Containers

Using Docker and Other Tools

Linux Containers

Container runtimes like Docker direct the Linux kernel to create containers.

Linux creates a container by putting one or more processes into a separate set of **namespaces** and **control groups**.

Namespaces

Classic chroot (“change root”) creates a kind of filesystem “namespace.”

Containers bring even more types of namespaces:

• UTS:	hostname
• PID:	process lists and information
• Network:	network interfaces, IP addresses, routing tables, …
• Mount:	filesystem
• IPC:	interprocess communication
• User:	distinct set of users and UIDs (not used by default in Docker)
• Time:	virtualizes two timers (CLOCK_BOOTTIME and CLOCK_MONOTONIC)

Control Groups

Control groups (cgroups) were initially created to allow a system owner to set resource utilization limits on groups of processes. Here are a few types of control groups:

• Resource Limitation:	Limit RAM and Swap by cgroup
• Prioritization:	Prioritize and limit CPU and disk I/O
• Accounting:	Track utilization by a group of processes
• Freezer:	Freeze, checkpoint and unfreeze processes

Control groups are particularly suited to the challenges of multitenant use of a system.

Multi-Tenancy: Containers vs Virtual Machines

Containers are the next evolutionary step in multi-tenancy, improving over virtualization's efficiency gains.

A virtual machine has its own kernel, core subsystems (syslog, cron, udev..) and far more running processes than one needs to separate one app from another.

Containers all share the same kernel and generally won't have all those other system components.

Container Administration

There are a number of container runtimes:

- Docker and runc
- LXC and LXD
- OpenVZ
- Rkt

This section focuses on either on containers without a runtime or on Docker.

Container Concepts

Containers are the jails that Docker helps create and facilitate.

Images are the persistent state of a Docker container. They contain filesystems and configurations.

Reference: https://opencontainers.org/about/overview/

Traditional Mounting vs Union Mounted Filesystems

Images use **union-mounted filesystems**, an innovation that's particularly useful in containerized environments.

In traditional Linux filesystem mounting, you first mount a / (root) filesystem. Other filesystems are mounted as subdirectories of that filesystem, making it impossible to access the original contents of those subdirectories.

```
/               partition_1
/home           partition_2
/usr/local      partition_3
```

Union Mounted Filesystems

Union-mounted filesystems have multiple layers, stacked on top of each other.

Each layer overlays the filesystem below, overruling only those **files** it brings.

Layer 2:	Installs a single go binary in `/usr/local/bin/myapp`
Layer 1:	Adds an `/etc/README.txt` file.
Bottom layer:	Ubuntu minimal filesystem

The Power of Union-Mounting

You want to add a 1 MB file to an application I develop.

You download my 100 MB container image from Docker Hub, run a container, and add a file to the container.

You push the modified container image to Docker Hub. Docker Hub already has all the layers of the original image, so your Docker engine sends only 1 MB to Docker Hub.

I pull down the modified container image. Since I have all the original layers, I only need to pull down the last layer you added, which contains only the 1MB file.

Read-Only vs Read-Write Image Layers

There’s another big benefit to container images: immutability.

A running container’s filesystem is made up of the image layers that it came with, along with a top layer that’s ephemeral.

Only the top layer is read-write.

Running a container from an image doesn’t alter the image at all. If you destroy the container, the filesystem changes disappear, unless you intentionally commit that layer.

Exercise: Creating Containers with Docker

http://localhost:10000/exercises/docker-intro/

Reference: Running a Container

- Run a container based on an image:

```
# docker run --name=container [registry:]user/repo[:tag]
```

- Attach to the container's PID 1 process' stdio:

```
# docker attach container
```

- Add an interactive shell to a running container:

```
# docker exec –it container /bin/sh
```

- Detach from a container via **`Ctrl-P-Q.`**

Reference: Investigating Containers

List running containers:

```
# docker ps
```

List running and stopped containers:

```
# docker ps -a
```

Gain information about the container as JSON:

```
# docker inspect container
```

Read the containers logs (`stdout` and `stderr`) – insert `-f` for live-scrolling logs:

```
# docker logs [-f] container
```

Reference: Stopping/Removing Containers

Stop a running container:

```
# docker stop container
```

Restart a stopped container:

```
# docker start container
```

Destroy (remove) a stopped container, including its filesystem's read-write layer:

```
# docker rm container
```

Stop and destroy (remove) a running container, including its filesystem's read-write layer:

```
# docker rm –f container
```

Reference: Investigating Images

List container images cached on this system:

```
# docker images
```

Gain information about a container image's layers:

```
# docker history image
```

Reference: Images and Repositories

Commit an image to a repository

```
# docker commit <container> <repo>[:tag]
```

Pull an image from a repository

```
# docker pull [registry:]repo[:tag]
```

Push an image to a repository:

```
# docker push [registry:]repo[:tag]
```

Tag a locally-cached image's set of layers with another name:

```
# docker tag [registry:]repo[:tag]
```

Containerizing a Workload

To containerize a workload, we create a container image and the minimum configuration required to run the container. Both of these can be expressed in a Dockerfile.

