# Docker:

### Docker concepts

Docker is a platform for developers and sysadmins to develop, deploy, and run applications with containers. The use of Linux containers to deploy applications is called containerization. Containers are not new, but their use for easily deploying applications is. Containerization is increasingly popular because containers are:

* Flexible: Even the most complex applications can be containerized.
* Lightweight: Containers leverage and share the host kernel.
* Interchangeable: You can deploy updates and upgrades on-the-fly.
* Portable: You can build locally, deploy to the cloud, and run anywhere.
* Scalable: You can increase and automatically distribute container replicas.
* Stackable: You can stack services vertically and on-the-fly.

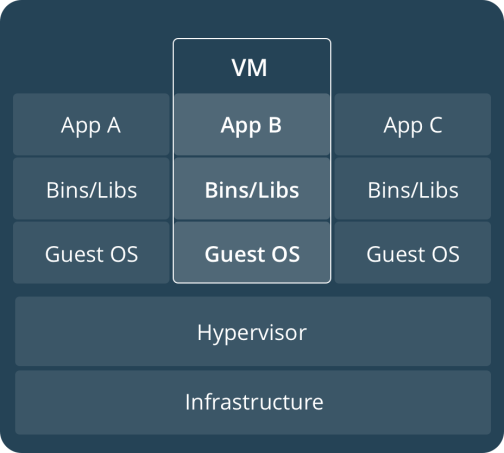
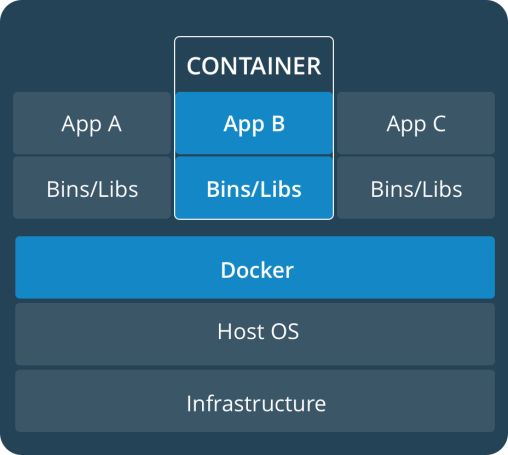
### Images and containers

A container is launched by running an image. An image is an executable package that includes everything needed to run an application--the code, a runtime, libraries, environment variables, and configuration files.

A container is a runtime instance of an image--what the image becomes in memory when executed (that is, an image with state, or a user process). You can see a list of your running containers with the command, docker ps, just as you would in Linux.

Features of Docker:

1. Docker has the ability to reduce the size of development by providing a smaller footprint of the operating system via containers.
2. With containers, it becomes easier for teams across different units, such as development, QA and Operations to work seamlessly across applications.
3. You can deploy Docker containers anywhere, on any physical and virtual machines and even on the cloud.
4. Since Docker containers are pretty lightweight, they are very easily scalable.



## List Docker CLI commands

docker

docker container --help

## Display Docker version and info

docker --version

docker version

docker info

## Execute Docker image

docker run hello-world

## List Docker images

docker image ls

## List Docker containers (running, all, all in quiet mode)

docker container ls

docker container ls --all

docker container ls -aq

### Get Started, Part 2: Containers

#### Your new development environment

In the past, if you were to start writing a Python app, your first order of business was to install a Python runtime onto your machine. But, that creates a situation where the environment on your machine needs to be perfect for your app to run as expected, and also needs to match your production environment.

With Docker, you can just grab a portable Python runtime as an image, no installation necessary. Then, your build can include the base Python image right alongside your app code, ensuring that your app, its dependencies, and the runtime, all travel together.

These portable images are defined by something called a Dockerfile.

#### Define a container with Dockerfile

Dockerfile defines what goes on in the environment inside your container. Access to resources like networking interfaces and disk drives is virtualized inside this environment, which is isolated from the rest of your system, so you need to map ports to the outside world, and be specific about what files you want to “copy in” to that environment. However, after doing that, you can expect that the build of your app defined in this Dockerfile behaves exactly the same wherever it runs.

