DOCKER

Docker is a container

A container is a group of processes run in isolation.
   ->All processes MUST be able to run on the shared kernel.

Each container has its own set of “namespaces” (isolated view)

- PID - process id
- USER - user & group id
- UTS - hostname & domain name
- NS - mount points
- NET - Network devices , stacks , ports
- IPC - inter-process communications , message queues

cgroups - control limits & monitoring of resources

VM vs Container

VM -> heavy/slow to start->Each VM has its own OS

Container -> Fast/lightweight ->Share kernel same base/Namespaces

What is Docker ?

At ist’s core , Docker is tooling to manage containers

-> Simplified existing technology to enable it for the masses

Enable developers to use containers for there applications

->Package dependencies with containers: “build once ,run anywhere”

Why Containers are Appealing to users

1. No more “Works on my machine”
2. Lightweight and fast
3. Better resource utilization
4. Ecosystem & tooling

Run a container

Use the Docker CLI to run your first container.

->Open a terminal on your local computer and run this command:

& docker container run -t ubuntu top

You use the docker container run command to run a container with the Ubuntu image by using the top command. The -t flag allocates a pseudo-TTY, which you need for the top command to work correctly.

```
$ docker container run -it ubuntu top
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
aafe6b5e13de: Pull complete
0a2b43a72660: Pull complete
18bdd1e546d2: Pull complete
8198342c3e05: Pull complete
f56970a44fd4: Pull complete
Digest: sha256:f3a61450ae43896c4332bda5e78b453f4a93179045f20c8181043b26b5e79028
Status: Downloaded newer image for ubuntu:latest
```

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 - 20:32:46 up 3 days, 17:40,  0 users,  load average: 0.00, 0.01, 0.00
Tasks:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.1 sy,  0.0 ni, 99.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  2046768 total,   173308 free,   117248 used,  1756212 buff/cache
KiB Swap:  1048572 total,  1048572 free,        0 used.  1548356 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
    1 root      20   0   36636   3072   2640 R   0.3  0.2   0:00.04 top
```

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

Containers use Linux namespaces to provide isolation of system resources from other containers or the host. The PID namespace provides isolation for process IDs. If you run top while inside the container, you will notice that it shows the processes within the PID namespace of the container, which is much

different than what you can see if you ran top on the host.

Even though we are using the Ubuntu image, it is important to note that the container does not have its own kernel. It uses the kernel of the host and the Ubuntu image is used only to provide the file system and tools available on an Ubuntu system.

Inspect the container:

docker container exec

This command allows you to enter a running container's namespaces with a new process.

In the new terminal, get the ID of the running container that you just created:

docker container ls

```
$ docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
b3ad2a23fab3        ubuntu              "top"               29 minutes ago      Up 29 minutes                           goofy_nobel
```

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 pseudo-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.

Notice the change in the prefix of your terminal, for example, root@b3ad2a23fab3:/. This is an indication that you are running bash inside the container.

Tip: This is not the same as **using ssh to a separate host or a VM.** You don't need an ssh server to connect with a bash process. Remember that containers use kernel-level features to achieve isolation and that containers run on top of the kernel. Your

container is just a group of processes running in isolation on the same host, and you can use the **command docker container exec to enter that isolation with the bash process**. After you run the command docker container exec, the group of processes running in isolation (in other words, the container) includes top and bash.

From the same terminal, inspect the running processes:

$ ps -ef

For Windows, you can inspect the running processes by using tasklist.

PID is just one of the Linux namespaces that provides containers with isolation to system resources. Other Linux namespaces include:

-
- MNT: Mount and unmount directories without affecting other namespaces.
- NET: Containers have their own network stack.
- IPC: Isolated interprocess communication mechanisms such as message queues.

- User: Isolated view of users on the system.
- UTC: Set hostname and domain name per container.

Tip: Namespaces are a feature of the Linux kernel. However, Docker allows you to run containers on Windows and Mac. The secret is that embedded in the Docker product is a Linux subsystem. Docker open-sourced this Linux subsystem to a new project: LinuxKit. Being able to run containers on many different platforms is one advantage of using the Docker tooling with containers.

In addition to running Linux containers on Windows by using a Linux subsystem, native Windows containers are now possible because of the creation of container primitives on the Windows operating system. Native Windows containers can be run on Windows 10 or Windows Server 2016 or later.

RUN Multiple Containers

1.Explore the Docker Store.

The Docker Store is the public central registry for Docker images. Anyone can share images here publicly. The Docker Store contains community and official images that can also be found on the Docker Hub.

When searching for images, you will find filters for Store and Community images. Store images include content that has been verified and scanned for security vulnerabilities by Docker. Go one step further and search for Certified images that are deemed enterprise-ready and are tested with Docker Enterprise Edition.

Run an NGINX server by using the official NGINX image from the Docker Store:

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

