

# Introduction to Docker

Instructor:

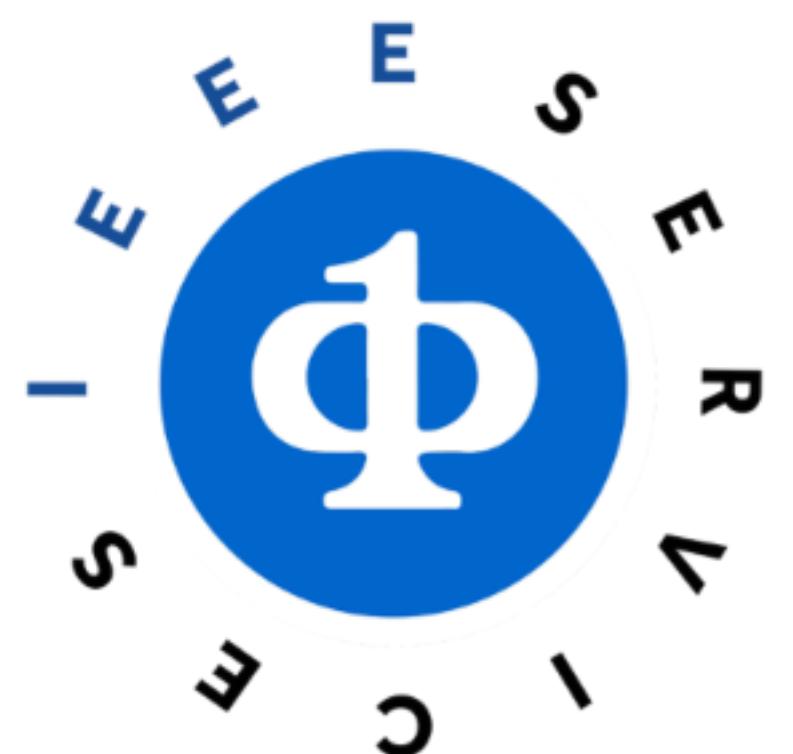
**John J Rofrano**

Senior Technical Staff Member, DevOps Champion

IBM T.J. Watson Research Center

[rofrano@us.ibm.com](mailto:rofrano@us.ibm.com)

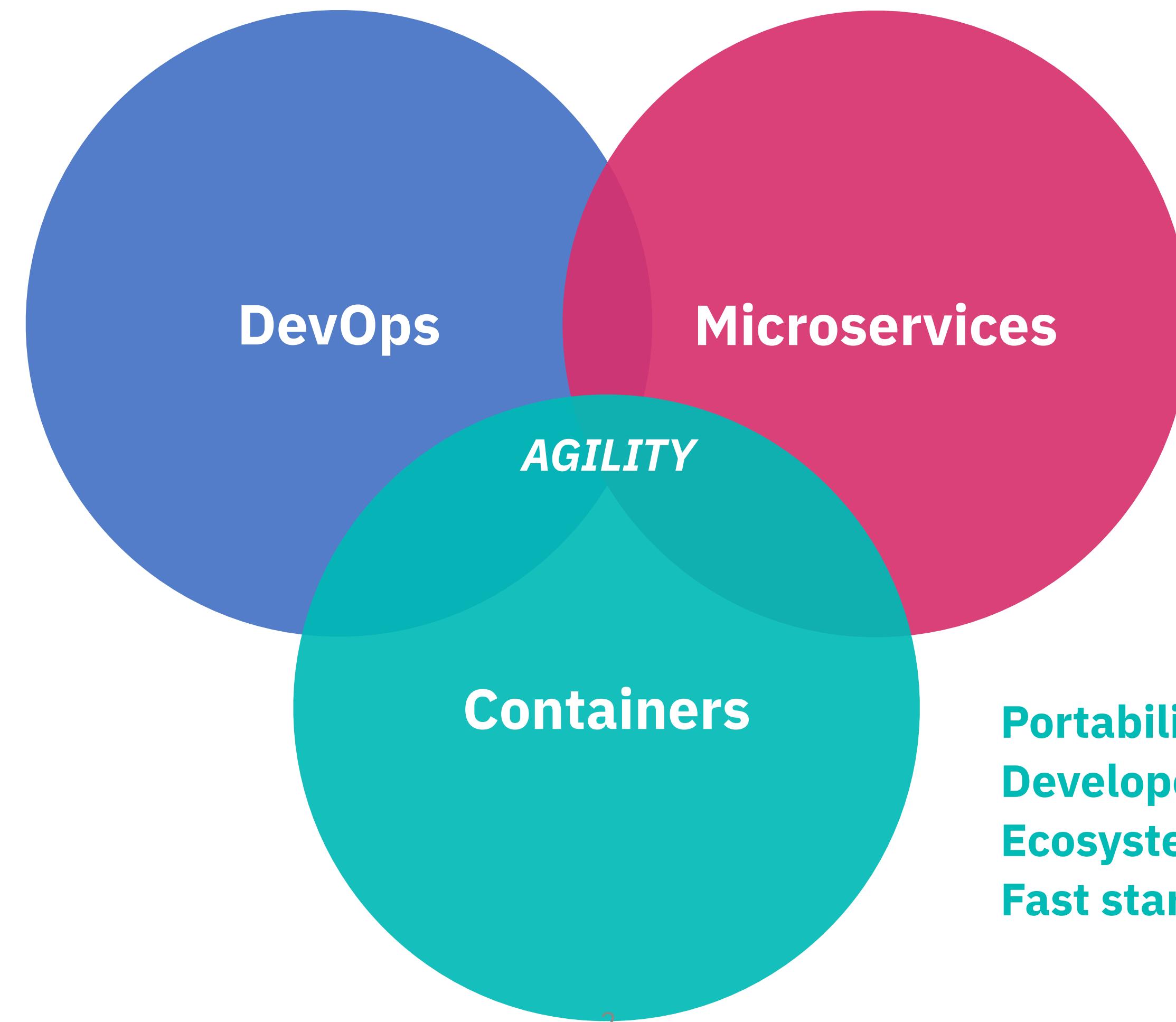
@JohnRofrano 



# THE PERFECT STORM



Cultural Change  
Automated Pipeline  
Everything as Code  
Immutable Infrastructure



Loose Coupling/Binding  
RESTful APIs  
Designed to resist failures  
Test by break / fail fast

Portability  
Developer Centric  
Ecosystem enabler  
Fast startup

# THE PERFECT STORM: Containers



Cultural Change  
Automated Pipeline  
Everything as Code  
Immutable Infrastructure

DevOps

Microservices

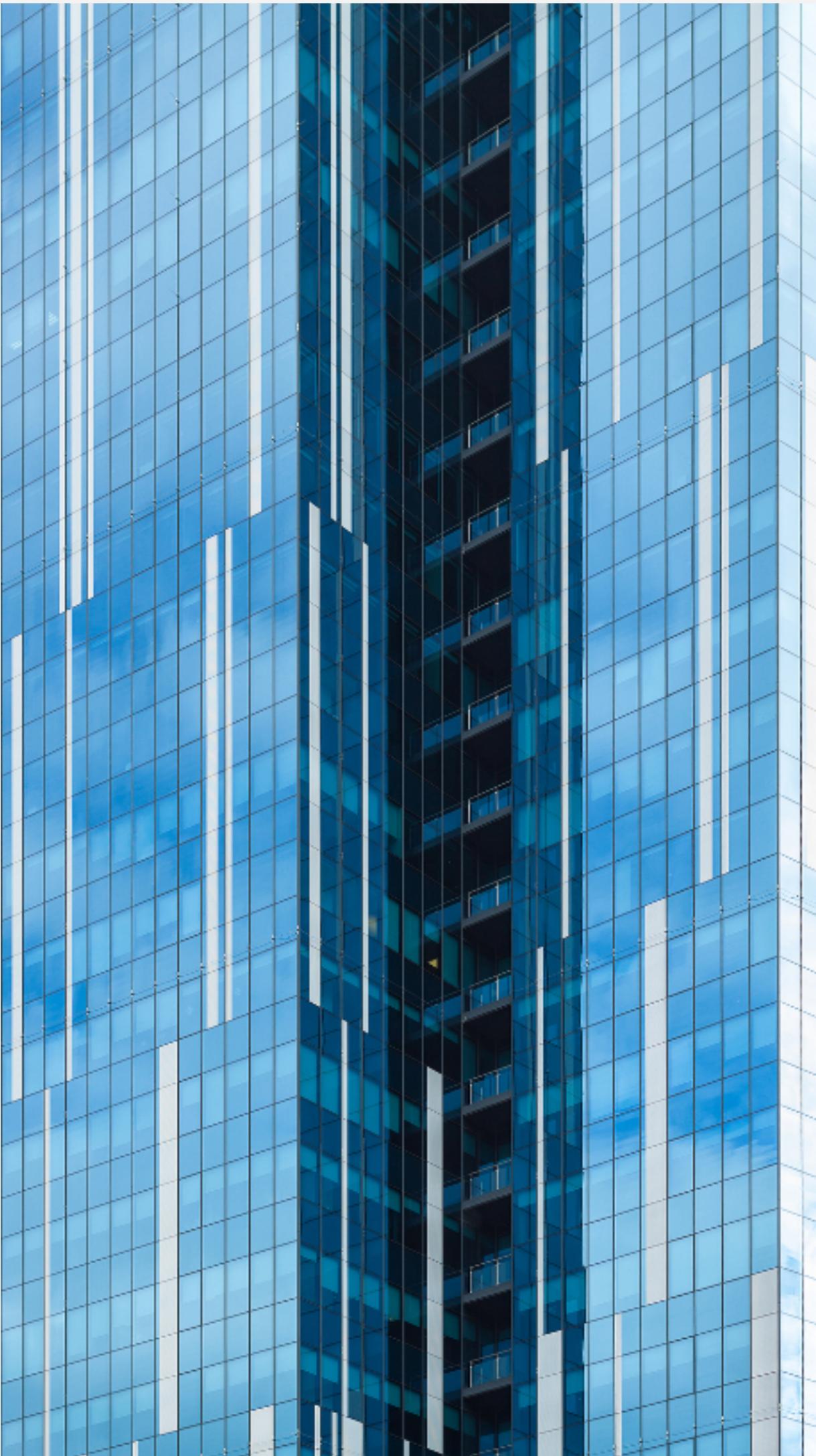
Containers

Loose Coupling/Binding  
RESTful APIs  
Designed to resist failures  
Test by break / fail fast

**Portability**  
**Developer Centric**  
**Ecosystem enabler**  
**Fast startup**

# What Will You Learn?

- What Docker is and isn't
- Why would you use Docker
- How to use Existing Docker Images
- How to Create your own Docker Images from Dockerfiles



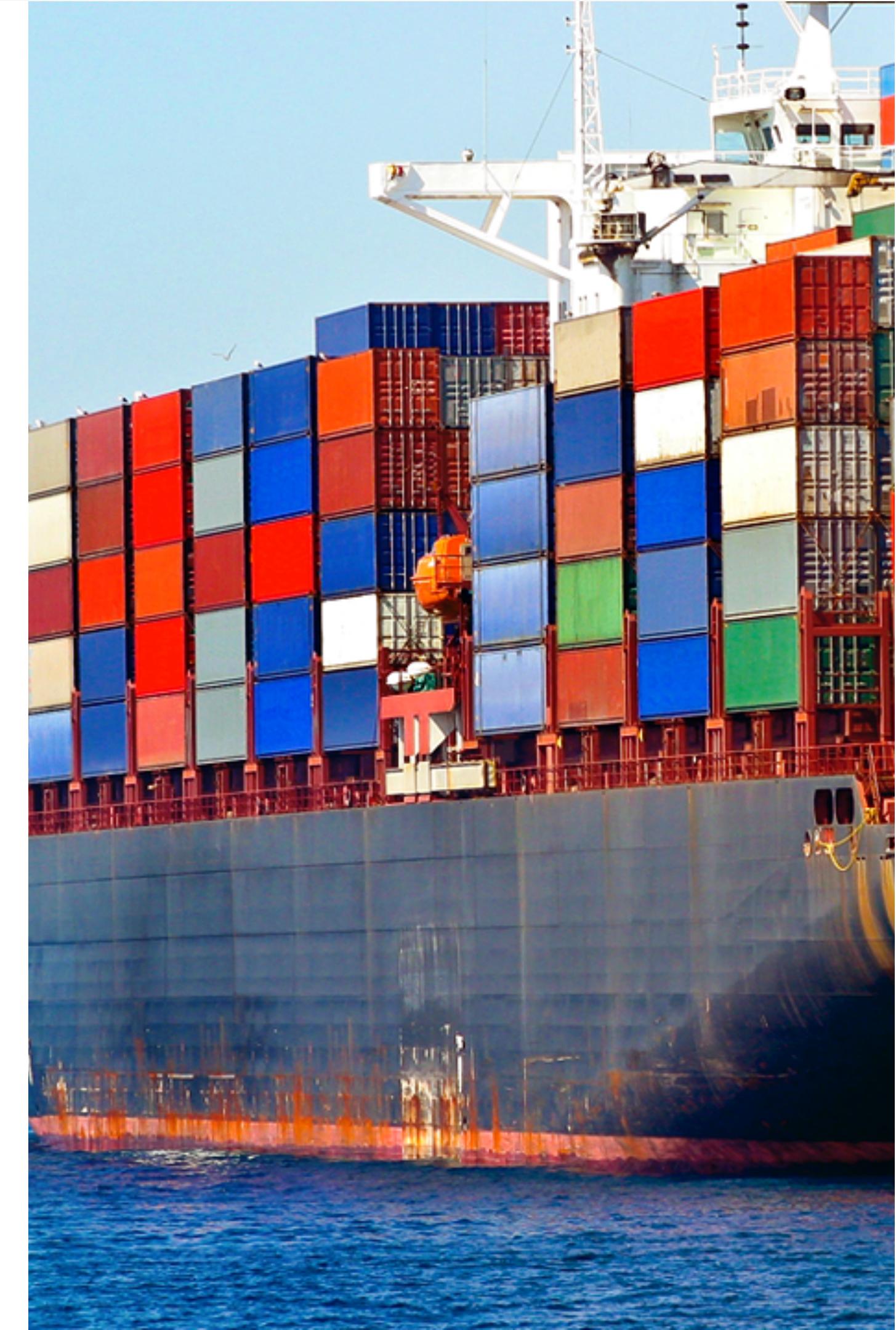
# The Lure of Containers



# The Shipping Industry's Solution to Transporting Cargo (post-1960)

How do you ship cargo of variable sizes efficiently?

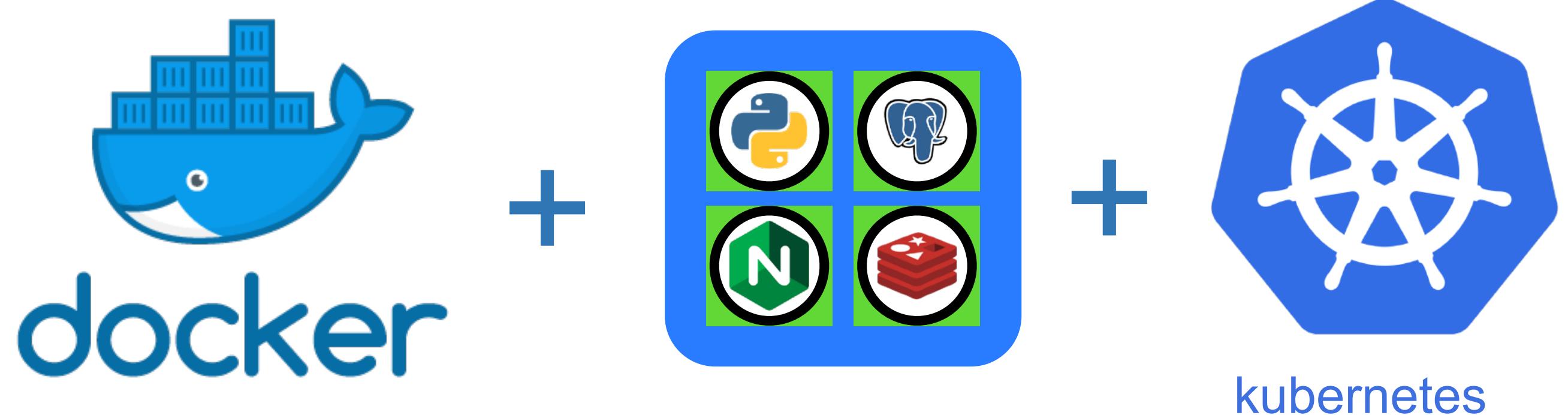
- Give the customer a standard sized container
- They can put anything they want in it, but it's their responsibility for what is inside of it
- Build transport ships that can efficiently hold these containers at scale
- Cargo size no longer matters because you only deal with the standard container



# The IT Industry's Solution to Deploying Applications

How do you deploy applications with variable dependencies?

- Give the developer a standard container
- They can put anything they want in it, but it's their responsibility for what is inside of it
- Build a programmable infrastructure that can efficiently run these containers at scale
- Application dependencies no longer matters because you only deal with the container



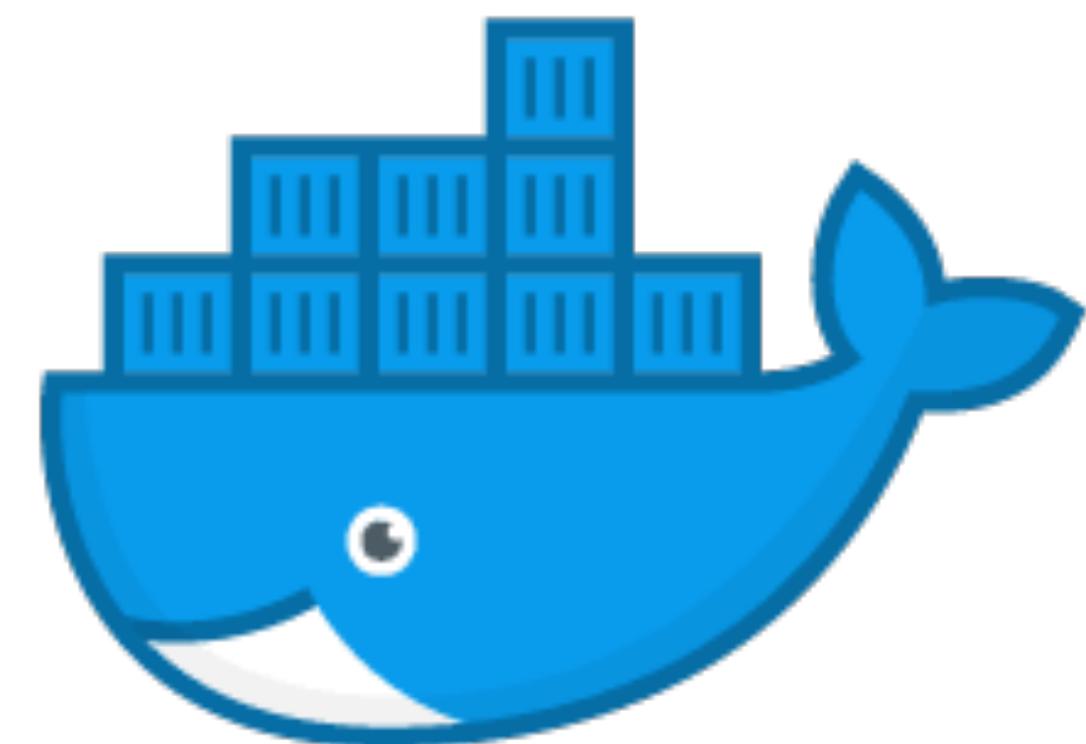
# Docker Concepts

- Docker is a platform for developers and sysadmins to develop, deploy, and run applications with containers
- The use of Linux containers to deploy applications is called *containerization*
- Containers are not new, but Docker make their use for easily deploying applications simple



# Docker Containers

- Docker is a light-weight container service that runs on Linux
  - File system overlay
  - One Process Space
  - One Network Interface
- Shares the Linux kernel with other containers and processes
  - Originally based on LXC (LinuX Containers)
- Containers encapsulate a run-time environment
  - They can contain code, libraries, package manager, data, etc.
- Almost no overhead
  - Containers spin up in seconds not minutes like VMs
  - Native performance because there is no emulation
  - Package only what you need



**docker**

# Benefits of Containers

- Great isolation
- Great manageability
- Container encapsulates implementation technology
- Efficient resource utilization
- Fast deployment



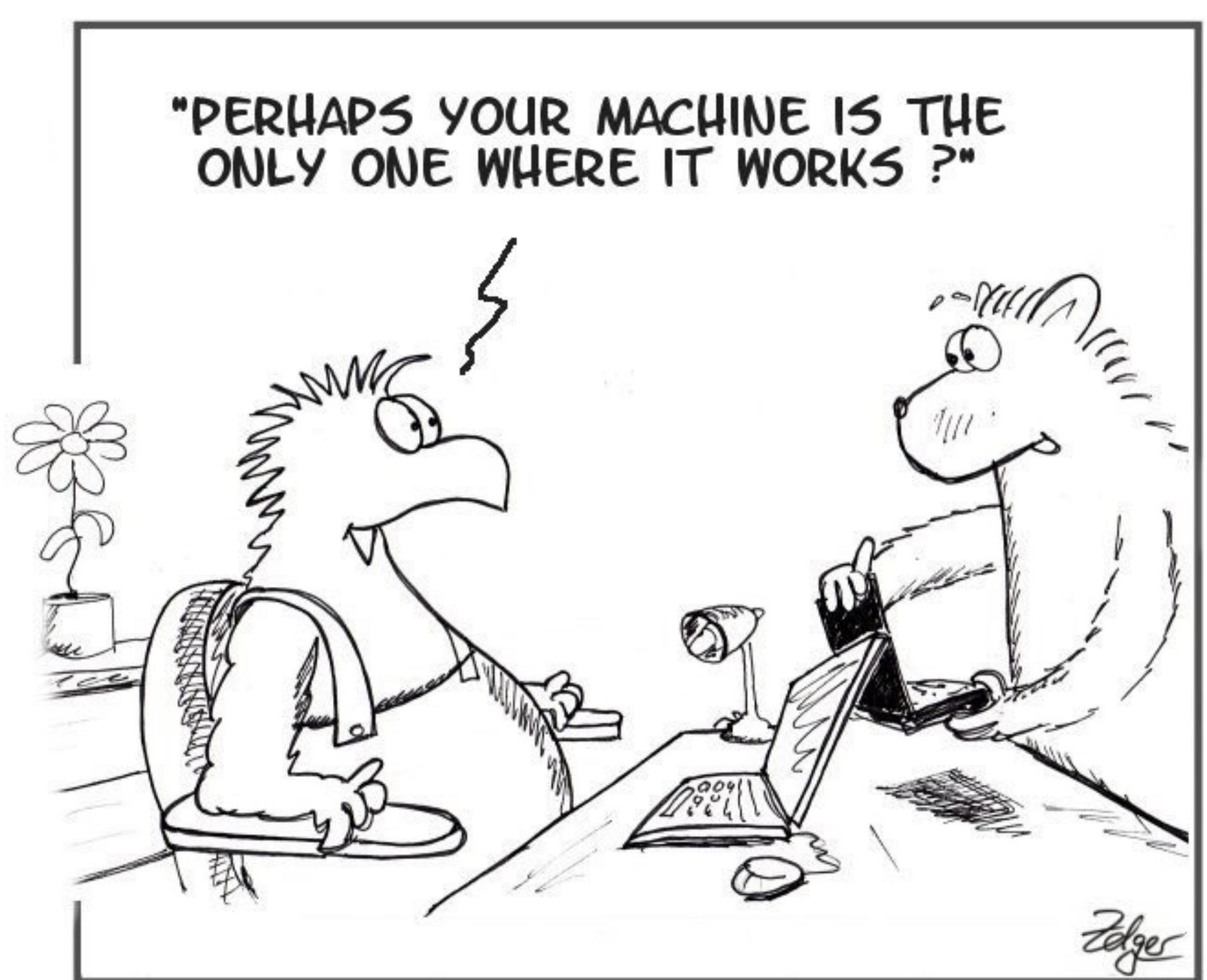
# What Can you Do with Docker?

