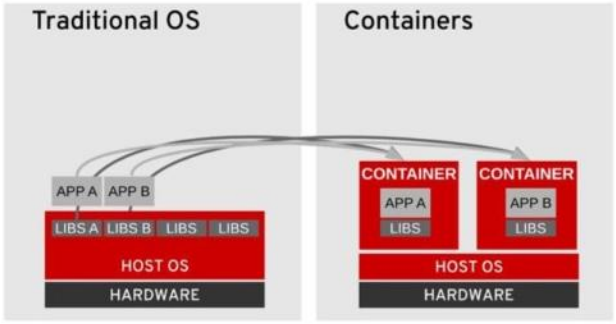
Containers and Docker

Outline

- Introduction to containers
- Introduction to Docker
- Building container images
- Container registries
- Running containers

Containerized Applications



Software applications are typically deployed as a single set of libraries and configuration files to a runtime environment. They are traditionally deployed to an operating system with a set of services running. Any updates or patches applied to the base OS might break the application. Moreover, if another application is sharing the same host OS and the same set of libraries, there might be a risk of breaking it if an update that fixes the first application libraries affects the second application.

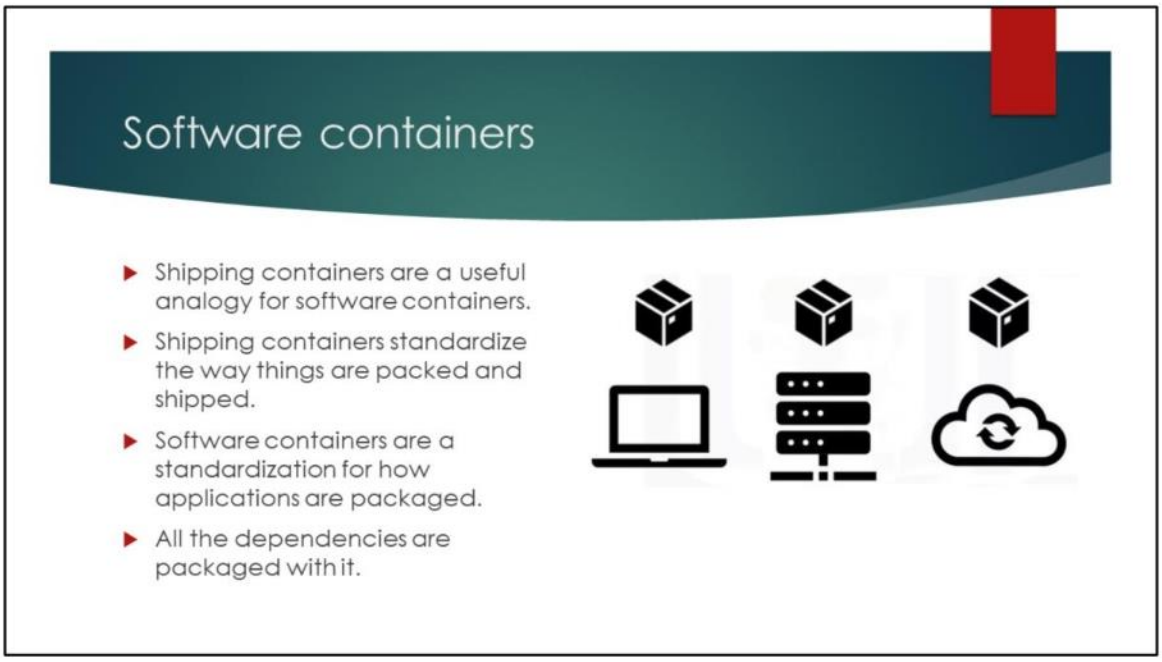
Alternatively, a system administrator can work with containers, which are a kind of isolated partition inside the operating system. They also isolate the libraries and the runtime environment (such as CPU and storage) used by an application to minimize the impact of any OS update.

What is a container?

- Executable unit of software
  - Encapsulates everything necessary to run
  - Can be run anywhere
- OS-level virtualization
  - Isolates processes
- Small, fast, and portable
  - Does not include a guest OS in every instance

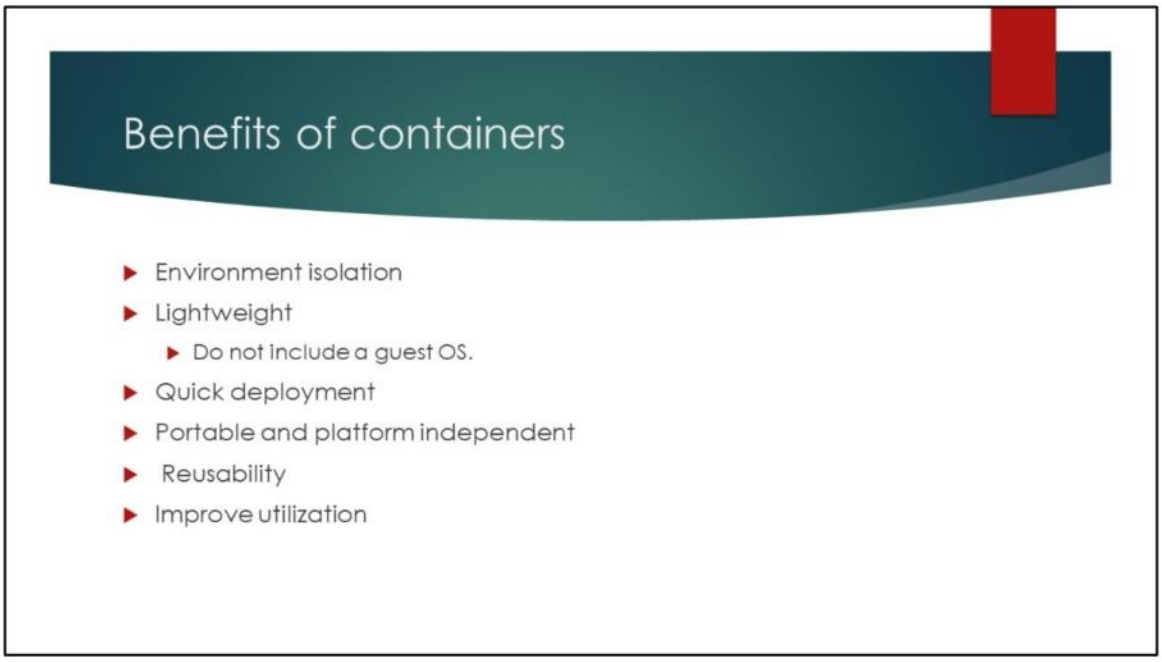
A container is an executable unit of software in which application code is packaged, along with its libraries and dependencies, in common ways so that it can be run anywhere, whether on a desktop, on-premises, or in the cloud. Containers are completely isolated environments. In other words, containers are technologies that allow you to package and isolate applications with their entire runtime environment—all of the files necessary to run.

