieldy. This often forces manual change or layers of deviation and unmanaged configuration on top of images, because the underlying images lack appropriate flexibility. Compared to traditional image models, Docker is a lot more lightweight: images are layered, and you can quickly iterate on them. There is some legitimate argument to suggest that these attributes alleviate many of the management problems traditional images present.

It is not immediately clear, though, that this alleviation represents the ability to totally replace or supplant configuration management tools. There is amazing power and control to be gained through the idempotence and introspection that configuration management tools can provide. Docker itself still needs to be installed, managed, and deployed on a host. That host also needs to be managed. In turn, Docker containers may need to be orchestrated, managed, and deployed, often in conjunction with external services and tools, which are all capabilities that configuration management tools are excellent in providing. It is also apparent that Docker represents (or, perhaps more accurately, encourages some different characteristics and architecture for hosts, applications, and services: they can be short-lived, immutable, disposable, and service oriented.

These behaviours do not lend themselves or resonate strongly with the need for configuration management tools. With these behaviours, you are rarely concerned with long-term management of state, entropy is less of a concern because containers rarely live long enough for it to be, and the recreation of state may often be cheaper than the remediation of state. Not all infrastructure can be represented with these behaviours, however.

Docker's ideal workloads will likely exist alongside more traditional infrastructure deployment for a little while. The long-lived host, perhaps also the host that needs to run on physical hardware, still has a role in many organizations. As a result of these diverse management needs, combined with the need to manage Docker it-self, both Docker and configuration management tools are likely to be deployed in the majority of organizations

Chapter 4: Install Docker Engine:

4.1 Supported platforms

Docker Engine is available on a variety of Linux distros, macOS, and Windows 10 through Docker Desktop, and as a static binary installation. Find your preferred operating system below.

4.2 Install Docker Engine on CentOS

To get started with Docker Engine on CentOS, make sure you meet the prerequisites, then install Docker.

Prerequisites??

OS requirements

To install Docker Engine, you need a maintained version of one of the following CentOS versions:

* CentOS 7

* CentOS 8 (stream)

* CentOS 9 (stream)

* Archived versions aren't supported or tested.

* The centos-extras repository must be enabled. This repository is enabled by default, but if you have disabled it, you need to re-enable it.

* The overlay2 storage driver is recommended.

* Uninstall old versions

* Older versions of Docker went by the names of docker or docker-engine. Uninstall any such older versions before attempting to install a new version, along with associated dependencies:

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

It's OK if yum reports that none of these packages are installed.

Images, containers, volumes, and networks stored in /var/lib/docker/ aren't automatically removed when you uninstall Docker.

4.3 Installation methods:

You can install Docker Engine in different ways, depending on your needs:

* You can set up Docker's repositories and install from them, for ease of installation and upgrade tasks. This is the recommended approach.

* You can download the RPM package and install it manually and manage upgrades completely manually. This is useful in situations such as installing Docker on air-gapped systems with no access to the internet.

* In testing and development environments, you can use automated convenience scripts to install Docker.

4.4 Install using the repository:

Before you install Docker Engine for the first time on a new host machine, you need to set up the Docker repository. Afterward, you can install and update Docker from the repository.

Set up the repository

Install the yum-utils package (which provides the yum-config-manager utility) and set up the repository.

```
$ sudo yum install -y yum-utils
```

```
$ sudo yum-config-manager \
    --add-repo \
    https://download.docker.com/linux/centos/docker-ce.repo
```

Install Docker Engine

1. Install Docker Engine, containerd, and Docker Compose:

o Latest

o Specific version

To install the latest version, run:

```
$sudo yum install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

If prompted to accept the GPG key, verify that the fingerprint matches 060A 61C5 1B55 8A7F 742B 77AA C52F EB6B 621E 9F35, and if so, accept it.

This command installs Docker, but it doesn't start Docker. It also creates a docker group, however, it doesn't add any users to the group by default.

1. Start Docker.

