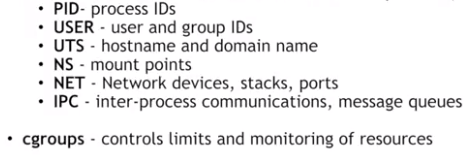
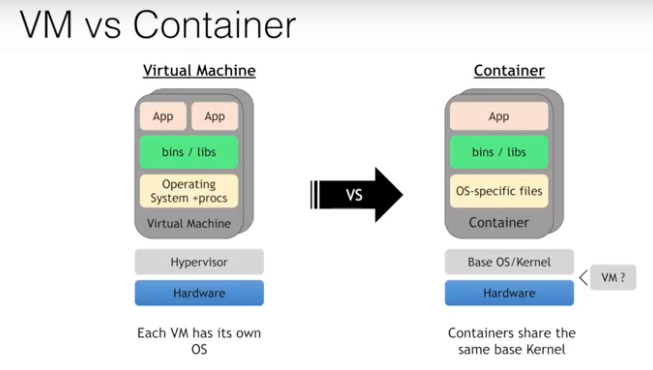
Docker Cognitive Class Notes:

# What are containers?

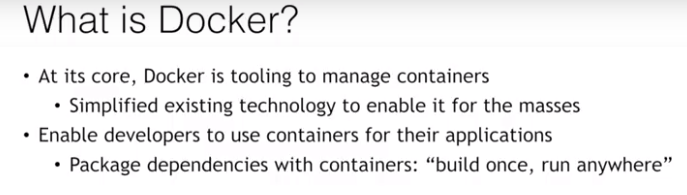
* A group of processes than run in isolation.
* All process must be able to run on a shared kernel.

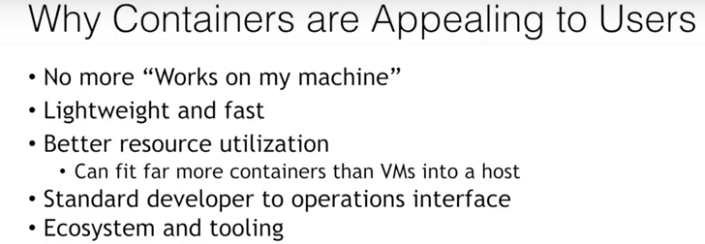
Each container has its own set of namespaces (That provides the isolated view)





Containers don’t replace virtual machines but can be run on top of VMs and complement them.



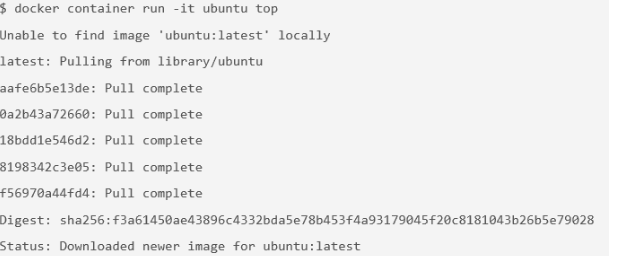


# Lab 1 – Run your container

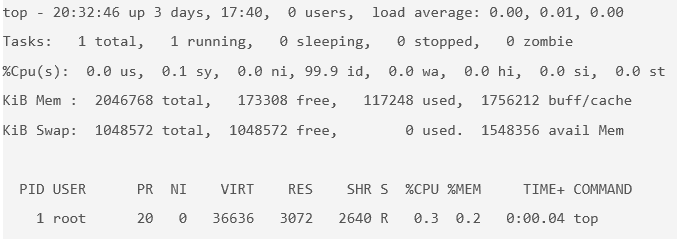
## Run a container:

$docker container run -it ubuntu top

You use the docker container run command to run a container with the Ubuntu image by using the top command.

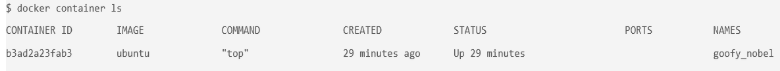


The docker run command first starts a docker pull to download the Ubuntu image onto your host. After it is downloaded, it will start the container. The output for the running container should look like this:



top is a Linux utility that prints the processes on a system and orders them by resource consumption. Notice that there is only a single process in this output: it is the top process itself. You don't see other processes from the host in this list because of the PID namespace isolation.

docker container ls - get the ID of the running container that you just created



Use that container ID to run bash inside that container by using the docker container exec command. Because you are using bash and want to interact with this container from your terminal, use the -it flag to run using interactive mode while allocating a psuedo-terminal:

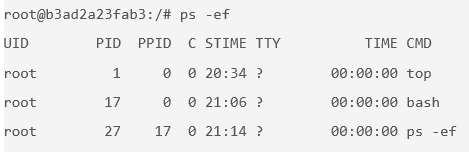
$ docker container exec -it b3ad2a23fab3 bash

root@b3ad2a23fab3:/#

You just used the docker container exec command to enter the container's namespaces with the bash process. Using docker container exec with bash is a common way to inspect a Docker container.

From the same terminal, inspect the running processes:

$ ps -ef



You should see only the top process, bash process, and your ps process.