Instead of virtualizing the underlying hardware, containers virtualize the operating system (typically Linux) so each individual container contains *only* the application and its libraries and dependencies. The absence of the guest OS is why containers are so lightweight and, thus, fast and portable.



Shipping containers serve as a useful analogy for software containers. The shipping industry originally lacked a standardized way to ship goods. This lack of standardization resulted in a less optimized shipping process. To remedy this situation, shipping containers were introduced to standardize the way things are packed and shipped.

Similarly, software containers are a standardization for how we package and ship software. All the dependencies needed to run an application are packaged with it, making it easy to move it around and run it in a variety of locations. This portability is incredibly useful in software development.



Environment isolation: A container works in a closed environment where changes made to the host OS or other applications do not affect the container.

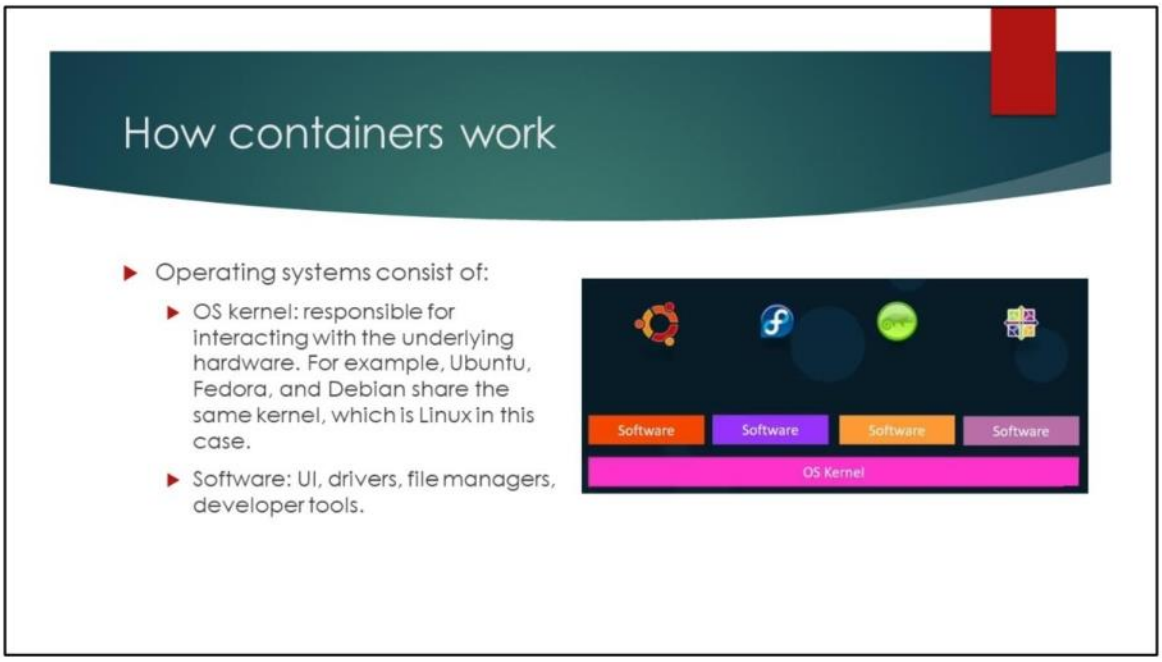
Lightweight: They share the host operating system kernel, eliminating the need for a full operating system instance per application. This makes container files small.

Quick deployment: Containers can be deployed quickly because there is no need for a full OS install or restart.

Portable and platform independent: Containers carry all their dependencies with them, meaning that software can be written once and then run without needing to be re-configured across laptops, cloud, and on-premises computing environments.

Reusability: The same container can be reused by multiple applications without the need to set up a full OS.

Improves utilization: Like VMs before them, containers improve the CPU and memory utilization of physical machines. Containers also enable a microservice architecture, which means that application components can be deployed and scaled more granularly.



To understand how containers work, let us revisit some basic concepts of operating systems. Operating systems, like Ubuntu, Fedora, Debian, etc. consist of two parts: OS kernel and a set of software.

The OS kernel is responsible for interacting with the underlying hardware. While the OS kernel remains the same, which is Linux in this case, it is the software above it that makes these operating systems different.

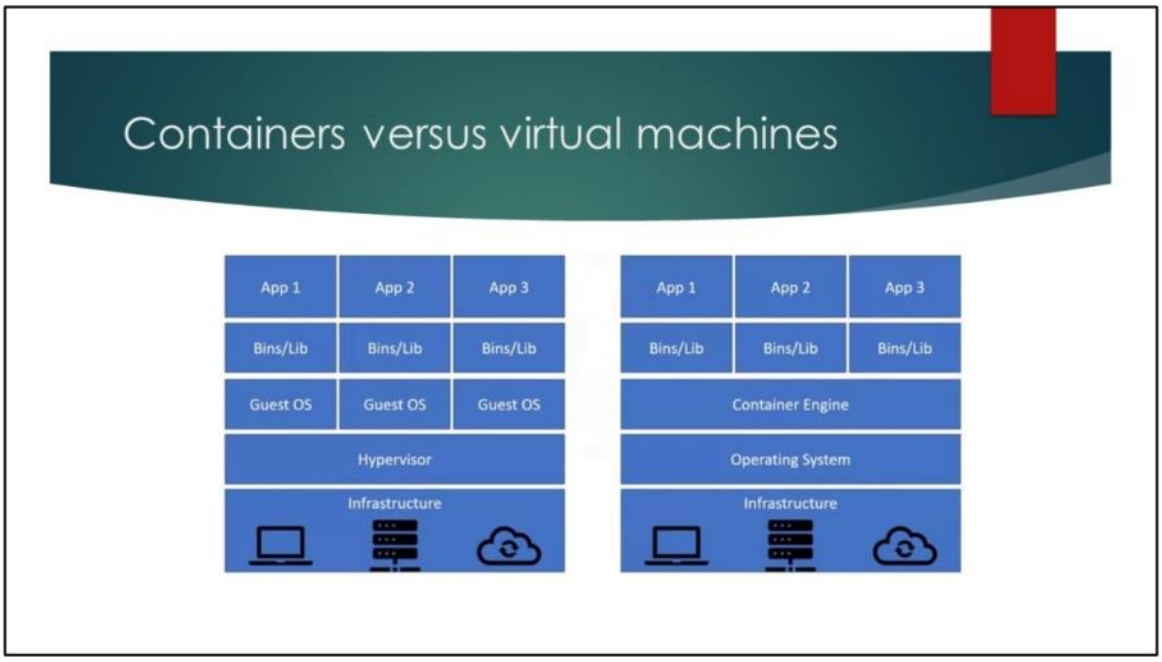
This software may consist of a different user interface, drivers, compilers, file managers, developer tools, etc. So, there is a common Linux kernel shared across all operating systems and some custom software that differentiates operating systems from each other.

