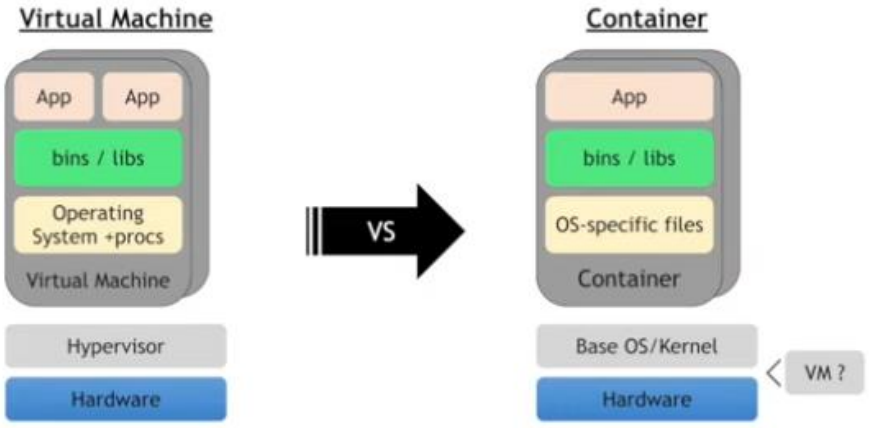
What are Containers?

- 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 IDs
  - **USER** - user and group IDs
  - **UTS** - hostname and domain name
  - **NS** - mount points
  - **NET** - Network devices, stacks, ports
  - **IPC** - inter-process communications, message queues
- **cgroups** - controls limits and monitoring of resources

VM vs Container

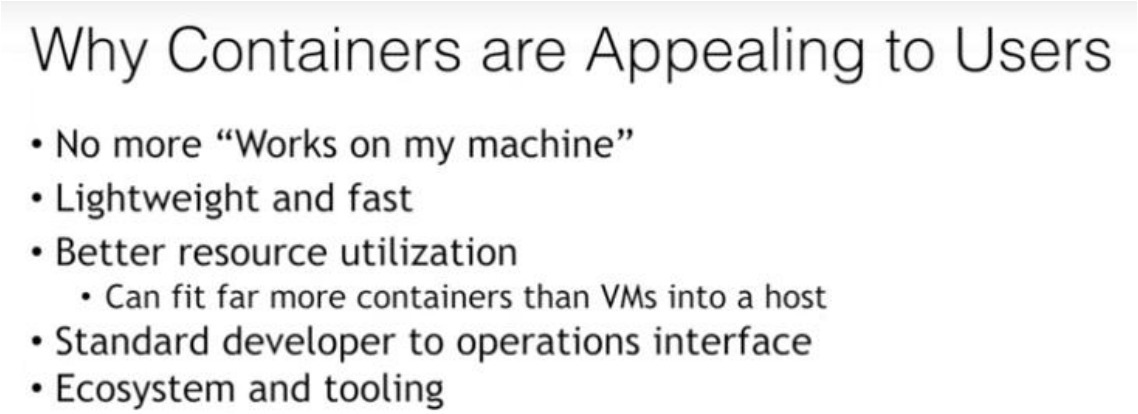


Each VM has its own OS

Containers share the same base Kernel

What is Docker?

- At its core, Docker is tooling to manage containers
  - Simplified existing technology to enable it for the masses
- Enable developers to use containers for their applications
  - Package dependencies with containers: “build once, run anywhere”



Lab overview

Containers are just a process (or a group of processes) running in isolation, which is achieved with Linux namespaces and control groups. Linux namespaces and control groups are features that are built into the Linux kernel. Other than the Linux kernel itself, there is nothing special about containers.

What makes containers useful is the tooling that surrounds them. The labs in this course use Docker, which has been the understood standard tool for using containers to build applications. Docker provides developers and operators with a friendly interface to build, ship, and run containers on any environment.

In the first part of this lab, run your first container, and learn how to inspect it. You will be able to witness the namespace isolation that you acquire from the Linux kernel.

After you run your first container, you can explore other uses of docker containers. You can find many examples of these on the Docker Store and can run several different types of containers on the same host, which allows you to see the benefit of isolation—where you can run multiple containers on the same host without conflicts.

You will use a few Docker commands in this lab. If interested, see the full documentation on available commands.

Run a container

Use the Docker CLI to run your first container.

1. 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.

2. Inspect the container:
3. `docker container exec`

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

4. Open a new terminal. To open a new terminal connected to node1 by using Play-With-Docker.com, click **Add New Instance** on the left and then ssh from node2 into node1 by using the IP that is listed by node1, for example:

```
[node2] (local) root@192.168.0.17 ~
$ ssh 192.168.0.18
[node1] (local) root@192.168.0.18 ~
$
```

5. In the new terminal, get the ID of the running container that you just created:
6. `docker container ls`

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

7. 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:
8. `$ 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`.

9. From the same terminal, inspect the running processes:
10. `$ ps -ef`

```
root@b3ad2a23fab3:/# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 20:34 ?        00:00:00 top
root        17     0  0 21:06 ?        00:00:00 bash
root        27    17  0 21:14 ?        00:00:00 ps -ef
```

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

11. 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`.

```
12.  root@b3ad2a23fab3:/# exit
13.  exit
14.  $ ps -ef
   # Lots of processes!
```

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.

These namespaces provide the isolation for containers that allow them to run together securely and without conflict with other containers running on the same system.

In the next lab, you'll see different uses of containers and the benefit of isolation as you run multiple containers on the same host.

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.

15. Clean up the container running the `top` processes:

```
<ctrl>-c
```

2. 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.

   It is important to avoid using unverified content from the Docker Store when you develop your own images that are intended to be deployed into the production environment. These unverified images might contain security vulnerabilities or possibly even malicious software.

   In the next step of this lab, you will start a couple of containers by using some verified images from the Docker Store: NGINX web server and Mongo database.

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

3. `$ 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.

How do you know port 80 is the default port for NGINX? Because it is listed in the documentation on the Docker Store. In general, the documentation for the verified images is very good, and you will want to refer to it when you run containers using those images.

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.

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

5. 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:d13c897516e497e898c229e2467f4953314b63e48d4990d3215d876ef9d1fc7
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.

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

It looks like you are trying to access MongoDB over HTTP on the native driver port.

7. Check your running containers:
8. `$ docker container ls`

```
$ 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
nginx
ead80a0db505        mongo               "docker-entrypoint..."   17 seconds ago           Up 19 seconds       0.0.0
mongo
af549dccd5cf        ubuntu              "top"                    5 minutes ago            Up 5 minutes
priceless_kepler
```

You should see that you 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.

Remember: You didn't have to install anything on your host (other than Docker) to run these processes! Each container includes the dependencies that it needs within the container, so you don't need to install anything on your host directly.

Running multiple containers on the same host gives us the ability to use the resources (CPU, memory, and so on) available on single host. This can result in huge cost savings for an enterprise.

Although running images directly from the Docker Store can be useful at times, it is more useful to create custom images and refer to official images as the starting point for these images. You'll learn to build your own custom images in the next lab.

3. Remove the containers

3. Remove the containers

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

1. Get a list of the running containers:

```
$ docker container ls
```

```
$ docker container ls
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
d6777df89fea        nginx               "nginx -g 'daemon ..."   3 minutes ago       Up 3 minutes        0.0.0.0:8080->8
ead80a0db505        mongo               "docker-entrypoint..."   3 minutes ago       Up 3 minutes        0.0.0.0:8081->2
af549dccd5cf        ubuntu              "top"                    8 minutes ago       Up 8 minutes
```

3. Stop the containers by running this command for each container in the list:
4. `$ 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.

5. Remove the stopped containers. The following command removes any stopped containers, unused volumes and networks, and dangling images:
6. `$ 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
```

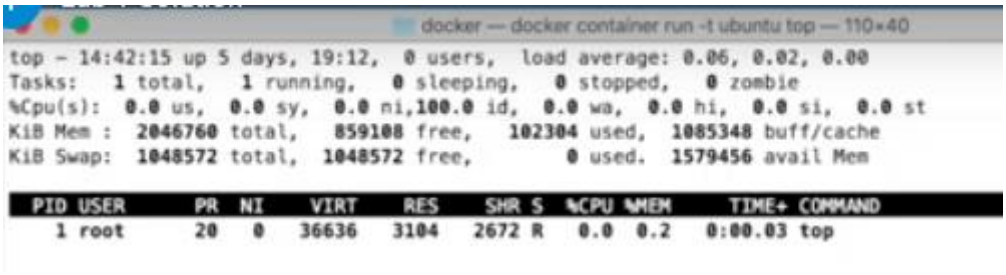
Lab 1 summary

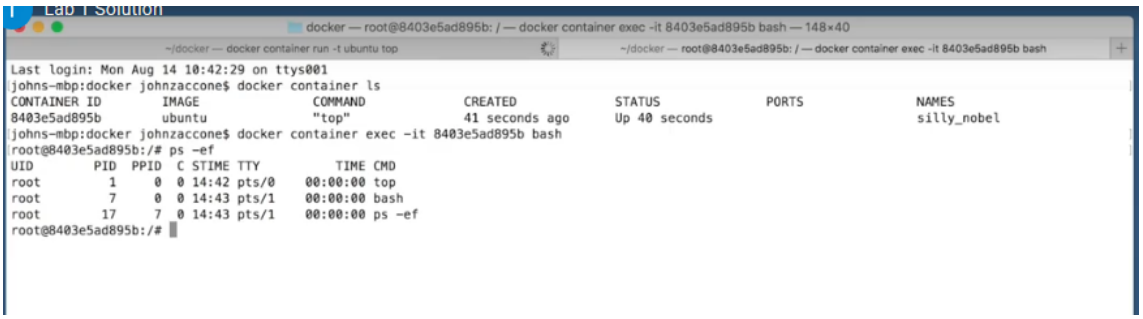
In this lab, you created your first Ubuntu, NGINX, and MongoDB containers.

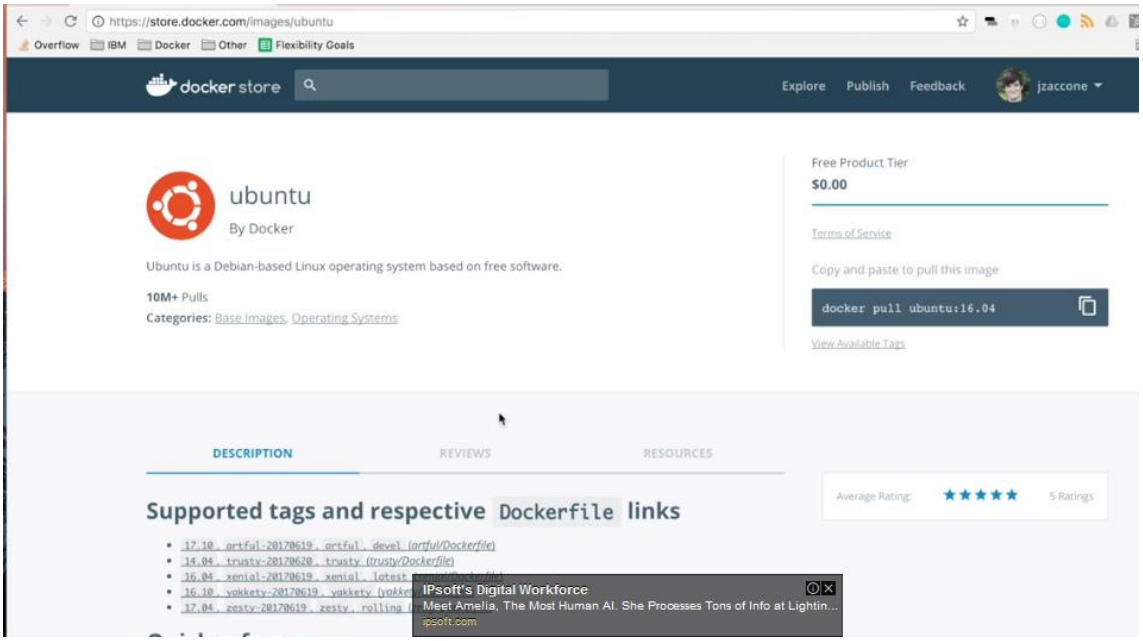
You should now understand more about containers:

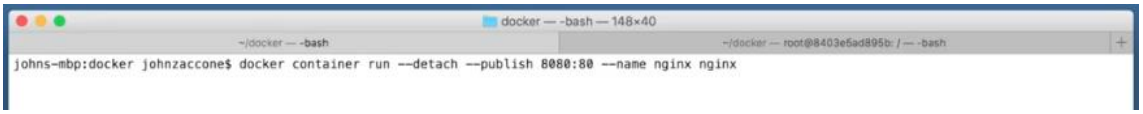
- Containers are composed of Linux namespaces and control groups that provide isolation from other containers and the host.

- 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.

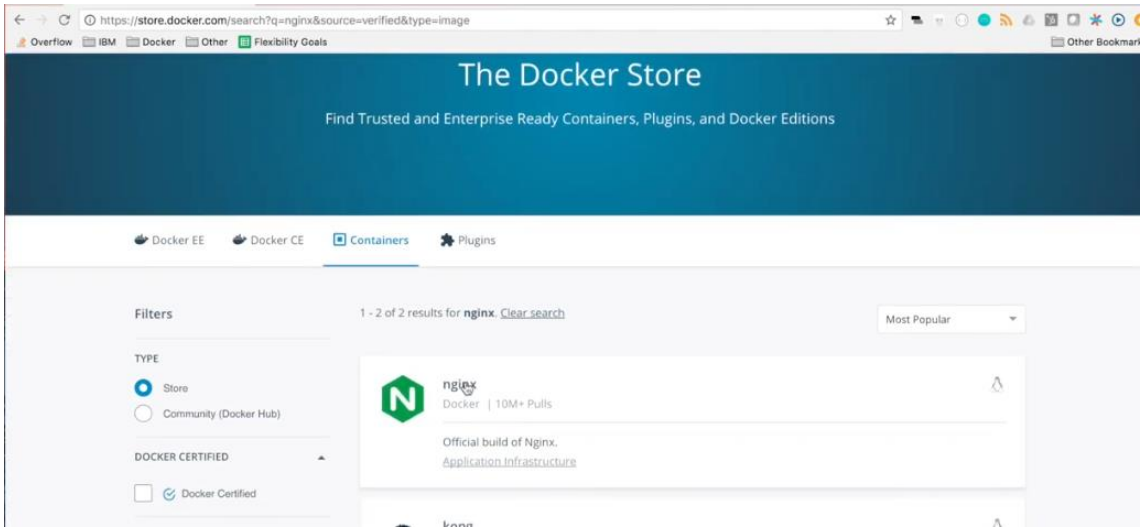