For comparison, exit the container and run ps -ef or top on the host. These commands will work on Linux or Mac. For Windows, you can inspect the running processes by using tasklist.

root@b3ad2a23fab3:/# exit

exit

$ ps -ef

# Lots of processes!

## Run multiple containers

1. Explore the [Docker Store](https://store.docker.com/).
2. Run an NGINX server by using the [official NGINX image](https://store.docker.com/images/nginx) from the Docker Store:

$ docker container run --detach --publish 8080:80 --name nginx nginx



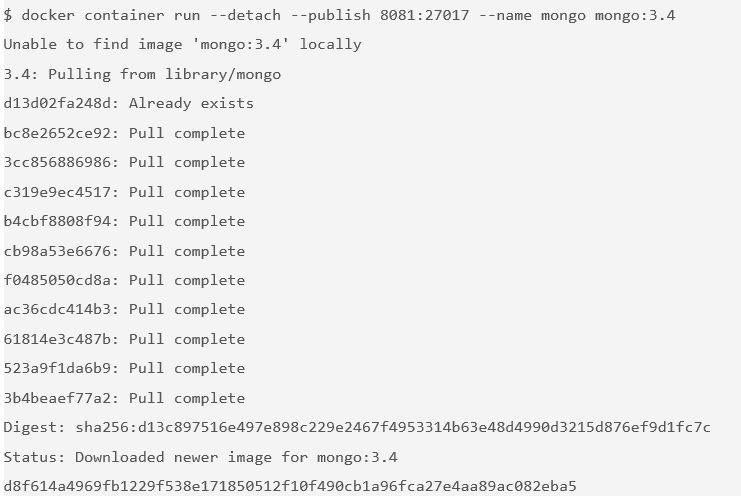
You are using a couple of new flags here. The --detach flag will run this container in the background. The publish flag publishes port 80 in the container (the default port for NGINX) by using port 8080 on your host. Remember that the NET namespace gives processes of the container their own network stack. The –publish flag is a feature that can expose networking through the container onto the host.

You are also specifying the --name flag, which names the container. Every container has a name. If you don't specify one, Docker will randomly assign one for you. Specifying your own name makes it easier to run subsequent commands on your container because you can reference the name instead of the id of the container. For example, you can specify docker container inspect nginx instead of docker container inspect 5e1.

NGINX is a lightweight web server. You can access it on port 8080 on your localhost.

1. Access the NGINX server on <http://localhost:8080>.
2. Run a MongoDB server. You will use the [official MongoDB image](https://store.docker.com/images/mongo) from the Docker Store. Instead of using the latest tag (which is the default if no tag is specified), use a specific version of the Mongo image: 3.4.

$ docker container run --detach --publish 8081:27017 --name mongo mongo:3.4



Again, because this is the first time you are running a Mongo container, pull the Mongo image from the Docker Store. You use the --publish flag to expose the 27017 Mongo port on your host. You must use a port other than 8080 for the host mapping because that port is already exposed on your host.

1. Access http://localhost:8081 to see some output from Mongo.
2. Check your running containers:

$ docker container ls



One thing you might notice is that the Mongo container is running the docker-entrypoint command. This is the name of the executable that is run when the container is started. The Mongo image requires some prior configuration before kicking off the DB process. You can see exactly what the script does by looking at it on [GitHub](https://github.com/docker-library/mongo/blob/master/3.0/docker-entrypoint.sh).

## Remove the containers

1. Get a list of the running containers:

$ docker container ls

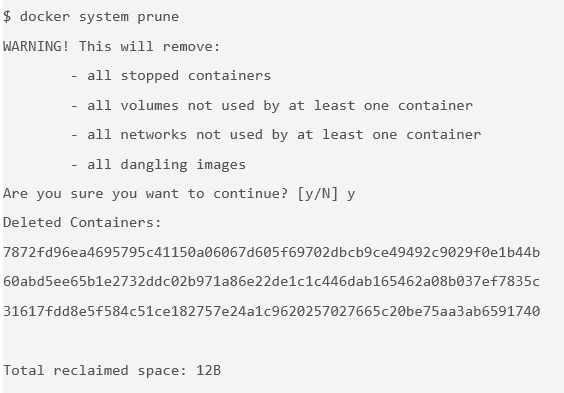
1. Stop the containers by running this command for each container in the list:

$ docker container stop [container id]

You can also use the names of the containers that you specified before

1. Remove the stopped containers. The following command removes any stopped containers, unused volumes and networks, and dangling images:

$ docker system prune



## Lab 1 – Quiz

1. Containers achieve isolation because of what feature in the Linux kernel?

**Namespaces.**

1. What is the difference between a Docker container and a Docker image?

**An image is the blueprint for spinning up containers. An image is a TAR of a file system, and a container is a file system plus a set of processes running in isolation.**

1. Control groups (cgroups) limit and monitor resources.

**True.**

1. Which statement is not true about Docker?

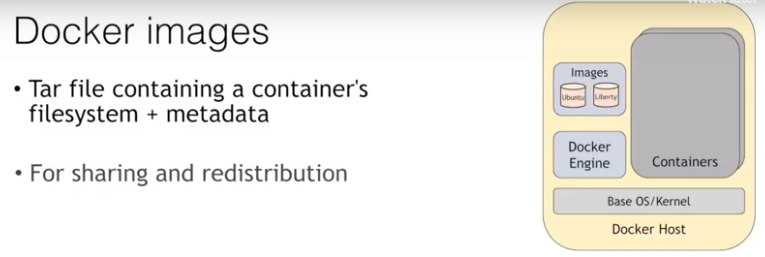
**Docker invented containers and Linux namespaces.**

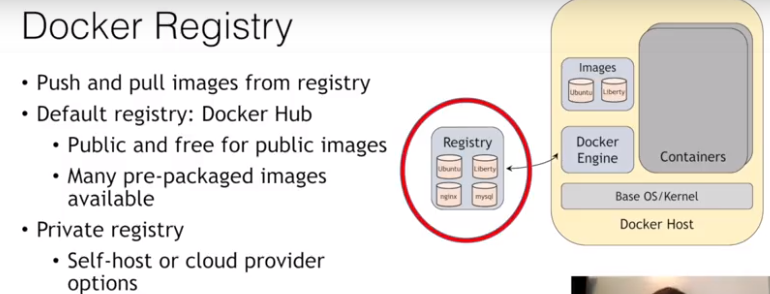
1. What tool makes it possible to run Docker containers on operating systems other than Linux?