Sharing the kernel

- Containers share the underlying kernel.
  - If the underlying OS is Ubuntu, container runtime can run a container based on another Linux distribution.
  - Each container only has the software that differentiates the distribution.
- Container are not meant to run different operating systems on the same hardware.

Containers share the underlying kernel. Let's say we have a system with an Ubuntu OS and a container engine installed on it. The container engine can run any flavor of OS on top of it as long as they are all based on the same kernel, in this case Linux. If the underlying OS is Ubuntu, the container engine can run a container based on another distribution, like Debian, Fedora, etc. Each container only has the additional software that makes these OSs different, the container engine utilizes the underlying kernel of the host OS.

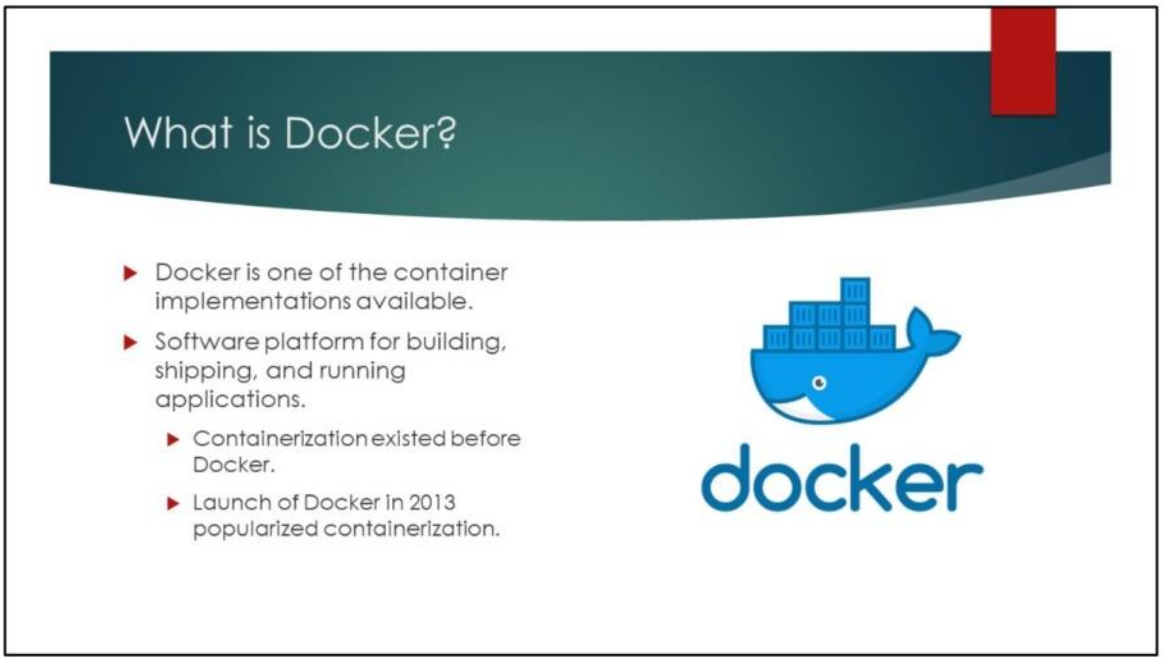
Unlike hypervisors, containers are not meant to virtualize and run different OSs on the same hardware. The main purpose of containers is to package and containerize applications and to ship them and run them.



With virtual machines, a hypervisor runs on top of your infrastructure; this hypervisor is responsible for creating virtual machines. It treats resources such as CPU and memory as a pool that can be utilized by the various virtual machines running on the hardware.

With containers, you still have physical infrastructure at the bottom of the stack. But instead of placing a hypervisor as the next layer to virtualize the hardware, you have an operating system. Above that is the container engine, which is responsible for running containers. This enables the infrastructure to have a single copy of the operating system, which saves a lot of space