- You can run **Containers** from the **Images** in the Docker Registry (e.g., Docker Hub)
- You can build Docker **Images** that hold your applications and their dependancies
- You can create Docker **Containers** from those Docker images to run your applications
- You can share those Docker images via **Docker Hub** or your own Docker registry
- You can pull those images from the Docker registry to **deploy** them as Containers on a server running Docker Engine
- You can even deploy those containers in the **Bluemix Container Cloud!**

# Containers Enable Immutable Delivery

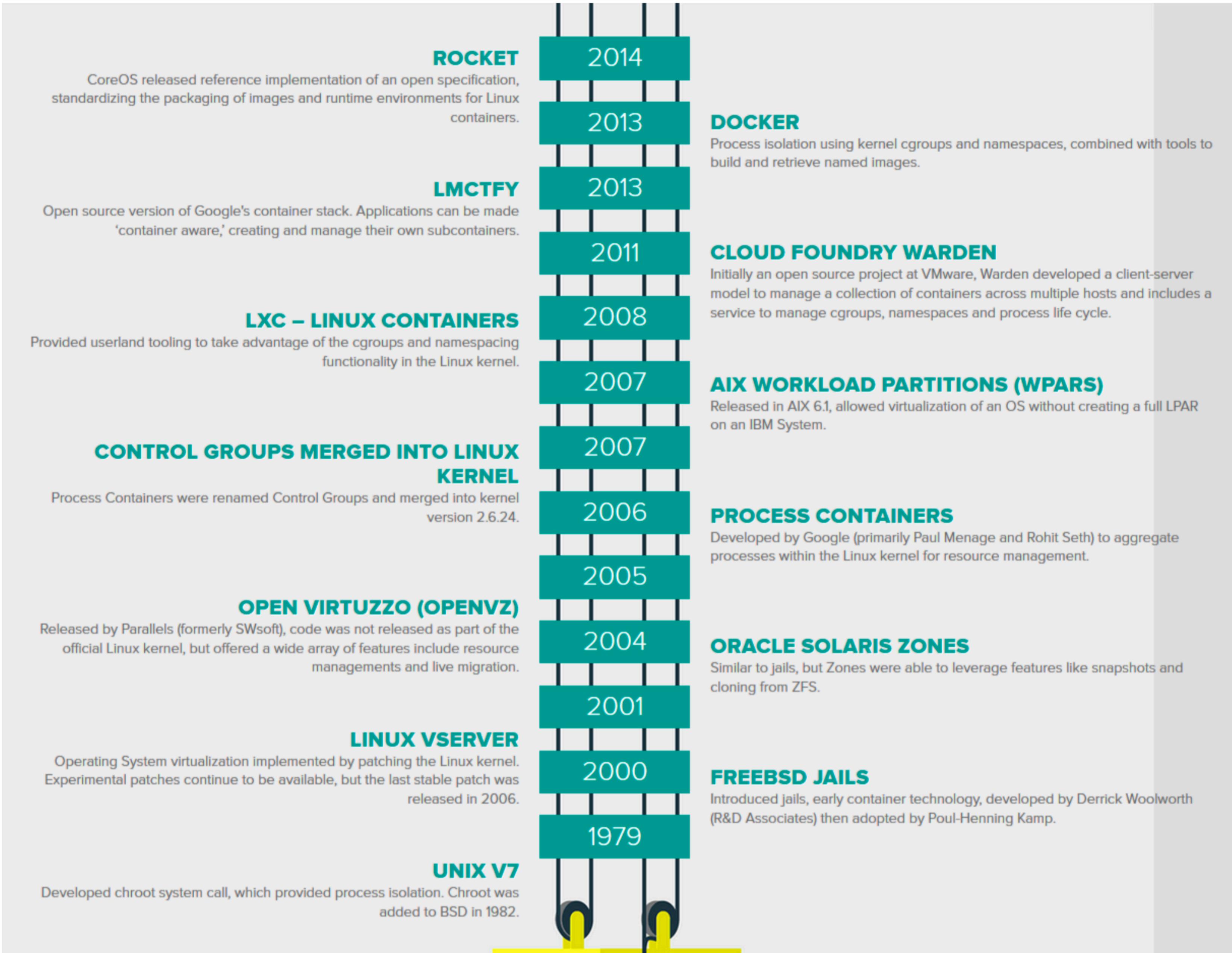
Build once... run anywhere

- The same binary that a developer runs on their laptop, runs in production
- All dependencies are package in the container
- Facilitates rolling updates with immediate roll-back
- Consistency limits side-effects

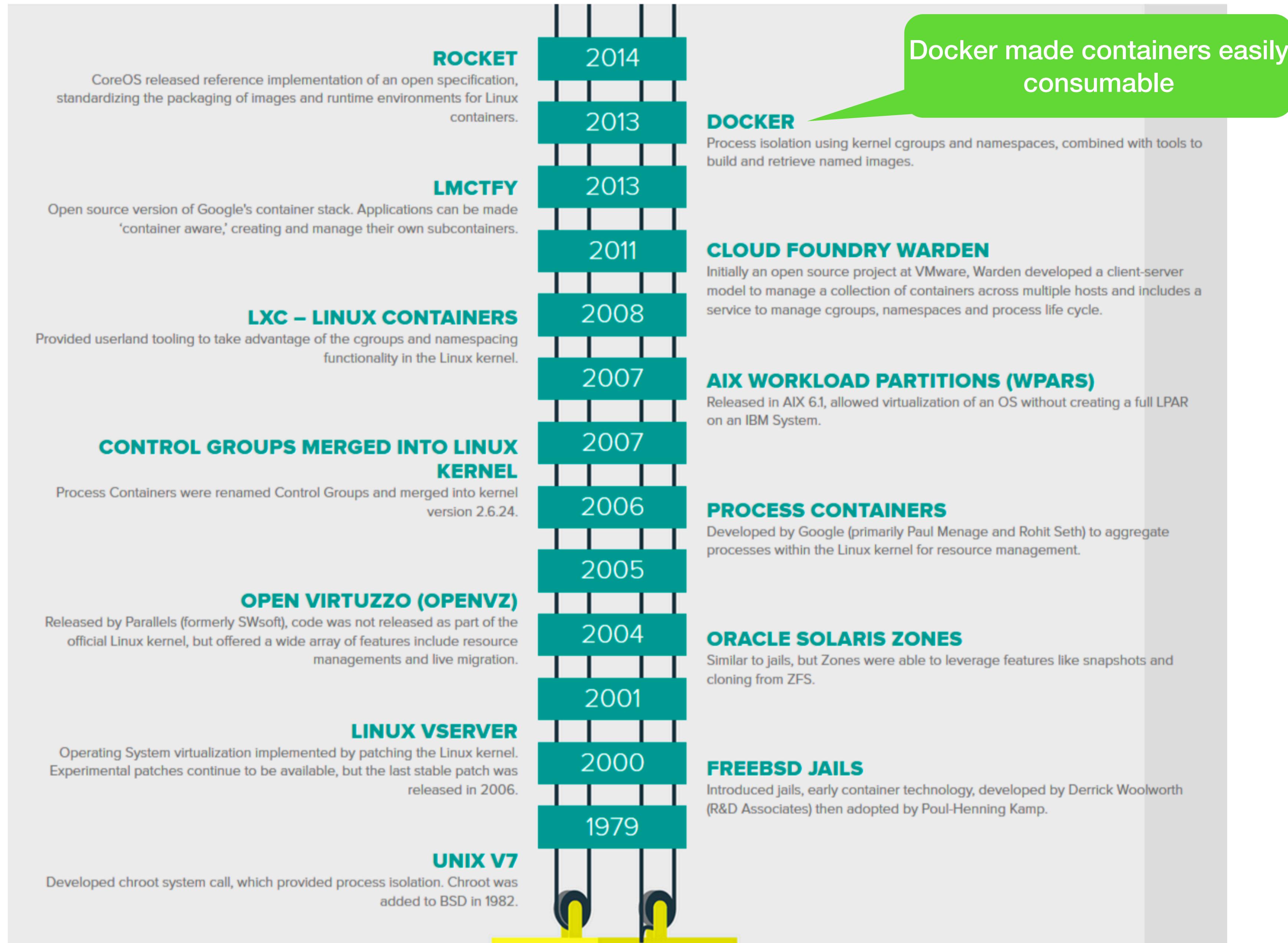


It works on my machine

# Containers are not new



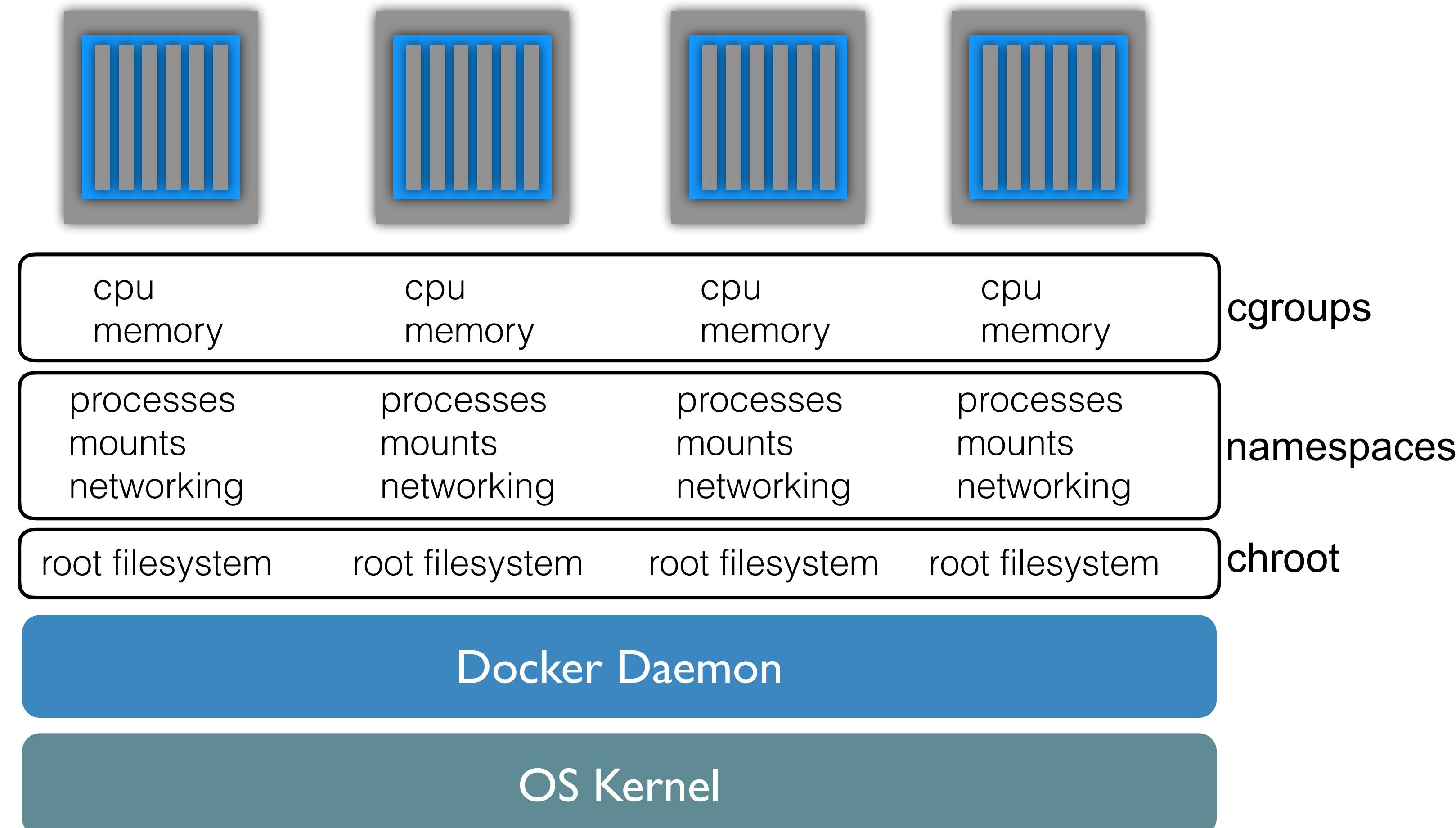
# Containers are not new



Docker made containers easily consumable

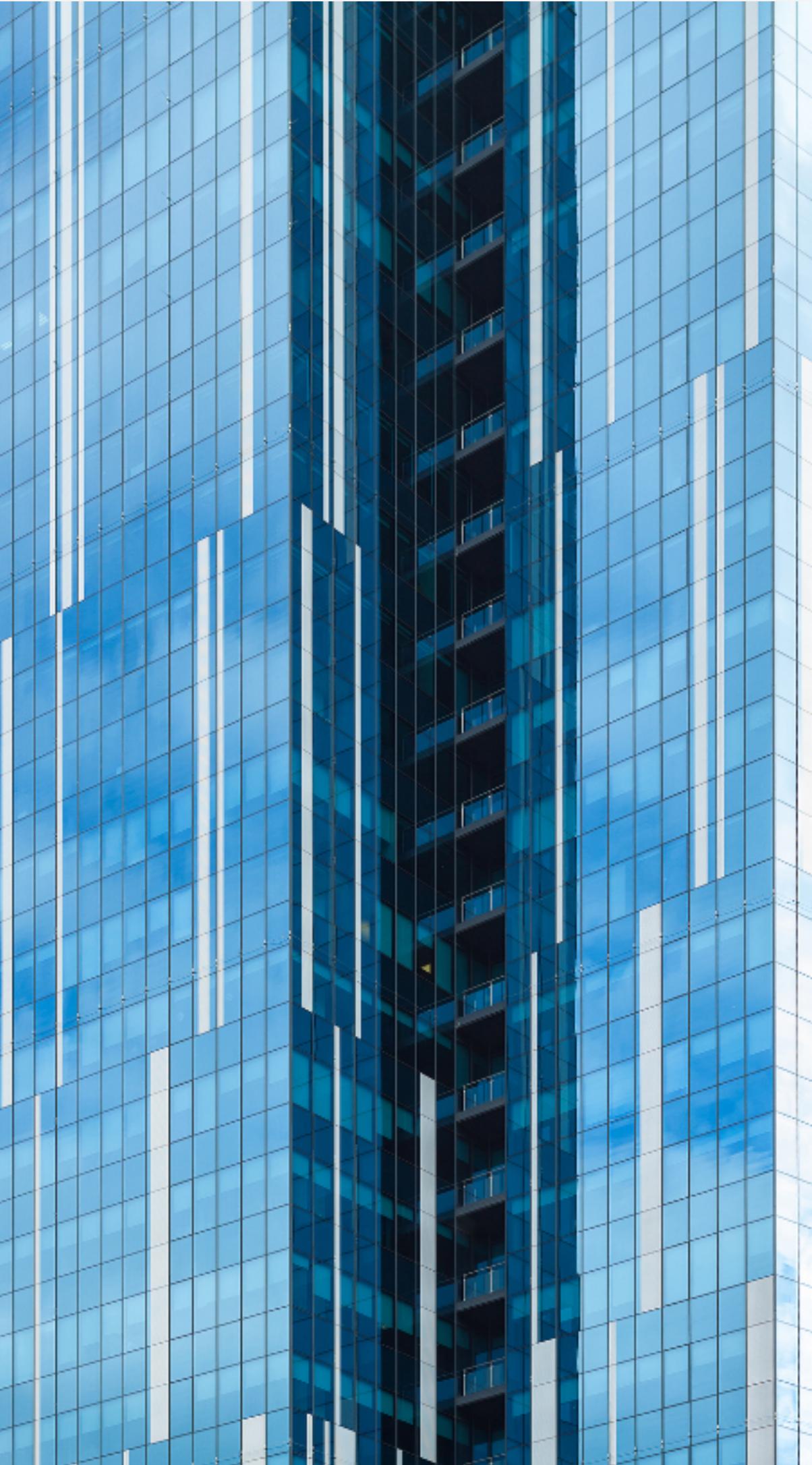
# Containers are just Linux Capabilities Under the Covers

- Containers are:
  - Linux® processes
  - with isolation and resource confinement
  - that enable you to run sandboxed applications
  - on a shared host kernel
  - using cgroups, namespaces, and chroot



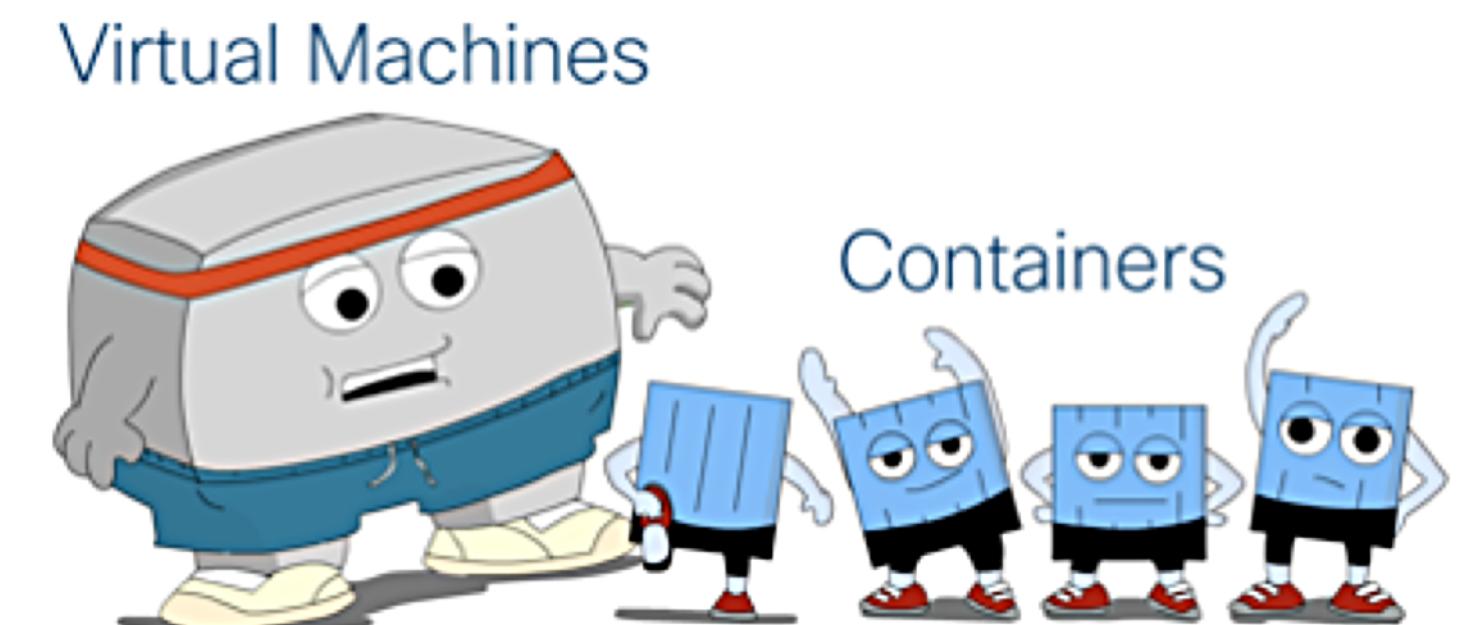
# Put another way

- Docker Containers allow you to control
  - What resources a process can **see**
  - What resources a process can **control**
  - What filesystem a process **uses**



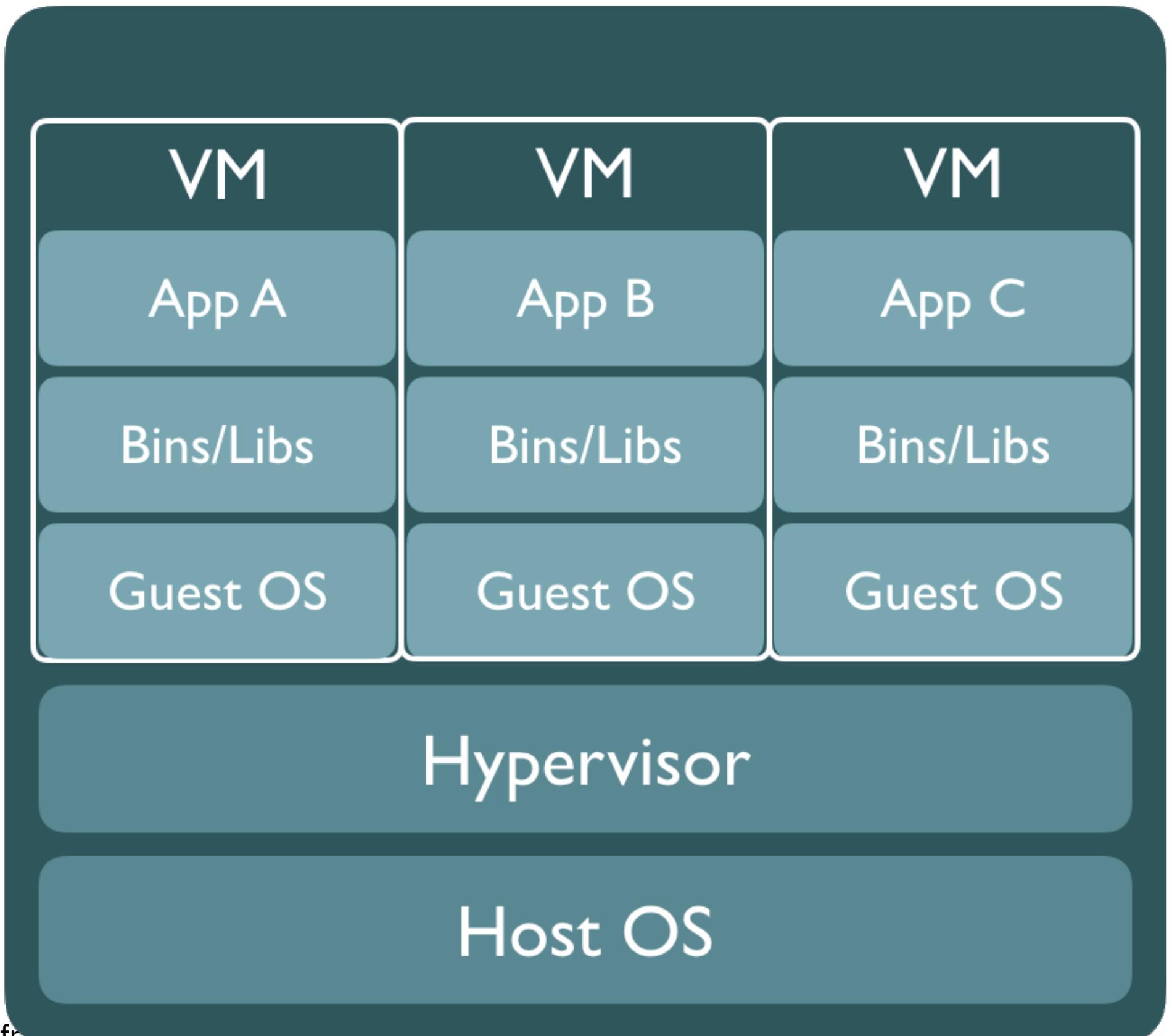
# Containers and Virtual Machines

- A container runs natively on Linux and shares the kernel of the host machine with other containers
  - It runs a discrete process, taking no more memory than any other executable, making it lightweight
- By contrast, a virtual machine (VM) runs a full-blown “guest” operating system with virtual access to host resources through a hypervisor
  - In general, VMs provide an environment with more resources than most applications need