**LinuxKit**

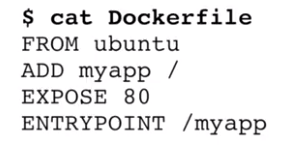
1. Top of Form
2. Top of Form

# Docker Images









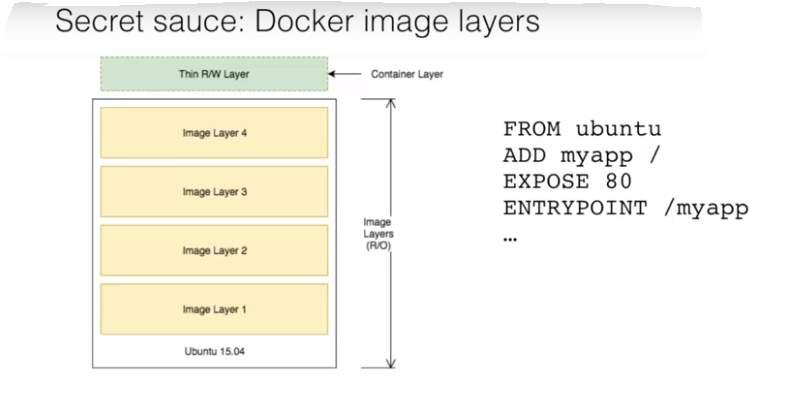
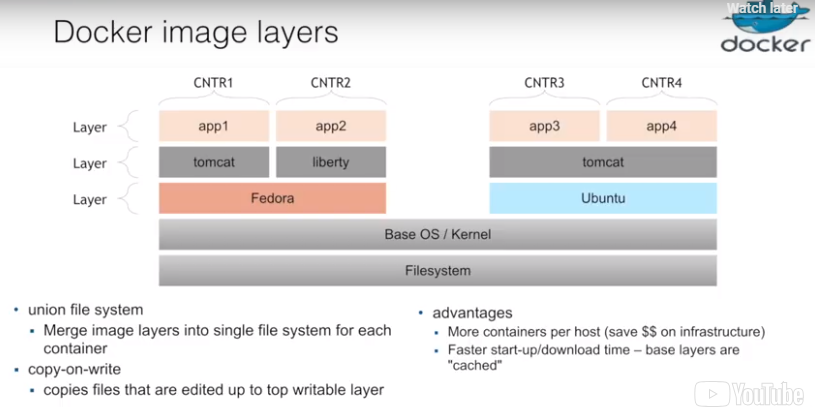


Image layers are cached and every layer is built on top of previous layer. If we change the last line of DOCKERFILE, the docker engine will reuse the other layers on top from cache and will only rebuild the last layer.

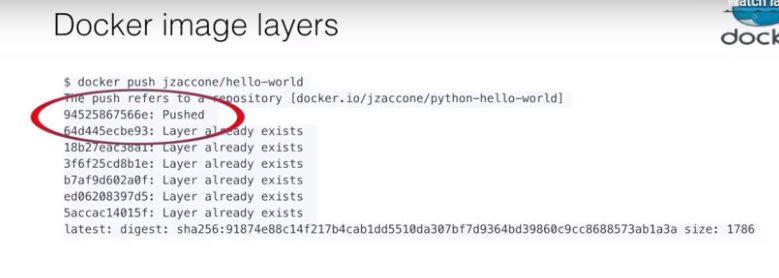
To optimize the caching for these layers, we need to change the DOCKERFILE in such a way that the layers that change the most are located at the last (e.g. put the source code at end since it changes every day).



app1 and app2 both run on the same host machine. We can see that both use the same underlying layer Fedora. How does the container manages to use the same underlying layer and not create a new layer for two separate apps? The same is the case with app3 and app4 which share tomcat and Ubuntu.

The union file system handles this by merging image layers into single file system for each container.

copy-on-write is used such that if for an application if the underlying layer has to be modified, the common shared layer is kept read only but the changes are copied to the top writable layer.



As you can see, only one layer is pushed while it says the other layers already exists which is fast.

## Create a sample python app

echo 'from flask import Flask

app = Flask(\_\_name\_\_)

@app.route("/")

def hello():

return "hello world!"

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host="0.0.0.0")' > app.py

This is a simple Python app that uses Flask to expose an HTTP web server on port 5000. (5000 is the default port for flask.)

## Create and build docker image

1. Create a file named Dockerfile and add the following content:

FROM python:3.6.1-alpine

RUN pip install flask

CMD ["python","app.py"]

COPY app.py /app.py

A Dockerfile lists the instructions needed to build a Docker image.

**FROM python:3.6.1-alpine**

This is the starting point for your Dockerfile. Every Dockerfile typically starts with a FROM line that is the starting image to build your layers on top of. In this case, you are selecting the python:3.6.1-alpine base layer because it already has the version of Python and pip that you need to run your application. The alpine version means that it uses the alpine distribution, which is significantly smaller than an alternative flavor of Linux. A smaller image means it will download (deploy) much faster, and it is also more secure because it has a smaller attack surface.

**RUN pip install flask**

The RUN command executes commands needed to set up your image for your application, such as installing packages, editing files, or changing file permissions. In this case, you are installing Flask. The RUN commands are executed at build time and are added to the layers of your image.

**CMD ["python","app.py"]**

CMD is the command that is executed when you start a container. Here, you are using CMD to run your Python application.

There can be only one CMD per Dockerfile. If you specify more than one CMD, then the last CMD will take effect.

**COPY app.py /app.py**

This line copies the app.py file in the local directory (where you will run docker image build) into a new layer of the image. This instruction is the last line in the Dockerfile. Layers that change frequently, such as copying source code into the image, should be placed near the bottom of the file to take full advantage of the Docker layer cache. This allows you to avoid rebuilding layers that could otherwise be cached. For instance, if there was a change in the FROM instruction, it will invalidate the cache for all subsequent layers of this image.

1. Build the Docker image. Pass in the -t parameter to name your image python-hello-world.

$ docker image build -t python-hello-world .

