Docker Fundamentals

Salesforce, Inc.

Paul L. Anderson
Office: (760) 436-9163
Email: paul@asgteach.com
Website: https://asgteach.com

Copyright Information

Paul L. Anderson
1212 Eolus Avenue
Encinitas, CA 92024-1728

Office: (760) 436-9163
Email: paul@asgteach.com
Website: **`https://asgteach.com`**

Follow Paul on Twitter at @paul_asgteach

12-22

Docker Fundamentals

Course Description

Docker is an open-source cross-platform tool that makes it easy to create, deploy, and run applications in microservice containers. These containers are portable across different machine platforms and Docker provides services that install, run, publish, and remove software. Docker has become the defacto standard for helping run distributed applications in the cloud and on local infrastructures.

Docker Fundamentals is a two-day course with hands-on workshops designed for those individuals who wish to learn how to use Docker. This course combines class lectures using slides and handouts, strengthening the material with hands-on lab exercises.

Learning Objectives

Docker Fundamentals is intended for software engineers, systems analysts and administrators, program managers, and user support personnel who wish to learn how to use Docker with microservice containers. Knowledge of Linux commands is helpful but not mandatory for this course.

The learning objectives of this course are:

- To learn the difference between containers and virtual machines.
- To learn how the Docker platform interacts with microservice containers.
- To understand the architecture of the Docker platform and its components.
- To learn the Docker commands and how to use them effectively.
- To learn how to push and pull images from the public Docker registry.
- To learn how to map and share files between containers.
- To learn how to use Docker with web service containers.
- To learn how to build Docker images from Dockerfiles.
- To learn how to do multi-stage builds to reduce Docker image sizes.
- To learn how to push and pull images from the private Docker registry.
- To learn how to create and use Docker volumes for persistence.
- To learn how to create and use Docker networks for container communications.
- To learn how to use Docker compose.

Docker Fundamentals

Seminar Leader

Paul L. Anderson, MSEE, has been teaching technical courses for over 25 years. As a consultant to industry, he has designed and implemented custom software under Linux, and Windows. As a seminar leader, he has taught Python, Go, Java, JavaFX, C/C++, Perl, Linux, C#, and OOD/UML courses for private industry and government. He has also taught seminars in Europe, Latin America, and India.

Mr. Anderson specializes in making the technical aspects of software engineering and operating systems understandable. He is a co-author and contributor of nine textbooks:

- *The Definitive Guide to Modern Java Clients with JavaFX-2nd Edition,* Apress, 2021
- *JavaFX Rich Client Programming on the NetBeans Platform,* Addison-Wesley, 2015
- *Essential JavaFX*, Sun Microsystems Press, Prentice Hall, 2009
- *Assemble the Social Web with zembly*, Sun Microsystems Press/Prentice Hall, 2009
- *Java Studio Creator Field Guide*, Sun Microsystems Press/Prentice Hall, 2006
- *Enterprise JavaBeans Component Architecture*, Prentice Hall, 2002
- *Navigating C++ and Object-Oriented Design*, Prentice Hall, 1989
- *Advanced C: Tips and Techniques*, Hayden Publishing, 1988
- *The UNIX C Shell Field Guide*, Prentice Hall, 1986

Paul is a Java Champion and Oracle Groundbreaker Ambassador. He is a frequent speaker at Devoxx, DevNexus, and Oracle Code conferences and has conducted tutorials and sessions at NetBeans Day and Java University. Paul is also a member of the NetBeans Dream Team and the author of LiveLesson training videos on JavaFX and Java Reflection.

For more information, visit **https://asgteach.com.** You can follow Paul on **Twitter** at **@paul_asgteach** and on **Facebook** at the **Anderson Software Group**.

Daily Class Schedule

Class begins at 9:00AM and ends at 5:00PM. There will be several breaks each day, in addition to hands-on lab. Lunch from 12:00-1:00pm.

Docker Fundamentals
Course Outline

- **Section 1 - Docker Overview**
  - Microservices and Containers
  - Monolithic vs. Microservice
  - Scaling Concepts
  - Containers vs. Virtual Machines
  - Docker Platform
  - Docker Architecture
  - Docker Commands
  - Creating and Running Containers
  - Pulling and Pushing Images
  - Detached Containers
  - Container Lifecycles

- **Section 2 - Working with Docker**
  - Dockerfiles
  - Why Use Dockerfiles?
  - Dockerfile Instructions
  - Building Images
  - Shell and Exec Forms
  - Docker Volumes
  - Bind Mounts
  - Mapping and Sharing Files
  - Web Server Containers

- **Section 3 - More Docker Topics**
  - Private Registries
  - Local Registry Images
  - Multi-Stage Builds
  - Docker Volumes
  - Creating and Using Volumes
  - Docker Networks
  - Creating and Using Networks

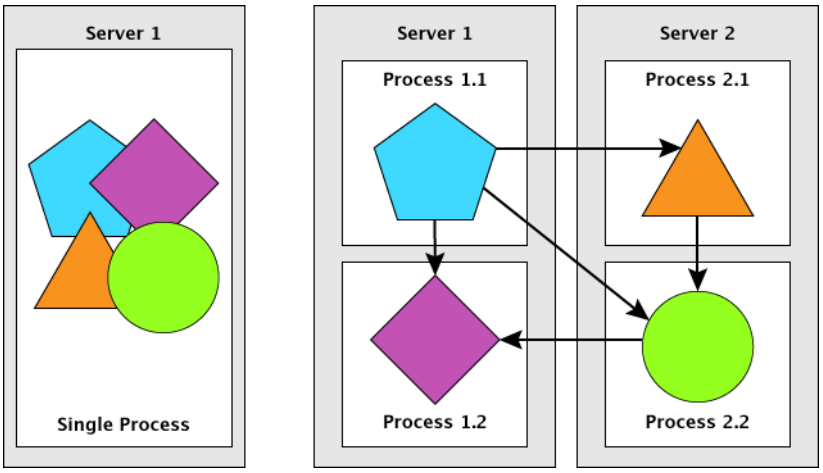
- **Section 4 - Docker Compose**
  - Overview
  - Create and Run Services
  - Pull Images and Build Services
  - Service Lifecycles
  - Service Environment Variables
  - Volume Services
  - Network Services

Section 1
Docker Overview

- **Microservices and Containers**
  - What are Microservices?
  - Monolithic vs. Microservice
  - Scaling Concepts
  - What are Containers?
  - Container Architecture
  - Containers vs. Virtual Machines
- **Docker Platform**
  - Architecture
  - Docker Concepts
  - Dependencies
- **Docker Commands**
  - Management Commands
  - Creating and Running Containers
  - Pull and Push Images
  - Executing Commands in Host and Container
  - Detached Containers
  - Saving Containers to Named Images
  - Fetching Logs
  - Archiving Containers and Images
  - Copying Files To/From Host/Container
  - Removing Containers and Images
  - Container Lifecyle

Microservices

- **What Are Microservices?**
  - Better alternative to *monolithic* applications
  - Runs as an independent process
  - Communicates with other microservices
  - Uses HTTP and REST APIs to communicate
- **Monolithic vs. Microservice**



- **Monolithic Applications**
  - Tightly coupled
  - Must be developed, deployed, and managed as *one* entity
  - Changes to one part requires redeployment of entire app
  - Typically requires powerful servers for app resources
- **Microservices**
  - May develop and deploy each microservice separately
  - Changes to one microservice doesn't affect another one
  - Forces developers to decouple large apps into smaller ones
  - Runs as separate processes within server

Scaling Concepts

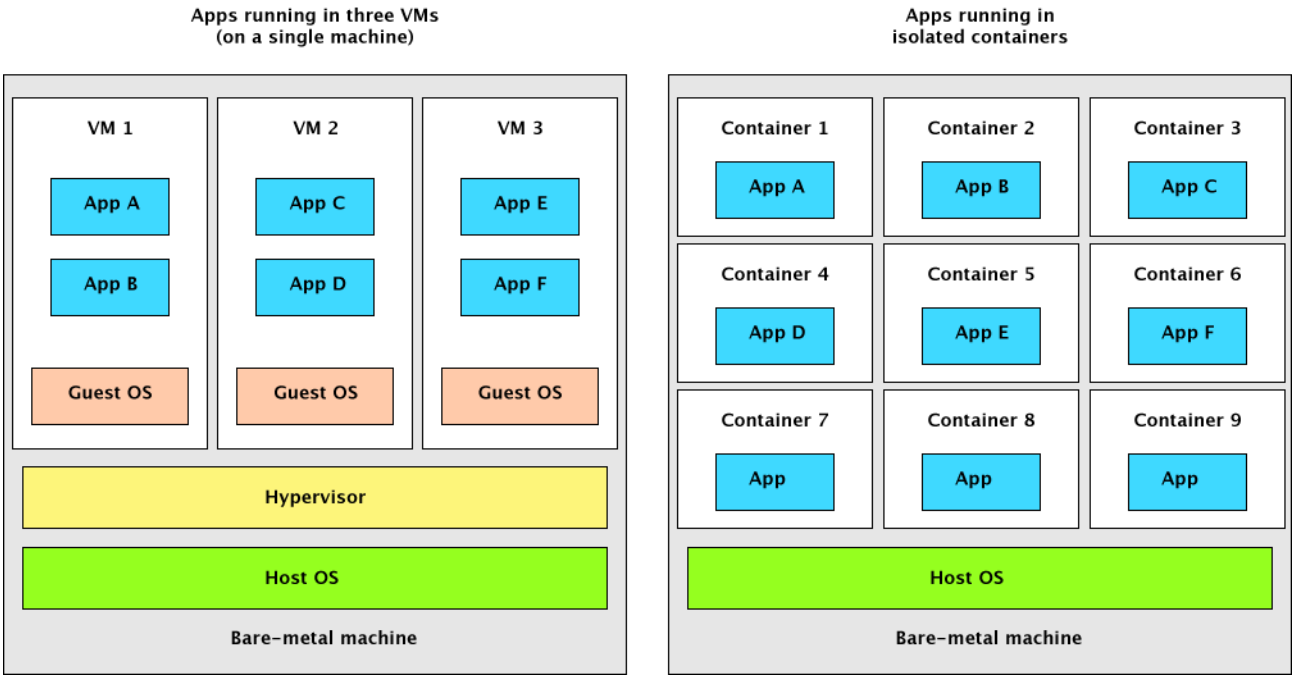
- **Vertical Scaling**
  - Scaling *up*
  - Add more CPUs and memory
  - May be expensive
  - Subject to upper limits
- **Horizontal Scaling**
  - Scaling *out*
  - Setup more servers and run *replicas* (copies) of the app
  - Relatively cheap
  - May require changes to application code
- **Monolithic Scaling**
  - Subject to unconstrained growth of inter-dependencies
  - Must scale up or out to deal with increasing loads on system
  - If any part isn't scalable, the whole app becomes unscalable
- **Microservice Scaling**
  - Done on a per-service basis
  - Scale only services that require more resources
  - Components may be replicated and run as multiple processes deployed on different servers
  - Others can run as a single application process
  - Splitting an app into microservices allows you to scale horizontally those parts that can scale out
  - For parts that can't scale out, scale up vertically

