Introduction to Containers and Docker

Chase Hennion

August 14, 2019

Presentation Outline

- Morning
  - Introductions
  - Session 1: Introduction to Containers and Docker
  - Break
  - Session 2: A Technical Overview of Docker
  - Break
  - Session 3: Docker Best Practices, Design Patterns and Use Cases
- Lunch
- Afternoon
  - Hands-On Docker Workshop
    - Bring a laptop (Linux, Macbook, Windows 10) to work on during the workshop.
    - We’ll get Docker installed during the workshop.

Who am I?

- Graduated from OU in 2013 with a degree in Engineering Physics and a specialty in Computer Science.
- Worked as a professional systems and software engineer for about 5 years now.
- Worked with Docker for about 5 years.
- Use Docker almost daily in development.
    - Used for unit testing, functional testing, integration testing, CI/CD pipelines, software packaging and delivery
- Fallen into many Docker pitfalls over the years so I can help you avoid them!

Introduction to Containers and Docker

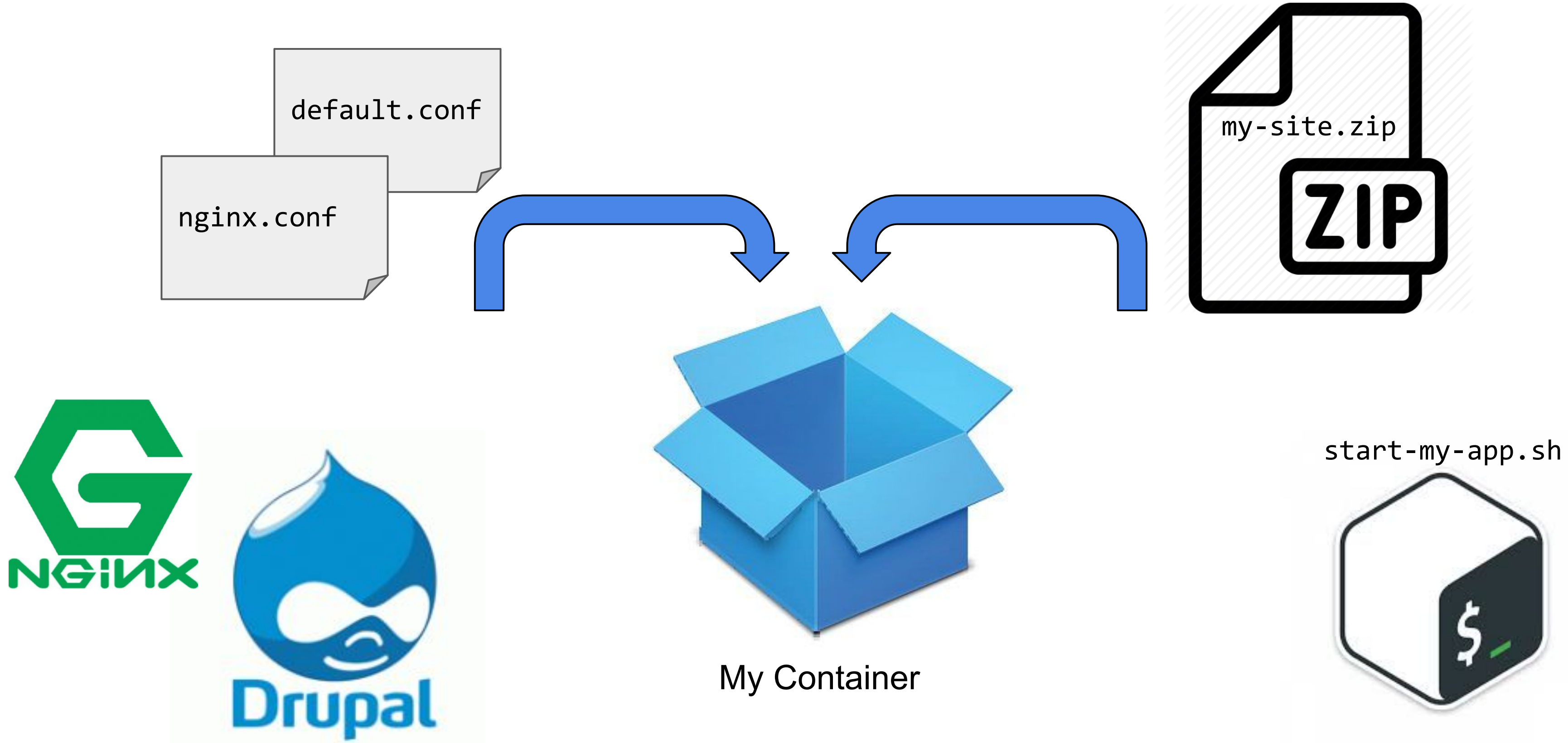
Outline

- What is a Container?
  - Brief History of Container Technology
  - Overview of Containers
  - Good and Bad Applications for Containers
  - Benefits of Containers
  - Containers vs Virtual Machines
- What is Docker?
  - Architecture
  - Main Components

What is a Container?

- A “box” that contains all of the packages, libraries, files, configuration etc. needed by an application or service.
- A container contains all of the stuff that your application or service needs to run in isolation.

What is a Container?



The Shipping Container Analogy

- We can’t talk about containers without covering the standard shipping container analogy.

The Shipping Container Analogy Cont.

- Imagine that you are responsible for packing a shipping container.
- As the packer, you are concerned about making sure everything that needs to be in the container is included and that everything is packed correctly and securely.
- Since you likely have to pack lots of containers, you create instructions for packing the container so you can pack each one the same way. This way you can hand off these instructions to other people so they can help pack containers too.

The Shipping Container Analogy Cont.

- Once you are done packing your container, you transport it to a shipping yard.
- At the shipping yard, the quartermaster takes the container from you and is then responsible for loading it on a ship safely.
- The quartermaster doesn’t care or perhaps even know about the specific contents of the container, they just know the pertinent details needed for loading the container (it’s weight, dimensions etc.)
- Since everything is packed up nicely in a standard container, it’s easy for the quartermaster to place the container wherever it fits on a ship.
- The container can be placed alongside containers that may have wildly different contents and the quartermaster doesn’t need to worry about it.

The Shipping Container Analogy Cont.

- Shipping containers are useful because they enforce uniformity. They all just look like boxes from the outside.
- Since everything is in a nice box, it’s easy to stack these boxes and move them around. You don’t need to worry that much about what’s in the box.
- **This is why containers are useful!**
- Whoever is building the container worries about the internal details.
- Once the container is packed, it just looks like a box.
- You can move that box around and stack wherever it fits without even caring about what’s in it.

The Bottom Line

- Putting stuff into boxes makes it easier to manage and move around.
  - You probably already knew that...
- Thinking about this beyond shipping containers, if you need more analogies:
  - Moving Boxes
  - Luggage
  - Grocery bags
- A container is just an implementation of this same idea but for software and applications.
- The “stuff” in your container is all of your packages, libraries, files, scripts that you need to run an application.
- Once all of your “stuff” is in the container, it’s easy to move that container around and run it on different computers.

What is a Container? - A More Technical Answer

- A container is essentially a set of one or more processes that are isolated on a host.
- Containers “contain” processes and applications in such a way that they are isolated from the underlying host.
- Virtualized at the OS level. Shares the underlying host’s kernel with the host and all other containers on the host.
- Once built, the container itself can be treated and managed as a single, atomic unit.

A Brief History of Containers

- Containers were not invented by Docker.
- Containers are not a new technology.
- The underlying technology still used by containers, `chroot`, was introduced on Version 7 Unix in 1979!
- `chroot` allows you to change the apparent root directory for a running process and its children, which is very useful for process isolation.
- FreeBSD took `chroot` and began using it for virtualization since around 2000, when the `jail` command was introduced.
- `jail` allowed system admins to partition a FreeBSD system into isolated mini-systems that all share the same kernel called jails.

A Brief History of Containers Cont.

- Solaris included the capacity to deploy “zones” in 2004.
  - “Zones” are very similar to FreeBSD jails.
  - Could only make zones of the same Solaris version as the host or one version older.
  - Forced the use needlessly weird zone-in-zone deployments.
- Cgroups were introduced to the Linux kernel in 2008 and would become a core technology that enables containers.
- The LXC (**L**inu**x** **C**ontainers) project was created shortly after the introduction of cgroups.
- These technologies tended to be difficult to use which hindered mainstream adoption of containers.

Why are containers popular now?

- If the underlying technology for containers has been around for awhile, why have they only really become popular in recent years?

- Docker introduced tools that made working with containers simple. No specialized knowledge was required to get started with containers.
- Docker enabled cross-platform container support. Previously container implementations were closely tied to specific operating systems.
- Docker coincided with the rise of the DevOps movement. Docker made it easier for developers and admins to work together by using containers.

Why should I care about containers?

- Containers are quicker to deploy and simpler to manage than VMs
    - Containers can be created from images in a few seconds with minimal overhead.
    - The creation and maintenance of container images is much simpler than maintaining VM images.
    - Container images are easily version controlled and hosted in image repositories.
    - Containers can be run on physical servers, VMs or via a variety of cloud services.
    - Container lifetime management is simpler than VMs. Containers are not designed to be manually patched and repaired by admins. If a container is broken, destroy the container, fix the image and redeploy.
    - Containers have a smaller attack surface than VMs and tend to be more secure out of the box. They do not contain and utilize administrative tools that could be exploited.
    - Containers allow you maximize your resource utilization. An single idle VM could run dozens of containers.

Why should I care about containers? Cont.

- Ease of Management
    - Containers let you package up applications in a tidy, easy to manage “box”.
    - Applications experts and developers are involved in the creation of your container images. Once the image is built, it’s simple to create as many container instances as you want.
    - Management of the containers created from the image becomes simple. Admins don’t need to worry about package and software dependencies for the application in the container. All of those dependencies are already in the container!

Why should I care about containers? Cont.

- Containers allow for you to package an application once and then run it almost anywhere.
  - All of the major cloud providers (AWS, GCP, Azure) have support for running containers in a variety of different ways.
  - These integrations make it easy to use application containerization as a means of enabling the migration of applications to the cloud.
  - Support from all major providers means your containers could feasibly be deployed across any major cloud platform and on-prim.
  - Running applications in containers greatly reduces the need for specialized infrastructure. The container itself has all of the prerequisite packages and configuration an app needs.
  - A container-centric approach drives uniformity in your infrastructure.

Why should I care about containers? Cont.

- Containers help to facilitate DevOps practices
  - The creation of container images involves your software developers as well as your system engineers and admins.
  - The image creation process keeps everyone on the same page as to how an application is to be deployed and used.
  - No more cases where software developers toss a poorly documented application over the wall to system engineers for them to deploy and manage.
  - Well defined images can be used to make rapidly reproducible development environments with minimal configuration drift.
  - Your development environment can look and behave the same as production because you are using the same images.

Why should I care about containers? Cont.

- Well designed containers scale easily and create loosely coupled application stacks.
    - A loosely coupled application stack allows for greater freedom in your application deployments.
    - Loosely coupled stacks also open up the opportunity for greater horizontal scaling, elastic scaling and advanced failover.
    - Loosely coupled stacks lend themselves easily to being deployed via a container orchestration tool like Kubernetes.

What applications work well in containers?

- Web applications
  - Web apps are a natural fit for containers.
  - Exposing web application ports for web app containers is very simple to do.
  - Simple to break up LAMP-type application stacks into multiple containers for each component of the stack and network them together. This makes it easy to scale your web app too!
- Open source software
  - Open source software in general is widely adopted and used in containers.
  - Oftentimes you can find pre-existing container images for a piece of open source software.
  - Many open source software projects come with Dockerfiles that allow for the building of container images for those projects.

What applications work well in containers? Cont.

- Command line tools
  - Containers are very useful for command line tools that may require very specific software requirements to run.
  - Some good examples: NMAP, Java JARs, Make
- Scalable applications
  - Well designed containers can easily run in isolation.
  - Following container design best practices will yield loosely coupled application stacks.
  - These loosely coupled stacks lend themselves easily to horizontal scaling.

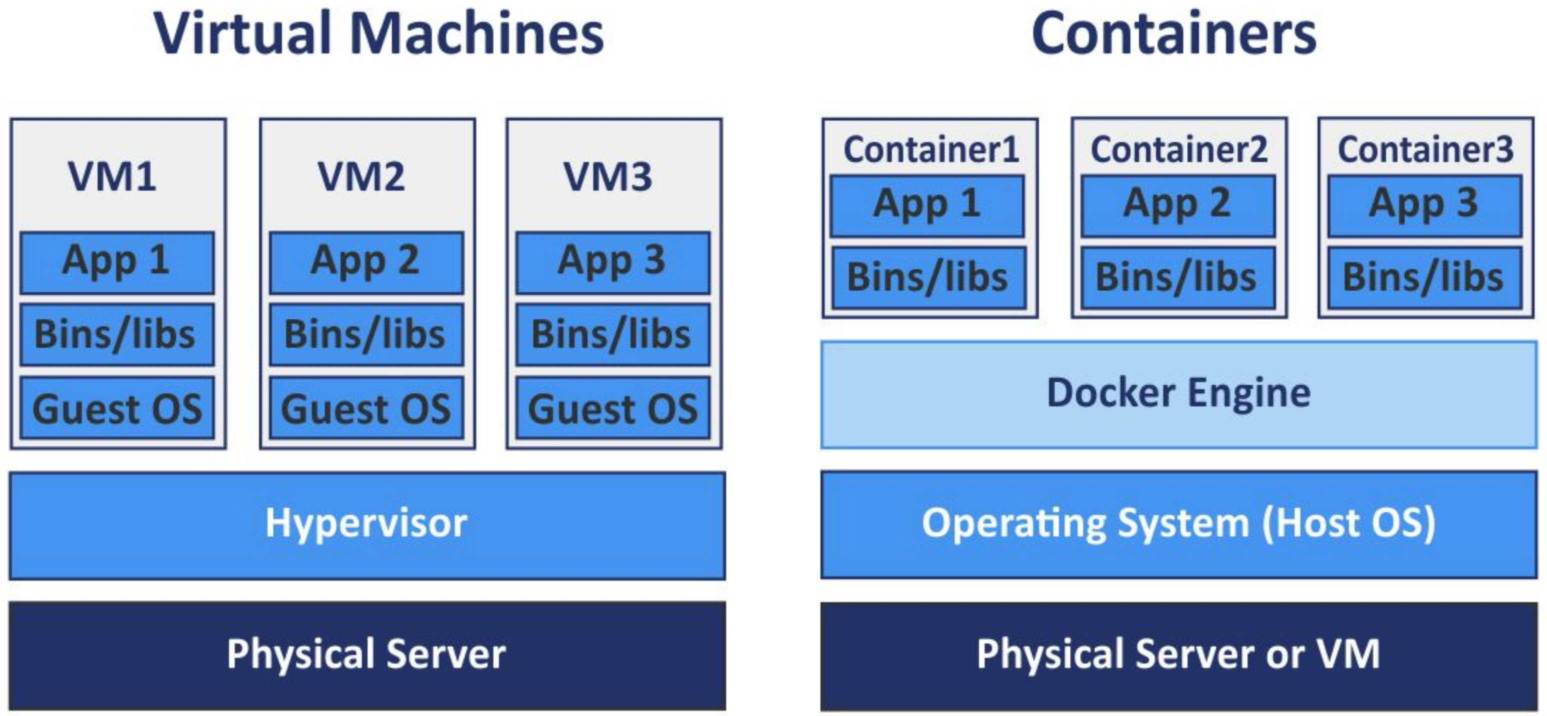
What applications work poorly in containers?