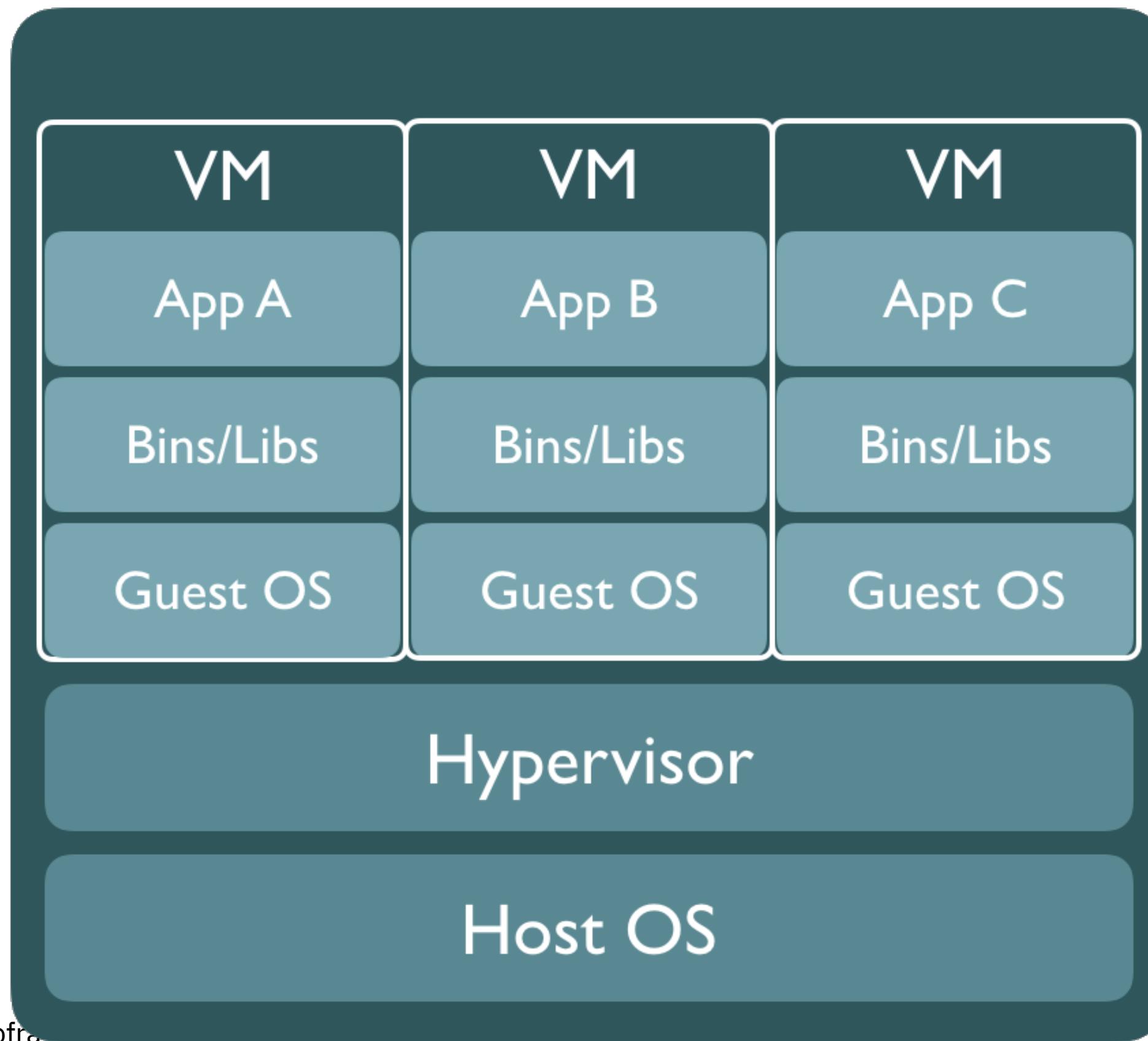
# Containers vs Virtual Machines

Virtual Machines are heavy-weight  
emulations of real hardware

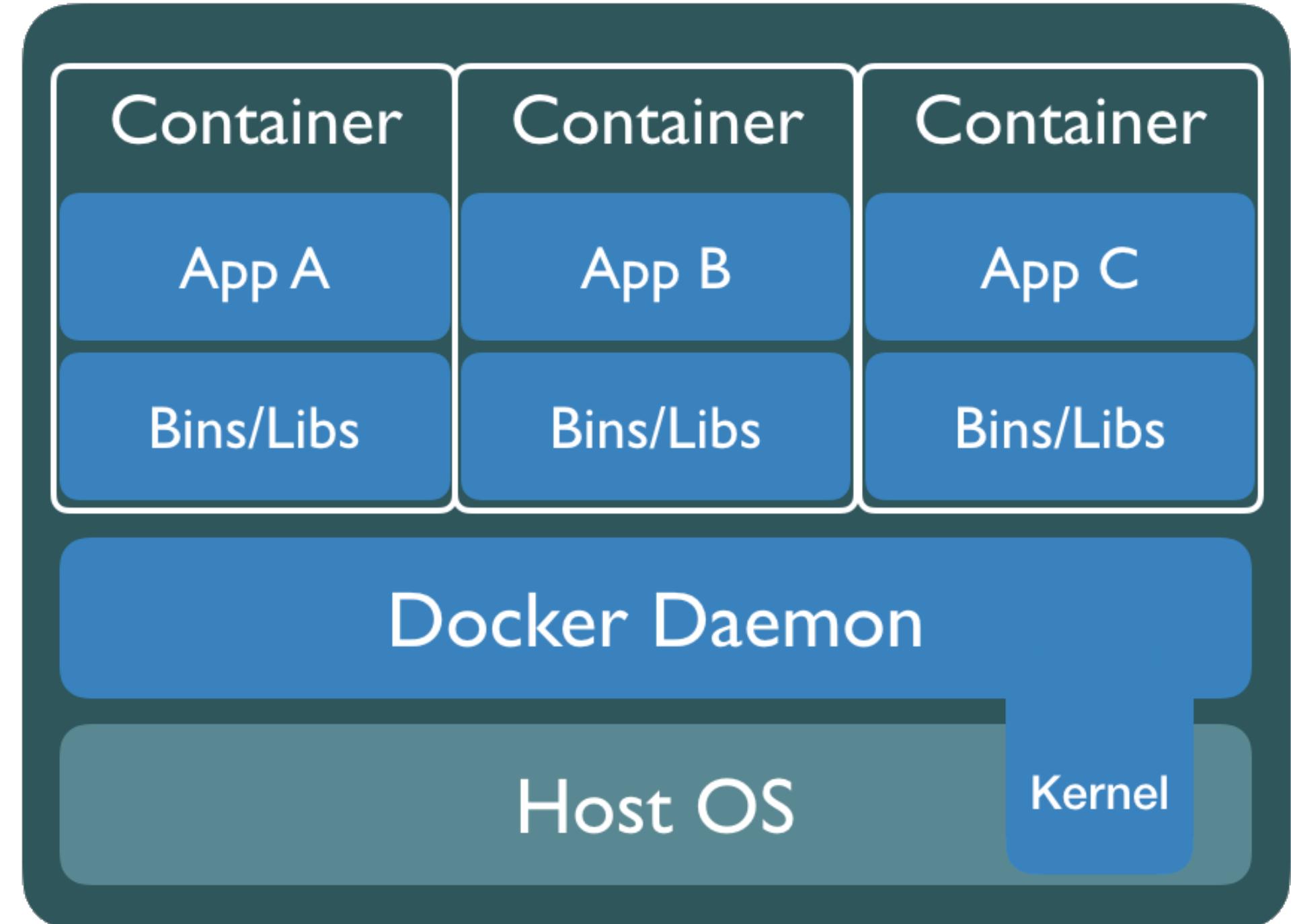


# Containers vs Virtual Machines

Virtual Machines are heavy-weight emulations of real hardware



Containers are light-weight process  
The app looks like it's running on the Host OS

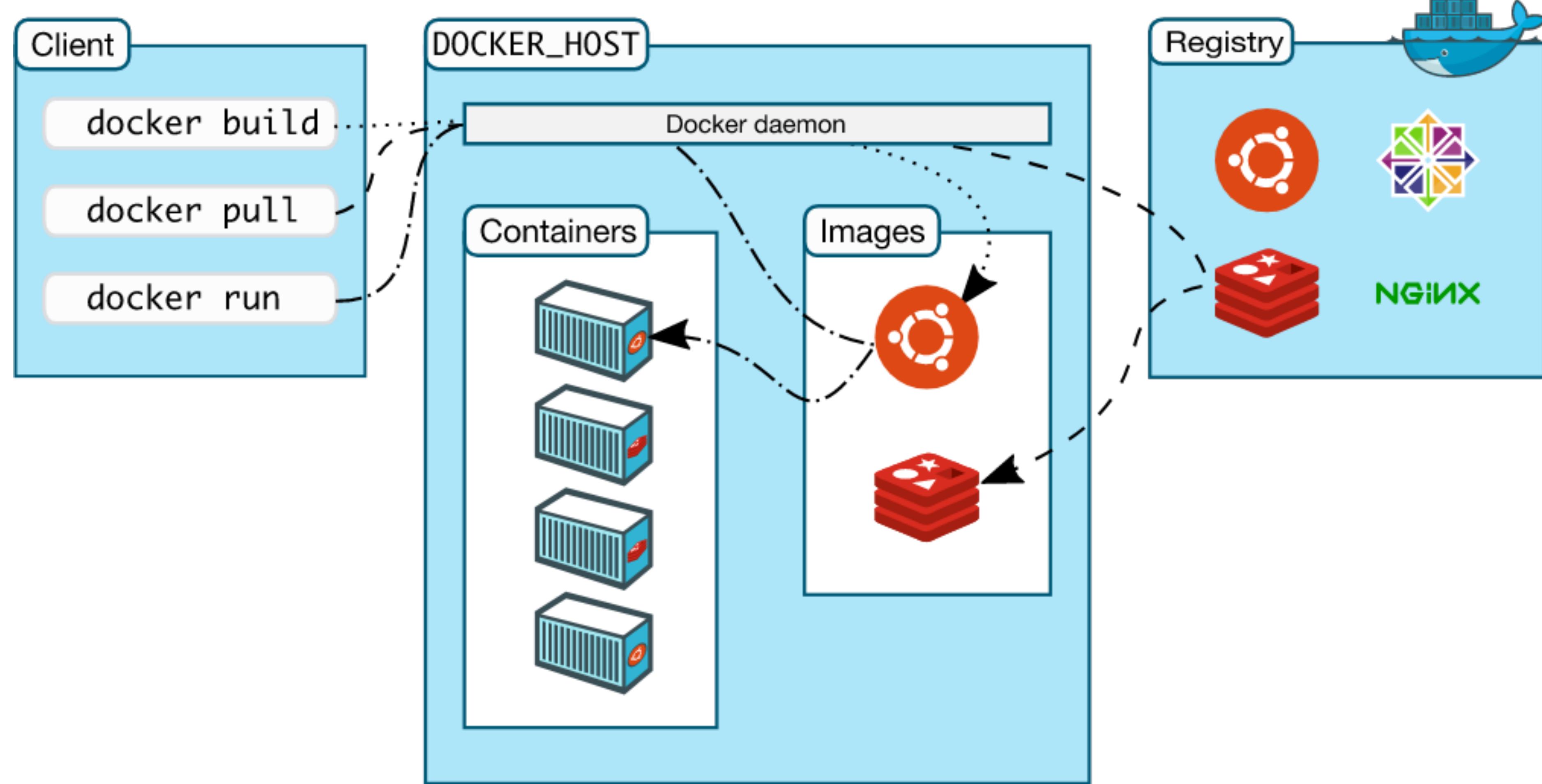


# Containers in Practice



# Where do Containers Come From?

## Docker @ 20,000 feet



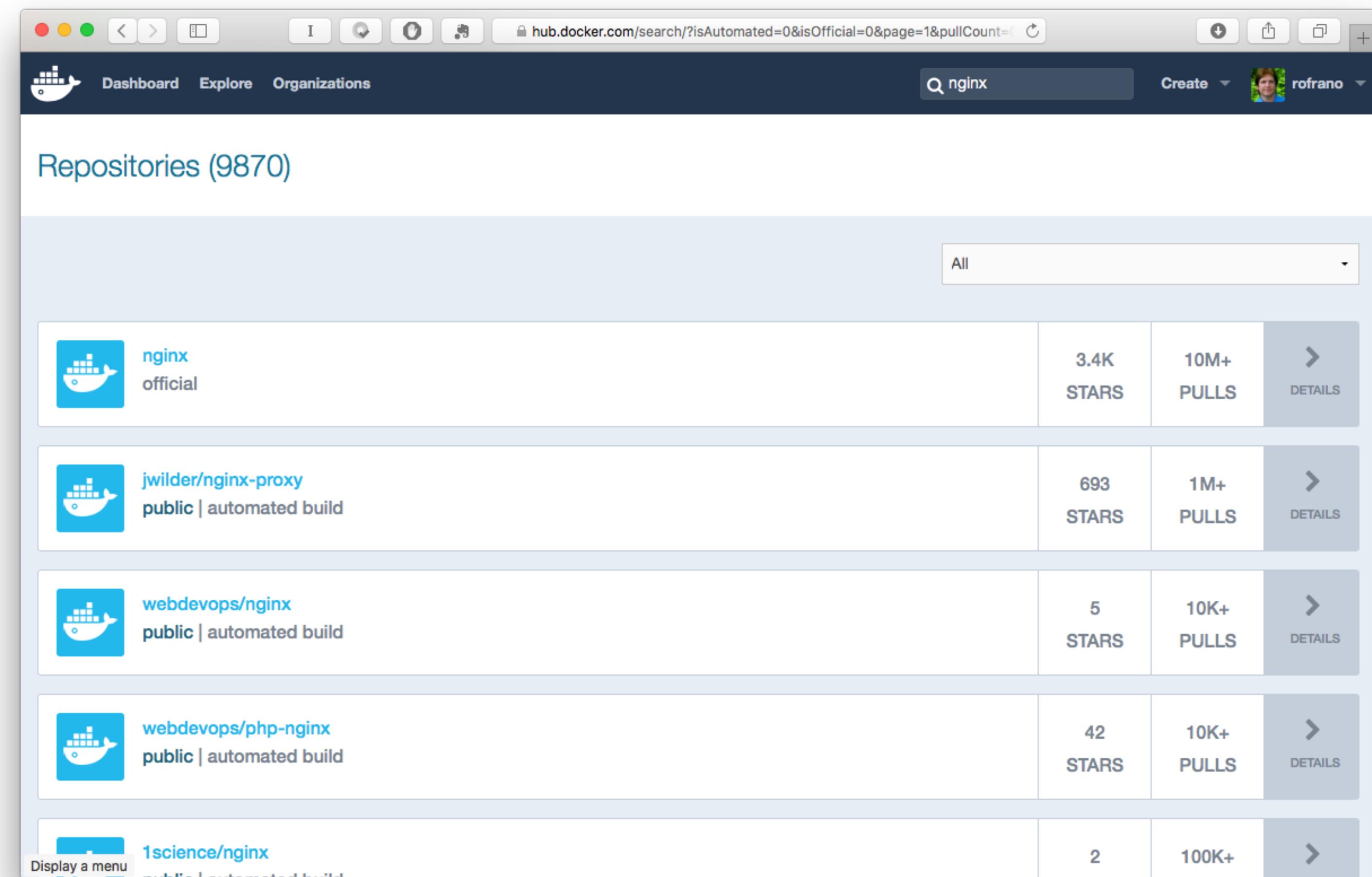
# Images and containers

- A container is launched by running an image
- An image is an executable package that includes everything needed to run an application--the code, a runtime, libraries, environment variables, and configuration files
- A container is a *runtime instance* of an image i.e., what the image becomes in memory when executed (that is, an image with state, or a user process)
- You can see a list of your running containers with the command, docker ps, just as you would in Linux

# Docker Hub Hosts Official and Community Images

<http://hub.docker.com>

- Contains 1000's of Docker images with almost every software you can imagine
- Usually “official” images come from the creator of the software
- Most have documentation on how to best use them



# Images Include Documentation

## Documentation

- Each image has documentation on how to use it
- From simply running the container
- To forwarding ports, mapping storage, etc.



### How to use this image 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 the 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.

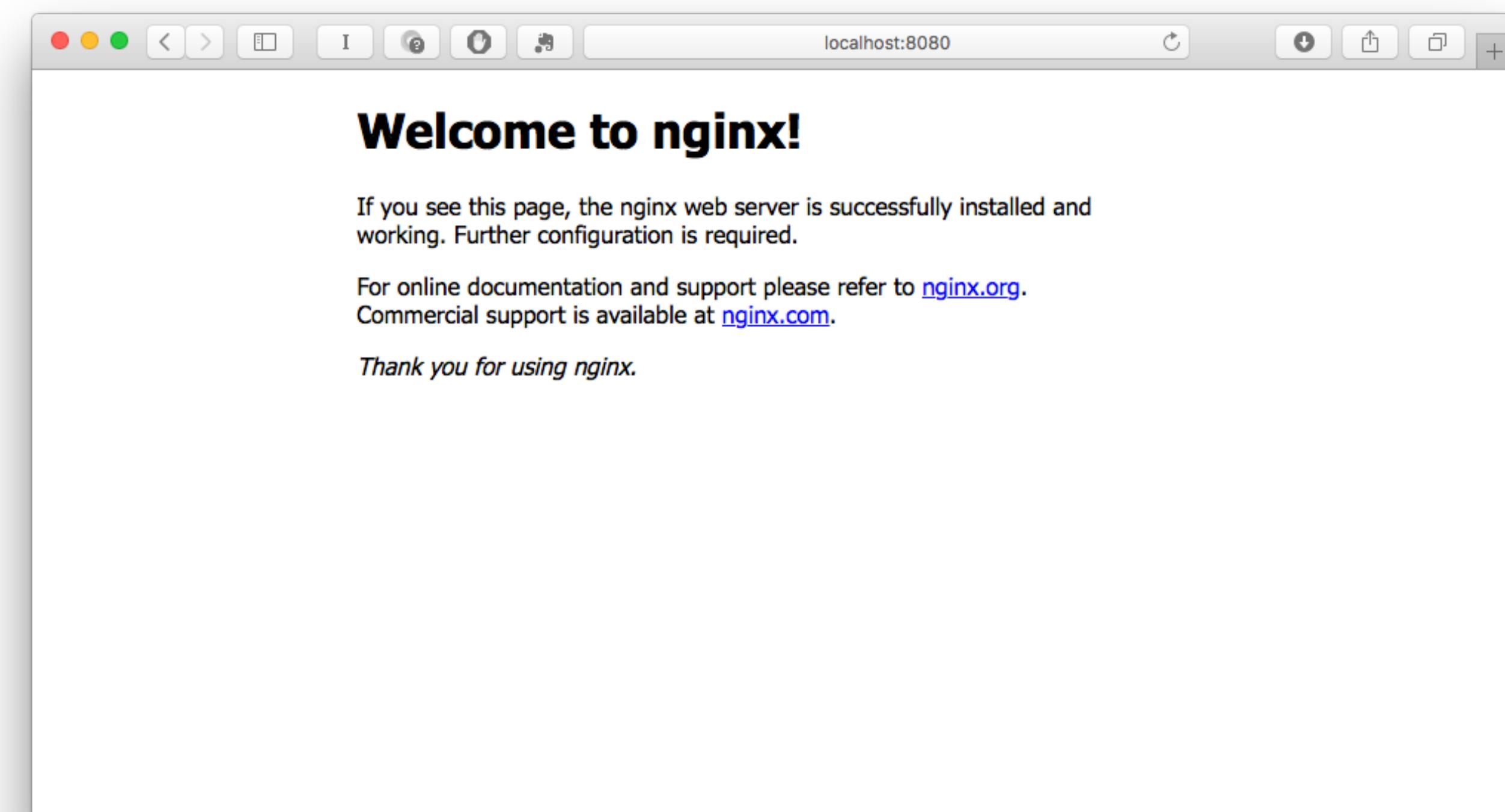
# Nginx installed and running with one command

We can run nginx as a web server with the command:

```
$ docker run -d -p 8080:80 nginx:alpine
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
6a5a5368e0c2: Pull complete
20a0fbbae148: Pull complete
2fbcd37c8684b: Pull complete
Digest:
sha256:e40499ca855c9edfb212e1c3ee1a6ba8b2d873a294d897b4840d49f94d20487c
Status: Downloaded newer image for nginx:latest
0d48962ddc380c421851fb808b9b3007c0c9c614bd08ae7e732955ddaa4c7b4a
```

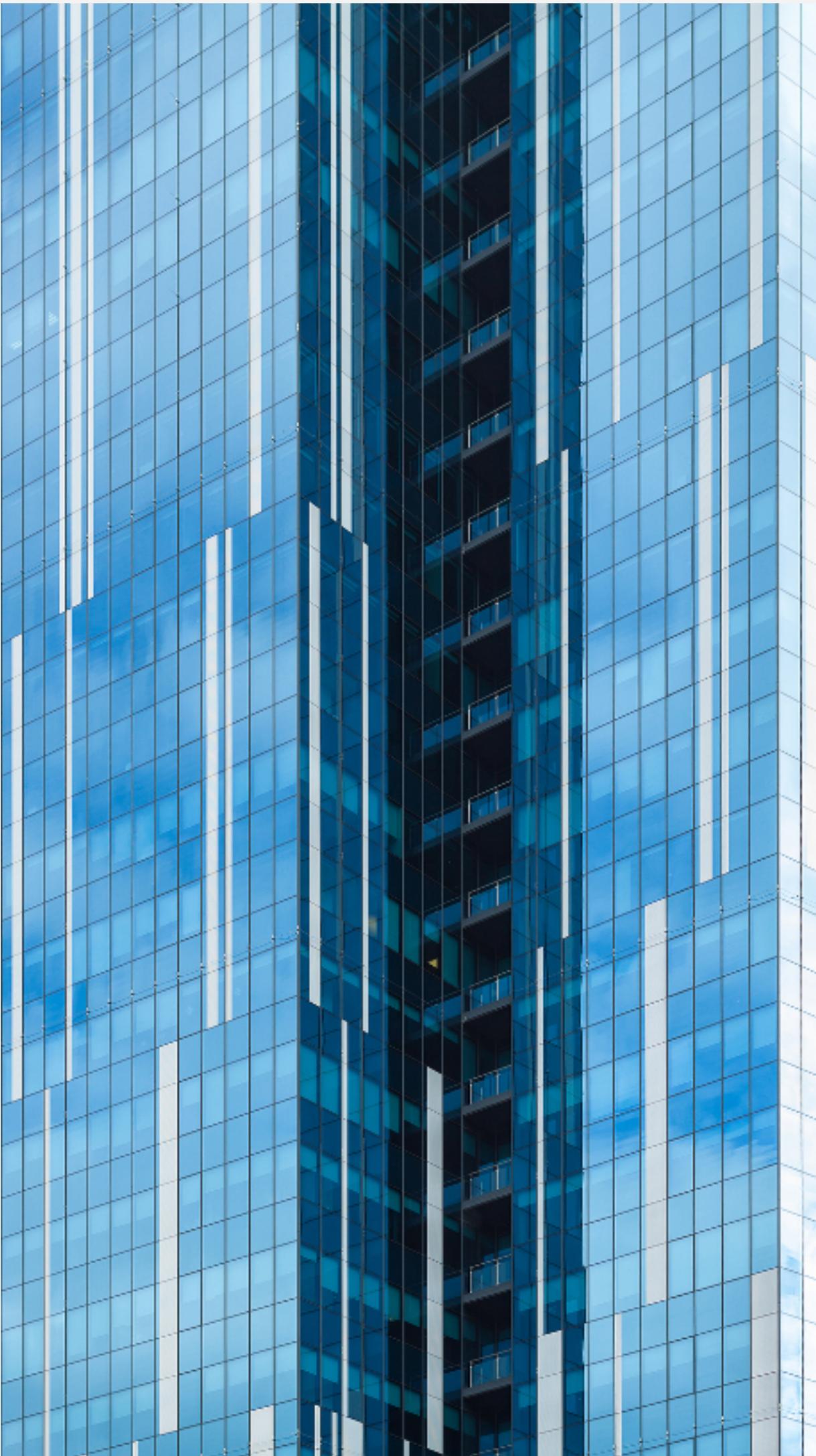
Since we don't have a local nginx image it will be pulled from Docker hub the first time

Nginx running in a container



# Docker Provides Portable Isolated Environments

- Docker gives you portable isolated environments
- You could have an application that requires Python 3.6 running in one container
- With an application that requires Python 2.7 running in another
- Completely isolated from each other will no chance of library collisions



# Wordpress and MySQL Traditional Installation



## Famous 5-Minute Installation

Here's the quick version of the instructions for those who are already comfortable with performing such installations. More [detailed instructions](#) follow.

If you are not comfortable with renaming files, step 3 is optional and you can skip it as the install program will create the `wp-config.php` file for you.

1. Download and unzip the WordPress package if you haven't already.
2. Create a database for WordPress on your web server, as well as a [MySQL](#) (or MariaDB) user who has all privileges for accessing and modifying it.
3. (Optional) Find and rename `wp-config-sample.php` to `wp-config.php`, then edit the file (see [Editing wp-config.php](#)) and add your database information.
4. Upload the WordPress files to the desired location on your web server:
  - If you want to integrate WordPress into the root of your domain (e.g. `http://example.com/`), move or upload all contents of the unzipped WordPress directory (excluding the WordPress directory itself) into the root directory of your web server.
  - If you want to have your WordPress installation in its own subdirectory on your website (e.g. `http://example.com/blog/`), create the `blog` directory on your server and upload the contents of the unzipped WordPress package to the directory via FTP.
  - **Note:** If your FTP client has an option to convert file names to lower case, make sure it's disabled.
5. Run the WordPress installation script by accessing the URL in a web browser. This should be the URL where you uploaded the WordPress files.
  - If you installed WordPress in the root directory, you should visit: `http://example.com/`
  - If you installed WordPress in its own subdirectory called `blog`, for example, you should visit: `http://example.com/blog/`

That's it! WordPress should now be installed.

