Docker

Basic Docker Components

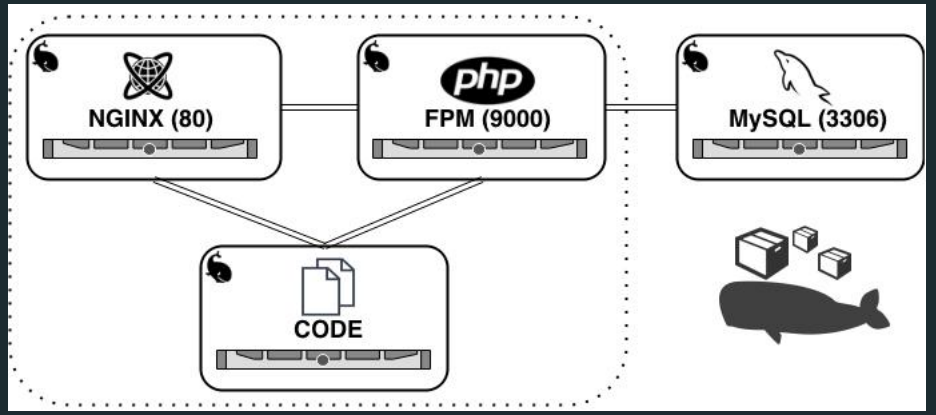
Courtesy John Willis

Today

- Introduction to containers
- Installing Docker
- Docker concepts and terms
- Introduction to images
- Running and managing containers
- Building images
- Container volumes
- Container networking

Workshop

- Docker Setup
- Docker RUN
- Docker Compose



Module 1:
Introduction to Containers

Module objectives

In this module we will:

- Introduce the concept of container based virtualization
- Outline the benefits of containers over virtual machines

What is Docker?

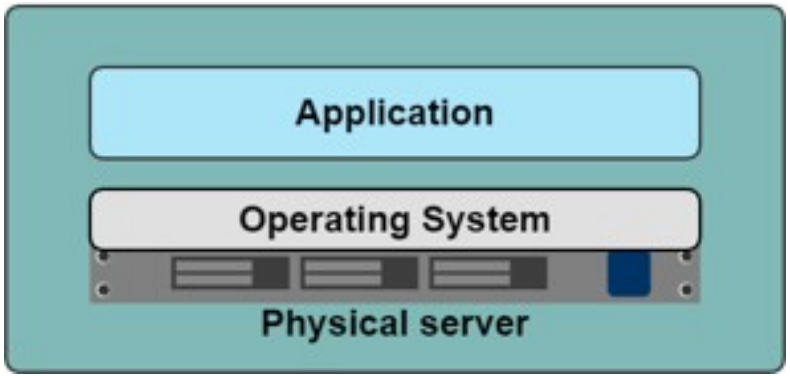
***Docker** is a platform for developing, shipping and running applications using container technology*

- The Docker Platform consists of multiple products/tools
  - Docker Engine
  - Docker Hub
  - Docker Trusted Registry
  - Docker Machine
  - Docker Swarm
  - Docker Compose

A History Lesson

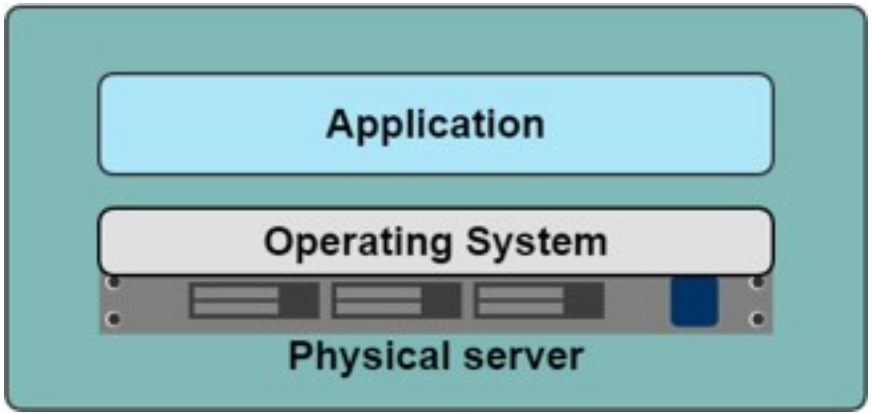
In the Dark Ages

One application on one physical server



Historical limitations of application deployment

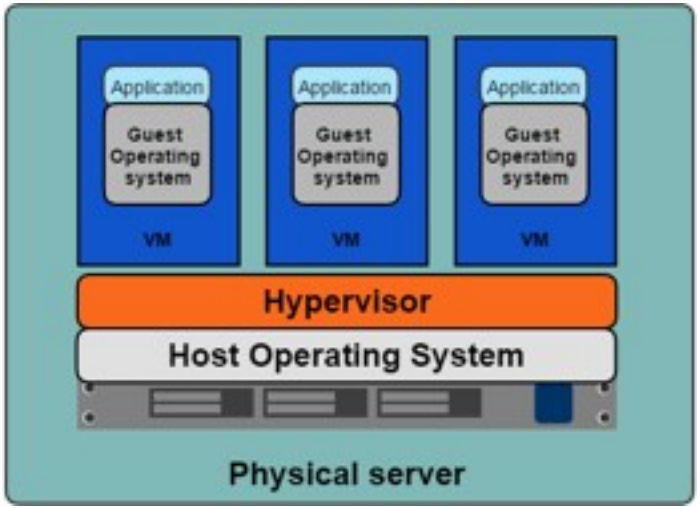
- Slow deployment times
- Huge costs
- Wasted resources
- Difficult to scale
- Difficult to migrate
- Vendor lock in



A History Lesson

Hypervisor-based Virtualization

- One physical server can contain multiple applications
- Each application runs in a virtual machine (VM)



Benefits of VM's

- Better resource pooling
  - One physical machine divided into multiple virtual machines
- Easier to scale
- VM's in the cloud
  - Rapid elasticity
  - Pay as you go model

Google Cloud

Limitations of VM's

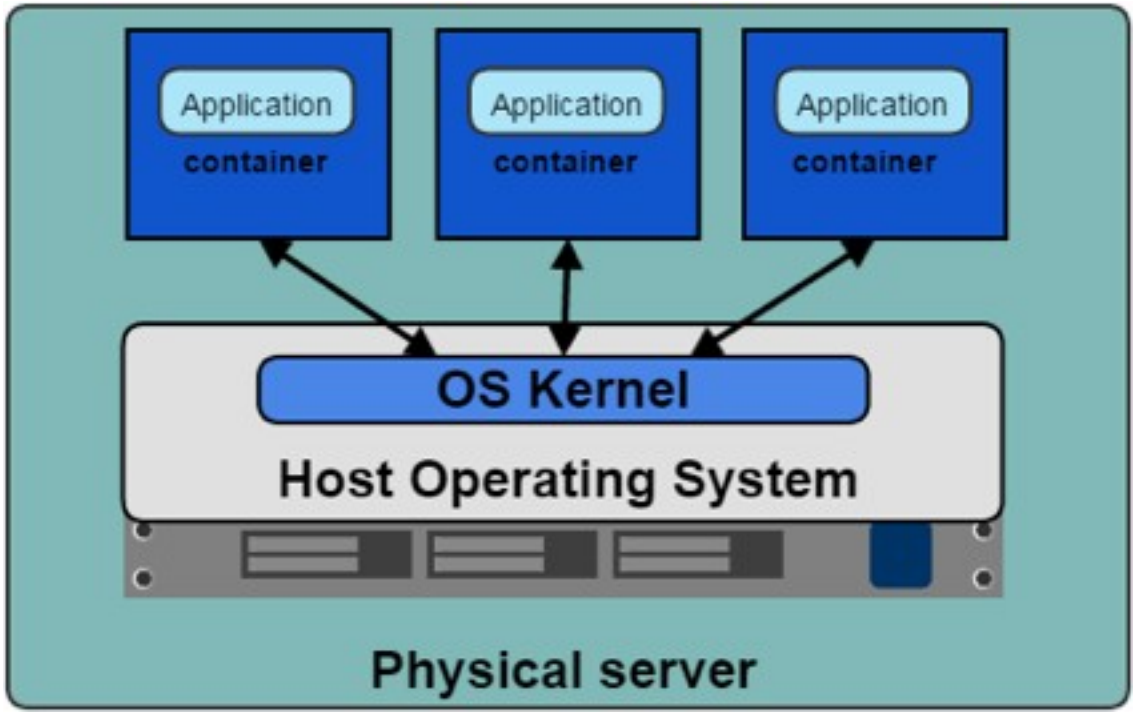
- Each VM stills requires
  - CPU allocation
  - Storage
  - RAM
  - An entire guest operating system
- The more VM's you run, the more resources you need
- Guest OS means wasted resources
- Application portability not guaranteed

Introducing Containers

Containerization uses the kernel on the host operating system to run multiple root file systems

- Each root file system is called a **container**
- Each container also has its own
  - Processes
  - Memory
  - Devices
  - Network stack

Containers


Application
container
Application
container
Application
container
OS Kernel
Host Operating System
Physical server

Containers vs VM’s

- Containers are more lightweight
- No need to install guest OS
- Less CPU, RAM, storage space required
- More containers per machine than VMs
- Speed of instantiation
- Greater portability

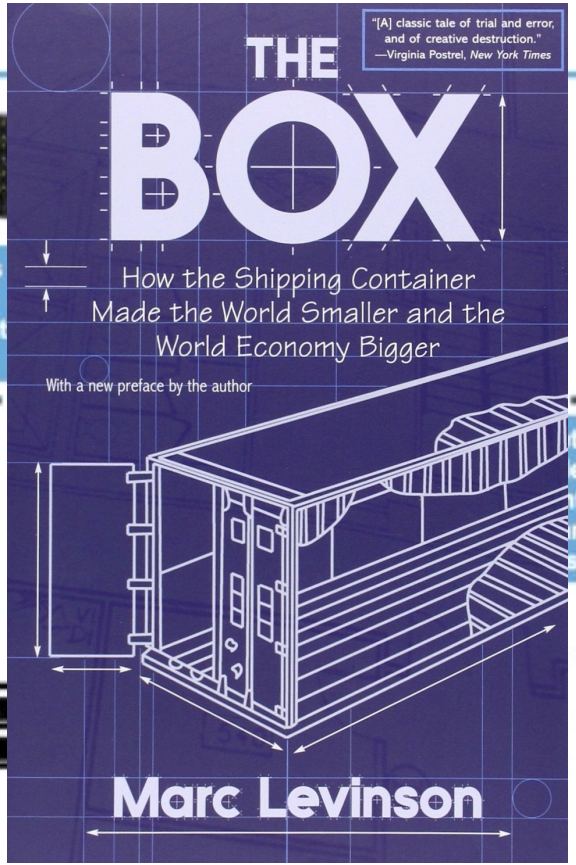
Why Docker?

- Isolation
- Lightweight
- Simplicity
- Workflow
- Community

The shipping container

Multiple types of goods

A standard container that is loaded with virtually any goods, and stays sealed unt it reaches final delivery.



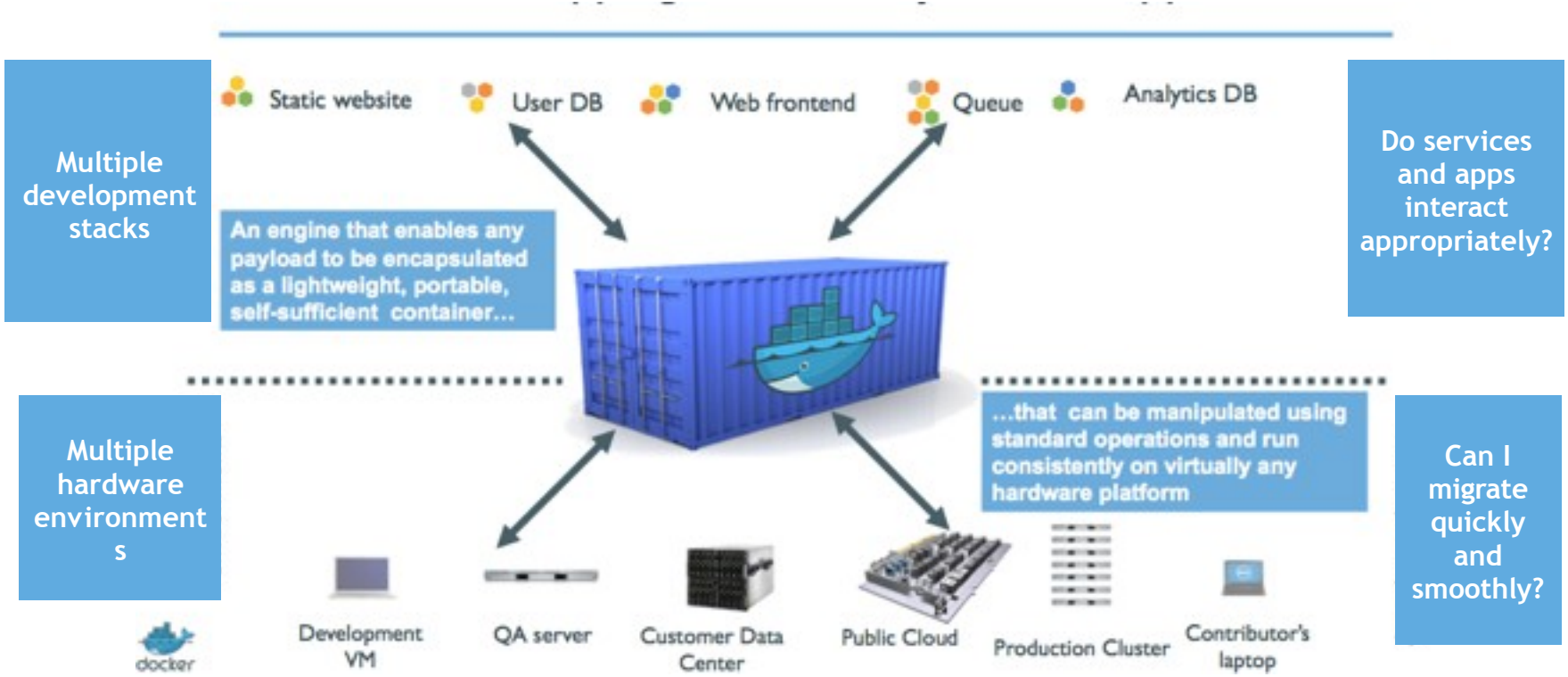
Do I worry about how goods interact? (i.e. place coffee beans next to spices)