- Large, monolithic applications
  - Containers work best when they are focused on hosting and running a single application or process.
  - Monolithic applications almost always violate this paradigm and make it very difficult or impossible to break them up.
  - You *can* run these applications in containers, but they tend to be very large and slow. The resources savings vs running these applications on VMs tends to be negligible.
- “Enterprise” Applications
  - These tend to require special tools, license files etc. to install and start up, which makes them a pain to create container images for.
  - Your mileage may vary depending on the application. Some may work just fine in containers.
- Low-Level System Applications
  - Applications that access or modify the kernel don’t fit well in containers.
  - These applications can run in containers but keep in mind they will be altering the same kernel used by the host and and all other containers on the host.

What applications work poorly in containers? Cont.

- Large Databases
  - In general, a variety of databases run just fine in containers.
  - There is widespread community support for running a variety of database engines (MySQL, MariaDB, Postgres etc.) in containers.
  - These containers work great for modest databases and for development. Just make sure you follow DB best practices to ensure you don’t lose data.
  - For large, production databases, you will likely need a cluster of database containers and replication to keep the databases performant.
- Graphical, non-HTTP applications
  - Containers tend to be command line / terminal based in order to keep them small. Container images don’t tend to come with support for graphical interfaces.
  - You can mount in various display elements to get graphical applications to work but this also tends to reduce the portability of the image.

Containers vs Virtual Machines



Containers vs Virtual Machines Cont.

- Both VMs and Containers provide a means for providing an isolated environment for applications and processes.
- Virtual Machines
  - Require a Hypervisor for deployment and management.
  - VMs tend to be much larger than containers.
  - Completely isolated from the underlying host.
  - VMs tend to be used for more than a single application or process.
  - VMs tend to not be ephemeral. They are not readily destroyed and replaced.
- Containers
  - Kernel-Level virtualization
  - Tend to be much smaller and quicker to deploy than VMs.
  - Containers are designed to run a single application or process.
  - Containers are designed to be ephemeral. The are designed to be destroyed and replaced.

Containers vs Virtual Machines Cont.

- Containers and virtual machines should NOT be managed the same way.
- Containers should:
  - **Be Ephemeral.**
  - Be rapidly reproducible.
  - Only run one service.
  - Only expose ports necessary for the service it is running.
  - Not expose or require traditional management services such as SSH and Sudo.
  - Not require additional manual configuration via shell access following deployment.
- Admins should not login to containers to fix issues. Rather the container should be destroyed and re-deployed.
- Issues should be addressed during the image build and deployment process.

What is Docker?

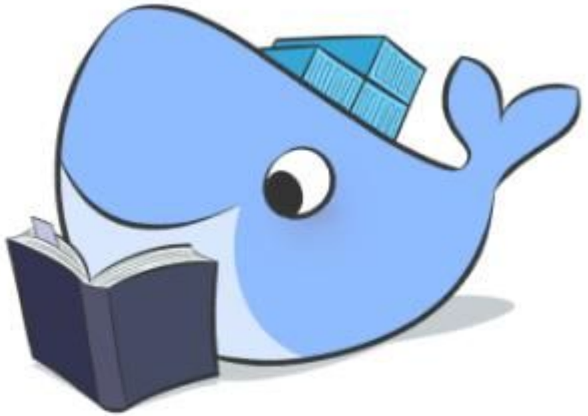
- Docker is a set of tools that aid in the creation, deployment and management of containers.
- Released as an open-source project in 2013
- Interface on top of a low-level container runtime.
        - containerd, runc, lxc etc.
- Consists of several components:
    - The Docker command line interface (CLI)
    - The Docker system daemon, dockerd
    - The Docker image registry
- Docker made working with containers easy.
    - Similar to how Vagrant made working with VMs easy.

What is Docker? Cont.

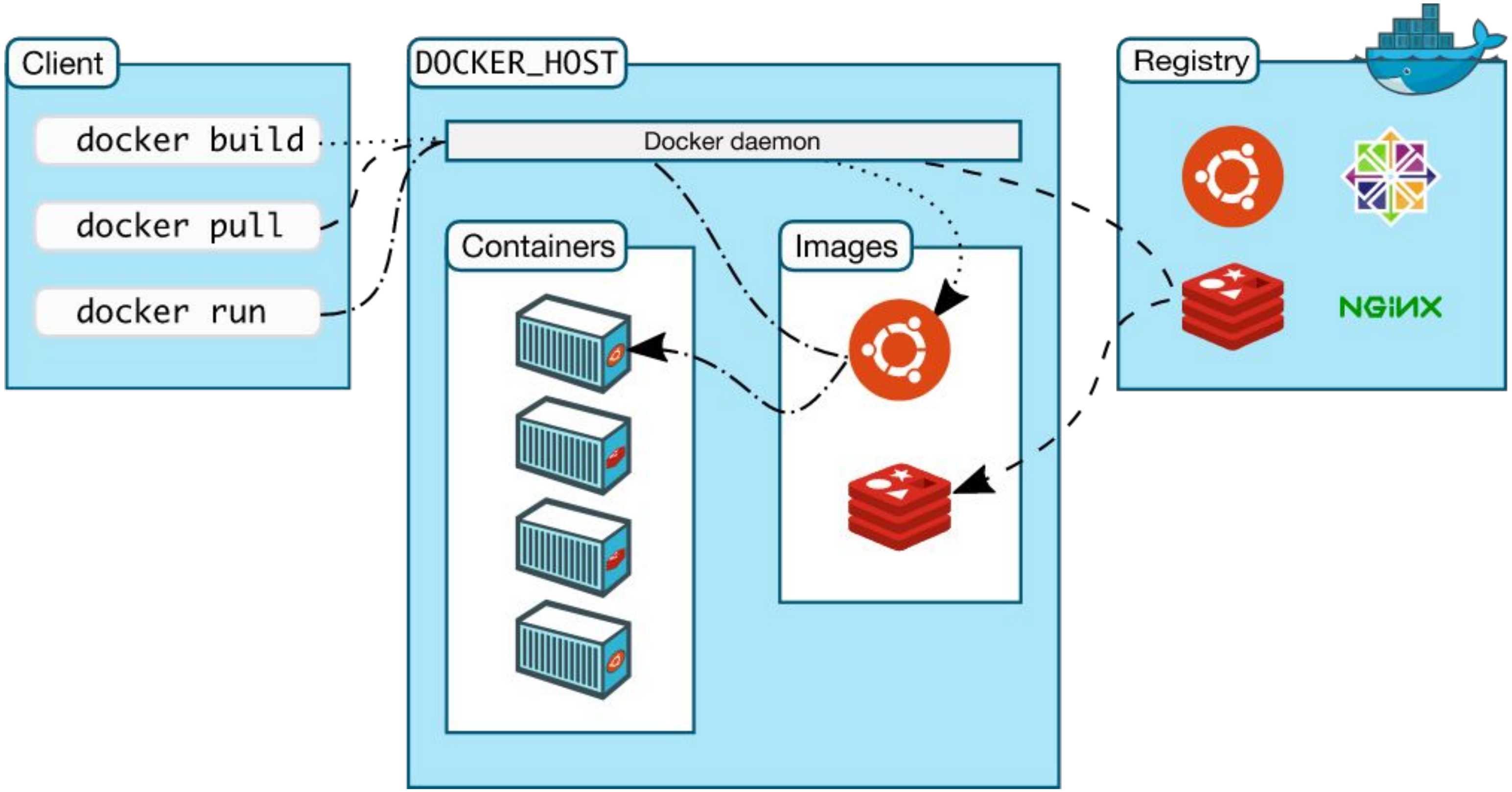
- Two Editions of Docker available
    - CE - Community Edition (Free, Open-source)
        - This presentation will focus exclusively on using the CE version of Docker.
    - EE - Enterprise Edition ($$$)
- Can be installed and used on Linux, Mac and Windows 10.

Basic Docker Terminology

- Container
  - An instance of a Docker image.
  - Ephemeral
- Image
  - A reusable template for a Docker container.
  - Often stored in a Docker registry for widespread use.
- Dockerfile
  - A file that defines a Docker image.
  - Can be built into a Docker image via the Docker CLI.
  - Often source controlled and used to periodically build images that are stored in a Docker registry.



Docker Architecture



Docker CLI

- The Docker CLI is the main means of interacting with the Docker daemon.
- Command line / terminal based. No GUI.
- The Docker CLI is used to:
  - Start and stop containers.
  - Build and manage container images.
  - Inspect and manage running and stopped containers.
  - And much more.
- The Docker API is RESTful. There are many freely available client libraries available in a variety of programming languages that can also be used to interact with the Docker API.

Docker Daemon

- The Docker daemon is responsible for scheduling and running containers.
- Exposes a RESTful API that the Docker CLI interacts with.
- Runs as a standard Unix service, typically via `systemd`.
- Typically bound to a Unix socket but can be externally exposed on a TCP port.
    - This is a potential security issue.
- On Mac and Windows, this service is managed for you via Docker Desktop.

Docker Images

- Docker images are used as the template for a containers.
- Similar to a VM Image (ova, vmdk, qcow2 etc.)
- Containers are instances of images.
- The image defines the underlying packages, libraries, configuration etc. for the image.
- Images are built from Dockerfiles.
- Commonly used images are typically stored in image registries so that they don’t have to be continually rebuilt.

Dockerfiles

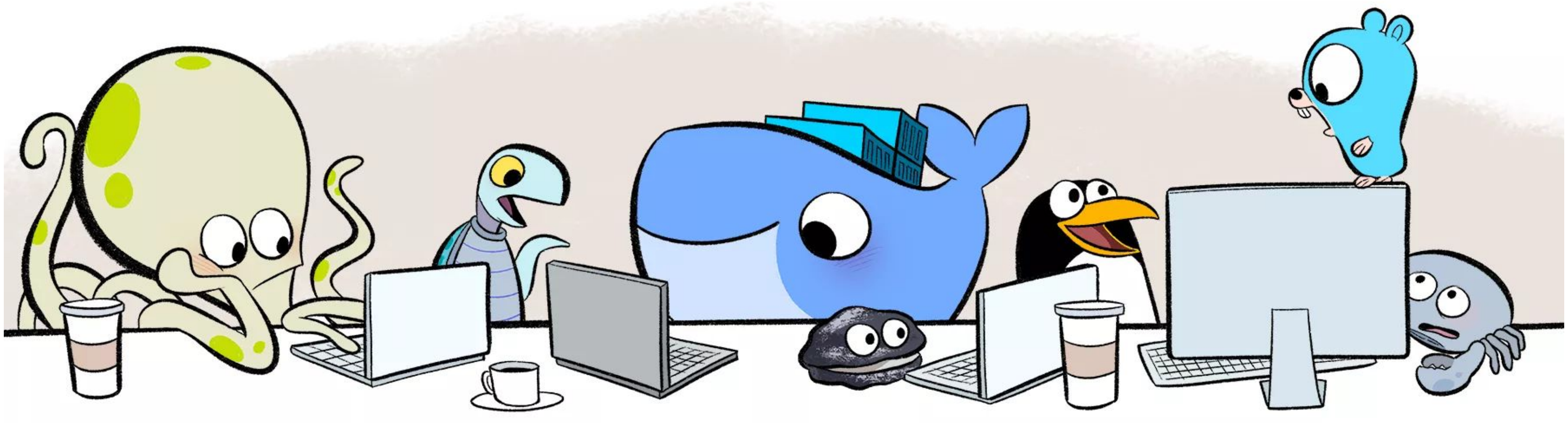
- Dockerfiles define how a Docker image is to be built.
- Dockerfiles are plaintext files written using the Dockerfile DSL.
- A Dockerfile consists of a variety of directives that are used to specify how a Docker image is built.
- This directives can include:
    - Defining the base image for the Dockerfile.
    - Specifying shell commands to run during the build of the image.
    - Specifying files and directories to copy into the image.
    - Exposing ports and volumes.
    - And much more.
- Dockerfiles tend to be versioned alongside the code/software that they are building images for and they provide a useful, self-documenting template for how to deploy an application or piece of software.

Docker Registry

- A Docker registry is used to store images so that they may be easily downloaded and used by others.
- Can easily run a Docker registry in a container itself for local image development and management.
- Dockerhub is a common public image registry managed by Docker itself.