Dockerfiles are simple – they’re named after and work like a Makefile, as we’ll see soon.

Dockerfile

Let's create our own Dockerfile, then build it.

```
FROM        centos:7
RUN         yum -y install httpd
EXPOSE      80/tcp
ENTRYPOINT  ["/usr/sbin/httpd"]
CMD         ["-D","FOREGROUND"]
```

Building our Image

```
Sending build context to Docker
daemon  3.095MB
Step 1/5 : FROM          centos:7
 ---> eeb6ee3f44bd
Step 2/5 : RUN           yum -y install httpd
 ---> Running in 90423e5ae39a
…
Complete!
Removing intermediate container 90423e5ae39a
 ---> 259e07e4c61d
Step 3/5 : EXPOSE        80/tcp
 ---> Running in 44447b648dff
Removing intermediate container 44447b648dff
 ---> 59e645b24dc0
```

```
Step 4/5 : ENTRYPOINT  ["/usr/sbin/httpd"]
 ---> Running in 26a38eeb1e8c
Removing intermediate container 26a38eeb1e8c
 ---> 828bc367842b
Step 5/5 : CMD          ["-D","FOREGROUND"]
 ---> Running in 7b951e83e13d
Removing intermediate container 7b951e83e13d
 ---> 00860ce307ea
Successfully built 00860ce307ea
Successfully tagged myimage:latest
```

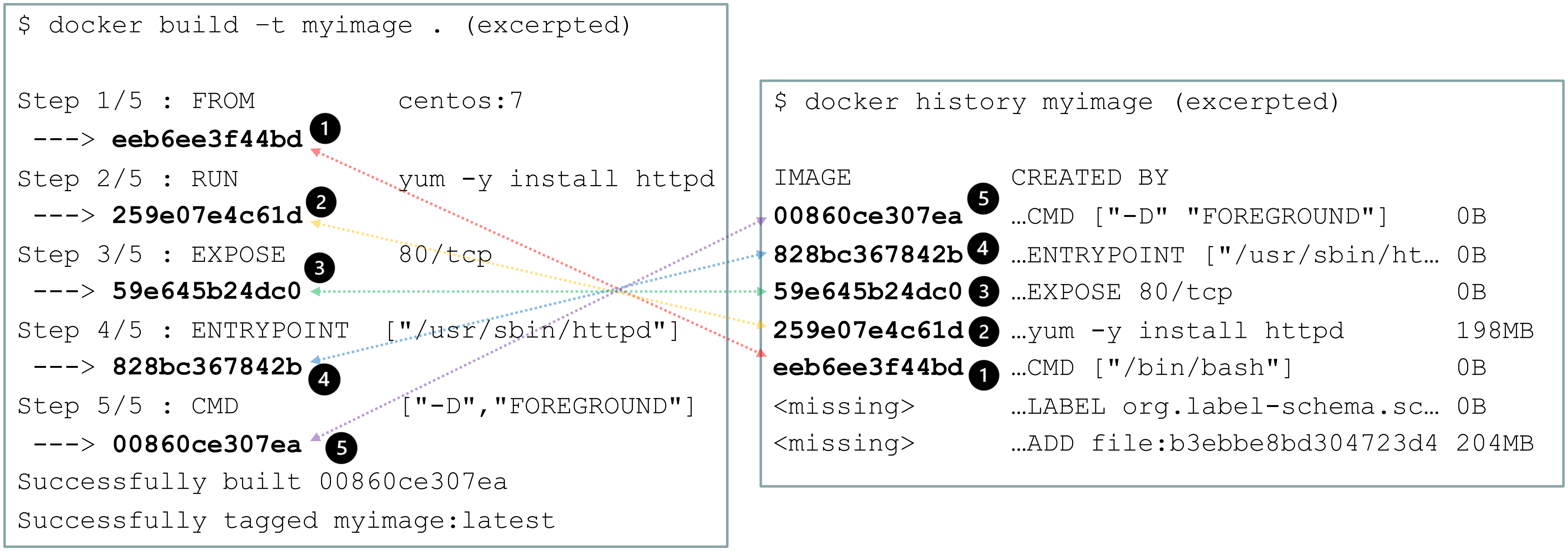
The Docker Cache

Docker has a very useful feature that uses this layered union-mounted filesystem.

When we build another container image whose Dockerfile has lines in common with a Dockerfile we've already built against… Docker keeps track of what filesystem layer contained the changes made by each step in the Dockerfile, and skips running the command when it knows what the results would be.