Multiple methods of transportati on

tween, can be loaded and ed, stacked, transported tly over long distances, nsferred from one mode sport to another

Can I transport quickly and smoothy? (i.e unload from ship onto train)

Docker containers



Benefits of Docker

- Separation of concerns
  - Developers focus on building their apps
  - System admins focus on deployment
- Fast development cycle
- Application portability
  - Build in one environment, ship to another
- Scalability
  - Easily spin up new containers if needed
- Run more apps on one host machine

Module 2:
Docker Concepts and Terms

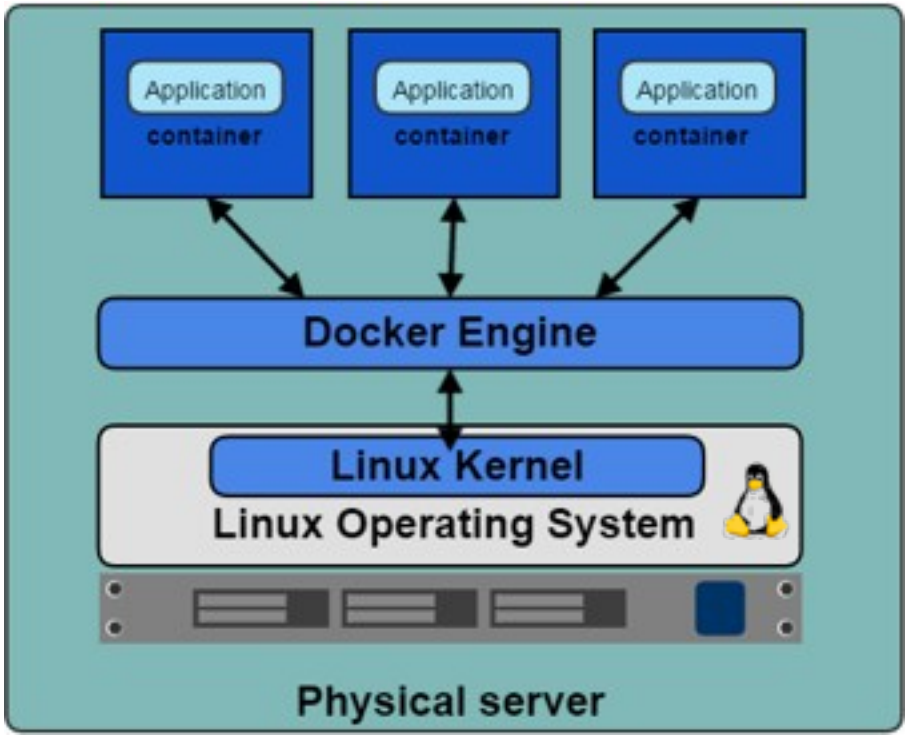
Module objectives

In this module we will:

- Explain all the main Docker concepts including
  - Docker Engine
  - Docker client
  - Containers
  - Images
  - Registry and Repositories
  - Docker Hub
  - Orchestration

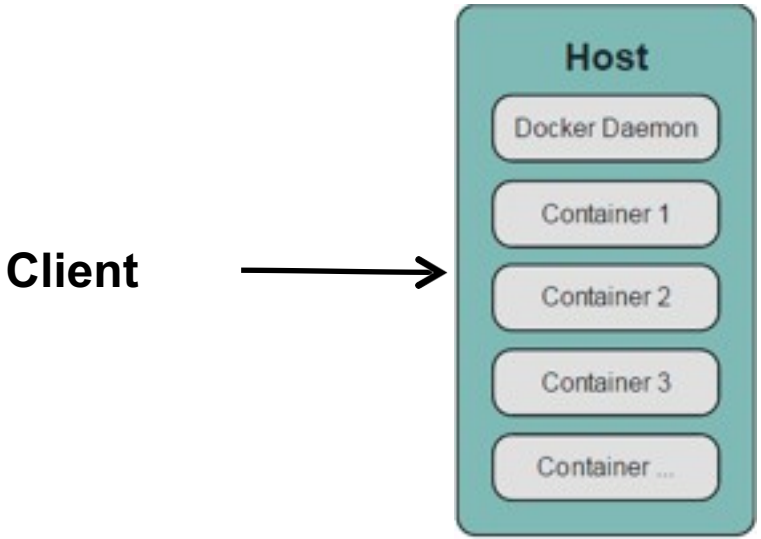
Docker and the Linux Kernel

- **Docker Engine** is the program that enables containers to be distributed and run
- Docker Engine uses Linux Kernel namespaces and control groups
- Namespaces give us the isolated workspace



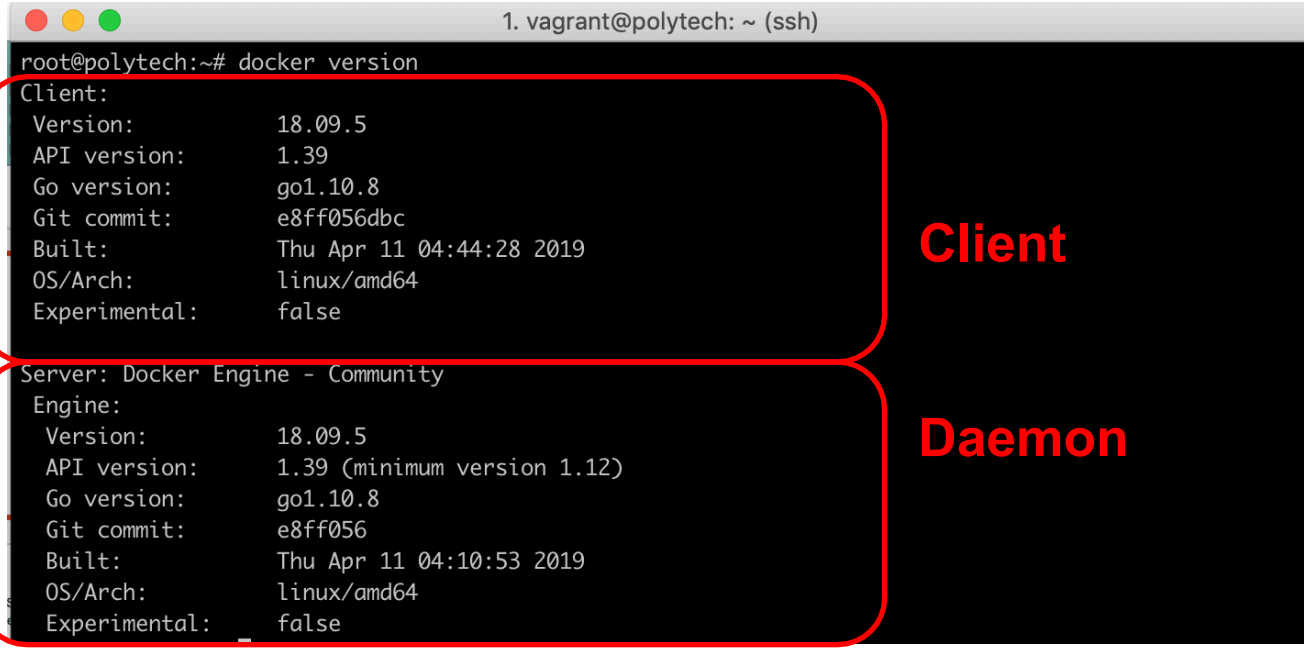
Docker Client and Daemon

- Client / Server architecture
- Client takes user inputs and sends them to the daemon
- Daemon runs and distributes containers
- Client and daemon can run on the same host or on different hosts
- CLI client and GUI (Kitematic)



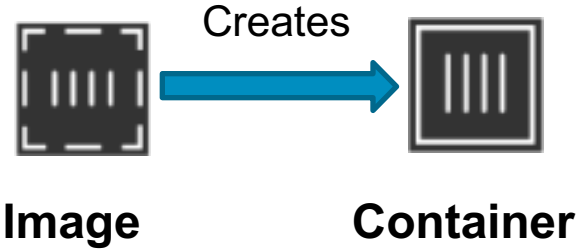
Checking Client and Daemon Version

- Run
  `docker version`

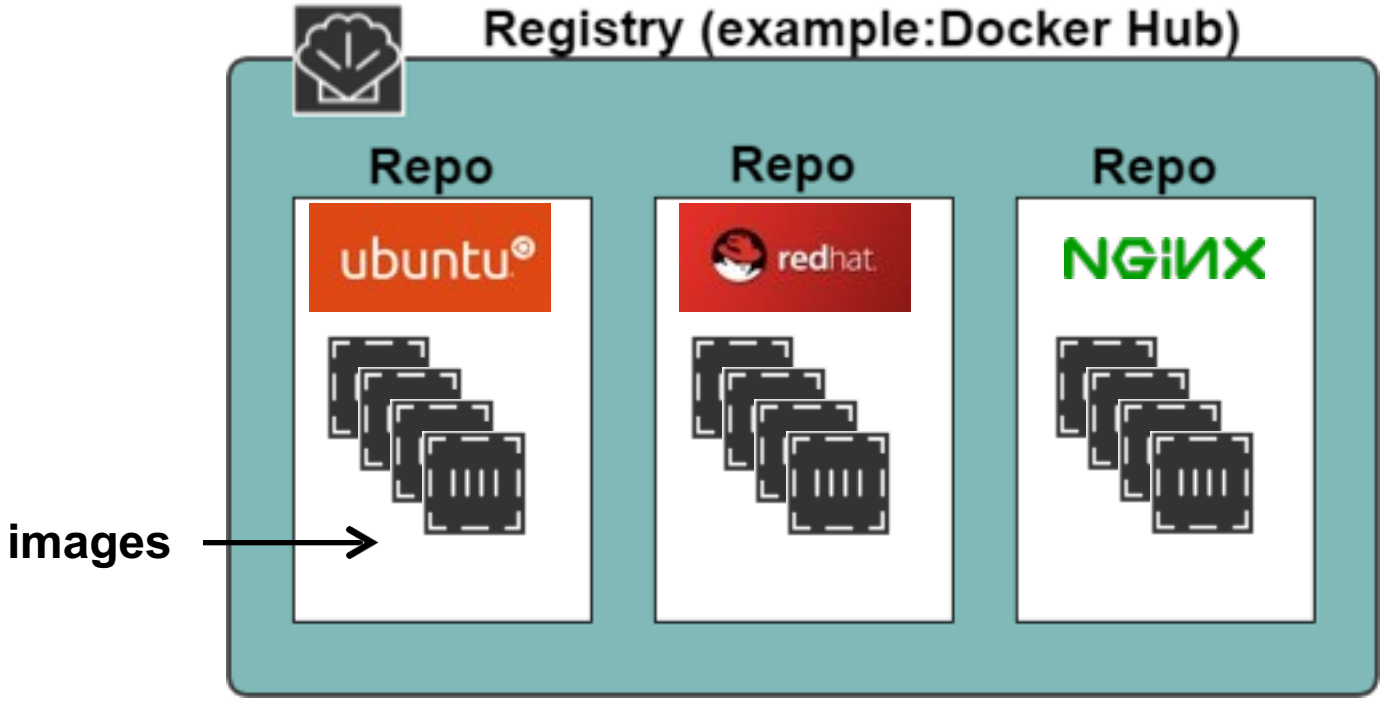


Docker Containers and Images

- Images
  - Read only template used to create containers (Binaries)
  - Built by you or other Docker users
  - Stored in Docker Hub, Docker Trusted Registry or your own Registry
- Containers
  - Isolated application platform
  - Contains everything needed to run your application
  - Based on one or more images

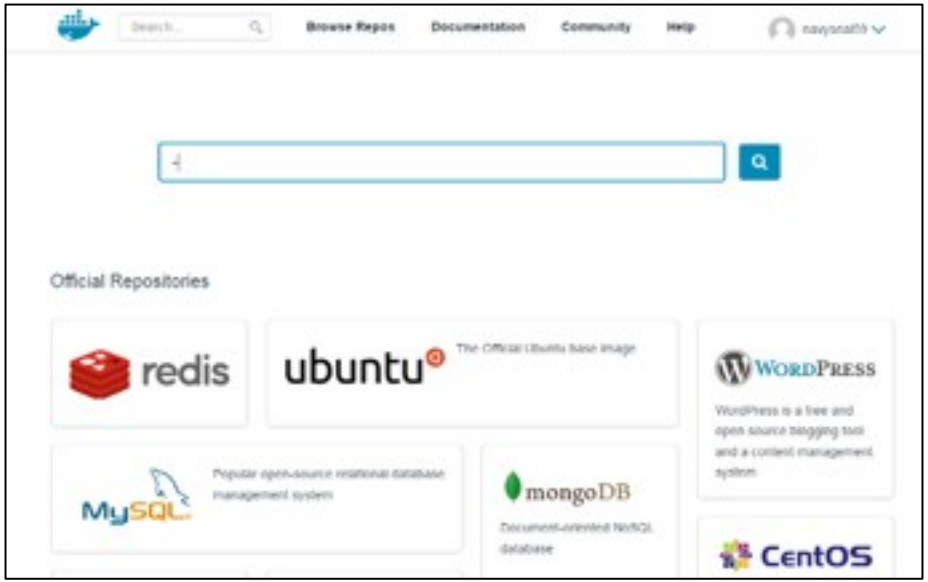


Registry and Repository



Docker Hub

Docker Hub *is the public registry that contains a large number of images available for your use*



Docker Orchestration

- Four tools for orchestrating distributed applications with Docker
- Docker Machine
  - Tool that provisions Docker hosts and installs the Docker Engine on them
- Docker Swarm
  - Tool that clusters many Engines and schedules containers
- Docker Compose
  - Tool to create and manage multi-container applications
- Kubernetes
  - Google Borg for community (swarm killer)

Module 3:
Installing Docker

Module objectives

In this module we will:

- Outline the ways to install Docker on various Linux operating systems
- Install Docker on our Amazon AWS Ubuntu instance
- Explain how to run Docker without requiring "`sudo`"
- Install Boot2Docker on our PC or Mac

Installation options overview

- Docker can be installed in a variety of ways depending on your platform
- Packaged distributions available for most Linux platforms
  - Ubuntu, CentOS, Fedora, Arch Linux, RHEL, Debian, Gentoo, openSUSE
- Binary download
- Installation script from Docker

Module 4:
Introduction to Images

Module objectives

In this module we will:

- Learn to search for images on Docker Hub and with the Docker client
- Explain what official repositories are
- Create a Docker Hub account
- Explain the concept of image tags
- Learn to pull images from Docker Hub

Search for Images on Docker Hub

- Lots of Images available for use
- Images reside in various Repositories

Search for images using Docker client

- Run the `docker search` command
- Results displayed in table form

1. vagrant@polytech: ~ (ssh)

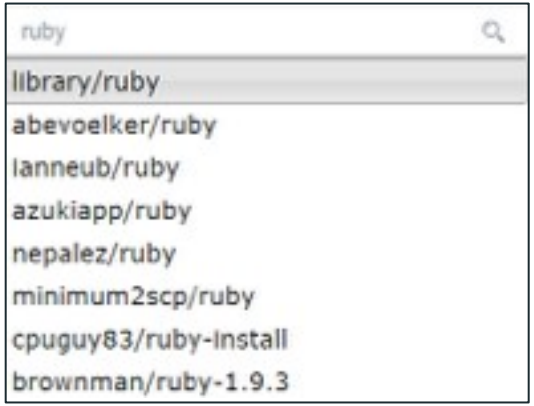
```
root@polytech:~# docker search wordpress
NAME                                              DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
wordpress                                         The WordPress rich content management system…   2934      [OK]
bitnami/wordpress                                 Bitnami Docker Image for WordPress              107                  [OK]
appcontainers/wordpress                           Centos/Debian Based Customizable Wordpress C…   34                   [OK]
etopian/alpine-php-wordpress                      Alpine WordPress Nginx PHP-FPM WP-CLI           20                   [OK]
centurylink/wordpress                             Wordpress image with MySQL removed.             14                   [OK]
raulr/nginx-wordpress                             Nginx front-end for the official wordpress:f…   12                   [OK]
arm32v7/wordpress                                 The WordPress rich content management system…   9
1and1internet/ubuntu-16-nginx-php-5.6-wordpress-4 ubuntu-16-nginx-php-5.6-wordpress-4             7                    [OK]
dalareo/wordpress-ldap                            Wordpress images with LDAP support automatic…   6                    [OK]
withinboredom/scalable-wordpress                  An easy to scale WordPress                      5                    [OK]
julianxhokaxhiu/docker-awesome-wordpress          A Docker that combines all the most awesome …   4                    [OK]
wodby/wordpress-nginx                             Nginx for WordPress                             4                    [OK]
wodby/wordpress-php                               PHP for WordPress                               4                    [OK]
1and1internet/ubuntu-16-nginx-php-7.1-wordpress-4 ubuntu-16-nginx-php-7.1-wordpress-4             3                    [OK]
awee/wordpress-sendmail-docker                    wordpress with sendmail                         2                    [OK]
```

Official repositories

- Official repositories are a certified and curated set of Docker repositories that are promoted on Docker Hub
- Repositories come from vendors such as NGINX, Ubuntu, Red Hat, Redis, etc…
- Images are **supported by their maintainers**, optimised and up to date
- Official repository images are a mixture of
  - Base images for Linux operating systems (Ubuntu, CentOS etc…)
  - Images for popular development tools, programming languages, web and application servers, data stores

Identifying an official repository

- There are a few ways to tell if a repository is official
  - Marked on the `OFFICIAL` column in the terminal output
  - Prefixed with "`library`" on autocomplete search results in Docker Hub
  - Repository has the Docker logo on the Docker Hub search results



Display Local Images

- Run `docker images`
- When creating a container Docker will attempt to use a local image first
- If no local image is found, the Docker daemon will look in Docker Hub unless another registry is specified

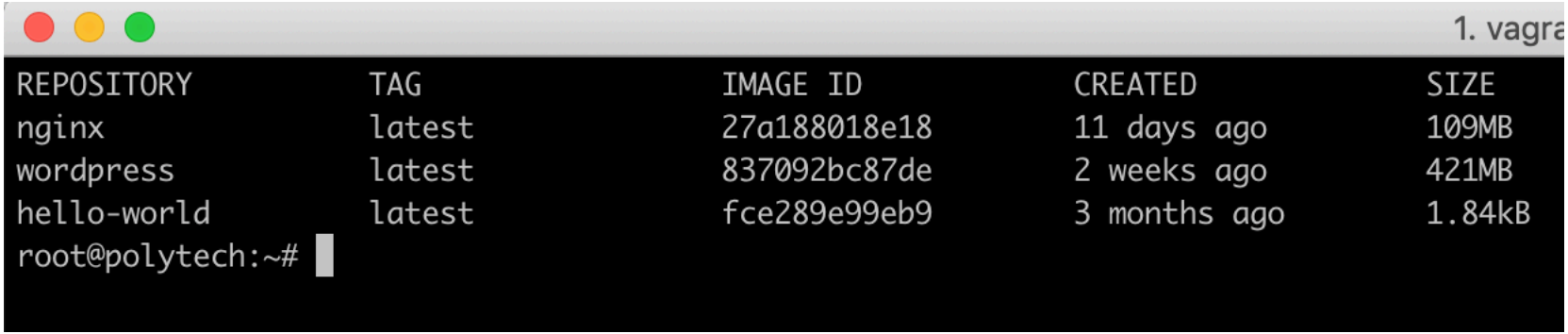


Image Tags

- Images are specified by **repository:tag**
- The same image may have multiple tags
- The default tag is `latest`
- Look up the repository on Docker Hub to see what tags are available

Pulling images

- To download an image from Docker Hub or any registry, use `docker pull` command
- When running a container with the `docker run` command, images are automatically pulled if no local copy is found

Pull the latest image from the Ubuntu repository in Docker Hub

```
docker pull ubuntu
```

Pull the image with tag 12.04 from Ubuntu repository in Docker Hub

```
docker pull ubuntu:12.04
```


Module 5:
Running and Managing Containers

Module objectives

In this module we will

- Learn how to launch containers with the `docker run` command
- Explore different methods of accessing a container
- Explain how container processes work
- Learn how to stop and start containers
- Learn how to check container logs
- Learn how to find your containers with the `docker ps` command
- Learn how to display information about a container

Container lifecycle

- Basic lifecycle of a Docker container
  - Create container from image
  - Run container with a specified process
  - Process finishes and container stops
  - Destroy container
- More advanced lifecycle
  - Create container from image
  - Run container with a specified process
  - Interact and perform other actions inside the container
  - Stop the container
  - Restart the container

Creating and running a Container

- Use `docker run` command
- The `docker run` command actually does two things
  - Creates the container using the image we specify
  - Runs the container
- Syntax
  `docker run [options] [image] [command] [args]`
- Image is specified with `repository:tag`

Examples

```
docker run ubuntu:14.04 echo “Hello World”
docker run ubuntu ps ax
```

Find your Containers

- Use **`docker ps`** to list running containers
- The `-a` flag to list all containers (includes containers that are stopped)

1. vagrant@polytech: ~ (ssh)

```
root@polytech:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
root@polytech:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS               NAMES
d5a122654320        debian              "bash"                   5 minutes ago       Exited (0) 5 minutes ago                        focused_wozniak
c966c7ad66ec        wordpress           "docker-entrypoint.s…"   13 minutes ago      Exited (0) 11 minutes ago                       kind_mahavira
f710b9acecef        hello-world         "/hello"                 28 minutes ago      Exited (0) 28 minutes ago                       amazing_dewdney
c5c0e162308a        hello-world         "/hello"                 28 minutes ago      Exited (0) 28 minutes ago                       ecstatic_blackwell
e5b550c10543        hello-world         "/hello"                 28 minutes ago      Exited (0) 28 minutes ago                       nifty_keldysh
c6b49b6c611f        hello-world         "/hello"                 28 minutes ago      Exited (0) 28 minutes ago                       tender_mahavira
39cce39c990b        hello-world         "/hello"                 30 minutes ago      Exited (0) 30 minutes ago                       dazzling_shirley
d5e1006c9fea        nginx               "nginx -g 'daemon of…"   30 minutes ago      Exited (0) 25 minutes ago                       stoic_kepler
root@polytech:~#
```

Container with Terminal

- Use `-i` and `-t` flags with docker run
- The `-i` flag tells docker to connect to STDIN on the container
- The `-t` flag specifies to get a pseudo-terminal
- **Note:** You need to run a terminal process as your command (e.g. `bash`)

Example

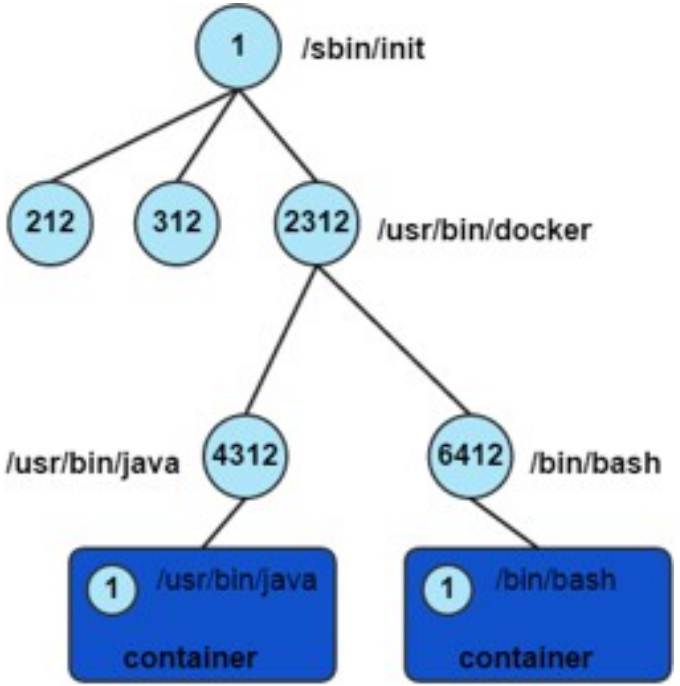
```
docker run -i -t ubuntu:latest bash
```

Exit the Terminal

- Type `exit` to quit the terminal and return to your host terminal
- Exiting the terminal will shutdown the container
- To exit the terminal without a shutdown, hit **`CTRL + P + Q`** together

Container Processes

- A container only runs as long as the process from your specified `docker run` command is running
- Your command's process is always PID 1 inside the container

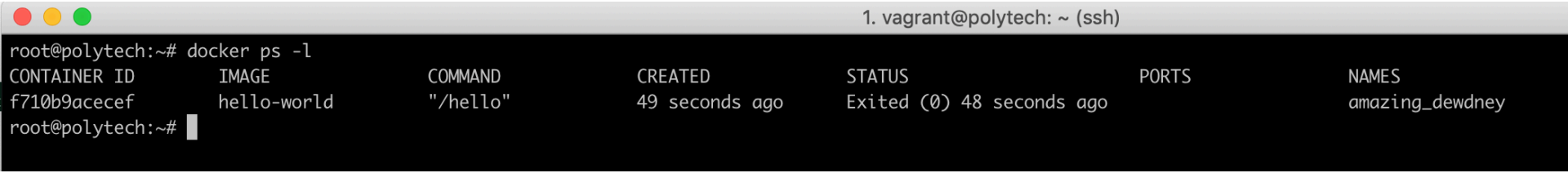


Container ID

- Containers can be specified using their ID or name
- Long ID and short ID
- Short ID and name can be obtained using `docker ps` command to list containers
- Long ID obtained by inspecting a container

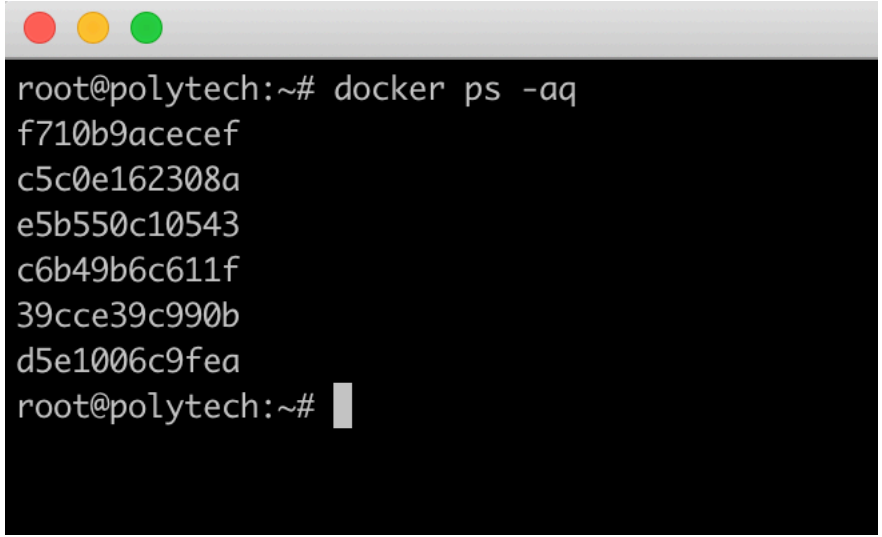
docker ps command

- To view only the container ID's (displays short ID)
  `docker ps -q`
- The view the last container that was started
  `docker ps -l`



docker ps command

- Combining flags to list all containers with only their short ID
  `docker ps -aq`
- Combining flags to list the short ID of the last container started
  `docker ps -lq`