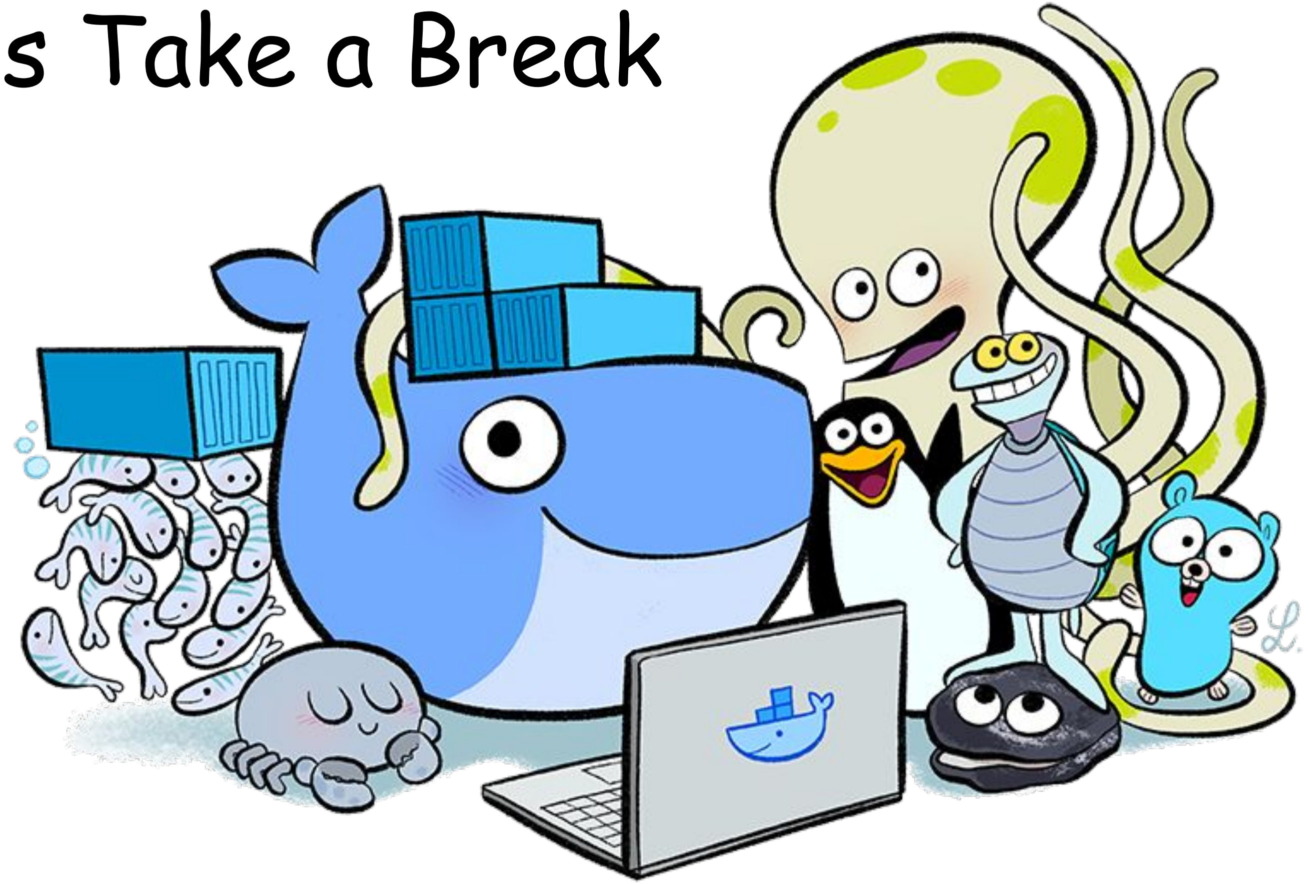
Dockerhub

- Public repository of images
- Free to use and contribute images
- Many images are maintained by core OS/software developers and companies
  - Canonical supports Ubuntu images.
  - CentOS Project supports CentOS images
- Docker verified images
  - Certified by Docker to adhere to exemplify practices.
- Always start with Dockerhub before creating a new image. It’s likely someone in the community has already contributed an image you can use or base your image off of.

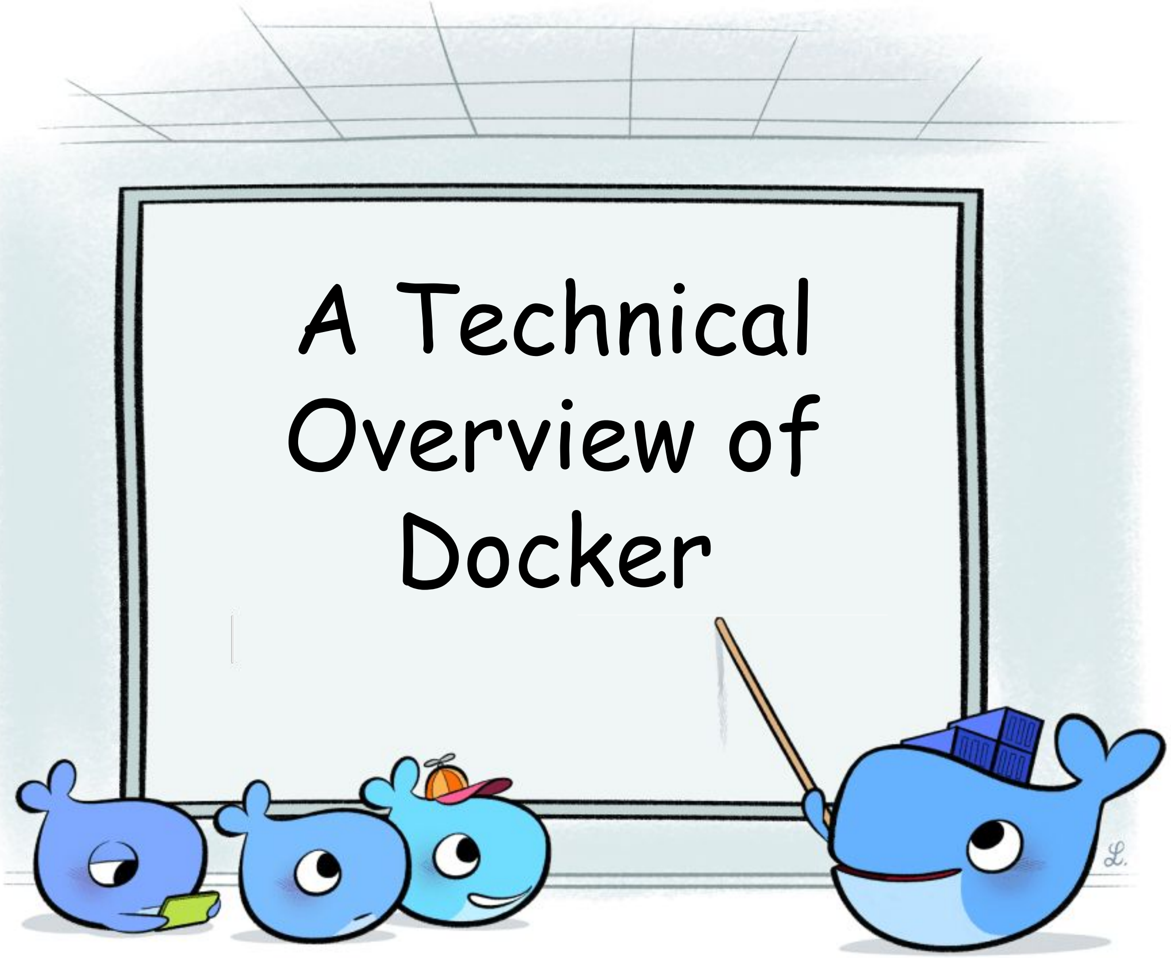


Questions?

Let's Take a Break



A Technical Overview of Docker



Outline

- Dockerfiles and Docker Images
- The Docker CLI
- Docker Volumes
- Docker Networking
- Docker-Compose
- Special Topics
    - Docker Images from Scratch
    - Privileged Containers
    - Windows Images and Containers
    - Docker Security

Dockerfiles

- Dockerfiles are text files that define how a Docker image is built.
- Dockerfiles contain a variety of directives that are used to copy in files, install packages and perform a variety of setup tasks during the build of the image.
- Plain text files written in a simple DSL.
- Let’s take a look at the various directives we can have in our Dockerfiles.

Common Dockerfile Directives

- `FROM`
  - Used to specify the base image for the Dockerfile.
  - Required for every Dockerfile.

```
# Use nginx:1.17 from dockerhub as the base image.
FROM nginx:1.17
```

- `LABEL`
  - Sets a metadata label.
  - Commonly used to specify the maintainer of the Dockerfile.
  - Show up in the container metadata when invoking `docker inspect`
  - A container can have as many labels as desired.

```
# Set the maintainer label
LABEL maintainer=”somebody@somewhere.com”
```

Common Dockerfile Directives Cont.

- `RUN`
  - Used to execute shell commands during the build of the Dockerfile.
  - Commands must be non-interactive.

```
# BAD
# Apt will ask for confirmation before install.
# This will cause the build to break.
RUN apt-get install nginx

# GOOD
# The -y flag will prevent apt from asking for confirmation
RUN apt-get install -y nginx
```

Common Dockerfile Directives Cont.

- COPY
  - Used to copy local files from the host into the Docker image.
  - Can also copy directories.
  - Can also change permissions and ownership of the file at runtime.
  - Files will be owned by root (UID: 0) in the image by default.

```
# Copy a file into the image.
COPY /path/on/host/myfile.txt /path/in/image/myfile.txt

# Copy a file into the image.
# Change the ownership of the file.
COPY --chown=www-data:www-data ./file.txt /file.txt

# Copy a directory into the image.
COPY ./my_dir /my_dir
```

Common Dockerfile Directives Cont.

- ADD
  - Used to copy in files from the host into the Docker image.
  - Can be used to retrieve files from URLs unlike COPY.
  - Cannot handle files at URLS that require authentication.
  - Files will be owned by root (UID: 0) in the image by default.
  - Will automatically decompress local archives (.zip, .tar, .tar.gz etc.)
    - Does not apply to archives fetched from URLs.

```
# Add a local file into the image.
ADD --chown=www-data:www-data ./myfile.txt /myfile.txt

# Add a file from a URL
# Change the ownership of the file.
COPY --chown=www-data:www-data ./file.txt /file.txt
```

COPY vs ADD

- ADD does more than COPY. Why not just always use ADD?

- ADD automatic archive decompression may not always be desired.
- If copying many files from URLs, having many ADD directives will add additional layers and potential bloat to your image.
  - If copying many files from URLs, a better approach would be to use cURL or wget in a script that is executed during the build of your image.
- Files copied from URLs may only need to be present temporarily
- Best Practice: Use COPY when the extra functionality of ADD is not required.

Common Dockerfile Directives

- `WORKDIR`
  - Sets the working directory within the Docker image.
  - Will be created if it doesn’t exist
  - COPY, ADD, RUN directives will execute relative to the specified WORKDIR.

```
# Set /var/www/html as the current working directory.
WORKDIR /var/www/html

# Will copy myfile.txt into /var/www/html since
# /var/www/html is the working directory in the image.
COPY ./myfile.txt ./myfile.txt

# Will execute /var/www/html/myscript.sh
RUN ./myscript.sh
```

Common Dockerfile Directives Cont.

- `USER`
  - Specifies the user to execute commands as within the container.
  - By default, command will be executed as root.

```
# Run commands as the www-data user.
USER www-data
```

- `ENV`
  - Used to specify an environment variable.
  - Environment variables can be passed in at runtime via the -e option.

```
# Set the PATH environment variable.
# <var>=<value> format.
ENV PATH=/bin:/sbin

# <var> <value> format.
ENV PATH /bin:/sbin
```

Common Dockerfile Directives Cont.

- `EXPOSE`
  - Used to expose a port.
  - By default, exposed ports will only be accessible via the Docker container’s private network interface.
  - Not required. Ports can be opened at runtime via the -p option.

```
# Expose port 80 for containers based on this image.
EXPOSE 80
```

Common Dockerfile Directives Cont.

- ENTRYPOINT
  - Specifies the entrypoint script or command.
  - A Dockerfile can only have one ENTRYPOINT directive. If more than one is provided, the last ENTRYPOINT directive will be used.
  - Can be overwritten at runtime via the --entrypoint flag.

```
# Exec Command Format ← Preferred Format
ENTRYPOINT [ “ls” ]


# Shell Command Format
ENTRYPOINT ls
```

Common Dockerfile Directives Cont.

- `CMD`
    - Specifies the default arguments to the entrypoint script.
    - A Dockerfile can only have one `CMD` directive. If more than one is provided, the last `CMD` directive will be used.
    - These arguments can be overwritten when a container is run via `docker run`
    - Can also include the entrypoint script itself. In this case, the Dockerfile should not include an `ENTRYPOINT` directive.

```
# Exec Command Format ← Preferred Format
CMD [ “-al”, “/etc ]

# Exec Command Format, including entrypoint script “ls”.
CMD [ “ls”, “-al”, “/etc” ]

# Shell Command Format
CMD ls -al /etc
```

CMD vs ENTRYPOINT

- The `CMD` and `ENTRYPOINT` directives specify the script/command that containers created from the image will run by default.
- It is possible to use `CMD` and `ENTRYPOINT` together or just `CMD` on it’s own.

```
# When a container built on this image is run without any
additional arguments, it will execute the command ls -al /etc

# Set the entrypoint binary/script.
ENTRYPOINT [ “ls”, “-al” ]

# Specify default arguments to the entrypoint script
CMD [ “/etc” ]
```

Specifying CMD and ENTRYPOINT at runtime

- It is possible to override the ENTRYPOINT and CMD directives at container runtime.
- Suppose we built the image my-image:latest from the previous Dockerfile examples.

```
# Run with default entrypoint and cmd. Will execute ‘ls -al /etc’
$ docker run my-image:latest

# Override the default cmd. Will execute ‘ls -al /var’
$ docker run my-image:latest /var

# Override the entrypoint and cmd. Will execute ‘cat /etc/hosts’
$ docker run --entrypoint=/bin/cat my-image:latest /etc/hosts
```

CMD vs ENTRYPOINT Best Practices

- Use an `ENTRYPOINT` and (optionally) a `CMD` when your image is providing general access to a script or binary.
  - This will allow users to easily provide customized arguments to the script or binary entrypoint and allows for easy image reuse.
  - Provide a `CMD` along with the `ENTRYPOINT` if there is a common default set of options/parameters that the binary or script should use.
- Just use a CMD when your image is intended to only run a specific command without any input from the user.