```
$ sudo systemctl start docker
```

Verify that Docker Engine installation is successful by running the hello-world image.

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

1. This command downloads a test image and runs it in a container. When the container runs, it prints a confirmation message and exits.

You have now successfully installed and started Docker Engine. The docker user group exists but contains no users, which is why you're required to use sudo to run Docker commands.

Continue to Linux postinstall to allow non-privileged users to run Docker commands and for other optional configuration steps.

4.5 Uninstall Docker Engine

1. Uninstall the Docker Engine, CLI, containerd, and Docker Compose packages:

$ sudo yum remove docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin docker-ce-rootless-extras
1. Images, containers, volumes, or customized configuration files on your host are not automatically removed. To delete all images, containers, and volumes:
2. $ sudo rm -rf /var/lib/docker
3. $ sudo rm -rf /var/lib/containerd
You must delete any edited configuration files manually

4.5.1 Install Docker Engine on Ubuntu

Set up the repository

1. Update the apt package index and install packages to allow apt to use a repository over HTTPS:
   $ sudo apt-get update

   $ sudo apt-get install ca-certificates curl gnupg

2. Add Docker's official GPG key:

   $ sudo install -m 0755 -d /etc/apt/keyrings

   $ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

   $ sudo chmod a+r /etc/apt/keyrings/docker.gpg

3. Use the following command to set up the repository:

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

Install Docker Engine

1. Update the apt package index:
   ```
   $ sudo apt-get update
   ```

2. To install the latest version, run:

   ```
   $ sudo apt-get install docker-ce docker-ce-cli containerd.io
   ```

4.6 Explore Containers

The Containers view lists all your running containers and applications. You must have running or stopped containers and applications to see them listed.

Container actions??

Use the Search field to search for any specific container.

From the Containers view you can perform the following actions on one or more containers at once:

* Pause/Resume

* Stop/Start

* Delete

* Open the application in VS code

* Open the port exposed by the container in a browser

* Copy docker run. This allows you to easily share container run details or modify certain parameters

Integrated terminal??

You also have the option to open an integrated terminal, on a running container, directly within Docker Desktop. This allows you to quickly run commands within your container so you can understand its current state or debug when something goes wrong.

Using the integrated terminal is the same as running docker exec -it <container-id> /bin/sh, or docker exec -it <container-id> cmd.exe if you are using Windows containers, in your external terminal. It also:

* Automatically detects the default user for a running container from the image's Dockerfile. If no use is specified it defaults to root.

* Persists your session if you navigate to another part of the Docker Dashboard and then return.

* Supports copy, paste, search, and clearing your session.

To open the integrated terminal, either:

* Hover over your running container and select the Show container actions menu. From the dropdown menu, select Open in terminal.

* Select the container and then select the Terminal tab.

To use your external terminal, change your settings.

4.7 Inspect a container:

You can obtain detailed information about the container when you select a container.

The container view displays Logs, Inspect, Terminal, Files, and Stats tabs and provides quick action buttons to perform various actions.

* Select Logs to see logs from the container. You can also:

o Use Cmd + f/Ctrl + f to open the search bar and find specific entries. Search matches are highlighted in yellow.

o Press Enter or Shift + Enter to jump to the next or previous search match respectively.

o Use the Copy icon in the top right-hand corner to copy all the logs to your clipboard.

o Automatically copy any logs content by highlighting a few lines or a section of the logs.

o Use the Clear terminal icon in the top right-hand corner to clear the logs terminal.

o Select and view external links that may be in your logs.

* Select Inspect to view low-level information about the container. You can see the local path, version number of the image, SHA-256, port mapping, and other details.

* Select Files to explore the filesystem of running or stopped containers. You can also:

o See which files have been recently added, modified, or deleted

o Edit a file straight from the built-in editor

o Drag and drop files and folders between the host and the container

o Delete unnecessary files when you right-click on a file

o Download file and folders from the container straight to the host

* Select Stats to view information about the container resource utilization. You can see the amount of CPU, disk I/O, memory, and network I/O used by the container.

4.8 Explore Images

The Images view is a simple interface that lets you manage Docker images without having to use the CLI. By default, it displays a list of all Docker images on your local disk.

You can also view Hub images once you have signed in to Docker Hub. This allows you to collaborate with your team and manage your images directly through Docker Desktop.

The Images view allows you to perform core operations such as running an image as a container, pulling the latest version of an image from Docker Hub, pushing the image to Docker Hub, and inspecting images.

The Images view displays metadata about the image such as the:

* Tag

* Image ID

* Date created

* Size of the image.

It also displays In Use tags next to images used by running and stopped containers. You can choose what information you want displayed by selecting the More options menu to the right of the search bar, and then use the toggle switches according to your preferences.

The Images on disk status bar displays the number of images and the total disk space used by the images and when this information was last refreshed.

Manage your images

Use the Search field to search for any specific image.

You can sort images by:

* In use

* Unused

* Dangling

Run an image as a container??

From the Images view, hover over an image and select Run.

When prompted you can either:

* Select the Optional settings drop-down to specify a name, port, volumes, environment variables and select Run

* Select Run without specifying any optional settings.

Inspect an image

To inspect an image, simply select the image row. Inspecting an image displays detailed information about the image such as the:

* Image history

* Image ID

* Date the image was created

* Size of the image

* Layers making up the image

* Base images used

* Vulnerabilities found

* Packages inside the image

* The image view is powered by Docker Scout. For more information about this view, see Image details view

* Pull the latest image from Docker Hub

* Select the image from the list, select the More options button and select Pull.

* Note

* The repository must exist on Docker Hub in order to pull the latest version of an image. You must be logged in to pull private images.

Push an image to Docker Hub

Select the image from the list, select the More options button and select Push to Hub.

Note

You can only push an image to Docker Hub if the image belongs to your Docker ID or your organization. That is, the image must contain the correct username/organization in its tag to be able to push it to Docker Hub.

Remove an image

Note

To remove an image used by a running or a stopped container, you must first remove the associated container.

You can remove individual images or use the Clean up option to delete unused and dangling images.

An unused image is an image which is not used by any running or stopped containers. An image becomes dangling when you build a new version of the image with the same tag.

To remove individual images, select the image from the list, select the More options button and select Remove

To remove an unused or a dangling image:

1. Select the Clean up option from the Images on disk status bar.

2. Use the Unused or Dangling check boxes to select the type of images you would like to remove.

The Clean up images status bar displays the total space you can reclaim by removing the selected images. 3.. Select Remove to confirm.

Interact with remote repositories

The Images view also allows you to manage and interact with images in remote repositories. By default, when you go to Images in Docker Desktop, you see a list of images that exist in your local image store. The Local and Hub tabs near the top toggles between viewing images in your local image store, and images in remote Docker Hub repositories that you have access to.

You can also connect JFrog Artifactory registries, and browse images in JFrog repositories directly in Docker Desktop.

4.9 Docker Hub

Switching to the Hub tab prompts you to sign in to your Docker ID, if you're not already signed in. When signed in, it shows you a list of images in Docker Hub organizations and repositories that you have access to.

Select an organization from the drop-down to view a list of repositories for that organization.

If you have enabled Vulnerability Scanning in Docker Hub, the scan results appear next to the image tags.

Hovering over an image tag reveals two options:

* Pull: pulls the latest version of the image from Docker Hub.
* View in Hub: opens the Docker Hub page and displays detailed information about the image.

Artifactory

The Artifactory integration lets you interact with images in JFrog Artifactory, and JFrog container registry, directly in the Images view of Docker Desktop. The integration described here connects your local Docker Desktop client with Artifactory. You can browse, filter, save, and pull images in the Artifactory instance you configure.
You may also want to consider activating automatic image analysis for your Artifactory repositories. Learn more about Artifactory integration with Docker Scout.
Connect an Artifactory registry
To connect a new Artifactory registry to Docker Desktop:
1. Sign in to an Artifactory registry using the docker login command:

$cat ./password.txt | docker login -u <username> --password-stdin <hostname>
1.
o password.txt: text file containing your Artifactory password.
o username: your Artifactory username.
o hostname: hostname for your Artifactory instance.
2. Open the Images view in Docker Desktop.
3. Select the Artifactory tab near the top of the image view to see Artifactory images.
When signed in, a new Artifactory tab appears in the Images view. By default, the image list shows images sorted by push date: the newest images appear higher in the list.

Chapter 5: Overview of Docker Build

Docker Build is one of Docker Engine's most used features. Whenever you are creating an image, you are using Docker Build. Build is a key part of your software development life cycle allowing you to package and bundle your code and ship it anywhere.

The Docker Engine uses a client-server architecture and is composed of multiple components and tools. The most common method of executing a build is by issuing a docker build command. The CLI sends the request to Docker Engine which, in turn, executes your build.

There are now two components in Engine that can be used to build an image. Starting with the 18.09 release, Engine is shipped with Moby BuildKit, the new component for executing your builds by default.

The new client Docker Buildx is a CLI plugin that extends the docker command with the full support of the features provided by BuildKit builder toolkit. docker buildx build command provides the same user experience as docker build with many new features like creating scoped builder instances, building against multiple nodes concurrently, outputs configuration, inline build caching, and specifying target platform. In addition, Buildx also supports new features that aren't yet available for regular docker build like building manifest lists, distributed caching, and exporting build results to OCI image tar balls.

Docker Build is more than a simple build command, and it's not only about packaging your code. It's a whole ecosystem of tools and features that support not only common workflow tasks but also provides support for more complex and advanced scenarios.

Packaging your software

5.1 Dockerfile

It all starts with a Dockerfile.

Docker builds images by reading the instructions from a Dockerfile. This is a text file containing instructions that adhere to a specific format needed to assemble your application into a container image and for which you can find its specification reference in the Dockerfile reference.

Here are the most common types of instructions:

Dockerfiles are crucial inputs for image builds and can facilitate automated, multi-layer image builds based on your unique configurations. Dockerfiles can start simple and grow with your needs and support images that require complex instructions. For all the possible instructions, see the Dockerfile reference.

The default filename to use for a Dockerfile is Dockerfile, without a file extension. Using the default name allows you to run the docker build command without having to specify additional command flags.

Some projects may need distinct Dockerfiles for specific purposes. A common convention is to name these <something>.Dockerfile. Such Dockerfiles can then be used through the --file (or -f shorthand) option on the docker build command. Refer to the "Specify a Dockerfile" section in the docker build reference to learn about the --file option.

Note

We recommend using the default (Dockerfile) for your project's primary Dockerfile.

Docker images consist of read-only layers, each resulting from an instruction in the Dockerfile. Layers are stacked sequentially and each one is a delta representing the changes applied to the previous layer.

5.2 Container Transparency

Developers commonly package applications as containers to, among other benefits, isolate the application process. While process isolation is one of the aims of containerization, it also means developers lose immediate visibility into the state of the containerized process and its

environment.

To regain that visibility, containerization tools such as Docker provide a way to start new processes within containers in the running state. This is useful for example when you want to read a log file, verify the value of an environment variable, or debug a process.

5.3 Container Layers

Container images are characterized as immutable and layered. Each image layer consists of a set of file system differences, or diffs. A diff signals a file system change from the previous layer, such as adding or modifying a file.

When you start a container, the container uses a new ephemeral layer over its base container image layers. This layer is the only read/write storage available for the container by default, and it is used for any runtime file system operations, such as creating working files, temporary files, and log files. Those files are considered volatile, which means the files are deleted when you delete the container. The container storage layer is exclusive to the running container, so if you create

another container from the same base image, the new container creates another read/write layer.

This ensures each container's runtime data are isolated from other containers. Ephemeral container storage is not sufficient for applications that need to keep data beyond the life of the container, such as databases. To support such applications, the administrator must

provide a container with persistent storage.

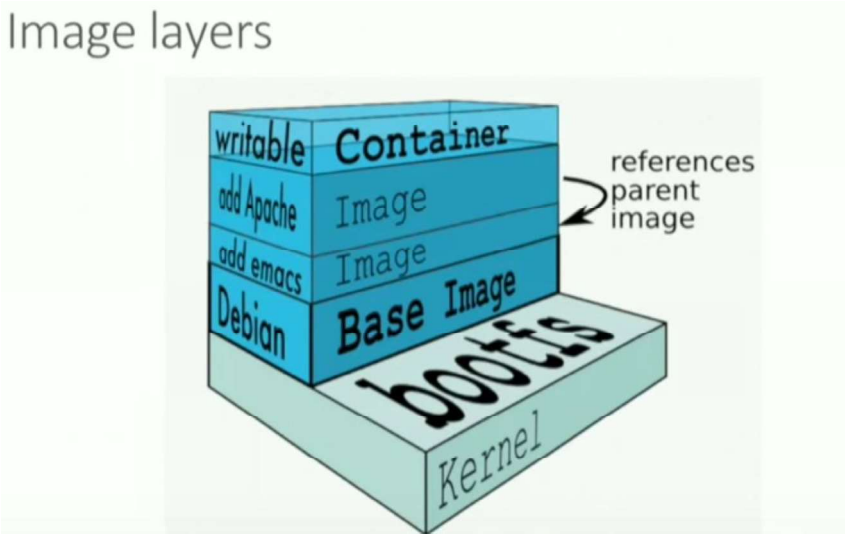


FIGURE 3: Layers of Containers

5.4 Managing Containers Lifecycle:

Dockers provides a set of subcommands to create and manage containers. You can use those subcommands to manage the container and container image lifecycle.
The following figures shows a summary of the most used docker's subcommands that change the container and image state:

Image handling Containers state

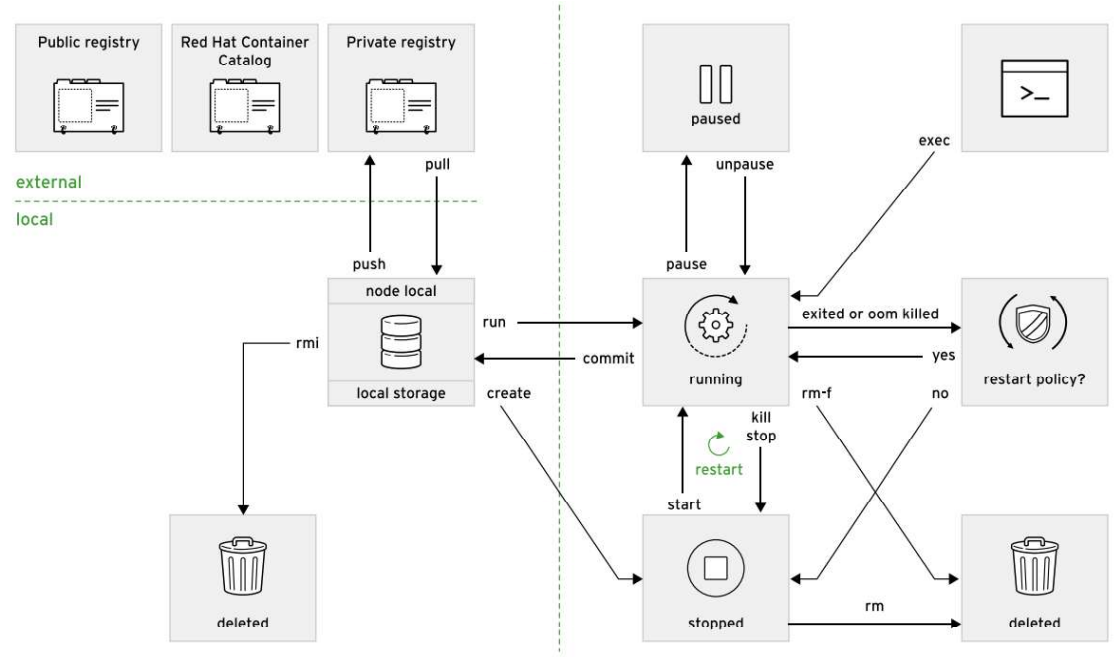
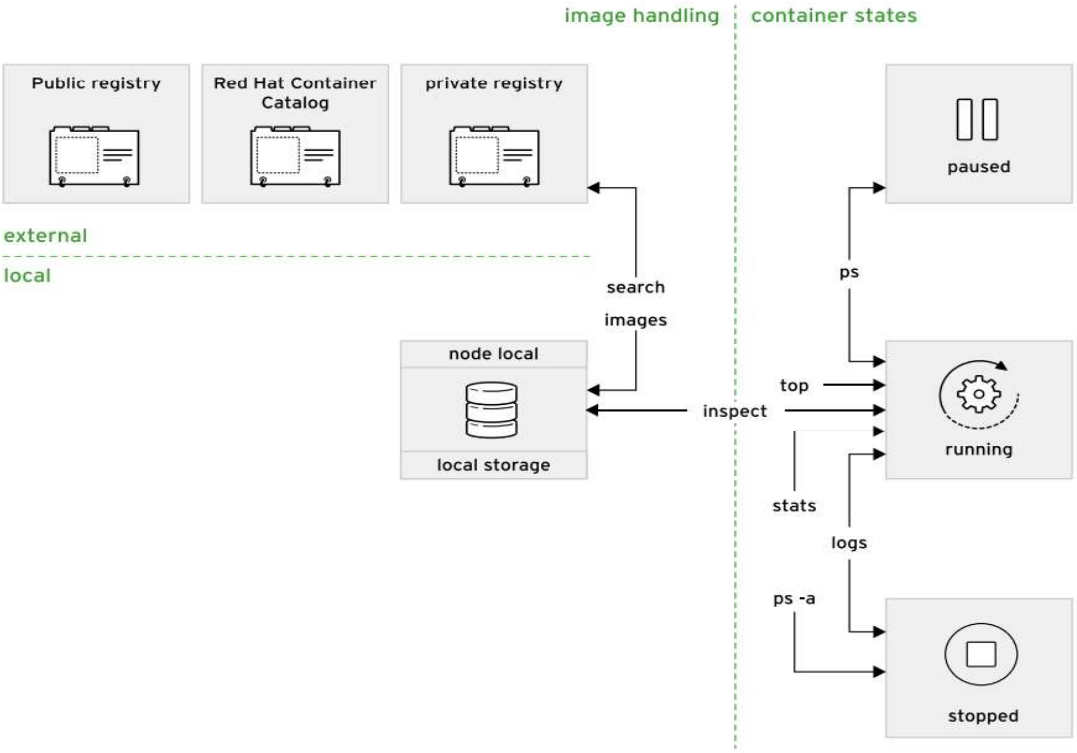


FIGURE 4: Lifecycle of Containers



Chapter 6: Docker Commands

6.1 Listing Containers

You can list running containers with the docker ps command.