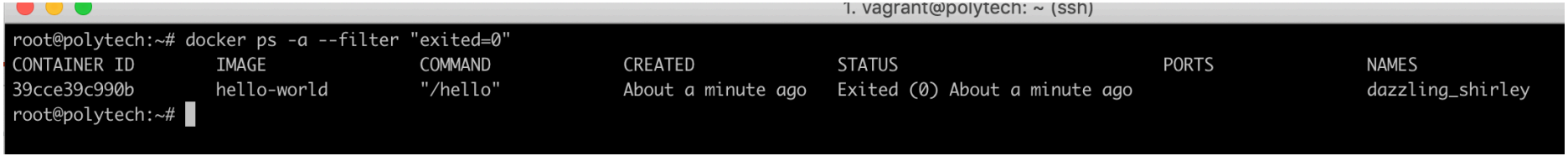


docker ps filtering

- Use the `--filter` flag to specify filtering conditions
- Currently you can filter based on the container's exit code and status
- Status can be one of
  - Restarting
  - Running
  - Exited
  - Paused
- To specify multiple conditions, pass multiple `--filter` flags

docker ps filtering examples

- List containers with an exit code of 1 (exited with error)



Running in Detached Mode

- Also known as running in the background or as a daemon
- Use `-d` flag
- To observe output use **`docker logs [container id]`**

Create a centos container and run the ping command to ping the container itself 50 times

```
docker run -d centos:7 ping 127.0.0.1 -c 50
```

A More Practical Container

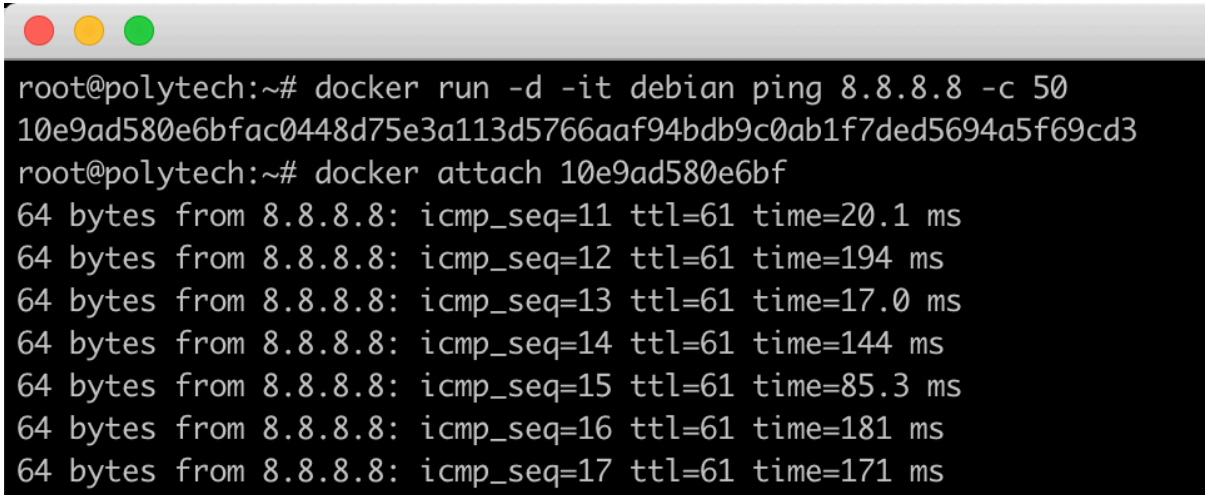
- Run a web application inside a container
- The `-P` flag to map container ports to host ports

Create a container using the tomcat image, run in detached mode and map the tomcat ports to the host port

```
docker run -d -P tomcat:7
```

Attaching to a container

- Attaching a client to a container will bring a container which is running in the background into the foreground
- The containers PID 1 process output will be displayed on your terminal
- Use `docker attach` command and specify the container ID or name
- **Warning:** Attaching to containers is error prone because if you hit `CTRL + C` by accident, you will stop the process and therefore stop the container



Detaching from a container

- Hit CTRL + P + Q together on your terminal
- Only works if the following two conditions are met
  - The container standard input is connected
  - The container has been started with a terminal
  - For example: `docker run -i -t ubuntu`
- Hitting CTRL + C will terminate the process, thus shutting down the container

Docker exec command

- `docker exec` command allows us to execute additional processes inside a container
- Typically used to gain command line access
- `docker exec -i -t [container ID] bash`
- Exiting from the terminal will not terminate the container

Inspecting container logs

- Container PID 1 process output can be viewed with **`docker logs`** command
- Will show whatever PID 1 writes to stdout
- Displays the entire log output from the time the container was created

View the output of the containers PID 1 process

```
docker logs <container name>
```


Following container logs

- Use `docker logs` command and specify the `-f` option
- Similar to Linux `tail -f` command
- `CTRL + C` to exit
- Will still follow the log output from the very beginning

Tailing container logs

- We can specify to only show the last “x” number of lines from the logs
- Use `--tail` option and specify the number

Show the last 5 lines from the container log

```
docker logs --tail 5 <container ID>
```

Show the last 5 lines and follow the log

```
docker logs --tail 5 -f <container ID>
```


Stopping a container

- Two commands we can use
  - `docker stop`
  - `docker kill`
- `docker stop` sends a SIGTERM to the main container process
  - Process then receives a SIGKILL after a grace period
  - Grace period can be specified with -t flag (default is 10 seconds)
- `docker kill` sends a SIGKILL immediately to the main container process

Restarting a container

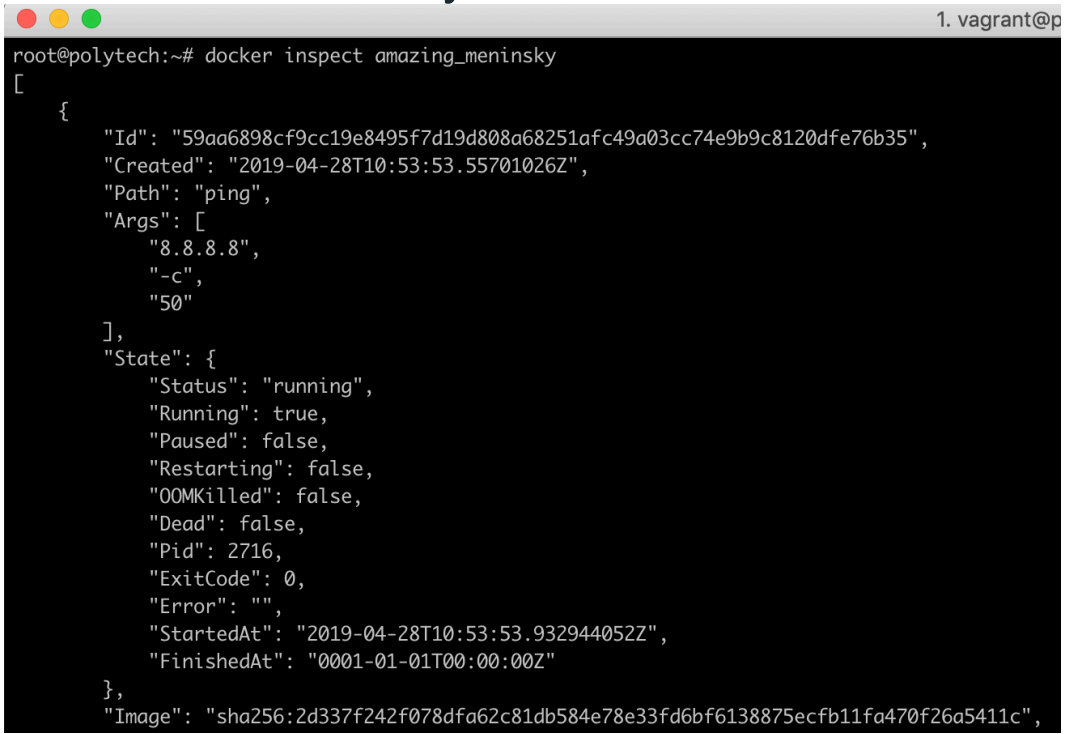
- Use `docker start` to restart a container that has been stopped
- Container will start using the same options and command specified previously
- Can attach to the container with `-a` flag

Start a stopped container and attach to the process that it is running

```
docker start -a <container ID>
```


Inspecting a container

- `docker inspect` command displays all the details about a container
- Outputs details in JSON array



Finding a specific property

- You can pipe the output of `docker inspect` to `grep` and use it to search a particular container property.
- Example
  `docker inspect <container name> | grep IPAddress`

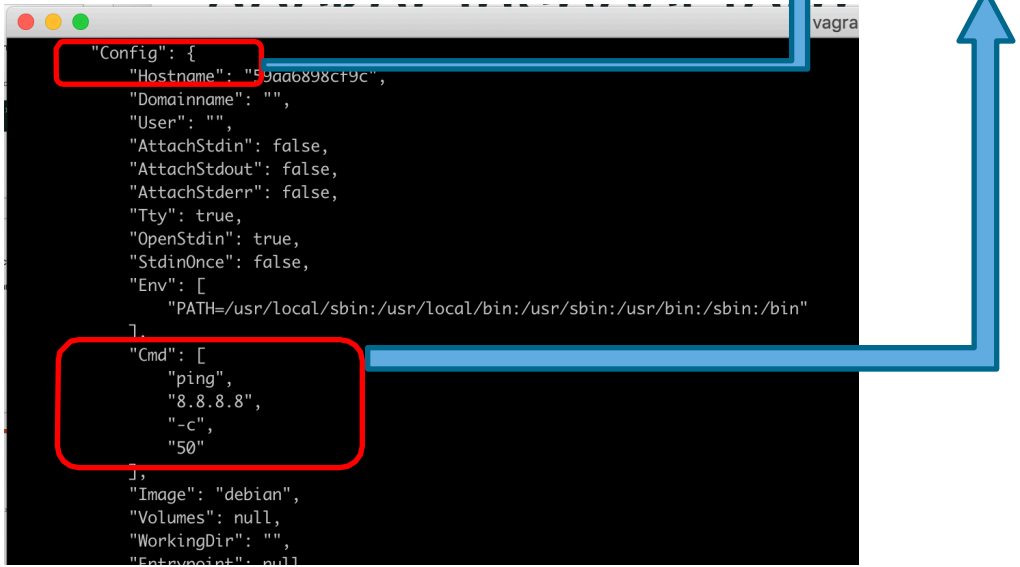
```
johnnytu@docker-ubuntu:~$ docker inspect jolly_davinci | grep IPAddress
        "IPAddress": "172.17.0.24",
```


Formatting docker inspect output

- Piping the output to `grep` is simple, but not the most effective method of formatting
- Use the `--format` option of the `docker inspect` command
- Format option uses Go's text/template package
  http://golang.org/pkg/text/template/

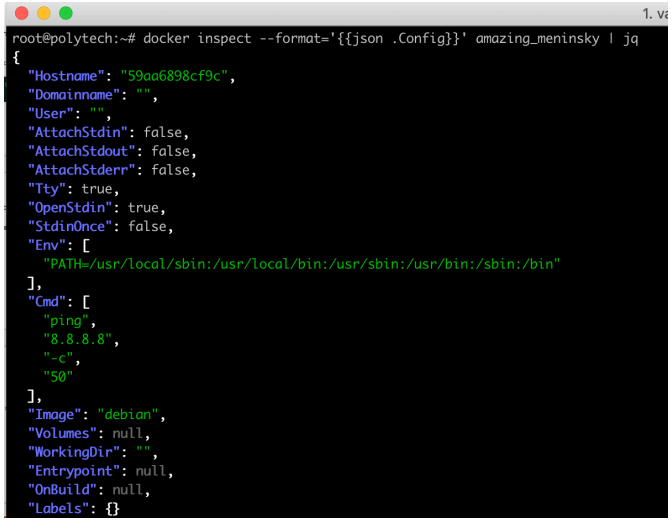
docker inspect formatting syntax

- `docker inspect --format='{{.<field1>.<field2>}}' \`
  `<container id>`
- Field names are case sensitive
- Example – to get the value of the `Cmd` field
  `docker inspect --format='{{.Config.Cmd}}' <container id>`



Inspecting a whole JSON object

- When you want to output all the fields of a JSON object you need to use the Go template's JSON function



Incorrect approach

```
docker inspect --format='{{.Config}}' <container name>
```

Correct approach

```
docker inspect --format='{{json .Config}}' <container name>
```


Deleting containers

- Can only delete containers that have been stopped
- Use `docker rm` command
- Specify the container ID or name

Delete all containers

- Use `docker ps -aq` to list the id's of all containers
- Feed the output into `docker rm` command

Delete all containers that are stopped

```
docker rm $(docker ps -aq)
```


Module Summary

- Containers can be run in the foreground and background
- A container only runs as long as the process we specified during creation is running
- The container log is the output of its PID 1 process
- Key commands we learnt
  - `docker run`
  - `docker ps`
  - `docker logs`
  - `docker inspect`

Module 6: BuildingImages

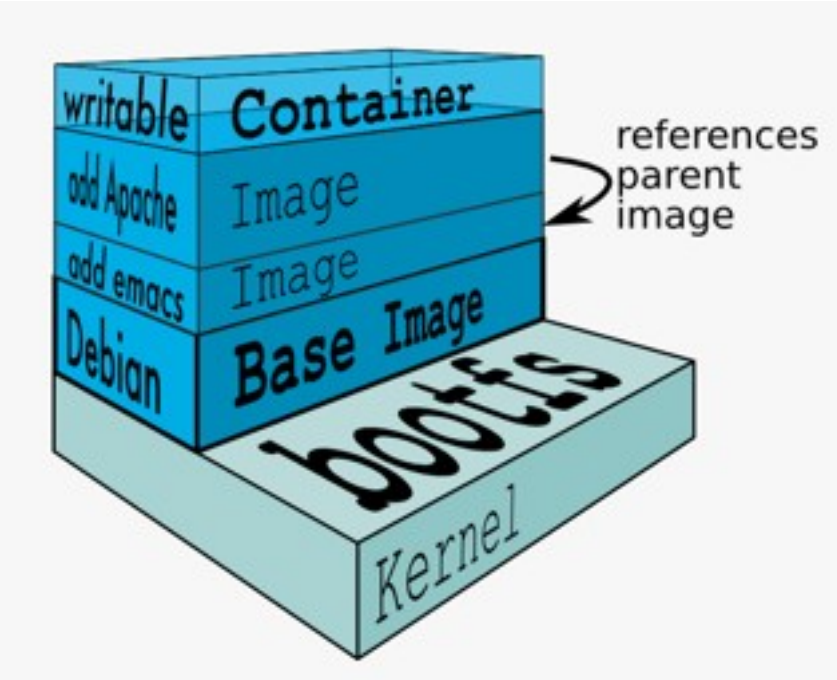
Module objectives

In this module we will

- Explain how image layers work
- Build an image by committing changes in a container
- Learn how to build images with a Dockerfile
- Work through examples of key Dockerfile instructions
  - RUN
  - CMD
  - ENTRYPOINT
  - COPY
- Talk about the best practices when writing a Dockerfile