```
$ docker container run --detach --publish 8080:80 --name nginx nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
36a46ebd5019: Pull complete
57168433389f: Pull complete
332ec8285c50: Pull complete
Digest: sha256:c15f1fb8fd55c60c72f940a76da76a5fccce2fefa0dd9b17967b9e40b0355316
Status: Downloaded newer image for nginx:latest
5e1bf0e6b926bd73a66f98b3cbe23d04189c16a43d55dd46b8486359f6fdf048
```

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

Because this is the first time you are running the NGINX container, it will pull down the NGINX image from the Docker Store. Subsequent containers created from the NGINX image will use the existing image located on your host.

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

Access the NGINX server on
http:// localhost:8080

Run a MongoDB server. You will use the official MongoDB image 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

```
$ docker container run --detach --publish 8081:27017 --name mongo mongo:3.4
Unable to find image 'mongo:3.4' locally
3.4: Pulling from library/mongo
d13d02fa248d: Already exists
bc8e2652ce92: Pull complete
3cc856886986: Pull complete
c319e9ec4517: Pull complete
b4cbf8808f94: Pull complete
cb98a53e6676: Pull complete
f0485050cd8a: Pull complete
ac36cdc414b3: Pull complete
61814e3c487b: Pull complete
523a9f1da6b9: Pull complete
3b4beaef77a2: Pull complete
Digest: sha256:d13c897516e497e898c229e2467f4953314b63e48d4990d3215d876ef9d1fc7c
Status: Downloaded newer image for mongo:3.4
d8f614a4969fb1229f538e171850512f10f490cb1a96fca27e4aa89ac082eba5
```

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.** See the documentation on the Docker Store to get more information about using the Mongo image.

Access http://localhost:8081 to see some output from Mongo

Check your running containers

$ docker container ls

You should see that you

```
$ docker container ls
CONTAINER ID        IMAGE               COMMAND                  CREATED                  STATUS              PORTS
NAMES
d6777df89fea        nginx               "nginx -g 'daemon ..."   Less than a second ago   Up 2 seconds        0.0.0.0:8080->80/tcp
nginx
ead80a0db505        mongo               "docker-entrypoint..."   17 seconds ago           Up 19 seconds       0.0.0.0:8081->27017/tcp
mongo
af549dccd5cf        ubuntu              "top"                    5 minutes ago            Up 5 minutes
priceless_kepler
```

have an **NGINX** web server container and a **MongoDB** container running on your host. Note that you have not configured these containers to talk to each other.

You can see the nginx and mongo names that you gave to the containers and the random name (in this example, priceless_kepler) that was generated **for the Ubuntu container.** You can also see that the port mappings that you specified with the **--publish flag.** For more information on these running containers, **use the docker container inspect [container id] command.**

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. Typically, you can find the link to the GitHub source from the image description page on the Docker Store website.

Containers are self-contained and isolated, which means you can avoid potential conflicts between containers with different system or runtime dependencies. For example, you can deploy an app that uses Java 7 and another app that uses Java 8 on the same host. Or you can run multiple NGINX containers that all have port 80 as their default listening ports. (If you're exposing on the host by using the --publish flag, the ports selected for the host must be unique.) Isolation benefits are possible because of Linux namespaces.

Remove the containers

Completing this lab creates several running containers on your host. Now, you'll stop and remove those containers.

Get a list of the running containers:

$ docker container ls