Containers

- **What are Containers?**
  - Isolated runtime environment
  - Exposes a different environment to each container
  - Isolates one container from another
  - May run multiple containers on same host machine
  - Similar to VMs but more lightweight and much less overhead
- **Container Implementation**
  - Similar to FreeBSD *jail* concepts for resource isolation
  - Uses Linux container technologies
- **Linux Namespaces**
  - Each process sees its own view of the system
  - Files, processes, network, hostname, etc.
  - Every container can have its own namespace
- **Linux Namespace Points**
  - Process ID, User ID
  - Mount, Network
  - IPC and UTS
- **Linux Control Groups**
  - Limits the amount of resources a process may consume
  - CPU, memory, network bandwidth, etc.
  - Implemented with Linux kernel *cgroups*

Containers vs. Virtual Machines

- **Architectures**

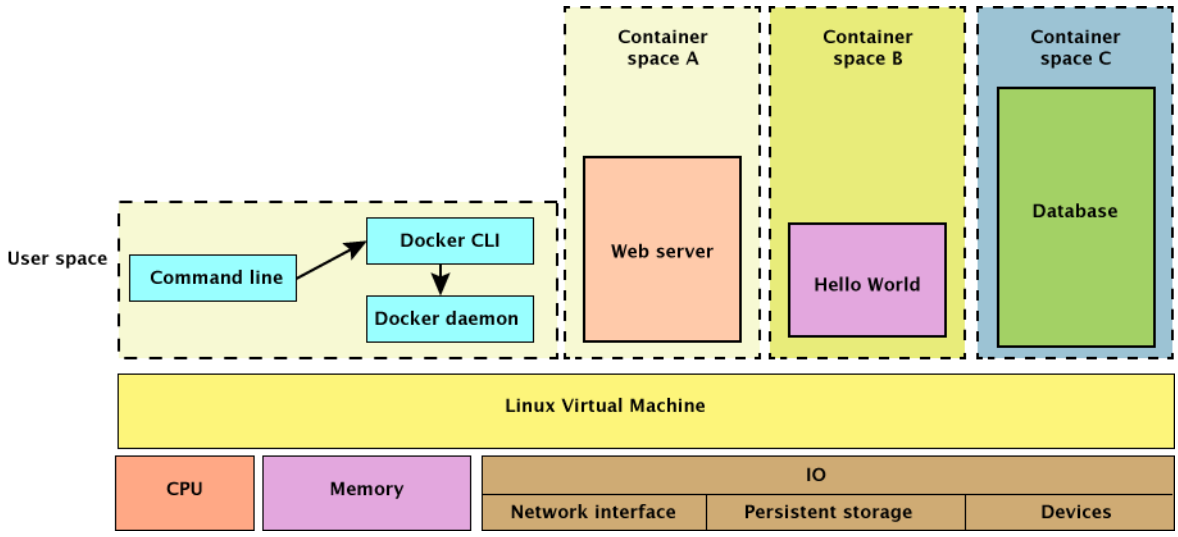


- **Virtual Machines**
  - Processes run in separate operating systems
  - Each VM runs its own set of system processes
  - Apps perform system calls to host through hypervisor
  - Large overhead, slow startup time

- **Containers**
  - Process runs inside host operating system
  - No separate set of system processes or hypervisor
  - Perform system calls directly to host
  - Lightweight, fast startup time

Docker Platform

- **What is Docker?**
  - Container system portable across different machines
  - Packages, distributes, and runs applications
  - Includes application *environment*
  - Command line program and a background *daemon*
  - Provides services that install, run, publish, remove software
- **Architecture**



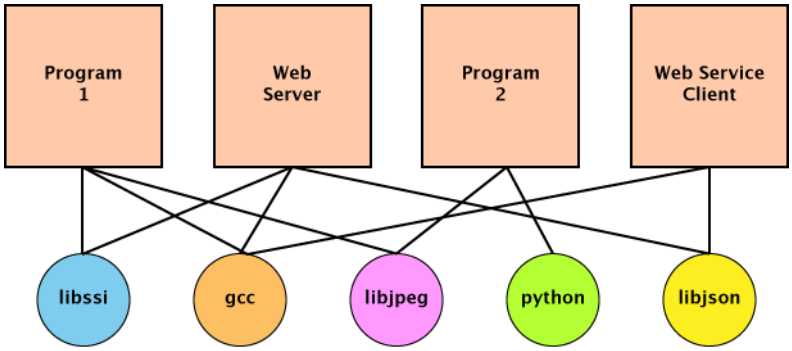
- **Important Concepts**
  - Users interact with CLI (Command Line Interface)
  - Docker CLI and Docker daemon run in user space
  - Each container is a *child* process of Docker daemon
  - Programs running in a container may access only their *own* memory and resources using namespaces and cgroups

Docker Concepts

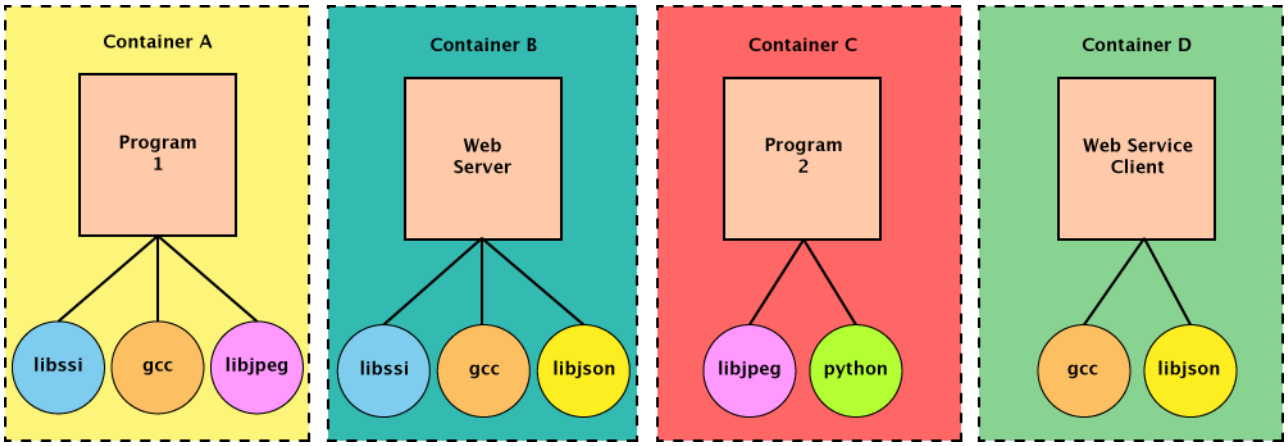
- **Images**
  - Packaged application with its environment and libraries
  - Includes file system and other metadata
  - Has image *layers*
- **Image Layers**
  - Every Docker image is built on top of another image
  - Different images may use the same *parent* image
  - Each layer is only stored *once*
  - Image layers are loaded *read-only*
  - Writable layer created on *top* of image layers when running
- **Registries**
  - Repository that stores Docker images
  - When building an image, you *push* (upload) to registry
  - When running an image, you *pull* (download) from registry
  - May be public or private (requires credentials)
- **Containers**
  - Linux container created from a Docker based image
  - Isolated from the host and all other processes
- **Why is Docker Important?**
  - Makes containers easy to work with and available to everyone
  - Industry wide acceptance with Amazon, Google, Microsoft, others
  - Registries are like app stores for mobile devices
  - Adoption of advanced isolation features with operating systems

Dependencies

- **Without Docker**



- **With Docker**



- **Important Notes**
  - **Without Docker, dependencies must be *shared***
  - **With Docker, dependencies are *copied***
  - **Docker keeps things organized and manageable with containers and images**

Docker Commands

- **Docker Version**

```
$ docker --version
$ docker -v
$ docker version
```

- **Docker Info**

```
$ docker info
```

- **Docker Help**

```
$ docker help
$ docker help cmd
```

- **Docker Command Syntax**

```
$ docker cmd options args
$ docker mgmtcmd options args
```

- **Management Commands**

```
builder             Manage builds
config              Manage Docker configs
container           Manage containers
context             Manage contexts
image               Manage images
network             Manage networks
node                Manage Swarm nodes
plugin              Manage plugins
secret              Manage Docker secrets
service             Manage services
stack               Manage Docker stacks
swarm               Manage Swarm
system              Manage Docker
trust               Manage trust on Docker images
volume              Manage volumes
```

Useful Management Commands

- **Builder**

```
$ docker help builder
build       Build image from Dockerfile
prune       Remove build cache
```

- **Container**

```
$ docker help container
attach      Attach STDIN,STDOUT,STDERR to container
commit      Create new image from container's changes
cp          Copy files/dirs from container to local fs
create      Create new container
diff        Inspect file/dir changes on container's fs
exec        Run command in container
export      Export container's filesystem as tarball
inspect     Display detailed info on containers

kill        Kill containers
logs        Fetch container logs
ls          List containers
pause       Pause all processes within containers
port        List port mappings for container
prune       Remove all stopped containers
rename      Rename container
restart     Restart containers

rm          Remove containers
run         Run command in new container
start       Start stopped containers
stats       Display container resource usage statistics
stop        Stop running containers
top         Display running processes of a container
unpause     Unpause all processes within containers
update      Update configuration containers
wait        Block until containers stop, show exit codes
```

Useful Management Commands

- **Image**

```
$ docker help image
build       Build image from Dockerfile
history     Show history of image
import      Import from tarball to create filesys image
inspect     Display detailed information on images
load        Load image from tarball or STDIN
ls          List images
prune       Remove unused images
pull        Pull image or repository from registry
push        Push image or repository to registry
rm          Remove images
save        Save image to tarball
tag         Create tag that refers to image
```

- **Network**

```
$ docker help network
connect     Connect container to network
create      Create network
disconnect  Disconnect container from network
inspect     Display detailed info on networks
ls          List networks
prune       Remove all unused networks
rm          Remove networks
```

- **Volume**

```
$ docker help volume
create      Create volume
inspect     Display detailed info on volumes
ls          List volumes
prune       Remove all unused local volumes
rm          Remove volumes
```

Using Docker Commands

- **Management Commands**

```
$ docker container run hello-world
$ docker run hello-world

$ docker container ls

$ docker image ls
$ docker images

$ docker network ls

$ docker volume ls
```

- **Other Commands**

```
$ docker events      Get real time events from server
$ docker images      List images
$ docker info        Show system-wide information
$ docker login       Log in to Docker registry
$ docker logout      Log out from Docker registry
$ docker ps          List containers
$ docker rmi         Remove one or more images
$ docker search      Search Docker Hub for images
$ docker version     Show Docker version information
```