#### Dockerfile

Create an empty directory on your local machine. Change directories (cd) into the new directory, create a file called Dockerfile, copy-and-paste the following content into that file, and save it. Take note of the comments that explain each statement in your new Dockerfile.

Sample app:

<https://docs.docker.com/get-started/part2/>

## Build the app

Now run the build command. This creates a Docker image, which we’re going to name using the --tagoption. Use -t if you want to use the shorter option.

docker build --tag=friendlyhello .

Where is your built image? It’s in your machine’s local Docker image registry:

$ docker image ls

REPOSITORY TAG IMAGE ID

friendlyhello latest 326387cea398

## Run the app

Run the app, mapping your machine’s port 4000 to the container’s published port 80 using -p:

docker run -p 4000:80 friendlyhello

Now let’s run the app in the background, in detached mode:

docker run -d -p 4000:80 friendlyhello

You should see a message that Python is serving your app at http://0.0.0.0:80. But that message is coming from inside the container, which doesn’t know you mapped port 80 of that container to 4000, making the correct URL http://localhost:4000.

Go to that URL in a web browser to see the display content served up on a web page.

Here is a list of the basic Docker commands from this page, and some related ones if you’d like to explore a bit before moving on

## Share your image

To demonstrate the portability of what we just created, let’s upload our built image and run it somewhere else. After all, you need to know how to push to registries when you want to deploy containers to production.

A registry is a collection of repositories, and a repository is a collection of images—sort of like a GitHub repository, except the code is already built. An account on a registry can create many repositories. The docker CLI uses Docker’s public registry by default.

docker build -t friendlyhello . # Create image using this directory's Dockerfile

docker run -p 4000:80 friendlyhello # Run "friendlyhello" mapping port 4000 to 80

docker run -d -p 4000:80 friendlyhello # Same thing, but in detached mode

docker container ls # List all running containers

docker container ls -a # List all containers, even those not running

docker container stop <hash> # Gracefully stop the specified container

docker container kill <hash> # Force shutdown of the specified container

docker container rm <hash> # Remove specified container from this machine

docker container rm $(docker container ls -a -q) # Remove all containers

docker image ls -a # List all images on this machine

docker image rm <image id> # Remove specified image from this machine

docker image rm $(docker image ls -a -q) # Remove all images from this machine

docker login # Log in this CLI session using your Docker credentials

docker tag <image> username/repository:tag # Tag <image> for upload to registry

docker push username/repository:tag # Upload tagged image to registry

docker run username/repository:tag # Run image from a registry

# docker push gautamdeka3/helloworld:tagname

# Get Started, Part 3: Services

#### About services

In a distributed application, different pieces of the app are called “services”. For example, if you imagine a video sharing site, it probably includes a service for storing application data in a database, a service for video transcoding in the background after a user uploads something, a service for the front-end, and so on.

Services are really just “containers in production.” A service only runs one image, but it codifies the way that image runs—what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on. Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.

Luckily it’s very easy to define, run, and scale services with the Docker platform -- just write a docker-compose.yml file.

### Your first docker-compose.yml file

A docker-compose.yml file is a YAML file that defines how Docker containers should behave in production.

docker-compose.yml

Save this file as docker-compose.yml wherever you want. Be sure you have pushed the image you created in Part 2 to a registry, and update this .yml by replacing username/repo:tag with your image details.

version: "3"

services:

web:

# replace username/repo:tag with your name and image details

image: username/repo:tag

deploy:

replicas: 5

resources:

limits:

cpus: "0.1"

memory: 50M

restart\_policy:

condition: on-failure

ports:

- "4000:80"

networks:

- webnet

networks:

webnet:

This docker-compose.yml file tells Docker to do the following:

* Pull [the image we uploaded in step 2](https://docs.docker.com/get-started/part2/) from the registry.
* Run 5 instances of that image as a service called web, limiting each one to use, at most, 10% of a single core of CPU time (this could also be e.g. “1.5” to mean 1 and half core for each), and 50MB of RAM.
* Immediately restart containers if one fails.
* Map port 4000 on the host to web’s port 80.
* Instruct web’s containers to share port 80 via a load-balanced network called webnet. (Internally, the containers themselves publish to web’s port 80 at an ephemeral port.)
* Define the webnet network with the default settings (which is a load-balanced overlay network).