Docker Build Context

- When building a Docker image from a Dockerfile, the Dockerfile can only access files and directories at or below the directory where the Dockerfile resides.
- This accessible directory tree called the “build context” of the Docker image.
- When Docker builds an image, it recursively sends the whole build context to the Docker daemon.
- Best Practice: Start with a directory with just your Dockerfile. Only add files that are required for the build to the directory.
- Don’t use / as your build context!

Docker Image Layers

- A Docker image consists of a set of read-only layers.
- Every Dockerfile directive will generate a new layer in the image.
- Can think of layers as the history of an image or the snapshot of the image at a certain stage of the image build.
    - Very similar to the Git commit history for a file.
- Use `docker history` to view all of the layers within an image.
- When a container is run, Docker places a writeable layer on top of the final layer of the image which allows the container to make changes to the filesystem.
- Try to keep the number of layers in an image to a minimum otherwise you can easily bloat the size of your final image.

Docker Images, Layers and Size Example

- Consider the following Dockerfiles. They do the same thing.

```
# Builds unoptimized:latest
FROM ubuntu:18.04

RUN apt-get update

RUN apt-get install -y wget

RUN wget
https://www.websecurity.symantec.com/conte
nt/dam/websitesecurity/support/digicert/sy
mantec/root/DigiCert_Global_Root_CA.pem

RUN apt-get remove -y wget

RUN apt-get clean

RUN rm -rf /var/lib/apt/lists*
```

```
# Builds optimized:latest
FROM ubuntu:18.04

RUN apt-get update \
  && apt-get install -y wget \
  && wget
https://www.websecurity.symantec.com/conte
nt/dam/websitesecurity/support/digicert/sy
mantec/root/DigiCert_Global_Root_CA.pem \
  && apt-get remove -y wget \
  && apt-get clean \
  && rm -rf /var/lib/apt/lists*
```

Docker Images, Layers and Size Example

```
$ docker images

REPOSITORY                    TAG               IMAGE ID          CREATED            SIZE
optimized                     latest            0a3429002c5a      17 minutes ago     71MB
unoptimized                   latest            d84e402aecf2      19 minutes ago     99.4MB
```

- There’s almost a 30% difference in size between the unoptimized and optimized images even though the do the same thing!
- Why is that?
- Let’s look at the layers of the image.

Docker Images, Layers and Size Example

- The optimized image only has one layer added to the image (the other layers are from the base Ubuntu image).
- Since we clean up the apt cache and purge wget when we are done with it we are able to keep the single layer small.

```
$ docker history optimized:latest

IMAGE            CREATED          CREATED BY                                     SIZE
0a3429002c5a     20 minutes ago   /bin/sh -c apt-get update   && apt-get insta…   6.85MB
3556258649b2     10 days ago      /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<missing>        10 days ago      /bin/sh -c mkdir -p /run/systemd && echo 'do…   7B
<missing>        10 days ago      /bin/sh -c set -xe   && echo '#!/bin/sh' > /…   745B
<missing>        10 days ago      /bin/sh -c [ -z "$(apt-get indextargets)" ]     987kB
<missing>        10 days ago      /bin/sh -c #(nop) ADD file:3ddd02d976792b6c6…   63.2MB
```

Docker Images, Layers and Size Example

- The unoptimized image has several more layers.
- Note that the cleanup layers have no size. They cannot change the size of a previous layer.

```
$ docker history unoptimized:latest

IMAGE            CREATED          CREATED BY                                     SIZE
d84e402aecf2     22 minutes ago   /bin/sh -c rm -rf /var/lib/apt/lists*          0B
77cbc551ce8f     22 minutes ago   /bin/sh -c apt-get clean                       0B
a4a321847e51     22 minutes ago   /bin/sh -c apt-get remove -y wget              926kB
fcab06e2a544     22 minutes ago   /bin/sh -c wget https://www.websecurity.syma…  1.36kB
2f84533140fa     22 minutes ago   /bin/sh -c apt-get install -y wget             7.42MB
dc208be85dc6     22 minutes ago   /bin/sh -c apt-get update                      26.9MB
3556258649b2     10 days ago      /bin/sh -c #(nop)  CMD ["/bin/bash"]           0B
<missing>        10 days ago      /bin/sh -c mkdir -p /run/systemd && echo 'do…  7B
<missing>        10 days ago      /bin/sh -c set -xe   && echo '#!/bin/sh' > /…  745B
<missing>        10 days ago      /bin/sh -c [ -z "$(apt-get indextargets)" ]    987kB
<missing>        10 days ago      /bin/sh -c #(nop) ADD file:3ddd02d976792b6c6…  63.2MB
```

Do Image Layers Always Add to the Size?

```
# Builds runtest_combined:latest
FROM ubuntu:18.04

RUN date && hostname -f && ls -al /etc
```

```
# Builds runtest_separate:latest
FROM ubuntu:18.04

RUN date
RUN hostname -f
RUN ls -al /etc
```

```
$ docker images

REPOSITORY                 TAG              IMAGE ID         CREATED           SIZE
runtest_combined           latest           e143088a83b4     22 minutes ago    64.2MB
runtest_separate           latest           29f27ad7f10e     21 minutes ago    64.2MB
```

- These images are the same size!
- Why is that?
- Let's look at the layers again.

Docker Images, Layers and Size Example

```
$ docker history runtest_combined:latest
IMAGE               CREATED             CREATED BY                                      SIZE
e143088a83b4        24 minutes ago      /bin/sh -c date && hostname -f && ls -al /etc    0B
3556258649b2        10 days ago         /bin/sh -c #(nop)  CMD ["/bin/bash"]             0B
<missing>           10 days ago         /bin/sh -c mkdir -p /run/systemd && echo 'do…   7B
<missing>           10 days ago         /bin/sh -c set -xe   && echo '#!/bin/sh' > /…   745B
<missing>           10 days ago         /bin/sh -c [ -z "$(apt-get indextargets)" ]      987kB
<missing>           10 days ago         /bin/sh -c #(nop) ADD file:3ddd02d976792b6c6…   63.2MB

docker history runtest_separate:latest
IMAGE               CREATED             CREATED BY                                      SIZE
      COMMENT
29f27ad7f10e        25 minutes ago      /bin/sh -c ls -al /etc                           0B
aef9eb231a50        25 minutes ago      /bin/sh -c hostname -f                           0B
b809621e0c9f        25 minutes ago      /bin/sh -c date                                  0B
3556258649b2        10 days ago         /bin/sh -c #(nop)  CMD ["/bin/bash"]             0B
<missing>           10 days ago         /bin/sh -c mkdir -p /run/systemd && echo 'do…   7B
<missing>           10 days ago         /bin/sh -c set -xe   && echo '#!/bin/sh' > /…   745B
<missing>           10 days ago         /bin/sh -c [ -z "$(apt-get indextargets)" ]      987kB
<missing>           10 days ago         /bin/sh -c #(nop) ADD file:3ddd02d976792b6c6…   63.2MB
```

Docker Images, Layers and Size Example

- Those run commands did not contribute any size to the image.
- Why is that?
- These commands do not cause any change to the filesystem and thus do not contribute anything to the size of the final image.

But wait a minute...

- `apt-get clean` and `rm -rf /var/lib/apt/lists*` definitely affect the filesystem.
- Why didn't they affect the size?

```
$ docker history unoptimized:latest

IMAGE           CREATED          CREATED BY                                       SIZE
d84e402aecf2    22 minutes ago   /bin/sh -c rm -rf /var/lib/apt/lists*            0B
77cbc551ce8f    22 minutes ago   /bin/sh -c apt-get clean                         0B
a4a321847e51    22 minutes ago   /bin/sh -c apt-get remove -y wget                926kB
fcab06e2a544    22 minutes ago   /bin/sh -c wget https://www.websecurity.syma…   1.36kB
2f84533140fa    22 minutes ago   /bin/sh -c apt-get install -y wget               7.42MB
dc208be85dc6    22 minutes ago   /bin/sh -c apt-get update                        26.9MB
3556258649b2    10 days ago      /bin/sh -c #(nop)  CMD ["/bin/bash"]             0B
<missing>       10 days ago      /bin/sh -c mkdir -p /run/systemd && echo 'do…   7B
<missing>       10 days ago      /bin/sh -c set -xe   && echo '#!/bin/sh' > /…   745B
<missing>       10 days ago      /bin/sh -c [ -z "$(apt-get indextargets)" ]      987kB
<missing>       10 days ago      /bin/sh -c #(nop) ADD file:3ddd02d976792b6c6…   63.2MB
```

What’s going on?

- Even though those commands clean up files in the final image they are still present in previous image layers.
- Docker essentially masks the files removed by those commands in the layers where they were “removed”.
- The layers are the cumulative history of the image. Even if a newer layer changes something that was done in a previous layer, the state of the image is still the same in the previous layer.
    - Like a Git history. The history of a file is always stored and available even if the file is removed in a later commit.
- The `docker build` command has an experimental option, `--squash`, which will squash all of your image layers into one image. We can use this option to compress the layers in unoptimized:latest.

--squash to the rescue!

- If we build the unoptimized image with the --squash flag we see that the final image is the same size as the optimized image!
- If you are familiar with Git, this behavior is analogous to squashing Git commits.
- Keep in mind that --squash is still an experimental feature.

```
$ docker images

REPOSITORY                 TAG             IMAGE ID        CREATED          SIZE
unoptimized_squashed       latest          37a440a06f35    3 seconds ago    71MB
optimized                  latest          0a3429002c5a    17 minutes ago   71MB
unoptimized                latest          d84e402aecf2    19 minutes ago   99.4MB
```

What does --squash do to the image layers?

- It merges them into one layer!

```
$ docker history unoptimized_squashed:latest

IMAGE               CREATED             CREATED BY                                      SIZE
COMMENT
37a440a06f35        20 seconds ago                                                      6.85MB
merge sha256:d84e402aecf2a34f2b20bce5076154bd239f6a92a070630a2979ccfe43435684 to
sha256:3556258649b2ef23a41812be17377d32f568ed9f45150a26466d2ea26d926c32
<missing>           About an hour ago   /bin/sh -c rm -rf /var/lib/apt/lists*           0B
<missing>           About an hour ago   /bin/sh -c apt-get clean                        0B
<missing>           About an hour ago   /bin/sh -c apt-get remove -y wget               0B
<missing>           About an hour ago   /bin/sh -c wget https://www.websecurity.syma…   0B
<missing>           About an hour ago   /bin/sh -c apt-get install -y wget              0B
<missing>           About an hour ago   /bin/sh -c apt-get update                       0B
<missing>           10 days ago         /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<missing>           10 days ago         /bin/sh -c mkdir -p /run/systemd && echo 'do…   7B
<missing>           10 days ago         /bin/sh -c set -xe   && echo '#!/bin/sh' > /…   745B
<missing>           10 days ago         /bin/sh -c [ -z "$(apt-get indextargets)" ]     987kB
<missing>           10 days ago         /bin/sh -c #(nop) ADD file:3ddd02d976792b6c6…   63.2MB
```

Docker Image Layer Caching

- Docker will cache the layers of an image during an image build.
- If the directive associated with the layer has not been updated in the Dockerfile, Docker will used a cached copy of the layer to speed up the build process.
- When a directive associated with a layer is updated, the cache will be invalidated for that layer and every subsequent layer in the image.
- Can force Docker to rebuild all layers of an image via the `--no-cache` option.
- Be wary of cached layers for things such as package installations etc. as Docker will not know if a new package installation would retrieve a newer version of a package.

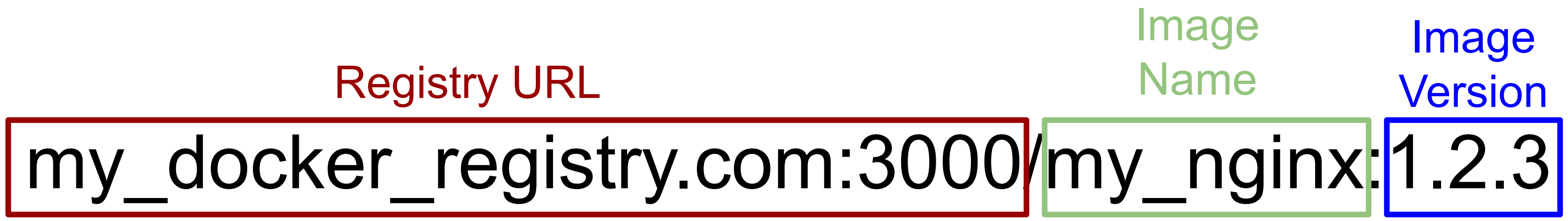
Why not always build with --squash?

- Why don’t we always build images with the `--squash` flag since it makes our images smaller?

- Similar reason to why we don’t always squash Git commits. It can be useful having the layer history available, especially during development.
- Can’t take advantage of layer caching if the layers are all squashed.
- Best Practice: Build normally while developing an image. Push up a squashed image to your image registry once it has been completed.