We'll see this in action in the next exercise.

The Docker Cache vs Docker History



```
$ docker build -t myimage . (excerpted)

Step 1/5 : FROM           centos:7
 ---> eeb6ee3f44bd ❶
Step 2/5 : RUN            yum -y install httpd
 ---> 259e07e4c61d ❷
Step 3/5 : EXPOSE         80/tcp
 ---> 59e645b24dc0 ❸
Step 4/5 : ENTRYPOINT  ["/usr/sbin/httpd"]
 ---> 828bc367842b ❹
Step 5/5 : CMD            ["-D","FOREGROUND"]
 ---> 00860ce307ea ❺
Successfully built 00860ce307ea
Successfully tagged myimage:latest
```

```
$ docker history myimage (excerpted)

IMAGE           CREATED BY
00860ce307ea ❺ …CMD ["-D" "FOREGROUND"]     0B
828bc367842b ❹ …ENTRYPOINT ["/usr/sbin/ht… 0B
59e645b24dc0 ❸ …EXPOSE 80/tcp               0B
259e07e4c61d ❷ …yum -y install httpd        198MB
eeb6ee3f44bd ❶ …CMD ["/bin/bash"]           0B
<missing>       …LABEL org.label-schema.sc… 0B
<missing>       …ADD file:b3ebbe8bd304723d4 204MB
```

Starting our Container

Now let's launch a container from our image. First, list the images.

```
# docker images

REPOSITORY    TAG       IMAGE ID        CREATED          VIRTUAL
SIZE
myimage       latest    844fd895bca4    2 minutes ago    269.5 MB
foo_is_jay    latest    9843d10249ab    19 hours ago     172.2 MB
```

Starting our Container

Start a container based on "myimage."

```
# docker run -d --name="mycontainer" myimage
a4a4f29ba888ff86325d68e96194ba6ebfb01beee86c8dc70e2f9ea2cc797807
```

Examining the Logs

We can see the logs from the container with docker logs.

```
# docker logs mycontainer
AH00558: httpd: Could not reliably determine the server's fully
qualified domain name, using 172.17.0.10. Set the 'ServerName'
directive globally to suppress this message
```

`docker logs -f` which works the same way as tail -f.

Let’s run a shell in our container with `docker exec`.

Accessing the Contained Program's Ports

Remember that EXPOSE entry in the Dockerfile?

We can reach that port from the Docker host, but nowhere else.

To **publish** the port to the outside world, use `docker run` **`-p`**.

```
# docker run -p 8123:80 -d --name=webserver myimage
```

This forwards the host's external 8123/tcp to webserver's port 80.

Docker Registry Exercise

http://localhost:10000/exercises/docker-registry-exercise/

Dockerfile Reference: FROM

FROM starts a new build stage.

Many Dockerfiles have only a single FROM.

During development, developers will often use a “full” Linux distribution image, like centos or ubuntu.

For production, there’s enormous value to using ”`FROM scratch`” or starting from a minimal image, like `busybox` or those from the `distroless` project.

Minimal Images

When you use “`FROM scratch`” in a build stage, Docker interprets this as a no-op. It creates no initial layer.

Alternatively, you can use a very minimal base layer, like `busybox` or one of the images from Google’s `distroless` project.

```
gcr.io/distroless/static : ca-certs,/etc/passwd, /tmp tzdata
gcr.io/distroless/base : (static), glibc, openssl, and libssl
```

https://hub.docker.com/_/busybox
https://github.com/GoogleContainerTools/distroless

Dockerfile Reference: multi-FROM Builds

FROM starts a new build stage.

Dockerfiles with more than one build stage will generally use the intermediary build stages following one of two patterns:

```
FROM centos:7 AS stage1
RUN somecommand

FROM stage1
RUN anothercommand
```

```
FROM centos:7 AS stage1
RUN somecommand

FROM scratch
COPY --from=stage1 /usr/ /
```

Dockerfile Reference: COPY and ADD

```
# Use COPY to copy files from the working directory to the image:
COPY --chown root:somegroup etc/ /etc/
COPY --chown www-data:www-data html/index.html /var/www/html/

# ADD works similarly to COPY, but also can extract a tar file
# onto the filesystem or pull a URL.
ADD html.tar /var/www/html/
ADD https://example.com/somefile.html /var/www/html/
```

COPY is recommended over ADD unless we need to extract a tar file or pull a URL.