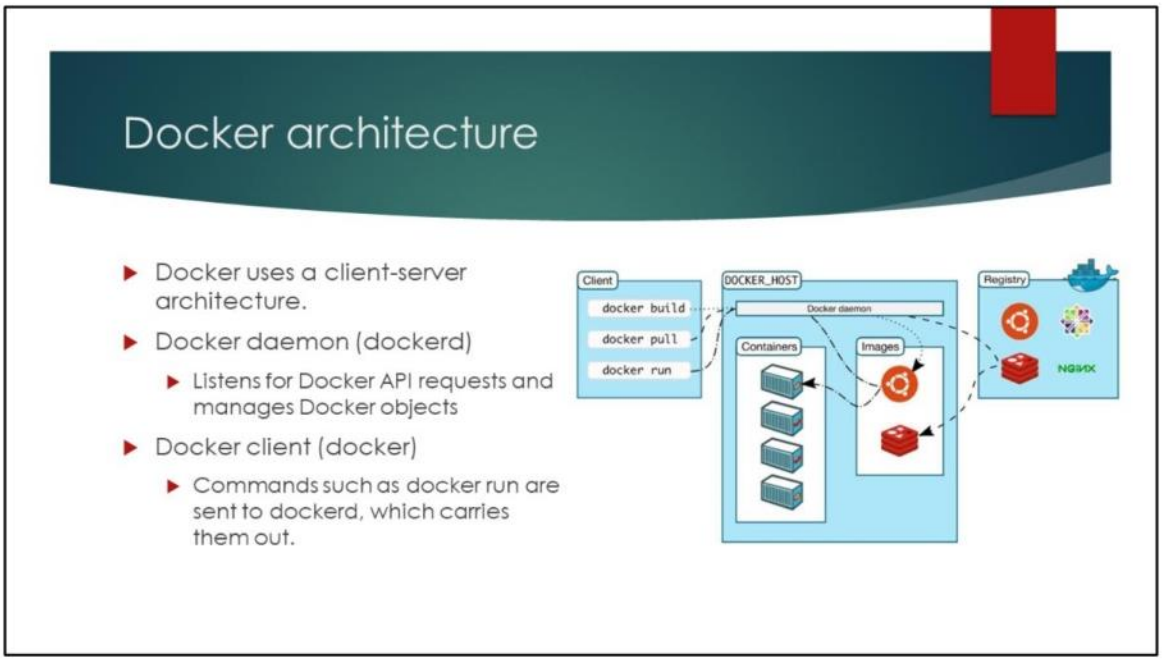
The clear benefit of containers is that you do not have to run a dedicated operating system instance for each virtual environment. Instead, one operating system is virtualized across all the containers.



Docker is one of the container implementations available for deployment. Docker is a software platform for building, running, and sharing applications as containers.

Although containerization technology existed long before Docker emerged on the scene, Docker has become inextricably linked to containers. This is because the launch of the Docker open source project in 2013 fueled the rapid growth in popularity of containers.

Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host.



Docker uses a client-server architecture.

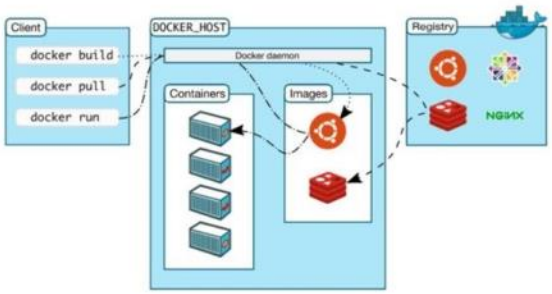
Client: The command-line tool (***docker***) is responsible for communicating with a server using a RESTful API to request operations.

Server: This service, which runs as a daemon on an operating system, does the heavy lifting of building, running, and downloading container images.

The daemon can run either on the same system as the ***docker*** client or remotely.

Docker architecture (continued)

- Docker core elements
  - Images
    - An *image* is a read-only template with instructions for creating a Docker container.
  - Docker registries
    - Stores Docker images. Docker Hub is a public registry that anyone can use.
  - Containers
    - A container is a runnable instance of an image.

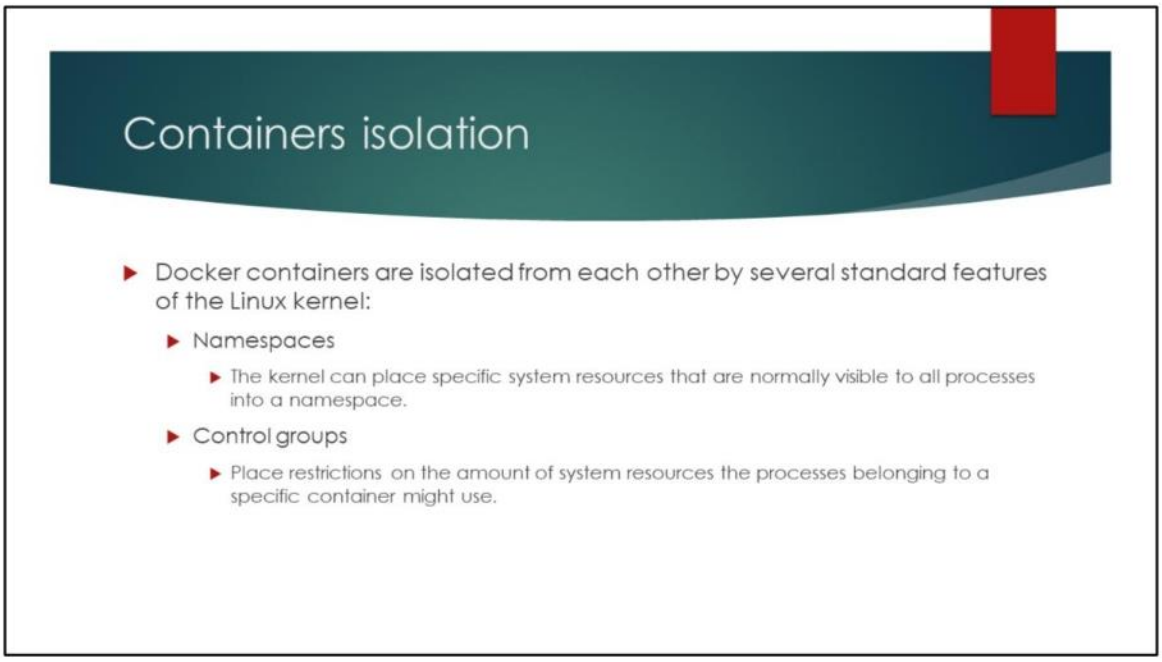


Docker depends on three major elements:

Images are read-only templates that contain a runtime environment that includes applications and libraries. Images are used to create containers. Images can be created, updated, or downloaded for immediate consumption.

Registries store Docker images. Docker Hub is a public registry that provides a large set of images developed by the community. Docker is configured to look for images on Docker Hub by default.