```
johns-mbp:docker johnzaccone$ docker container run --detach --publish 8080:80 --name nginx nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
94ed0c431eb5: Downloading [==================================>                ]  17.45MB/22.49MB
9406c100a1c3: Download complete
aa74daafd50c: Download complete
```



Hosting some simple static content

```
$ docker run --name some-nginx -v /some/content:/usr/share/nginx/html:ro -d nginx
```

Alternatively, a simple `Dockerfile` can be used to generate a new image that includes the necessary content (which is a much cleaner solution than the bind mount above):

```
FROM nginx
COPY static-html-directory /usr/share/nginx/html
```

Place this file in the same directory as your directory of content ("static-html-directory"), run `docker build -t some-content-nginx .`, then start your container:

```
$ docker run --name some-nginx -d some-content-nginx
```

Exposing external port

```
$ docker run --name some-nginx -d -p 8080:80 some-content-nginx
```

Then you can hit `http://localhost:8080` or `http://host-ip:8080` in your browser.

Complex configuration

```
$ docker run --name my-custom-nginx-container -v
/host/path/nginx.conf:/etc/nginx/nginx.conf:ro -d nginx
```

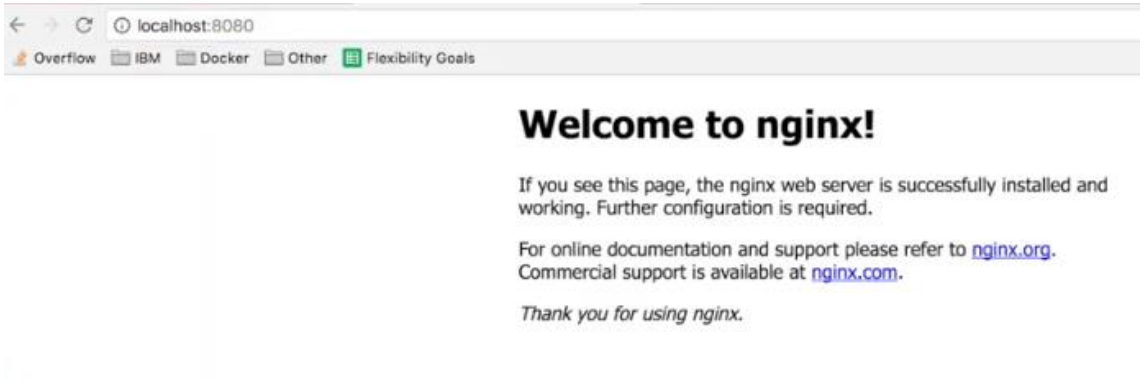
For information on the syntax of the nginx configuration files, see the official documentation (specifically the Beginner's Guide).

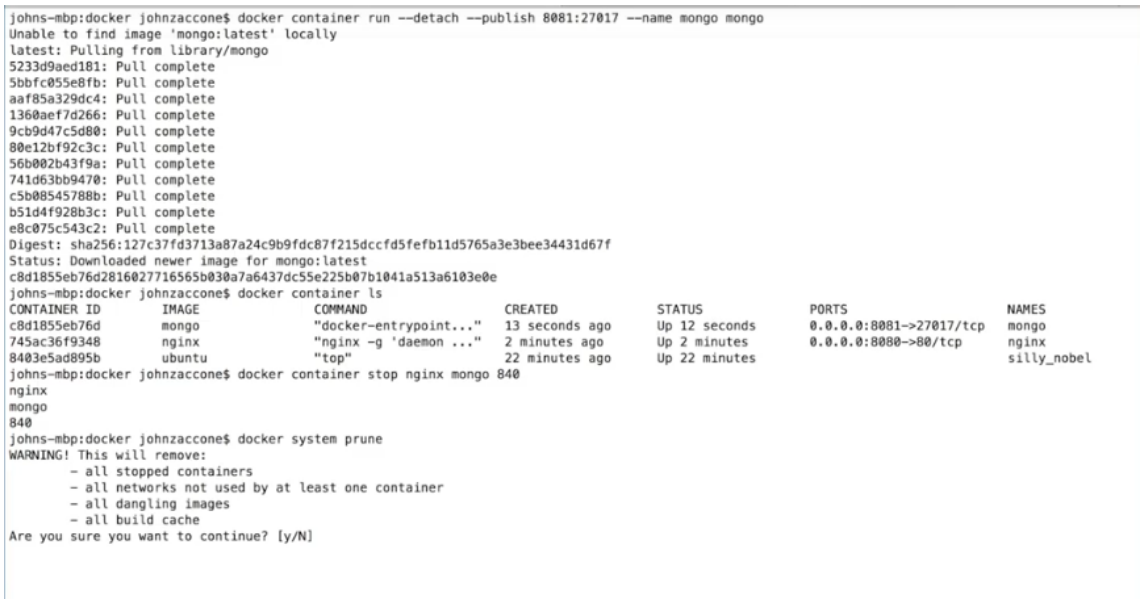
If you wish to adapt the default configuration, use something like the following to copy it from a running nginx container:

```
$ docker run --name tmp-nginx-container -d nginx
$ docker cp tmp-nginx-container:/etc/nginx/nginx.conf /host/path/nginx.conf
$ docker rm -f tmp-nginx-container
```


This can also be accomplished more cleanly using