# Wordpress and MySQL Traditional Installation



## Famous 5-Minute Installation

Here's the quick version of the instructions for those who are already comfortable with performing such installations. More [detailed instructions](#) follow.

If you are not comfortable with renaming files, step 3 is optional and you can skip it as the install program will create the `wp-config.php` file for you.

1. Download and unzip the WordPress package if you haven't already.
  2. Create a database for WordPress on your web server, as well as a [MySQL](#) (or MariaDB) user who has all privileges for accessing and modifying it.
  3. (Optional) Find and rename `wp-config-sample.php` to `wp-config.php`, then edit the file (see [Editing wp-config.php](#)) and add your database information.
  4. Upload the WordPress files to the desired location on your web server:
    - If you want to integrate WordPress into the root of your domain (e.g. `http://example.com/`), move or upload all contents of the unzipped WordPress directory (excluding the WordPress directory itself) into the root directory of your web server.
    - If you want to have your WordPress installation in its own subdirectory on your website (e.g. `http://example.com/blog/`), create the `blog` directory on your server and upload the contents of the unzipped WordPress package to the directory via FTP.
    - **Note:** If your FTP client has an option to convert file names to lower case, make sure it's disabled.
  5. Run the WordPress installation script by accessing the URL in a web browser. This should be the URL where you uploaded the WordPress files.
    - If you installed WordPress in the root directory, you should visit: `http://example.com/`
    - If you installed WordPress in its own subdirectory called `blog`, for example, you should visit: `http://example.com/blog/`
- That's it! WordPress should now be installed.

## Steps for a Fresh Installation of MySQL

### 1. Adding the MySQL APT Repository

First, add the MySQL APT repository to your system's software repository list. Follow these steps:

- Go to the download page for the MySQL APT repository at <https://dev.mysql.com/downloads/repo/apt/>.
- Select and download the release package for your Linux distribution.
- Install the downloaded release package with the following command, replacing `version-specific-package-name` with the name of the downloaded package (preceded by its path, if you are not running the command inside the folder where the package is):

```
shell> sudo dpkg -i /PATH/version-specific-package-name.deb
```

For example, for version `w.x.y-z` of the package, the command is:

```
shell> sudo dpkg -i mysql-apt-config_w.x.y-z_all.deb
```

Note that the same package works on all supported Debian and Ubuntu platforms.

- During the installation of the package, you will be asked to choose the versions of the MySQL server and other components (for example, the MySQL Workbench) that you want to install. If you are not sure which version to choose, do not change the default options selected for you. You can also choose `none` if you do not want a particular component to be installed. After making the choices for all components, choose `Ok` to finish the configuration and installation of the release package.
- You can always change your choices for the versions later; see [Selecting a Major Release Version](#) for instructions.
- Update package information from the MySQL APT repository with the following command (*this step is mandatory*):

```
shell> sudo apt-get update
```

### 2. Installing MySQL with APT

Install MySQL by the following command:

```
shell> sudo apt-get install mysql-server
```

This installs the package for the MySQL server, as well as the packages for the client and for the database common files. During the installation, you are asked to supply a password for the root user for your MySQL installation.

### 3. Starting and Stopping the MySQL Server

The MySQL server is started automatically after installation. You can check the status of the MySQL server with the following command:

```
shell> sudo service mysql status
```

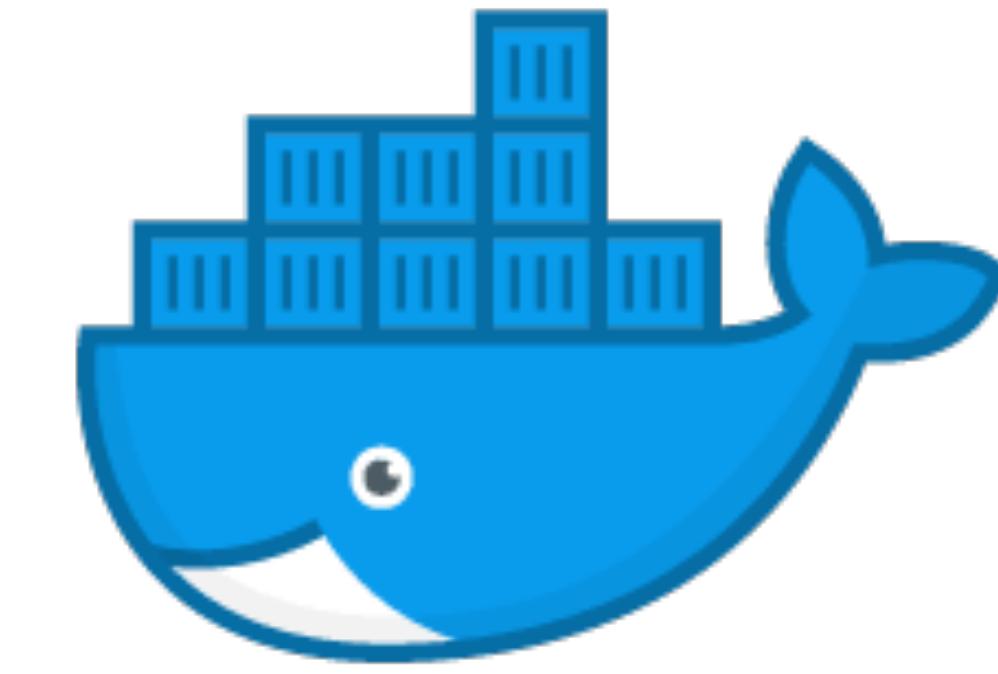
Stop the MySQL server with the following command:

```
shell> sudo service mysql stop
```

To restart the MySQL server, use the following command:

```
shell> sudo service mysql start
```

# Wordpress and MySQL with Docker

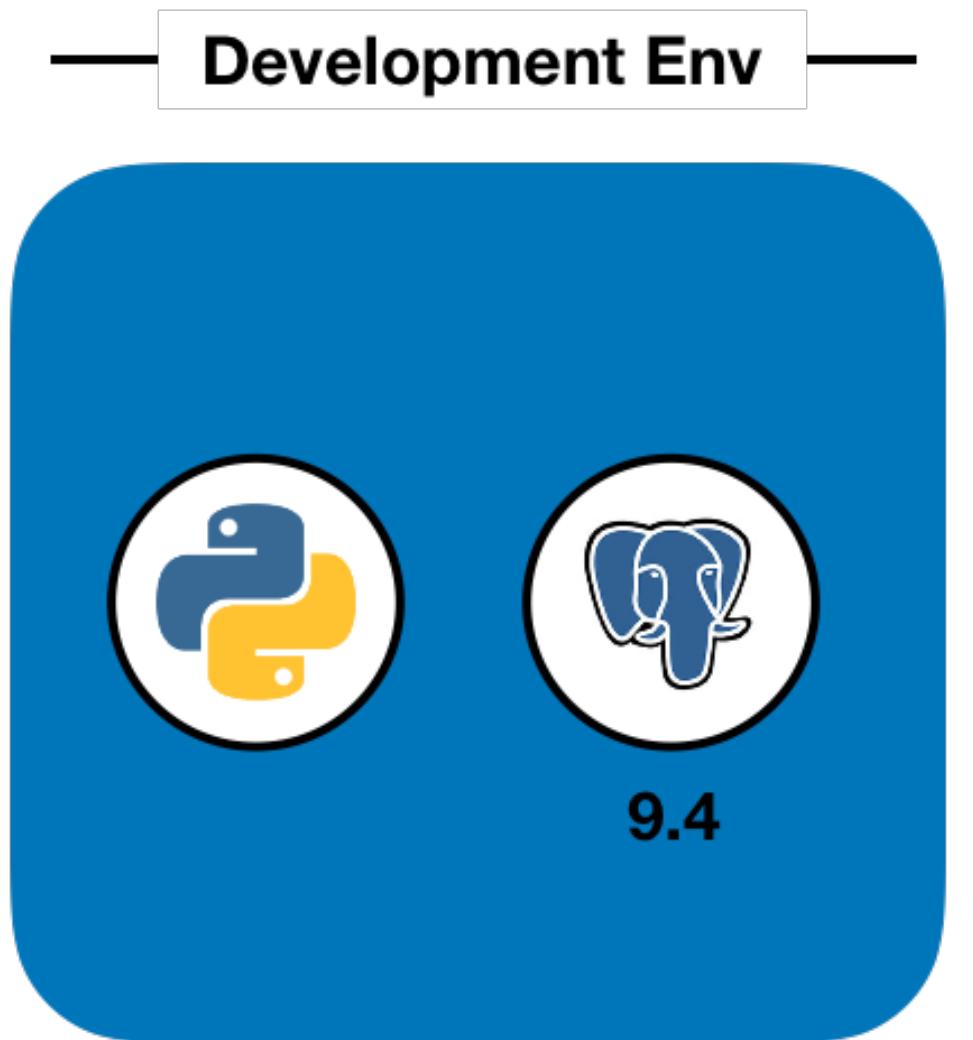


# docker

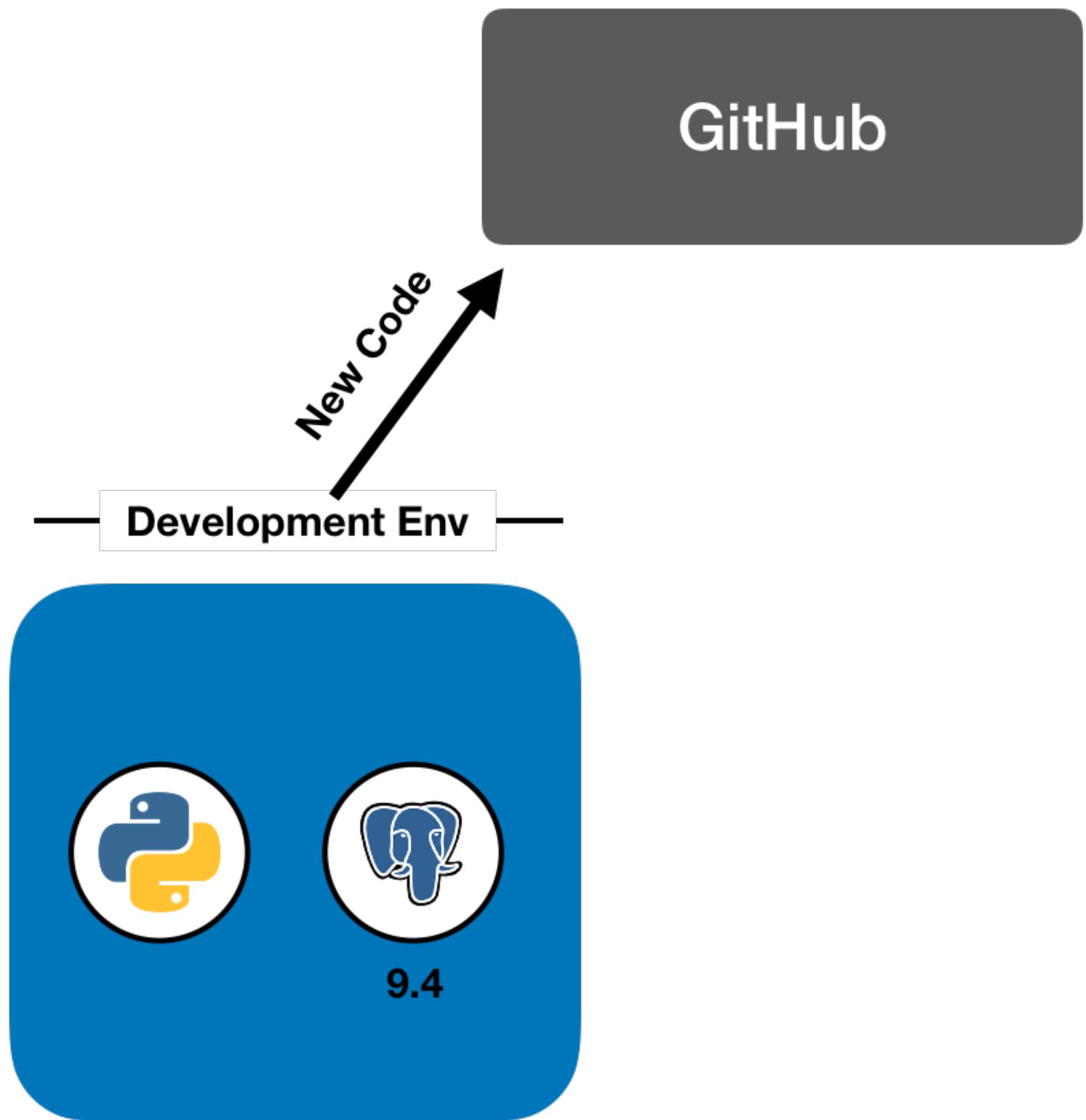
```
docker run -d --name db mysql
```

```
docker run -d --link db -p 80:80 wordpress
```