1. Verify that your image shows in your image list:

$ docker image ls

## Run the docker image

1. Run the docker image

$ docker run -p 5001:5000 -d python-hello-world

The -p flag maps a port running inside the container to your host. In this case, you're mapping the Python app running on port 5000 inside the container to port 5001 on your host. Note that if port 5001 is already being used by another application on your host, you might need to replace 5001 with another value.

1. Navigate to http://localhost:5001 in a browser to see the results.
2. Check the log output of the container.

If you want to see logs from your application, you can use the docker container logs command. By default, docker container logs prints out what is sent to standard out by your application.

$ docker container logs [container id]

The Dockerfile is used to create reproducible builds for your application. A common workflow is to have your CI/CD automation run docker image build as part of its build process. After images are built, they will be sent to a central registry where they can be accessed by all environments (such as a test environment) that need to run instances of that application. In the next section, you will push your custom image to the public Docker registry, which is the Docker Hub, where it can be consumed by other developers and operators.

## Push to a central registry

1. Log in to the Docker registry account by entering docker login on your terminal:

$ docker login

Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to create one.

Username:

1. Tag the image with your username.

The Docker Hub naming convention is to tag your image with [dockerhub username]/[image name]. To do this, tag your previously created image python-hello-world to fit that format.

$ docker tag python-hello-world [dockerhub username]/python-hello-world

1. After you properly tag the image, use the docker push command to push your image to the Docker Hub registry:

$ docker push jzaccone/python-hello-world

1. Check your image on Docker Hub in your browser.

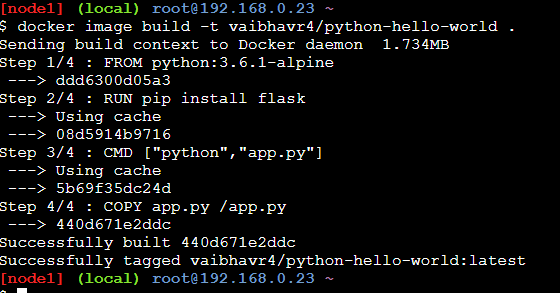
Navigate to Docker Hub and go to your profile to see your uploaded image.

Now that your image is on Docker Hub, other developers and operators can use the docker pull command to deploy your image to other environments.

## Deploy a change

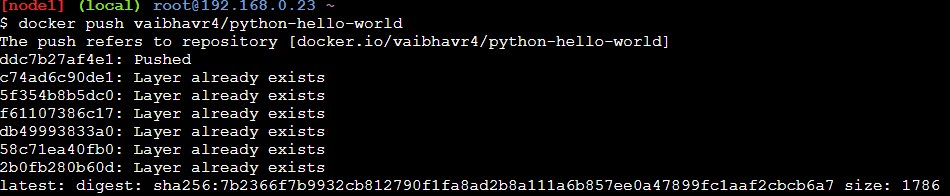
1. Update app.py by replacing the string "Hello World" with "Hello Beautiful World!" in app.py.
2. Rebuild the app by using your Docker Hub username in the build command:

$ docker image build -t vaibhavr4/python-hello-world .



Notice the "Using cache" for Steps 1 - 3. These layers of the Docker image have already been built, and the docker image build command will use these layers from the cache instead of rebuilding them.

$ docker push jzaccone/python-hello-world



There is a caching mechanism in place for pushing layers too. Docker Hub already has all but one of the layers from an earlier push, so it only pushes the one layer that has changed.

## Understanding Image layers

One of the important design properties of Docker is its use of the union file system.

Consider the Dockerfile that you created before:

FROM python:3.6.1-alpine

RUN pip install flask

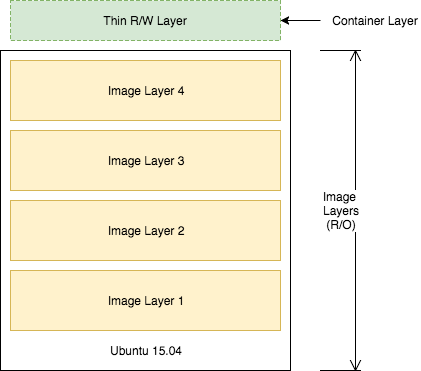
CMD ["python","app.py"]

COPY app.py /app.py

Each of these lines is a layer. Each layer contains only the delta, or changes from the layers before it. To put these layers together into a single running container, Docker uses the union file system to overlay layers transparently into a single view.

Each layer of the image is read-only except for the top layer, which is created for the container. The read/write container layer implements "copy-on-write," which means that files that are stored in lower image layers are pulled up to the read/write container layer only when edits are being made to those files. Those changes are then stored in the container layer.

The "copy-on-write" function is very fast and in almost all cases, does not have a noticeable effect on performance. You can inspect which files have been pulled up to the container level with the docker diffcommand. For more information, see the command-line reference on the [docker diff](https://docs.docker.com/engine/reference/commandline/diff/" \t "_blank) command.



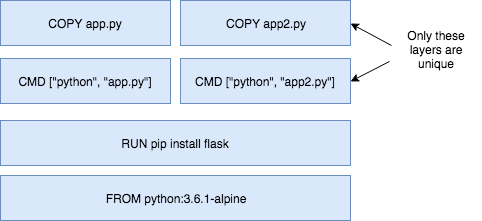
Because image layers are read-only, they can be shared by images and by running containers. For example, creating a new Python application with its own Dockerfile with similar base layers will share all the layers that it had in common with the first Python application.

FROM python:3.6.1-alpine

RUN pip install flask

CMD ["python","app2.py"]

COPY app2.py /app2.py



You can also see the sharing of layers when you start multiple containers from the same image. Because the containers use the same read-only layers, you can imagine that starting containers is very fast and has a very low footprint on the host.

You might notice that there are duplicate lines in this Dockerfile and the Dockerfile that you created earlier in this lab. Although this is a trivial example, you can pull common lines of both Dockerfiles into a base Dockerfile, which you can then point to with each of your child Dockerfiles by using the FROM command.