```
FROM nginx
COPY nginx.conf /etc/nginx/nginx.conf
```





What is a Docker Image?

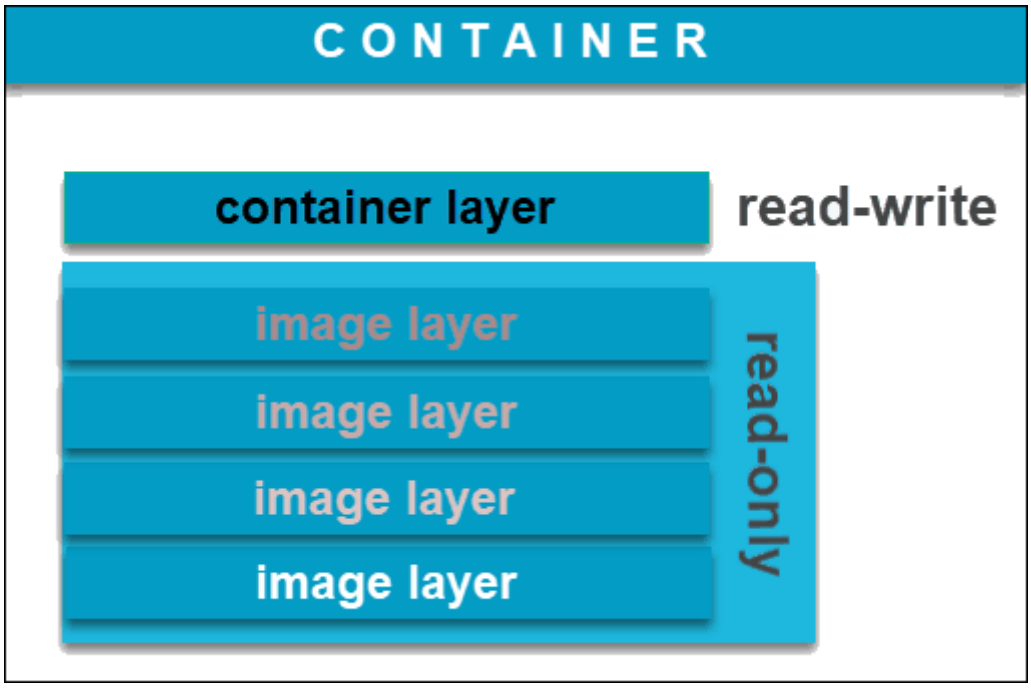
A **Docker image** is an immutable (unchangeable) file that contains the source code, libraries, dependencies, tools, and other files needed for an application to run.

Due to their **read-only** quality, these images are sometimes referred to as snapshots. They represent an application and its virtual environment at a specific point in time. This consistency is one of the great features of Docker. It allows developers to test and experiment software in stable, uniform conditions.

Since images are, in a way, just **templates**, you cannot start or run them. What you can do is use that template as a base to build a container. A container is, ultimately, just a running image. Once you create a container, it adds a writable layer on top of the immutable image, meaning you can now modify it.

The image-based on which you create a container exists separately and cannot be altered. When you run a containerized environment, you essentially create a **read-write**

copy of that filesystem (docker image) inside the container. This adds a **container layer** which allows modifications of the entire copy of the image.



You can create an unlimited number of Docker images from one **image base**. Each time you change the initial state of an image and save the existing state, you create a new template with an additional layer on top of it.

Docker images can, therefore, consist of a **series of layers**, each differing but also originating from the previous one. Image layers represent read-only files to which a container layer is added once you use it to start up a virtual environment.

What is a Docker Container?

A **Docker container** is a virtualized run-time environment where users can isolate applications from the underlying system. These containers are compact, portable units in which you can start up an application quickly and easily.

A valuable feature is the **standardization** of the computing environment running inside the container. Not only does it ensure your application is working in identical circumstances, but it also simplifies sharing with other teammates.

As containers are autonomous, they provide strong isolation, ensuring they do not interrupt other running containers, as well as the server that supports them. Docker claims that these units “provide the strongest isolation capabilities in the industry”. Therefore, you won’t have to worry about keeping your machine **secure** while developing an application.

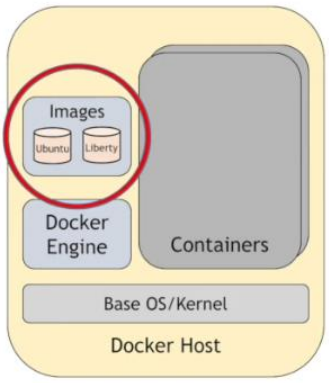
Unlike virtual machines (VMs) where virtualization happens at the hardware level, containers virtualize at the app layer. They can utilize one machine, share its kernel, and

virtualize the operating system to run isolated processes. This makes containers extremely **lightweight**, allowing you to retain valuable resources.

LAB 2 – ADD CLI/CD VALUE WITH DOCKER IMAGES

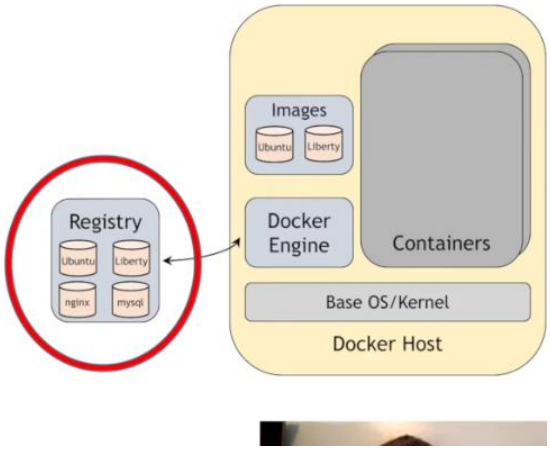
Docker images

- Tar file containing a container's filesystem + metadata
- For sharing and redistribution



Docker Registry

- Push and 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

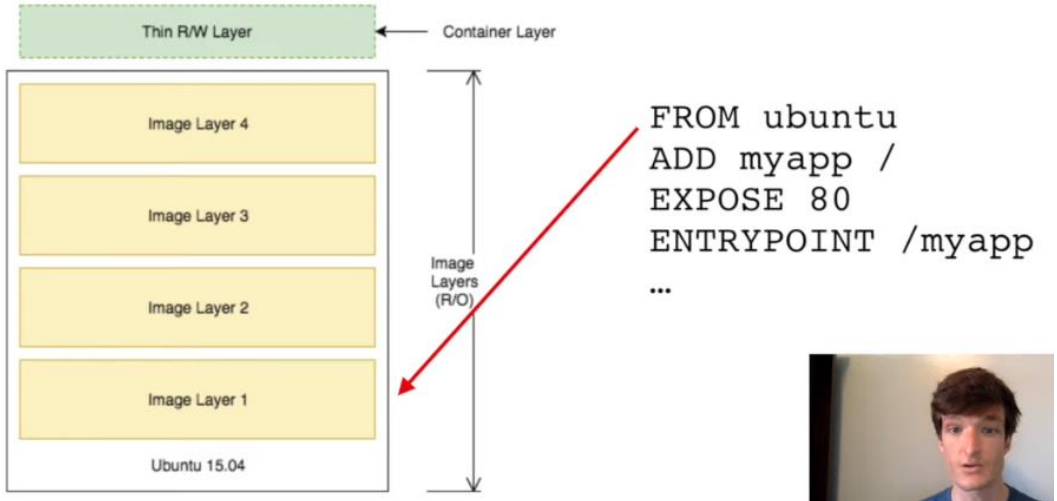


Creating a Docker image - with Docker build

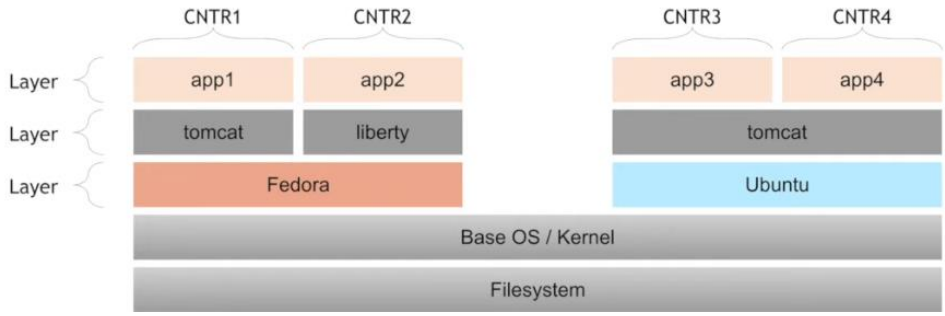
- Create a "Dockerfile"
  - List of instructions for how to construct the container
- `docker build -f Dockerfile`

```
$ cat Dockerfile
FROM ubuntu
ADD myapp /
EXPOSE 80
ENTRYPOINT /myapp
```

Secret sauce: Docker image layers



Docker image layers



- union file system
  - Merge image layers into single file system for each container
- copy-on-write
  - copies files that are edited up to top writable layer

- advantages
  - More containers per host (save $$ on infrastructure)
  - Faster start-up/download time – base layers are "cached"

Docker image layers

```
$ docker push jzaccone/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
```

Lab objectives

- 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!

Lab 2 overview

In this lab, build on your knowledge from Lab 1 where you used Docker commands to run containers. Create a custom Docker image built from a Dockerfile. After you build the image, you will push it to a central registry where it can be pulled to be deployed on other environments.

Also, you'll get a brief understanding of image layers and how Docker incorporates copy-on-write and the union file system to efficiently store images and run containers. You will use a few Docker commands in this lab. See the documentation on for information on available commands.

Prerequisites

You must have Docker installed or use Play-With-Docker.

1. Create a Python app (without using Docker)

1. 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
```

   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.) Don't worry if you are not too familiar with Python or Flask. These concepts can be applied to an application written in any language.

12. **Optional:** If you have Python and pip installed, run this app locally. If not, move on to the next section of this lab.

```
$ python3 --version
Python 3.6.1
$ pip3 --version
pip 9.0.1 from /usr/local/lib/python3.6/site-packages (python 3.6)
$ pip3 install flask
Requirement already satisfied: flask in /usr/local/lib/python3.6/site-packages
Requirement already satisfied: Werkzeug>=0.7 in /usr/local/lib/python3.6/site-packages (from flask)
Requirement already satisfied: itsdangerous>=0.21 in /usr/local/lib/python3.6/site-packages (from flask)
Requirement already satisfied: Jinja2>=2.4 in /usr/local/lib/python3.6/site-packages (from flask)
Requirement already satisfied: click>=2.0 in /usr/local/lib/python3.6/site-packages (from flask)
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.6/site-packages (from Jinja2>=2.4->flask)
johns-mbp:test johnzaccone$ pip3 install flask
Requirement already satisfied: flask in /usr/local/lib/python3.6/site-packages
Requirement already satisfied: itsdangerous>=0.21 in /usr/local/lib/python3.6/site-packages (from flask)
Requirement already satisfied: Jinja2>=2.4 in /usr/local/lib/python3.6/site-packages (from flask)
```

```
28.  Requirement already satisfied: click>=2.0 in
   /usr/local/lib/python3.6/site-packages (from flask)
29.  Requirement already satisfied: Werkzeug>=0.7 in
   /usr/local/lib/python3.6/site-packages (from flask)
30.  Requirement already satisfied: MarkupSafe>=0.23 in
   /usr/local/lib/python3.6/site-packages (from Jinja2>=2.4->flask)
31.  $ python3 app.py
    * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

2. Create and build the Docker image

2. 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.

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

```
2. FROM python:3.6.1-alpine
3. RUN pip install flask
4. CMD ["python","app.py"]
5. 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.

     For security reasons, you must understand the layers that you build your docker image on top of. For that reason, it is highly recommended to only use official images found in the Docker Hub, or noncommunity images found in the Docker Store. These images are vetted to meet certain security requirements, and also have very good documentation for users to follow. You can find more information about this Python base image and other images that you can use on the Docker store.

For a more complex application, you might need to use a `FROM` image that is higher up the chain. For example, the parent Dockerfile for your Python application starts with `FROM alpine`, then specifies a series of `CMD` and `RUN` commands for the image. If you needed more control, you could start with `FROM alpine` (or a different distribution) and run those steps yourself. However, to start, it's recommended that you use an official image that closely matches your needs.

- **`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.

6. 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:
sha256:768360b3fad01adffcf5ad9eccb4aa3ccc83bb0ed341bbdc45951e893
35082ce
 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/a8/66/24d655233c757e178d45dea2de22a04
c6d92766abfb741129a
   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/e39a54a87bcbe25308fa3ca64e8ddc7
5d9b3e5afa21ee32d57
 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
```

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

```
54.  $ docker image ls
55.
56.  REPOSITORY           TAG                 IMAGE ID
   CREATED             SIZE
57.  python-hello-world   latest              f1b2781b3111        26
   seconds ago      99.3MB
58.  python                3.6.1-alpine        c86415c03c37        8
   days ago          88.7MB
```

Notice that your base image, python:3.6.1-alpine, is also in your list.

3. Run the Docker image

3. Run the Docker image

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

1. Run the Docker image:

```
2. $ 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.

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

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

4. 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. 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.

4. Push to a central registry

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

   For 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.