Understanding image layers

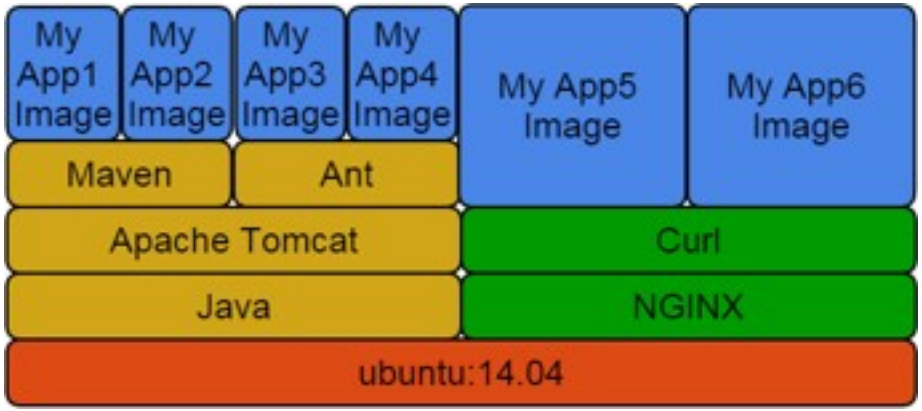
- An image is a collection of files and some meta data
- Images are comprised of multiple layers
- A layer is also just another image
- Each image contains software you want to run
- Every image contains a base layer
- Docker uses a copy on write system
- Layers are read only



- COW/Union Filesystems (AUFS/BTRFS)

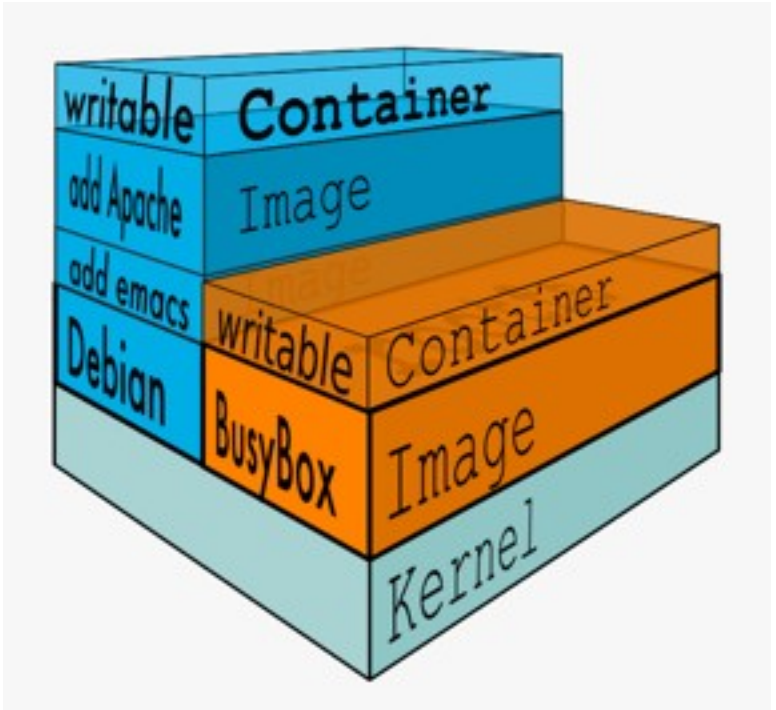
Sharing layers

- Images can share layers in order to speed up transfer times and optimize disk and memory usage
- Parent images that already exists on the host do not have to be downloaded



The container writable layer

- Docker creates a top writable layer for containers
- Parent images are read only
- All changes are made at the writeable layer
- When changing a file from a read only layer, the copy on write system will copy the file into the writable layer



Methods of building images

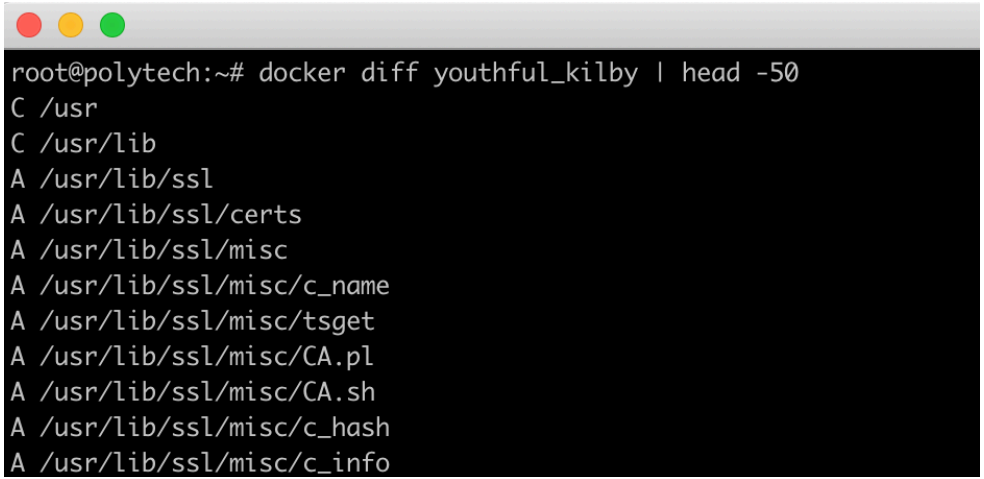
- Three ways
  - Commit changes from a container as a new image
  - Build from a Dockerfile
  - Import a tarball into Docker as a standalone base layer

Committing changes in a container

- Allows us to build images interactively
- Get terminal access inside a container and install the necessary programs and your application
- Then save the container as a new image using the `docker commit` command

Comparing container changes

- Use the `docker diff` command to compare a container with its parent image
  - Recall that images are read only and changes occur in a new layer
  - The parent image (the original) is being compared with the new layer
- Copy on write system ensures that starting a container from a large image does not result in a large copy operation
- Lists the files and directories that have changed



Docker Commit

- `docker commit` command saves changes in a container as a new image
- Syntax
  `docker commit [options] [container ID] [repository:tag]`
- Repository name should be based on username/application
- Can reference the container with container name instead of ID

Save the container with ID of 984d25f537c5 as a new image in the repository johnnytu/myapplication. Tag the image as 1.0

```
docker commit 984d25f537c5 johnnytu/myapplication:1.0
```


Image namespaces

- Image repositories belong in one of three namespaces
  - Root
    ```
    ubuntu:14.04
    centos:7
    nginx
    ```
  - User OR organization
    ```
    johnnytu/myapp
    mycompany/myapp
    ```
  - Self hosted
    ```
    registry.mycompany.com:5000/my-image
    ```


Uses for namespaces

- Root namespace is for official repositories
- User and organization namespaces are for images you create and plan to distribute on Docker Hub
- Self-hosted namespace is for images you create and plan to distribute in your own registry server

Intro to Dockerfile

*A **Dockerfile** is a configuration file that contains instructions for building a Docker image*

- Provides a more effective way to build images compared to using `docker commit`
- Easily fits into your development workflow and your continuous integration and deployment process

Process for building images from Dockerfile

1. Create a Dockerfile in a new folder or in your existing application folder
2. Write the instructions for building the image
   - What programs to install
   - What base image to use
   - What command to run
3. Run `docker build` command to build an image from the Dockerfile

Dockerfile Instructions

- Instructions specify what to do when building the image
- **FROM** instruction specifies what the base image should be
- **RUN** instruction specifies a command to execute
- Comments start with "#"

```
#Example of a comment
FROM ubuntu:14.04
RUN apt-get install vim
RUN apt-get install curl
```


FROM instruction

- Must be the first instruction specified in the Dockerfile (not including comments)
- Can be specified multiple times to build multiple images
  - Each `FROM` marks the beginning of a new image
- Can use any image including, images from official repositories, user images and images in self hosted registries.

Examples

```
FROM ubuntu
FROM ubuntu:14.04
FROM johnnytu/myapplication:1.0
FROM company.registry:5000/myapplication:1.0
```


More about RUN

- `RUN` will do the following:
  - Execute a command.
  - Record changes made to the filesystem.
  - Works great to install libraries, packages, and various files.
- `RUN` will NOT do the following:
  - Record state of *processes*.
  - Automatically start daemons.

Docker Build

- Syntax
  `docker build [options] [path]`
- Common option to tag the build
  `docker build -t [repository:tag] [path]`

Build an image using the current folder as the context path. Put the image in the johnnytu/myimage repository and tag it as 1.0

```
docker build -t johnnytu/myimage:1.0 .
```

As above but use the myproject folder as the context path

```
docker build -t johnnytu/myimage:1.0 myproject
```


Build output

```
johnnytu@docker-ubuntu:~/myimage$ docker build -t myimage .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
 ---> 07f8e8c5e660
Step 1 : RUN apt-get update
 ---> Running in fd009309d260
...
...
Reading package lists...
 ---> 161253fe4244
Removing intermediate container fd009309d260
Step 2 : RUN apt-get install -y wget
 ---> Running in 69ba8a082150
Reading package lists...
...
...
 ---> c7cc72b567e4
Removing intermediate container 69ba8a082150
Successfully built c7cc72b567e4
```


The build context

```
johnnytu@docker-ubuntu:~/myimage$ docker build -t myimage .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
```

- The build context is the directory that the Docker client sends to the Docker daemon during the `docker build` command
- Directory is sent as an archive
- Docker daemon will build using the files available in the context
- Specifying “.” for the build context means to use the current directory

Examining the build process

- Each RUN instruction will execute the command on the top writable layer of a new container

```
Step 1 : RUN apt-get update
 ---> Running in fd009309d260
```

- At the end of the execution of the command, the container is committed as a new image and then deleted

```
 ---> 161253fe4244
Removing intermediate container fd009309d260
```


Examining the build process (cont'd)

- The next RUN instruction will be executed in a new container using the newly created image

```
Step 2 : RUN apt-get install -y wget
 ---> Running in 69ba8a082150
```

- The container is committed as a new image and then removed. Since this RUN instruction is the final instruction, the image committed is what will be used when we specify the repository name and tag used in the `docker build` command

```
 ---> c7cc72b567e4
Removing intermediate container 69ba8a082150
Successfully built c7cc72b567e4
```


The build cache

- Docker saves a snapshot of the image after each build step
- Before executing a step, Docker checks to see if it has already run that build sequence previously
- If yes, Docker will use the result of that instead of executing the instruction again
- Docker uses exact strings in your Dockerfile to compare with the cache
  - Simply changing the order of instructions will invalidate the cache
- To disable the cache manually use the `--no-cache` flag

```
docker build --no-cache myimage .
```


Run Instruction aggregation (Best Practice)

- Can aggregate multiple RUN instructions by using "`&&`"
- Commands will all be run in the same container and committed as a new image at the end
- Reduces the number of image layers that are produced

```
RUN apt-get update && apt-get install -y \
      curl \
      vim \
      openjdk-7-jdk
```


Viewing Image layers and history

- `docker history` command shows us the layers that make up an image
- See when each layer was created, its size and the command that was run

```
IMAGE               CREATED             CREATED BY                                      SIZE
10f1e1747aa1        12 seconds ago      /bin/sh -c apt-get install -y wget              6.119 MB
9b6aeef1e9cc        23 seconds ago      /bin/sh -c apt-get install -y vim               43.12 MB
334d0289feff        10 minutes ago      /bin/sh -c apt-get update                       20.86 MB
07f8e8c5e660        2 weeks ago         /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
37bea4ee0c81        2 weeks ago         /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$/   1.895 kB
a82efea989f9        2 weeks ago         /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic   194.5 kB
e9e06b06e14c        2 weeks ago         /bin/sh -c #(nop) ADD file:f4d7b4b3402b5c53f2   188.1 MB
```


CMD Instruction

- `CMD` defines a default command to execute when a container is created
- Shell format and EXEC format
- Can only be specified once in a Dockerfile
  - If specified multiple times, the last CMD instruction is executed
- **Can be overridden at run time**
- (Array format doesn't append /bin/sh -c)

Shell format

```
CMD ping 127.0.0.1 –c 30
```

Exec format

```
CMD [“ping”, “127.0.0.1”, “-c”, “30”]
```


ENTRYPOINT Instruction

- Defines the command that will run when a container is executed
- Run time arguments and `CMD` instruction are passed as parameters to the `ENTRYPOINT` instruction
- Shell and EXEC form
- Container essentially runs as an executable
- Bu default ENTRYPOINT is not overridden (can be using —entrypoint flag)

```
ENTRYPOINT [“/usr/sbin/nginx”]
CMD [“-h”]
```


Using CMD with ENTRYPOINT

- If `ENTRYPOINT` is used, the `CMD` instruction can be used to specify default parameters
- Parameters specified during `docker run` will override `CMD`
- If no parameters are specified during `docker run`, the `CMD` arguments will be used for the `ENTRYPOINT` command

Shell vs exec format

- The RUN, CMD and ENTRYPOINT instructions can be specified in either shell or exec form

In shell form, the command will run inside a shell with `/bin/sh -c`

```
RUN apt-get update
```

Exec format allows execution of command in images that don't have `/bin/sh`

```
RUN ["apt-get", "update"]
```


Shell vs exec format

- Shell form is easier to write and you can perform shell parsing of variables
- For example

```
CMD sudo -u ${USER} java ….
```

- Exec form does not require image to have a shell
- For the ENTRYPOINT instruction, using shell form will prevent the ability to specify arguments at run time
  - The CMD arguments will not be used as parameters for ENTRYPOINT