Docker Image Tagging

- When you build an image from a Dockerfile, you should include a descriptive tag.
- A Docker image tag contains several important pieces of information about the image:
  - (Optional) The Registry where the image is located.
    - If omitted, Docker will assume the image is available on Dockerhub.
  - The name of the image.
  - The version of the image.
- Can also tag existing images at any time via `docker tag`



```
my_docker_registry.com:3000/my_nginx:1.2.3
```

Using the Docker CLI

- Once a Dockerfile has been completed, it can then be built into an image and used to create containers via the Docker CLI.
- The Docker CLI is essentially a REST client that interacts with the Docker daemon RESTful API.
- The Docker CLI provides a simple interface for building and running containers, inspecting containers and interacting with the Docker daemon in a variety of ways.
- Let’s take a look at several of the more common Docker CLI commands.

Common Docker Commands

- `docker run`
  - Creates and starts a Docker container based on a given container Image

```
$ docker run my-image:latest
```

- `docker ps`
  - Lists running and stopped/exited Docker containers.

```
# List only running containers.
$ docker ps

# List running and stopped containers.
$ docker ps -a
```

Common Docker Commands Cont.

- `docker pull`
  - Pulls a Docker image from a image repository (usually Dockerhub)

```
# Pull an nginx image from Dockerhub
$ docker pull nginx:latest

# Pull my-image:1.2.3 from my-registry
$ docker pull my-registry:3000/my-image:1.2.3.
```

- `docker build`
  - Builds a Docker image from a Dockerfile

```
# Build an image, my-image:latest, from a file called Dockerfile in /my-path
$ docker build -t my-image:latest /my-path

# Build an image from a Dockerfile not literally called Dockerfile
$ docker build -t my-image:latest -f /my-path/My-Dockerfile /my-path
```

Common Docker Commands Cont.

- `docker tag`
  - Creates a new image tag that refers to an existing image.
  - Tags are used to determine the registry associated with a given image.

```
# Tag the nginx image with a new name.
$ docker tag nginx:latest my-sweet-nginx:latest

# Tag my-image:1.2.3 with a new registry.
$ docker tag my-registry:3000/my-image:1.2.3 my-other-registry/my-image:1.2.3
```

- `docker push`
  - Pushes a Docker image to a Docker image registry.

```
# Push newly tagged my-image to my-other-registry
$ docker push my-other-registry/my-image:1.2.3
```

Common Docker Commands Cont.

- `docker logs`
  - Gets the log messages printed to STDOUT from a given container.

```
# Get logs from container with ID fbfed2779717
$ docker logs fbfed2779717
```

- `docker cp`
  - Copies file(s) out of a container.

```
# Copy my_file.txt out of container with ID fbfed2779717
$ docker cp fbfed2779717:path/in/container/my_file.txt /path/on/host/my_file.txt
```

Common Docker Commands Cont.

- `docker exec`
  - Executes a given command within a Docker container.

```
# Execute ls -al /etc/ within container with ID fbfed2779717
$ docker exec fbfed2779717 ls -al /etc

# Start an interactive bash shell session.
$ docker exec -it fbfed2779717 /bin/bash

# Execute a command and redirect the output to a file on the host.
$ docker exec -i fbfed2779717 /usr/bin/mysqldump --user=user --password=password
mydb > my_db.sql
```

Common Docker Commands Cont.

- `docker inspect`
  - Returns low-level information about a given Docker container.
  - Can parse returned data with Go templates via the -f flag.

```
# Inspect container with ID fbfed2779717
$ docker inspect fbfed2779717

# Get the name of the container with ID fbfed2779717
$ docker inspect -f '{{.Name}}' fbfed2779717

# Get the IP address of the container with ID fbfed2779717
$ docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
fbfed2779717
```

Docker Volumes

- Docker volumes are used to persist the data generated by a Docker container.
- Volumes can be shared between containers.
- Can create and manage volumes via the `docker volume` CLI.
- There a variety of Docker volume plugin and drivers that allow for volumes to be backended by a variety of file system implementations.
  - GlusterFS, GCE persistent disks, AWS EFS etc.
- There are three types of Docker volumes
  - `volume`
  - `bind`
  - `tmpfs`

`Volume` Type Volumes

- The standard volume type for Docker.
- A volume type volume is essentially a directory tree allocated and managed by Docker on the host filesystem that a container can write to. When the container is stopped, the data written to the volume is persisted.
- The preferred way of handling persistent volumes in Docker.
- Can be backended by a variety of file system and block storage systems via plugins.
- Managed by Docker and supported across operating systems.

Bind-Mount Type Volumes

- A Bind-Mount allows you to mount a directory or file from the host filesystem into a container.
- Use a Bind-Mount when you want to provide data to a container and allow a container to optionally modify/update the data.
- More limited than volumes.
- Bind-Mounts require a full system path to the file or directory that is to be mounted.
- Where possible, try to use volumes rather than Bind-Mounts to allow easier management and cross platform support.

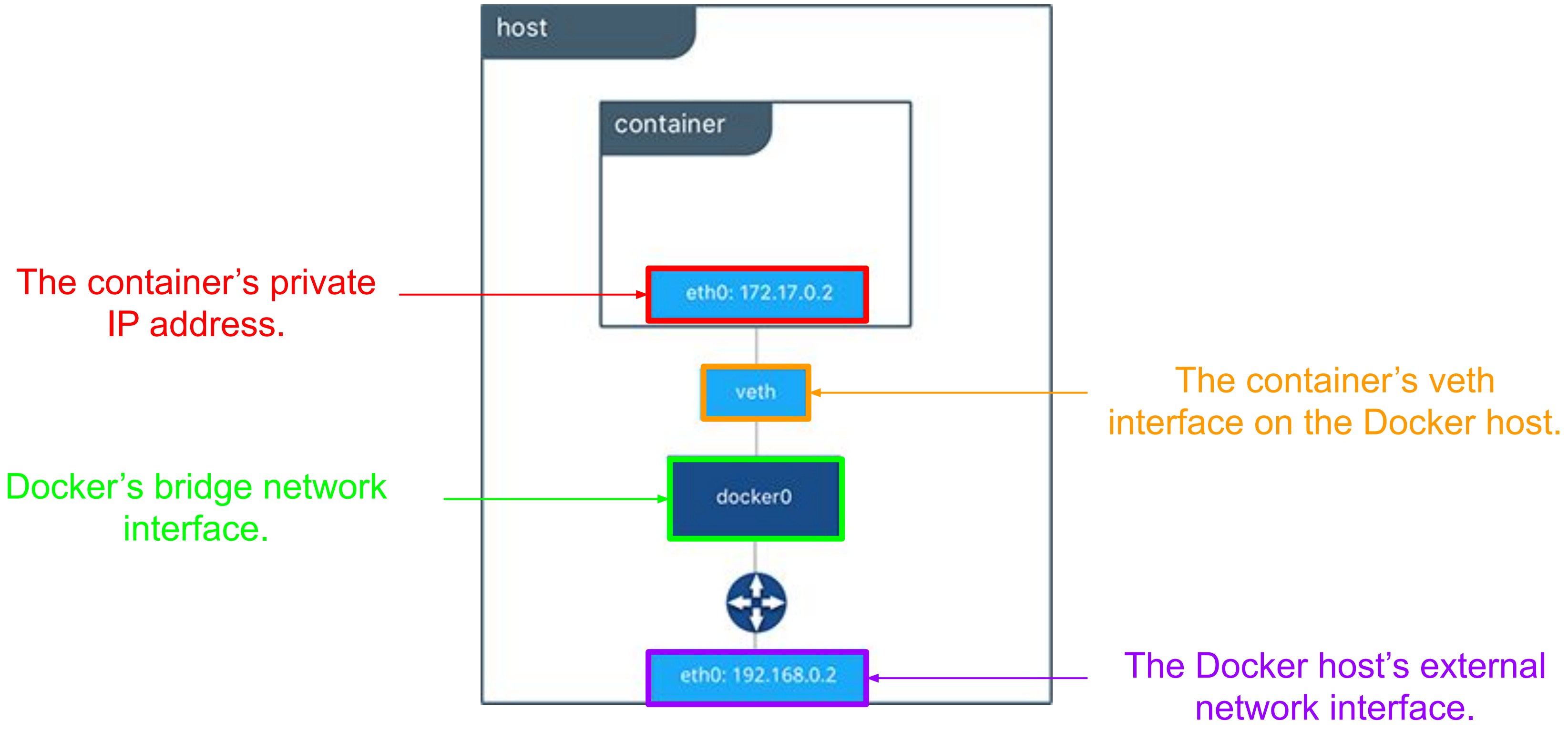
`Tmpfs` Type Volumes

- A tmpfs type volume is a volume whose contents are stored entirely in memory.
- Good for I/O intensive and read-only applications.
    - Memory tends to be more limited than disk space.
    - Be cautious of letting a container consume too much memory!
- Files written to a tmpfs mount cannot be persisted. Once the container stops, the tmpfs volume is removed.
- Tmpfs volumes cannot be shared between containers.

Docker Network Architecture

- Docker creates and manages the network architecture required for your containers to connect to each other and the external internet.
- When the Docker daemon starts, it will automatically create an initial private network and a bridged network interface (`docker0`) on your host.
- The private network is used for IP address allocation for containers.
- The bridged network interface allows containers to reach networks external to the Docker host via the public network interface of the host.
- When a container is created, it will receive an IP address on the Docker private network by default and will appear as a veth network interface on the Docker host.
- This IP address will only be reachable from the Docker host and from containers on the same private network.

Docker Networking: A Simple Example


host
container
eth0: 172.17.0.2
veth
docker0
eth0: 192.168.0.2
The container's private IP address.
The container's veth interface on the Docker host.
Docker's bridge network interface.
The Docker host's external network interface.

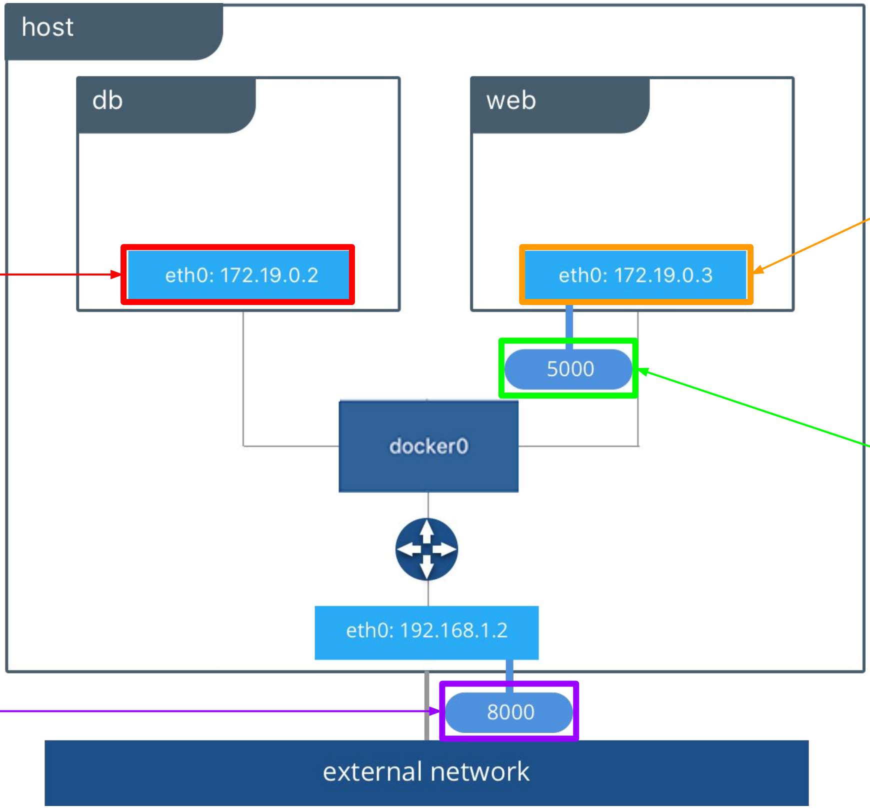
Docker Networking: Port Mapping

- When a container is run, it is possible to map a port and a network interface from the Docker host to a port within the container.
- This allows an application running inside a container to be reached via the IP address/FQDN of the Docker host on the exposed port.
- This is done via the `--publish, -p` option for `docker run.`

```
# Map port 8080 to port 80 in the nginx container only on the loopback interface.
$ docker run -p 127.0.0.1:8080:80 nginx:1.17

# Check to see that the exposed port is in the LISTEN state.
$ netstat -tulpn
Proto Recv-Q  Send-Q  Local Address  Foreign Address  State   PID/Program name
tcp     0       0    127.0.0.1:8080    0.0.0.0:*      LISTEN  19112/docker-proxy
```

Docker Networking: Port Mapping



No port is mapped for the DB container. It is only reachable via it's private IP.

The web container has a port mapped. It is reachable via it's private IP and via the eth0 interface on the Docker host via port 8000.

