**Docker Containers**

**What are containers?**

The industry standard today is to use Virtual Machines (VMs) to run software applications. VMs run applications inside a guest Operating System, which runs on virtual hardware powered by the server’s host OS.

VMs are great at providing full process isolation for applications: there are very few ways a problem in the host operating system can affect the software running in the guest operating system, and vice-versa. But this isolation comes at great cost — the computational overhead spent virtualizing hardware for a guest OS to use is substantial.

Containers take a different approach: by leveraging the low-level mechanics of the host operating system, containers provide most of the **isolation** of virtual machines at a **fraction of the computing power**.

Containers offer a **logical packaging** mechanism in which applications can be **abstracted** from the environment in which they actually run. This **decoupling** allows container-based applications to be deployed easily and consistently, regardless of whether the target environment is a private data center, the public cloud, or even a developer’s personal laptop. This gives developers the ability to create predictable environments that are isolated from the rest of the applications and can be run anywhere.

Q: What is Docker Container  
A: A Docker Container is a form of encapsulation to the application which holds all the dependencies which share the kernel with other containers in the duration of running the isolated processes on the host operating system. A Docker container can be created by creating a Docker image. These Docker images can be run after that using Docker commands. Docker containers are the instances of the Docker images at the runtime. Docker images can be stored in any public hosts or private hosts like Docker hub. Docker Image is a set of files which can be run in an isolated process.

 Interview Q&A

**Run your first container**

We’re going to start with checking that Docker is working correctly, and then we’re going to take a look at the basic Docker workflow: creating and managing containers. We’ll take a container through its typical lifecycle from creation to a managed state and then stop and remove it.

Firstly, let’s check that the docker **binary**exists and is functional:  
input :

$ sudo docker info

output :

Client:

Debug Mode: false

Server:

Containers: 2

Running: 0

Paused: 0

Stopped: 2

Images: 2

Server Version: 19.03.8

Storage Driver: overlay2

Backing Filesystem:

Supports d\_type: true

Native Overlay Diff: true

Logging Driver: json-file

Cgroup Driver: cgroupfs

Plugins:

Volume: local

Network: bridge host ipvlan macvlan null overlay

Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog

Swarm: inactive

Runtimes: runc

Default Runtime: runc

Init Binary: docker-init

containerd version: 7ad184331fa3e55e52b890ea95e65ba581ae3429

runc version: dc9208a3303feef5b3839f4323d9beb36df0a9dd

init version: fec3683

Security Options:

apparmor

seccomp

Profile: default

Kernel Version: 4.15.0-1057-aws

Operating System: Ubuntu 18.04.3 LTS

OSType: linux

Architecture: x86\_64

CPUs: 1

Total Memory: 983.7MiB

Name: ip-172-31-34-103

ID: ZXJP:HBON:RWBH:KERY:4CO3:VFSD:LLJV:T4LZ:RZXL:UOLK:2H7J:XVVH

Docker Root Dir: /var/lib/docker

Debug Mode: false

Registry: https://index.docker.io/v1/

Labels:

Experimental: false

Insecure Registries:

127.0.0.0/8

Live Restore Enabled: false

WARNING: No swap limit support

Here, we’ve passed the info command to the docker binary, which returns a list of any containers, any images (the building blocks Docker uses to build containers), the execution and storage drivers. Docker is using, and its basic configuration.

**Your Container**

Now let’s try and launch our first container with Docker. We’re going to use the docker run command to create a container. The docker run command provides all of the ”launch” capabilities for Docker. We’ll be using it a lot to create new containers.