```
[user@host ~]$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
 NAMES
0ae7be593698 ...server:latest /bin/sh -c python... ...ago Up... ...8000/tcp  http-server
c42e7dca12d9 ...helloworld:latest /bin/sh -c nginx... ...ago Up... ...8080/tcp  nginx
```

Each row describes information about the container, such as the image used to start the container, the command executed when the container started, and the container uptime. You can include stopped containers in the output by adding the --all or -a flag to the docker ps command.

```
[user@host ~]$ docker ps --all
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
 NAMES
0ae7be593698 ...server:latest /bin/sh -c python... ...ago Up... ...8000/
tcp http-server
bd5ada1b6321 ...httpd-24:latest /usr/bin/run-http... ...ago Exited... ...8080/tcp upbeat...
c42e7dca12d9 ...helloworld:latest /bin/sh -c nginx ... ...ago Up... ...8080/tcp nginx
```

6.2 Inspecting Containers:

In the context of docker, inspecting a container means retrieving the full information of the container. The docker inspect command returns a JSON array with information about different aspects of the container, such as networking settings, CPU usage, environment variables, status, port mapping, or volumes. The following snippet is a sample output of the command.

```
[user@host ~]$ docker inspect 7763097d11ab
[
 {
 "Id": "7763...cbc0",
 "Created": "2022-05-04T10:00:32.988377257-03:00",
 "Path": "container-entrypoint",
 "Args": [
 "/usr/bin/run-httpd"
 ],
 "State": {
 "OciVersion": "1.0.2-dev",
 "Status": "running",
 "Running": true,
 "Paused": false,
 "Restarting": false,
 "OOMKilled": false,
 "Dead": false,
 "Pid": 9746,
...output omitted...
```

```
 "Image": "d2b9...fa0a",
 "ImageName": "registry.access.redhat.com/ubi8/httpd-24:latest",
 "Rootfs": "",
...output omitted...
 "Env": [
 "PATH=/opt/app-root/src/bin:/opt/app-root/bin:/usr/local/sbin:/
usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
 "TERM=xterm",
 "container=oci",
 "HTTPD_VERSION=2.4",
...output omitted...
 "CpuCount": 0,
 "CpuPercent": 0,
 "IOMaximumIOps": 0,
 "IOMaximumBandwidth": 0,
 "CgroupConf": null
```

6.3 Stopping Containers:

You can stop a container gracefully by giving the container some time to perform a clean up. If you want to stop the container immediately regardless of any running processes, then you can stop the container forcefully

6.4 Stopping Containers Gracefully:

You can stop a container gracefully by using the docker stop command. The following command stops a container with a container ID 1b982aeb75dd.

```
[user@host ~]$ docker stop 1b982aeb75dd 1b982aeb75dd
```

You can stop all the running containers by using the --all flag. In the following example, the command stops three containers.

```
[user@host ~]$ docker stop --all
4aea164104108426ca9e6f0741985f81532ce555b154126bbdd6558663af0c2a
6b18b31e44c4e0464baf5354f298a9131decb937fa7d7fcfcb8c0a42a97754ea
7763097d11ab3d7def5b99f22e1984c0031d6096af7d606ff40a6e626fdbcbc0
```

When you execute the docker stop command, Docker sends a SIGTERM signal to the container. This signal gives time for the container to perform any clean up before stopping. After some seconds, if the container has not stopped on its own, Docker sends a SIGKILL signal to forcefully stop the container. By default, Docker waits 10 seconds before sending the SIGKILL signal. You can change the default behavior by using the --time flag

```
[user@host ~]$ docker stop --time=100
```

In the previous example, Docker gives the container a grace period of 100 seconds before sending the killing signal.

6.5 Stopping Containers Forcefully

You can directly send the SIGKILL signal by using the docker kill command. In the following example, a container called httpd is stopped forcefully.

```
[user@host ~]$ docker kill httpd
Httpd
```

6.6 Pausing Containers

Both docker stop and docker kill eventually send a SIGKILL signal to the container. The docker pause command suspends all processes in the container by sending the SIGSTOP signal.

6.7 Restarting Containers

Because a stopped container is not automatically removed, you can use the docker restart command to start a stopped container again. If the command is applied to a running container, then the container is also restarted.

The following command restarts a container called nginx.

```
[user@host ~]$ docker restart nginx
1b982aeb75dd
```

6.8 Removing Containers:

Use the docker rm command to remove a stopped container. The following command removes a stopped container with the container ID c58cfd4b90df.

```
[user@host ~]$ docker rm c58cfd4b90df
c58cfd4b90df50c46fe40bddc68791ac67107452da71b4c2cc3aa2f869c93150
```

By default, you cannot remove running containers. First, you must stop the running container and then remove it. The following command tries to remove a running container.

```
[user@host ~]$ docker rm c58cfd4b90df
```

Error: cannot remove container c58c...3150 as it is running - running or paused containers cannot be removed without force: container state improper However, you can add the --force flag to remove the container forcefully.

```
[user@host ~]$ docker rm c58cfd4b90df --force
c58cfd4b90df50c46fe40bddc68791ac67107452da71b4c2cc3aa2f869c93150
```

You can also add the --all flag to remove all stopped containers. This flag fails to remove running containers. The following command removes two containers.

```
[user@host ~]$ docker rm --all
6b18b31e44c4e0464baf5354f298a9131decb937fa7d7fcfcb8c0a42a97754ea
6c0d18bee2c21759eba4d8b3c54a1da6cebed2caebeeb7152366d7105120a6fb
```

6.9 Container Image Registries

A container image is a packaged version of your application, with all the dependencies necessary for the application to run. You can use image registries to store container images to later share them in a controlled manner. Some examples of image registries include Quay.io, Red Hat Registry, Docker Hub, or Amazon ECR. For example, consider the following Docker command.

```
[user@host ~]$ docker pull registry.redhat.io/ubi8/ubi:8.6
Trying to pull registry.redhat.io/ubi8/ubi:8.6...
Getting image source signatures
...output omitted...
Writing manifest to image destination
Storing signatures
3434...8f6b
```

The registry.redhat.io/ubi8 image is stored in the Red Hat Registry (notice the host where the image is stored, registry.redhat.io)

6.10 The Containerfile

A Containerfile, often called Dockerfile, is a file that contains instructions indicating how the image is created. Usually, container images are created based on other container images. For example, an image that outputs Hello world might be created by using a Linux parent image. The Linux image provides the capabilities to output text, and the child image uses the capabilities of the parent image to print the text.

Consider the following Containerfile:

```
FROM registry.redhat.io/ubi9
CMD echo "Hello world"

# The FROM instruction indicates the parent image to use. In this case, a Universal Base Image (UBI), Version 9 container image is used.
# The CMD instruction executes a command. In this case, the echo command is used to output some text.
```

Chapter 7 : Guided Exercises For Docker Containers

In this exercise, you will install Docker on a virtual machine hosted in Google cloud platform.

Ensure your GCP credentials are working and you could able to access the virtual machine over SSH.

Steps

1. Switch as root user

```
username@node1# sudo -i
```

2. Install docker from yum repository

```
username@node1# yum install docker
```

```
[oselabsproduction@node1 ~]$
[oselabsproduction@node1 ~]$
[oselabsproduction@node1 ~]$ su -
Password:
[oselabsproduction@node1 ~]$ sudo -i
[root@node1 ~]#
[root@node1 ~]#
[root@node1 ~]# cat /etc/redhat-release
CentOS Linux release 7.9.2009 (Core)
[root@node1 ~]#
[root@node1 ~]#
[root@node1 ~]# yum install docker
Loaded plugins: fastestmirror
Determining fastest mirrors
epel/x86_64/metalink                                                    |  18 kB  00:00:00
 * base: mirror.genesisadaptive.com
 * epel: dfw.mirror.rackspace.com
 * extras: mirrors.cmich.edu
 * updates: repos.forethought.net
base                                                                    | 3.6 kB  00:00:00
epel                                                                    | 4.7 kB  00:00:00
http://mirrors.cmich.edu/centos/7.9.2009/extras/x86_64/repodata/repomd.xml: [Errno 12] Timeout on http://mirror
s.cmich.edu/centos/7.9.2009/extras/x86_64/repodata/repomd.xml: (28, 'Operation too slow. Less than 1000 bytes/s
ec transferred the last 30 seconds')
Trying other mirror.
extras                                                                  | 2.9 kB  00:00:00
google-cloud-sdk                                                        | 1.4 kB  00:00:00
google-compute-engine                                                   | 1.4 kB  00:00:00
updates                                                                 | 2.9 kB  00:00:00
(1/9): base/7/x86_64/group_gz                                           | 153 kB  00:00:00
(2/9): base/7/x86_64/primary_db                                         | 6.1 MB  00:00:00
(3/9): epel/x86_64/group_gz                                             |  99 kB  00:00:00
(4/9): epel/x86_64/updateinfo                                           | 1.0 MB  00:00:00
(5/9): extras/7/x86_64/primary_db                                       | 249 kB  00:00:00
(6/9): google-compute-engine/primary                                    | 3.9 kB  00:00:00
(7/9): google-cloud-sdk/primary                                         | 732 kB  00:00:00
```

Few Outputs are omitted ……