The exposed port for the web container

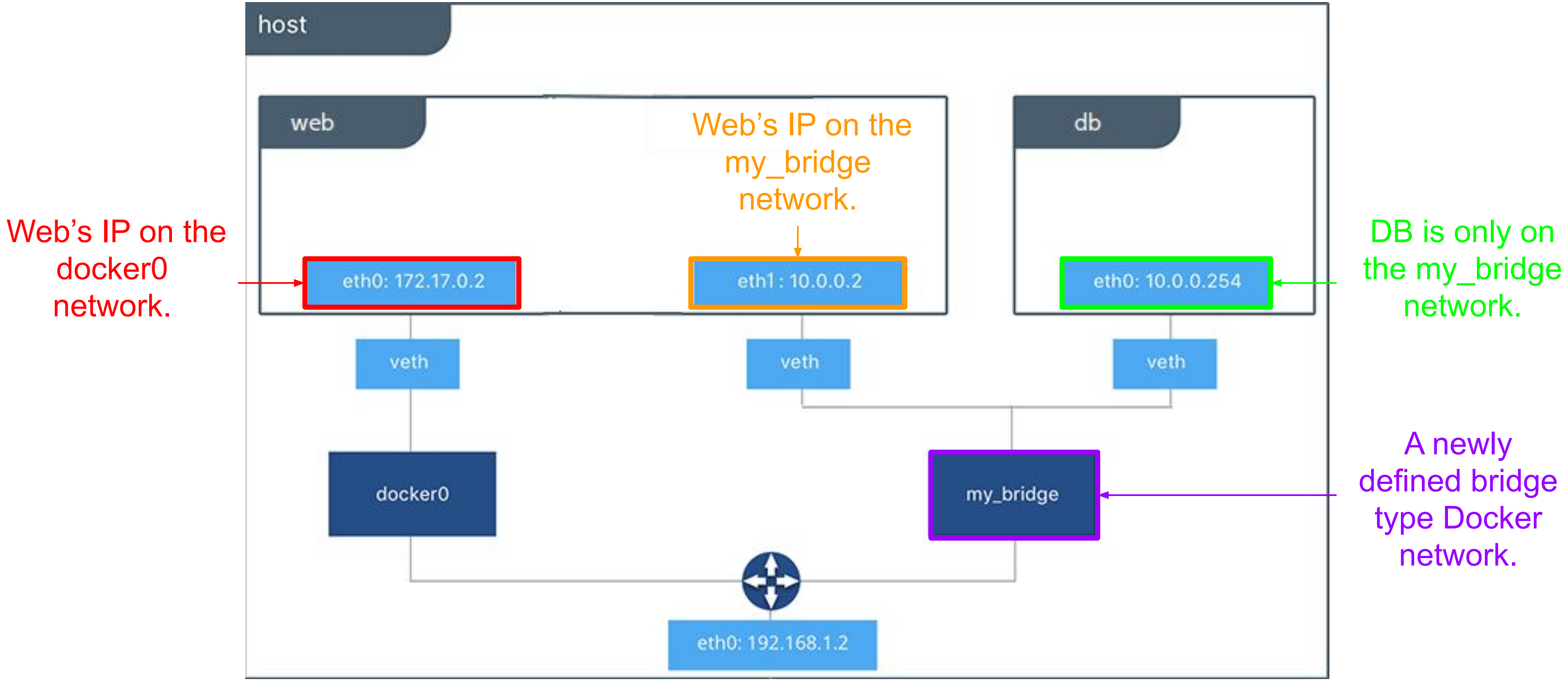
The mapped port on the Docker host.

Requests to http://192.168.1.2:8000 will be routed to port 5000 in the web container.

Advanced Docker Networking

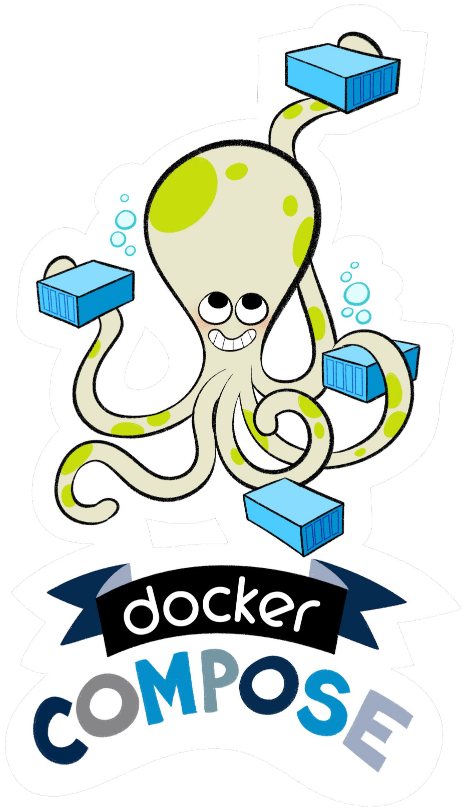
- It is possible to create additional Docker networks beyond `docker0`.
- A container may be granted network interfaces on several different networks to enable cross network communication between containers.
- There are also different types of network drivers beyond the standard bridge driver.
    - `Host`: Removes network isolation between containers and the Docker host.
    - `Overlay`: Used to connect multiple Docker daemons together for networking across hosts.
    - `Macvlan`: Assign MAC addresses to containers so they look like physical devices.
    - `None`: Don’t do any networking at all. Usually used alongside a custom plugin.
    - Or write your own network plugin.
- Can also customize the DNS server used by containers.
    - By default, containers will use the resolv.conf from the host to handle DNS resolution.
- Your Docker networking setup can be as simple or complex as you want.

Advanced Docker Networking Example


host
web
db
Web's IP on the my_bridge network.
Web's IP on the docker0 network.
DB is only on the my_bridge network.
eth0: 172.17.0.2
eth1 : 10.0.0.2
eth0: 10.0.0.254
veth
veth
veth
docker0
my_bridge
A newly defined bridge type Docker network.
eth0: 192.168.1.2

Docker-Compose

- `docker-compose` is a tool for defining and running multi-container applications.
- Very useful for deploying complete application stacks in separate containers locally.
- Makes it simple to setup the networking required to allow many containers to communicate. Containers are resolvable via their service name defined in the compose file.
- Application containers defined via a simple YAML DSL in docker-compose.yml files.
- Let’s take a look at a few common docker-compose commands.



Common docker-compose Commands

- `docker-compose up`
  - Builds images (optionally) and starts the containers defined in a docker-compose.yml

```
# Start containers in docker-compose.yml in the background.
$ docker-compose up -d

# Build containers before starting.
$ docker-compose up -d --build

# Target a custom docker-compose.yml file.
$ docker-compose -f my-docker-compose.yml up -d
```

- `docker-compose stop`
  - Stops the containers defined in a docker-compose.yml.

```
# Stop all containers defined in docker-compose.yml
$ docker-compose stop
```

Common docker-compose Commands Cont.

- `docker-compose restart`
  - Restarts the containers define in a docker-compose.yml

```
# Restart all containers defined in docker-compose.yml
$ docker-compose restart

# Restart my-container defined in docker-compose.yml
$ docker-compose restart my-container
```

- `docker-compose down`
  - Stops containers defined in a docker-compose.yml and removes the network created by docker-compose for the container stack.
  - Can also be used to purge volumes used by the containers.

```
# Stop containers in docker-compose.yml and remove network.
$ docker-compose down

# Also remove volumes.
$ docker-compose down -v
```

docker-compose.yml Example

Defines the Compose file format version.

```
version: '3'

services:
  app:
    build: ./path/to/app/Dockerfile/
    ports:
      - "127.0.0.1:5000:5000"
    command: /start-my-app.sh

  db:
    image: "mysql:8"
    Ports:
      - "127.0.0.1:3306:3306"
    volumes:
      - db-data:/var/lib/mysql

volumes:
  db-data:
```

Path to the Dockerfile for the “app” service.

Use a public image for the “db” service.

Save MySQL DB data to db-data volume.

Defines a named volume for persistent DB data

Special Topics: Docker Images from Scratch

- The scratch image is the common ancestor of all Docker images.
- The scratch image is empty. It cannot be pulled from Dockerhub and used to create containers on its own.
- The scratch image is typically used for base OS images, like Debian and Ubuntu.
- It can also be used to execute statically compiled binaries.

```
# A Debian base OS image example.
FROM scratch

# Copy in and decompress the root filesystem.
ADD rootfs.tar.xz /

CMD ["/bin/bash"]
```

Special Topics: Privileged Containers

- Recall that containers are virtualized at the OS level and share the kernel with the Docker host.
- What happens when the service running in a container needs to alter something in the kernel? Normally, a container cannot do this.
- If you run a container with the `--privileged` flag, that will allow the container to access and alter the kernel.
- This is useful if you are running a timekeeping service in a container (ntpd, chronyd etc.) or other services that require kernel access.
- Use privileged containers very sparingly.

```
# Run a privileged container.
Docker run --privileged my_image:latest
```

Special Topics: Windows Containers

- It is possible to build Windows Docker images and containers.
- Some Restrictions and Gotchas
    - Can only run Windows containers on a host with the same version of Windows running.
    - Need to use Linux style paths in Windows Dockerfiles (forward slashes instead of backslashes) for Dockerfile directives.
    - Scripts and commands requiring paths should use standard Windows paths.

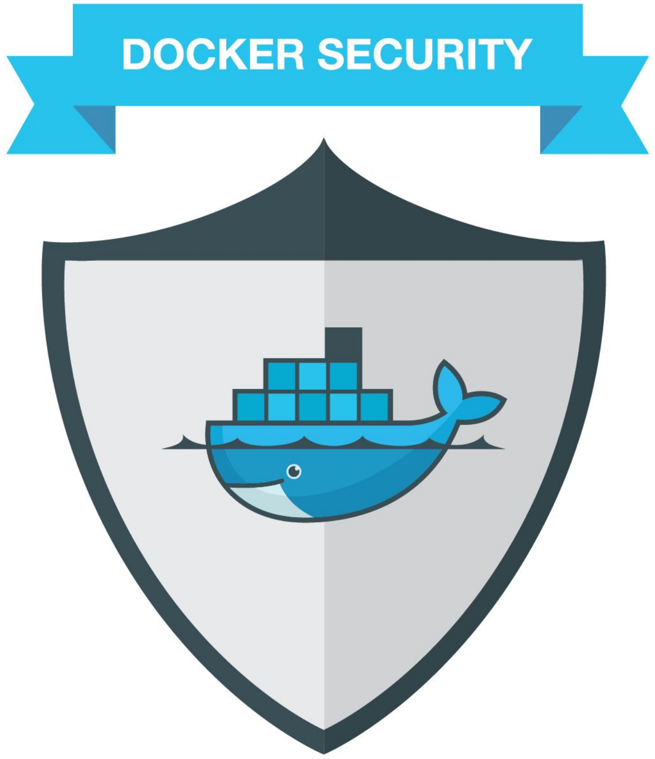
```
# Use a Windows Server base image.
FROM mcr.microsoft.com/windows/servercore:ltsc2019

# Copy in a Powershell script. Note Linux style path.
ADD script.ps1 /windows/temp/script.ps1

# Run the Powershell script. Note Windows style path.
RUN powershell.exe -executionpolicy bypass c:\windows\temp\script.ps1
```

Special Topics: Docker Security

- It is important to secure applications running within containers.
- Well designed containers can be very secure compared to a VM.
- There are three major security tiers to consider when securing Docker containers:
  - Application-Level Security
  - Container-Level Security
  - Docker Host-Level Security



Special Topics: Application-Level Security

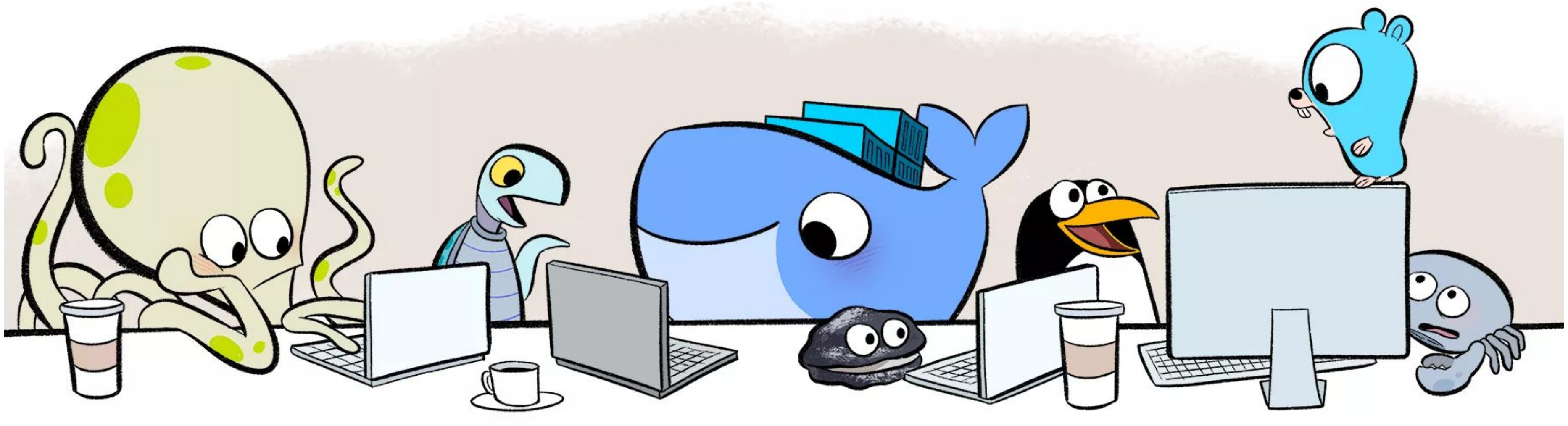
- Applications and software deployed in Docker images must be secure.
- If deploying software that you maintain, follow best practices on software design and implementation.
- If deploying third-party software, always deploy the latest security patched versions of the software.
- Implement and follow best practices for application access control (good passwords etc.)
- Avoid leaving backdoors, initial user accounts in applications.

Special Topics: Container-Level Security

- Following best practices for designing and building images and containers will yield containers that are easier to secure.
  - Don’t install extraneous packages.
  - Don’t open more ports than you need. Keep the attack surface for the container small.
  - Consider running commands and services within the container as a non-privileged user.
- Keep the container up to date by rebuilding the container image when updates are available and redeploying the container.
- DON’T PATCH RUNNING CONTAINERS! REBUILD THEM!