2. 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:
```

5. 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
```

6. 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
The push refers to a repository [docker.io/jzaccone/python-hello-world]
2bce026769ac: Pushed
 64d445ecbe93: Pushed
 18b27eac38a1: Mounted from library/python
 3f6f25cd8b1e: Mounted from library/python
 b7af9d602a0f: Mounted from library/python
 ed06208397d5: Mounted from library/python
 5accac14015f: Mounted from library/python
 latest: digest: sha256:508238f264616bf7bf962019d1a3826f8487ed6a48b80bf41fd3996c7175fd0f size: 1786
```

17. 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.

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.

5. Deploy a change

5. Deploy a change

1. 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 Hub registry.

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

```
$  docker image build -t jzaccone/python-hello-world .
Sending build context to Docker daemon  3.072kB
Step 1/4 : FROM python:3.6.1-alpine
 ---> c86415c03c37
Step 2/4 : RUN pip install flask
 ---> Using cache
 ---> ce41f2517c16
Step 3/4 : CMD python app.py
  ---> Using cache
  ---> 0ab91286958b
 Step 4/4 : COPY app.py /app.py
  ---> 3e08b2eeace1
 Removing intermediate container 23a955e881fc
 Successfully built 3e08b2eeace1
 Successfully tagged jzaccone/python-hello-world:latest
```

   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:
sha256:91874e88c14f217b4cab1dd5510da307bf7d9364bd39860c9cc868857
3ab1a3a 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.

6. Understand image layers

6. Understand image layers

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

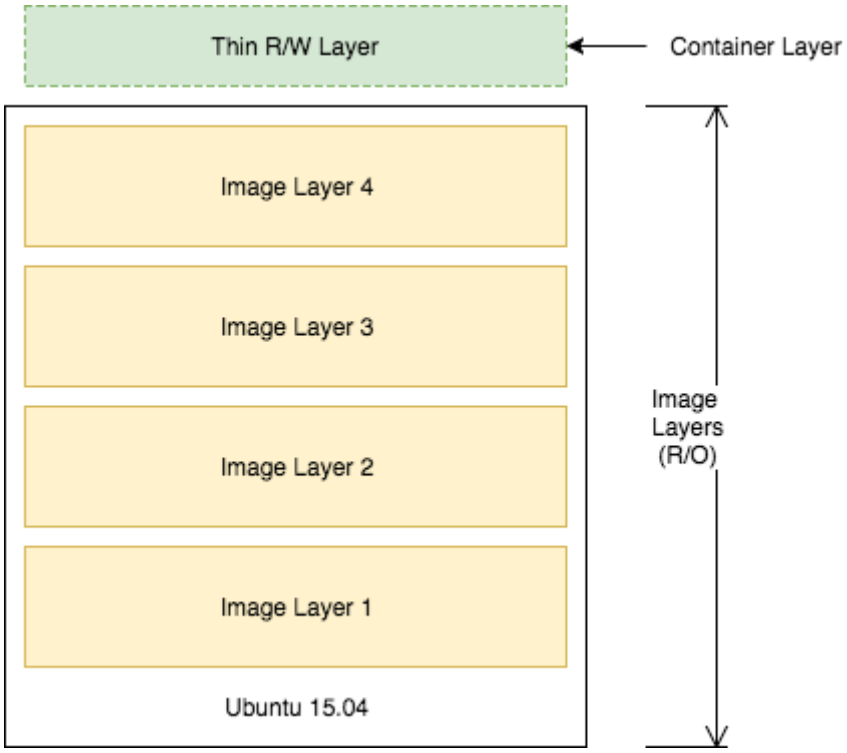
Consider the Dockerfile that you created before:

```
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.

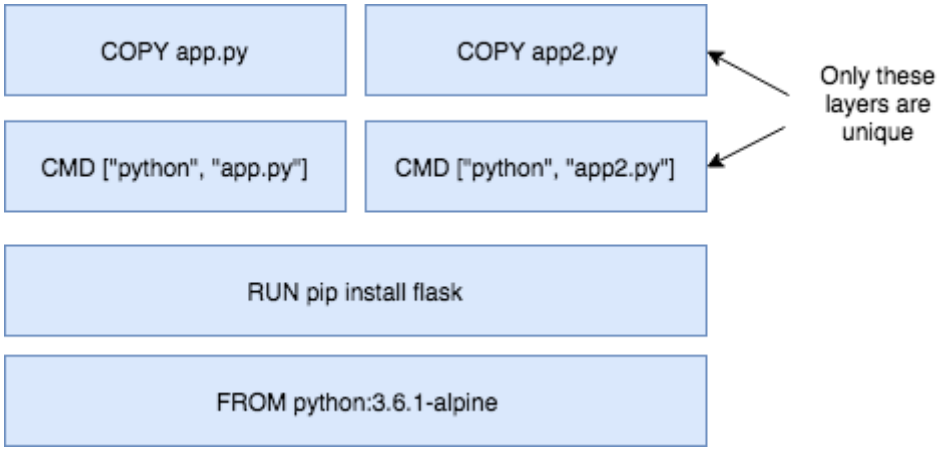
Each layer of the image is read-only except for the top layer, which is created for the container. The read/write container layer implements "copy-on-write," which means that files that are stored in lower image layers are pulled up to the read/write container layer only when edits are being made to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

```
$ 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:
sha256:91874e88c14f217b4cab1dd5510da307bf7d9364bd39860c9cc8688573ab1a3
a 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
<missing>           8 days ago          /bin/sh -c #(nop)  ENV
PYTHON_PIP_VERSION=...   0B
<missing>           8 days ago          /bin/sh -c cd /usr/local/bin
&& ln -s idl...   32B
<missing>           8 days ago          /bin/sh -c set -ex  && apk add
--no-cache ...   77.5MB
<missing>           8 days ago          /bin/sh -c #(nop)  ENV
PYTHON_VERSION=3.6.1     0B
<missing>           8 days ago          /bin/sh -c #(nop)  ENV
GPG_KEY=0D96DF4D411...   0B
<missing>           8 days ago          /bin/sh -c apk add --no-cache
ca-certificates   618kB
<missing>           8 days ago          /bin/sh -c #(nop)  ENV
LANG=C.UTF-8             0B
<missing>           8 days ago          /bin/sh -c #(nop)  ENV
PATH=/usr/local/bin...   0B
<missing>           9 days ago          /bin/sh -c #(nop)  CMD
["/bin/sh"]              0B
<missing>           9 days ago          /bin/sh -c #(nop) ADD
file:cf1b74f7af8abcf...   4.81MB
```

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.

7. Remove the containers

7. Remove the containers

Completing this lab results in a lot of running containers on your host. You'll stop and remove these containers.

1. Get a list of the containers running by running the command `docker container ls`:

```
$ docker container ls
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
NAMES
0b2ba61df37f        python-hello-world   "python app.py"     7
minutes ago       Up 7 minutes        0.0.0.0:5001->5000/tcp
practical_kirch
```

5. Run `docker container stop [container id]` for each container in the list that is running:

```
$ docker container stop 0b2
0b2
```

8. Remove the stopped containers by running `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:
 0b2ba61df37fb4038d9ae5d145740c63c2c211ae2729fc27dc01b82b5aaafa2
6

 Total reclaimed space: 300.3kB
```

Lab 2 summary

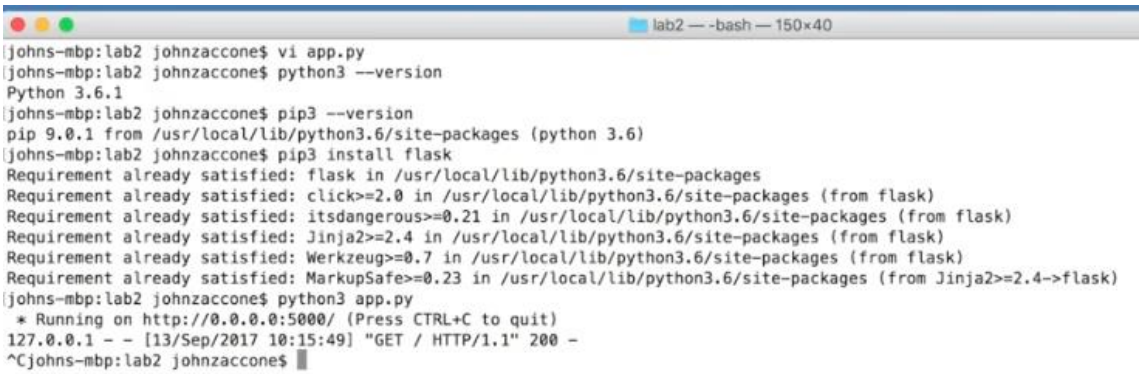
Lab 2 summary

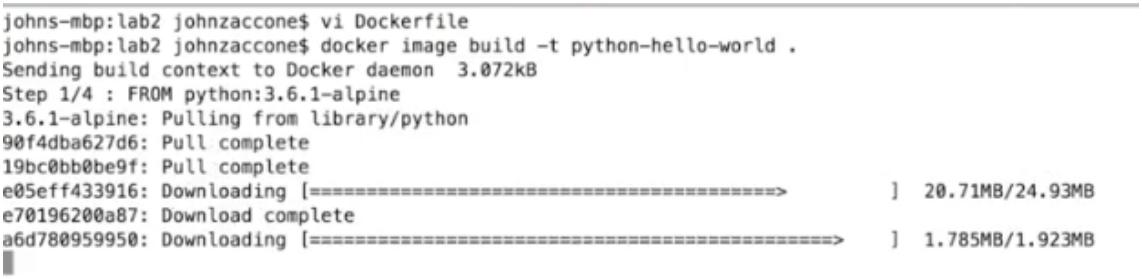
In this lab, you started adding value by creating your own custom Docker containers. Remember these key points:

- 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

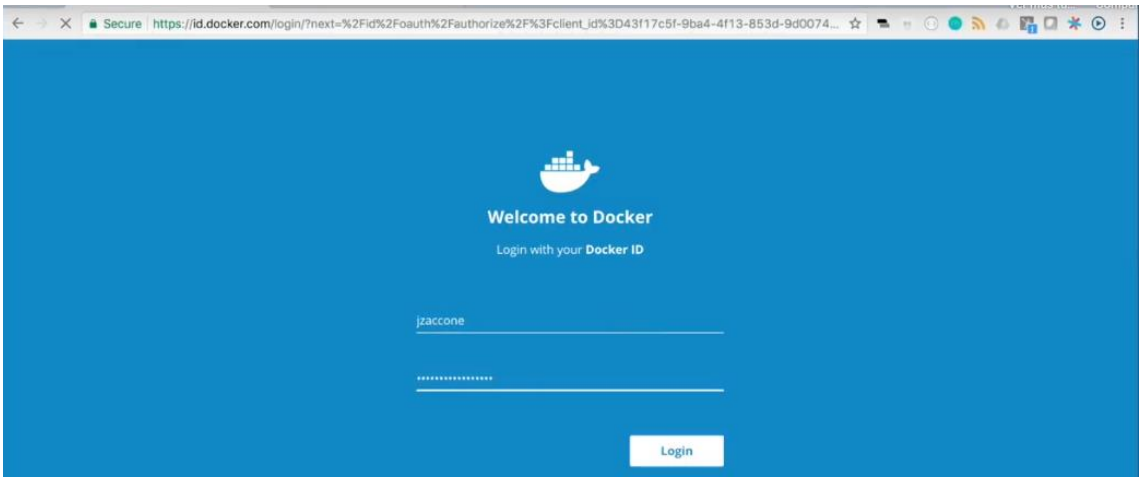
drift (version differences) when you rely on dependencies that are installed on every environment you deploy to.

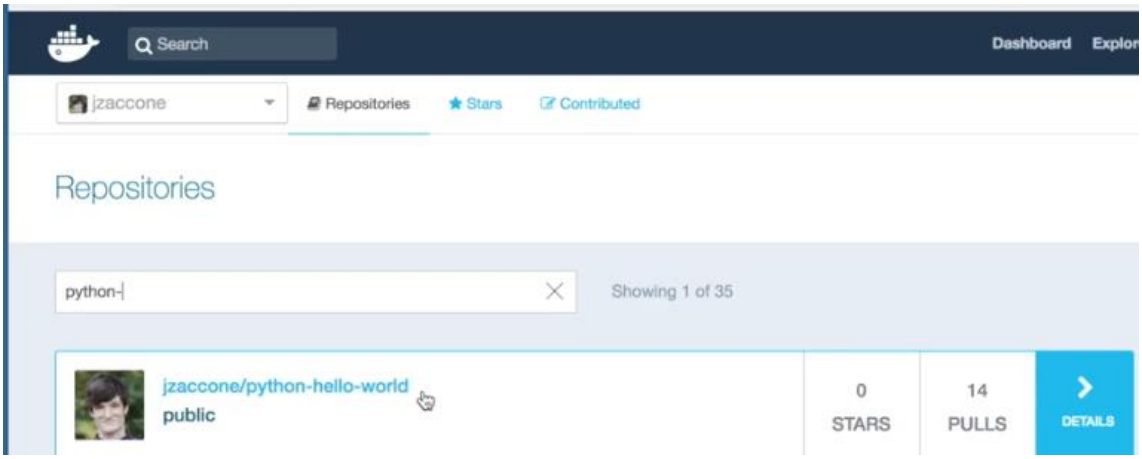
- Docker uses of the union file system and "copy-on-write" to reuse layers of images. This lowers the footprint of storing images and significantly increases the performance of starting containers.
- Image layers are cached by the Docker build and push system. There's no need to rebuild or repush image 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.