```
 oci-umount                                x86_64    2:2.5-3.el7                      extras      33 k
 policycoreutils-python                    x86_64    2.5-34.el7                       base       457 k
 python-IPy                                noarch    0.75-6.el7                       base        32 k
 python-backports                          x86_64    1.0-8.el7                        base       5.8 k
 python-backports-ssl_match_hostname       noarch    3.5.0.1-1.el7                    base        13 k
 python-dateutil                           noarch    1.5-7.el7                        base        85 k
 python-dmidecode                          x86_64    3.12.2-4.el7                     base        83 k
 python-ethtool                            x86_64    0.8-8.el7                        base        34 k
 python-inotify                            noarch    0.9.4-4.el7                      base        49 k
 python-ipaddress                          noarch    1.0.16-2.el7                     base        34 k
 python-pytoml                             noarch    0.1.14-1.git7dea353.el7          extras      18 k
 python-setuptools                         noarch    0.9.8-7.el7                      base       397 k
 python-six                                noarch    1.9.0-2.el7                      base        29 k
 python-syspurpose                         x86_64    1.24.51-1.el7.centos             updates    275 k
 setools-libs                              x86_64    3.3.8-4.el7                      base       620 k
 slirp4netns                               x86_64    0.4.3-4.el7_8                    extras      81 k
 subscription-manager                      x86_64    1.24.51-1.el7.centos             updates    1.1 M
 subscription-manager-rhsm                 x86_64    1.24.51-1.el7.centos             updates    334 k
 subscription-manager-rhsm-certificates    x86_64    1.24.51-1.el7.centos             updates    237 k
 usermode                                  x86_64    1.111-6.el7                      base       193 k
 yajl                                      x86_64    2.0.4-4.el7                      base        39 k

Transaction Summary
================================================================================================================
Install  1 Package (+45 Dependent packages)

Total download size: 31 M
Installed size: 105 M
Is this ok [y/d/N]:
```

When prompted for [y/N], type “y” and press enter.

```
  lvm2-libs.x86_64 7:2.02.187-6.el7_9.5
  oci-register-machine.x86_64 1:0-6.git2b44233.el7
  oci-systemd-hook.x86_64 1:0.2.0-1.git05c6923.el7_6
  oci-umount.x86_64 2:2.5-3.el7
  policycoreutils-python.x86_64 0:2.5-34.el7
  python-IPy.noarch 0:0.75-6.el7
  python-backports.x86_64 0:1.0-8.el7
  python-backports-ssl_match_hostname.noarch 0:3.5.0.1-1.el7
  python-dateutil.noarch 0:1.5-7.el7
  python-dmidecode.x86_64 0:3.12.2-4.el7
  python-ethtool.x86_64 0:0.8-8.el7
  python-inotify.noarch 0:0.9.4-4.el7
  python-ipaddress.noarch 0:1.0.16-2.el7
  python pytoml.noarch 0:0.1.14 1.git7dea353.el7
  python-setuptools.noarch 0:0.9.8-7.el7
  python-six.noarch 0:1.9.0-2.el7
  python-syspurpose.x86_64 0:1.24.51-1.el7.centos
  setools-libs.x86_64 0:3.3.8-4.el7
  slirp4netns.x86_64 0:0.4.3-4.el7_8
  subscription-manager.x86_64 0:1.24.51-1.el7.centos
  subscription-manager-rhsm.x86_64 0:1.24.51-1.el7.centos
  subscription-manager-rhsm-certificates.x86_64 0:1.24.51-1.el7.centos
  usermode.x86_64 0:1.111-6.el7
  yajl.x86_64 0:2.0.4-4.el7

Complete!
[root@node1 ~]#
```

The above image confirms the successful completion of Docker installation.

3. Verify the Docker installation

```
username@node1# yum list installed docker
```

```
[root@node1 ~]# yum list installed docker
Loaded plugins: fastestmirror, product-id, search-disabled-repos, subscription-manager

This system is not registered with an entitlement server. You can use subscription-manager to register.

Loading mirror speeds from cached hostfile
 * base: mirror.genesisadaptive.com
 * epel: dfw.mirror.rackspace.com
 * extras: mirrors.cmich.edu
 * updates: repos.forethought.net
Installed Packages
docker.x86_64                          2:1.13.1-209.git7d71120.el7.centos                          @extras
[root@node1 ~]#
```

4. List Docker images

```
username@node1#docker images
```

```
[root@node1 ~]# docker images
Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?
```

The above docker image command throws error for which we must start the docker daemon.

5. Starting Docker container Service

```
username@node1# systemctl start docker.service
```

```
[root@node1 ~]# systemctl start docker.service
[root@node1 ~]# systemctl status docker.service
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; disabled; vendor preset: disabled)
   Active: active (running) since Mon 2023-04-17 10:12:27 UTC; 8s ago
     Docs: http://docs.docker.com
 Main PID: 1782 (dockerd-current)
   CGroup: /system.slice/docker.service
           ├─1782 /usr/bin/dockerd-current --add-runtime docker-runc=/usr/libexec/docker/docker-runc-current...
           └─1788 /usr/bin/docker-containerd-current -l unix:///var/run/docker/libcontainerd/docker-containe...

Apr 17 10:12:26 node1 dockerd-current[1782]: time="2023-04-17T10:12:26.266994774Z" level=info msg="libco...788"
Apr 17 10:12:27 node1 dockerd-current[1782]: time="2023-04-17T10:12:27.362238602Z" level=info msg="Graph...nds"
Apr 17 10:12:27 node1 dockerd-current[1782]: time="2023-04-17T10:12:27.363806905Z" level=info msg="Loadi...rt."
Apr 17 10:12:27 node1 dockerd-current[1782]: time="2023-04-17T10:12:27.397653955Z" level=info msg="Firew...rue"
Apr 17 10:12:27 node1 dockerd-current[1782]: time="2023-04-17T10:12:27.622540769Z" level=info msg="Defau...ess"
Apr 17 10:12:27 node1 dockerd-current[1782]: time="2023-04-17T10:12:27.810281539Z" level=info msg="Loadi...ne."
Apr 17 10:12:27 node1 dockerd-current[1782]: time="2023-04-17T10:12:27.836548539Z" level=info msg="Daemo...ion"
Apr 17 10:12:27 node1 dockerd-current[1782]: time="2023-04-17T10:12:27.836590910Z" level=info msg="Docke...13.1
Apr 17 10:12:27 node1 dockerd-current[1782]: time="2023-04-17T10:12:27.842002711Z" level=info msg="API 1...ock"
Apr 17 10:12:27 node1 systemd[1]: Started Docker Application Container Engine.
Hint: Some lines were ellipsized, use -l to show in full.
[root@node1 ~]#
```

6. Re-run the command “docker images” after starting the docker daemon.

```
[root@node1 ~]#
[root@node1 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
[root@node1 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
        NAMES
[root@node1 ~]#
```

7. Search images from registry server

username@node1# docker search ubi9

```
[root@node1 ~]# docker search ubi9
INDEX       NAME                                    DESCRIPTION                                     STARS
  OFFICIAL   AUTOMATED
docker.io   docker.io/redhat/ubi9                   Red Hat Universal Base Image 9                  15

docker.io   docker.io/redhat/ubi9-minimal           Red Hat Universal Base Image 9 Minimal          6

docker.io   docker.io/redhat/ubi9-micro             Red Hat Universal Base Image 9 Micro            4

docker.io   docker.io/redhat/ubi9-init              Red Hat Universal Base Image 9 Init             2
```

Here the ubi is an image name of universal base image.

8. Create and run a container from an image, with a custom name:

docker run --name <container_name> <image_name>

```
[root@node1 ~]# docker run --name myos docker.io/redhat/ubi9
Unable to find image 'docker.io/redhat/ubi9:latest' locally
Trying to pull repository docker.io/redhat/ubi9 ...
latest: Pulling from docker.io/redhat/ubi9
72d37ae8760a: Pull complete
Digest: sha256:49124e4acd09c98927882760476d617a85f155cb45759aea56b2ab020563c4b8
Status: Downloaded newer image for docker.io/redhat/ubi9:latest
[root@node1 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
        NAMES
[root@node1 ~]# docker ps -a
CONTAINER ID        IMAGE                  COMMAND             CREATED             STATUS
PORTS               NAMES
e65a1705c3d5        docker.io/redhat/ubi9  "/bin/bash"         11 seconds ago      Exited (0) 8 seconds ago
                    myos
```

The container creation was NOT successful in the above command as the container doesnot have any purpose or tasks.

9. Add sleep tasks to the existing command