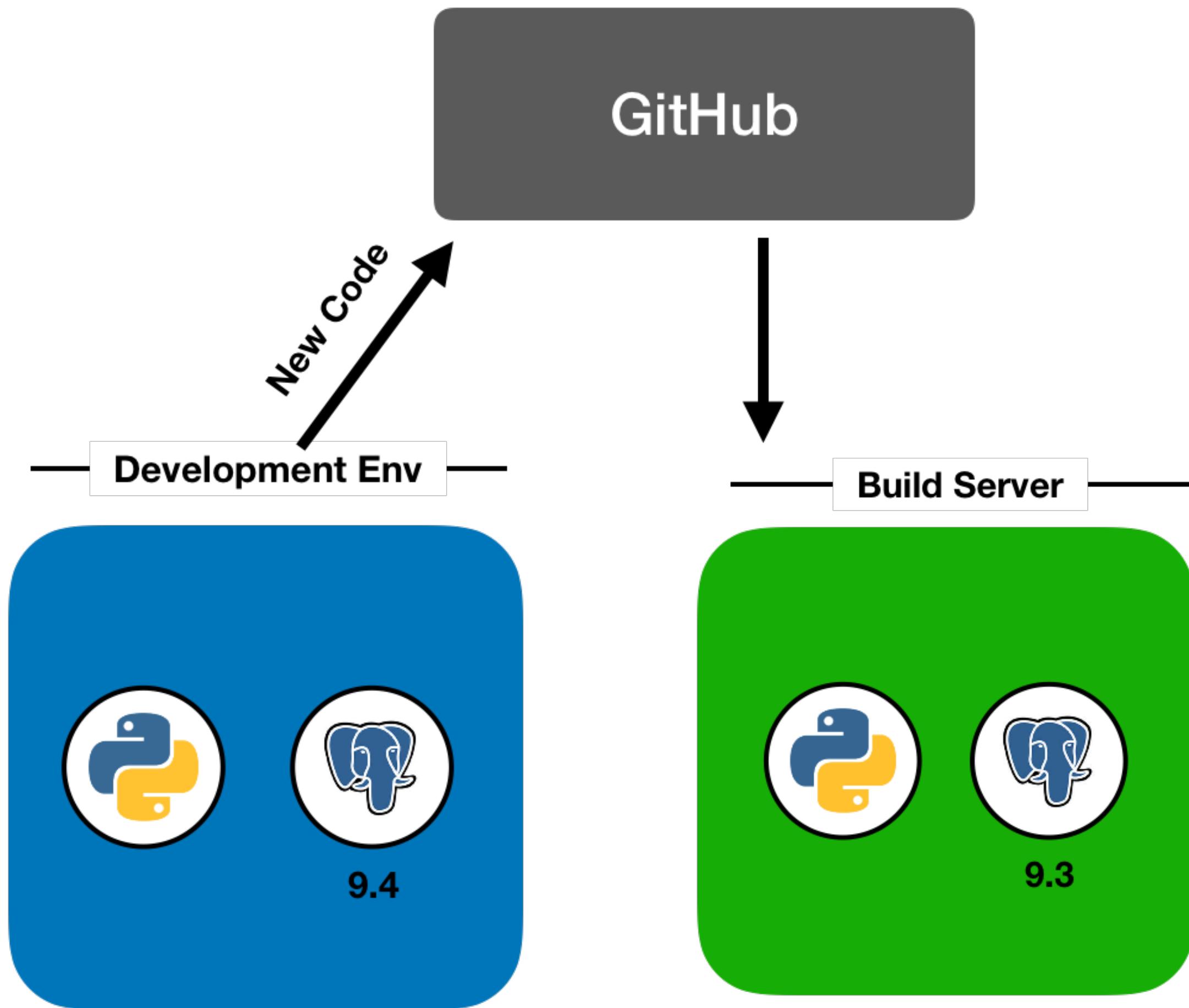
# Traditional Development Workflow



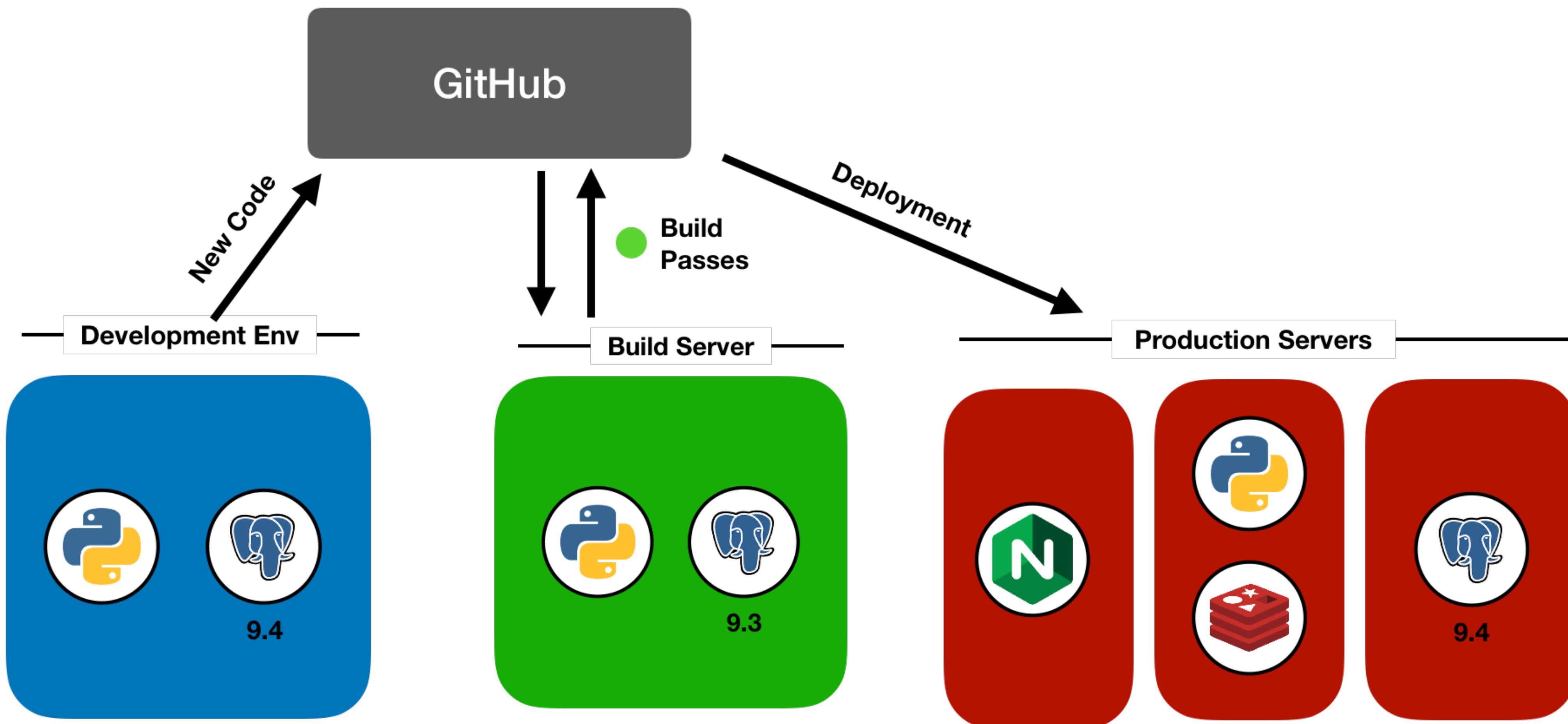
# Traditional Development Workflow



# Traditional Development Workflow



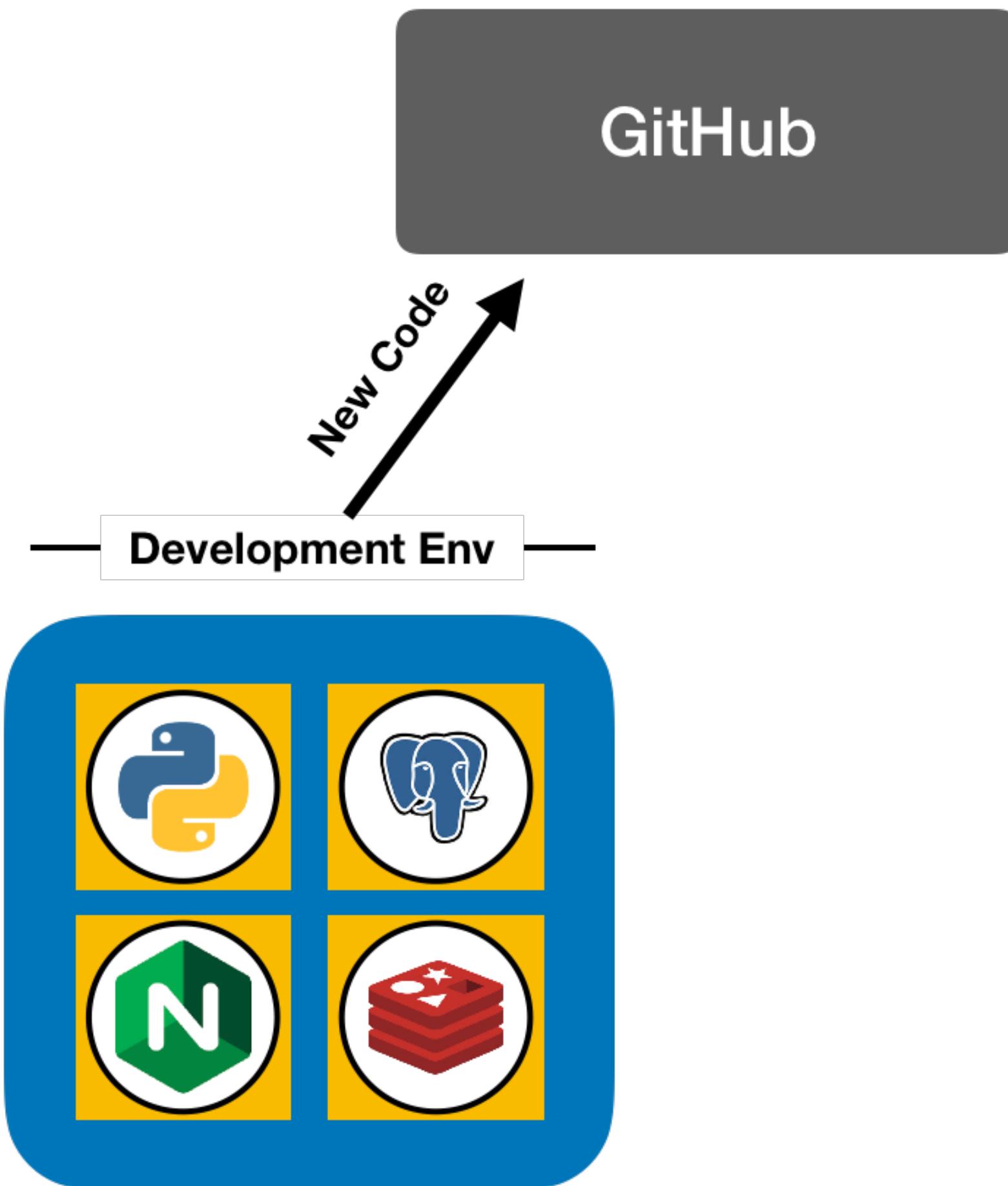
# Traditional Development Workflow



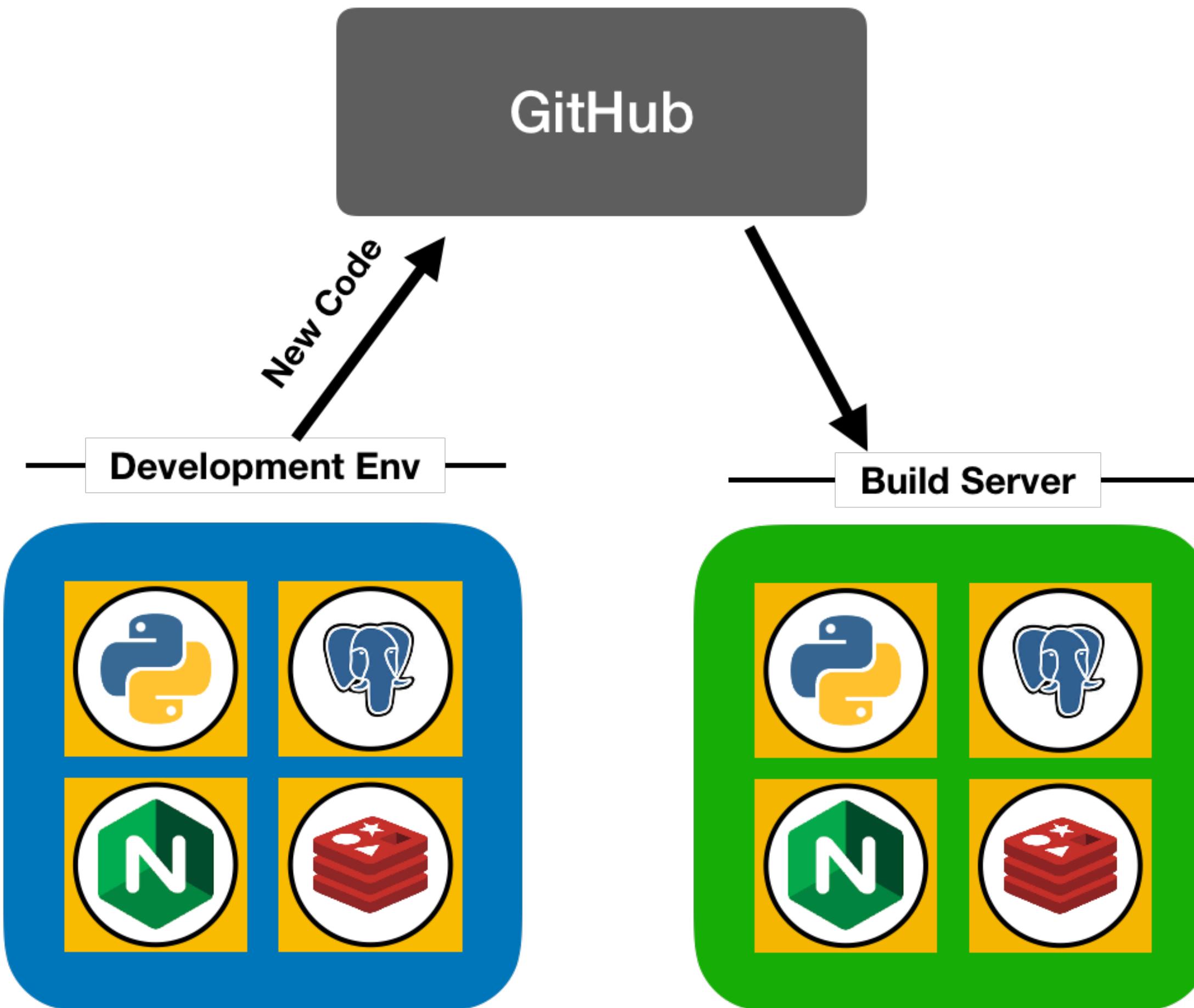
# Containerized Development Workflow



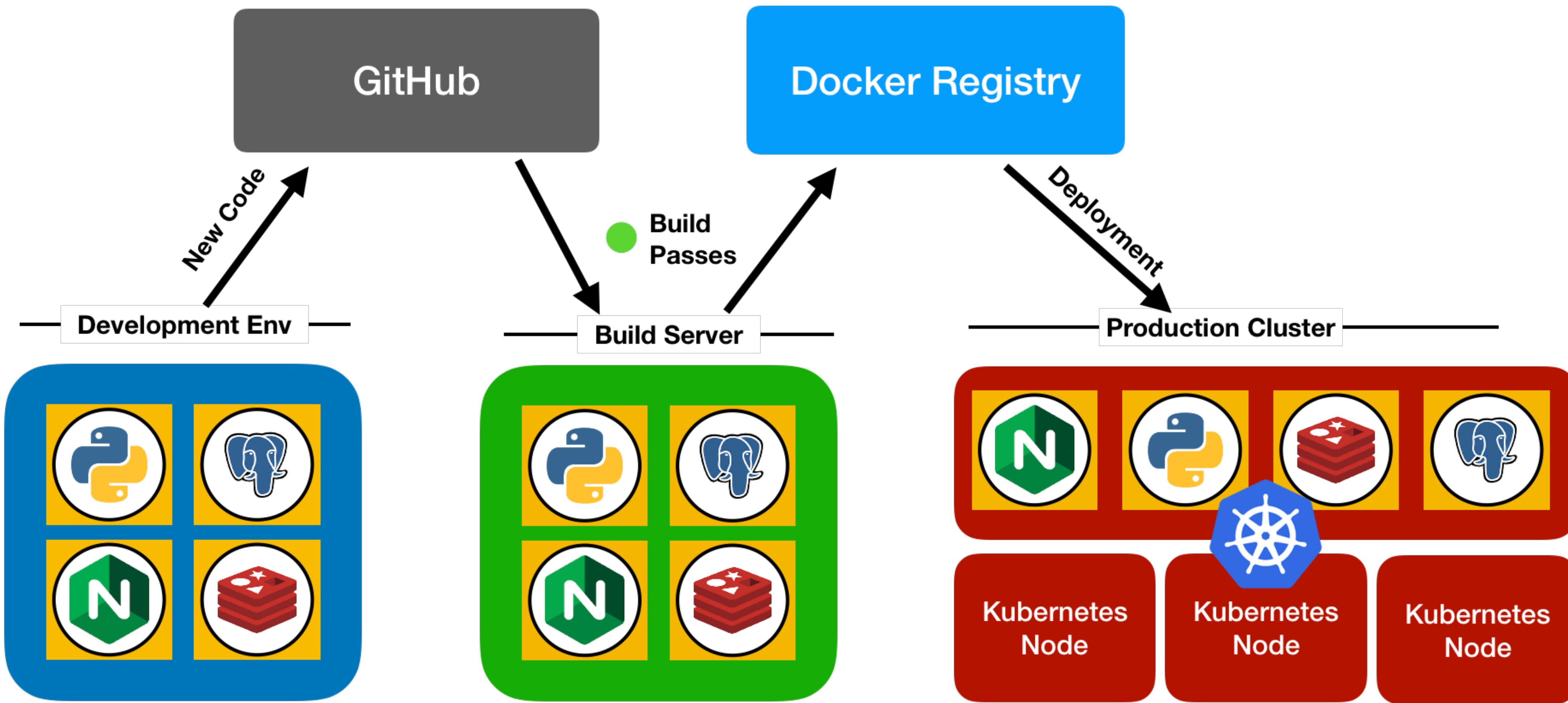
# Containerized Development Workflow



# Containerized Development Workflow

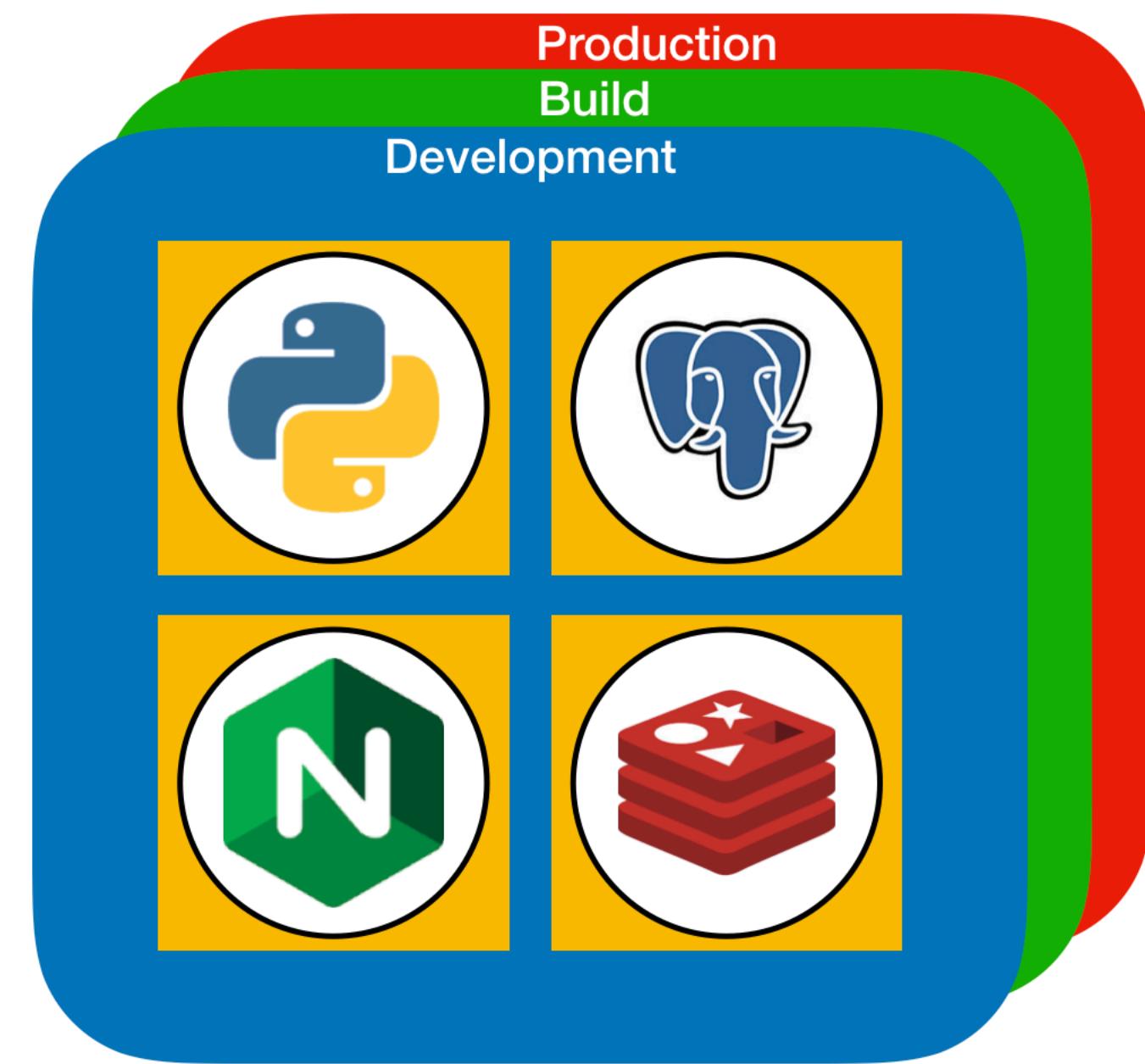


# Containerized Development Workflow



# This is a Very Powerful Concept

- Significant time is spent trying to make development, test, and production environments the same
- Docker allow the same container that is built and tested in development to run unchanged in production
- You can literally deploy an entire environment with a single command



# Creating Your Own Docker Images



# Images from Dockerfiles

- Whatever is needed to build the image is specified in the Dockerfile and checked into GitHub ("infrastructure as code")
- Anyone who uses this image is guaranteed to get the same service instance
- You can start from an operating system image and install all of the prerequisites just like on a VM
- Then copy your application code and resolve and library dependencies
- Finally add the command to run your application

```
FROM alpine:3.7
MAINTAINER John Rofrano "rofrano@us.ibm.com"

# Install just the Python runtime (no dev)
RUN apk add --no-cache \
    python \
    py-pip \
    ca-certificates && \
    pip install --upgrade pip && \
    rm -r /root/.cache

# Expose any ports the app is expecting
ENV PORT 5000
EXPOSE $PORT

# Set up a working folder and install the pre-reqs
WORKDIR /app
ADD requirements.txt /app
RUN pip install -r requirements.txt
ADD . /app

# Run the service
ENV GUNICORN_BIND 0.0.0.0:$PORT
CMD ["gunicorn", "--log-file=-", "app:app"]
```

# Docker build command

- The docker build command is used to build an image from a Dockerfile
- Docker uses a layered filesystem and will reuse layers that have previously been built and not modified
- It will even reuse layers across images
- In the example on the right, all layers except the application layer are cached resulting in a very quick build

```
$ docker build -t my-service .
Sending build context to Docker daemon 36.16MB
Step 1/11 : FROM alpine:3.7
--> 34ea7509dcad
Step 2/11 : MAINTAINER John Rofrano "rofrano@us.ibm.com"
--> Using cache
--> 663be28f442f
Step 3/11 : RUN apk add --no-cache python py-pip certificates && pip install --upgrade pip && rm -r /root/.cache
--> Using cache
--> f0e8446bde3b
Step 4/11 : ENV PORT 5000
--> Using cache
--> e7a1719d988a
Step 5/11 : EXPOSE $PORT
--> Using cache
--> 8ec0b9957940
Step 6/11 : WORKDIR /app
--> Using cache
--> 8f77c8e95bb7
Step 7/11 : ADD requirements.txt /app
--> Using cache
--> 46873628e505
Step 8/11 : RUN pip install -r requirements.txt
--> Using cache
--> ced8d4272148
Step 9/11 : ADD . /app
--> b8df76d195b6
Step 10/11 : ENV GUNICORN_BIND 0.0.0.0:$PORT
--> Running in 20c6c979b175
Removing intermediate container 20c6c979b175
--> 9ae6c0c452bf
Step 11/11 : CMD ["gunicorn", "--log-file=-", "app:app"]
--> Running in ed5c32e4efa1
Removing intermediate container ed5c32e4efa1
--> e4366d80570a
Successfully built e4366d80570a
Successfully tagged my-service:latest
```

# Docker Layered Filesystem

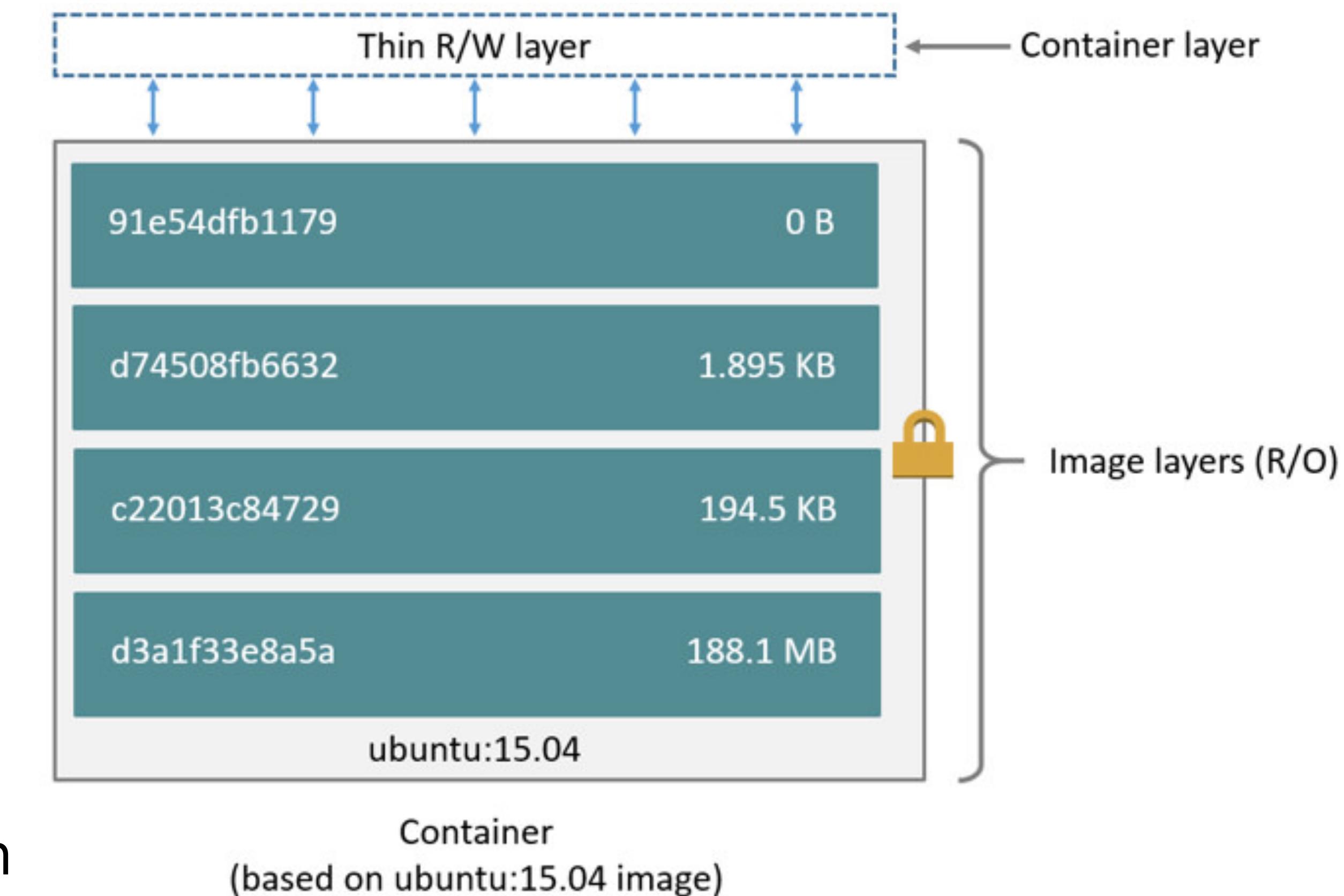
- Docker uses a Copy-On-Write layered filesystem
  - Only changes from the read-only layers are copied
- You can see the layers when you pull or push an image

```
$ docker pull ubuntu:15.04

15.04: Pulling from library/ubuntu
1ba8ac955b97: Pull complete
f157c4e5ede7: Pull complete
0b7e98f84c4c: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:5e279a9df07990286cce22e1b0f5b0490629ca6d187698746ae5e28e604a640e
Status: Downloaded newer image for ubuntu:15.04
```

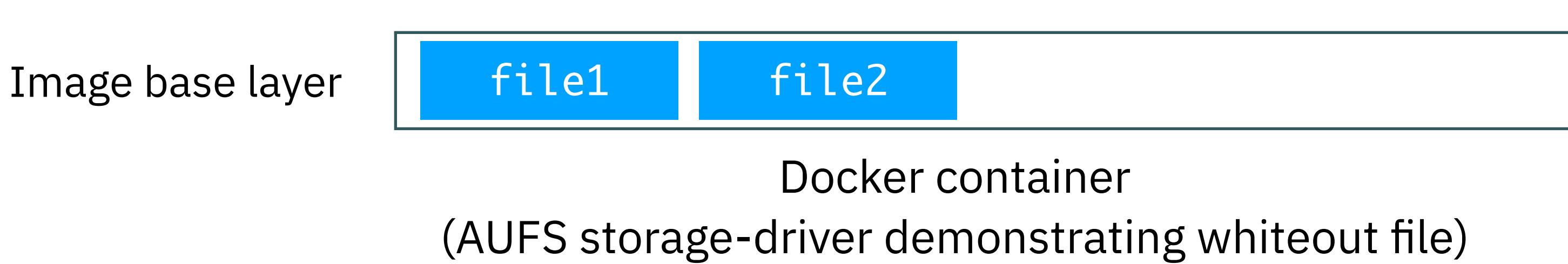
# Images and Layers

- Each Docker image references a list of read-only layers that represent filesystem differences
- Layers are stacked on top of each other to form a base for a container's root filesystem
- When you create a new container, you add a new, thin, writable layer on top of the underlying stack
- All changes made to the running container - such as writing new files, modifying existing files, and deleting files - are written to this thin writable container layer