Image layering enables the docker caching mechanism for builds and pushes. For example, the output for your last docker push shows that some of the layers of your image already exist on the Docker Hub.

$ docker push jzaccone/python-hello-world

The push refers to a repository [docker.io/jzaccone/python-hello-world]

94525867566e: Pushed

64d445ecbe93: Layer already exists

18b27eac38a1: Layer already exists

3f6f25cd8b1e: Layer already exists

b7af9d602a0f: Layer already exists

ed06208397d5: Layer already exists

5accac14015f: Layer already exists

latest: digest: sha256:91874e88c14f217b4cab1dd5510da307bf7d9364bd39860c9cc8688573ab1a3a size: 1786

To look more closely at layers, you can use the docker image history command of the Python image you created.

$ docker image history python-hello-world

IMAGE CREATED CREATED BY SIZE COMMENT

f1b2781b3111 5 minutes ago /bin/sh -c #(nop) COPY file:0114358808a1bb... 159B

0ab91286958b 5 minutes ago /bin/sh -c #(nop) CMD ["python" "app.py"] 0B

ce41f2517c16 5 minutes ago /bin/sh -c pip install flask 10.6MB

c86415c03c37 8 days ago /bin/sh -c #(nop) CMD ["python3"] 0B

<missing> 8 days ago /bin/sh -c set -ex; apk add --no-cache -... 5.73MB

Each line represents a layer of the image. You'll notice that the top lines match to the Dockerfile that you created, and the lines below are pulled from the parent Python image. Don't worry about the <missing> tags. These are still normal layers; they have just not been given an ID by the Docker system.

## Summary

* Use the Dockerfile to create reproducible builds for your application and to integrate your application with Docker into the CI/CD pipeline.
* Docker images can be made available to all of your environments through a central registry. The Docker Hub is one example of a registry, but you can deploy your own registry on servers you control.
* A Docker image contains all the dependencies that it needs to run an application within the image. This is useful because you no longer need to deal with environment drift (version differences) when you rely on dependencies that are installed on every environment you deploy to.
* Docker uses of the union file system and "copy-on-write" to reuse layers of images. This lowers the footprint of storing images and significantly increases the performance of starting containers.
* Image layers are cached by the Docker build and push system. There's no need to rebuild or repush image layers that are already present on a system.
* Each line in a Dockerfile creates a new layer, and because of the layer cache, the lines that change more frequently, for example, adding source code to an image, should be listed near the bottom of the file.

## Lab 2 – Quiz

* 1. Which file should you use to create reproducible builds for Docker images?

**Dockerfile**

* 1. To rebuild and re-push images quickly, you should optimize your Dockerfile for what?

**The layer cache: put lines that change more frequently near the end of the file**

* 1. You must use Docker Hub as the central registry to share the Docker images that you create.

**False**

* 1. What's the purpose of the FROM line in a dockerfile? Select all that apply.

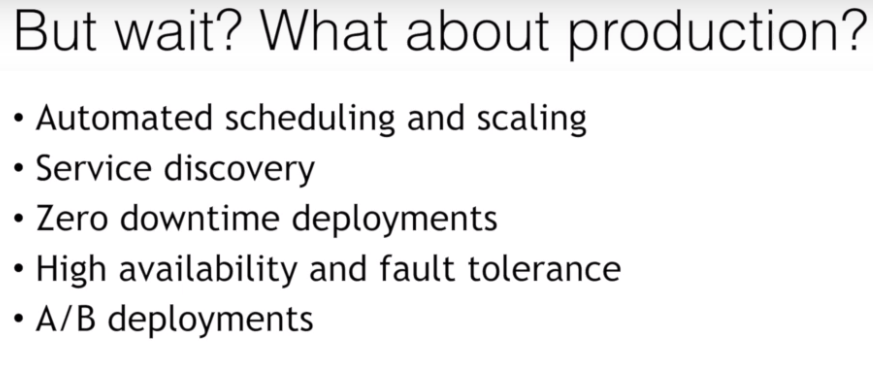
**It's required as the first line in a Dockerfile.**

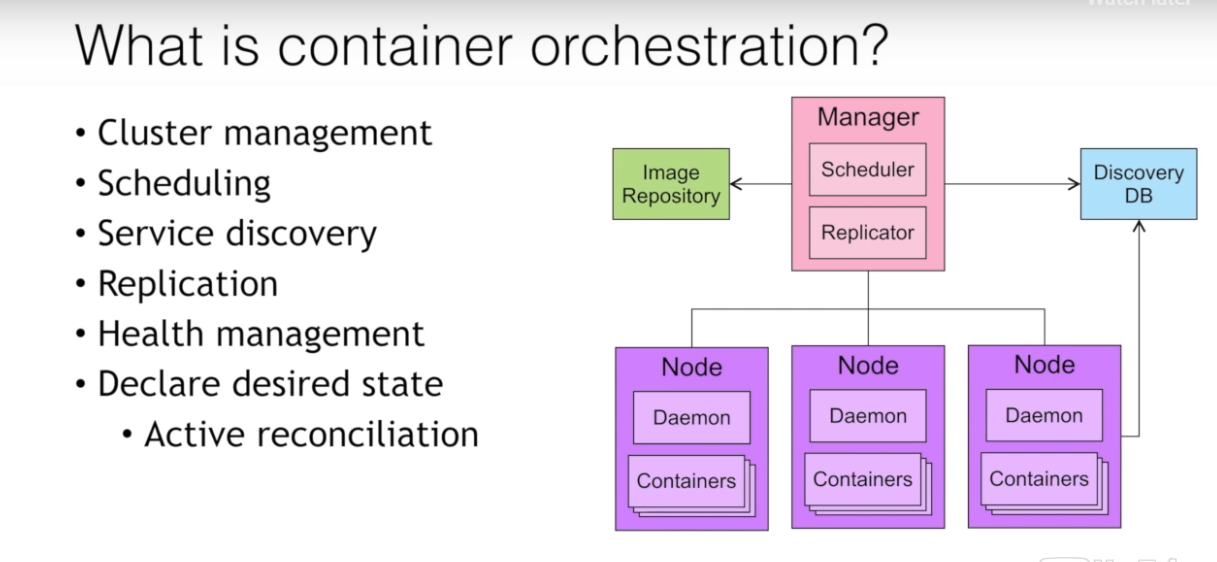
**It specifies the starting image to build other image layers on top of.**