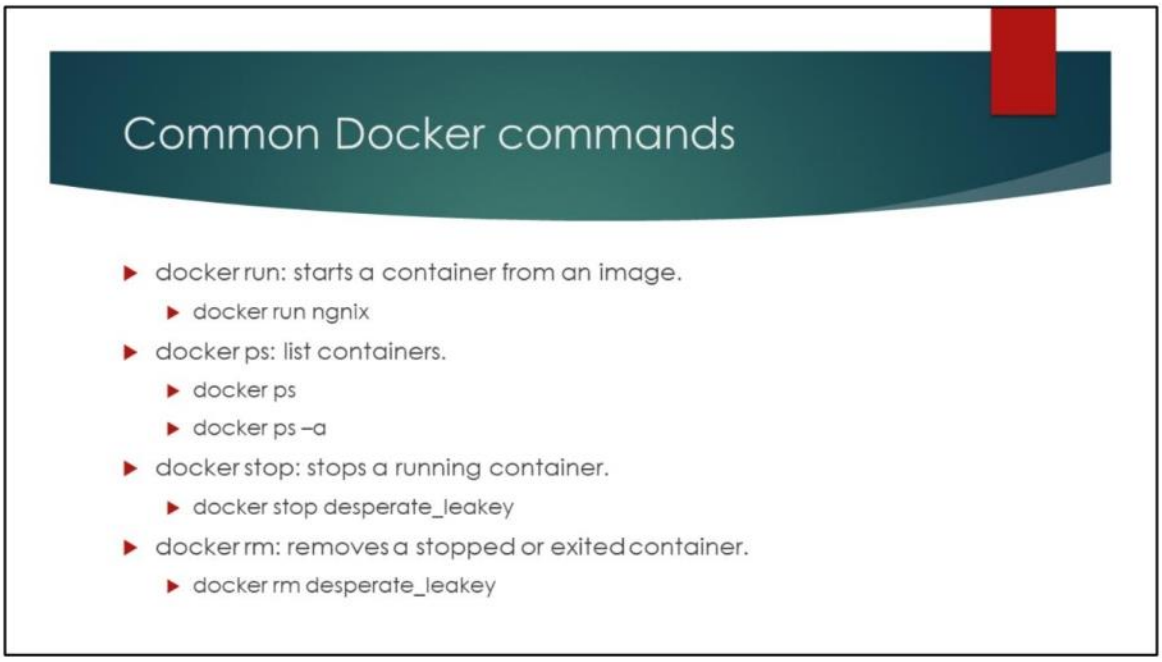
Containers are segregated environments for running applications isolated from other applications sharing the same host OS.



Containers created by Docker are isolated from each other by several standard features of the Linux kernel:

Namespaces: The kernel can place specific system resources that are normally visible to all processes into a namespace. Inside a namespace, only processes that are members of that namespace can see those resources.

Control groups ***(cgroups)*** place restrictions on the amount of system resources the processes belonging to a specific container might use. This keeps one container from using too many resources on the container host.

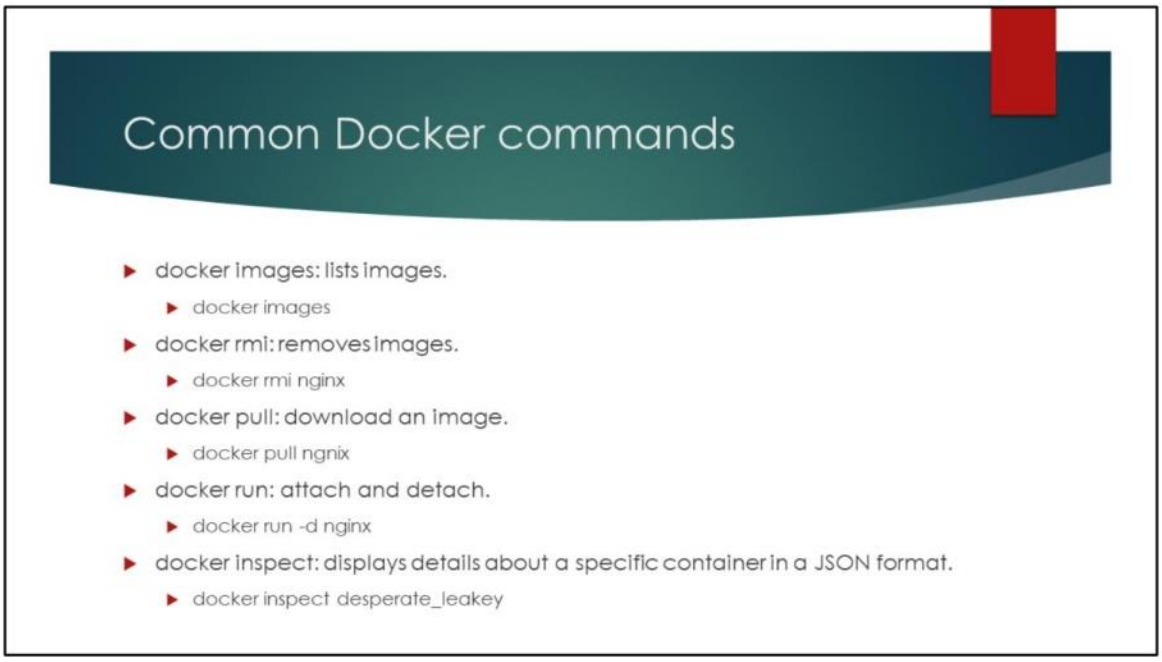


The docker run command is used to run a container from an image. For example, the command ***docker run nginx*** will run an instance of the nginx application on the Docker host if it already exists. If the image is not present on the host, it will pull the image down from the Docker hub.

The ***docker ps*** command lists all running containers and some basic information about them, such as, the container ID, the name of the image used to run the containers, the current status, and the name of the container. Each container automatically gets a random ID and name created for it. To see all containers running or not, use the ***-a*** option. This outputs all running, as well as, previously stopped or exited containers.

To stop a running container, use the ***docker stop*** command. You must provide either the container ID or the container name.

The ***docker rm*** command removes a stopped or exited container permanently.