- **Configuration Files**
  - Docker stores config files in `.docker` by default in home direc
  - Use `DOCKER_CONFIG` env var or `--config` option to modify
  - Option `--config` overrides `DOCKER_CONFIG`
  - Use config files in `mydir` when running `ps` command

```
$ docker --config ~/mydir/ ps
```

Docker Commands

- **Search Docker Images**

```
$ docker search hello-world
$ docker search ubuntu
$ docker search busybox
```

- **Run Container**

```
$ docker run hello-world
```

- **Create Container**

```
$ docker create hello-world
d20b1c00af6f....d62ac93d4df12101ce9cc8d300931
```

- **List Containers**

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND
d20b1c00af6f        hello-world         "/hello"
```

- **Start Container**

```
$ docker start -a d20b1c00af6f
```

- **Equivalent to Run**

```
$ docker start -a $(docker create hello-world)
```

- **List Images**

```
$ docker image ls hello-world
REPOSITORY      TAG         IMAGE ID        SIZE
hello-world     latest      fce289e99eb9    1.84kB
```

Docker Commands

- **Pull Images**

```
$ docker pull alpine
$ docker pull busybox
$ docker pull ubuntu
$ docker pull debian:10.4
```

- **Execute Commands from Host**

```
$ docker run busybox echo hello from container!
hello from container!
```

- **Execute Commands in Container**

```
$ docker run -it ubuntu /bin/bash
root@2c65433d9b67:/# ls /usr/local
bin  etc  games  include  lib  man  sbin  share  src
root@2c65433d9b67:/# exit
```

- **Launch Detached Container**

```
$ docker run -itd --name myubc ubuntu
188c6980a45f...66779d4087c5d74037af181f528bf9164
```

- **Execute Detached Command in Container**

```
$ docker exec -d myubc touch /tmp/myfile
```

- **Attach to Container**

```
$ docker attach myubc
root@188c6980a45f:/# ls /tmp/myfile
/tmp/myfile
root@188c6980a45f:/# exit
```

Docker Commands

- **Saving Containers to Named Images**

```
$ docker run -it --name myubc ubuntu /bin/bash
root@fce46cad24d6:/# apt-get update
root@fce46cad24d6:/# apt-get install vim
root@fce46cad24d6:/# vi newfile

root@fce46cad24d6:/# apt-get install tree
root@fce46cad24d6:/# tree /usr/local
root@fce46cad24d6:/# exit

$ docker commit myubc myubi

$ docker image ls myubi
REPOSITORY    TAG       IMAGE ID       SIZE
myubi         latest    25e17dcb78d5   153MB

$ docker run -it myubi /bin/bash
root@72b9a84a2367:/# vi newfile
root@72b9a84a2367:/# tree /usr/local
root@72b9a84a2367:/# exit
```

- **Display History of Docker Image**

```
$ docker history myubi
IMAGE          CREATED             CREATED BY
25e17dcb78d5   2 minutes ago       /bin/bash
```

- **Fetch Logs from Container**

```
$ docker run -it --name alpc alpine /bin/sh
/ # touch /tmp/myfile; hostname
9d8738ad809a
/ # exit

$ docker logs alpc
/ # touch /tmp/myfile; hostname
9d8738ad809a
/ # exit
```

Docker Commands

- **Archiving Containers**

```
$ docker export myubc -o myubc.tar
$ docker import myubc.tar myubi
```

- **Archiving Images**

```
$ docker save myubi -o myubi.tar
$ docker load -i myubi.tar
```

- **Push Image to Registry**

```
$ docker login
$ docker push username/myubi
```

- **Copying Files**

```
$ docker cp myfile myubc:/myfile
$ docker cp myfile myubc:/tmp

$ docker attach myubc
root@d587eca92164:/# ls myfile /tmp/myfile
/tmp/myfile  myfile
root@d587eca92164:/# exit

$ docker cp myubc:/etc/shells .
$ cat shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
```

- **Saving Container IDs**

```
$ docker create --cidfile /tmp/ub.id ubuntu
$ cat /tmp/ub.id
2a4c24be4461...e18a85407b1d2cc21b8a98e4a1a0b
```

Docker Commands

- **Stop and Start Containers**

```
$ docker stop myubc
$ docker start myubc
```

- **Restart Containers**

```
$ docker restart myubc
```

- **Kill Containers**

```
$ docker kill myubc
```

- **Remove Containers**

```
$ docker rm myubc          # must be stopped
$ docker rm -f myubc       # force removal
$ docker run --rm busybox echo Hello and Goodbye!

$ docker container prune -f      # all stopped containers
$ docker rm -vf $(docker ps -qa) # all containers
```

- **Remove Images**

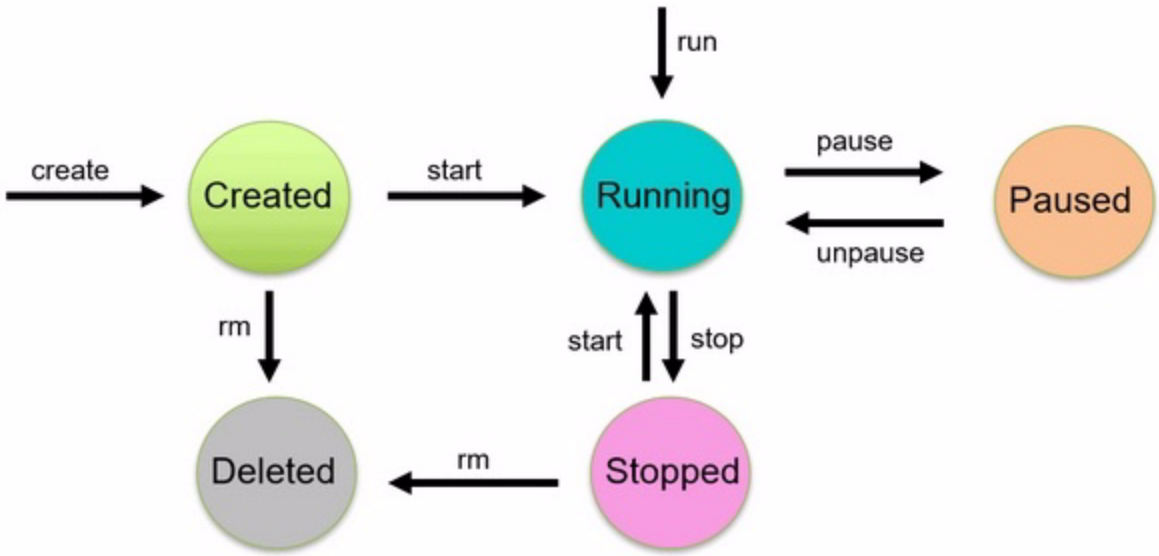
```
$ docker rmi alpine        # remove image

$ docker rmi -f $(docker images | awk '{print $3}') # all
$ docker image prune -f    # remove all dangling images
```

- **Important Notes**
  - Docker `stop` sends `SIG_HUP` to container
  - Safer and allows container to cleanup
  - Docker `kill` and `rm -f` send `SIG_KILL` to container
  - May result in file corruption and network issues

Container LifeCycle

- **Container States**
  - ***Created*: has been created but not started**
  - ***Running*: running with all its processes**
  - ***Paused*: processes have been paused**
  - ***Stopped*: processes have been stopped**
  - ***Deleted*: dead state**

- **State Diagram**



- **Examples**

```
$ docker create --name myc1 ubuntu
$ docker start myc1
$ docker run -it --name myc2 ubuntu
$ docker pause myc2
$ docker unpause myc2
$ docker stop myc1
$ docker rm myc2
```

Section 2
Working with Docker

- **Dockerfiles**
  - What Are Dockerfiles?
  - Why Use Dockerfiles?
  - Dockerfile Instructions
  - Building Images
  - Shell and Exec Forms
- **Docker Volumes**
  - Bind Mounts
  - Mapping Files
  - Sharing Files
- **Web Server Containers**
  - Apache
  - Nginx

Dockerfiles

- **What Are Dockerfiles?**
  - Text file of instructions to build Docker images
  - Instructions defined with value pair syntax
  - Instruction word followed by parameters
  - Every instruction has its own line in the file
  - Instructions are uppercase words by default
- **Why Use Dockerfiles?**
  - Better than configuring running containers and committing
  - Persistent, portable, and manageable
  - Flexible and more configurable
- **Important Notes**
  - Order of instructions in Dockerfile is significant
  - Instructions evaluated in sequential order
  - Docker images made up of *layers*
  - All instructions in Dockerfile generate *new* layer
  - Use instructions that change least *early* in the Dockerfile
  - Use instructions that change frequently in *later* part of Dockerfile
  - Best to keep images as *small* as possible
- **Dockerfile Instructions**

```
FROM        LABEL       COPY
ADD         ENV         ARG
USER        WORKDIR     VOLUME
EXPOSE      RUN         CMD
ENTRYPOINT  HEALTHCHECK ONBUILD
STOPSIGNAL  SHELL
```

Dockerfile Instructions

- **FROM**
  - Sets base image for the new image
  - Must be *first* instruction in Dockerfile
  - Creates first layer in new image
  - Use `scratch` to build images from an empty base layer

```
FROM busybox
FROM ubuntu:14.04
FROM scratch
```

- **LABEL**
  - Adds metadata to your Docker image
  - Uses embedded key-value pairs
  - Images may have more than one `LABEL`
  - Typically used for maintainer and version info

```
LABEL maintainer="Joe Jones <joe@company.com>"
LABEL version="1.0"
```

- **COPY**
  - Copy files and directories to Docker image being built
  - Format: `COPY [--chown User:Group] source dest`
  - *`source`* may be files with wildcards or directory
  - *`dest`* is a filename or path inside image being built
  - For *`source`* directory, all files copied but *not* directory itself

```
COPY testfile* /       # copy test files to /
COPY mydir /images     # copy all files in mydir

# copy mycert to / with user 10, group 20
COPY --chown=10:20 mycert /    # linux only
```

Dockerfile Instructions

- **ENV**
  - Define environment variables
  - Variables *persist* when a container is run with the image
  - Formats: `ENV key value`
    `ENV key1=value1 key2=value2 ...`
  - Images may have more than one `ENV`
  - `ENV` vars may be overridden with `run --env` command

```
ENV TOPDIR /usr/bob
ENV CUSTOMER1="mysql" CUSTOMER2="hadoop"
```

- **WORKDIR**
  - Changes current working directory in the image
  - May be used with `RUN`, `CMD`, `ENTRYPOINT`, `COPY`, `ADD` instructions
  - Format: `WORKDIR pathtodir`
  - Combination of Linux `mkdir` and `cd` commands
  - Path segments of `pathtodir` that do not exist will be created
  - May use `ENV` and `ARG` parameters for all or part of `pathtodir`
  - Images may have more than one `WORKDIR`
  - Relative paths become relative to the path of previous `WORKDIR`
  - Creates zero byte-sized layer in image