```
[root@node1 ~]# docker run --name myos2 docker.io/redhat/ubi9 sleep 30
```

10. Open another terminal to check the container "myos2" status

```
[root@node1 ~]# docker ps
CONTAINER ID        IMAGE                    COMMAND             CREATED             STATUS              PORTS
              NAMES
60bac13d19e6        docker.io/redhat/ubi9    "sleep 30"          8 seconds ago       Up 6 seconds
              myos2
[root@node1 ~]# docker ps  -a
CONTAINER ID        IMAGE                    COMMAND             CREATED             STATUS
PORTS               NAMES
60bac13d19e6        docker.io/redhat/ubi9    "sleep 30"          11 seconds ago      Up 9 seconds
                    myos2
e65a1705c3d5        docker.io/redhat/ubi9    "/bin/bash"         2 minutes ago       Exited (0) 2 minutes ago
                    myos
[root@node1 ~]#
```

Once task (sleep 30) is completed, then it will come to prompt – means the container is terminated now.

```
[root@node1 ~]# docker run --name myos2 docker.io/redhat/ubi9 sleep 30
[root@node1 ~]#
[root@node1 ~]#
```

7.1 Containerizing Application in detached mode

1. Search for web server image in registry

```
username@node1# docker search httpd
```

```
[root@node1 ~]# docker search httpd
INDEX       NAME                                      DESCRIPTION                                     STAR
S     OFFICIAL   AUTOMATED
docker.io   docker.io/httpd                           The Apache HTTP Server Project                  4398
      [OK]
docker.io   docker.io/centos/httpd-24-centos7         Platform for running Apache httpd 2.4 or b...   45

docker.io   docker.io/centos/httpd                                                                    36
                 [OK]
```

2. Using httpd image, we can spin a container;

```
username@node1# docker run –name myweb docker.io/centos/httpd
```

```
[root@node1 ~]# docker run --name myweb docker.io/centos/httpd
Unable to find image 'docker.io/centos/httpd:latest' locally
Trying to pull repository docker.io/centos/httpd ...
latest: Pulling from docker.io/centos/httpd
a02a4930cb5d: Pull complete
628eaef4a9e0: Pull complete
20c0ca1c0cd5: Pull complete
30cf2fb1a57e: Pull complete
Digest: sha256:26c6674463ff3b8529874b17f8bb55d21a0dcf86e025eafb3c9eeee15ee4f369
Status: Downloaded newer image for docker.io/centos/httpd:latest
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2. Set th
e 'ServerName' directive globally to suppress this message
```

But it will remain run in foreground…

```
[root@node1 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
         NAMES
[root@node1 ~]# docker ps -a
CONTAINER ID        IMAGE                    COMMAND             CREATED             STATUS
    PORTS               NAMES
8bd7d4e2ce9a        docker.io/centos/httpd   "/run-httpd.sh"     6 minutes ago       Exited (137) 26 seconds ag
o                       myweb
60bac13d19e6        docker.io/redhat/ubi9    "sleep 30"          11 minutes ago      Exited (0) 11 minutes ago
                        myos2
e65a1705c3d5        docker.io/redhat/ubi9    "/bin/bash"         14 minutes ago      Exited (0) 13 minutes ago
                        myos
[root@node1 ~]#
```

3. To make this running in background/ detached mode, we can use "-d" option for detach mode:

```
[root@node1 ~]# docker run -d --name myweb2 docker.io/centos/httpd
4201f506e0a02a6cb072627d1a0d2c68b7c67a0f49554a3aecd03b663036db1b
[root@node1 ~]# docker ps
CONTAINER ID        IMAGE                    COMMAND             CREATED             STATUS              PORTS
            NAMES
4201f506e0a0        docker.io/centos/httpd   "/run-httpd.sh"     4 seconds ago       Up 3 seconds        80/tcp
            myweb2
[root@node1 ~]#
```

7.2 Docker Inspect command

1. Docker Inspect to find the details of containers about ports and images;

username@node1# docker inspect <container-name>

```
[root@node1 ~]# docker ps
CONTAINER ID        IMAGE                    COMMAND             CREATED             STATUS              PORTS
            NAMES
[root@node1 ~]# docker inspect myweb2
[
    {
        "Id": "4201f506e0a02a6cb072627d1a0d2c68b7c67a0f49554a3aecd03b663036db1b",
        "Created": "2023-04-17T10:46:27.470657877Z",
        "Path": "/run-httpd.sh",
        "Args": [],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 2612,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2023-04-17T10:46:27.880730799Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
        "Image": "sha256:2cc07fbb5000234e85b7ef63b6253f397491959af2a24251b6ae20c207beb814",
        "ResolvConfPath": "/var/lib/docker/containers/4201f506e0a02a6cb072627d1a0d2c68b7c67a0f49554a3aecd03b663
036db1b/resolv.conf",
        "HostnamePath": "/var/lib/docker/containers/4201f506e0a02a6cb072627d1a0d2c68b7c67a0f49554a3aecd03b66303
6db1b/hostname",
        "HostsPath": "/var/lib/docker/containers/4201f506e0a02a6cb072627d1a0d2c68b7c67a0f49554a3aecd03b663036db
1b/hosts",
        "LogPath": "",
```

2. To find exposed ports:

```
[root@node1 ~]# docker inspect myweb2 | grep expose
[root@node1 ~]# docker inspect myweb2 | grep -i expose
            "ExposedPorts": {
[root@node1 ~]# docker inspect myweb2 | grep -A5 -i expose
            "ExposedPorts": {
                "80/tcp": {}
            },
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
[root@node1 ~]#
[root@node1 ~]#
[root@node1 ~]#
```

3. To check with curl command:

```
[root@4201f506e0a0 html]# curl localhost
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"><html><head>
<meta http equiv="content type" content="text/html; charset=UTF 8">
                <title>Apache HTTP Server Test Page powered by CentOS</title>
                <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

    <!-- Bootstrap -->
    <link href="/noindex/css/bootstrap.min.css" rel="stylesheet">
    <link rel="stylesheet" href="noindex/css/open-sans.css" type="text/css" />

<style type="text/css"><!--

body {
  font-family: "Open Sans", Helvetica, sans-serif;
  font-weight: 100;
  color: #ccc;
  background: rgba(10, 24, 55, 1);
  font-size: 16px;
}
```

4. Create an “index.html” in /var/www/html which is documentRoot.
   Note: this is executing inside the container

```
[root@4201f506e0a0 html]# cat > index.html
Welcome to Containers!!!
[root@4201f506e0a0 html]#
[root@4201f506e0a0 html]# curl localhost
Welcome to Containers!!!
[root@4201f506e0a0 html]#
```

Once created, verify with the above “curl localhost” command which will hit port 80 by default;

Note: To access the web page from the host or external source, at this point it’s not possible. Hence, we are configuring port mapping.