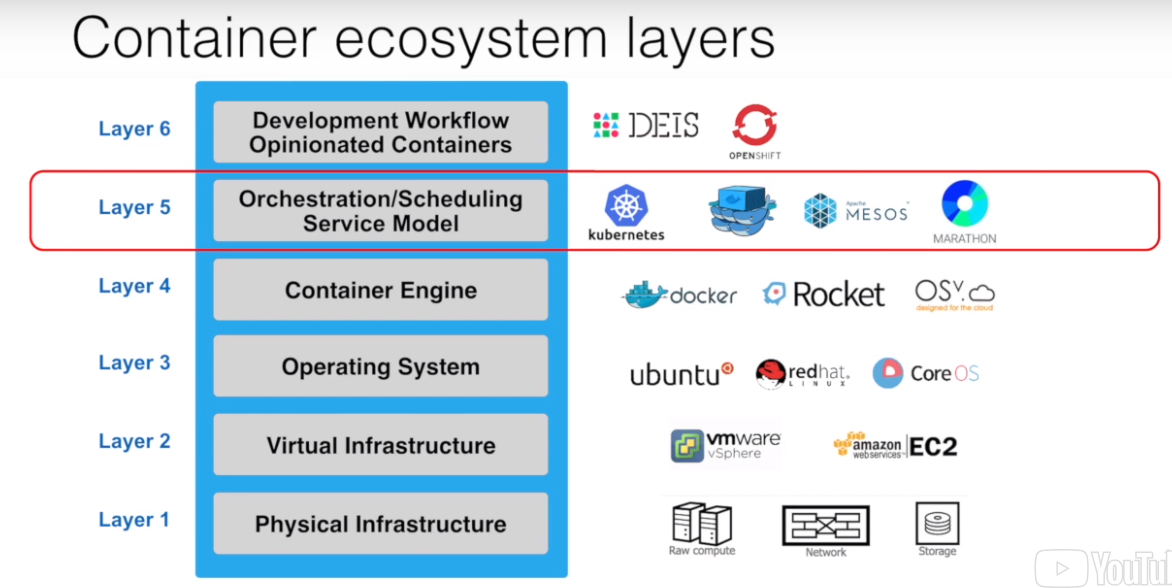
* 1. What does this command do: $ docker system prune.

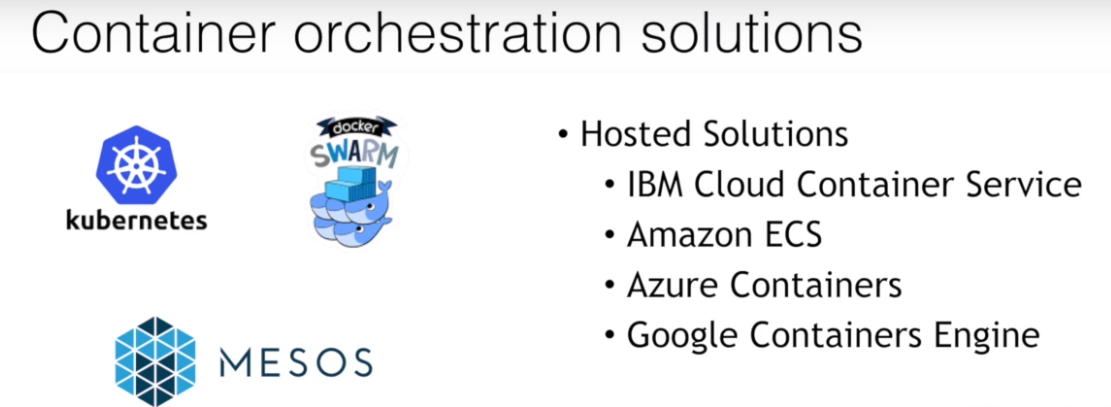
**Removes containers that are already stopped**

# Container Orchestration









## Create your first swarm

1. Navigate to [Play-with-Docker](http://play-with-docker.com/). You're going to create a swarm with three nodes.
2. Click **Add new instance** on the left side three times to create three nodes.
3. Initialize the swarm on node 1:

$ docker swarm init --advertise-addr eth0

Swarm initialized: current node (vq7xx5j4dpe04rgwwm5ur63ce) is now a manager.

To add a worker to this swarm, run the following command:

docker swarm join \

--token SWMTKN-1-50qba7hmo5exuapkmrj6jki8knfvinceo68xjmh322y7c8f0pj-87mjqjho30uue43oqbhhthjui \ 10.0.120.3:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

You can think of Docker Swarm as a special mode that is activated by the command: docker swarm init. The --advertise-addr option specifies the address in which the other nodes will use to join the swarm.

This docker swarm init command generates a join token. The token makes sure that no malicious nodes join the swarm. You need to use this token to join the other nodes to the swarm. For convenience, the output includes the full command docker swarm join, which you can just copy/paste to the other nodes.

1. On both node2 and node3, copy and run the docker swarm join command that was outputted to your console by the last command.

You now have a three-node swarm!

1. Back on node1, run docker node ls to verify your three-node cluster:

$ docker node ls

ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS

7x9s8baa79l29zdsx95i1tfjp node3 Ready Active

x223z25t7y7o4np3uq45d49br node2 Ready Active

zdqbsoxa6x1bubg3jyjdmrnrn \* node1 Ready Active Leader

This command outputs the three nodes in your swarm. The asterisk (\*) next to the ID of the node represents the node that handled that specific command (docker node ls in this case).