```
WORKDIR /usr/paul/music
WORKDIR jazz
WORKDIR rock

ENV SRC_DIR /usr/paul/myfiles
WORKDIR $SRC_DIR
COPY data $SRC_DIR
```

Dockerfile Instructions

- **RUN**
  - Executes command in new layer of image and commits
  - Often used for installing software packages
  - Shell form: `RUN` *`command`*
  - Exec form: `RUN ["`*`executable`*`", "`*`param1`*`", "`*`param2`*`"]`
  - Every RUN instruction creates new layer in image
  - Shell form uses default Linux shell or Windows command
  - Exec form better for containers *without* shells or commands
  - RUN instructions execute when *building* images

```
FROM ubuntu
RUN mkdir /mydir
RUN echo "Here is line one." > /mydir/myfile

RUN ["/bin/chmod", "664", "/mydir/myfile"]
RUN ["/bin/bash", "-c", \
    "echo Here is line two. >> /mydir/myfile"]
```

- **CMD**
  - Provide defaults for executing containers
  - May override default for `CMD` with `docker run` arguments
  - Shell form: `CMD` *`command param1 param2`*
  - Exec form: `CMD ["`*`executable`*`", "`*`param1`*`", "`*`param2`*`"]`
  - `ENTRYPOINT` form: `CMD ["`*`param1`*`", "`*`param2`*`"]`
  - Exec form better for containers *without* shells or commands
  - CMD instructions execute when *running* containers
  - Creates zero byte-sized layer in image

```
CMD echo "How many words here" | wc -w
CMD ["/usr/bin/wc", "-l", "/etc/passwd"]
CMD ["/bin/bash"]
```

Dockerfile Instructions

- **ENTRYPOINT**
  - Configures a container to run as an executable
  - Shell form: `ENTRYPOINT` *`command param1 param2`*
  - Exec form: `ENTRYPOINT ["`*`executable`*`","`*`parm1`*`","`*`parm2`*`"]`
  - Shell form starts shell and does *not* receive signals
  - `ENTRYPOINT` should always be used with `CMD` exec form
  - `CMD` instruction provides parameters to `ENTRYPOINT` application
  - Allows `docker run` arguments to be passed to entry point
  - Overrides `CMD` instruction that follows `ENTRYPOINT`
- **Example**

```
$ cat Dockerfile
FROM busybox
ENTRYPOINT ["/bin/head", "-5"]
CMD ["/etc/passwd"]

$ docker build -t head .

$ docker run head
root:x:0:0:root:/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/false
bin:x:2:2:bin:/bin:/bin/false
sys:x:3:3:sys:/dev:/bin/false
sync:x:4:100:sync:/bin:/bin/sync

$ docker run head /etc/group
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:
```

Building Images

- **Docker Build Command**
  - Sends build context and Dockerfile to docker daemon
  - Parses Dockerfile and builds image layer by layer

    Format: `docker build` ***`options path | url | -`***

  - Useful ***`options`*** with defaults

```
--rm            remove intermediate containers
--build-arg     set build-time vars
--tag           name tag for name:tag format
--file          Dockerfile name (path/Dockerfile)
```

  - Examples

```
$ docker build --tag demo .
$ docker build -t user-demo:1.0 .
$ docker build -t demo -f demo.df .
$ docker build --rm --build-arg user=50 .
$ docker build -t demo -f demos/demo.df .
```

- **Docker Build Context**
  - Build context is file system root for commands in Dockerfile
  - Every image build shows the build context size
  - Try to keep as small as possible

- **Ignoring Files and Directories**
  - Use `.dockerignore` to *exclude* files from build context
  - Place `.dockerignore` in *root* directory of build context

```
# .dockerignore that excludes files
**/*~
**/*.log
**/.DS_STORE
```

Building Images

- **Dockerfile**

```
FROM progrium/busybox
RUN opkg-install curl bash
CMD ["bin/bash"]
```

- **Build Image and Run**

```
$ docker build -t install .

$ docker run -it install
bash-5.0# which curl
/usr/bin/curl
bash-5.0# which bash
/bin/bash
```

- **Dockerfile**

```
FROM alpine
RUN apk add curl
ENTRYPOINT ["curl"]
CMD ["--help"]
```

- **Build Image and Run**

```
$ docker build -t curl .

$ docker run curl
Usage: curl [options...] <url>
-a, --append       Append to target file when uploading
-d, --data <data>  HTTP POST data
. . .

$ docker run curl google.com
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
. . .
```

Shell and Exec Forms

- **Dockerfile with Shell**

```
FROM busybox
ENV dir=/usr/bob/bkup
CMD echo "backing up to $dir"
```

- **Build Image and Run**

```
$ docker build -t env1 .
$ docker run env1
backing up to /usr/bob/bkup
```

- **Dockerfile with Exec**

```
FROM busybox
ENV dir=/usr/bob/bkup
CMD ["/bin/echo", "backing up to $dir"]
```

- **Build Image and Run**

```
$ docker build -t env2 .
$ docker run env2
backing up to $dir
```

- **Dockerfile Exec with Shell**

```
FROM busybox
ENV dir=/usr/bob/bkup
CMD ["/bin/sh", "-c", "echo backing up to $dir"]
```

- **Build Image and Run**

```
$ docker build -t env3 .
$ docker run env3
backing up to /usr/bob/bkup
```

Shell and Exec Forms

- **Dockerfile with Shell**

```
FROM ubuntu:trusty
CMD sleep 60
```

- **Build Image and Run**

```
$ docker build -t sleep1 .
$ docker run -d sleep1
8bf145a1e783...efbdbac6ad8a02414eefe047d21838aadf7

$ docker ps -l
CONTAINER ID    IMAGE             COMMAND
8bf145a1e783    sleep1    "/bin/sh -c 'sleep 60'"

$ docker exec 8bf145a1e783 ps -f
UID  PID  PPID  C STIME            CMD
root   1    0   0 23:15    /bin/sh -c sleep 60
root   6    1   0 23:15    sleep 60
```

- **Dockerfile with Exec**

```
FROM ubuntu:trusty
CMD ["/bin/sleep", "60"]
```

- **Build Image and Run**

```
$ docker build -t sleep2 .
$ docker run -d sleep2
87cb500a620a...90ce7623f9ffb964adf698712533b0e8913

$ docker ps -l
CONTAINER ID    IMAGE         COMMAND
87cb500a620a    sleep2    "/bin/sleep 60"

$ docker exec 87cb500a620a ps -f
UID  PID  PPID  C STIME         CMD
root   1    0   0 23:15    /bin/sleep 60
```

Dockerfile Instructions

- **USER**
  - When image build starts, user (UID), group (GID) are 0 (`root`)
  - `USER` sets user and group for all instructions that follow
  - May be used with `RUN`, `CMD`, `ENTRYPOINT` instructions
  - Formats: `USER` ***`user`*****`:`*****`group`**
    `USER` ***`UID`*** `[:`***`GID`***`]`
  - Named users and groups must exist in `passwd` file
  - Available to containers running this image
  - Creates zero byte-sized layer in image

```
USER games:games
USER 10:10
```

- **ADD**
  - Copy files and directories to Docker image being built
  - Same format as `COPY`
  - Typically used to extract tar archives
  - Contents of tar file extracted into directory inside image

```
ADD text.tar /text
```

- **ARG**
  - Define variables for customized builds
  - Format: `ARG` ***`var`*** `[=`***`defvalue`***`]`
  - ***`defvalue`*** is optional and may be set or overridden
  - Overridden with `--build-arg` option when building images
  - Images may have more than one ARG
  - *ARG* variables do *not* persist into running containers

```
ARG username="10"
ARG appdir
```

Dockerfile Instructions

- **ONBUILD**
  - Adds trigger instruction to be executed at a later time
  - May be used in another dockerfile with FROM instruction
  - Format: `ONBUILD` ***`instruction`***
  - Any build instruction can be registered as a trigger
  - Triggers inherited by *child* builds only, one time use

```
ONBUILD COPY package.json .
ONBUILD COPY install.sh .
ONBUILD RUN ./install.sh > log
```

- **STOPSIGNAL**
  - Sets signal sent to exit the container
  - Format: `STOPSIGNAL` ***`signal`***
  - ***`signal`*** may be a signal number or valid signal name
  - Used with `docker container stop` command

```
STOPSIGNAL 10
STOPSIGNAL SIGUSR1
```

- **SHELL**
  - Overrides default shell or command for containers
  - Linux default shell: `/bin/sh -c`
  - Windows default command: `cmd /S /C`
  - Format: `SHELL ["`***`executable`***`" "`***`parameters`***`"]`
  - `SHELL` may be used more than once to change shells
  - Possible shells are `bash`, `zsh`, `csh`, `powershell`

```
SHELL ["/bin/bash", "-c"]
SHELL ["powershell", "-command"]
```

Dockerfile Instructions

- **VOLUME**
  - Creates mount point with specified name
  - Holds externally mounted volumes from host or other containers
  - Format: `VOLUME` *`mountpath`* `...`
  - Files in *`mountpath`* *persist* after the life of the container
  - Allows you to *share* data between containers
  - Any changes to data within volume in build steps are discarded
  - Creates zero byte-sized layer in image

```
VOLUME /myvol
VOLUME /var/log /var/db
```

- **Example**

```
$ cat Dockerfile
FROM alpine
RUN mkdir /myvol
RUN echo "this is some text" > /myvol/data
VOLUME /myvol
CMD ["/bin/sh"]

$ docker build -t vol .
$ docker run -it vol
/ # cat /myvol/data
this is some text
/ # exit

$ docker volume ls
DRIVER    VOLUME NAME
local     a2d19de6460e9ee...ced761c5d5f6280b9ed77f1

$ docker volume inspect a2d19de6460e9ee...b9ed77f1
. . .
    "Mountpoint": "/var/lib/docker/volumes/
    a2d19de6460e9ee...ced761c5d5f6280b9ed77f1/_data",
. . .
```

Docker Volumes

- **Data Directory on Host**

```
$ cat data/file.txt
This is line 1.
```

- **Bind Mounts**

```
$ docker run -d --name volc --mount type=bind,
     source="$(pwd)"/data,target=/app
        alpine tail -f /dev/null
```

- **Check Container**

```
$ docker ps
CONTAINER ID  IMAGE     COMMAND       STATUS    NAMES
5bd0eb407287  alpine "tail -f ..."   Up 3 secs   volc

$ docker exec volc cat /app/file.txt
This is line 1.
```

- **Change Data in Container**

```
$ docker exec -it volc sh
/ # ls
app  dev  home   media  opt    root   sbin   sys  usr
bin  etc  lib    mnt    proc   run    srv    tmp  var
/ # cd /app
/ # ls
file.txt
/ # echo "This is line 2." >> file.txt
/ # exit
```

- **Updated Data Directory on Host**

```
$ cat data/file.txt
This is line 1.
This is line 2.
```

Docker Volumes

- **Mapping Files with Volumes**
  - Easier to use `-v` option than `--mount`
  - Format: `docker run -v host:/container`
- **Example**

```
$ docker run -d --name volc1 -v "$(pwd)"/data:/app \
     alpine tail -f /dev/null

$ docker exec volc1 cat /app/file.txt
This is line 1.
This is line 2.
```

- **Data Volume Containers**
  - Volumes may be *shared* between containers
  - Format: `docker run --volumes-from container`
- **Example**

```
$ docker run -d --name volc2 --volumes-from volc1 \
     alpine tail -f /dev/null

$ docker exec volc2 cat /app/file.txt
This is line 1.
This is line 2.
```

- **Updates to Host**

```
$ cat data/file.txt
This is line 1.