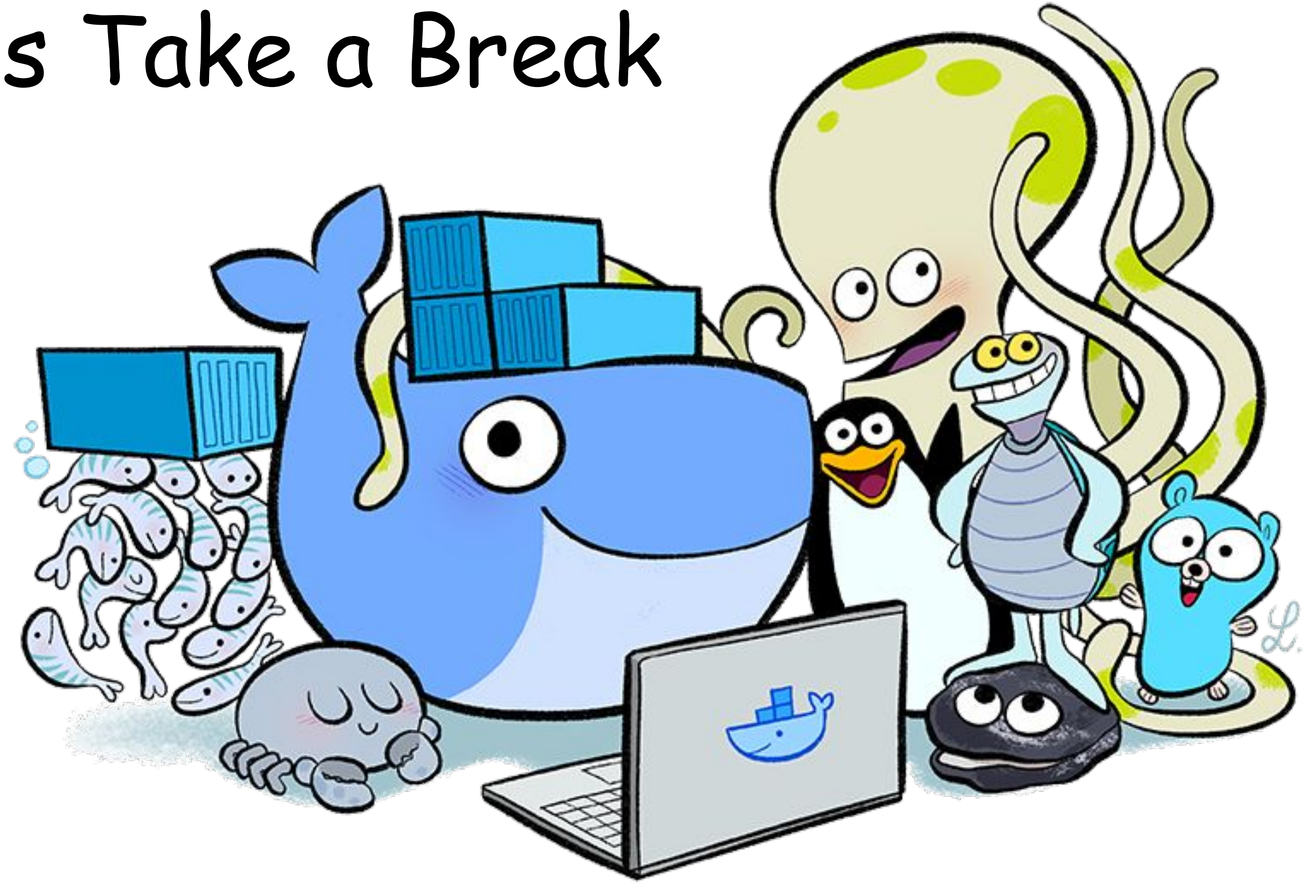
Special Topics: Docker Host-Level Security

- Implement good practices on access control for your Docker hosts.
- Don’t expose the Docker daemon on a TCP port.
  - Docker RBAC isn’t the best.
- Keep access to the `docker` binary controlled via sudoers.
- Keep your kernel patched and up to date.
  - Containers all share the host’s kernel.
  - An exploit that affects the kernel of the host will also affect the containers on the host.
- Keep the Docker daemon patched and up to date.
- Consider only allowing the Docker daemon to pull and run signed Docker images.
- Consider using SELinux or AppArmor to further protect your Docker host kernel.



Questions?

Let's Take a Break



Docker Best Practices and Use Cases

Overview of Part 3

- Docker Best Practices
  - Image Design
  - Environment Cleanliness
- Common Docker Pitfalls and Gotchas
- Docker Design Patterns
- Docker and Container Use Cases
- Container Orchestration Beyond docker-compose
- Docker Integrations with Cloud Providers
- Docker Success Stories

Docker Image Design Best Practices

- Minimize the number of image layers.
    - Good practice to do this even when we have `--squash`.
    - If a layer is getting complex, consider offloading the layer logic to a dedicated script that can be added to the image and run during the build.
- Don't install extraneous packages.
    - If a package is only required for a single layer, remove it in the same layer.
    - Avoid leaving development/compilation tools and libraries in production images.
- Run only one service per image.
- Expose only the ports needed for the service.
- Version your images with a documented standard. Don't just use :latest.
    - Semantic Versioning (semver), Calendar Versioning (calver) etc.
    - Useful to place a version file or version env var in your image during a build as well. Helps debug the inevitable "It says I'm running :latest" issues.

Docker Image Design Best Practices Con.

- Optimize your Dockerfile to allow for caching
    - Recall that a layer change will invalidate the cache of all following layers.
    - Put long running but infrequent directives at the beginning of the Dockerfile (heavy package installs etc.) so that this layer can be cached and reused.
    - Put short running and frequent directives at the end of the Dockerfile (source code copying etc)
- Avoid hardcoding configuration for your application in your Dockerfiles.
    - Using environment variables to drive configuration works well. It’s easy to provide these arguments at runtime.
    - docker-compose also makes it simple to inject environment variables at runtime into container stacks. This makes environment variables ideal to use for specifying host information for other other containerized services.
    - For complex configuration, consider relying upon users to mount in configuration files at runtime.

Docker Environment Cleanliness

- As you build more images and run more containers, you will likely see your file system fill up with old images, volumes and stopped containers.
- From time to time, clean up old data in your environment
- This can be done easily via the various docker prune command
  - `docker image prune`
  - `docker volume prune`
  - `docker container prune`
  - `docker network prune`
- More extreme cleanup
  - `docker system prune -a --volumes`

Gotcha: Docker-In-Docker

- There seems to always come a point where you want to run Docker inside a Docker container.
- This is a **BAD** idea. Performance will be poor and data integrity will be questionable at best.
- A better solution: Mount the Docker socket into the container that needs to run docker commands.
- This allows the container to spin up sibling Docker containers and achieves similar functionality.

```
# Mount in the Docker socket so my_container can interact with Docker.
$ docker run -d -v /var/run/docker.sock:/var/run/docker.sock my_container:latest
```

Gotcha: Image Caching

- Docker caches image layers during image builds in order to speed up the image building process.
- Suppose a package version in a build was updated. How does Docker know that the package installation will be different in this build?
  - It doesn't!
  - Docker won't try to rebuild the layer unless the layer has been explicitly updated in the Dockerfile.
- In cases like these, you need to build the image with the `--no-cache` flag so that all layers will be rebuilt.

```
# Build my-image:1.2.3 with the --no-cache option.
$ docker build --no-cache my-image:1.2.3 path/to/my/Dockerfile/
```

Gotcha: Bind Mount Permissions

- When a bind mount is mounted into a container, it keeps the same permissions as on the host.
- What if the host user doesn’t exist in the container? What if a process in the container isn’t running as `root`?
- In cases where files need to be accessed by non-root users within a container, be sure to update the permissions of the bind-mount prior to mounting it into the container.

Gotcha: Service Management in Containers

- On a regular Linux host, you typically manage services via systemctl or service.
- This doesn’t work in containers.
- A container’s lifetime is tied directly to it’s CMD. If the CMD exits, the container exits.
- A container should use foregrounded scripts/binaries instead of trying to manage services via a service management framework.
- If a script or binary requires some setup before being run, consider creating a script as your entrypoint that handles this setup.

Gotcha: Is :latest really :latest?

- If you always work with :latest there will inevitably come a point where your version of :latest differs from another’s version of :latest.
- This can be difficult to detect without checking the image version hash.
- Always tag images that you are going to distribute with a documented standard.
- The :latest tag should always be an alias for a standard version tag.
- Avoiding using :latest outside of development.

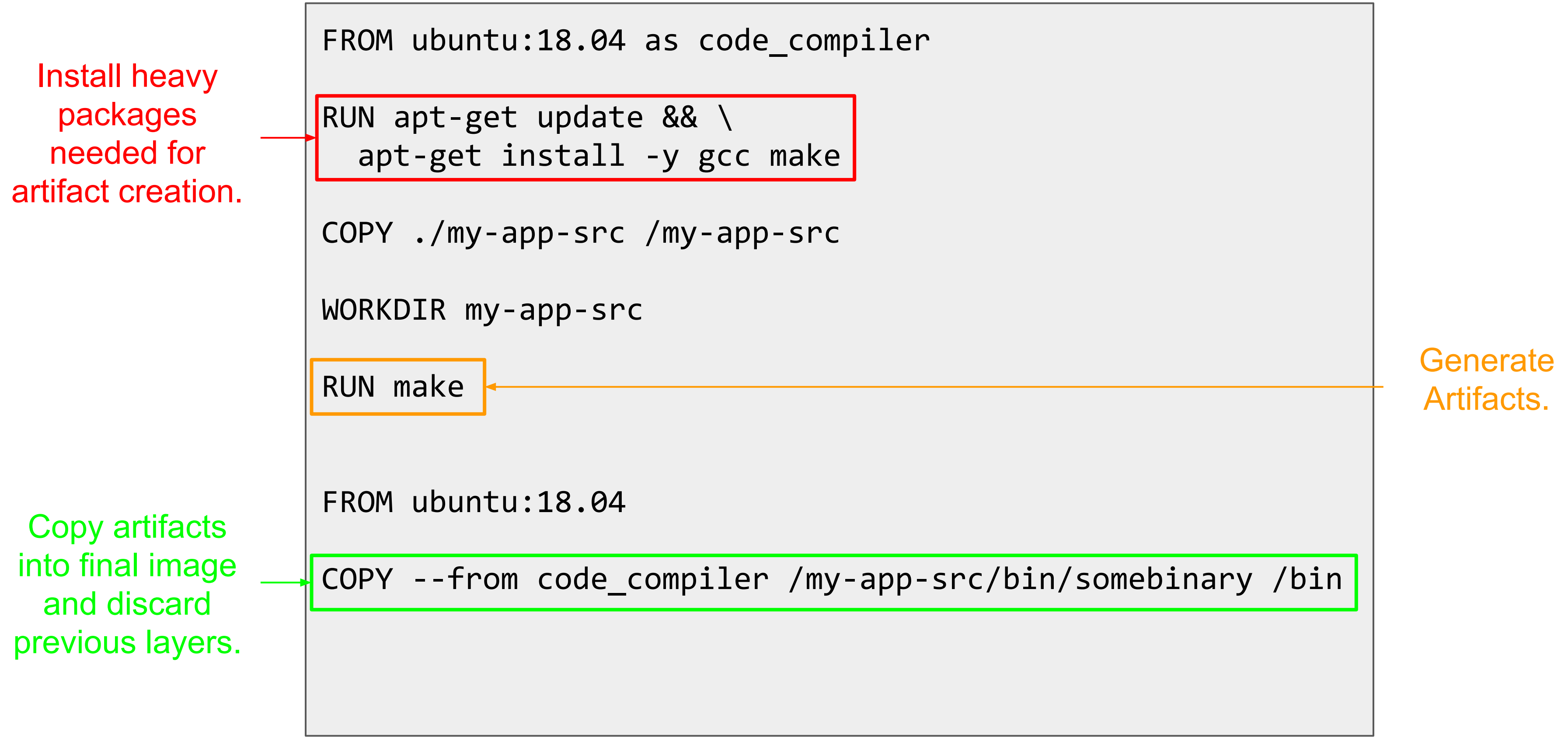
Docker Design Patterns

- Design patterns utilizing Docker and containers are still very much under development and are not as well established as other software and infrastructure design patterns.
- There are several patterns that have emerged in recent years that seem to have gained traction in the Docker community.
- A few design patterns that I’ve found success with:
    - Multi-Stage Build Pattern
    - Sidecar Pattern
    - Worker Queue Pattern

Multi-Stage Build Pattern

- As part of an image build, you may run into scenarios where you require many libraries/packages in order to produce artifacts.
    - Source code compilation, download and extraction of archives etc.
- Keeping all of these tools in your image causes image bloat.
- The multi-stage build pattern addresses this. This pattern allows a single Dockerfile to be split into separate stages, each with their own set of layers.
- Produce your artifacts in the stage where you download and install all of the relevant tools you need. Copy the artifacts from this stage into your final image and discard the earlier stage.
- This can be useful for maintaining a development and production image in the same Dockerfile.
- Unlike other patterns, this pattern is actually a newer feature of Dockerfiles.

Multi-Stage Build Dockerfile Example



Install heavy packages needed for artifact creation.

Generate Artifacts.

Copy artifacts into final image and discard previous layers.

```
FROM ubuntu:18.04 as code_compiler

RUN apt-get update && \
  apt-get install -y gcc make

COPY ./my-app-src /my-app-src

WORKDIR my-app-src

RUN make


FROM ubuntu:18.04

COPY --from code_compiler /my-app-src/bin/somebinary /bin
```

Sidecar Pattern

- A well designed container has one responsibility / runs one service.
- How do we handle scenarios where an important but otherwise unrelated service needs to run alongside a container’s service?
- For example, what if a web server also needed a service for handling log aggregation?
- We can use the sidecar pattern for this.
- Decouple services into separate containers but share pertinent data between containers via shared volumes.
- The web server can write logs as usual and the log aggregation container can have access to the logs via a shared volume.

Worker Queue Pattern

- Containers lend themselves well to batch processing and the worker queue pattern leverages this.
- Design a “master” container that can ingest and delegate batch tasks to “worker” containers via a message queue system.
- Design “worker” containers that can easily be horizontally scaled and can operate independently of one another.
    - `docker-compose` has the `--scale` option to easily do this.
- “Worker” containers fetch batch tasks from the message queue and process them.
- Scale the number of “worker” containers based upon need.