```
$ docker container ls
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                     NAMES
d6777df89fea        nginx               "nginx -g 'daemon ..."   3 minutes ago       Up 3 minutes        0.0.0.0:8080->80/tcp      nginx
ead80a0db505        mongo               "docker-entrypoint..."   3 minutes ago       Up 3 minutes        0.0.0.0:8081->27017/tcp   mongo
af549dccd5cf        ubuntu              "top"                    8 minutes ago       Up 8 minutes                                  priceless_kepler
```

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:

$ docker container stop d67 ead af5
d67
ead
af5

Tip: You need to enter only enough digits of the ID to be unique. Three digits is typically adequate.

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

$ docker system prune

```
$ docker system prune
WARNING! This will remove:
        - all stopped containers
        - all volumes not used by at least one container
        - all networks not used by at least one container
        - all dangling images
Are you sure you want to continue? [y/N] y
Deleted Containers:
7872fd96ea4695795c41150a06067d605f69702dbcb9ce49492c9029f0e1b44b
60abd5ee65b1e2732ddc02b971a86e22de1c1c446dab165462a08b037ef7835c
31617fdd8e5f584c51ce182757e24a1c9620257027665c20be75aa3ab6591740

Total reclaimed space: 12B
```

- Containers are composed of Linux namespaces and control groups that provide isolation from other containers and the host.
- Because of the isolation properties of containers, you can schedule many containers on a single host without worrying about conflicting dependencies. This makes it easier to run multiple containers on a single host: using all resources allocated to that host and ultimately saving server costs.
- That you should avoid using unverified content from the Docker Store when developing your own images because these images might contain security vulnerabilities or possibly even malicious software.
- Containers include everything they need to run the processes within them, so you don't need to install additional dependencies on the host.

Docker Images

Tar file containing a container’s filesystem + metadata

For sharing & redistribution.

Docker Registry

Push & Pull images from registry

Default registry: Docker Hub

->Public and free for public images

->Many pre-packaged images available

Private Registry

->Self-host or cloud provider options

Creating a Docker image - with Docker build

Create a “Dockerfile”

->List of instructions for how to construct the container

-> ’docker build -f Dockerfile’

-> $cat Dockerfile

FROM ubuntu

ADD myapp /

EXPOSE 80

ENTRYPOINT /myapp

Docker Image Layers

Image Layers

Docker Image Layers

Union File System

->Merge image layers into a single file system for each container.

Copy-on-Write

->Copies files that are edited up to top writable layer

Advantages

->More containers per host

->Faster start-up/download time -base layers are “cached”

DOcker Image Layers

Docker push dfdfdfde/hello-world

…..layers already exist/ cached..

- Create custom image using a Dockerfile

- Build and run your image locally
- Push your image to your account on DockerHub
- Update your image with a code change
- ->Watch Docker image layering/caching in action!

Create a Python app (without using Docker)

Copy and paste this entire command into the terminal. The result of running this command will create a file named app.py