```
johns-mbp:lab2 johnzaccone$ 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 (jzaccone): jzaccone
Password:
Login Succeeded
johns-mbp:lab2 johnzaccone$ docker image tag python-hello-world jzaccone/python-hello-world
johns-mbp:lab2 johnzaccone$ docker push jzaccone/python-hello-world
The push refers to a repository [docker.io/jzaccone/python-hello-world]
e820e32eaa61: Pushing [==================================================>]  2.048kB
4a4d17448a81: Pushing [==================================================>]  11.37MB
5f354b8b5dc0: Layer already exists
f61107386c17: Layer already exists
db49993833a0: Layer already exists
58c71ea40fb0: Layer already exists
2b0fb280b60d: Layer already exists
```





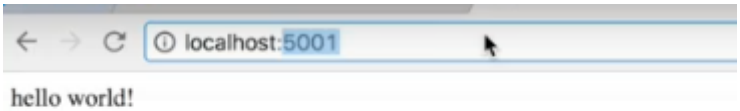

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
    return "hello beautifulworld!'

if __name__ == "__main__":
    app.run(host='0.0.0.0')
```

```
johns-mbp:lab2 johnzaccone$ vi app.py
johns-mbp:lab2 johnzaccone$ docker image build -t jzaccone/python-hello-world .
Sending build context to Docker daemon  3.072kB
Step 1/4 : FROM python:3.6.1-alpine
 ---> ddd6300d05a3
Step 2/4 : RUN pip install flask
 ---> Using cache
 ---> b831b7ce8ffa
Step 3/4 : CMD python app.py
 ---> Using cache
 ---> 2ace47b97dfc
Step 4/4 : COPY app.py /app.py
 ---> b20f05781138
Successfully built b20f05781138
Successfully tagged jzaccone/python-hello-world:latest
johns-mbp:lab2 johnzaccone$ docker container run -p 5002:5000 -d jzaccone/python-hello-world
```




```
johns-mbp:lab2 johnzaccone$ vi app.py
johns-mbp:lab2 johnzaccone$ docker image build -t jzaccone/python-hello-world .
Sending build context to Docker daemon  3.072kB
Step 1/4 : FROM python:3.6.1-alpine
 ---> ddd6300d05a3
Step 2/4 : RUN pip install flask
 ---> Using cache
 ---> b831b7ce8ffa
Step 3/4 : CMD python app.py
 ---> Using cache
 ---> 2ace47b97dfc
Step 4/4 : COPY app.py /app.py
 ---> b20f05781138
Successfully built b20f05781138
Successfully tagged jzaccone/python-hello-world:latest
johns-mbp:lab2 johnzaccone$ docker container run -p 5002:5000 -d jzaccone/python-hello-world
99839f1c185a5c516f7cac9e002357a9c6fe6842679636d7a6466863062a7e11
johns-mbp:lab2 johnzaccone$ docker push jzaccone/python-hello-world
The push refers to a repository [docker.io/jzaccone/python-hello-world]
7177478f8f31: Pushed
4a4d17448a81: Layer already exists
5f354b8b5dc0: Layer already exists
f61107386c17: Layer already exists
db49993833a0: Layer already exists
58c71ea40fb0: Layer already exists
2b0fb280b60d: Layer already exists
latest: digest: sha256:2ef5b61a634eaf33da5016a0c9f019accec1215fc190cbeebfe94f962c23482e size: 1786
johns-mbp:lab2 johnzaccone$
```

```
johns-mbp:lab2 johnzaccone$ docker container ls
CONTAINER ID        IMAGE                         COMMAND             CREATED              STATUS              PORTS                    NAMES
99839f1c185a        jzaccone/python-hello-world   "python app.py"     About a minute ago   Up About a minute   0.0.0.0:5002->5000/tcp   dazzling_dubin
sky
422f694e979c        python-hello-world            "python app.py"     4 minutes ago        Up 4 minutes        0.0.0.0:5001->5000/tcp   festive_wright
johns-mbp:lab2 johnzaccone$ docker container stop 99 422
99
422
johns-mbp:lab2 johnzaccone$ docker system prune
WARNING! This will remove:
        - all stopped containers
        - all networks not used by at least one container
        - all dangling images
        - all build cache
Are you sure you want to continue? [y/N] y
Deleted Containers:
99839f1c185a5c516f7cac9e002357a9c6fe6842679636d7a6466863062a7e11
422f694e979cd3c25d8f1deb953946274f6f5f0b99914d45d79296eabb18aa0a

Total reclaimed space: 600.5kB
johns-mbp:lab2 johnzaccone$
```

LAB 3 ORCHESTRATE APPLICATIONS WITH DOCKER SWARM

Why containers are appealing to users

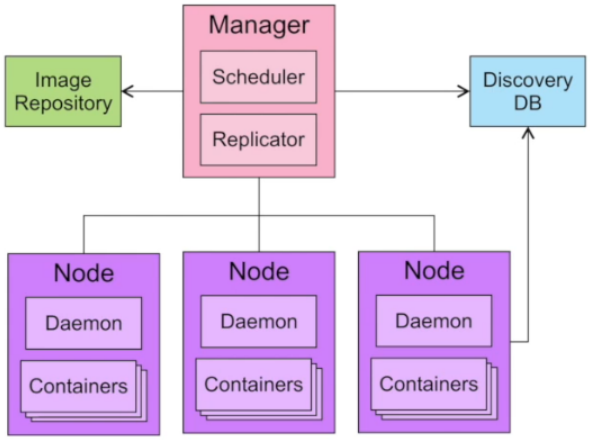
- No more “Works on my machine”
- Lightweight and fast
- Better resource utilization
  - Can fit far more containers than VMs into a host
- Standard developer to operations interface
- Ecosystem and tooling

But wait? What about production?

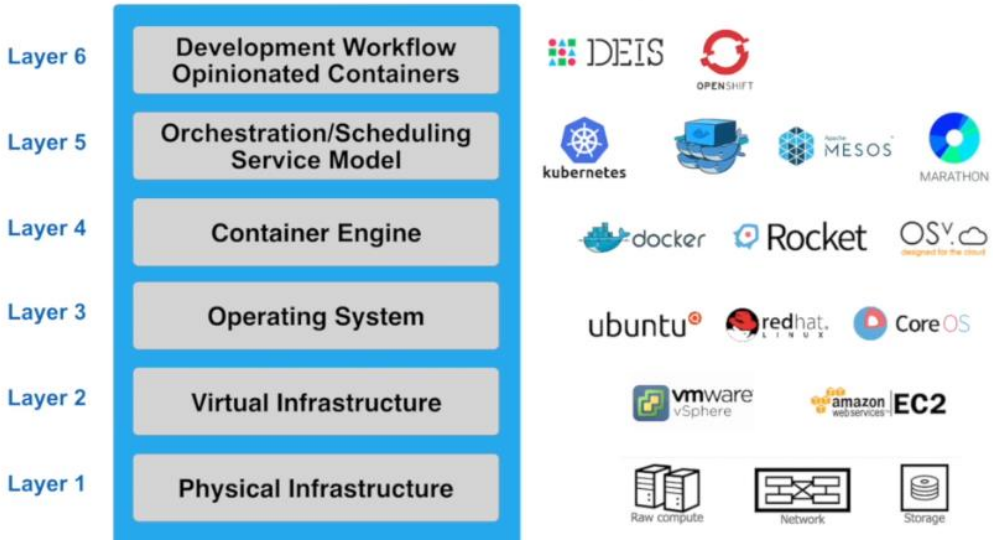
- Automated scheduling and scaling
- Service discovery
- Zero downtime deployments
- High availability and fault tolerance
- A/B deployments

What is container orchestration?

- Cluster management
- Scheduling
- Service discovery
- Replication
- Health management
- Declare desired state
  - Active reconciliation

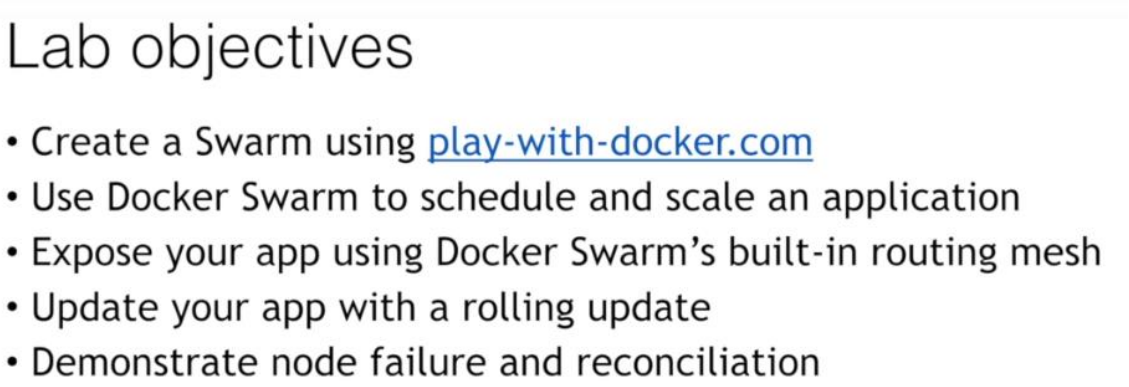


Container ecosystem layers



Container orchestration solutions

- Hosted Solutions
  - IBM Cloud Container Service
  - Amazon ECS
  - Azure Containers
  - Google Containers Engine



Lab 3 overview

Lab 3 overview

So far, you have learned how to run applications by using Docker on your local machine. But what about running Dockerized applications in production? A number of problems come with building an application for production, for example:

- Scheduling services across distributed nodes
- Maintaining high availability
- Implementing reconciliation
- Scaling
- Logging

Several orchestration solutions are available to help you solve some of these problems. One example is the IBM Cloud Kubernetes Service, which uses Kubernetes to run containers in production.

Before you learn about Kubernetes, you will learn how to orchestrate applications by using Docker Swarm. Docker Swarm is the orchestration tool that is built in to the Docker Engine.

This lab uses a few Docker commands. For a complete list of commands, see the Docker documentation.

To complete a lab about orchestrating an application that is deployed across multiple hosts, you need multiple hosts. To make things easier, this lab uses the multi-node support provided by Play-with-Docker. This is the easiest way to test Docker Swarm without having to install Docker on multiple hosts.

Be sure you have a Docker Hub account.

1. Create your first swarm

1. Create your first swarm

In this section, you will create your first swarm by using Play-with-Docker.

1. Navigate to Play-with-Docker. 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.

14. 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!

15. 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).

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.

2. Deploy your first service

2. 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.

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,typ
e=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.