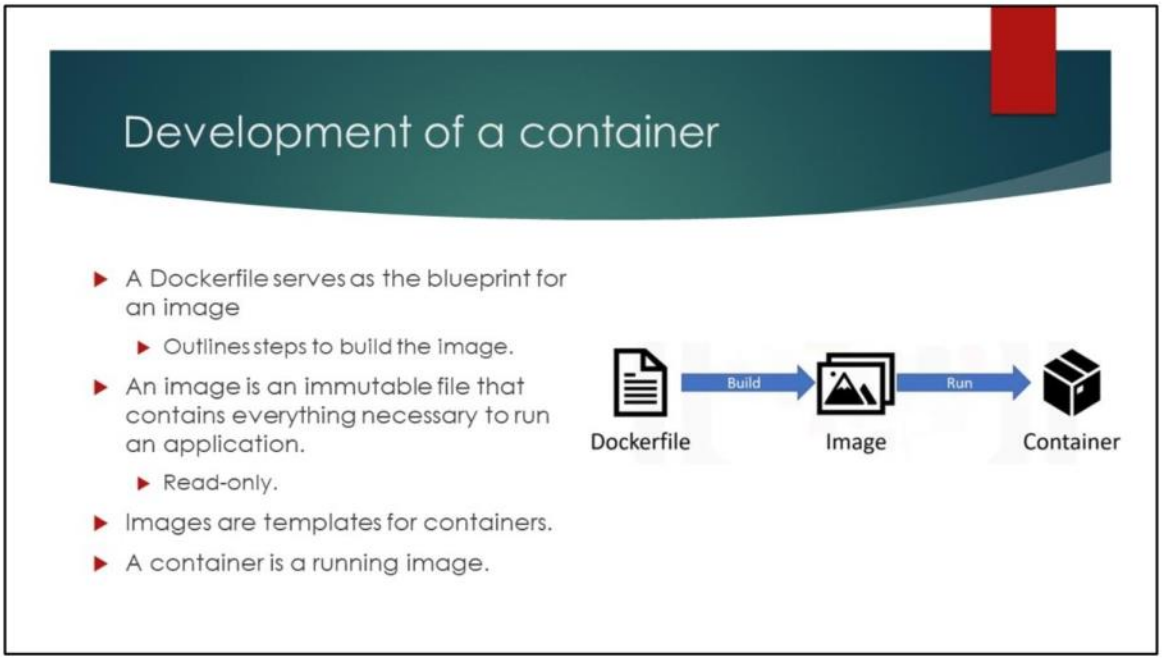
The ***docker images*** command displays a list of available images and their sizes.

To remove an image that you no longer plan to use, run the ***docker rmi*** command. You must stop and delete all dependent containers to remove an image.

If we simply want to download the image and keep it on the local host. So that when we run the docker run command, we don't have to wait for it to download. Use the ***docker pull*** command to only pull the image and not run the container.

By default, docker run command runs in the foreground or in an attached mode, meaning you will be attached to the console or the standard out of the Docker container, you will see the output of the process on your screen, you won't be able to do anything else until this Docker container stops. Another option is to run the Docker container in the detached mode by providing the ***-d*** option. This will run the Docker container in the background mode, and you will be back to your prompt immediately, the container will continue to run in the backend.

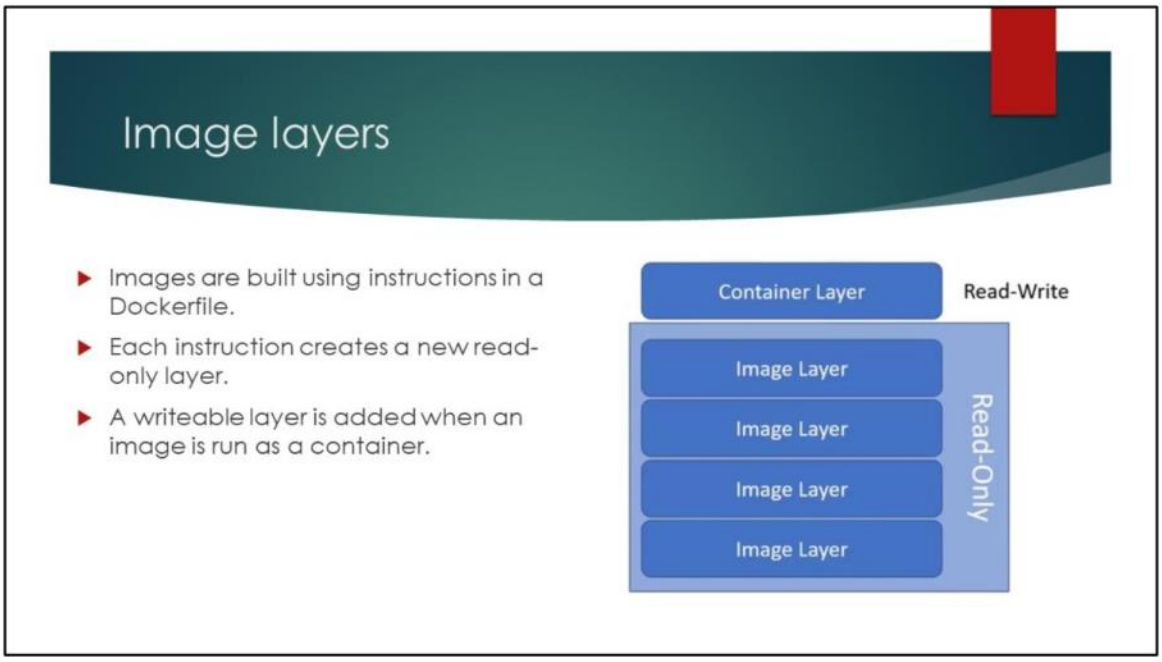
To see additional details about a specific container, use the ***docker inspect*** command to return all details of a container in a JSON format.



To create a new image, there are two approaches:

- **Using a running container**: A running container can be changed in place and its layers saved to create a new container image. This feature is provided by the ***docker commit*** command.