$ docker exec volc1 cat /app/file.txt
This is line 1.

$ docker exec volc2 cat /app/file.txt
This is line 1.
```

Dockerfile Instructions

- **EXPOSE**
  - Makes container listen on network port at runtime
  - Documents what port the image expects to be open
  - Formats: `EXPOSE` *`port`*
    `EXPOSE` *`port/protocol`*
  - *`protocol`* may be `tcp` (default) or udp
  - Images may have more than one `EXPOSE`
  - Does *not* make the container ports accessible to host
  - Must use `run -p` or `run -P` to open network ports in container
  - Creates zero byte-sized layer in image

```
EXPOSE 80
EXPOSE 80/udp
```

- **Example**

```
$ cat Dockerfile
# Dockerfile for Apache Web Server
FROM httpd:latest
COPY index.html /usr/local/apache2/htdocs/
EXPOSE 80
```

- **Web Page Source**

```
$ cat index.html
<html>
<body>
<h1>Anderson Software Group, Inc.</h1>
   <h2><a href="https://asgteach.com/"
      title="Anderson Software Group, Inc.">
      https://asgteach.com</a>
   </h2>
</body>
```

Apache Web Server

- **Port Mappings**
  - Port option `-P` creates *random* ports

```
$ docker run --name container -P image
$ docker port container
80/tcp -> 0.0.0.0:32769
```

  - Port option `-p` lets you *name* port

```
$ docker run --name container -p hport:cport image
$ docker port container
cport/tcp -> 0.0.0.0:hport
```

- **Build Apache Image**

```
$ docker build -t apache .
```

- **Run Web Server and Map Ports**

```
$ docker run -d --name webserver -p 1234:80 apache
```

- **Show Port Mappings**

```
$ docker port webserver
80/tcp -> 0.0.0.0:1234
```

- **Show HTML**

```
$ curl localhost:1234
<html>...</html>
```

- **Web Page URL**

```
http://localhost:1234
```

Nginx Web Server

- **Nginx Default Banner**

```
$ docker run -d --name webserver -p 5678:80 nginx
```

- **Web Page URL**

```
http://localhost:5678
```

- **Nginx Custom Banner**

```
$ cat html/index.html
<html>
<body>
<h1>Hello from Docker!</h1>
</body>
</html>
```

- **Run Web Server with Volumes and Map Ports**

```
$ docker run -d --name webserver
      -v "$(pwd)"/html:/usr/share/nginx/html:ro
         -p 5678:80 nginx
```

- **Show Port Mappings**

```
$ docker port webserver
80/tcp -> 0.0.0.0:5678
```

- **Web Page URL**

```
http://localhost:5678
```

Dockerfile Instructions

- **HEALTHCHECK**
  - Tests containers application health
  - Runs command inside container, checks exit status
  - Exit status 0 is healthy, 1 is unhealthy
  - Formats: `HEALTHCHECK` *`options`* `CMD` *`command`*
    `HEALTHCHECK NONE`
  - List of *`options`* with defaults

```
--interval          time between tests (30s)
--timeout           max time for test (30s)
--start-period      no-fail time during startup (0s)
--retries           max number of failures (3)
```

  - Only one `HEALTHCHECK` instruction in Dockerfile

- **Example**

```
HEALTHCHECK --interval=10s --timeout=3s \
   CMD curl --fail http://localhost:1234/ || exit 0
```

- **Build Apache Image with Health Check**

```
$ docker build -t apache .
```

- **Run Web Server and Map Ports**

```
$ docker run -d --name webserver -p 1234:80 apache
```

- **Check Health**

```
$ docker ps
CONTAINER ID IMAGE      COMMAND           STATUS
9454e1146d5e apache "/usr/sbin/httpd" Up 32 sec (healthy)
```

Section 3
More Docker Topics

- **Private Registries**
  - Creating a Local Registry
  - Hostname and Ports
  - Pushing and Pulling Images
  - Managing Registries
- **Multi-Stage Builds**
  - Dockerfile Instructions
  - Reduce Docker Image Sizes
- **Docker Volumes**
  - Management Commands
  - Creating Volumes
  - Inspecting Volumes
  - Mapping and Sharing Files
- **Docker Networks**
  - Management Commands
  - Default Local Networks
  - Creating Networks
  - Connecting to Networks
  - Inspecting Networks
  - Verifying Connections

Private Registries

- **What Are Private Registries?**
  - Instance of `registry` image
  - Repository container, runs within Docker
  - Download container for registry
  - Registry is local
  - Access with port number

- **Creating a Local Registry**

```
$ docker run -d -p 5500:5000 --restart=always \
--name registry registry:latest

$ docker image ls registry
REPOSITORY    TAG      IMAGE ID        SIZE
registry      latest   b8604a3fe854    26.2MB
```

- **Show Registry Running**

```
$ docker ps -a
CONTAINER ID  IMAGE           COMMAND          PORTS
c6afd63ebf42  registry:latest "/entrypoint.sh" 5500->5000/tcp
```

- **Pull Image from Docker Hub**

```
$ docker pull debian:10.4

$ docker image ls debian
REPOSITORY    TAG    IMAGE           SIZE
debian        10.4   ae8514941ea4    114MB
```

- **Tag Image with Hostname and Port**

```
$ docker tag debian:10.4 localhost:5500/my-debian
```

Private Registries

- **Push Image to Local Registry**

```
$ docker push localhost:5500/my-debian
```

- **Remove Locally Cached Images**

```
$ docker rmi debian:10.4

$ docker rmi localhost:5500/my-debian
```

- **Pull Image from Local Registry**

```
$ docker pull localhost:5500/my-debian

$ docker image ls localhost:5500/my-debian
REPOSITORY                  TAG      IMAGE          SIZE
localhost:5500/my-debian    latest   ae8514941ea4   114MB
```

- **Stop Local Registry**

```
$ docker stop registry
```

- **Restart Local Registry**

```
$ docker restart registry
```

- **Cleanup Local Registry**

```
$ docker stop registry

$ docker rm registry

$ docker rmi registry:latest
```

Multi-Stage Builds

- **Why Reduce Docker Image Size?**
  - Keeps only required artifacts in final image
  - Removes unnecessary data
  - Downloads and runs containers faster
  - Faster to push images to registries
- **Best Practices**
  - Use `.dockerignore` to remove content not needed from builds
  - Avoid installing package and dependencies you don't need
  - Keep layers to a minimum
  - Use `alpine` images wherever possible
  - Use *multi-stage* builds to reduce image size, improve performance, make Dockerfiles easy to read and understand
- **Multi-Stage Builds**
  - Divides Dockerfile into multiple stages to pass artifacts from one stage to another
  - Avoids shell scripts and cleanup instructions
  - Uses multiple `FROM` instructions
- **Dockerfile Instructions**

```
FROM image1 AS mystage1
# create artifact1
FROM image2 AS mystage2
# create artifact2
FROM image3
. . .
COPY --from=mystage1 artifact1 .
COPY --from=mystage2 artifact2 .
. . .
```

Build Examples

- **Single Stage Build**

```
$ cat Dockerfile
FROM ubuntu:latest
RUN apt-get update && apt-get install -y gcc

WORKDIR /root
COPY greeting.c .
RUN gcc greeting.c -o greeting
CMD ["./greeting"]

$ docker build -t myub1 .

$ docker image ls myub1
REPOSITORY  TAG      IMAGE ID      CREATED       SIZE
myub1       latest   8bd897a6144c  3 weeks ago   281MB
```

- **Multi-Stage Build**

```
$ cat Dockerfile
FROM ubuntu:latest AS compile-image
RUN apt-get update && apt-get install -y gcc

WORKDIR /root
COPY greeting.c .
RUN gcc greeting.c -o greeting

FROM ubuntu:latest AS runtime-image
COPY --from=compile-image /root/greeting .
CMD ["./greeting"]

$ docker build -t myub2 .

$ docker image ls myub2
REPOSITORY  TAG      IMAGE ID      CREATED       SIZE
myub2       latest   b351671a7d41  3 weeks ago   77.8MB
```

Docker Volumes

- **What are Docker Volumes?**
  - Persistent storage location on host
  - Mounted for access inside running container
  - May be modified by both host and container
  - Can be accessed by more than one container
  - Volumes have *drivers*, default is local host
  - Volume drivers may be remote hosts or cloud providers
- **Creating Volumes**
  - Command: `docker volume create [OPTIONS] [VOLUME]`
  - Options (default is "`local`" for driver name)

```
-d, --driver string       # driver name
--label list              # volume metadata
-o, --opt map             # driver options
```

- **Create Volume with Random Name**

```
$ docker volume create
3f54949e70f7...2d4e27653a29638bfbaa11a208

$ docker volume ls
DRIVER      VOLUME NAME
local       3f54949e70f7...2d4e27653a29638bfbaa11a208
```

- **Create Named Volume**

```
$ docker volume create myvol
myvol

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

Docker Volumes

- **Inspect Volume**

```
$ docker volume inspect myvol
[
  {
    "CreatedAt": "2021-11-20T18:24:11Z",
    "Driver": "local",
    "Labels": {},
    "Mountpoint": "/var/lib/docker/volumes/myvol/_data",
    "Name": "myvol",
    "Options": {},
    "Scope": "local"
  }
]
```

- **Volume Mount Point**

```
# ls -l /var/lib/docker/volumes/myvol
total 4
drwxr-xr-x 2 root root 4096 Nov 20 18:24 _data

# ls /var/lib/docker/volumes/myvol/_data
#
```

- **Mapping Files with Volumes**

```
# cat file.txt
This is line 1.
This is line 2.

# cp file.txt /var/lib/docker/volumes/myvol/_data

# cat /var/lib/docker/volumes/myvol/_data/file.txt
This is line 1.
This is line 2.
```

Docker Volumes

- **File Access in Container 1**

```
$ docker run -d --name volc1 -v myvol:/app \
     alpine tail -f /dev/null

$ docker exec volc1 cat /app/file.txt
This is line 1.
This is line 2.
```

- **Share File in Container 2**

```
$ docker run -d --name volc2 --volumes-from volc1 \
    alpine tail -f /dev/null

$ docker exec volc2 cat /app/file.txt
This is line 1.
This is line 2.
```

- **Update File in Container 2**

```
$ docker exec -it volc2 sh
/ # echo This is line 3. >> /app/file.txt
/ # cat /app/file.txt
This is line 1.
This is line 2.
This is line 3.
/ # exit
```