Overriding ENTRYPOINT

- Specifying parameters during `docker run` will result in your parameters being used as arguments for the `ENTRYPOINT` command
- To override the command specified by `ENTRYPOINT`, use the `--entrypoint` flag.
- Useful for troubleshooting your images

Run a container using the image “myimage” and specify to run a bash terminal instead of the program specified in the image ENTRYPOINT instruction

```
docker run -it --entrypoint bash myimage
```


Copying source files

- When building “real” images you would want to do more than just install some programs
- Examples
  - Compile your source code and run your application
  - Copy configuration files
  - Copy other content
- How do we get our content on our host into the container?
- Use the `COPY` instruction

COPY instruction

- The `COPY` instruction copies new files or directories from a specified **source** and adds them to the container filesystem at a specified **destination**
- Syntax
  `COPY <src> <dest>`
- The `<src>` path must be inside the build context
- If the `<src>` path is a directory, all files in the directory are copied. The directory itself is not copied
- You can specify multiple `<src>` directories

COPY examples

Copy the server.conf file in the build context into the root folder of the container

```
COPY server.conf /
```

Copy the files inside the `data/server` folder of the build context into the `/data/server` folder of the container

```
COPY data/server /data/server
```


Dockerize an application

- The Dockerfile is essential if we want to adapt our existing application to run on containers
- Take a simple Java program as an example. To build and run it, we need the following on our host
  - The Java Development Kit (JDK)
  - The Java Virtual Machine (JVM)
  - Third party libraries depending on the application itself
- You compile the code, run the application and everything looks good

Dockerize an application

- Then you distribute the application and run it on a different environment and it fails
- Reasons why the Java application fails?
  - Missing libraries in the environment
  - Missing the JDK or JVM
  - Wrong version of libraries
  - Wrong version of JDK or JVM
- So why not run your application in a Docker container?
- Install all the necessary libraries in the container
- Build and run the application inside the container and distribute the image for the container
- Will run on any environment with the Docker Engine installed

Specify a working directory

- Previously all our instructions have been executed at the root folder in our container
- `WORKDIR` instruction allows us to set the working directory for any subsequent `RUN`, `CMD`, `ENTRYPOINT` and `COPY` instructions to be executed in
- Syntax
  `WORKDIR /path/to/folder`
- Path can be absolute or relative to the current working directory
- Instruction can be used multiple times

MAINTAINER Instruction

- Specifies who wrote the Dockerfile
- Optional but best practice to include
- Usually placed straight after the `FROM` instruction

Example

```
MAINTAINER Docker Training <education@docker.com>
```


ENV instruction

- Used to set environment variables in any container launched from the image
- Syntax

```
ENV <variable> <value>
```

Examples

```
ENV JAVA_HOME /usr/bin/java
ENV APP_PORT 8080
```


ADD instruction

- The `ADD` instruction copies new files or directories from a specified **source** and adds them to the container filesystem at a specified **destination**
- Syntax
  `ADD <src> <dest>`
- The `<src>` path is relative to the directory container the Dockerfile
- If the `<src>` path is a directory, all files in the directory are copied. The directory itself is not copied
- You can specify multiple `<src>` directories

Example

```
ADD /src /myapp/src
```


COPY vs ADD

- Both instructions perform a near identical function
- ADD has the ability to auto unpack tar files
- ADD instruction also allows you to specify a URL for your content (although this is not recommended)
- Both instructions use a checksum against the files added. If the checksum is not equal then the test fails and the build cache will be invalidated
  - Because it means we have modified the files

Best practices for writing Dockerfiles

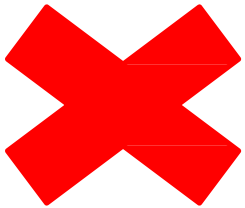
- Remember, each line in a Dockerfile creates a new layer
- You need to find the right balance between having lots of layers created for the image and readability of the Dockerfile
- Don’t install unnecessary packages
- One ENTRYPOINT per Dockerfile
- Combine similar commands into one by using “&&” and “\”

Example

```
RUN apt-get update && \
    apt-get install -y vim && \
    apt-get install -y curl
```


Best practices for writing Dockerfiles

- Use the caching system to your advantage
  - The order of statements is important
  - Add files that are least likely to change first and the ones most likely to change last
- The example below is not ideal because our build system does not know whether the `requirements.txt` file has changed



```
FROM ubuntu:14.04
MAINTAINER Docker Education Team <education@docker.com>
RUN apt-get update
RUN DEBIAN_FRONTEND=noninteractive apt-get install -y -q \
    python-all python-pip
ADD ./webapp /opt/webapp/
WORKDIR /opt/webapp
RUN pip install -qr requirements.txt
EXPOSE 5000
CMD ["python", "app.py"]
```


Best practices for writing Dockerfiles

- **Correct way** would be in the example below.
- `requirements.txt` is added in a separate step so Docker can cache more efficiently. If there's no change in the file the RUN pip install instruction does not have to execute and Docker can use the cache for that layer.
- The rest of the files are added afterwards

```
FROM ubuntu:14.04
MAINTAINER Docker Education Team <education@docker.com>
RUN apt-get update
RUN DEBIAN_FRONTEND=noninteractive apt-get install -y -q \
    python-all python-pip
ADD ./webapp/requirements.txt /tmp/requirements.txt
RUN pip install -qr /tmp/requirements.txt
ADD ./webapp /opt/webapp/
WORKDIR /opt/webapp
EXPOSE 5000
CMD ["python", "app.py"]
```


Module summary

- Image are made up of multiple layers
- Committing changes we make in a container as a new image is a simple way to create our own images but is not a very effective method as part of a development workflow
- A Dockerfile is the preferred method of creating images
- Key Dockerfile instructions we learnt about
  - `RUN`
  - `CMD`
  - `ENTRYPOINT`
  - `COPY`
- Key commands
  - `docker build`

Module 7: Volumes

Module objectives

In this module we will

- Explain what volumes are and what they are used for
- Learn the different methods of mounting a volume in a container
- Mount volumes during the `docker run` command and also in a Dockerfile
- Explain how data containers work
- Create some data containers

Volumes

*A **Volume** is a designated directory in a container, which is designed to persist data, independent of the container's life cycle*

- Volume changes are excluded when updating an image
- Persist when a container is deleted (kind of.. not documented dirs)
- Can be mapped to a host folder
- Can be shared between containers

Volumes and copy on write

- Volumes bypass the copy on write system
- Act as passthroughs to the host filesystem
- When you commit a container as a new image, the content of the volumes will not be brought into that image
- If a `RUN` instruction in a Dockerfile changes the content of a volume, those changes are not recorded neither.

Uses of volumes

- De-couple the data that is stored, from the container which created the data
- Good for sharing data between containers
  - Can setup a data containers which has a volume you mount in other containers
  - Share directories between multiple containers
- Bypassing the copy on write system to achieve native disk I/O performance
- Share a host directory with a container
- Share a single file between the host and container

Mount a Volume

- Volumes can be mounted when running a container
- Use the `-v` option on `docker run`
- Volume paths specified must be absolute
- Can mount multiple volumes by using the `-v` option multiple times

Execute a new container and mount the folder `/myvolume` into its file system

```
docker run -d -P -v /myvolume nginx:1.7
```

Example of mounting multiple volumes

```
docker run -d -P -v /data/www -v /data/images nginx
```


Where are our volumes?

- Volumes exist independently from containers
- If a container is stopped, we can still access our volume
- To find where the volume is use `docker inspect` on the container

```
    "Volumes": {
        "/www/data": "/var/lib/docker/vfs/dir/1bff820731cbbaa2ec764269298deedd21f8bee55809cae309d03390892bf72a"
    },
    "VolumesRW": {
        "/www/data": true
    }
```


Deleting a volume

- Volumes are not deleted when you delete a container
- To remove the volumes associated with a container use the `-v` option in the `docker rm` command

Delete a container and remove it's associated volumes

```
docker rm -v <container ID>
```


Deleting volumes

- If you created multiple containers which referenced the same volume, you will be able to access that volume as long as one of those containers is still present (running or stopped)
- When you remove the last container referencing a volume, that volume will be orphaned
- Orphaned volumes still exists but are very difficult to access and remove
- **Best practice:** When deleting a container, make sure you delete the volume associated with it, unless there are other containers using that volume

Mounting host folders to a volume

- When running a container, you can map folders on the host to a volume
- The files from the host folder will be present in the volume
- Changes made on the host are reflected inside the container volume
- Syntax
  `docker run -v [host path]:[container path]:[rw|ro]`
- `rw` or `ro` controls the write status of the volume

Simple Example

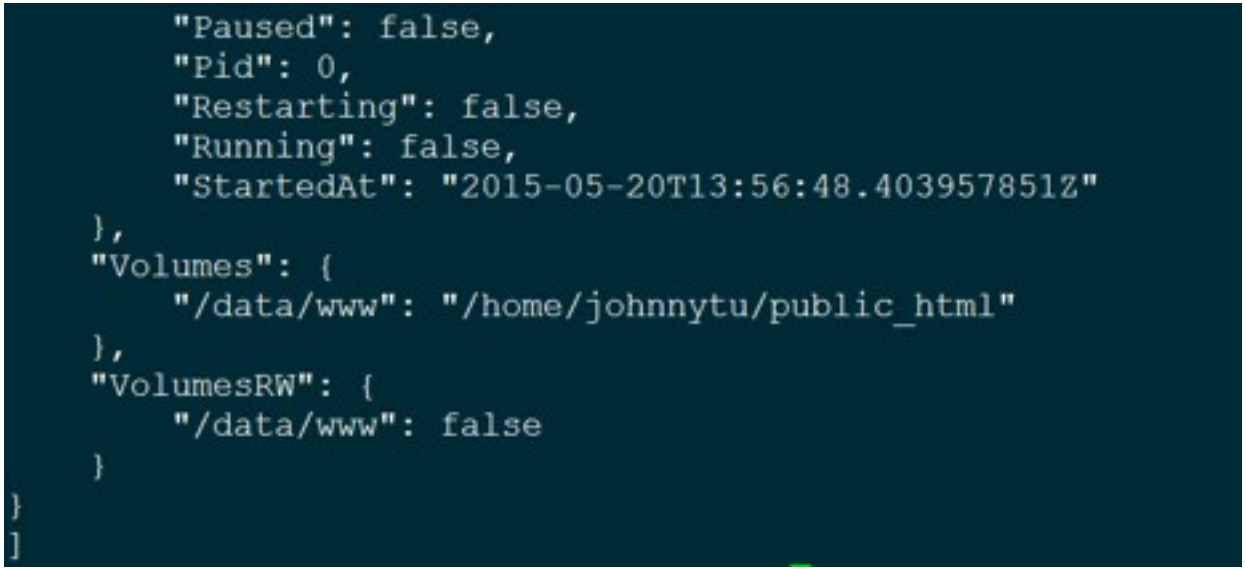
- In the example below, files inside `/home/user/public_html` on the hosts will appear in the `/data/www` folder of the container
- If the host path or container path does not exist, it will be created
- If the container path is a folder with existing content, the files will be replaced by those from the host path

Mount the contents of the public_html folder on the hosts to the container volume at /data/www

```
docker run -d -v /home/user/public_html:/data/www ubuntu
```


Inspecting the mapped volume

- The volumes field from `docker inspect` will show the container volume being mapped to the host path specified during `docker run`



A more practical example with NGINX

- Let's run an NGINX container and have it serve web pages that we have on the host machine
- That way we can conveniently edit the page on the host instead of having to make changes inside the container
- Quick NGINX 101
  - NGINX starts with a one default server on port 80
  - Default location for pages is the `/usr/share/nginx/html` folder
  - By default the folder has an `index.html` file which is the welcome page

The NGINX welcome page

`index.html` page in folder `/usr/share/nginx/html`

Running the container

- Key aspects to remember when running the container
  - Use `-d` to run it in detached mode
  - Use `-P` to automatically map the container ports (more on this in Module 9)
- When we visit our server URL we should now see our `index.html` file inside our public_html folder instead of the default NGINX welcome page

Run an nginx container and map the our public_html folder to the volume at the /usr/share/nginx/html folder in the container. <path> is the path to public_html

```
docker run -d -P -v <path>:/usr/share/nginx/html nginx
```


Running the container (More Examples)

```
docker run -it -v ~/.bash_history:/.bash_history my image

docker run --rm -v $(pwd):/myfiles busybox sh

docker run --rm --volumes-from john1 -v $(pwd):/backup busybox tar cvf /
backup/john2.tar /john
```


Use cases for mounting host directories

- You want to manage storage and snapshots yourself.
  - With LVM, or a SAN, or ZFS, or anything else!)
- You have a separate disk with better performance (SSD) or resiliency (EBS) than the system disk, and you want to put important data on that disk.
- You want to share your source directory between your host (where the source gets edited) and the container (where it is compiled or executed).
  - Good for testing purposes but not for production deployment

Volumes in Dockerfile

- VOLUME instruction creates a mount point
- Can specify arguments in a JSON array or string
- Cannot map volumes to host directories
- Volumes are initialized when the container is executed

String example

```
VOLUME /myvol
```

String example with multiple volumes

```
VOLUME /www/website1.com /www/website2.com
```

JSON example

```
VOLUME ["myvol", "myvol2"]
```


Example Dockerfile with Volumes

- When we run a container from this image, the volume will be initialized along with any data in the specified location
- If we want to setup default files in the volume folder, the folder and file must be created first

```
FROM ubuntu:14.04

RUN apt-get update
RUN apt-get install -y vim \
    wget

RUN mkdir /data/myvol -p && \
    echo "hello world" > /data/myvol/testfile
VOLUME ["/data/myvol"]
```


Data containers

- A data container is a container created for the purpose of referencing one or many volumes
- Data containers don’t run any application or process
- Used when you have persistent data that needs to be shared with other containers
- When creating a data container, you should give it a custom name to make it easier to reference

Custom container names

- By default, containers we create, have a randomly generated name
- To give your container a specific name, use the `--name` option on the `docker run` command
- Existing container can be renamed using the `docker rename` command
  `docker rename <old name> <new name>`

Create a container and name it mynginx

```
docker run -d -P --name mynginx nginx
```

Rename the container called happy_einstein to mycontainer

```
docker rename happy_einstein mycontainer
```