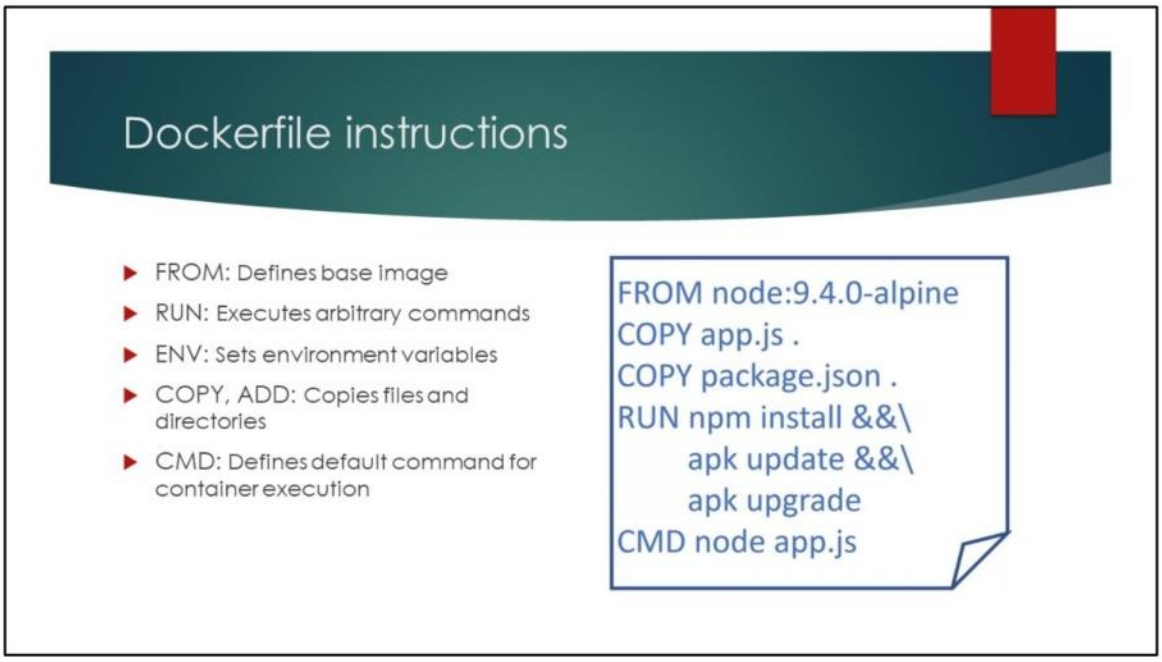
- **Using a Dockerfile**: Container images can be built from a base image using a set of steps called instructions. Each instruction creates a new layer on the image that is used to build the final container image. This is the suggested approach to building images, because it controls which files are added to each layer.



Docker can automatically build container images by using the instructions from a Dockerfile. A Dockerfile is simply a text file that contains all the commands a user would call on the command line to create the image.

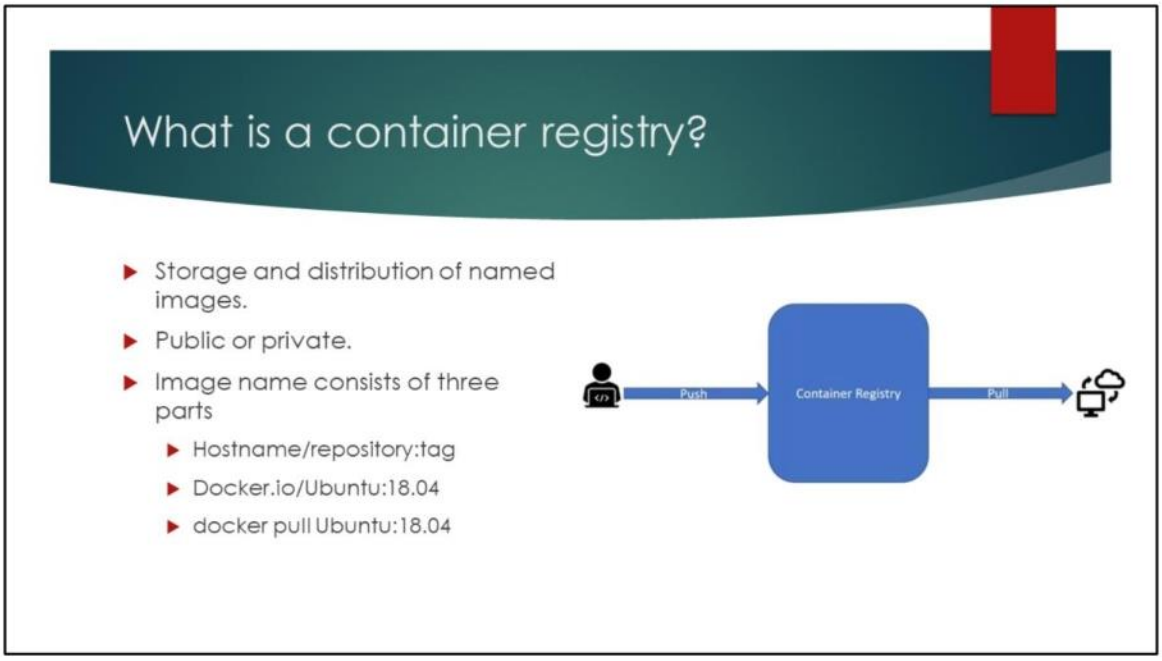
Images are then built layer by layer, with each Docker instruction creating a new layer on top of the existing layers. As Docker instructions are run, new read-only layers are created for the image. When all the Docker instructions from a Dockerfile have been executed, the image is built and complete.

Running an image results in a container. When we instantiate this image, we get a running container. At this point, a writeable container layer is added on top of the read-only layers. Containers are not immutable like images are.



Dockerfile is a text file that consists of a set of steps called instructions. Each instruction creates a new layer on the image that is used to build the final container image. A common convention is to make instructions all uppercase to improve visibility.

The first instruction must be a FROM instruction to specify the base image to build upon.
RUN executes commands in a new layer on top of the current image, then commits the results.
ENV is responsible for defining environment variables that will be available to the container.
COPY copies new files and directories and adds them to the container file system.
CMD defines the default command to execute when the container is created.