- **Container 1 and Host Sharing**

```
$ docker exec volc1 cat /app/file.txt
This is line 1.
This is line 2.
This is line 3.

# cat /var/lib/docker/volumes/myvol/_data/file.txt
This is line 1.
This is line 2.
This is line 3.
```

Docker Networks

- **What are Docker Networks?**
  - Collection of connected endpoints
  - Allows communication between connections
  - Containers use network to communicate with each other
  - On same local host, remote hosts, or outside systems
  - Includes sandbox, endpoints, network interfaces
  - Has a name, address, ID, and network type
  - Networks have *drivers*, default is local host
  - Network drivers may be local or remote

- **Default Local Docker Networks**

```
$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
a25d1e64dcb7   bridge    bridge    local
88455cd35b8a   host      host      local
6da33a989cef   none      null      local
```

- **Show Network Drivers**

```
$ docker info
Client:
 Debug Mode: false
 . . .
Server:
 . . .
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  . . .
 Swarm: inactive
 . . .
```

Docker Networks

- **None Network**
  - Uses `null` driver
  - Does *not* configure network interfaces inside container
  - Containers are isolated, only a loopback is created
  - Loopback bound to `localhost` address (`127.0.0.1`)
  - Does *not* configure any IP address for containers
  - No access to external networks or other containers
  - Each Docker host can only have *one* network using this driver
  - Use this to *disable* networking on containers
- **Why Use None?**
  - Security and backward compatibility
  - Volume containers, backups, batch jobs, diagnostic tools
- **Run Container on None Network**

```
$ docker run -d --name mynone --network none \
     alpine tail -f /dev/null
c47a11d734f5...b22b229d8e3bd0a4e8488

$ docker ps -a
CONTAINER ID    IMAGE    COMMAND                NAMES
c47a11d734f5    alpine   "tail -f /dev/null     mynone

$ docker exec mynone ifconfig
lo    Link encap:Local Loopback
      inet addr:127.0.0.1  Mask:255.0.0.0
      UP LOOPBACK RUNNING  MTU:65536  Metric:1
      RX packets:0 errors:0 dropped:0 overruns:0 frame:0
      TX packets:0 errors:0 dropped:0
      overruns:0 carrier:0
      collisions:0 txqueuelen:1000
      RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Docker Networks

- **Inspect None Network**

```
$ docker network inspect none
[
    {
        "Name": "none",
        "Id": "6da33a989cef...61aba4318b10784e76d1eb8",
        "Created": "2020-03-01T12:13:43.617113901Z",
        "Scope": "local",
        "Driver": "null",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": []
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "c47a11d734f5...b22b229d8e3bd0a4e8488": {
                "Name": "mynone",
               "EndpointID": "f8aa88174c22...37363f5b5",
                "MacAddress": "",
                "IPv4Address": "",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

Docker Networks

- **Host Network**
  - Uses `host` driver
  - Removes network isolation between containers and host
  - Containers do *not* have their own IP addresses
  - Containers only communicate through the host interface
  - Each port is only available to only *one* container
  - Each Docker host can only have *one* network using this driver
  - Containers *share* the host network stack (ports, etc.)
  - All host interfaces are available to containers
  - Network performance essentially native but not secure!
- **Why Use Host?**
  - Containers running high throughput applications
  - Containers are in host namespace, so use sparingly!
- **Run Container on Host Network**

```
$ docker run -d --name myhost --network host \
     alpine tail -f /dev/null
7ac2448a107b...7c6819ff13e13ed034ba3

$ docker ps -a
CONTAINER ID    IMAGE    COMMAND                NAMES
7ac2448a107b    alpine   "tail -f /dev/null     myhost

$ docker exec myhost ifconfig
docker0  Link encap:Ethernet   HWaddr 02:42:FC:E5:3D:2C
         inet addr:172.18.0.1  Bcast:172.18.0.255
         Mask:255.255.255.0
         inet6 addr: fe80::42:fcff:fee5:3d2c/64
         Scope:Link
         UP BROADCAST RUNNING MULTICAST
         MTU:1500  Metric:1
```

Docker Networks

- **Inspect Local Host Network**

```
$ docker network inspect host
[
    {
        "Name": "host",
        "Id": "88455cd35b8a...8d845e2ed4063f12747e9",
        "Created": "2020-03-01T12:13:43.664425732Z",
        "Scope": "local",
        "Driver": "host",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": []
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "7ac2448a107b...7c6819ff13e13ed034ba3": {
                "Name": "myhost",
                "EndpointID": "67fa39852b20..c0540d749",
                "MacAddress": "",
                "IPv4Address": "",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

Docker Networks

- **Bridge Network**
  - Uses `bridge` driver
  - All attached containers on network are able to communicate
  - Containers have access to host interfaces
  - Each container is assigned its own IP address
  - Docker host may have more than one network with this driver
  - Containers in one bridge network are *not* accessible to others
  - Default for new containers, most common network
- **Why Use Bridge?**
  - Container processes need network access
  - Containers need to communicate with each other
- **Run Container on Host Network**

```
$ docker run -d --name mybridge alpine tail -f /dev/null
86df478b4f9c...ab1ff670c1ac8dfa5860be

$ docker ps -a
CONTAINER ID    IMAGE    COMMAND                NAMES
86df478b4f9c    alpine   "tail -f /dev/null     mybridge

$ docker exec mybridge ifconfig
eth0    Link encap:Ethernet  HWaddr 02:42:AC:12:00:02
        inet addr:172.18.0.2  Bcast:172.18.0.255
        Mask:255.255.255.0
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:54 errors:0 dropped:0
        overruns:0 frame:0
        TX packets:0 errors:0 dropped:0
        overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:6928 (6.7 KiB)  TX bytes:0 (0.0 B)
        . . .
```

Docker Networks

- **Inspect Local Bridge Network**

```
$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "a6b80153eb2b...ae2db7225ce36fbec5b0f9b",
        . . .
        "Containers": {
            "86df478b4f9c...ab1ff670c1ac8dfa5860be": {
                "Name": "mybridge",
                "EndpointID": "b75942bd8d1b...6ba7aa8b",
                "MacAddress": "02:42:ac:12:00:03",
                "IPv4Address": "172.18.0.3/24",
                "IPv6Address": ""
            }
        },
        . . .
    }
]
```

- **Verify Connections**

```
$ ping -c3 172.18.0.3       # host to container
PING 172.18.0.3 (172.18.0.3) 56(84) bytes of data.
64 bytes from 172.18.0.3: icmp_seq=1 ttl=64 time=0.094 ms
64 bytes from 172.18.0.3: icmp_seq=2 ttl=64 time=0.067 ms
64 bytes from 172.18.0.3: icmp_seq=3 ttl=64 time=0.076 ms

$ docker exec -it mybridge sh
/ # ping -c3 python.org     # container to outside
PING python.org (138.197.63.241): 56 data bytes
64 bytes from 138.197.63.241: seq=0 ttl=51 time=89.214 ms
64 bytes from 138.197.63.241: seq=1 ttl=51 time=88.296 ms
64 bytes from 138.197.63.241: seq=2 ttl=51 time=87.986 ms
. . .
/ # exit
$
```

Docker Networks

- **Creating Networks**
  - Command: `docker network create [OPTIONS] [NETWORK]`
  - Options (default is "bridge" for driver name)

```
-d, --driver string    # driver name
--label list           # network metadata
-o, --opt map          # driver options
--subnet strings       # network subnet
--ip-range strings     # allocate container ip
--gateway strings      # IPv4 or IPv6
```

- **Create New Bridge Network**

```
$ docker network create myNetwork
1881d1bad7bb...575bbeb59672056fda6886e3341ed35e
```

- **Show Networks**

```
$ docker network ls
NETWORK ID     NAME        DRIVER    SCOPE
a25d1e64dcb7   bridge      bridge    local
88455cd35b8a   host        host      local
1881d1bad7bb   myNetwork   bridge    local
6da33a989cef   none        null      local
```

- **Create Containers**

```
$ docker run -d --name mybridge1 alpine tail -f /dev/null
801a6b714102...beeb472f5a605cc72ba23413bcfb5f3

$ docker run -d --name mybridge2 alpine tail -f /dev/null
42e996e721c5...c9147b9111f1f82ba43a3a03d4d01e0

$ docker ps -a
CONTAINER ID   IMAGE    COMMAND                  NAMES
42e996e721c5   alpine   "tail -f /dev/null"      mybridge2
801a6b714102   alpine   "tail -f /dev/null"      mybridge1
```

Docker Networks

- **Connect Containers to Network**

```
$ docker network connect myNetwork mybridge1

$ docker network connect myNetwork mybridge2
```

- **Container to Container**

```
$ docker exec mybridge1 ping -c3 mybridge2
PING mybridge2 (172.19.0.3): 56 data bytes
64 bytes from 172.19.0.3: seq=0 ttl=64 time=0.064 ms
64 bytes from 172.19.0.3: seq=1 ttl=64 time=0.098 ms
64 bytes from 172.19.0.3: seq=2 ttl=64 time=0.134 ms

$ docker exec mybridge2 ping -c3 mybridge1
PING mybridge1 (172.19.0.2): 56 data bytes
64 bytes from 172.19.0.2: seq=0 ttl=64 time=0.088 ms
64 bytes from 172.19.0.2: seq=1 ttl=64 time=0.241 ms
64 bytes from 172.19.0.2: seq=2 ttl=64 time=0.151 ms
```

- **Container to Outside**

```
$ docker exec mybridge1 ping -c3 python.org
PING python.org (138.197.63.241): 56 data bytes
64 bytes from 138.197.63.241: seq=0 ttl=51 time=88.209 ms
64 bytes from 138.197.63.241: seq=1 ttl=51 time=88.391 ms
64 bytes from 138.197.63.241: seq=2 ttl=51 time=88.139 ms

$ docker exec -it mybridge2 sh
/ # ping -c3 python.org
PING python.org (138.197.63.241): 56 data bytes
64 bytes from 138.197.63.241: seq=0 ttl=51 time=89.214 ms
64 bytes from 138.197.63.241: seq=1 ttl=51 time=88.296 ms
64 bytes from 138.197.63.241: seq=2 ttl=51 time=87.986 ms
. . .
/ # exit
$
```

Docker Networks

- **Inspect Bridge Network**

```
$ docker network inspect myNetwork
[
    {
        "Name": "myNetwork",
        "Id": "1881d1bad7bb...72056fda6886e3341ed35e",
        "Created": "2021-11-21T22:58:54.114938578Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.19.0.0/16",
                    "Gateway": "172.19.0.1"
                }
            ]
        },
        . . .
        "Containers": {
            "42e996e721c5...111f1f82ba43a3a03d4d01e0": {
                "Name": "mybridge2",
                "EndpointID": "3d5e7f60ef8b...2b6ba95a",
                "MacAddress": "02:42:ac:13:00:03",
                "IPv4Address": "172.19.0.3/16",
                "IPv6Address": ""
            },
            "801a6b714102...f5a605cc72ba23413bcfb5f3": {
                "Name": "mybridge1",
                "EndpointID": "18562f54ffa6...04f4dc912",
                "MacAddress": "02:42:ac:13:00:02",
                "IPv4Address": "172.19.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