Creating data containers

- We just need to run a container and specify a volume
- Should run a container using a lightweight image such as busybox
- No need to run any particular process, just run "`true`"

Run our data container using the busybox image

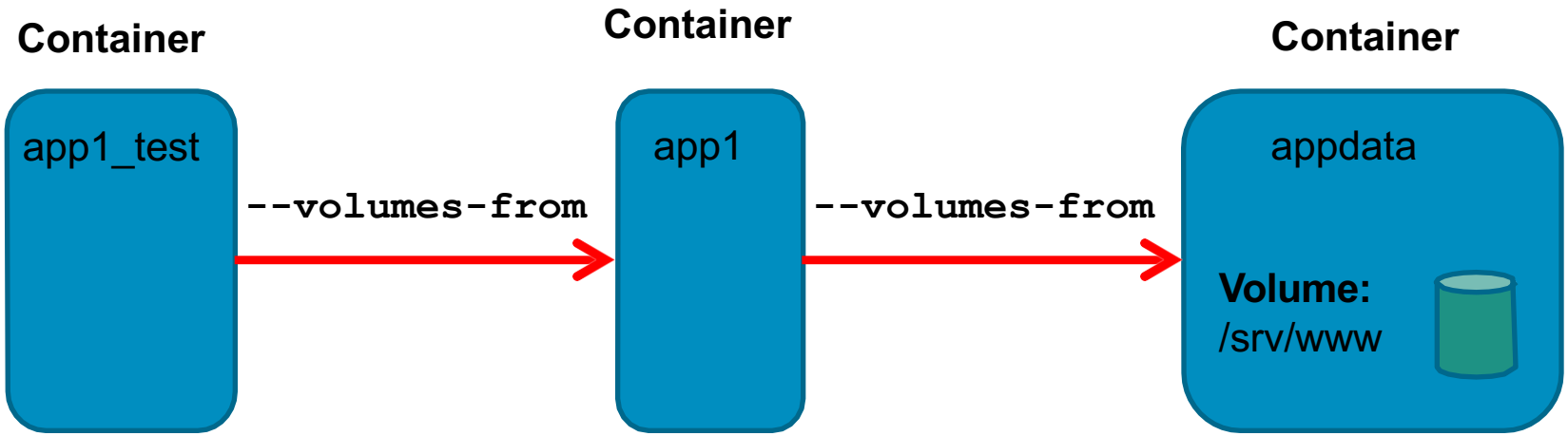
```
docker run --name mydata -v /data/app1 busybox true
```


Using data containers

- Data containers can be used by other containers via the `--volumes-from` option in the `docker run` command
- Reference your data container by its container name. For example: `--volumes-from datacontainer …`

```
$ docker run --name appdata -v /data/app1 busybox
65d22a45d8ca11d77eed0faa8689d511a2265de1790e9bc41302378d0ac93c75

johnnytu@docker-ubuntu:~$ docker run -it --volumes-from appdata ubuntu:14.04
root@4ef756eeb298:/#
root@4ef756eeb298:/# cd /data/app1/
root@4ef756eeb298:/data/app1#
```


Chaining containers

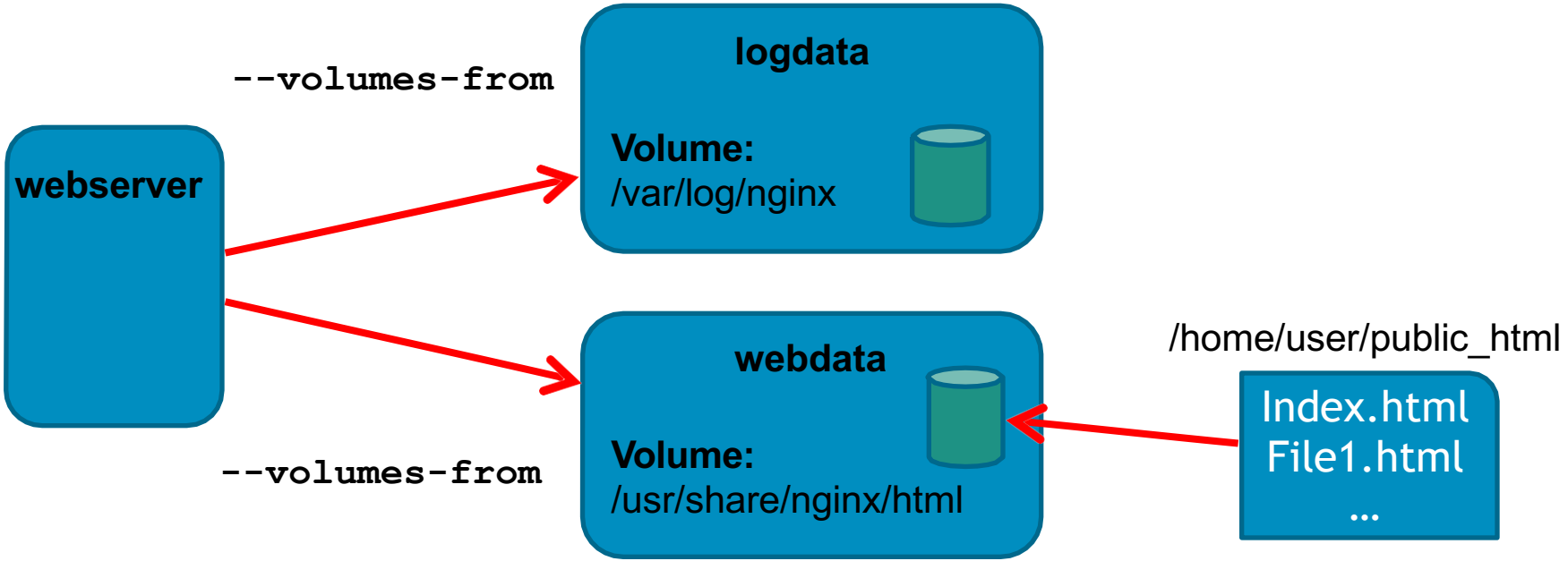


- Container `app1` will mount the `/srv/www` volume from `appdata`
- Container `app1_test` will mount all volumes in `app1`, which will be the same `/srv/www` volume from `appdata`

A more practical example

- Let’s run an NGINX container but separate its operation across multiple containers
  - One container to handle the log files
  - One container to handle the web content that is to be served
  - One container to run NGINX

Diagram of example



Backup your data containers

- It’s a good idea to back up data containers such as our logdata container, which has our NGINX log files
- Backups can be done with the following process:
  - Create a new container and mount the volumes from the data container
  - Mount a host directory as another volume on the container
  - Run the tar process to backup the data container volume onto your host folder

```
$ docker run --volumes-from logdata \
    -v /home/johnnytu/backups:/backup \
    ubuntu:14.04 \
    tar cvf /backup/nginxlogs.tar /var/log/nginx
```


Volumes defined in images

- Most images will have volumes defined in their Dockerfile
- Can check by using `docker inspect` command against the image
- `docker inspect` can be run against an image or a container
- To run against an image, specify either the image repository and tag or the image id.

Inspect the properties of the ubuntu:14.04 image

```
docker inspect ubuntu:14.04
```

OR

```
docker inspect <image id>
```


Inspecting an image


```
        "Hostname": "d05bc6205be4",
        "Image": "e59ba510498bb53d2298ccc585b3140f4072f91070cd9b1c2bb504be87a4985b",
        "Labels": {},
        "MacAddress": "",
        "Memory": 0,
        "MemorySwap": 0,
        "NetworkDisabled": false,
        "OnBuild": [],
        "OpenStdin": false,
        "PortSpecs": null,
        "StdinOnce": false,
        "Tty": false,
        "User": "",
        "Volumes": {
            "/var/cache/nginx": {}
        },
        "WorkingDir": ""
    },
    "Created": "2015-04-30T23:05:04.539666944Z",
    "DockerVersion": "1.6.0",
    "Id": "42a3cf88f3f0cce2b4bfb2ed714eec5ee937525b4c7e0a0f70daff18c3f2ee92",
    "Os": "linux",
    "Parent": "e59ba510490bb53d2290ccc585b3140f4072f91070cd9b1c2bb504be87a4985b",
    "Size": 0,
    "VirtualSize": 132822037
}
]
```


Module summary

- Volumes can be mounted when we run a container during the `docker run` command or in a Dockerfile
- Volumes bypass the copy of write system
- We can map a host directory to a volume in a container
- A volume persists even after its container has been deleted
- A data container is a container created for the purpose of referencing one or many volumes

Module 8:
Container Networking

Module objectives

In this module we will

- Explain the Docker networking model for containers
- Learn how to map container ports to host ports manually and automatically
- Learn how to link containers together

Docker networking model

- Containers cannot have a public IPv4 address
- They are allocated a private address
- Services running on a container must be exposed port by port
- Container ports have to be mapped to the host port to avoid conflicts

- (Pre Libnetwork)

The docker0 bridge

- When Docker starts, it creates a virtual interface called `docker0` on the host machine
- `docker0` is assigned a random IP address and subnet from the private range defined by RFC 1918

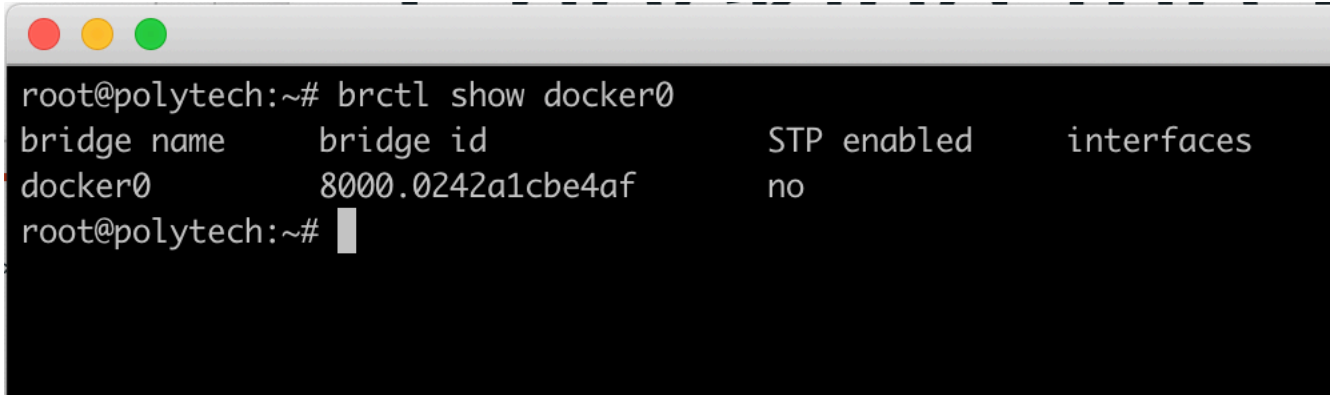
```
johnnytu@docker-ubuntu:~$ ip a
…
…
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state
DOWN group default
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::5484:7aff:fefe:9799/64 scope link
       valid_lft forever preferred_lft forever
```

The docker0 bridge

- The `docker0` interface is a virtual Ethernet bridge interface
- It passes or switches packets between two connected devices just like a physical bridge or switch
  - Host to container
  - Container to container
- Each new container gets one interface that is automatically attached to the `docker0` bridge

Checking the bridge interface

- We can use the `brctl` (bridge control) command to check the interfaces on our `docker0` bridge
- Install `bridge-utils` package to get the command
  `apt-get install bridge-utils`
- Run
  `brctl show docker0`



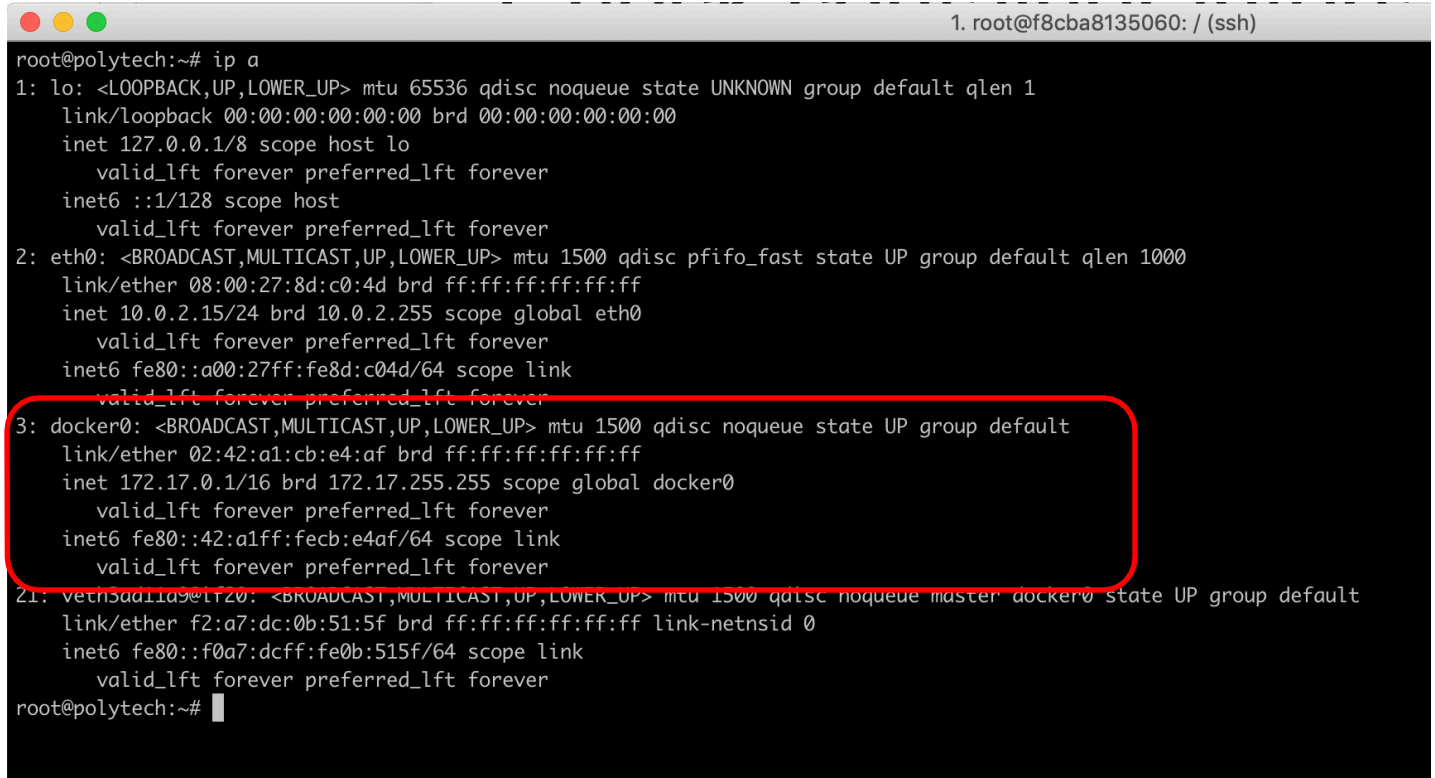
Checking the bridge interface

- Spin up some containers and then check the bridge again

```
root@polytech:~# docker run -it ubuntu:14.04
root@4044037139ee:/# root@polytech:~#
root@polytech:~#
root@polytech:~# docker run -it ubuntu:14.04
root@1f360ca99363:/#
root@1f360ca99363:/# root@polytech:~#
root@polytech:~# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.0242a1cbe4af       no              veth3ad11a9
                                                        vethab15bc3
                                                        vethd8b2a19
                                                        vethdea8a8b

root@polytech:~#
```