Dockerfile Reference: ENTRYPOINT and COMMAND

To specify the program that runs in a Dockerfile, you have three options:

```
# Option 1 allows docker run to override what program runs.
CMD ["program-can-override",“arg1-can-override”,...]

# Option 2 restricts Docker from overriding anything.
ENTRYPOINT [“program”,”arg1”,...]

# Option 3 restricts overriding of the program and first two args
ENTRYPOINT [“program”,”arg1”,”arg2”]
CMD [“arg3-changing”,”arg4-changing”]
```

Dockerfile Reference: ENV

ENV allows you to set default environment variables in the image.

```
ENV DBNAME myapp
ENV READWRITE no
```

You can override or add to these environment variables when you create a container:

```
docker run -e READWRITE=yes -e ENVIRONMENT=prod image
```

Try to avoid using anything specific to any environment or even to your organization in the image's environment variables. This makes it easier to avoid unforeseen security issues.

Dockerfile Reference: ARG

ARG allows you to set variables for the Dockerfile itself.

```
FROM centos:7
ARG FILE=/newfile
RUN somecommand $FILE
```

These variables are scoped to the specific build stage.

In the second example, the second `somecommand` call gets a blank argument.

```
FROM centos:7 AS stage1
ARG FILE=/newfile
RUN somecommand $FILE
FROM busybox
RUN somecommand $FILE
```

IPTABLES in Docker

Docker creates iptables rules by itself, like this:

```
NAT Table:
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE

FILTER Table:
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -
j ACCEPT
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
```

IPTABLES: Port Publishing

When we published a port, it added these two rules:

```
-A DOCKER ! -i docker0 -p tcp -m tcp --dport 8123 -j DNAT --to-destination 172.17.0.11:80
-A DOCKER -d 172.17.0.11/32 ! -i docker0 -o docker0 -p tcp -m tcp --dport 80 -j ACCEPT
```

You can configure this with two daemon options, both of which default to true:

`--icc=false`	stop inter-container communications
`--iptables=false`	iptables should be manual, not automatic

Logging with Syslog

Docker containers don't log to syslog by default. In fact, they don't have `/dev/log` device! Let's add that.

```
# docker run -v /dev/log:/dev/log -it img /bin/bash
[root@9426cbdfb662 /]# logger "Log from the container"
```

```
# grep logger /var/log/messages
Jul 19 16:09:14 localhost logger: Log from the
container
```

Volume Mounts

Wait, what was that `-v` argument to `docker run`?

```
docker run -v /dev/log:/dev/log -it img /bin/bash
```

This shared the host's `/dev/log` with the container.

In general, the syntax is:

```
      -v /host_dir:/container_dir
```

This shares the host's `/host_dir` directory into the container's `/container_dir`.

Exercise: DockerDud

Please:

Open the Firefox browser on the class machine to: http://localhost:10000/exercises/dockerdud-dockersecurity

Docker Man Pages

When in doubt, read the docs. Each of these is a man page!

docker-attach(1)	Attach to a running container
docker-build(1)	Build an image from a Dockerfile
docker-commit(1)	Create a new image from a container's changes
docker-cp(1)	Copy files/folders from a container's filesystem to the host
docker-create(1)	Create a new container
docker-diff(1)	Inspect changes on a container's filesystem
docker-events(1)	Get real time events from the server
docker-exec(1)	Run a command in a running container
docker-export(1)	Stream the contents of a container as a tar archive
docker-history(1)	Show the history of an image
docker-images(1)	List images
docker-import(1)	Create a new filesystem image from the contents of a tarball
docker-info(1)	Display system-wide information

Docker Man Pages: 2 of 3

docker-inspect(1)	Return low-level information on a container or image
docker-kill(1)	Kill a running container (all processes inside it)
docker-load(1)	Load an image from a tar archive
docker-login(1)	Register or login to a Docker Registry Service
docker-logout(1)	Log the user out of a Docker Registry Service
docker-logs(1)	Fetch the logs of a container
docker-pause(1)	Pause all processes within a container
docker-port(1)	Lookup the public-facing port which is NAT-ed to PRIVATE_PORT
docker-ps(1)	List containers
docker-pull(1)	Pull an image or a repository from a Docker Registry Service
docker-push(1)	Push an image or a repository to a Docker Registry Service
docker-restart(1)	Restart a running container
docker-rm(1)	Remove one or more containers
docker-rmi(1)	Remove one or more images
docker-run(1)	Run a command in a new container

Docker Man Pages: 3 of 3

docker-save(1)	Save an image to a tar archive
docker-search(1)	Search for an image in the Docker index
docker-start(1)	Start a stopped container
docker-stats(1)	Display a live stream of one or more containers' resource usage statistics
docker-stop(1)	Stop a running container
docker-tag(1)	Tag an image into a repository
docker-top(1)	Lookup the running processes of a container
docker-unpause(1)	Unpause all processes within a container
docker-version(1)	Show the Docker version information
docker-wait(1)	Block until a container stops, then print its exit codeindex