**💡 Tips:**

* You can find a full list of the available Docker commands [**here**](https://docs.docker.com/engine/reference/commandline/cli/)or by typing **docker help**. You can also use the Docker man pages (e.g., man docker-run).

input :

$ sudo docker run -i -t ubuntu /bin/bash

output :

Unable to find image 'ubuntu:latest' locally

latest: Pulling from library/ubuntu

5bed26d33875: Pull complete

f11b29a9c730: Pull complete

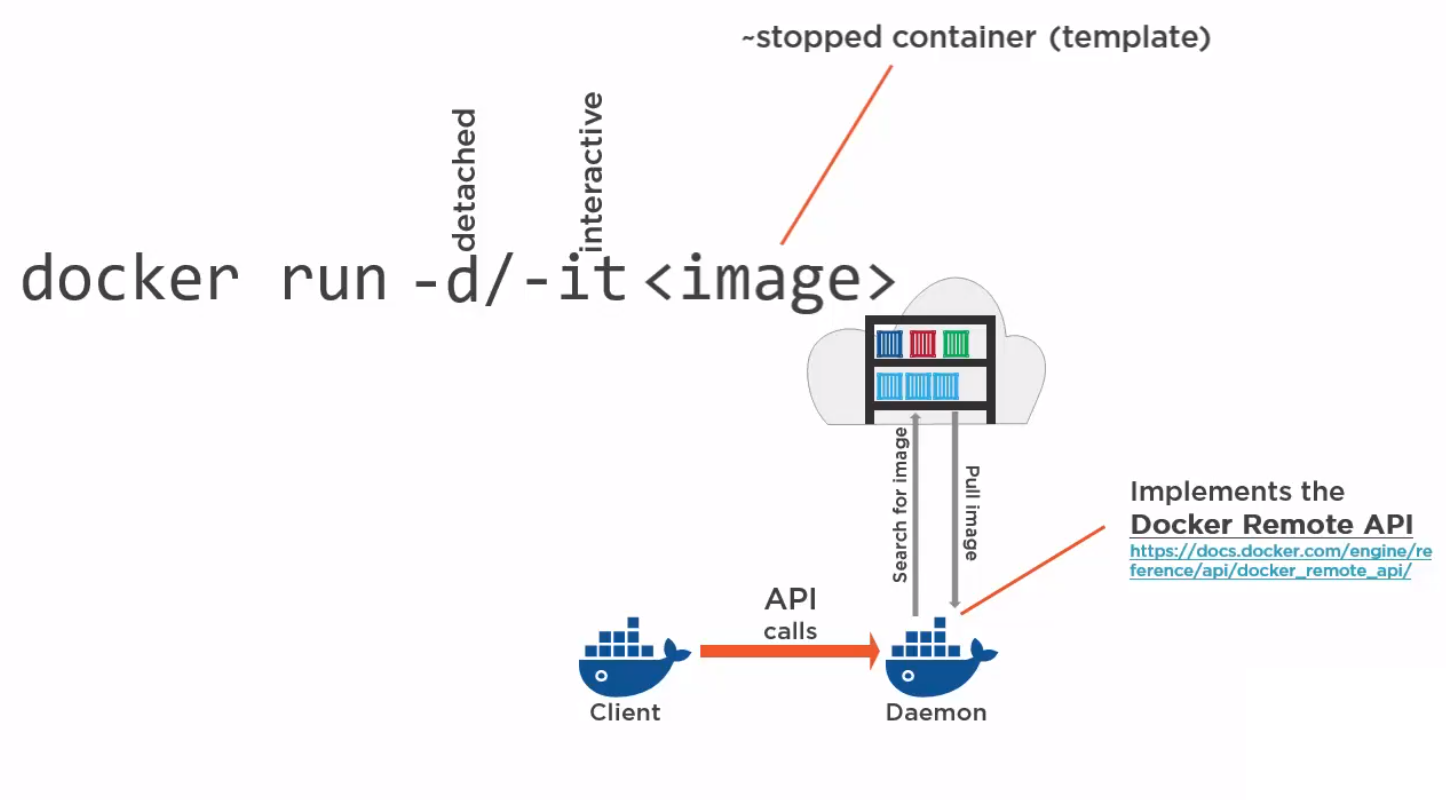
930bda195c84: Pull complete

78bf9a5ad49e: Pull complete

Digest: sha256:bec5a2727be7fff3d308193cfde3491f8fba1a2ba392b7546b43a051853a341d

Status: Downloaded newer image for ubuntu:latest

root@dc87c855a5ae:/#



**Docker Run**

First, we told Docker to run a command using docker run. We passed it two command line flags: **-i** and **-t.**

The -i flag keeps STDIN open from the container, even if we’re not attached to it. This persistent standard input is one half of what we need for an interactive shell.