```
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
```

```
$ python3 --version
```

```
$ pip3 --version
```

```
$ pip3 install flask
```

```
$ python3 app.py
```

Create and build the Docker image

If you don't have Python installed locally, don't worry because you don't need it. One of the advantages of using Docker containers is that you can build Python into your containers without having Python installed on your host.

- 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. Let's go through the Dockerfile line by line.

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.

Here you are using the 3.6.1-alpine tag for the Python image. Look at the available tags for the official Python image on the Docker Hub. It is best practice to use a

specific tag when inheriting a parent image so that changes to the parent dependency are controlled. If no tag is specified, the latest tag takes effect, which acts as a dynamic pointer that points to the latest version of an image.

- **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 applcation.

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

python:3.6.1-alpine also specifies a CMD (CMD python2). You can look at the Dockerfile for the official python:alpine image.

You can use the official Python image directly to run Python scripts without installing Python on your host. However, in this case, you are creating a custom image to include your source so that you can build an image with your application and ship it to other environments.

- **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. You'll see this little later in this lab.

It seems counter-intuitive to put this line after the CMD ["python","app.py"] line. Remember, the CMD line is executed only when the container is started, so you won't get a file not found error here.

And there you have it: a very simple Dockerfile. See the full list of commands that you can put into a Dockerfile. Now that you've defined the Dockerfile, you'll use it to build your custom docker image.

Build the Docker image.

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

docker image build -t python-hello-world

```
Sending build context to Docker daemon
3.072kB

Step 1/4 : FROM python:3.6.1-alpine

3.6.1-alpine: Pulling from library/python

acb474fa8956: Pull complete

967ab02d1ea4: Pull complete

640064d26350: Pull complete

db0225fcac8f: Pull complete

5432cc692c60: Pull complete

Digest:
sha256:768360b3fad01adffcf5ad9eccb4aa3ccc83
bb0ed341bbdc45951e89335082ce

Status: Downloaded newer image for
python:3.6.1-alpine

 ---> c86415c03c37

Step 2/4 : RUN pip install flask

 ---> Running in cac3222673a3

Collecting flask
```

```
  Downloading
Flask-0.12.2-py2.py3-none-any.whl (83kB)

Collecting itsdangerous>=0.21 (from flask)

  Downloading itsdangerous-0.24.tar.gz
(46kB)

Collecting click>=2.0 (from flask)

  Downloading
click-6.7-py2.py3-none-any.whl (71kB)

Collecting Werkzeug>=0.7 (from flask)

  Downloading
Werkzeug-0.12.2-py2.py3-none-any.whl
(312kB)

Collecting Jinja2>=2.4 (from flask)

  Downloading
Jinja2-2.9.6-py2.py3-none-any.whl (340kB)

Collecting MarkupSafe>=0.23 (from
Jinja2>=2.4->flask)

  Downloading MarkupSafe-1.0.tar.gz

Building wheels for collected packages:
itsdangerous, MarkupSafe
```

```
  Running setup.py bdist_wheel for
itsdangerous: started

  Running setup.py bdist_wheel for
itsdangerous: finished with status 'done'

  Stored in directory:
/root/.cache/pip/wheels/fc/a8/66/24d655233c
757e178d45dea2de22a04c6d92766abfb741129a

  Running setup.py bdist_wheel for
MarkupSafe: started

  Running setup.py bdist_wheel for
MarkupSafe: finished with status 'done'

  Stored in directory:
/root/.cache/pip/wheels/88/a7/30/e39a54a87b
cbe25308fa3ca64e8ddc75d9b3e5afa21ee32d57

Successfully built itsdangerous MarkupSafe

Installing collected packages:
itsdangerous, click, Werkzeug, MarkupSafe,
Jinja2, flask

Successfully installed Jinja2-2.9.6
MarkupSafe-1.0 Werkzeug-0.12.2 click-6.7
flask-0.12.2 itsdangerous-0.24

 ---> ce41f2517c16
```

```
Removing intermediate container
cac3222673a3

Step 3/4 : CMD python app.py

 ---> Running in 2197e5263eff

 ---> 0ab91286958b

Removing intermediate container
2197e5263eff

Step 4/4 : COPY app.py /app.py

 ---> f1b2781b3111

Removing intermediate container
b92b506ee093

Successfully built f1b2781b3111

Successfully tagged
python-hello-world:latest
```

Verify that your image shows in your image list:

```
$ docker image ls
```

Run the Docker image

Now that you have built the image, you can run it to see that it works.

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

0b2ba61df37fb4038d9ae5d145740c63c2c211ae2729fc27dc01b82b5aaafa26

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, such as 5002.

Navigate to http://localhost:5001 in a browser to see the results

You should see "hello world!" in your browser.

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 print out what is sent to standard out by your application. Use the command docker container ls to find the ID for your running container.

$ docker container logs [container id]

```
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)

172.17.0.1 - - [28/Jun/2017 19:35:33] "GET / HTTP/1.1" 200 -
```

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

Navigate to Docker Hub and create a free account if you haven't already.

or this lab, you will use the Docker Hub as your central registry. Docker Hub is a free service to publicly store available images. You can also pay to store private images.

Most organizations that use Docker extensively will set up their own registry internally. To simplify things, you will use Docker Hub, but the following concepts apply to any registry.

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

$ docker login

Tag the image with your username.

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

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

Check your image on Docker Hub in your browser.

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.

Remember: Docker images contain all the dependencies that they need to run an application within the image. This is useful because you no longer need to worry about environment drift (version differences) when you rely on dependencies that are installed on every environment you deploy to. You also don't need to follow more steps to provision these environments. Just one step: install docker, and that's it.

Deploy a change

Update app.py by replacing the string "Hello World" with "Hello Beautiful World!" in app.py.

Your file should have the following contents:

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def hello():
    return "Hello Beautiful World!"
if __name__ == "__main__":
    app.run(host='0.0.0.0')
```

Now that your application is updated, you need to rebuild your app and push it to the Docker Hubregistry.

Rebuild the app by using your Docker Hub username in the build command:

docker image build -t jzaccone/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

```
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:
sha256:91874e88c14f217b4cab1dd5510da307bf7d9364bd
39860c9cc8688573ab1a3a size: 1786
```

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.

When you change a layer, every layer built on top of that will have to be rebuilt. Each line in a Dockerfile builds a new layer that is built on the layer created from the lines before it. This is why the order of the lines in your Dockerfile is important. You optimized your Dockerfile so that the layer that is most likely to change **(COPY app.py /app.py)** is the last line of the Dockerfile. Generally for an application, your code changes at the most frequent rate.

This optimization is particularly important for CI/CD processes where you want your automation to run as fast as possible.

Understand 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.

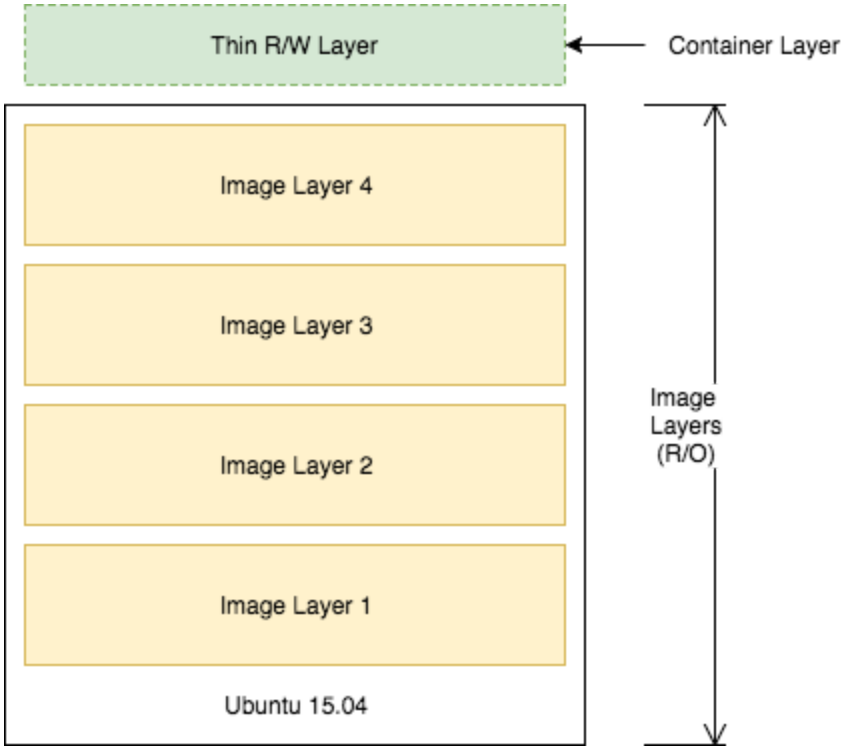
->To put these layers together into a single running container,