4. 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
```

8. 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.

9. 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.

3. Scale your service

3. 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

   ```
   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 4 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.

2. Check the running instances.

   After a few seconds, you should see that the swarm did its job and successfully started 4 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               PORTS
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
```

3. 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.

4. 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`.

5. Reconcile problems with containers

5. 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.

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:

```
2. $ docker service create --detach=true --name nginx2 --replicas=5
   --publish 81:80  --mount
   source=/etc/hostname,target=/usr/share/nginx/html/index.html,typ
   e=bind,ro nginx:1.12
3. aiqdh5n9fyacgvb2g82s412js
```

4. 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:

```
Every 1s: docker service ps nginx1 2                                                                                2017-05-12 15:29:20

ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE            ERROR               PORTS
6koehbhsfbi7        nginx2.1            nginx:1.12          node3               Running             Running 21 seconds ago
dou2brjfr6lt        nginx2.2            nginx:1.12          node1               Running             Running 26 seconds ago
8jc41tgwowph        nginx2.3            nginx:1.12          node2               Running             Running 27 seconds ago
n5n8zryzg6g6        nginx2.4            nginx:1.12          node1               Running             Running 26 seconds ago
cnofhk1v5bd8        nginx2.5            nginx:1.12          node2               Running             Running 27 seconds ago
[node1] (loc
```

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

```
6. $ 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.

7. 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.

```
$ docker service ps nginx2
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE            ERROR               PORTS
jeq4604k1v9k        nginx2.1            nginx:1.12          node1               Running             Running 5 seconds ago
6koehbhsfbi7         \_ nginx2.1        nginx:1.12          node3               Shutdown            Running 21 seconds ago
dou2brjfr6lt        nginx2.2            nginx:1.12          node1               Running             Running 26 seconds ago
8jc41tgwowph        nginx2.3            nginx:1.12          node2               Running             Running 27 seconds ago
n5n8zryzg6g6        nginx2.4            nginx:1.12          node1               Running             Running 26 seconds ago
cnofhk1v5bd8        nginx2.5            nginx:1.12          node2               Running             Running 27 seconds ago
[node1] (loc
```

6. Determine how many nodes you need

6. Determine how many nodes you need

In this lab, your 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 mucprobleh 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.

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.

Lab 3 summary

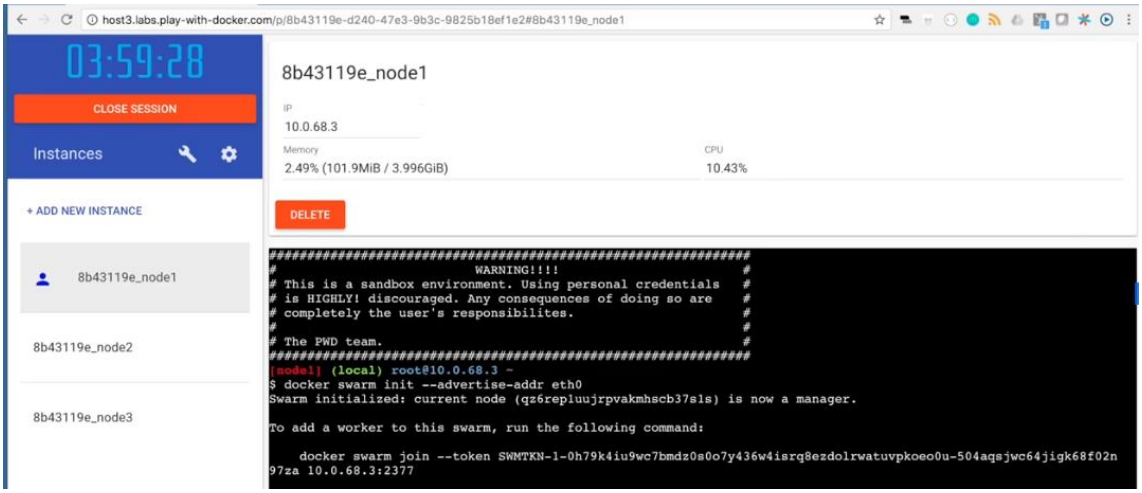
In this lab, you got an introduction to problems that come with running containers in production, such as scheduling services aenvcross distributed nodes, maintaining high availability, implementing reconciliation, scaling, and logging. You used the orchestration tool that comes built-in to the Docker Engine, Docker Swarm, to address some of these issues.

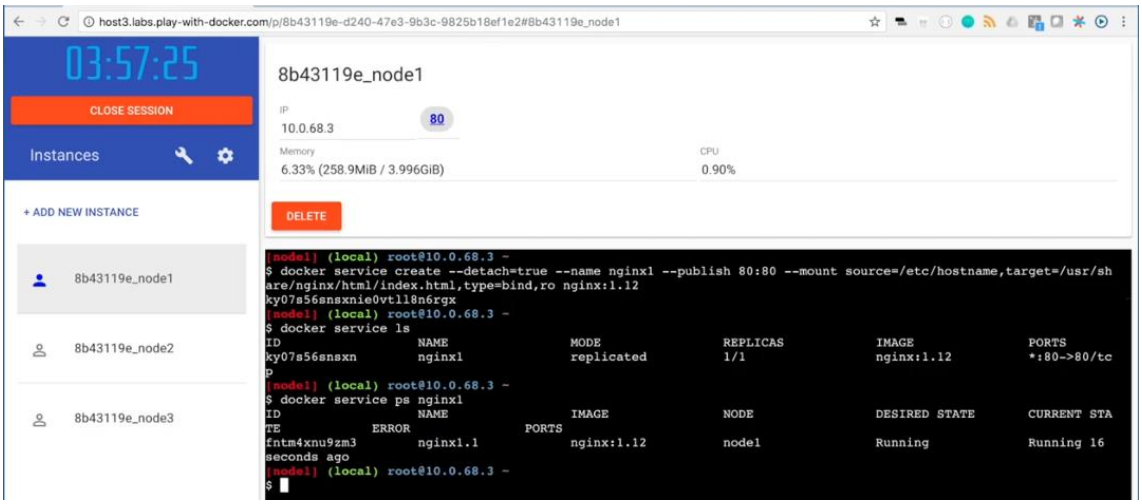
Remember these key points:

- Docker Swarm schedules services by using a declarative language. You declare the state, and the swarm attempts to maintain and reconcile to make sure the actual state equals the desired state.
- Docker Swarm is composed of manager and worker nodes. Only managers can maintain the state of the swarm and accept commands to modify it. Workers have high scalability and are only used to run containers. By default, managers can also run containers.
- The routing mesh built into Docker Swarm means that any port that is published at the service level will be exposed on every node in the swarm. Requests to a published

service port will be automatically routed to a container of the service that is running in the swarm.

- You can use other tools to help solve problems with orchestrated, containerized applications in production, including Docker Swarm and the [IBM Cloud Kubernetes Service].






```
[node1] (local) root@10.0.68.3 ~
$ docker service update --replicas=5 --detach=true nginx1
nginx1
[node1] (local) root@10.0.68.3 ~
$ docker service ps nginx1
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE            ERROR               PORTS
fntm4xnu9zm3        nginx1.1            nginx:1.12          node1               Running             Running about a minute ago
rh3lmo63of69        nginx1.2            nginx:1.12          node2               Running             Running 2 seconds ago
uzpn5e90cz3w        nginx1.3            nginx:1.12          node2               Running             Running 3 seconds ago
j58qngik7rhm        nginx1.4            nginx:1.12          node1               Running             Running 8 seconds ago
srwz2tt0ix1g        nginx1.5            nginx:1.12          node3               Running             Running 3 seconds ago
```

```
j58qngik7rhm        nginx1.4            nginx:1.12          node1               Running             Running 8 seconds ago
srwz2tt0ix1g        nginx1.5            nginx:1.12          node3               Running             Running 3 seconds ago
[node1] (local) root@10.0.68.3 ~
$ curl localhost:80
node3
[node1] (local) root@10.0.68.3 ~
$ curl localhost:80
node2
[node1] (local) root@10.0.68.3 ~
$ curl localhost:80
node1
[node1] (local) root@10.0.68.3 ~
$ curl localhost:80
node1
[node1] (local) root@10.0.68.3 ~
$ curl localhost:80
node2
[node1] (local) root@10.0.68.3 ~
$ curl localhost:80
node3
[node1] (local) root@10.0.68.3 ~
$ docker service logs nginx1
```

```
[node1] (local) root@10.0.68.3 ~
$ docker service update --image nginx:1.13 --detach=true nginx1
nginx1
[node1] (local) root@10.0.68.3 ~
$ docker service ps nginx1
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE            ERROR               PORTS
fntm4xnu9zm3        nginx1.1            nginx:1.12          node1               Running             Running 5 minutes ago
rh3lmo63of69        nginx1.2            nginx:1.12          node2               Running             Running 3 minutes ago
v8rol0zmhz7d        nginx1.3            nginx:1.13          node2               Ready               Ready less than a second ago
uzpn5e90cz3w         \_ nginx1.3        nginx:1.12          node2               Shutdown            Running 2 seconds ago
j58qngik7rhm        nginx1.4            nginx:1.12          node1               Running             Running 3 minutes ago
i85hgklvu9w9        nginx1.5            nginx:1.13          node3               Running             Running 3 seconds ago
srwz2tt0ix1g         \_ nginx1.5        nginx:1.12          node3               Shutdown            Shutdown 3 seconds ago
[node1] (local) root@10.0.68.3 ~
$ docker service ps nginx1
```