Your node consists of one manager node and two workers nodes. Managers handle commands and manage the state of the swarm. Workers cannot handle commands and are simply used to run containers at scale. By default, managers are also used to run containers.

All docker service commands for the rest of this lab need to be executed on the manager node (Node1).

## Deploy a service

Now that you have your three-node Swarm cluster initialized, you'll deploy some containers. To run containers on a Docker Swarm, you need to create a service. A service is an abstraction that represents multiple containers of the same image deployed across a distributed cluster.

Let's do a simple example using NGINX. For now, you will create a service with one running container, but you will scale up later.

1. Deploy a service by using NGINX:

$ docker service create --detach=true --name nginx1 --publish 80:80 --mount source=/etc/hostname,target=/usr/share/nginx/html/index.html,type=bind,ro nginx:1.12

pgqdxr41dpy8qwkn6qm7vke0q

This command statement is declarative, and Docker Swarm will try to maintain the state declared in this command unless explicitly changed by another docker service command. This behavior is useful when nodes go down, for example, and containers are automatically rescheduled on other nodes. You will see a demonstration of that a little later in this lab.

The --mount flag is useful to have NGINX print out the hostname of the node it's running on. You will use this later in this lab when you start load balancing between multiple containers of NGINX that are distributed across different nodes in the cluster and you want to see which node in the swarm is serving the request.

You are using NGINX tag 1.12 in this command. You will see a rolling update with version 1.13 later in this lab.

The --publish command uses the swarm's built-in routing mesh. In this case, port 80 is exposed on every node in the swarm. The routing mesh will route a request coming in on port 80 to one of the nodes running the container.

1. Inspect the service.  Use the command docker service ls to inspect the service you just created:

$ docker service ls

ID NAME MODE REPLICAS IMAGE PORTS

pgqdxr41dpy8 nginx1 replicated 1/1 nginx:1.12 \*:80->80/tcp

1. Check the running container of the service.

To take a deeper look at the running tasks, use the command docker service ps. A task is another abstraction in Docker Swarm that represents the running instances of a service. In this case, there is a 1-1 mapping between a task and a container.

$ docker service ps nginx1

ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR PORTS

iu3ksewv7qf9 nginx1.1 nginx:1.12 node1 Running Running 8 minutes ago

If you know which node your container is running on (you can see which node based on the output from docker service ps), you can use the command docker container ls to see the container running on that specific node.

1. Test the service.

Because of the routing mesh, you can send a request to any node of the swarm on port 80. This request will be automatically routed to the one node that is running the NGINX container.

Try this command on each node:

$ curl localhost:80

node1

Curling will output the hostname where the container is running. For this example, it is running on node1, but yours might be different.

## Scale your service

In production, you might need to handle large amounts of traffic to your application, so you'll learn how to scale.

1. Update your service with an updated number of replicas.

Use the docker service command to update the NGINX service that you created previously to include 5 replicas. This is defining a new state for the service.

$ docker service update --replicas=5 --detach=true nginx1

nginx1

When this command is run, the following events occur:

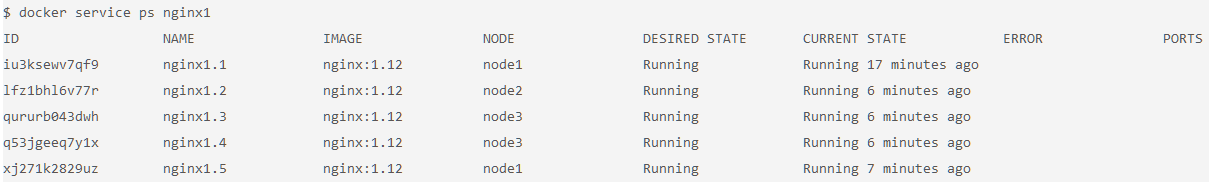
* + The state of the service is updated to 5 replicas, which is stored in the swarm's internal storage.
  + Docker Swarm recognizes that the number of replicas that is scheduled now does not match the declared state of 5.
  + Docker Swarm schedules 5 more tasks (containers) in an attempt to meet the declared state for the service.

This swarm is actively checking to see if the desired state is equal to actual state and will attempt to reconcile if needed.

1. Check the running instances.

After a few seconds, you should see that the swarm did its job and successfully started 9 more containers. Notice that the containers are scheduled across all three nodes of the cluster. The default placement strategy that is used to decide where new containers are to be run is the emptiest node, but that can be changed based on your needs.

$ docker service ps nginx1



1. Send a lot of requests to http://localhost:80.

The --publish 80:80 parameter is still in effect for this service; that was not changed when you ran the docker service update command. However, now when you send requests on port 80, the routing mesh has multiple containers in which to route requests to. The routing mesh acts as a load balancer for these containers, alternating where it routes requests to.

Try it out by curling multiple times. Note that it doesn't matter which node you send the requests. There is no connection between the node that receives the request and the node that that request is routed to.

$ curl localhost:80

node3

$ curl localhost:80

node3

$ curl localhost:80

node2

$ curl localhost:80

node1

$ curl localhost:80

node1

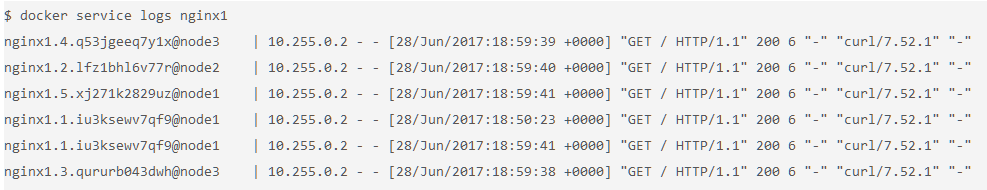
You should see which node is serving each request because of the useful --mount command you used earlier.

**Limits of the routing mesh:** The routing mesh can publish only one service on port 80. If you want multiple services exposed on port 80, you can use an external application load balancer outside of the swarm to accomplish this.