The -t flag is the other half and tells Docker to assign a pseudo -tty to the container we’re about to create. This provides us with an interactive shell in the new container. This line is the base configuration needed to create a container with which we plan to interact on the command line rather than run as a daemonized service.

**💡 Tips:**

* You can find a full list of the available Docker run flags by typing **docker run help**. You can also use the Docker man pages (e.g., man docker-run).

We told Docker which image to use to create a container, in this case, the ubuntu image. The ubuntu image is a stock image, also known as a ”base” image, provided by Docker, Inc., on the Docker Hub registry.

So what was happening in the background here? Firstly, Docker checked locally for the ubuntu image. If it can’t find the image on our local Docker host, it will reach out to the Docker Hub registry run by Docker, Inc., and look for it there. Once Docker had found the image, it downloaded the image and stored it on the local host.

Docker then used this image to create a new container inside a filesystem. The container has a network, IP address, and a bridge interface to talk to the local host. Finally, we told Docker which command to run in our new container, in this case launching a Bash shell with the /bin/bash command.

**Working with Container**

We are now logged into a new container, with the catchy ID of 1946f83d53a0, as the root user. This is a fully-fledged Ubuntu host, and we can do anything we would like in it. Let’s start by asking for its hostname.

root@1946f83d53a0:/# hostname

1946f83d53a0

root@1946f83d53a0:/#

Installing a package in container

root@1946f83d53a0:/# apt-get update && apt-get install nano

You can keep playing with the container for as long as you like. When you’re done, type exit, and you’ll return to the command prompt of your Ubuntu host.

root@1946f83d53a0:/# exit

exit

ubuntu@ip-172-31-34-103:~$

It has now stopped running. The container only runs for as long as the command we specified, /bin/bash, is running. Once we exited the container, that command ended, and the container was stopped.

**docker ps -a command**

The container still exists; we can show a list of current containers using the docker ps -a command.

ubuntu@ip-172-31-34-103:~$ docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES 1946f83d53a0 ubuntu "/bin/bash"14 hrs. ago Exited (0)13 hours ago tender\_visves varaya

By default, when we run just docker ps, we will only see the running containers. When we specify the -a flag, the docker ps -a command will show us all containers, both stopped and running.

We learn quite a bit of information about our container: its ID, the image used to create it, the command it last ran, when it was created, and its exit status (in our case, 0, because it was exited normally using the exit command). We can also see that each container has a name.

**Tips:**

There are three ways containers can be identified:

* a short UUID (1946f83d53a0),
* a longer UUID (like 1946f83d53a021548190053c4053f93fda09c3e5fa6bd670acbc7697a96b9961), and
* a name (tender\_visvesvaraya).

**Container naming**

Docker will automatically generate a name at random for each container we create. We see that the container we’ve created is called tender\_visvesvaraya. If we want to specify a particular container name in place of the automatically generated name, we can do so using the --name flag.

sudo docker run --name clarusway -i -t ubuntu /bin/bash

root@05c455bdca71:/#

You can check it in another terminal and verify the following output.

ubuntu@ip-172-31-34-103:~$ sudo docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

2f5a96f3ccde ubuntu "/bin/bash"5 hours ago Exited (0)5 hours ago clarusway

ubuntu@ip-172-31-34-103:~$

This would create a new container called clarusway. A valid container name can contain the following characters: a to z, A to Z, the digits 0 to 9, the underscore, period, and dash (or, expressed as a regular expression: [a-zA-Z0-9\_.-]).

We can use the container name in place of the container ID in most Docker commands. It is also much easier to remember a specific container name than a container ID or even a random name.

Names are unique. If we try to create two containers with the same name, the command will fail. We need to delete the previous container with the same name before we can create a new one. We can do so with the docker rm command.