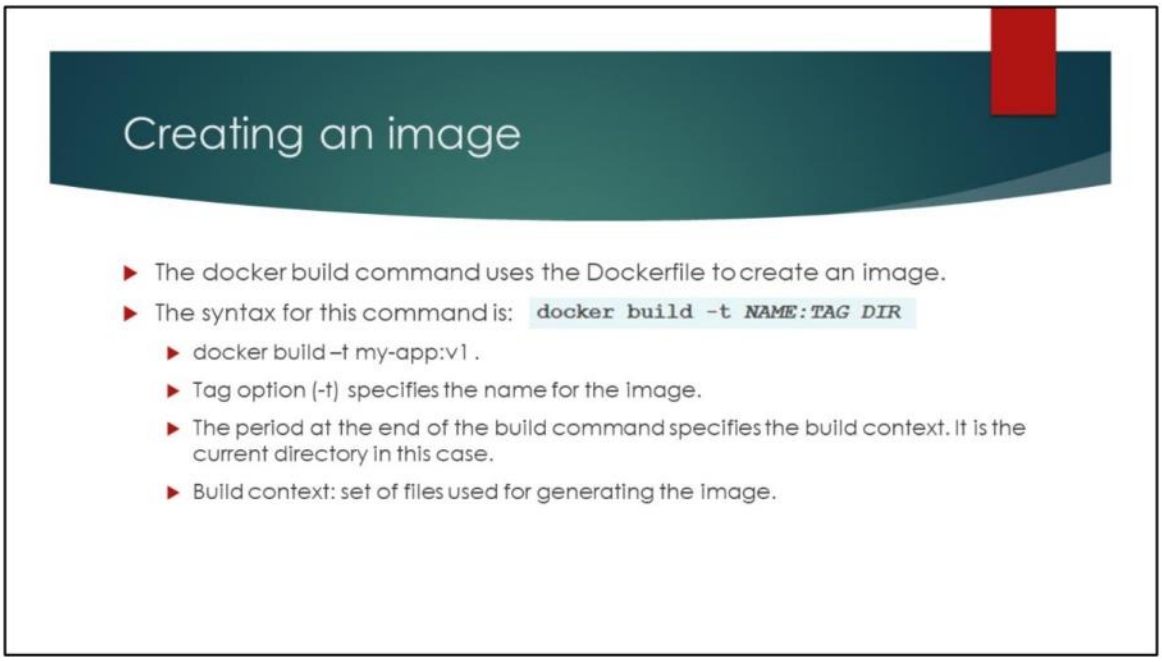


Where do we store and manage container images? The answer? Container registries. A container registry is used for the storage and distribution of named container images.

Registries can be public or private. When an image is pushed to a public registry, anyone can pull that image. Most enterprises will opt to use a private registry, which restricts access to the images to authorized users only.

When storing an image in a registry, we say that you "push" an image to the registry. Similarly, when retrieving an image from a registry, we say that you "pull" an image from a registry.

There's a specific format that is generally followed for image naming. The image name usually consists of three parts: the "hostname“, the "repository“, and the "tag“. The hostname identifies the registry to which this image should be pushed. For Docker Hub, the registry is "docker.io“. A repository is a group of related container images. So, the name of the application or service makes a good repository name. The tag provides information about a specific version or variant of an image. If the hostname is excluded, it is “Docker.io” by default.



The ***docker build*** command processes the Dockerfile and builds a new image based on the instructions it contains. The syntax for this command is as follows:

docker build -t NAME:TAG DIR

DIR is the path to the working directory. It can be the current directory as designated by a period (.) symbol. NAME:TAG is a name with a tag that is assigned to the new image. It is specified with the -t option. If the TAG is not specified, then the image is automatically tagged as latest.

Running a container

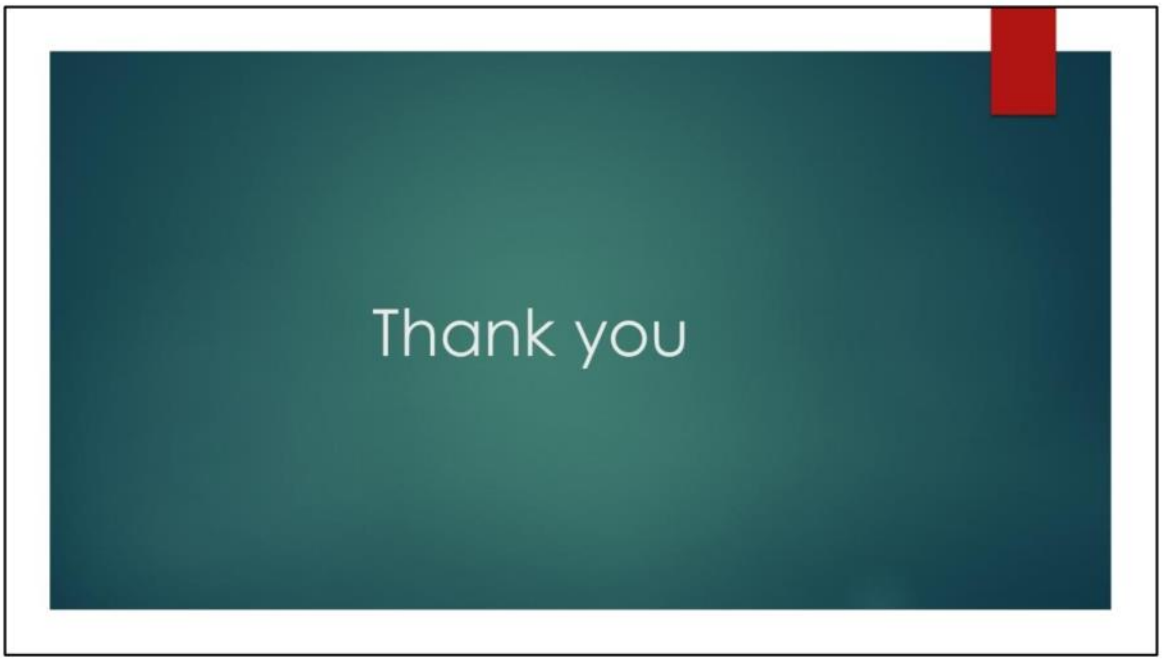
- The output of the docker build command is shown on the right.
- To verify that the image is created, use docker images command

```
$ docker images
REPOSITORY   TAG   IMAGE ID       CREATED          SIZE
my-app       v1    b8b15c59b352   2 minutes ago    78.2MB
```

- To run the image as a container, user the docker run command.

```
$ docker run my-app:v1
```

```
$ docker build -t my-app:v1 .
Sending build context to Docker daemon  15.36kB
Step 1/5 : FROM node:9.4.0-alpine
 ---> b5f94997f35f
Step 2/5 : COPY app.js .
 ---> Using cache
 ---> 3327e0636765
Step 3/5 : COPY package.json .
 ---> Using cache
 ---> f513959ca800
Step 4/5 : RUN npm install &&   apk update &&   apk upgrade
 ---> Using cache
 ---> b61f68e0c0fa
Step 5/5 : CMD node app.js
 ---> Running in a8f1c8d5c3c6
Removing intermediate container a8f1c8d5c3c6
 ---> b8b15c59b352
Successfully built b8b15c59b352
Successfully tagged my-app:v1
```


Thank you