Docker Networks

- **Host to Container Connections**

```
$ ping -c3 172.19.0.2       # mybridge1 container
PING 172.19.0.2 (172.19.0.2) 56(84) bytes of data.
64 bytes from 172.19.0.2: icmp_seq=1 ttl=64 time=0.098 ms
64 bytes from 172.19.0.2: icmp_seq=2 ttl=64 time=0.080 ms
64 bytes from 172.19.0.2: icmp_seq=3 ttl=64 time=0.077 ms

$ ping -c3 172.19.0.3       # mybridge2 container
PING 172.19.0.3 (172.19.0.3) 56(84) bytes of data.
64 bytes from 172.19.0.3: icmp_seq=1 ttl=64 time=0.098 ms
64 bytes from 172.19.0.3: icmp_seq=2 ttl=64 time=0.080 ms
64 bytes from 172.19.0.3: icmp_seq=3 ttl=64 time=0.077 ms
```

- **Disconnect from Network**

```
$ docker network disconnect myNetwork mybridge1
$ docker network disconnect myNetwork mybridge2
```

- **Inspect Bridge Network**

```
$ docker network inspect myNetwork
[
    {
        "Name": "myNetwork",
        "Id": "1881d1bad7bb...72056fda6886e3341ed35e",
        "Created": "2021-11-21T22:58:54.114938578Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {} . . .
```

Section 4
Docker Compose

- **Overview**
  - What is Docker Compose?
  - Why Use Docker Compose?
  - What Does Docker Compose Do?
  - How Do You Use Docker Compose?
- **Services**
  - Create and Run Services
  - Pull Images and Build Services
  - Port Services
  - Service Lifecycles
  - Service Environment Variables
- **Volumes**
  - Volume Services
  - Mount Points
- **Networks**
  - Multiple Services
  - Network Services
  - Network Behaviors

Overview

- **What is Docker Compose?**
  - Configuration tool for Docker
  - Automated multi-container workflow
  - Applies rules declared in a YAML file
  - Almost every rule replaces a specific Docker command
  - Helps define, launch, and manage *services*
- **Why Use Docker Compose?**
  - Streamlines development tasks
  - Avoids shell scripting for configurations
  - Lets you stop focusing on individual containers
  - Describes full environments with interacting service components
- **What Does Docker Compose Do?**
  - Builds Docker images
  - Runs container applications as services
  - Launches full systems of services
  - Manages individual services
  - Scales services up or down
  - Views logs for containers in services
- **How Do You Use Docker Compose?**
  - Define app environment in a `Dockerfile`
  - Define app services in `docker-compose.yaml`
  - Start and run app with `docker-compose up`

Services

- **Configuration File Format**

```
$ cat docker-compose.yaml
version: "3"
services:
  . . .
volumes:
  . . .
networks:
```

- **Create and Run a Service**

```
$ cat docker-compose.yaml
version: '3'
services:
  hello:
    image: alpine:latest
    command: [/bin/echo, 'Hello from alpine!']

$ docker-compose up
Creating network "greeting_default" with default driver
Creating greeting_hello_1 ...
Creating greeting_hello_1 ... done
Attaching to greeting_hello_1
hello_1  | Hello from alpine!
greeting_hello_1 exited with code 0
```

- **Run Service Again**

```
$ docker-compose up
Starting greeting_hello_1 ...
Starting greeting_hello_1 ... done
Attaching to greeting_hello_1
hello_1  | Hello from alpine!
greeting_hello_1 exited with code 0
```

Services

- **Pull Image and Run Service**

```
$ cat docker-compose.yaml
version: '3'
services:
  hello:
    image: busybox
    command: [/bin/echo, 'Hello from busybox!']

$ docker-compose up
Pulling hello (busybox:latest)...
latest: Pulling from library/busybox
e685c5c858e3: Pull complete
Digest: sha256:e7157b6d7ebb...9b9f90a9ec42e57323546c353
Status: Downloaded newer image for busybox:latest
Creating pullimage_hello_1 ... done
Attaching to pullimage_hello_1
hello_1  | Hello from busybox!
pullimage_hello_1 exited with code 0
```

- **Run Service Again**

```
$ docker-compose up
Starting pullimage_hello_1 ... done
Attaching to pullimage_hello_1
hello_1  | Hello from busybox!
pullimage_hello_1 exited with code 0
```

- **Show Container**

```
$ docker ps -a
CONTAINER ID  IMAGE    COMMAND       NAMES
20e6c268afa3  busybox  "/bin/echo…"  pullimage_hello_1
```

Services

- **Build Image and Run Service**

```
$ cat Dockerfile
FROM ubuntu:latest
RUN apt-get update && apt-get install -y golang
COPY echob.go .

$ cat docker-compose.yaml
version: '3'
services:
  mygo:
    build: .
    image: mygo
    command: [go,'run','echob.go','one','two','three']

$ docker-compose build
Building mygo
Step 1/3 : FROM ubuntu:latest
 ---> 13b66b487594
Step 2/3 : RUN apt-get update && apt-get install -y
golang
 ---> Using cache
 ---> 9038345c1ddb
Step 3/3 : COPY echob.go .
 ---> Using cache
 ---> 51d1e3e68f92
Successfully built 51d1e3e68f92
Successfully tagged mygo:latest

$ docker image ls mygo
REPOSITORY  TAG       IMAGE ID       SIZE
mygo        latest    51d1e3e68f92   881MB

$ docker-compose up
Starting build_mygo_1 ...
Starting build_mygo_1 ... done
Attaching to build_mygo_1
mygo_1  | three two one
build_mygo_1 exited with code 0
```

Services

- **Build Image with Ports**

```
$ cat Dockerfile
# Dockerfile for Apache Web Server
FROM httpd:latest
COPY index.html /usr/local/apache2/htdocs/
EXPOSE 80

$ cat docker-compose.yaml
version: '3'
services:
  myports:
    build: .
    image: myports
    ports:
      - "1234:80"

$ cat index.html
<html>
<body>
<h1>Anderson Software Group, Inc.</h1>
    <h2><a href="https://asgteach.com/"
        title="Anderson Software Group, Inc.">
        https://asgteach.com</a>
    </h2>
</body>
</html>
```

- **Run Service**

```
$ docker-compose up -d
Starting ports_myports_1 ...
Starting ports_myports_1 ... done

$ curl localhost:1234
<html>...</html>

$ docker-compose stop
Stopping ports_myports_1 ... done
```

Service Lifecycles

- **Startup**
  - Build image, pull if necessary
  - Start containers, volumes, networks

```
$ docker-compose up
$ docker-compose up -d
```

  - Start again after first time

```
$ docker-compose start
```

- **Shutdown**
  - Stop active services
  - Preserve containers, volumes, networks

```
$ docker-compose stop
```

  - Reset active services
  - Release containers, networks
  - Preserve external volumes

```
$ docker-compose down
```

- **Restart**
  - Restarts all stopped and running services
  - Changes to `docker-compose.yaml` are *not* seen
  - Changes to environment variables are *not* updated

```
$ docker-compose restart
```

Environment Variables

- **Static Environment Variable**

```
$ cat docker-compose.yaml
version: '3'
services:
  webserver:
    image: nginx
    environment:
      - API_KEY=8a53m7x02p58d8s1
    ports:
      - "8001:80"

$ docker-compose up -d
Creating network "env1_default" with the default driver
Creating env1_webserver_1 ...
Creating env1_webserver_1 ... done

$ docker exec env1_webserver_1 env | grep API_KEY
API_KEY=8a53m7x02p58d8s1
```

- **Dynamic Environment Variable**

```
$ cat docker-compose.yaml
version: '3'
services:
  webserver:
    image: nginx
    environment:
      - API_KEY=${KEY}
    ports:
      - "8002:80"

$ cat .env
KEY=8a53m7x02p58d8s1

$ docker-compose up -d
Creating env2_webserver_1 ...
Creating env2_webserver_1 ... done

$ docker exec env2_webserver_1 env | grep API_KEY
API_KEY=8a53m7x02p58d8s1
```

Volumes

- **Build Image with Volumes**

```
$ cat Dockerfile
# Dockerfile for Apache Web Server
FROM httpd:latest
COPY index.html /usr/local/apache2/htdocs/
EXPOSE 80

$ cat docker-compose.yaml
version: '3'
services:
  myvols:
    build: .
    image: myvols
    ports:
      - "1234:80"
    volumes:
      - ./index.html:/usr/local/apache2/htdocs/index.html
```

- **Run Service**

```
$ docker-compose up -d
Creating network "volumes_default" with default driver
Creating volumes_myvols_1 ...
Creating volumes_myvols_1 ... done

$ curl localhost:1234
<html>...</html>

# make mods to index.html
```

- **Restart Service**

```
$ docker-compose restart
Restarting volumes_myvols_1 ... done
```

Networks

- **Services with Networks**

```
$ cat docker-compose.yaml
version: '3'
services:
  web1:
    image: nginx
    ports:
      - "8001:80"
    networks:
      - myPublic
  myapp:
    image: busybox
    networks:
      - myPublic
    command: [tail, '-f', '/dev/null']
  mypgm:
    image: alpine
    networks:
      - myPrivate
    command: [tail, '-f', '/dev/null']
networks:
  myPublic: {}
  myPrivate: {}

$ docker-compose up -d
Creating network "networks_myPublic" with default driver
Creating network "networks_myPrivate" with default driver
Creating networks_web1_1 ...
Creating networks_mypgm_1 ...
Creating networks_myapp_1 ...
Creating networks_mypgm_1
Creating networks_web1_1
Creating networks_myapp_1 ... done

$ docker-compose ps
Name                     Command       State    Ports
-------------------------------------------------------
networks_myapp_1   tail -f /dev/null  Up
networks_mypgm_1   tail -f /dev/null  Up
networks_web1_1    /docker... nginx   Up   8001->80/tcp
```

Networks

- **Services with Networks**

```
$ docker network ls
NETWORK ID     NAME                    DRIVER    SCOPE
2b8eeec4b2eb   bridge                  bridge    local
88455cd35b8a   host                    host      local
91c6fff25575   networks_myPrivate      bridge    local
0fa7323aac9b   networks_myPublic       bridge    local
6da33a989cef   none                    null      local

$ curl localhost:8001
<html>
. . .
<title>Welcome to nginx!</title>
. . .
</html>
```