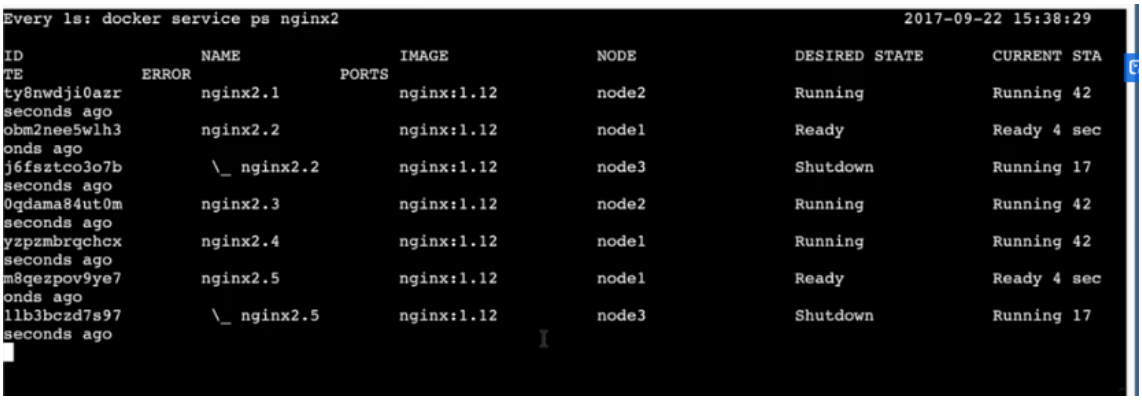
```
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STA
TE                    ERROR               PORTS
wmfbfmy0pcey        nginx1.1            nginx:1.13          node1               Running             Running 4 s
econds ago
fntm4xnu9zm3         \_ nginx1.1        nginx:1.12          node1               Shutdown            Shutdown 5
seconds ago
xixbz7ct72k8        nginx1.2            nginx:1.13          node2               Ready               Ready less
than a second ago
rh3lmo63of69         \_ nginx1.2        nginx:1.12          node2               Shutdown            Running les
s than a second ago
v8rol0zmhz7d        nginx1.3            nginx:1.13          node2               Running             Running 8 s
econds ago
uzpn5e90cz3w         \_ nginx1.3        nginx:1.12          node2               Shutdown            Shutdown 9
seconds ago
5nr2l6stxs9o        nginx1.4            nginx:1.13          node1               Running             Running les
s than a second ago
j58qngik7rhm         \_ nginx1.4        nginx:1.12          node1               Shutdown            Shutdown 1
second ago
i85hgklvu9w9        nginx1.5            nginx:1.13          node3               Running             Running 13
seconds ago
srwz2tt0ix1g         \_ nginx1.5        nginx:1.12          node3               Shutdown            Shutdown 13
 seconds ago
[node1] (local) root@10.0.68.3 ~
$
```

```
[node1] (local) root@10.0.68.3 ~
$ docker service create --detach=true --name nginx2 --replicas=5 --publish 81:80 --mount source=/etc/hostname,t
arget=/usr/share/nginx/html/index.html,type=bind,ro nginx:1.12
jlsm8openi8o7ysphg4s6cnkz
[node1] (local) root@10.0.68.3 ~
$ watch -n 1 docker service ps nginx2
```

```
Every 1s: docker service ps nginx2                                                  2017-09-22 15:38:04

ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STA
TE                ERROR               PORTS
ty8nwdji0azr        nginx2.1            nginx:1.12          node2               Running             Running 17
seconds ago
j6fsztco3o7b        nginx2.2            nginx:1.12          node3               Running             Running 16
seconds ago
0qdama84ut0m        nginx2.3            nginx:1.12          node2               Running             Running 17
seconds ago
yzpzmbrqchcx        nginx2.4            nginx:1.12          node1               Running             Running 17
seconds ago
1lb3bczd7s97        nginx2.5            nginx:1.12          node3               Running             Running 17
seconds ago
```

```
################################################################
#                          WARNING!!!!                         #
# This is a sandbox environment. Using personal credentials    #
# is HIGHLY! discouraged. Any consequences of doing so are     #
# completely the user's responsibilites.                       #
#                                                              #
# The PWD team.                                                #
################################################################
[node3] (local) root@10.0.68.5 ~
$ docker swarm join --token SWMTKN-1-0h79k4iu9wc7bmdz0s0o7y436w4isrq8ezdolrwatuvpkoeo0u-504aqsjwc64jigk68f02n97
za 10.0.68.3:2377
This node joined a swarm as a worker.
[node3] (local) root@10.0.68.5 ~
$ docker swarm leave
Node left the swarm.
[node3] (local) root@10.0.68.5 ~
$
```



ARTÍCULO

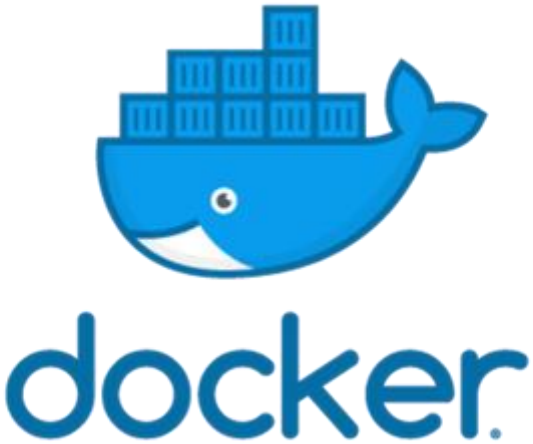
Allá por 2013 Docker empezó a ganar popularidad permitiendo a los desarrolladores crear, ejecutar y escalar rápidamente sus aplicaciones creando contenedores. Parte de su éxito se debe a ser Open Source y al apoyo de compañías como IBM, Microsoft, RedHat o Google. Docker en apenas dos años había sido capaz de convertir una tecnología nicho en una herramienta fundamental al alcance de todos gracias a su mayor facilidad de uso.

Su evolución ha sido imparable, representando actualmente **uno de los mecanismos comunes para desplegar software en cualquier servidor** por medio de contenedores software. Tanto Docker como Kubernetes, conocidos por ser uno de los más populares gestores de contenedores de software, se han convertido con méritos propios en **los estándares de *facto* de la industria**.

En Xataka

La otra guerra entre Microsoft, Google y Amazon: la batalla por controlar los servicios en la nube para desarrolladores

Google, Microsoft, Amazon, Oracle, WMware, IBM, RedHat están apostando fuertemente por estas tecnologías, ofreciendo todo tipo de servicios a los desarrolladores en la nube. Hoy por hoy todo va encaminado a ser *dockerizado*, como popularmente se refiere en castellano al hecho de empaquetar una aplicación software para ser distribuida y ejecutada mediante el uso de esos contenedores software.



Entendemos que no todo el mundo está familiarizado con el término, pero si eres desarrollador de software debes empezar a aprender más sobre ello, ya que ha sido la auténtica revolución en años de la industria del software. Gracias de él, los desarrolladores **hemos podido independizarnos en cierta medida de los *sysadmin* y hemos abrazado el concepto de *devops* más abiertamente**. Es decir, somos capaces tanto de crear código como de distribuirlo de forma sencilla sin quebraderos de cabeza.

¿Qué son los contenedores de software?

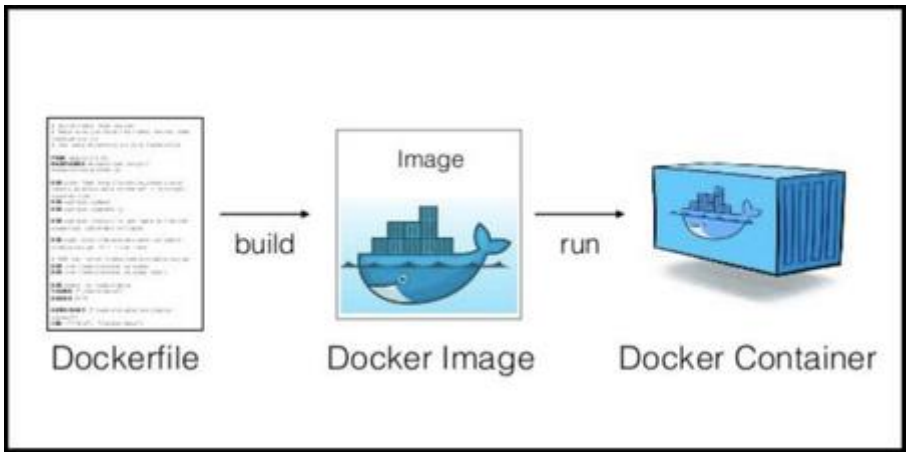
Para **explicar qué son los contenedores software** vamos a bajar al nivel más simple de abstracción. Buscando alguna analogía con el mundo real **podemos hablar de esos *containers* que vemos siendo transportados en barco de un sitio a otro**. No nos importa su contenido sino su forma modular para ser almacenados y transportados de un sitio a otro como cajas.

Algo parecido ocurre con los contenedores software. Dentro de ellos **podemos alojar todas las dependencias que nuestra aplicación necesite** para ser ejecutada: empezando por el propio código, las librerías del sistema, el entorno de ejecución o cualquier tipo de configuración. Desde fuera del contenedor no necesitamos mucho más. Dentro están aislados para ser ejecutados en cualquier lugar.

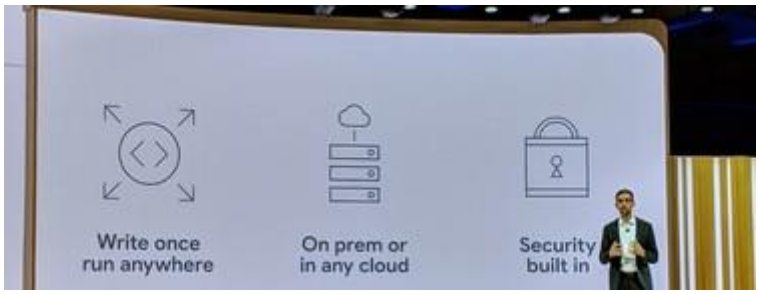
Hemos podido independizarnos en cierta medida de los sysadmin y abrazado el concepto de devops más abiertamente

Los contenedores son la solución al problema habitual, por ejemplo, de moverse entre entornos de desarrollo como puede ser una máquina local o en un entorno real de producción. Podemos probar de forma segura una aplicación sin preocuparnos de que nuestro código se comporte de forma distinta. Esto es debido a que, como nos referíamos antes, todo lo que necesitamos está dentro de ese contenedor.

Solomon Hykes, creador de Docker, lo explicaba del siguiente modo: "utilizando contenedores para ejecutar tu código solventamos **el típico quebradero de cabeza de que estés usando una versión de Python 2.7 diferente en local** o en el entorno de pruebas pero en producción la versión sea totalmente distinta como Python 3 u otras dependencias propia del entorno de ejecución, incluso el sistema operativo. Todo lo que necesitas está dentro del propio contenedor y es invariable".



En definitiva, **los contenedores representan un mecanismo de empaquetado lógico donde las aplicaciones tienen todo lo que necesitan para ejecutarse**. Describiéndolo en un pequeño archivo de configuración. Con la ventaja de poder ser versionado, reutilizado y replicado fácilmente por otros desarrolladores o por los administradores de sistemas que tenga que escalar esa aplicaciones sin necesidad de conocer internamente cómo funciona nuestra aplicación. El fichero de Docker bastará para adecuar el entorno de ejecución y configurar el servidor dónde va a ser escalado. A partir de ese fichero se puede generar una imagen que puede ser desplegada en un servidor en segundos.



En Genbeta

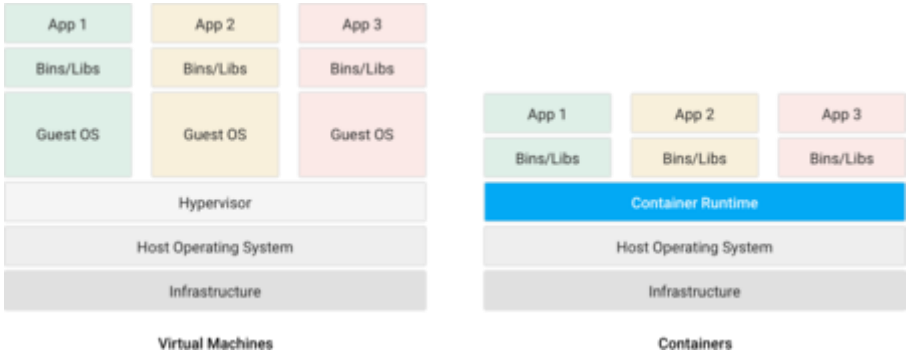
Los cuatro motivos con los que Google Cloud quiere arrebatar el control de la nube a AWS y Azure

Contenedores versus Virtualización

Una de las principales dudas es en qué se diferencia entonces un **contenedor software y una máquina virtual**. De hecho, este concepto es mucho más anterior que el de los propios contenedores.

Gracias a la virtualización somos capaces, usando un mismo ordenador, de tener distintas máquinas virtuales con su propio sistema operativo invitado, Linux o Windows. Todo ello ejecutándose en un sistema operativo anfitrión y con acceso virtualizado al hardware.

La virtualización es una práctica habitual en servidores para alojar diferentes aplicaciones o en nuestro propio entorno de trabajo para ejecutar distintos sistemas operativos, por ejemplo. Muchos alojamientos de hosting tradicionales se han basado en crear máquinas virtuales limitadas sobre el mismo servidor para alojar nuestros servidores web de forma aislada, siendo compartido por una decena de clientes.