Docker and Container Use Cases

- Docker and Containers lend themselves well to a variety of different use cases.
- A few good use cases:
    - Reproducible Development Environment
    - Software Testing
    - CI/CD
    - Software/Application Packaging and Deployment
    - Managing Diverse Codesets
    - Managing Aging Applications
    - Implementing Microservices and Distributed Systems
    - Easing Application Migrations to the Cloud

Development with Docker

- Docker containers make excellent development environments.
- Easy to setup an environment containing all of the tools and packages required to run development software and tests via a Dockerfile and version it along with your codeset.
- Developers just need to have Docker installed to get a development environment up and running.
- When you run the container, mount in your local copy of the codeset for testing inside the development environment.
- Helps to ensure consistency between development environments, minimize the “it works on my dev machine” issues.

Testing with Docker

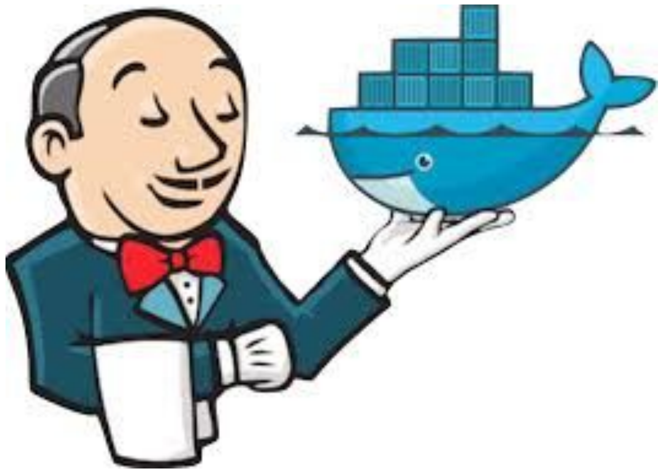
- Along with development environments, Docker containers also make excellent environments for running tests.
- A development environment packaged as a Docker image for a set of software makes unit testing simple.
    - Just include your unit testing framework and libraries in the development environment!
    - Include a script or other command that can be used to easily run unit tests.
- Including a docker-compose.yml with your codeset makes it simple to spin up a complete application stack with your codeset. This makes functional and integration testing for your stack easier.
- Functional testing require custom tools? Put them in their own Docker image and run it alongside your stack!

Software Packaging and Deployment with Docker

- Docker containers can be easily used as a packaging mechanism for software.
- The Dockerfile defines the OS, packages, libraries and runtime software needs.
- Complex software stacks can be easily and repeatedly deployed via docker-compose.
- Bundling your required packages with your software makes management of the software much simpler.
- The Dockerfile documents the software and deployment needs for your software.
- Helps to enable and drive DevOps practices. Involves developers with the packaging and hosting typically associated with system engineers.

CI/CD with Docker

- When your software is packaged in a Docker image, it becomes very easy to deploy in CI/CD environments for testing and other uses.
- Can spin up an entire application stack and run a variety of tests and other interactions against it in a completely automated fashion.
- No need for dedicated agents with specific sets of software installed for testing. Your agent just needs Docker and docker-compose.



Managing Diverse Codesets with Docker

- Since Docker containers are self-contained environments, containers make it simple to manage diverse codesets across a variety of languages.
- If a piece of software requires a certain language and/or other packages, just include them in the Docker image.
- Helps address the “If all you have is a hammer, everything looks like a nail.” issue.
- Lets you use the right tools and language for the job without having to worry about the underlying management of the software.

Managing Aging Applications with Docker

- Docker makes it simple to provide customized environments for software and applications. This is especially useful for aging applications that may require a very particular set of older software to run.
- Place these aging apps in containers so you don’t need to maintain dedicated environments for running them.
- Don’t understand how an older app is deployed? It’s all defined in the Dockerfile!
- No need to hunt down the resources that originally deployed the application.
- When the application is EoL’d, just remove the container. The hardware it is running on is still useful for other applications.

Microservices and Distributed Systems

- Well designed images that adhere to Docker best practices yield containers that are easily scalable and loosely coupled.
- These two features are key ingredients for a distributed systems.
- In effect, well designed containers *are* microservices and well designed container-based application stacks *are* distributed systems.
- If you have well designed containers and container stacks, the only other critical component for a complete distributed system is a means of container orchestration, such as Kubernetes or Docker Swarm.

Cloud Migrations

- Docker containers are widely supported across a variety of cloud providers.
- This widespread support can be leveraged to ease application migrations to the cloud.
- Once you have your application containerized, it becomes simple to move this application to run in a cloud-based Docker environment or on cloud hosts running Docker.

Container Orchestration beyond Docker-Compose

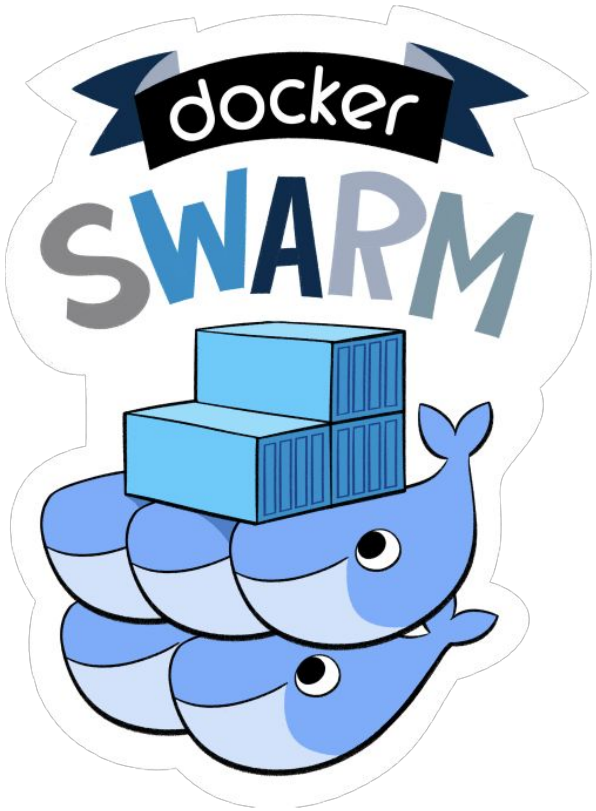
- Orchestrating large collections of containers across multiple Docker hosts requires functionality beyond that offered by Docker-Compose.
- There are a variety of container orchestration platforms that can be used to manage large collections of containers.
- Several Major Container Orchestration platforms:
  - Kubernetes
  - Docker Swarm
  - Apache Mesos

Kubernetes

- Probably the most well known and adopted container orchestration platform.
- Open source.
- Developed by Google.
- Integrated with many cloud providers, including AWS, GCP and Azure.
- Can also be deployed on your own hardware.
- Highly customizable and complex.
- Steep learning curve.

Docker Swarm

- Integrated with Docker.
- Allows a group of Docker hosts to be joined into a cluster.
- Application stacks defined in docker-compose.yml files can be deployed into the swarm cluster via the `docker stack` command set.
- Less of a learning curve than Kubernetes.
- Lets you leverage docker-compose.yml files you may already have in a more distributed fashion.

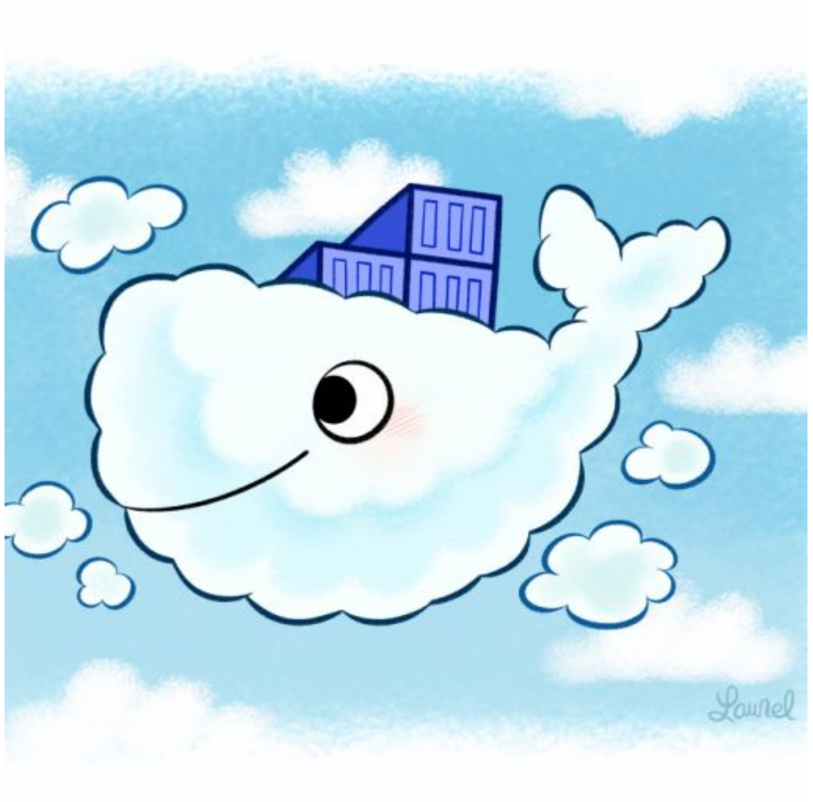


Apache Mesos

- Developed by the Apache Foundation
- Open Source
- Larger in scope than Kubernetes or Docker Swarm.
- Aims to provide complete data center resource abstraction rather than just container orchestration.
- Steep learning curve.
- More complex to deploy than Kubernetes or Docker Swarm but offers more features.

Cloud Provider Container Services

- Most major cloud providers have support for running Docker containers.
- AWS
  - ECS (Elastic Container Service)
  - ECR (Elastic Container Registry)
  - EKS (Elastic Kubernetes Service)
  - Fargate
- GCP
  - Run containers on GCE instances
  - GCR (Google Container Registry)
- Azure
  - Azure Container Instances
  - Azure Container Registry



Docker and AWS

- AWS has very robust support for containers.
- ECS
  - AWS container orchestration service.
- ECR
  - AWS hosted Docker image registry
- EKS
  - AWS hosted Kubernetes
- Fargate
  - Allows you to run containers without supporting servers.
  - Used by ECS.

Docker Success Stories - PayPal

- PayPal is an online money transfer platform.
- High volume, ~200 user payments per second across a variety of platforms (PayPal, Venmo, Braintree etc.)
- Maintaining different platforms across architectures and clouds was time consuming and managing uptime was difficult.
- Migrated 700+ applications to Docker Enterprise to enable a consistent operating platforms across cloud environments.
- Currently run over 200,000 containers as part of their production infrastructure.
- Achieved a 50% increase in productivity in building, testing and deploying applications via Docker

Docker Success Stories - Cornell University

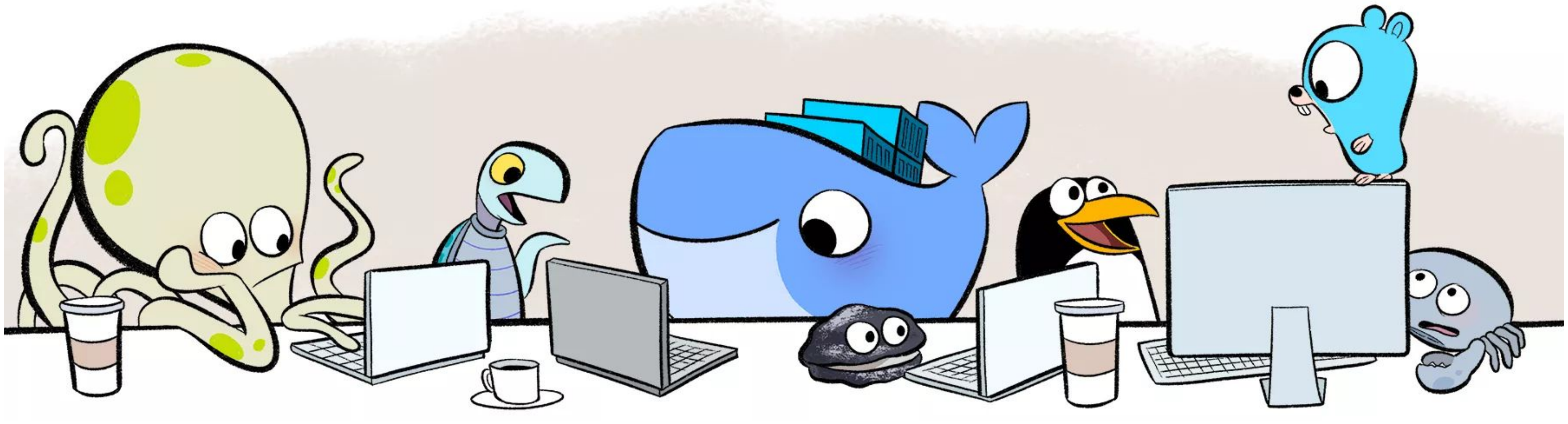
- Widely spread infrastructure and IT was difficult to coordinate, manage and support.
- Diverse applications could not be unified to one common platform nor moved into a cloud environment.
- Have migrated 49 applications to Docker containers and run more than 1000 containers on over 200 Docker Enterprise nodes in AWS.
- Docker helped to simplify their build processes and architecture, leading to 13x faster application deployments.

Docker Success Stories - Indiana University

- Desired to accelerate their application development process and migrate their 150 current applications by eliminating manual deployment steps.
- Moved to adopt a microservices architecture.
- Switched to using Docker to deploy all of their existing applications and new application platforms
- Greatly improved their application packaging and deployment times by standardizing their application builds on Docker and eliminating manual steps needed to deploy and manage applications.

References

- Docker CLI Reference
- Dockerfile Reference
- PayPal Customer Story
- Cornell Customer Story
- Indiana University Customer Story
- AWS Docker Support
- Kubernetes
- Docker Swarm
- Apache Mesos



Questions?

Thanks for Attending!