->Docker **uses the union file system to overlay layers** transparently into a single view.

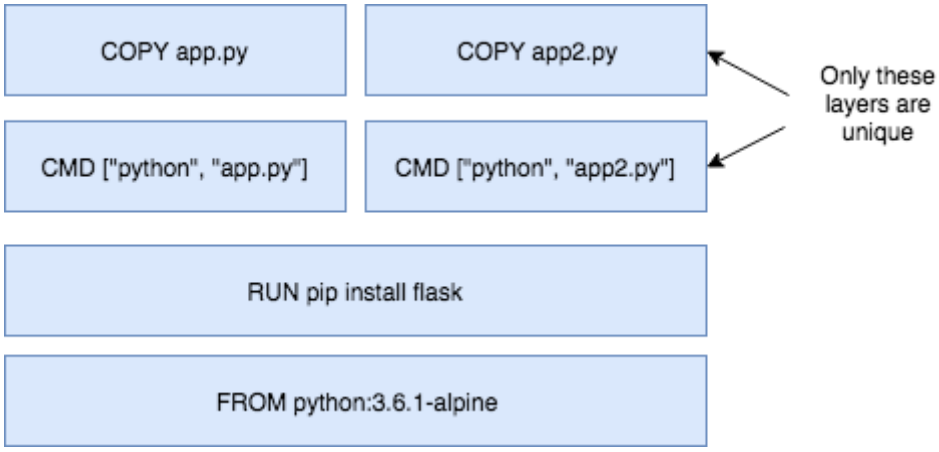
Each layer of the image is read-only except for the top layer, which is created for the container.

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 diff command. For more information, see the command-line reference on the docker diff 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.

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

Remove the containers

$ docker container ls

$ docker container stop 0b2
0b2

$ docker system prune

Intro to container Orchestration

Cluster Management
Scheduling
Service Discovery
Replication
Health Management
Declare desired state
  ->Active reconciliation

Container orchestration solutions

Kubernetes , Docker Swarm , Mesos

Hosted Solutions ->

IBM Cloud Container Service
Amazon ECS
Azure Container
Google Containers Engine

- Create a SWARM using play-with-docker
- Use Docker Swarm to schedule and Scale an Application
- Expose your app using Docker Swarm's built-in routing mesh
- Update your app with a rolling update
- Demonstrate node failure and reconciliation.

Create your first swarm

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-1-50qba7hmo5exuapkmrj6jki8knfvince
o68xjmh322y7c8f0pj-87mjqjho30uue43oqbhhth
jui \
    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.

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!

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.

Note: Although you control the swarm directly from the node in which its running, you can control a Docker swarm remotely by connecting to the Docker Engine of the manager by using the remote API or by activating a remote host from your local Docker installation (using the $DOCKER_HOST and $DOCKER_CERT_PATH environment variables). This will become useful when you want to remotely control production applications, instead of using SSH to directly control production servers.

Deploy your first 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.