```
[root@node1 ~]# docker ps
CONTAINER ID        IMAGE                    COMMAND             CREATED             STATUS              PORTS
              NAMES
4201f506e0a0        docker.io/centos/httpd   "/run-httpd.sh"     10 minutes ago      Up 10 minutes       80/tcp
              myweb2
[root@node1 ~]# docker stop myweb2
myweb2
[root@node1 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
         NAMES
[root@node1 ~]#
```

7.3 Port Mapping:

1. Create container with mapping port 8080 of host operating system to port 80 of container named “myweb3”

```
username@node1# docker run -d –name myweb3 -p 8080:80 docker.io/centos/httpd
```

```
[root@node1 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
         NAMES
[root@node1 ~]#
[root@node1 ~]# docker run -d --name myweb3 -p 8080:80 docker.io/centos/httpd
691b597347870820e3f3553fb937fd1daa7e42f5433f5166d64f166209752b4b
[root@node1 ~]# docker ps
CONTAINER ID        IMAGE                  COMMAND             CREATED             STATUS              PORTS
                NAMES
691b59734787        docker.io/centos/httpd   "/run-httpd.sh"     5 seconds ago       Up 4 seconds        0.0.0.
0:8080->80/tcp   myweb3
[root@node1 ~]#
```

2. Logging into the container interactive shell and editing the index.html file

```
[root@node1 ~]# docker exec -it myweb3 /bin/bash
[root@691b59734787 /]# cd /var/www/html
[root@691b59734787 html]# ls
[root@691b59734787 html]# cat >index.html
Welcome to Containers - Updated!!!
[root@691b59734787 html]#
[root@691b59734787 html]# curl localhost
Welcome to Containers - Updated!!!
[root@691b59734787 html]# exit
exit
[root@node1 ~]# curl localhost:8080
Welcome to Containers - Updated!!!
[root@node1 ~]#
```

3. Update and try Accessing the webpage both within the containers and outside the containers using curl command.

7.4 Containerizing Stateful Application (MySQL)

```
[root@node1 ~]# docker run -d --name mydb docker.io/mariadb
Unable to find image 'docker.io/mariadb:latest' locally
Trying to pull repository docker.io/library/mariadb ...
latest: Pulling from docker.io/library/mariadb
74ac377868f8: Pull complete
9f8acee20aa1: Pull complete
11b336495e01: Pull complete
20ab1641dd41: Pull complete
eaf0c5c99086: Pull complete
239335430207: Pull complete
931baaab2c80: Pull complete
f2e86cc8f052: Pull complete
Digest: sha256:9ff479f244cc596aed9794d035a9f352662f2caed933238c533024df64569853
Status: Downloaded newer image for docker.io/mariadb:latest
11fb2a5ed3bb00440a72e2d7d4359c340573cf6548163e10e563c595b676e632
[root@node1 ~]# docker ps
CONTAINER ID        IMAGE                  COMMAND             CREATED             STATUS              PORTS
                NAMES
691b59734787        docker.io/centos/httpd   "/run-httpd.sh"     10 minutes ago      Up 10 minutes       0.0.0.
0:8080->80/tcp   myweb3
[root@node1 ~]# docker ps -a
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS
         PORTS                  NAMES
11fb2a5ed3bb        docker.io/mariadb      "docker-entrypoint..."   25 seconds ago      Exited (1) 23 seconds
 ago                            mydb
691b59734787        docker.io/centos/httpd   "/run-httpd.sh"          10 minutes ago      Up 10 minutes
        0.0.0.0:8080->80/tcp   myweb3
4201f506e0a0        docker.io/centos/httpd   "/run-httpd.sh"          23 minutes ago      Exited (137) 12 minut
es ago                         myweb2
8bd7d4e2ce9a        docker.io/centos/httpd   "/run-httpd.sh"          42 minutes ago      Exited (137) 36 minut
es ago                         myweb
60bac13d19e6        docker.io/redhat/ubi9    "sleep 30"               47 minutes ago      Exited (0) 47 minutes
 ago                           myos2
e65a1705c3d5        docker.io/redhat/ubi9    "/bin/bash"              50 minutes ago      Exited (0) 49 minutes
 ago                           myos
[root@node1 ~]#
```

Not like webserver, DB needs some env variables.

```
[root@node1 ~]# docker run -d --name mydb2 docker.io/mysql
Unable to find image 'docker.io/mysql:latest' locally
Trying to pull repository docker.io/library/mysql ...
latest: Pulling from docker.io/library/mysql
328ba678bf27: Pull complete
f3f5ff008d73: Pull complete
dd7054d6d0c7: Pull complete
70b5d4e8750e: Pull complete
cdc4a7b43bdd: Pull complete
3e9c0b61a8f3: Pull complete
806a08b6c085: Pull complete
021b2cebd832: Pull complete
ad31ba45b26b: Pull complete
0d4c2bd59d1c: Pull complete
148dcef42e3b: Pull complete
Digest: sha256:f496c25da703053a6e0717f1d52092205775304ea57535cc9fcaa6f35867800b
Status: Downloaded newer image for docker.io/mysql:latest
4dbf8bde8181020c1b9ef9c6982eeb769eacfea1dd77b130aa79c5f8c818ac4b
[root@node1 ~]#
[root@node1 ~]# docker ps
CONTAINER ID        IMAGE                    COMMAND             CREATED             STATUS              PORTS
               NAMES
691b59734787        docker.io/centos/httpd   "/run-httpd.sh"     16 minutes ago      Up 16 minutes       0.0.0.
0:8080->80/tcp   myweb3
```

Check docker logs for detailed error message.

```
[root@node1 ~]# docker logs mydb2
2023-04-17 11:15:55+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 8.0.32-1.el8 started.
2023-04-17 11:15:55+00:00 [Note] [Entrypoint]: Switching to dedicated user 'mysql'
2023-04-17 11:15:55+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 8.0.32-1.el8 started.
2023-04-17 11:15:56+00:00 [ERROR] [Entrypoint]: Database is uninitialized and password option is not specified
    You need to specify one of the following as an environment variable:
    - MYSQL_ROOT_PASSWORD
    - MYSQL_ALLOW_EMPTY_PASSWORD
    - MYSQL_RANDOM_ROOT_PASSWORD
[root@node1 ~]# docker run -d --name mydb3 -e MYSQL_ROOT_PASSWORD=redhat docker.io/mysql
1cece8c3bf177f3c5172b13bf65204797c8846da0af6f77126573b0326dc1927
[root@node1 ~]# docker ps
CONTAINER ID        IMAGE                    COMMAND                  CREATED             STATUS              P
ORTS                  NAMES
1cece8c3bf17        docker.io/mysql          "docker-entrypoint..."   3 seconds ago       Up 3 seconds        3
306/tcp, 33060/tcp    mydb3
691b59734787        docker.io/centos/httpd   "/run-httpd.sh"          17 minutes ago      Up 17 minutes       0
.0.0.0:8080->80/tcp   myweb3
[root@node1 ~]#
```

Environment variables are missing

Logging in to container and verify the MySQL DB.

```
[root@node1 ~]# docker exec -it myweb3 /bin/bash
```

```
bash-4.4# mysql -uuser1 -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 13
Server version: 8.0.32 MySQL Community Server - GPL

Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

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

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| performance_schema |
+--------------------+
2 rows in set (0.01 sec)

mysql>
```

6.5 Persistent Storage for containers (Volume Mapping):

```
[root@node1 ~]# docker stop myweb3
myweb3
[root@node1 ~]# mkdir -p webcontent/html
[root@node1 ~]# cd webcontent/html
[root@node1 html]# cat > index.html
Welcome to DevOps!  Msg from node1
[root@node1 html]#
```

```
[root@node1 html]# docker run -d --name myweb5 -p 8081:80 -v /root/webcontent:/var/www:Z docker.io/centos/httpd

b5f638d339eb5a4cdf8fde22a1454f54d3a8b7dd968d32514effbf9621454f06
[root@node1 html]# docker ps
CONTAINER ID        IMAGE                    COMMAND                  CREATED             STATUS              P
ORTS                  NAMES
b5f638d339eb        docker.io/centos/httpd   "/run-httpd.sh"          3 seconds ago       Up 2 seconds        0
.0.0.0:8081->80/tcp   myweb5
edc7539fabcf        docker.io/mysql          "docker-entrypoint..."   18 minutes ago      Up 18 minutes       3
306/tcp, 33060/tcp    mydb4
[root@node1 html]#
```