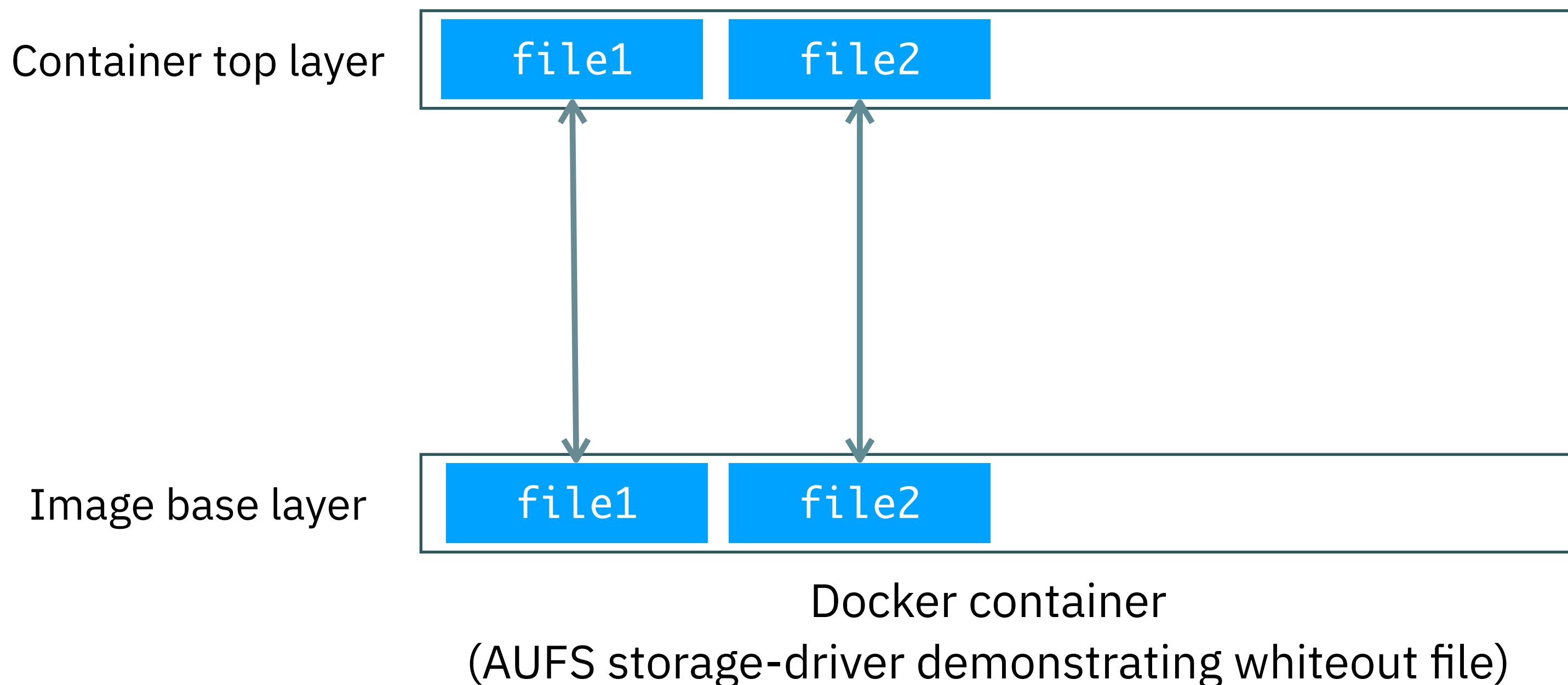
# Base Image

- Start with a base image



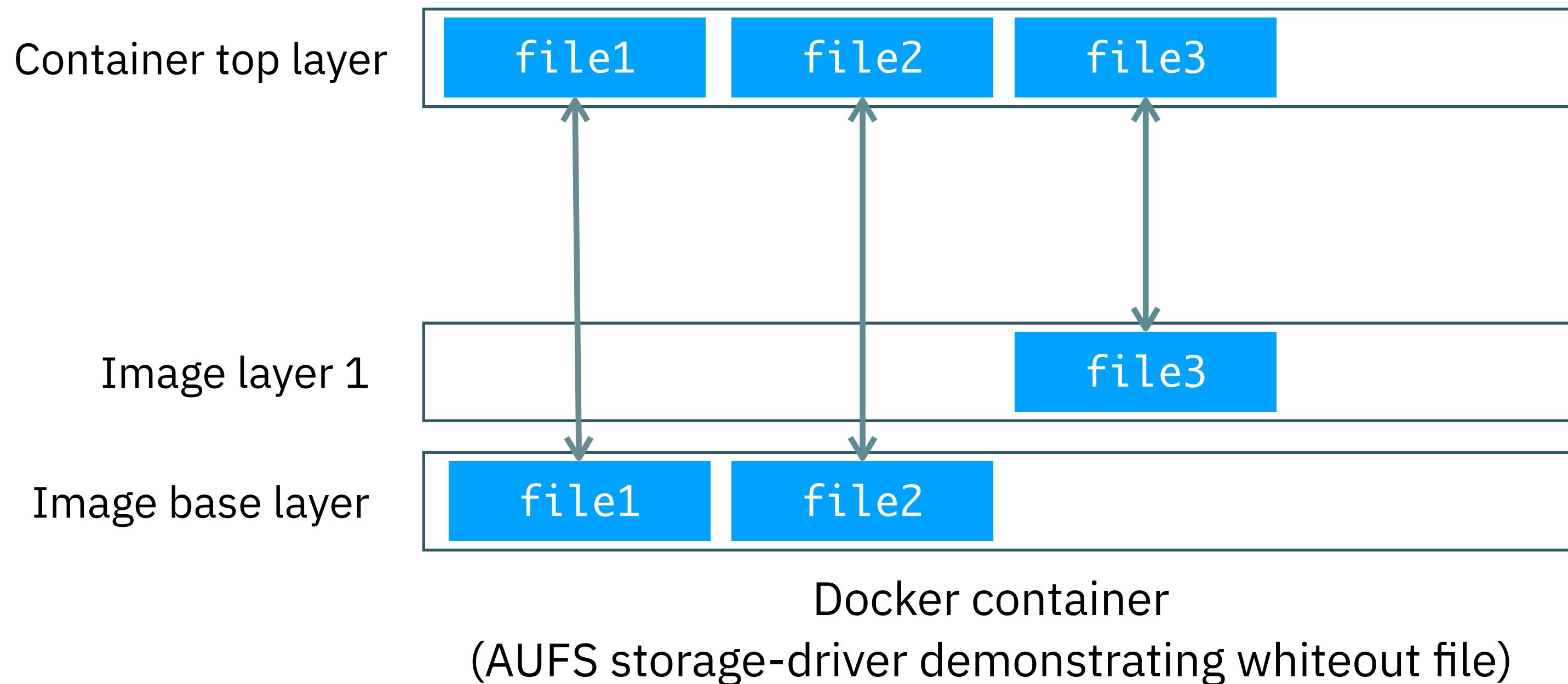
# Container Layers are exposed to Top

- Files from the read-only layers below are visible to the top layer



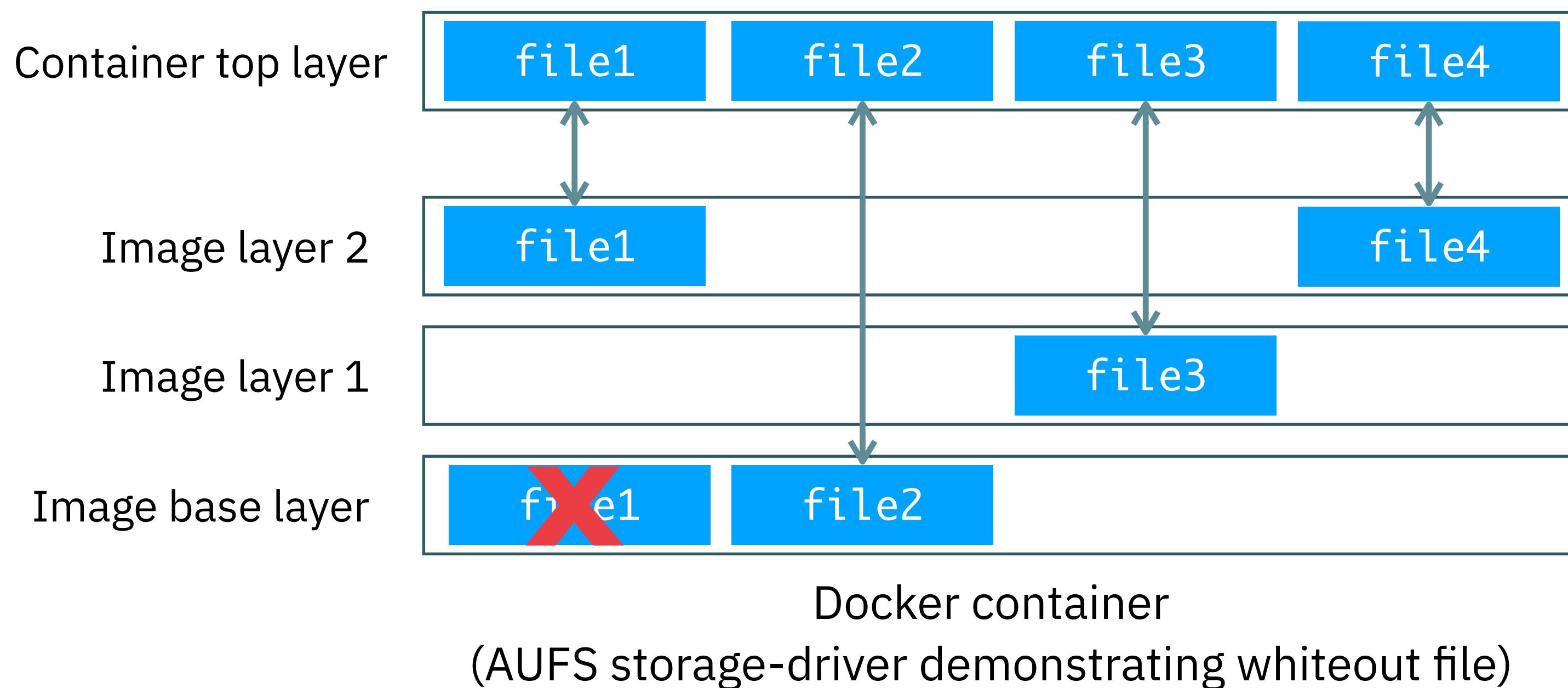
# Add Layers with more Files

- As you add layers more files become visible at the top layer



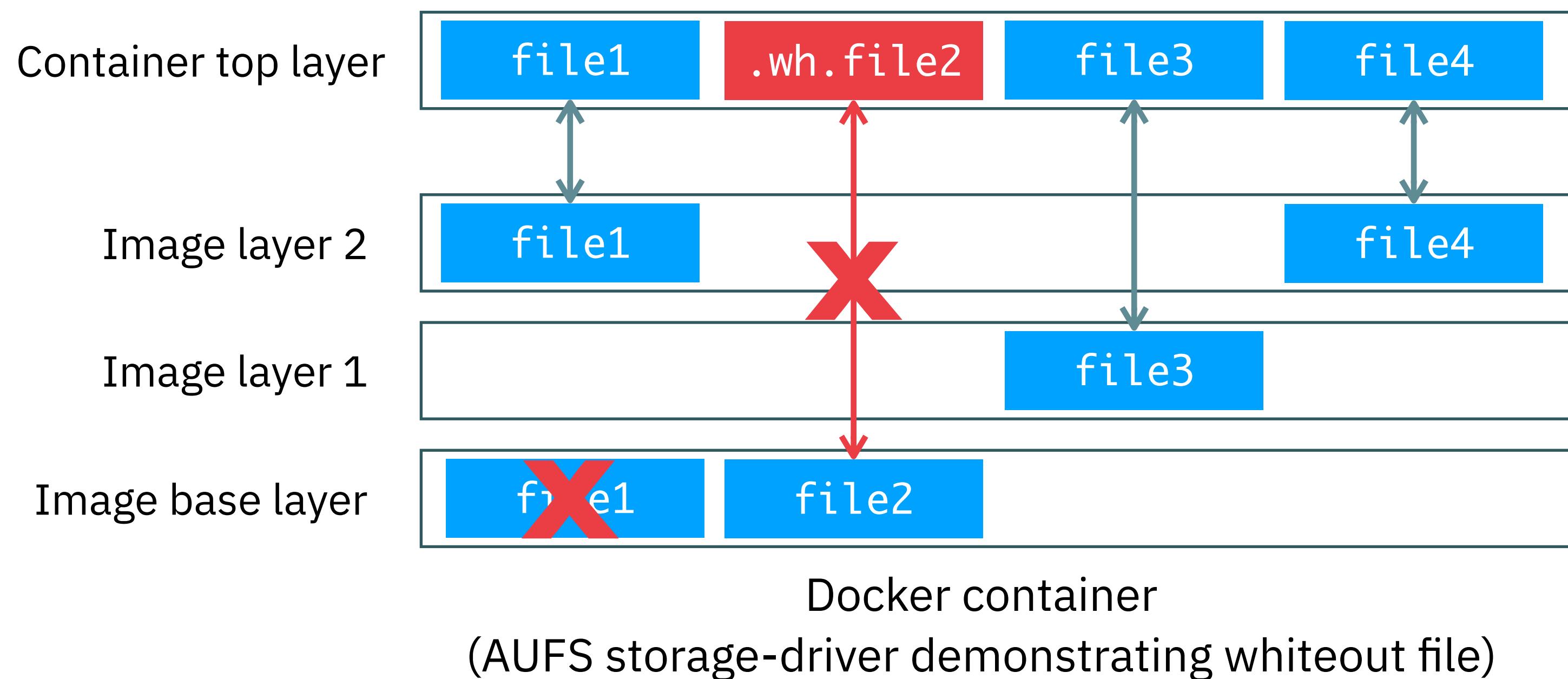
# New Versions

- A new version of file1 is added and it hides the old file1 version



# White-out Files

- Special files called "white-out" files are used to make files appear to be deleted



# How Layers are Reused

- Note when you provision with Vagrant how Redis:Alpine reuses Alpine layer:

```
==> default: Running provisioner: docker...
    default: Installing Docker onto machine...
==> default: Pulling Docker images...
==> default: -- Image: alpine:latest
==> default: stdin: is not a tty
==> default: latest: Pulling from library/alpine
==> default: 3690ec4760f9: Pulling fs layer
==> default: 3690ec4760f9: Verifying Checksum
==> default: 3690ec4760f9: Download complete
==> default: 3690ec4760f9: Pull complete
==> default: Digest: sha256:1354db23ff5478120c980eca1611a51c9f2b88b61f24283ee8200bf9a54f2e5c
==> default: Status: Downloaded newer image for alpine:latest
==> default: -- Image: redis:alpine
==> default: stdin: is not a tty
==> default: alpine: Pulling from library/redis
==> default: 3690ec4760f9: Already exists
==> default: 5e231f7bdf9d: Pulling fs layer
==> default: 5a74fb2950f8: Pulling fs layer
. . .
```

# How Layers are Reused

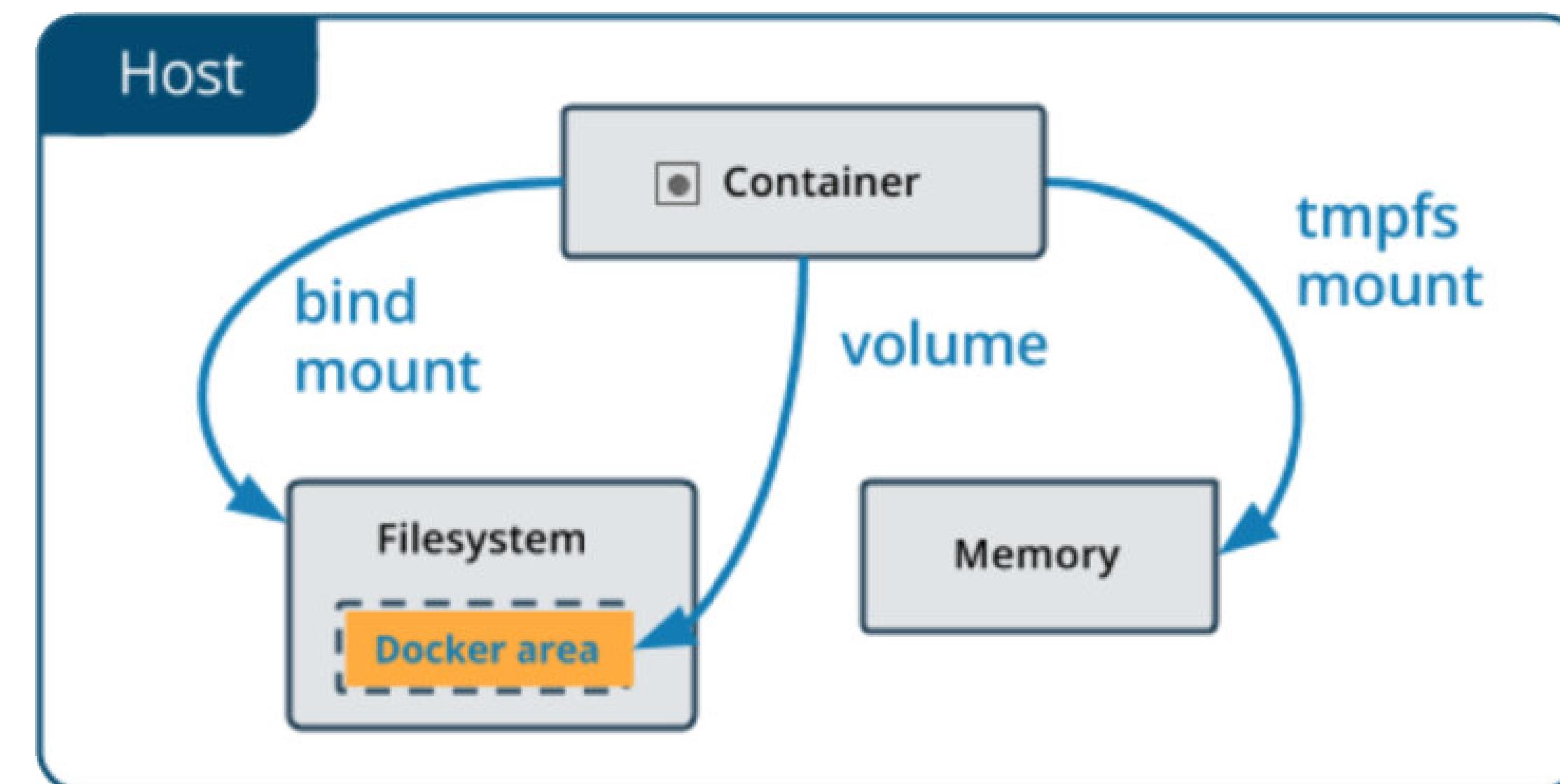
- Note when you provision with Vagrant how Redis:Alpine reuses Alpine layer:

```
==> default: Running provisioner: docker...
    default: Installing Docker onto machine...
==> default: Pulling Docker images...
==> default: -- Image: alpine:latest
==> default: stdin: is not a tty
==> default: latest: Pulling from library/alpine
==> default: 3690ec4760f9: Pulling fs layer
==> default: 3690ec4760f9: Verifying Checksum
==> default: 3690ec4760f9: Download complete
==> default: 3690ec4760f9: Pull complete
==> default: Digest: sha256:1554ab25ff5478120c980eca1611a51c9f2b88b61f24283ee8200bf9a54f2e5c
==> default: Status: Downloaded newer image for alpine:latest
==> default: -- Image: redis:alpine
==> default: stdin: is not a tty
==> default: alpine: Pulling from library/redis
==> default: 3690ec4760f9: Already exists
==> default: 3e231f7ba79a: Pulling fs layer
==> default: 5a74fb2950f8: Pulling fs layer
. . .
```

Alpine layer is being reused  
by Redis:Alpine

# Docker Volumes

- Volumes are easier to back up or migrate than bind mounts.
- You can manage volumes using Docker CLI commands or the Docker API.
- Volumes work on both Linux and Windows containers.
- Volumes can be more safely shared among multiple containers.
- Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
- New volumes can have their content pre-populated by a container



# Bind Mount Volumes for Redis

```
#####
# Add Redis docker container
#####
config.vm.provision "shell", inline: <<-SHELL
  # Prepare Redis data share
  sudo mkdir -p /var/lib/redis/data
  sudo chown vagrant:vagrant /var/lib/redis/data
SHELL

# Add Redis docker container
config.vm.provision "docker" do |d|
  d.pull_images "redis:alpine"
  d.run "redis:alpine",
    args: "--restart=always -d --name redis -h redis -p 6379:6379 -v /var/lib/redis/data:/data"
end
```

# Bind Mount Volumes for Redis

```
#####
# Add Redis docker container
#####
config.vm.provision "shell", inline: <<-SHELL
  # Prepare Redis data share
  sudo mkdir -p /var/lib/redis/data
  sudo chown vagrant:vagrant /var/lib/redis/data
SHELL

# Add Redis docker container
config.vm.provision "docker" do |d|
  d.pull_images "redis:alpine"
  d.run "redis:alpine",
    args: "--restart=always -d --name redis -h redis -p 6379:6379 -v /var/lib/redis/data:/data"
end
```

A red arrow points from the highlighted path `/var/lib/redis/data` in the first code block to the corresponding path `/var/lib/redis/data` in the second code block, indicating they represent the same volume.

volume on the host

# Bind Mount Volumes for Redis

```
#####
# Add Redis docker container
#####
config.vm.provision "shell", inline: <<-SHELL
  # Prepare Redis data share
  sudo mkdir -p /var/lib/redis/data
  sudo chown vagrant:vagrant /var/lib/redis/data
SHELL

# Add Redis docker container
config.vm.provision "docker" do |d|
  d.pull_images "redis:alpine"
  d.run "redis:alpine",
    args: "--restart=always -d --name redis -h redis -p 6379:6379 -v /var/lib/redis/data:/data"
end
```

volume on the host

Volume in container

# Using a Docker Volume for Redis

- We simply give docker a name for the volume like: redis\_data
- Docker manages this volume for us

```
# Add Redis docker container
config.vm.provision "docker" do |d|
  d.pull_images "redis:alpine"
  d.run "redis:alpine",
    args: "--restart=always -d --name redis -h redis -p 6379:6379 -v redis_data:/data"
end
```

# Using a Docker Volume for Redis

- We simply give docker a name for the volume like: `redis_data`
- Docker manages this volume for us

```
# Add Redis docker container
config.vm.provision "docker" do |d|
  d.pull_images "redis:alpine"
  d.run "redis:alpine",
    args: "--restart=always -d --name redis -h redis -p 6379:6379 -v redis_data /data"
end
```

Docker managed volume  
on the host



# Using a Docker Volume for Redis

- We simply give docker a name for the volume like: redis\_data
- Docker manages this volume for us

```
# Add Redis docker container
config.vm.provision "docker" do |d|
  d.pull_images "redis:alpine"
  d.run "redis:alpine",
    args: "--restart=always -d --name redis -h redis -p 6379:6379 -v redis_data:/data"
end
```

Docker managed volume  
on the host

Volume in container



# Docker volume ls

- We can use docker volume ls to list the volumes:

```
$ docker volume ls
DRIVER          VOLUME NAME
local           redis_volume
```

An example with named and un-named volumes

```
$ docker volume ls
DRIVER          VOLUME NAME
local           4e469efb9599682f89ff417174256b752af870c54546021a36d2955d379c4f5
local           87df74bab30c1a4e9f78d4e4a7771b69a2cc5b347d55184d3791204997eaa109
local           368efa07e9f64be73ace7d75c788a363433f3f2007b3de8b74972632d8fcab29
local           cbd7d35d1695b927b59613ee7c9f2cb9b65ed6a9aecbf7745aa233552a9e24de
local           mysql-data
local           postgres-data
```

# Use Minimal Images Whenever Possible

- Minimal images are more secure because they have a smaller attack surface
  - Don't load anything into an image that you don't need
- Minimal images load faster
  - Alpine is only 4.4MB compared to Ubuntu 16.04 at 117MB!
  - PostgreSQL Alpine is 71MB vs 228MB

Docker Images		
REPOSITORY	TAG	SIZE
centos	latest	202MB
debian	stretch-slim	55.3MB
debian	latest	101MB
ubuntu	16.04	117MB
ubuntu	latest	86.7MB
python	2-alpine	58.3MB
python	2.7-slim	120MB
python	3.7-alpine	78.2MB
python	3.7-slim	143MB
bitnami/minideb	latest	53.7MB
alpine	latest	4.41MB
alpine	3.7	4.21MB
postgres	alpine	71.6MB
postgres	latest	228MB



# Containers should be...

- Stateless
  - All state should be maintained in a Database, Object Store, or Persistent Volume
- Light Weight
  - Only one process per container i.e., Container dies when process dies
- Immutable
  - Do not install an ssh daemon or any other means of entering the container!
- Run from Docker Registry Images or Built from Dockerfiles
  - Treated like code, versioned, and reconstituted when needed... not built by hand!

# Hands-On

# Docker Images

- We can run `docker images` to see what Images we have locally

```
(venv) vagrant@kubernetes:~$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
python	3.8-slim	dbb175e1c337	4 days ago	174MB
redis	alpine	6f63d037b592	4 weeks ago	29.3MB
alpine	latest	965ea09ff2eb	4 weeks ago	5.55MB

# Docker Images

- We can run `docker images` to see what Images we have locally

These are from the Vagrantfile

```
(venv) vagrant@kubernetes:~$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
python	3.8-slim	dbb175e1c337	4 days ago	174MB
redis	alpine	6f63d037b592	4 weeks ago	29.3MB
alpine	latest	965ea09ff2eb	4 weeks ago	5.55MB

# Working With Docker

- We can run `docker ps` to see what containers we have running

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                 NAMES
ab46390b31b7        redis:alpine       "docker-entrypoint.sh"   26 minutes ago    Up 26 minutes      0.0.0.0:6379->6379/tcp   redis
```

# Working With Docker

- We can run `docker ps` to see what containers we have running

```
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
ab46390b31b7 redis:alpine "docker-entrypoint.sh" 26 minutes ago Up 26 minutes 0.0.0.0:6379->6379/tcp redis
```

- We can use the `-a` switch to see all containers, even stopped ones

```
$ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
9a25d7302cf0 hello-world "/hello" 17 minutes ago Exited (0) 17 minutes ago silly_boyd
ab46390b31b7 redis:alpine "docker-entrypoint.sh" 29 minutes ago Up 29 minutes 0.0.0.0:6379->6379/tcp redis
```

# Remove Exited Containers

- We can remove a container that has exited with the command:

```
docker rm <container id> | <name>
```

```
$ docker ps -a
CONTAINER ID  IMAGE          COMMAND           CREATED          STATUS          PORTS          NAMES
9a25d7302cf0  hello-world   "/hello"         17 minutes ago  Exited (0) 17 minutes ago
ab46390b31b7  redis:alpine  "docker-entrypoint.sh"  29 minutes ago  Up 29 minutes  0.0.0.0:6379->6379/tcp  redis
silly_boyd

$ docker rm silly_boyd
silly_boyd

$ docker ps -a
CONTAINER ID  IMAGE          COMMAND           CREATED          STATUS          PORTS          NAMES
ab46390b31b7  redis:alpine  "docker-entrypoint.sh"  29 minutes ago  Up 29 minutes  0.0.0.0:6379->6379/tcp  redis
```

# Remove Exited Containers

- We can remove a container that has exited with the command:

```
docker rm <container id> | <name>
```

```
$ docker ps -a
CONTAINER ID  IMAGE          COMMAND           CREATED        STATUS        PORTS     NAMES
9a25d7302cf0  hello-world   "/hello"          17 minutes ago  Exited (0)  17 minutes ago
ab46390b31b7  redis:alpine  "docker-entrypoint.sh"  29 minutes ago  Up 29 minutes  0.0.0.0:6379->6379/tcp  redis
silly_boyd

$ docker rm silly_boyd
silly_boyd
```

Remove the container named “silly\_boyd”

```
$ docker ps -a
CONTAINER ID  IMAGE          COMMAND           CREATED        STATUS        PORTS     NAMES
ab46390b31b7  redis:alpine  "docker-entrypoint.sh"  29 minutes ago  Up 29 minutes  0.0.0.0:6379->6379/tcp  redis
```

# Remove Unwanted Images

- We can remove an image that we no longer with the:

```
docker rmi <image id> | <repository:tag>
```

```
$ docker images
REPOSITORY          TAG        IMAGE ID      CREATED       SIZE
redis               alpine     906fffc0e2f4  35 hours ago  20.38 MB
alpine              latest     baa5d63471ea  39 hours ago  4.803 MB
hello-world         latest     c54a2cc56cbb  3 months ago  1.848 kB
$ docker rmi c54a2cc56cbb
Untagged: hello-world:latest
Untagged: hello-world@sha256:0256e8a36e2070f7bf2d0b0763dbabdd67798512411de4cdcf9431a1feb60fd9
Deleted: sha256:c54a2cc56cbb2f04003c1cd4507e118af7c0d340fe7e2720f70976c4b75237dc
Deleted: sha256:a02596fdd012f22b03af6ad7d11fa590c57507558357b079c3e8cebceb4262d7

$ docker images
REPOSITORY          TAG        IMAGE ID      CREATED       SIZE
redis               alpine     906fffc0e2f4  35 hours ago  20.38 MB
alpine              latest     baa5d63471ea  39 hours ago  4.803 MB
```

# Remove Unwanted Images

- We can remove an image that we no longer with the:

```
docker rmi <image id> | <repository:tag>
```

```
$ docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
redis           alpine   906fffc0e2f4  35 hours ago  20.38 MB
alpine          latest   baa5d63471ea  39 hours ago  4.803 MB
hello-world     latest   c54a2cc56cbb  3 months ago  1.848 kB

$ docker rmi c54a2cc56cbb
Untagged: hello-world:latest
Untagged: hello-world@sha256:0256e8a36e20701b12a0b763ababab // 98512411ae4cac19431a1fe091a9
Deleted: sha256:c54a2cc56cbb2f04003c1cd4507e118af7c0d340fe7e2720f70976c4b75237dc
Deleted: sha256:a02596fdd012f22b03af6ad7d11fa590c57507558357b079c3e8cebceb4262d7

$ docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
redis           alpine   906fffc0e2f4  35 hours ago  20.38 MB
alpine          latest   baa5d63471ea  39 hours ago  4.803 MB
```

A red arrow points from the text "Remove the image with id ‘c54a2cc56cbb’" to the command \$ docker rmi c54a2cc56cbb.

# Creating Containers from Images



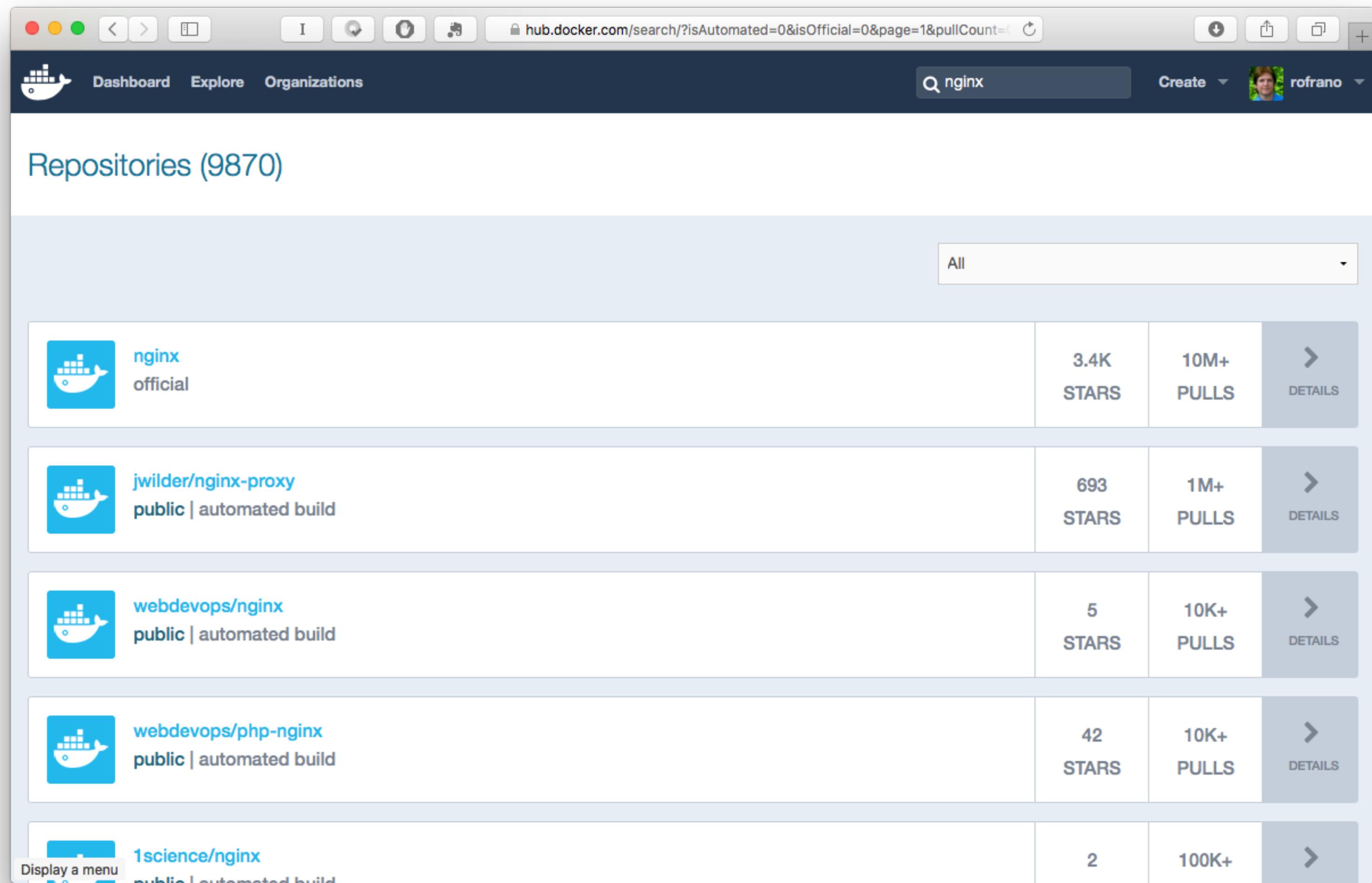
# Lets Use an Existing Docker Image

- Docker Images can be used as-is as a quick way of providing middleware
- Let's run NGINX without installing anything!

# Where To Get Images?

- **Docker Hub**
- Contains 1000's of Docker images with almost every software you can imagine
- Usually “official” images come from the creator of the software
- Most have documentation on how to best use them

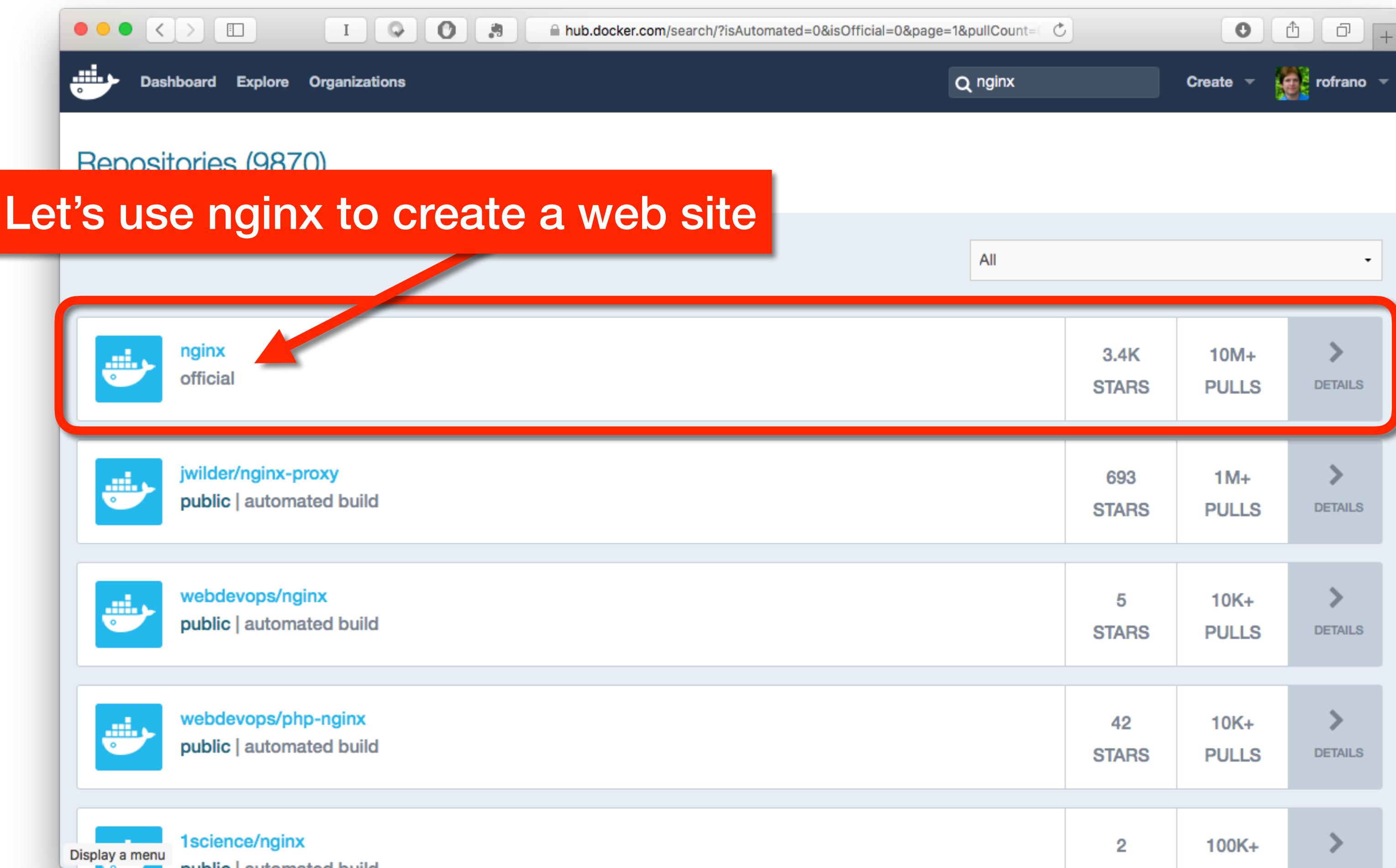
[https://hub.docker.com/\\_/nginx/](https://hub.docker.com/_/nginx/)



# Where To Get Images?

- **Docker Hub**
- Contains 1000's of Docker images with almost every software you can imagine
- Usually “official” images come from the creator of the software
- Most have documentation on how to best use them

[https://hub.docker.com/\\_/nginx/](https://hub.docker.com/_/nginx/)



# Most Images Have Documentation



## How to use this image 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 the 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.

# Most Images Have Documentation

**nginx**

Container docs will tell you how to run them

How to use this image  
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
```

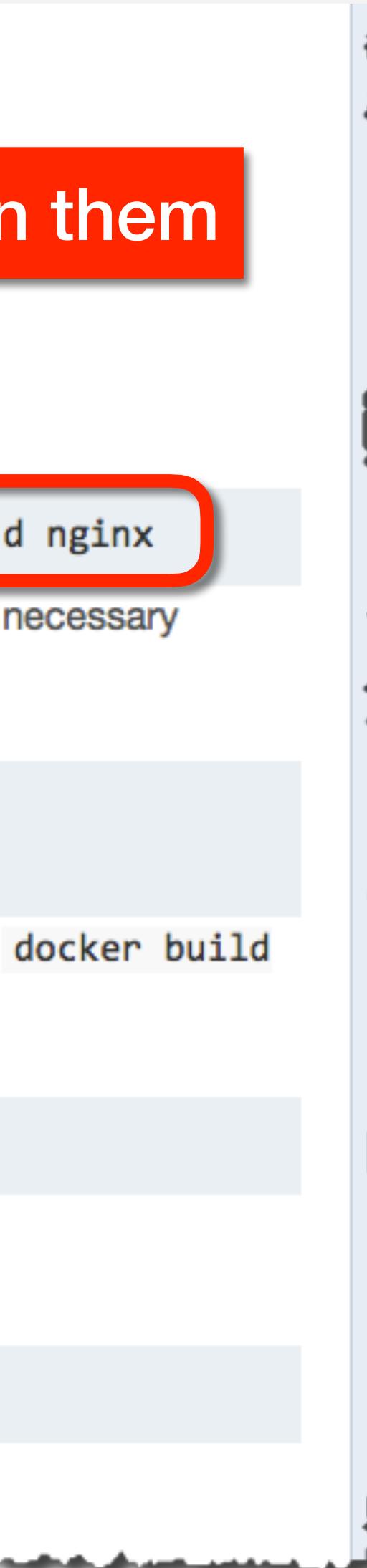
Place this file in the same directory as your directory of content ("static-html-directory"), run docker build -t some-content-nginx . , then start your container:

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

## exposing the 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.



# Let's Run Nginx

- We can run nginx as a web server with the command:

```
docker run -d -p 8080:80 nginx:alpine
```

```
$ docker run -d -p 8080:80 nginx:alpine
8e82325aa105cb0bc4f192649f01c32e988045bc59b5d036d5c90505f2cf662d
```

# Let's Run Nginx

- We can run nginx as a web server with the command:

```
docker run -d -p 8080:80 nginx:alpine
```

Run as a Daemon

```
$ docker run -d -p 8080:80 nginx:alpine  
8e82325aa105cb0bc4f192649f01c32e988045bc59b5d036d5c90505f2cf662d
```

# Let's Run Nginx

- We can run nginx as a web server with the command:

```
docker run -d -p 8080:80 nginx:alpine
```

Run as a Daemon

```
$ docker run -d -p 8080:80 nginx:alpine  
8e82325aa105cb0bc4f182649f01c32e988045bc59b5d036d5c90505f2cf662d
```

Map port 80 in container  
to 8080 on the Host

# Let's Run Nginx

- We can run nginx as a web server with the command:

```
docker run -d -p 8080:80 nginx:alpine
```

Run as a Daemon

```
$ docker run -d -p 8080:80 nginx:alpine  
8e82325aa105cb0bc4f182649f01c32e9 3045bc59b5d036d5c90505f2cf662d
```

Map port 80 in container to 8080 on the Host

Run nginx from local cache or Docker Hub

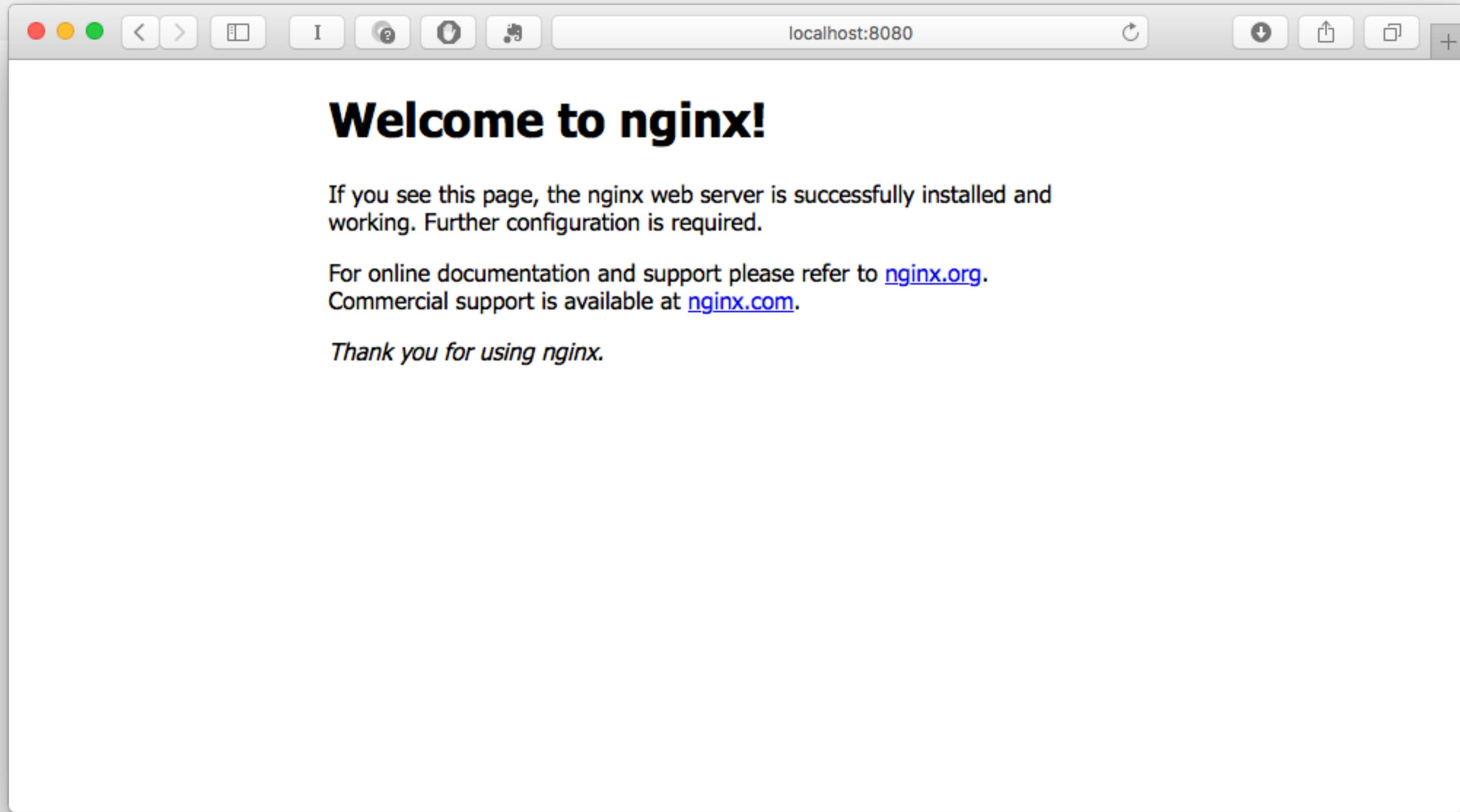
# NGINX will be Pulled

- Since we don't have a local nginx image it will be pulled from Docker hub the first time

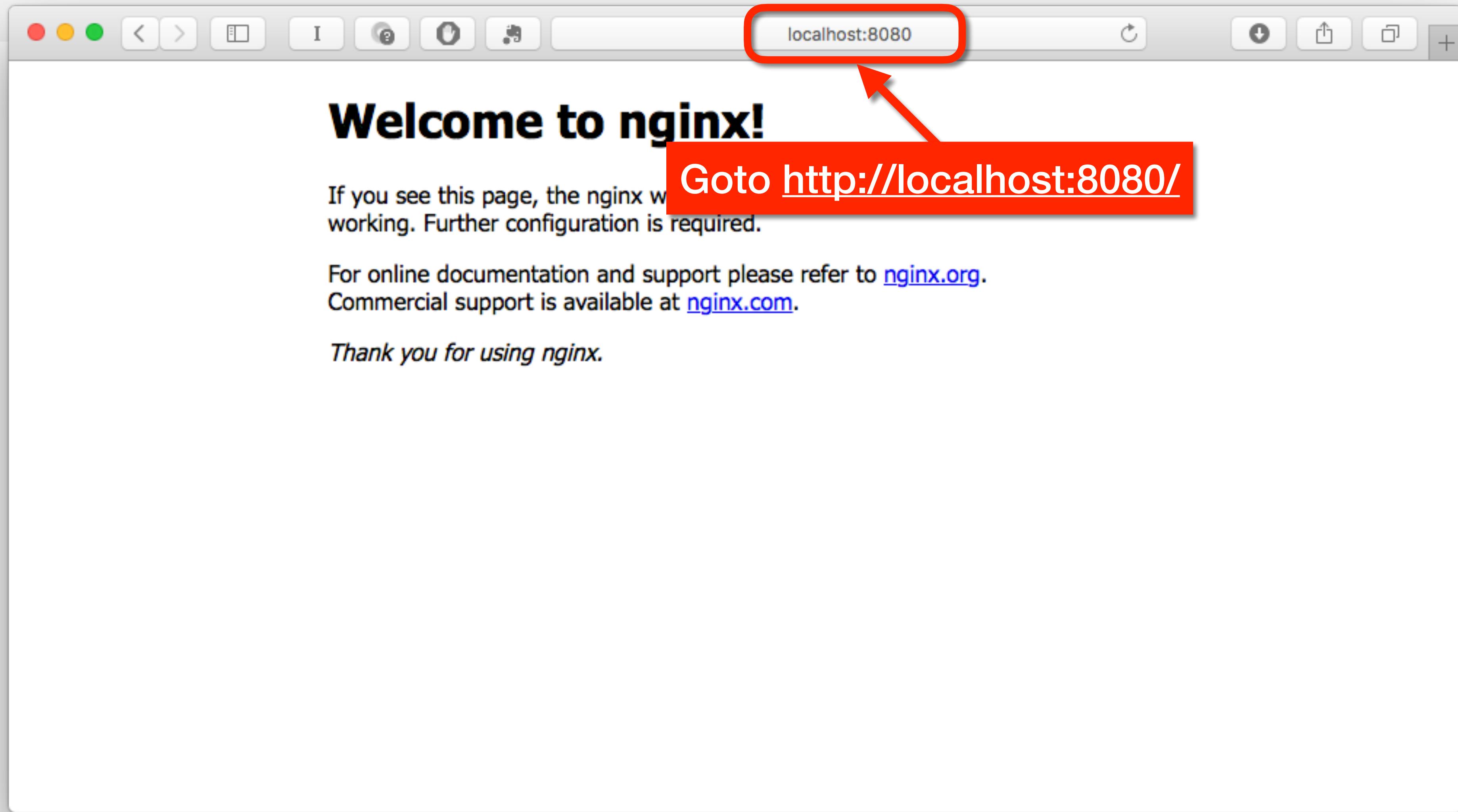
```
$ docker run -d -p 8080:80 nginx:alpine
```

```
Unable to find image 'nginx:alpine' locally
alpine: Pulling from library/nginx
89d9c30c1d48: Already exists
110ad692b782: Pull complete
Digest: sha256:559cde0e812dba817c131173217712bf34c2c80a0b6892409cbc7d760fca8be
Status: Downloaded newer image for nginx:alpine
53cee8d8cf12ca91e7b2fd9173d03772757c36851cf70a855210005e9fa9b26e
```

# In A Web browser



# In A Web browser



# Stop a Container

- We can stop the running container with the command:

```
docker stop <container name> | <container id>
```

```
$ docker ps
CONTAINER ID  IMAGE      COMMAND           CREATED          STATUS          PORTS          NAMES
0d48962ddc38  nginx:alpine "nginx -g 'daemon off'"  4 minutes ago   Up  4 minutes   443/tcp, 0.0.0.0:8080->80/tcp   stoic_shaw
ab46390b31b7  redis:alpine  "docker-entrypoint.sh"  51 minutes ago  Up  51 minutes  0.0.0.0:6379->6379/tcp   redis

$ docker stop stoic_shaw
stoic_shaw

$ docker rm stoic_shaw
stoic_shaw

$ docker ps -a
CONTAINER ID  IMAGE      COMMAND           CREATED          STATUS          PORTS          NAMES
ab46390b31b7  redis:alpine  "docker-entrypoint.sh"  52 minutes ago  Up  52 minutes  0.0.0.0:6379->6379/tcp   redis
```

# Stop a Container

- We can stop the running container with the command:

```
docker stop <container name> | <container id>
```

```
$ docker ps
CONTAINER ID  IMAGE      COMMAND           CREATED        STATUS          PORTS          NAMES
0d48962ddc38  nginx:alpine "nginx -g 'daemon off'"  4 minutes ago  Up 4 minutes  443/tcp, 0.0.0.0:8080->80/tcp  stoic_shaw
ab46390b31b7  redis:alpine  "docker-entrypoint.sh"  51 minutes ago  Up 51 minutes  0.0.0.0:6379->6379/tcp  redis

$ docker stop stoic_shaw
stoic_shaw

$ docker rm stoic_shaw
stoic_shaw

$ docker ps -a
CONTAINER ID  IMAGE      COMMAND           CREATED        STATUS          PORTS          NAMES
ab46390b31b7  redis:alpine  "docker-entrypoint.sh"  52 minutes ago  Up 52 minutes  0.0.0.0:6379->6379/tcp  redis
```



Stop container “stoic\_shaw”

# Stop a Container

- We can stop the running container with the command:

```
docker stop <container name> | <container id>
```

```
$ docker ps
CONTAINER ID  IMAGE      COMMAND           CREATED        STATUS          PORTS          NAMES
0d48962ddc38  nginx:alpine "nginx -g 'daemon off'"  4 minutes ago  Up 4 minutes  443/tcp, 0.0.0.0:8080->80/tcp  stoic_shaw
ab46390b31b7  redis:alpine  "docker-entrypoint.sh"  51 minutes ago  Up 51 minutes  0.0.0.0:6379->6379/tcp  redis

$ docker stop stoic_shaw
$ docker rm stoic_shaw
```