# Run your new load-balanced app

To recap, while typing docker run is simple enough, the true implementation of a container in production is running it as a service. Services codify a container’s behavior in a Compose file, and this file can be used to scale, limit, and redeploy our app. Changes to the service can be applied in place, as it runs, using the same command that launched the service: docker stack deploy.

Some commands to explore at this stage:

docker stack ls # List stacks or apps

docker stack deploy -c <composefile> <appname> # Run the specified Compose file

docker service ls # List running services associated with an app

docker service ps <service> # List tasks associated with an app

docker inspect <task or container> # Inspect task or container

docker container ls -q # List container IDs

docker stack rm <appname> # Tear down an application

docker swarm leave --force # Take down a single node swarm from the manager

# Get Started, Part 4: Swarms

<https://docs.docker.com/get-started/part4/>

#### Introduction

In part 3, you took an app you wrote in part 2, and defined how it should run in production by turning it into a service, scaling it up 5x in the process.

Here in part 4, you deploy this application onto a cluster, running it on multiple machines. Multi-container, multi-machine applications are made possible by joining multiple machines into a “Dockerized” cluster called a swarm.

In part 4 you learned what a swarm is, how nodes in swarms can be managers or workers, created a swarm, and deployed an application on it. You saw that the core Docker commands didn’t change from part 3, they just had to be targeted to run on a swarm master. You also saw the power of Docker’s networking in action, which kept load-balancing requests across containers, even though they were running on different machines. Finally, you learned how to iterate and scale your app on a cluster.

Here are some commands you might like to run to interact with your swarm and your VMs a bit:

docker-machine create --driver virtualbox myvm1 # Create a VM (Mac, Win7, Linux)

docker-machine create -d hyperv --hyperv-virtual-switch "myswitch" myvm1 # Win10

docker-machine env myvm1 # View basic information about your node

docker-machine ssh myvm1 "docker node ls" # List the nodes in your swarm

docker-machine ssh myvm1 "docker node inspect <node ID>" # Inspect a node

docker-machine ssh myvm1 "docker swarm join-token -q worker" # View join token

docker-machine ssh myvm1 # Open an SSH session with the VM; type "exit" to end

docker node ls # View nodes in swarm (while logged on to manager)

docker-machine ssh myvm2 "docker swarm leave" # Make the worker leave the swarm

docker-machine ssh myvm1 "docker swarm leave -f" # Make master leave, kill swarm

docker-machine ls # list VMs, asterisk shows which VM this shell is talking to

docker-machine start myvm1 # Start a VM that is currently not running

docker-machine env myvm1 # show environment variables and command for myvm1

eval $(docker-machine env myvm1) # Mac command to connect shell to myvm1

& "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env myvm1 | Invoke-Expression # Windows command to connect shell to myvm1

docker stack deploy -c <file> <app> # Deploy an app; command shell must be set to talk to manager (myvm1), uses local Compose file

docker-machine scp docker-compose.yml myvm1:~ # Copy file to node's home dir (only required if you use ssh to connect to manager and deploy the app)

docker-machine ssh myvm1 "docker stack deploy -c <file> <app>" # Deploy an app using ssh (you must have first copied the Compose file to myvm1)

eval $(docker-machine env -u) # Disconnect shell from VMs, use native docker

docker-machine stop $(docker-machine ls -q) # Stop all running VMs

docker-machine rm $(docker-machine ls -q) # Delete all VMs and their disk images

# Get Started, Part 5: Stacks

<https://docs.docker.com/get-started/part5/>

Selenium Grid using Docker:

<https://github.com/SeleniumHQ/docker-selenium>