Check container interface

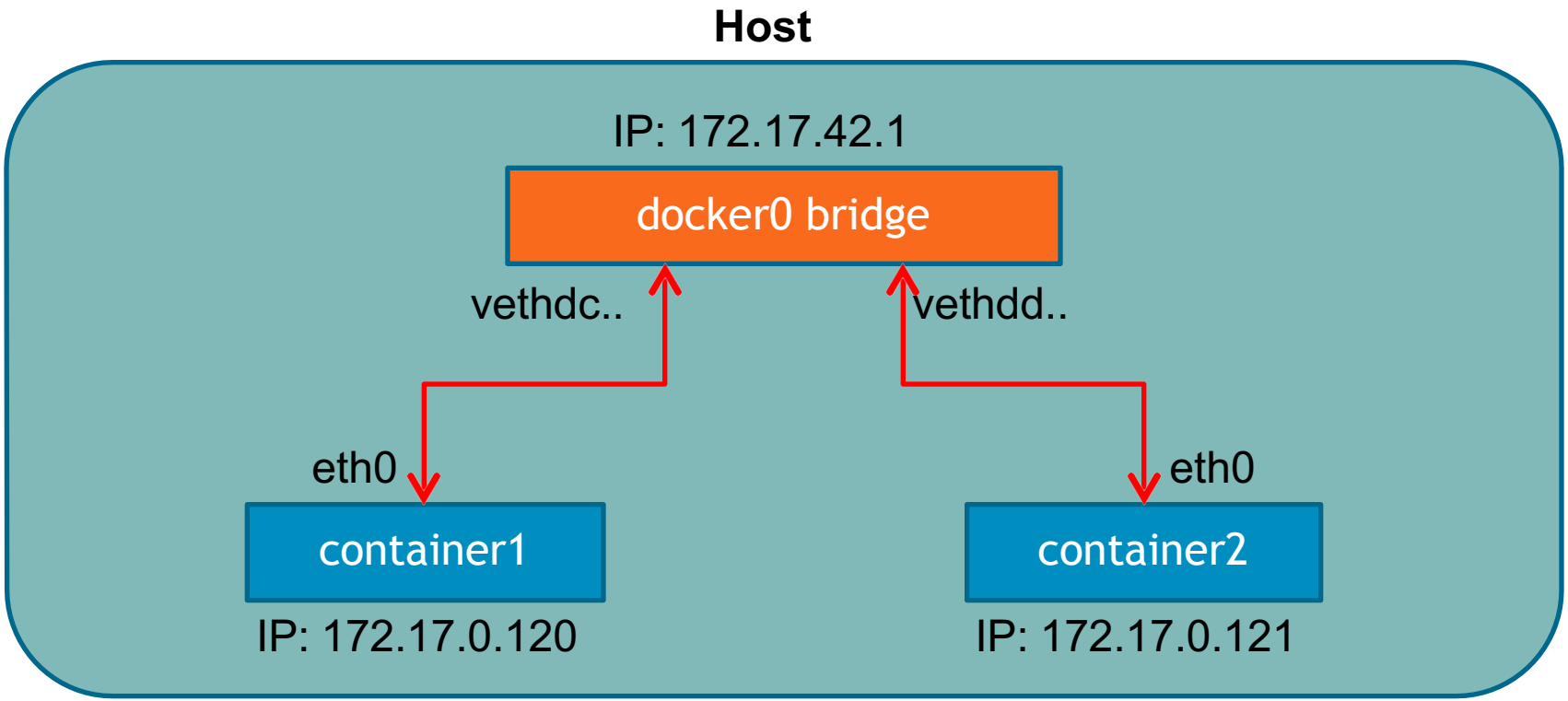
- Get into the container terminal and run the `ip a` command



Check container networking properties

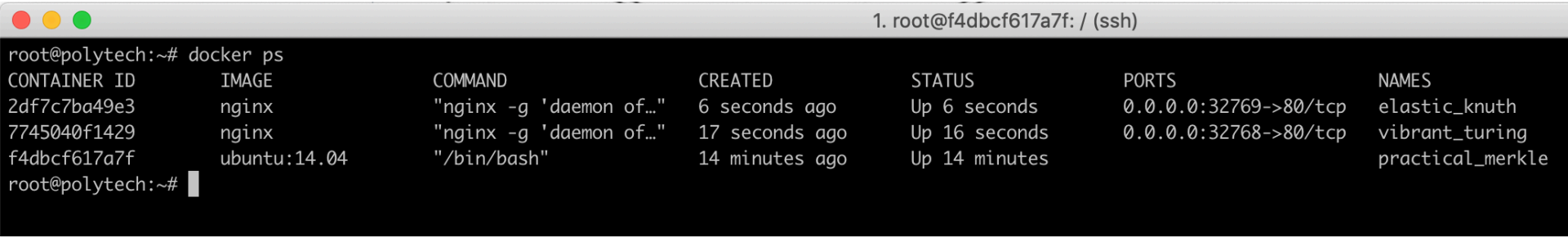
- Use `docker inspect` command and look for the `NetworkSettings` field

Diagram of networking model



Mapping ports

- **Revision:** containers have their own network and IP address
- Map exposed container ports to ports on the host machine
- Ports can be manually mapped or auto mapped
- You can see the port mapping for each container on the `docker ps` output



Manual port mapping

- Uses the `-p` option (smaller case p) in the `docker run` command
- Syntax
  `-p [host port]:[container port]`
- To map multiple ports, specify the `-p` option multiple times

Map port 8080 on the tomcat container to port 80 on the host

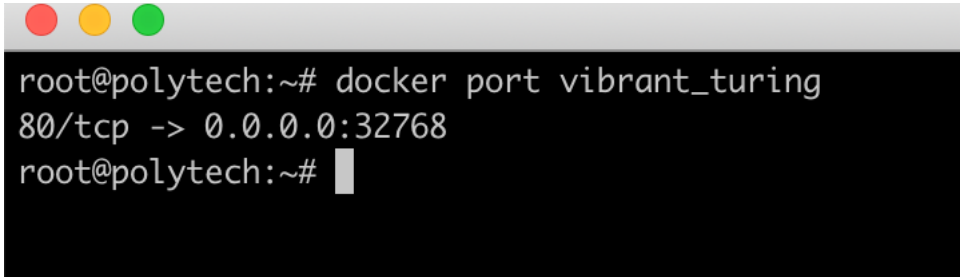
```
docker run -d -p 80:8080 tomcat
```

Map port on the host to port 80 on the nginx container and port 81 on the host to port 8080 on the nginx container

```
docker run -d -p 80:80 -p 81:8080 nginx
```


Docker port command

- `docker ps` output is not very ideal for display port mappings
- We can use the `docker port` command instead



Automapping ports

- Use the `-P` option in `docker run` command
- Automatically maps exposed ports in the container to a port number in the host
- Host port numbers used go from 49153 to 65535
- Only works for ports defined in the Dockerfile EXPOSE instruction

Auto map ports exposed by the NGINX container to a port value on the host

```
docker run -d -P nginx:1.7
```


EXPOSE instruction

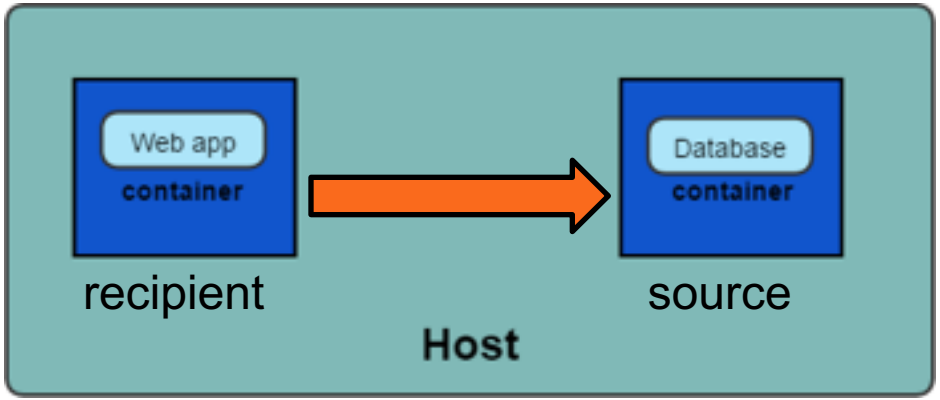
- Configures which ports a container will listen on at runtime
- Ports still need to be mapped when container is executed

```
FROM ubuntu:14.04
RUN apt-get update
RUN apt-get install -y nginx

EXPOSE 80 443

CMD ["nginx", "-g", "daemon off;"]
```


Linking Containers

Linking *is a communication method between containers which allows them to securely transfer data from one to another*

- Source and recipient containers
- Recipient containers have access to data on source containers
- Links are established based on container names



Uses of Linking

- Containers can talk to each other without having to expose ports to the host
- Essential for micro service application architecture
- Example:
  - Container with Tomcat running
  - Container with MySQL running
  - Application on Tomcat needs to connect to MySQL

Creating a Link

1. Create the source container first
2. Create the recipient container and use the `--link` option

- **Best practice –** give your containers meaningful names
- Format for linking
  `name:alias`

Create the source container using the postgres

```
docker run -d --name database postgres
```

Create the recipient container and link it

```
docker run -d -P --name website --link database:db nginx
```


The underlying mechanism

- Linking provides a secure tunnel between the containers
- Docker will create a set of environment variables based on your --link parameter
- Docker also exposes the environment variables from the source container.
  - Only the variables created by Docker are exposed
  - Variables are prefixed by the link alias
  - `ENV` instruction in the container Dockerfile
  - Variables defined during `docker run`
- DNS lookup entry will be added to `/etc/hosts` file based on your alias

Environment variables in source container

- Let's inspect the variables defined in our "appserver" Tomcat container

```
$ docker exec appserver env
PATH=/usr/local/tomcat/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=91794033c488
LANG=C.UTF-8
JAVA_VERSION=7u79
JAVA_DEBIAN_VERSION=7u79-2.5.5-1~deb8u1
CATALINA_HOME=/usr/local/tomcat
TOMCAT_MAJOR=8
TOMCAT_VERSION=8.0.22
TOMCAT_TGZ_URL=https://www.apache.org/dist/tomcat/tomcat-8/v8.0.22/bin/apache-tomcat-8.0.22.tar.gz
HOME=/root
```


Environment variables in recipient container

- Now we will check the environment variables in the Ubuntu container which is linked to our “appserver” container

```
$ docker exec client env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=0a17932c340d
TERM=xterm
APPSERVER_PORT=tcp://172.17.0.133:8080
APPSERVER_PORT_8080_TCP=tcp://172.17.0.133:8080
APPSERVER_PORT_8080_TCP_ADDR=172.17.0.133
APPSERVER_PORT_8080_TCP_PORT=8080
APPSERVER_PORT_8080_TCP_PROTO=tcp
APPSERVER_NAME=/client/appserver
APPSERVER_ENV_LANG=C.UTF-8
APPSERVER_ENV_JAVA_VERSION=7u79
APPSERVER_ENV_JAVA_DEBIAN_VERSION=7u79-2.5.5-1~deb8u1
APPSERVER_ENV_CATALINA_HOME=/usr/local/tomcat
APPSERVER_ENV_TOMCAT_MAJOR=8
APPSERVER_ENV_TOMCAT_VERSION=8.0.22
APPSERVER_ENV_TOMCAT_TGZ_URL=https://www.apache.org/dist/tomcat/tomcat-8/v8.0.22/bin/apache-tomcat-8.0.22.tar.gz
HOME=/root
```

Connection information created from the --link option

Created from source container

Module summary

- Docker containers run in a subnet provisioned by the docker0 bridge on the host machine
- Auto mapping of container ports to host ports only applies to the port numbers defined in the Dockerfile EXPOSE instruction
- Linking provides a secure method of communication between containers
- Key Dockerfile instructions
  - `EXPOSE`

Module summary

- Docker Compose makes it easier to manage micro service applications by making it easy to spin up and manage multiple containers
- Each service defined in the application is created as a container and can be scaled to multiple containers
- Docker Compose can create and run containers in a Swarm cluster

Module 9:
Container Orchestration

~~Docker Swarm~~ Kubernetes

- Go language
- Fork from Google Borg
- CNI Plugins
- Prototypage avec Minikube
- Offre SaaS : GKE, EKS, AKZ, …

Further Information

Additional resources

- Docker homepage - http://www.docker.com/
- Docker Hub - https://hub.docker.com
- Docker documentation - http://docs.docker.com/
- Docker Getting Started Guide - http://www.docker.com/gettingstarted/
- Docker code on GitHub - https://github.com/docker/docker
- Kubernetes - https://kubernetes.io/
- Minikube - https://kubernetes.io/docs/setup/minikube/

THANK YOU