- **Network Behaviors**
  - Service `myapp` can ping `web1`, same network
  - Service `myapp` *cannot* ping `mypgm`, different network
  - Service `mypgm` *cannot* ping `web1` or `myapp`, different network

```
$ docker-compose exec myapp sh
/ # ping -c3 web1
PING web1 (172.19.0.3): 56 data bytes
64 bytes from 172.19.0.3: seq=0 ttl=64 time=0.266 ms
64 bytes from 172.19.0.3: seq=1 ttl=64 time=0.155 ms
64 bytes from 172.19.0.3: seq=2 ttl=64 time=0.119 ms

/ # ping -c3 mypgm
ping: bad address 'mypgm'
/ # exit

$ docker-compose exec mypgm sh
/ # ping -c3 web1
ping: bad address 'web1'
/ # ping -c3 myapp
ping: bad address 'myapp'
/ # ping -c3 python.org
```

Multiple Services

- **Cooperating Services**

```
services:
  service1:
    image: myapp1
    networks:
      - my-public-network
    ...
  service2:
    image: nginx:latest
    networks:
      - my-public-network
    ...
  service3:
    image: alpine:latest
    networks:
      - my-private-network
    ...
networks:
  my-public-network: {}
  my-private-network: {}
```

- **Example Web App**

```
services:
  front-tier:
    image: my-nodejs-app
    ...
  back-tier:
    image: my-springboot-app
    ...
  db:
    image: postgres
    ...
```

Docker
Bibliography

Docker Quick Start Guide

Waud, Earl
Packt, 2018
ISBN 1-78-934732-7

Docker Up & Running

Kane, Sean P. and Karl Matthias
O'Reilly, 2018
ISBN 1-49-203673-0

Docker in Practice, Second Edition

Miell, Ian and Aidan Hobson Sayers
Manning, 2019
ISBN 1-61-729480-2

Docker in Action, Second Edition

Nickoloff, Jeff and Stephen Kuenzil
Manning, 2019
ISBN 1-61-729476-4

Mastering Docker, Fourth Edition

McKendrick, Russ
Packt, 2020
ISBN 1-83-921657-3

Docker for Developers

Bullington-McGuire, Dennis, Schwartz
Packt, 2020
ISBN 1-78-953605-7

Docker Fundamentals

Exercises

Docker Overview

1. In this lab you will use Docker to create your own custom container for software development using Python and Go in your **exer/mygo** directory. Here are the steps:
   - Pull the `ubuntu` image from the Docker Hub.
   - Name your container `myubc` and run the `ubuntu` image with no detachment.
   - In the bash shell of your container, run these commands.

     ```
     # mkdir mydev
     # cd mydev
     # apt-get update
     ```

   - Install the `vim` editor with this command.

     ```
     # apt-get install vim
     ```

   - Install Python and Go with these commands.

     ```
     # apt-get install python3
     # apt-get install golang
     ```

   - Verify that all languages are installed.

     ```
     # python3 --version
     # go version
     ```

   - Now, exit your container and return to the host.

     ```
     # exit
     ```

   - Use docker commands to perform the following.
     - Copy the **echob.py** program in your **exer** directory to the **mydev** directory inside your container.
     - Do the same for the **echob.go** program.
     - Run your container again and return to the bash shell inside the container.
   - Run these commands.

     ```
     # cd mydev
     # ls
     ```

   - Verify the programs you just copied from the host are in the container's **mydev** directory.

- In the bash shell of your container, run the following commands to execute your Python and Go programs.

```
# python3 echob.py one two three
three two one

# go run echob.go one two three
three two one
```

- Exit your container and return to the host.

```
# exit
```

- Use docker commands to perform the following.
  - Save your `myubc` container to a named image called `myubi`.
  - Save your `myubi` image to a tar archive called **myubi.tar**.
  - Remove the `myubc` container and `myubi` image.
  - Load your customized image from the tar archive.
  - Verify that the `myubi` image has been loaded.
  - Run a container with the restored image and enter the bash shell.
- In the bash shell of your container, verify that you can compile and run all the language programs.
- Exit your container and return to the host.

```
# exit
```

- Using docker, remove the `myubc` container and the `myubi` image.

Working with Docker

1. In this lab you will write a Dockerfile for an image that runs a Python and Go program. Using your editor, edit the **Dockerfile** in your **exer/mygo** directory. Add Dockerfile instructions to perform the following steps.
   - Use `ubuntu:latest` as your base image.
   - Use `apt-get update` and `apt-get install` to install `python3` and `golang` in your image.
   - Make **mydev** your working directory.
   - Copy **echob.py** and **echob.go** in your **exer/mygo** directory to the **mydev** directory inside your container.
   - Use `chmod +x` to make **echob.py** executable.
   - Make `/bin/bash` your default command.
   - Save the **Dockerfile** and exit your editor.

   Now do these steps to build your image and run it in a container.
   - Use `docker build` to build an image called **mygo**.
   - Run the **mygo** image in a container interactively.
   - In the bash shell of your container, verify the **echob.py** and **echob.go** programs are in the container's **mydev** directory.
   - Check the permissions for **echob.py** and make sure it's executable.
   - Run the following commands to execute your Python and Go programs.

     ```
     # ./echob.py one two three
     three two one

     # go run echob.go one two three
     three two one
     ```

   - Exit your container and return to the host.

   Bring up the **Dockerfile** again in your editor and make these changes.
   - Use `ENTRYPOINT` with `go run` and **echob.go**. Replace `CMD` with the program's arg's.
   - Save the **Dockerfile** and exit your editor.
   - Rebuild your image.

   Test your new **mygo** image with the following commands.

   ```
   $ docker run mygo
   three two one

   $ docker run mygo uno dos tres
   tres dos uno
   ```

More Docker Topics

1. In this lab you will modify a Dockerfile to create a smaller image using a multi-stage build. The image built is a Go program that shows a line of text on port 8080 of a web page.

   Here's the Go program in your **exer/msbuild** directory.

```
// hello.go - Go program with net library
package main

import (
   "fmt"
   "net/http"
)

func main() {
   http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
       fmt.Fprintf(w, "Hello from Docker! ")
       fmt.Fprintf(w, "Your request path is: %s\n", r.URL.Path)
   })
   http.ListenAndServe(":8080", nil)
}
```

   Here is the **Dockerfile** in your **exer/msbuild** directory. Do you see what it does?

```
# Dockerfile - Single Stage
FROM golang:alpine
COPY hello.go /app/
WORKDIR /app
ENV CGO_ENABLED=0
RUN go build hello.go
EXPOSE 8080
CMD ["./hello"]
```

   Use this **Dockerfile** to perform the following steps.

   - Build the `hello1` image.

```
$ docker build -t hello1 .
```

   - Run the image in a detached `goserver` container with ports 8080 mapped.

```
$ docker run --name goserver -d -p 8080:8080 hello1
```

   - Use `curl` with `localhost` and port 8080 to see the output from the Go program.
   - View the program output in your browser with `localhost` and port 8080.
   - Now run this command to see the size of the image you built.

```
$ docker image ls hello1
```

   Why is this image so large?

   Can you make it smaller with a multi-stage build in this **Dockerfile**?

Using your editor, use these steps to modify the **Dockerfile** for a multi-stage build.

- Use `AS` with the `FROM` instruction to create a stage called `builder`.
- At the end of the **Dockerfile**, use `FROM` to load the latest `alpine` image.
- Use `COPY` to copy the **hello** executable from the `builder` stage to this stage.
- Use `CMD` to run the **hello** executable in the container.
- Save the **Dockerfile** and exit your editor.

Now do these steps.

- Stop the `goserver` container and remove it.
- Build a new image called `hello2`.
- Run the `hello2` image in detached container `goserver` with ports 8080 mapped.
- Bring up your browser with `localhost` or your Machine IP with port 8080 and verify that the **hello** program is working.
- Run this command to see the size of the `hello2` image you just built.

```
$ docker image ls hello2
```

Why is this image so much smaller?

Extra Credit: Can you make this image even smaller in size?

2. In this lab you will use Docker to create a volume and network for two `nginx` web server containers that share a custom banner.

   Examine the custom banner in your **exer/mynetwork** directory.

   ```
   $ cat html/index.html
   <html>
   <body>
   <h1>Hello from Docker!</h1>
   </body>
   </html>
   ```

   Here are the steps to create the Docker volume and run the web server containers.

   - Create a volume called **myvol** and verify that it was created.
   - Use this command to create a docker container to store the custom banner.

     ```
     $ docker create --name myctr -v myvol:/root alpine
     ```

   - Use this command to copy the banner directory to the docker container.

     ```
     $ docker cp html myctr:/root/html
     ```

   - Use the following command to run a web container with the `nginx` web server. Do you see what this command does?

     ```
     $ docker run -d --name web1 \
     -v myvol:/usr/share/nginx/:ro -p 8001:80 nginx
     ```

   - Use the following command to run a second web container with `nginx`. Note this container shares the same mount point as the other web container.

     ```
     $ docker run -d --name web2 \
     -v myvol:/usr/share/nginx/:ro -p 8002:80 nginx
     ```

   - Verify the containers are up and running.
   - Show the ports for the `web1` and `web2` containers.
   - Use `curl` commands with `localhost` and port numbers to access the banner from both `nginx` containers.

   Here are the steps to create a Docker network and connect to the web server containers.

   - Create a network called **myNetwork** and verify that it was created.
   - Connect the `web1` and `web2` containers to this network.
   - Inspect **myNetwork**. Do you see the two web containers in the output?
   - Use `curl` and the container IP addresses to access the banner.
   - Cleanup the containers, volume, and network. What order should you do this?

Docker Compose

1. In this lab you will use Docker Compose to deploy two `apache` web containers that share a custom banner and belong to the same custom network.

   Examine these files in your **exer/mycompose** directory.

```
$ cat Dockerfile
# Dockerfile for Apache Web Server
FROM httpd:latest
COPY index.html /usr/local/apache2/htdocs/
EXPOSE 80

$ cat docker-compose.yaml
version: '3'
services:
 web1:
   build: .
   image: web1
   ports:
     - "8001:80"
   volumes:
     - ./index.html:/usr/local/apache2/htdocs/index.html

$ cat index.html
<html>
<body>
<h1>Hello from Docker!</h1>
</body>
</html>
```

   The **Dockerfile** builds an `apache` web server image that contains a custom banner in `index.html` and exposes port `80`. The docker compose file launches a `web1` service with a `volumes` mount point for the banner.

   First, perform these steps to launch the `web1` service and interact with its exposed port.

   - Run the **docker-compose up -d** command to build the service and launch it.
   - Run the **docker-compose ps** command to see the service running.
   - Use **curl localhost:8001** to see the custom banner.
   - Run the **docker-compose down** command to take down the `web1` service.

   Now, modify the **docker-compose.yaml** file with your editor. Here is what you need to do.

   - Add a second service that builds `web2` with ports mapped as `"8002:80"`.
   - For the `web2` service, include the same `volumes` element as `web1`.
   - Inside the `web1` and `web2` services, add a `networks` element that names `myNetwork`.
   - Outside the `web1` and `web2` services, add a `networks` element for `myNetwork`.
   - Save your changes and exit the editor.
   - Now run the above commands again. Use **curl** to see the custom banner in `web2`.