```
[root@node1 html]# docker exec -it myweb5 /bin/bash
[root@b5f638d339eb /]# curl localhost
Welcome to DevOps!  Msg from node1
[root@b5f638d339eb /]# exit
exit
[root@node1 html]# curl localhost:8081
Welcome to DevOps!  Msg from node1
[root@node1 html]# pwd
/root/webcontent/html
[root@node1 html]# cat >> index.html
This line is appended in node1
[root@node1 html]# curl localhost:8081
Welcome to DevOps!  Msg from node1
This line is appended in node1
[root@node1 html]# docker exec -it myweb5 /bin/bash
[root@b5f638d339eb /]# cd /var/www/html
[root@b5f638d339eb html]# cat >> index.html
This line is appended in guest
[root@b5f638d339eb html]#
[root@b5f638d339eb html]# curl localhost
Welcome to DevOps!  Msg from node1
This line is appended in node1
This line is appended in guest
[root@b5f638d339eb html]# exit
exit
[root@node1 html]# curl localhost:8081
Welcome to DevOps!  Msg from node1
This line is appended in node1
This line is appended in guest
[root@node1 html]#
```

Chapter 8: Docker Swarm

Docker was designed with multi-host, horizontally scaled production operations in mind. This is a great benefit to your projects, because you can run multiple instances of your application containers and distribute them on multiple servers, so that you can cope with load peaks coming from user requests.

Docker has built-in features to manage your application across several computers, i.e. a computer cluster. This solution is called the Swarm mode.

In order to avoid single point of failure we need multiple Docker hosts to create a swarm cluster.

In the FIG 8.1, manager nodes are no difference to the worker nodes in sharing container workload; except that manager nodes are also responsible for maintaining the status of the cluster using a distributed state store

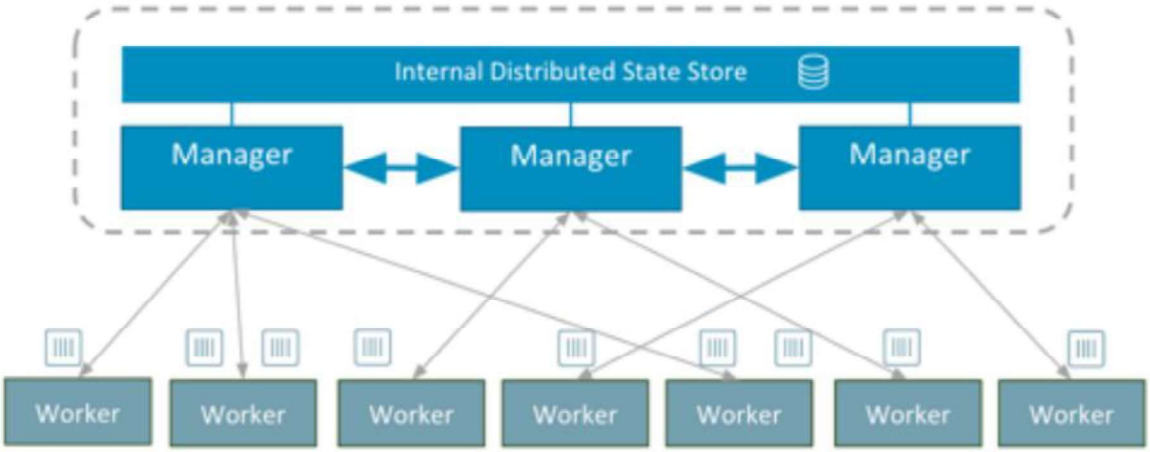


FIG 8.1: Docker Swarm Architecture

8.1 Service and stack:

In the swarm cluster, a container can be started with multiple instances (i.e. replicas). The term service is used to refer to the replicas of the same container. A stack is referred to a group of connected services. Like the single-node orchestration, a stack is also described by a docker-compose file with extra attributes specific for the Docker swarm.

8.2 Docker Compose:

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

Chapter 9: Kubernetes Architecture

Kubernetes is a popular open-source platform for *container orchestration* — that is, for the management of applications built out of multiple, largely self-contained runtimes called *containers*. Containers have become increasingly popular since the Docker containerization project launched in 2013, but large, distributed containerized applications can become increasingly difficult to coordinate. By making containerized applications dramatically easier to manage at scale, Kubernetes has become a key part of the container revolution.

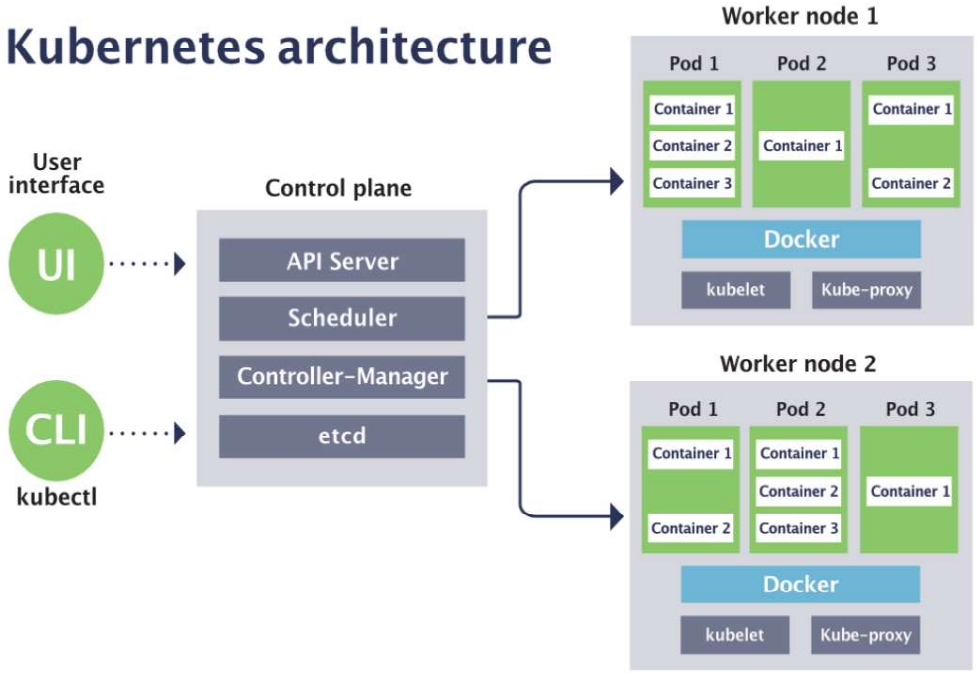


FIG 9.1 Kubernetes Architecture

Kubernetes clusters

The highest-level Kubernetes abstraction, the *cluster*, refers to the group of machines running Kubernetes (itself a clustered application) and the containers managed by it. A Kubernetes cluster must have a *master*, the system that commands and controls all the other Kubernetes machines in the cluster. A highly available Kubernetes cluster replicates the master's facilities across multiple machines. But only one master at a time runs the job scheduler and controller-manager.

Kubernetes nodes and pods

Each cluster contains Kubernetes *nodes*. Nodes might be physical machines or VMs. Again, the idea is abstraction: Whatever the app is running on, Kubernetes handles deployment on that substrate. Kubernetes even makes it possible to ensure that certain containers run only on VMs or only on bare metal.

Nodes run *pods*, the most basic Kubernetes objects that can be created or managed. Each pod represents a single instance of an application or running process in Kubernetes, and consists of one or more containers. Kubernetes starts, stops, and replicates all containers in a pod as a

group. Pods keep the user's attention on the application, rather than on the containers themselves. Details about how Kubernetes needs to be configured, from the state of pods on up, is kept in *Etcd*, a distributed key-value store.

Pods are created and destroyed on nodes as needed to conform to the desired state specified by the user in the pod definition. Kubernetes provides an abstraction called a *controller* for dealing with the logistics of how pods are spun up, rolled out, and spun down. Controllers come in a few different flavors depending on the kind of application being managed. For instance, the StatefulSet controller is used to deal with applications that need persistent state. The Deployment controller is used to scale an app up or down, update an app to a new version, or roll back an app to a known-good version if there's a problem.

Kubernetes services

Because pods live and die as needed, we need a different abstraction for dealing with the application lifecycle. An application is supposed to be a persistent entity, even when the pods running the containers that comprise the application are not themselves persistent. To that end, Kubernetes provides an abstraction called a *service.*

A service in Kubernetes describes how a given group of pods (or other Kubernetes objects) can be accessed via the network. As the Kubernetes documentation puts it, the pods that constitute the back-end of an application might change, but the front-end shouldn't have to know about that or track it. Services make this possible.

********** END **********