En contraposición a las máquinas virtuales, **los contenedores se ejecutan sobre el mismo sistema operativo anfitrión** de forma aislada también, pero sin necesidad un sistema operativo propio, ya que comparten el mismo Kernel, lo que los hace mucho más ligeros. De donde sacamos 3 máquinas virtuales probablemente podemos multiplicarlo por un gran número de contenedores software.

Un contenedor de Docker puede ocupar tan solo unas cuantas decenas de megas mientras que una máquina virtual, al tener que emular todo un sistema operativo, puede ocupar varios gigas de memoria. Lo cual representa un primer punto en el ahorro de coste.

La evolución de Docker ha sido imparable, representando actualmente uno de los mecanismos comunes para desplegar software en cualquier servidor por medio de contenedores software

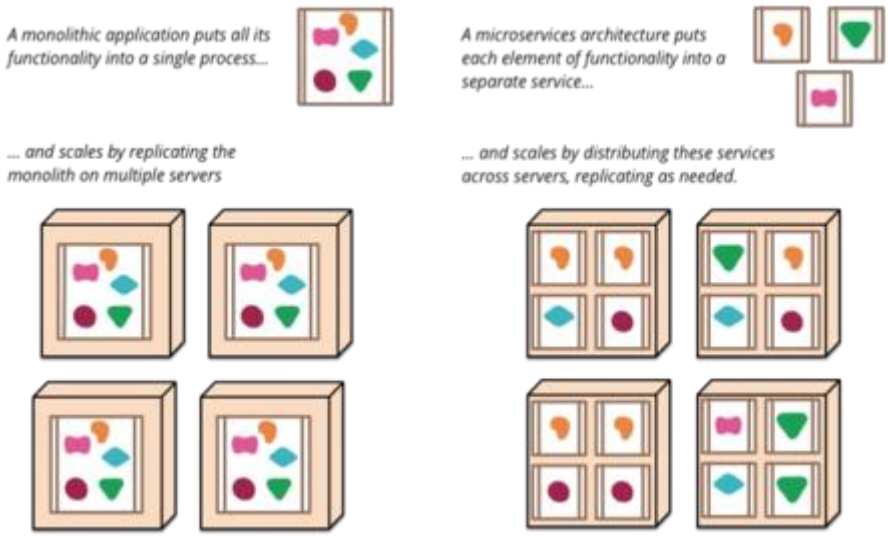
Habitualmente cada aplicación en Docker va en su propio contenedor totalmente aislado, mientras que en las VMs es habitual debido al dimensionamiento tener varias aplicaciones en la misma máquina con sus propias dependencias, mucho peor para escalar de forma horizontal.

Básicamente, los contenedores se basan en dos mecanismos para aislar procesos en un mismo sistema operativo. El primero de ellos, se trata de los **namespace que provee**

Linux, lo que permite que cada proceso solamente sea capaz de ver su propio sistema "virtual" (ficheros, procesos, interfaces de red, hostname o lo que sea). El segundo concepto son los **CGroups**, por el cual somos capaces de limitar los recursos que puede consumir (CPU, memoria, ancho de banda, etc)

De aplicaciones monolíticas a microservicios

Antes de pasar a hablar de **Kubernetes** como otro de los actores importantes de cómo ha cambiado la forma de desarrollar y escalar aplicaciones, vamos a analizar la evolución de esas arquitecturas en estos últimos años.



La definición clásica de **una aplicación monolítica se refiere a un conjunto de componentes acoplados totalmente y que tienen que ser desarrollados, desplegados y gestionados como una única entidad**. Prácticamente encajonados en un mismo proceso muy difícil de escalar, solo de forma vertical añadiendo más CPU, memoria. A todo esto hay que añadir el hecho de que para desarrollar un programador necesita tener todo ese código y ejecutar las pruebas levantando un única instancia con todo, a pesar de que el cambio que quiera realizar sea mínimo. Sin hablar de lo costoso que se convierte cada vez que se quiera hacer una nueva release tanto en desarrollo, pruebas como despliegue.

En contraposición a esto, surgió **el concepto de microservicios que permite que varias pequeñas aplicaciones se comunican entre sí** para ofrecer una funcionalidad específica concreta.

Tenemos el caso de **Netflix**, por ejemplo, una de las compañías que comenzó a hacer uso de forma más intensiva de los microservicios: Aunque no tenemos un cifra concreta, podemos estimar según los datos de muchas de sus charlas técnicas que cuenta con más de 700 microservicios.

Google es probablemente la primera compañía que se dio cuenta de que necesitaba una mejor forma de implementar y administrar sus componentes software para escalar a nivel mundial

En un ejemplo podemos hablar de un contenedor con un microservicio que se encargue de servir el vídeo según la plataforma desde donde accedemos, ya sea móvil, smart tv o tablet. También podríamos tener otro que se encargue del historial de contenido que hayamos visto, otro para las recomendaciones y, por último, otro para el pago de la suscripción.

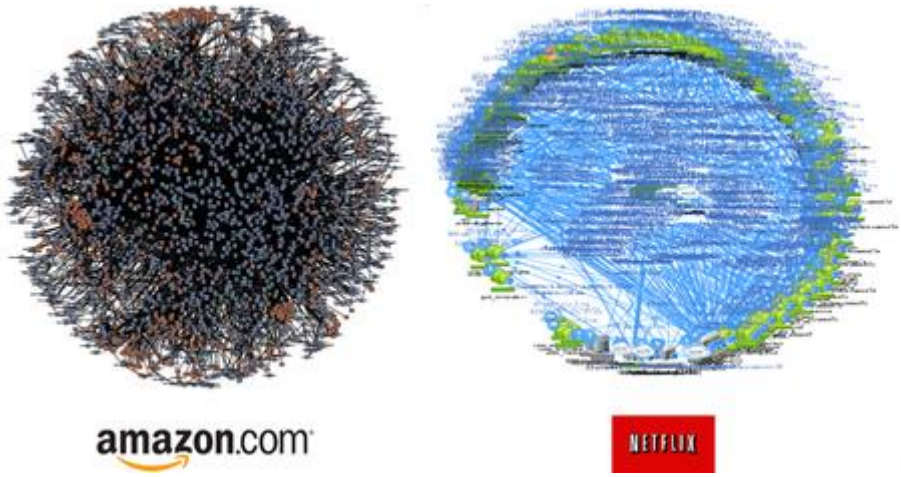


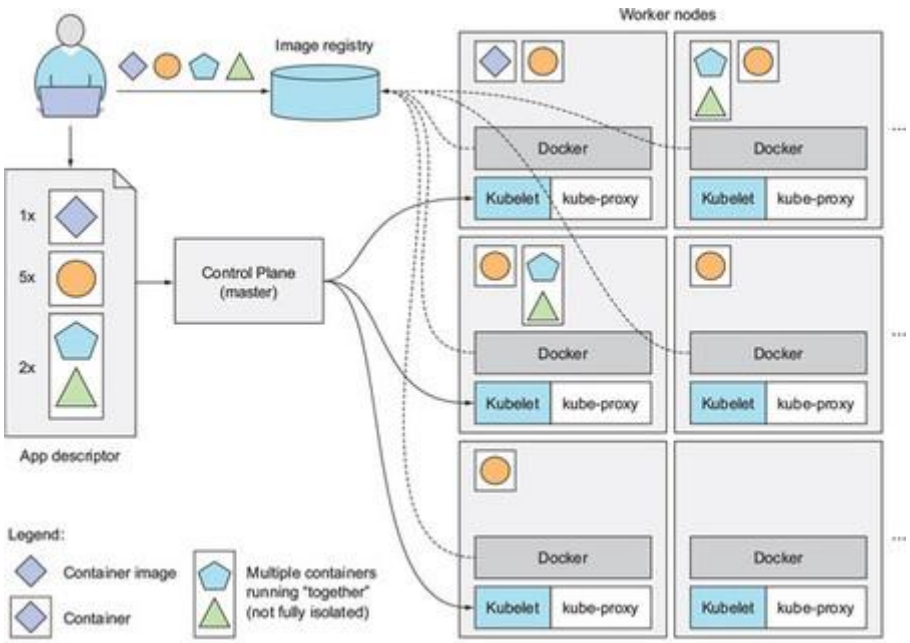
Foto a gran escala de la nube de microservicios de Netflix y Amazon

Todos ellos pueden convivir en la nube de microservicios de Netflix y comunicarse entre sí. No necesitamos modificarlos todos a la vez, ya que podemos escalar algunos de los contenedores que tenga alguno de los microservicios y ser reemplazados prácticamente al vuelo.

Después de esto podemos ver de una forma más clara como todos estos microservicios han ido adoptando la forma de contenedores de *dockerizados* comunicándose entre sí a través de nuestro sistema.

Kubernetes: la necesidad de tener un maestro de orquesta

Si el número de aplicaciones crece en nuestro sistema, se convierte en complicado de gestionar. **Docker** no es suficiente, ya que necesitamos una coordinación para hacer el despliegue, la supervisión de servicios, el reemplazo, el escalado automático y, en definitiva, la administración de los distintos servicios que componen nuestra arquitectura distribuida.



Google es probablemente la primera compañía que se dio cuenta de que necesitaba una mejor forma de implementar y administrar sus componentes software para escalar a nivel mundial. Durante años, Google desarrolló internamente **Borg** (más tarde llamado Omega).

En 2014, después de casi una década de uso intensivo interno, se presentó de forma pública Kubernetes como un sistema Open Source basado en el aprendizaje utilizando servicios a gran escala.

Fue en el la **DockerCon de 2014** cuando **Eric Brewer, VP de Engineering**, lo presentó bromeando como que era otra plataforma de orquestación más. De hecho, en la DockerCon de 2014 se presentaron una decena de sistemas similares, algunos públicos y otros internos como Facebook o Spotify. Finalmente, después de cinco años, el proyecto sigue avanzando a toda velocidad, y hoy, Kubernetes es el estándar de facto para implementar y desplegar aplicaciones distribuidas.

Lo más importante es que **Kubernetes fue diseñado para utilizarse en cualquier lugar**, de modo que puede orquestar despliegues in situ, en nubes públicas y en despliegues híbridos.

En Genbeta

3 tendencias emergentes que marcarán el futuro de la programación

El futuro de los contenedores

La adopción en el uso de contenedores continuará creciendo. También estamos viendo cierta estandarización en torno a Kubernetes y Docker. Esto impulsará el crecimiento de un gran número herramientas de desarrollo relacionadas.

El stack tecnológico empieza a madurar bastante y casi todos **los proveedores empiezan a ser compatibles entre sí** gracias a Docker y Kubernetes. Google, Microsoft, Amazon o IBM, por ejemplo, ya lo son y trabajan bajo un mismo estándar. La lucha ahora se encuentra en mover toda esa carga de trabajo que aún no está en la nube: la nube híbrida.

Quedan retos pendientes como **seguir simplificando la curva de aprendizaje**, aunque ya se ha mejorado bastante si echamos la vista a los últimos cinco años. A pesar de ello, los desarrolladores necesitan todavía aprender cómo producir una imagen de Docker, cómo implementarla en un sistema de orquestación, cómo configurarla y más detalles de seguridad. Algo nada trivial al principio. Estamos seguros de que en poco tiempo veremos cómo eso se simplifica aún más, ya que los desarrolladores trabajarán sobre niveles de abstracción superiores, gracias al ecosistema creciente en torno a Docker y Kubernetes.