Deploy a service by using NGINX:

```
$ docker service create --detach=true
--name nginx1 --publish 80:80  --mount
source=/etc/hostname,target=/usr/share/ng
inx/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.

The --mount flag is useful to have NGINX print out the hostname of the node it's running on.

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.

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

$ docker service ls

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.

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.

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

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 the actual state and will attempt to reconcile if needed.

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

```
$ docker service ps nginx1
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE            ERROR
iu3ksewv7qf9        nginx1.1            nginx:1.12          node1               Running             Running 17 minutes ago
lfz1bhl6v77r        nginx1.2            nginx:1.12          node2               Running             Running 6 minutes ago
qururb043dwh        nginx1.3            nginx:1.12          node3               Running             Running 6 minutes ago
q53jgeeq7y1x        nginx1.4            nginx:1.12          node3               Running             Running 6 minutes ago
xj271k2829uz        nginx1.5            nginx:1.12          node1               Running             Running 7 minutes ago
```

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.**

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

```
$ docker service logs nginx1
nginx1.4.q53jgeeq7y1x@node3    | 10.255.0.2 - - [28/Jun/2017:18:59:39 +0000] "GET / HTTP/1.1" 200 6 "-" "curl/7.52.1" "-"
nginx1.2.lfz1bhl6v77r@node2    | 10.255.0.2 - - [28/Jun/2017:18:59:40 +0000] "GET / HTTP/1.1" 200 6 "-" "curl/7.52.1" "-"
nginx1.5.xj271k2829uz@node1    | 10.255.0.2 - - [28/Jun/2017:18:59:41 +0000] "GET / HTTP/1.1" 200 6 "-" "curl/7.52.1" "-"
nginx1.1.iu3ksewv7qf9@node1    | 10.255.0.2 - - [28/Jun/2017:18:50:23 +0000] "GET / HTTP/1.1" 200 6 "-" "curl/7.52.1" "-"
nginx1.1.iu3ksewv7qf9@node1    | 10.255.0.2 - - [28/Jun/2017:18:59:41 +0000] "GET / HTTP/1.1" 200 6 "-" "curl/7.52.1" "-"
nginx1.3.qururb043dwh@node3    | 10.255.0.2 - - [28/Jun/2017:18:59:38 +0000] "GET / HTTP/1.1" 200 6 "-" "curl/7.52.1" "-"
```

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

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.

$ docker service ps nginx1

```
$ docker service ps nginx1
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE            ERROR               PORTS
di2hmdpw0j0z        nginx1.1            nginx:1.13          node1               Running             Running 50 seconds ago
iu3ksewv7qf9         \_ nginx1.1        nginx:1.12          node1               Shutdown            Shutdown 52 seconds ago
qsk6gw43fgfr        nginx1.2            nginx:1.13          node2               Running             Running 47 seconds ago
lfz1bhl6v77r         \_ nginx1.2        nginx:1.12          node2               Shutdown            Shutdown 49 seconds ago
r4429oql42z9        nginx1.3            nginx:1.13          node3               Running             Running 41 seconds ago
qururb043dwh         \_ nginx1.3        nginx:1.12          node3               Shutdown            Shutdown 43 seconds ago
jfkepz8tqy9g        nginx1.4            nginx:1.13          node2               Running             Running 44 seconds ago
q53jgeeq7y1x         \_ nginx1.4        nginx:1.12          node3               Shutdown            Shutdown 45 seconds ago
n15o01ouv2uf        nginx1.5            nginx:1.13          node3               Running             Running 39 seconds ago
xj271k2829uz         \_ nginx1.5        nginx:1.12          node1               Shutdown            Shutdown 40 seconds ago
```

Reconcile problems with containers

In the previous section, you updated the state of your service by using the command docker service update. You saw Docker Swarm in action as it recognized the mismatch between desired state and actual state, and attempted to solve the issue.

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.

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/ng
inx/html/index.html,type=bind,ro
nginx:1.12
Aiqdh5n9fyacgvb2g82s412js
```

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

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

$ docker swarm leave

Determine how many nodes you need

Docker Swarm cluster consists of one master and two worker nodes. This configuration is not highly available.

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

perform well under a lot of traffic and a large number of nodes.

If you are using Play-with-Docker, you can easily deploy multiple manager node clusters by using the built in templates. Click the **Templates** icon in the upper left to view the available templates.