**Stop container “stoic\_shaw”**

**Remove container “stoic\_shaw”**

```
$ docker ps -a
CONTAINER ID  IMAGE      COMMAND           CREATED        STATUS          PORTS          NAMES
ab46390b31b7  redis:alpine  "docker-entrypoint.sh"  52 minutes ago  Up 52 minutes  0.0.0.0:6379->6379/tcp  redis
```

# Creating Containers from Dockerfiles



# Using A Dockerfile

- We can create a new Image from a Dockerfile
- The Dockerfile is just a text file that contains the commands to build the image
- This allows you to treat Images like code

# DevOps Workshop Dockerfile

```
FROM python:3.8-slim

# Create working folder and install dependencies
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy the application contents
COPY service/ ./service/

# Expose any ports in the environment
ENV PORT 8080
EXPOSE $PORT

ENV GUNICORN_BIND 0.0.0.0:$PORT
CMD ["gunicorn", "--log-level=info", "service:app"]
```

# Build the Dockerfile

```
$ cd /vagrant/  
$ docker build -t hitcounter:1.0 .  
  
Sending build context to Docker daemon 443.4kB  
Step 1/9 : FROM python:3.7-slim  
--> 07ee12a5eb2a  
Step 2/9 : WORKDIR /app  
--> Running in c1e559050e81  
Removing intermediate container c1e559050e81  
--> 3038f6baa90d  
Step 3/9 : COPY requirements.txt /app  
--> f1f0fa408d34  
Step 4/9 : RUN pip install --no-cache-dir -r requirements.txt  
--> Running in 282e12ad02ff  
Collecting Flask==1.1.1  
  Downloading https://files.pythonhosted.org/packages/9b/  
93/628509b8d5dc749656a9641f4caf13540e2cdec85276964ff8f43bbb1d3b/Flask-1.1.1-py2.py3-none-any.whl (94kB)  
  
... lots of python packages get installed here ...  
  
Step 5/9 : COPY service/ /app/service/  
--> 434eb6e1e911  
Step 6/9 : ENV PORT 8080  
--> Running in ae8c0dbce3b7  
Removing intermediate container ae8c0dbce3b7  
--> 38b81826fcce  
Step 7/9 : EXPOSE $PORT  
--> Running in 9a359968e7b3  
Removing intermediate container 9a359968e7b3  
--> a8e7d7e360f6  
Step 8/9 : ENV GUNICORN_BIND 0.0.0.0:$PORT  
--> Running in f4b72400abec  
Removing intermediate container f4b72400abec  
--> 813303bb1801  
Step 9/9 : CMD ["gunicorn", "--log-level=info", "service:app"]  
--> Running in 3b71a2ccbe1e  
Removing intermediate container 3b71a2ccbe1e  
--> 8ac3c707ab08  
Successfully built 8ac3c707ab08  
Successfully tagged hitcounter:1.0
```

# Check Your Images

- docker images

```
$ docker images
REPOSITORY          TAG        IMAGE ID      CREATED       SIZE
hitcounter           1.0        8ac3c707ab08  2 minutes ago  200MB
python               3.7-slim   07ee12a5eb2a  4 days ago    179MB
nginx               alpine     b6753551581f  4 weeks ago   21.4MB
redis               alpine     6f63d037b592  4 weeks ago   29.3MB
alpine              latest     965ea09ff2eb  4 weeks ago   5.55MB
$
```

# Check Your Images

- docker images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hitcounter	1.0	8ac3c707ab08	2 minutes ago	200MB
python	3.7-slim	07ee12a5eb2a	4 days ago	179MB
nginx	alpine	b6753551581f	4 weeks ago	21.4MB
redis	alpine	6f63d037b592	4 weeks ago	29.3MB
alpine	latest	965ea09ff2eb	4 weeks ago	5.55MB

This is the new hit counter:1.0 image that you just built

# Run the Container

- docker run --rm -p 5000:5000 hitcounter:1.0

```
$ docker run --rm -p 5000:5000 hitcounter:1.0

[2019-11-20 00:54:33 +0000] [1] [INFO] Starting gunicorn 20.0.0
[2019-11-20 00:54:33 +0000] [1] [INFO] Listening at: http://0.0.0.0:5000 (1)
[2019-11-20 00:54:33 +0000] [1] [INFO] Using worker: sync
[2019-11-20 00:54:33 +0000] [9] [INFO] Booting worker with pid: 9
```

# What Happened?



A screenshot of a web browser window titled "localhost". The address bar shows "localhost". The content area displays a JSON-like error response:

```
- {
  error : "Service is unavailable",
  message : "503 Service Unavailable: 'NoneType' object has no attribute 'get'",
  status : 503
}
```

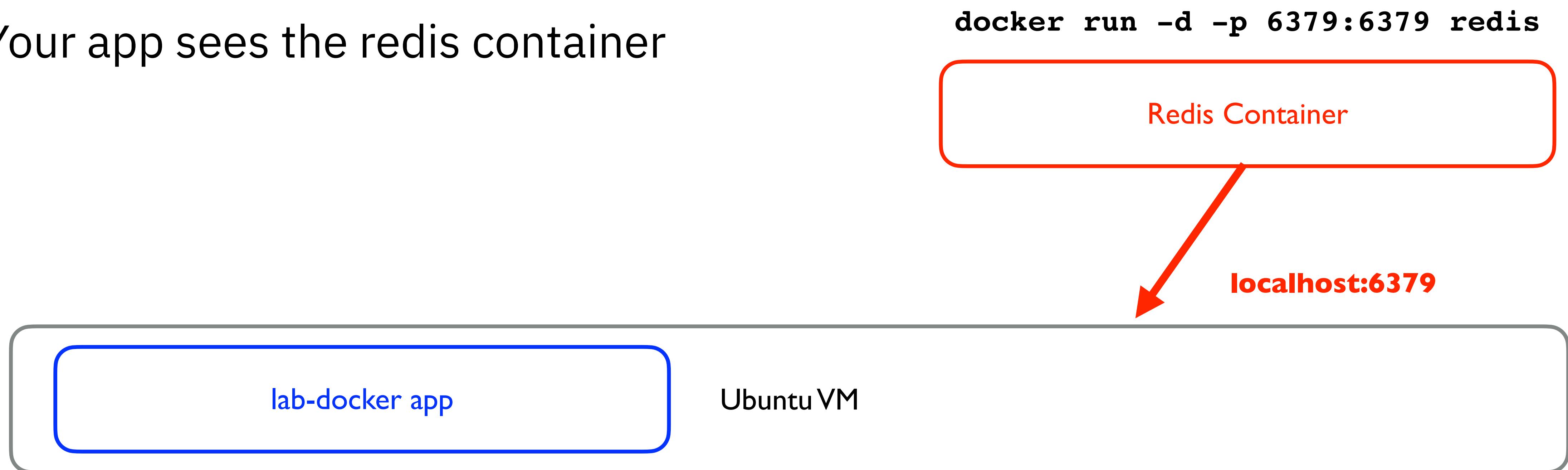
The "Parsed" tab is selected in the top right corner of the browser window.

# Where is Redis?

- Redis is running in a container which responds to 127.0.0.1:6379 because we forwarded the port to the VM
- You can talk to from the VM through it's forwarded port
- Python running in a Container cannot talk to the VM and so it cannot talk to Redis
- We must link the containers together

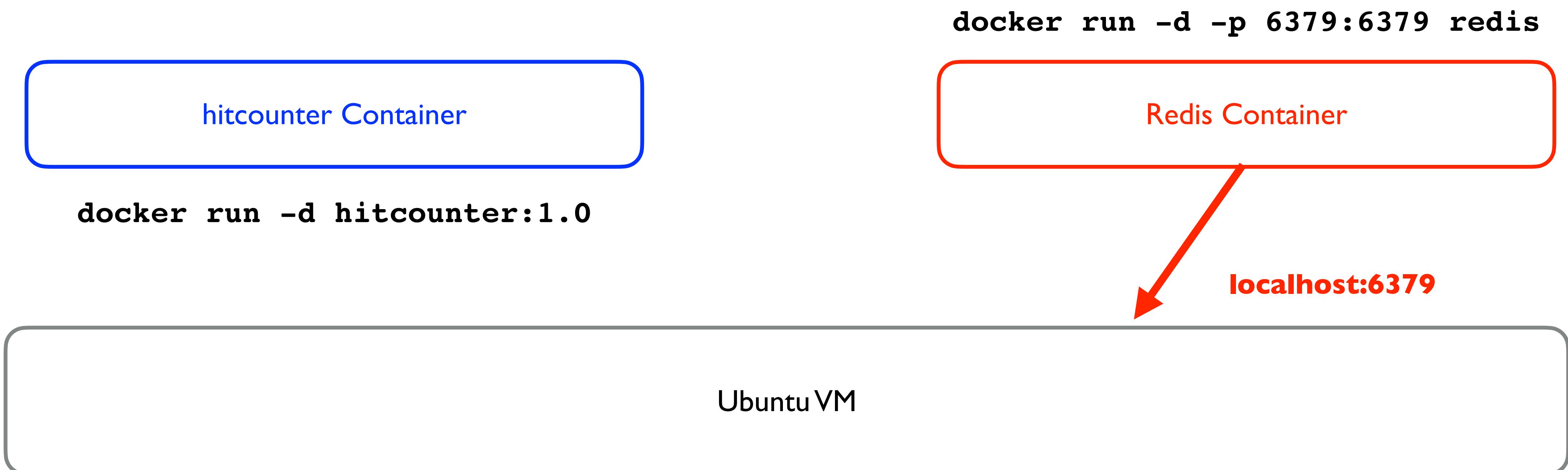
# Running in a VM

- Your code is running on localhost
- The container is exposed on localhost
- Your app sees the redis container



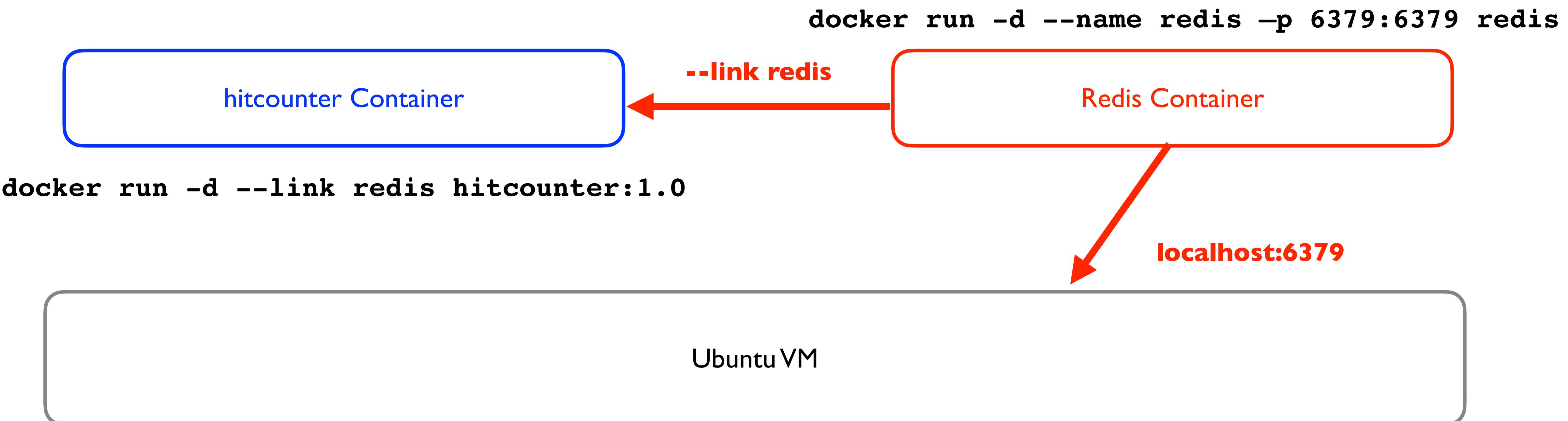
# Running in a Container

- Your code is running in a container not on localhost
- Your app can no longer see redis



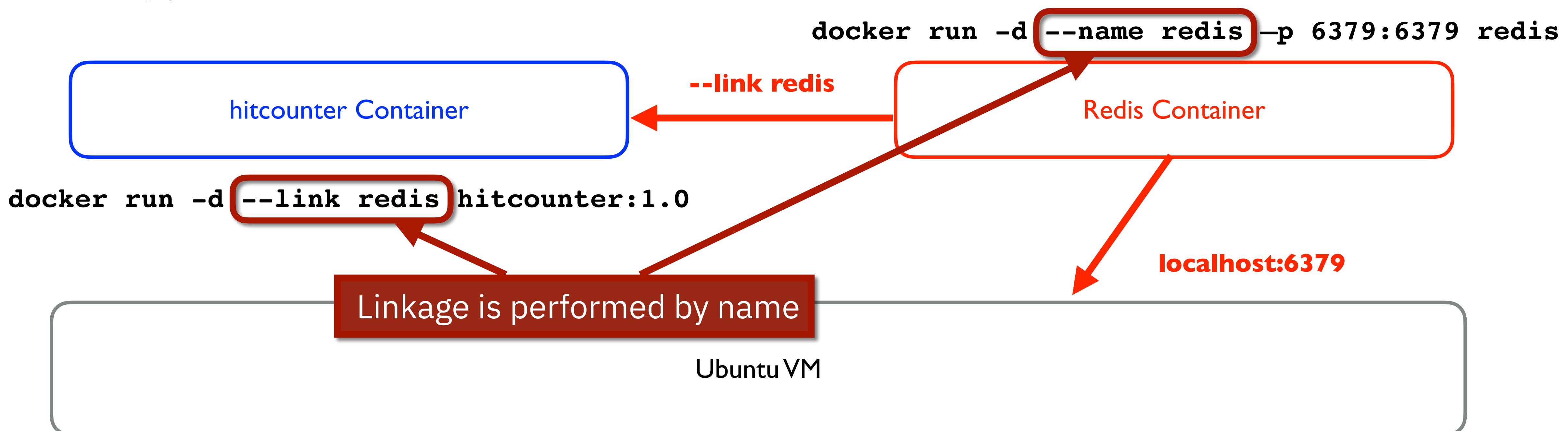
# Solution

- Use the `--link` command to link to redis container
- Your app can now see redis



# Solution

- Use the `--link` command to link to redis container
- Your app can now see redis



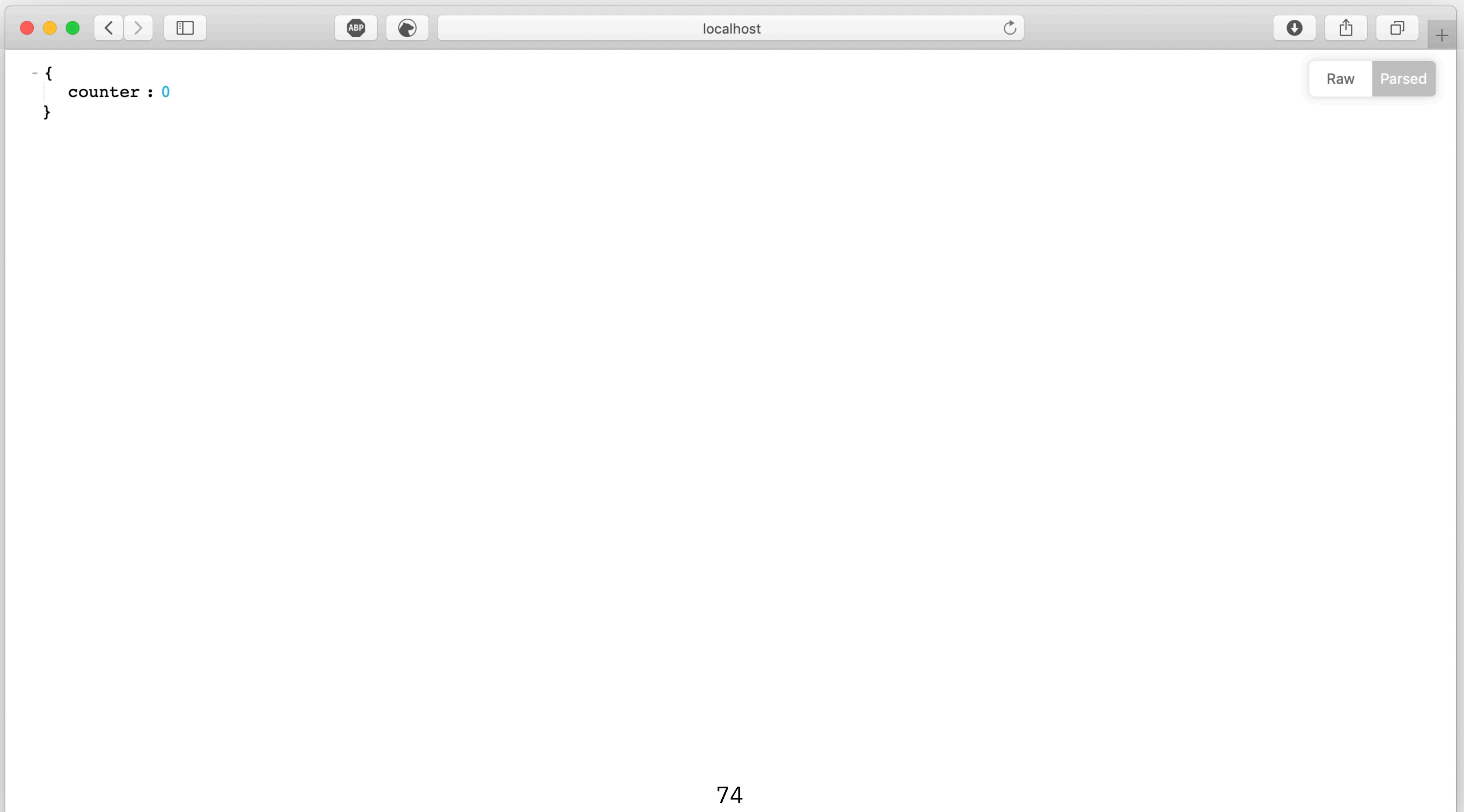
# Add environment variable to match link

- We can use the **--link** command to tell lab-docker where redis is:
- We use an environment variable to give our app the new **DATABASE\_URI**
  - `docker run --rm -p 5000:5000 \ --link redis \ -e DATABASE_URI="redis://redis:6379" \ hitcounter:1.0`

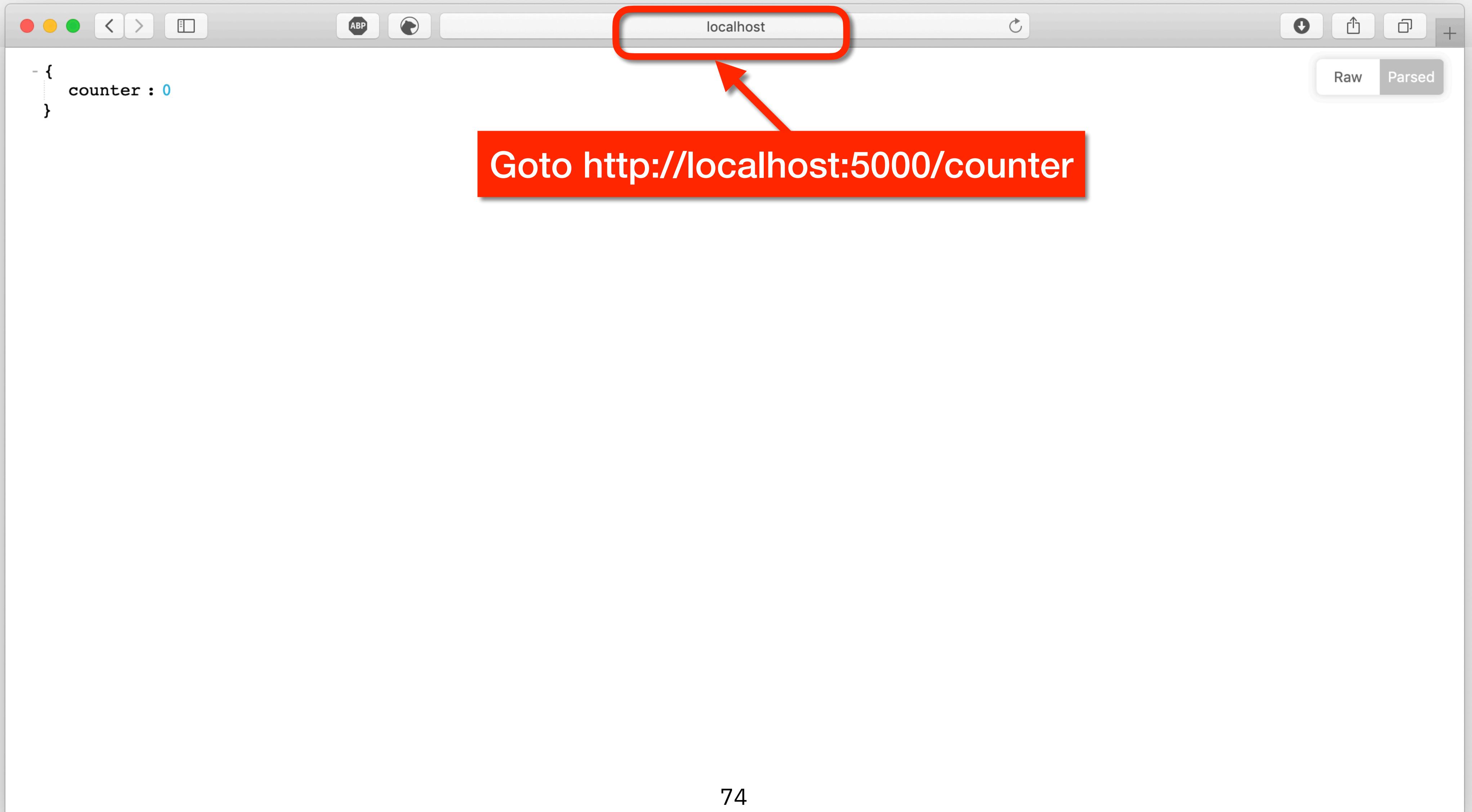
# Linking Containers

```
$ docker run --rm -p 5000:5000 --link redis -e DATABASE_URI="redis://redis:6379" hitcounter:1.0  
  
[2019-11-20 01:18:02 +0000] [1] [INFO] Starting gunicorn 20.0.0  
[2019-11-20 01:18:02 +0000] [1] [INFO] Listening at: http://0.0.0.0:5000 (1)  
[2019-11-20 01:18:02 +0000] [1] [INFO] Using worker: sync  
[2019-11-20 01:18:02 +0000] [9] [INFO] Booting worker with pid: 9  
[2019-11-20 01:18:02 +0000] [9] [INFO] Service initialized!
```

# Try Again



# Try Again



# Running Commands

- Start a container called "redis"
  - `docker run -d --name redis -p 6379:6379 redis:alpine`
- Call `redis-cli` with parameters
  - `docker exec -it redis redis-cli incr mycounter`

# Useful Docker Commands

- **Create a Dockerfile for your image**
  - vi Dockerfile
- **Build an image with:**
  - docker build -t <your\_image\_name> .
- **Run a container with:**
  - docker run -d --name <your\_container\_name> <your\_image\_name>
- **Show the containers logs with:**
  - docker logs <your\_container\_name>
- **Stop the container with:**
  - docker stop <your\_container\_name>
- **Remove the container with:**
  - docker rm <your\_container\_name>
- **Remove the image with:**
  - docker rmi <your\_image\_name>

# Useful Docker Flags

- **Linking** (`--link <other>`)
  - You can link containers so that they don't need to know each others ip addresses
- **Volumes** (`-v <host>:<cont>`)
  - You can mount volumes so that data can remain outside of the container and keep the containers stateless
- **Ports** (`-P or -p <host_port>:<cont_port>`)
  - You can map ports to expose them outside of the container `<host>:<container>`
- **Hostname** (`-h <name>`)
  - Assigns a hostname to the container
- **Daemon** (`-d`)
  - Run the container as a daemon process
- **Fault Tolerance** (`--always-restart`)
  - Tells Docker to restart the container if the main process ever fails
- **Environment Variables** (`-e <ENV_VAR>=xxx`)
  - Passes environment variables into a container at startup

# Summary

- You just created your first Docker containers
- You learned how to create our own Images
- You deployed your containers to Bluemix
- You can now check your Dockerfile into github so that others can create the same containers when working on your project.