1. Check the aggregated logs for the service.

Another easy way to see which nodes those requests were routed to is to check the aggregated logs. You can get aggregated logs for the service by using the command docker service logs [service name]. This aggregates the output from every running container, that is, the output from docker container logs [container name].

$ docker service logs nginx1



Based on these logs, you can see that each request was served by a different container.

In addition to seeing whether the request was sent to node1, node2, or node3, you can also see which container on each node that it was sent to. For example, nginx1.5 means that request was sent to a container with that same name as indicated in the output of the command docker service ps nginx1.

## Apply rolling updates

Now that you have your service deployed, you'll see a release of your application. You are going to update the version of NGINX to version 1.13.

1. Run the docker service update command:

$ docker service update --image nginx:1.13 --detach=true nginx1

This triggers a rolling update of the swarm. Quickly enter the command docker service ps nginx1 over and over to see the updates in real time.

You can fine-tune the rolling update by using these options:

* + --update-parallelism: specifies the number of containers to update immediately (defaults to 1).
  + --update-delay: specifies the delay between finishing updating a set of containers before moving on to the next set.

1. After a few seconds, run the command docker service ps nginx1 to see all the images that have been updated to nginx:1.13.

$ docker service ps nginx1



You have successfully updated your application to the latest version of NGINX.

## Reconcile problems with containers

The inspect-and-then-adapt model of Docker Swarm enables it to perform reconciliation when something goes wrong. For example, when a node in the swarm goes down, it might take down running containers with it. The swarm will recognize this loss of containers and will attempt to reschedule containers on available nodes to achieve the desired state for that service.

You are going to remove a node and see tasks of your nginx1 service be rescheduled on other nodes automatically.

1. To get a clean output, create a new service by copying the following line. Change the name and the publish port to avoid conflicts with your existing service. Also, add the --replicas option to scale the service with five instances:

$ docker service create --detach=true --name nginx2 --replicas=5 --publish 81:80 --mount source=/etc/hostname,target=/usr/share/nginx/html/index.html,type=bind,ro nginx:1.12

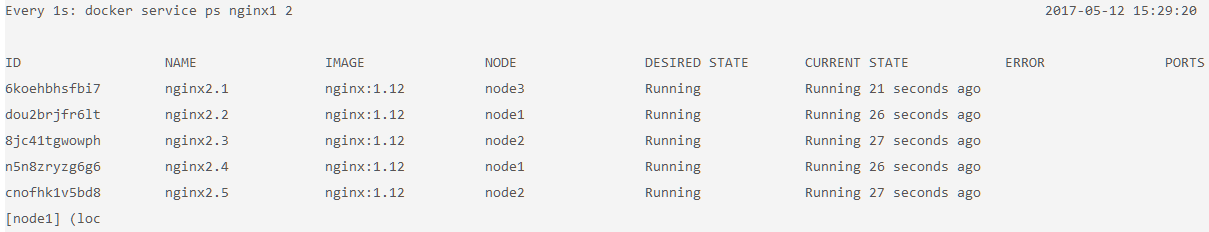
aiqdh5n9fyacgvb2g82s412js

1. On node1, use the watch utility to watch the update from the output of the docker service ps command.

**Tip:** watch is a Linux utility and might not be available on other operating systems.

$ watch -n 1 docker service ps nginx2

This command should create output like this:

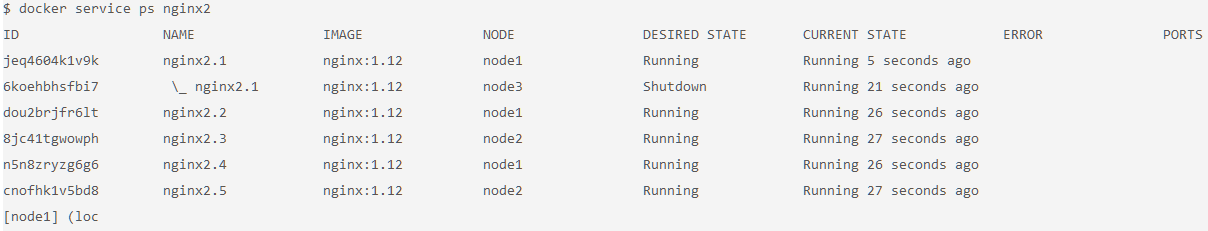


1. Click node3 and enter the command to leave the swarm cluster:

$ docker swarm leave

**Tip**: This is the typical way to leave the swarm, but you can also kill the node and the behavior will be the same.

1. Click node1 to watch the reconciliation in action. You should see that the swarm attempts to get back to the declared state by rescheduling the containers that were running on node3 to node1 and node2 automatically.



## Determine the number of nodes

The manager node contains the necessary information to manage the cluster, but if this node goes down, the cluster will cease to function. For a production application, you should provision a cluster with multiple manager nodes to allow for manager node failures.

You should have at least three manager nodes but typically no more than seven. Manager nodes implement the raft consensus algorithm, which requires that more than 50% of the nodes agree on the state that is being stored for the cluster. If you don't achieve more than 50% agreement, the swarm will cease to operate correctly. For this reason, note the following guidance for node failure tolerance:

* Three manager nodes tolerate one node failure.
* Five manager nodes tolerate two node failures.
* Seven manager nodes tolerate three node failures.

It is possible to have an even number of manager nodes, but it adds no value in terms of the number of node failures. For example, four manager nodes will tolerate only one node failure, which is the same tolerance as a three-manager node cluster. However, the more manager nodes you have, the harder it is to achieve a consensus on the state of a cluster.

While you typically want to limit the number of manager nodes to no more than seven, you can scale the number of worker nodes much higher than that. Worker nodes can scale up into the thousands of nodes. Worker nodes communicate by using the gossip protocol, which is optimized to be perform well under a lot of traffic and a large number of nodes.