Containers

Container Engines

docker

LXC

rkt

podman

What are containers? (High-Level Answer)

- Collective Term for “Containerized Application”
- Collection of processes that create a contained, pseudo-environment that “looks like” and “behaves like” a complete system
- Spawned from “Images” or built from Dockerfiles
- Allows one to very easily and quickly run applications without having to install them to evaluate the application

What are containers

- A collection of files, kernel namespaces, cgroups and process isolation coordinated by a runtime environment (The Docker Engine)
- Disks: Collection of files that behave as a layered disk
- Network: Virtual Switches, Virtual Interfaces
- CPU/RAM: Host
- Operating System as an App
- Just enough OS to support and run a **singular process**
- Can be used Interactively or as a background Service
- Similar to Virtual Machines (but not!)

Containers vs. Virtual Machines

Virtual Machines

- Boots a full OS kernel
- Requires Type 1 or Type 2 Hypervisor (Operating System)
- Virtual (Software) Hardware needs:
  - Drivers
  - Updating
  - Rebooting
- Only “Loosely-Coupled” Automated Configuration
- Only single, live configuration at a time
- Only Real Integration with External Environment is through TCP/IP Stack or hypervisor

Containers

- Shares host kernel
- Only a Runtime (no hypervisor)
- No Virtual (software) Hardware
- Version Controllable, “Source Code” Build Files Possible
- RO Images / RW Container Wall
- Completely Automated Build process
- API Access for Integration and Management
- Smaller and less resource-intensive

What are containers good for?

- *Reproducibility*
  - System Can Be Saved along WITH your code
- *Avoid “Dependency Hells”*
  - Easily use Collections or Combinations of Applications or Libraries
- Use multiple containers with different and/or conflicting environments at the same time.
  - R 2.6, 3.0, 3.5
  - Python 2.6, 3.5.7
  - (in theory) DOS 5 with MacOS 10
- Have a consistent environment to run applications
- Run applications in an environment exactly as the developer intended

What are containers good for? (pt. 2)

- Code Testing
  - “What steps are needed to run my code on RedHat 8?”
- Encapsulate a workflow
  - Long workflows (5-40 apps) can have many underlying applications and libraries that will eventually need to be upgraded/updated
- “Collections” of Softwares in One Object
  - Python+Tensorflow
  - Hadoop+HDFS+HBase+Spark
- Version Control
  - Dockerfiles
  - Configuration Files
- Creating a Highly Portable environment for Code

Dockerhub - http://hub.docker.com

- Hosts pre-built containers
- Official and unofficial
  - centos:7
  - random_guy/random_app:broken
- Searchable and community rated
  - From CLI: “docker search”
  - From web: https://hub.docker.com/
- Information and Metadata for the images
  - Run suggestions and requirements (Environment Variables, Volumes, Port Numbers)
  - Maintainer and support email
  - Discord servers
  - etc…
- Hosting for your images

Reproducibility

- Container Images are Read Only
- Any Change to a Container Build file can be Version Controlled to increase the likelihood that running it at a certain revision will always result in the same versions of applications and libraries installed. Even if those are not currently backed by distro repositories
- Entire workflows, including the OS, can be containerized and shared as a singular, independant object. Steps to build it can be analyzed and reproduced exactly as they were built.
- Container Images are versioned (e.g.: centos:7, centos:7.44159, etc)
- Container Images can be stored (images are large-ish, but as small as they can be)
- Container makefiles (Dockerfiles, Singularity SPECs, etc) are version controlled
- Application-level configurations are text files which can be version controlled alongside code
- Specific versions of libraries and applications can be installed using Dockerfiles

Dependency Hell - Flavors (an acquired taste)

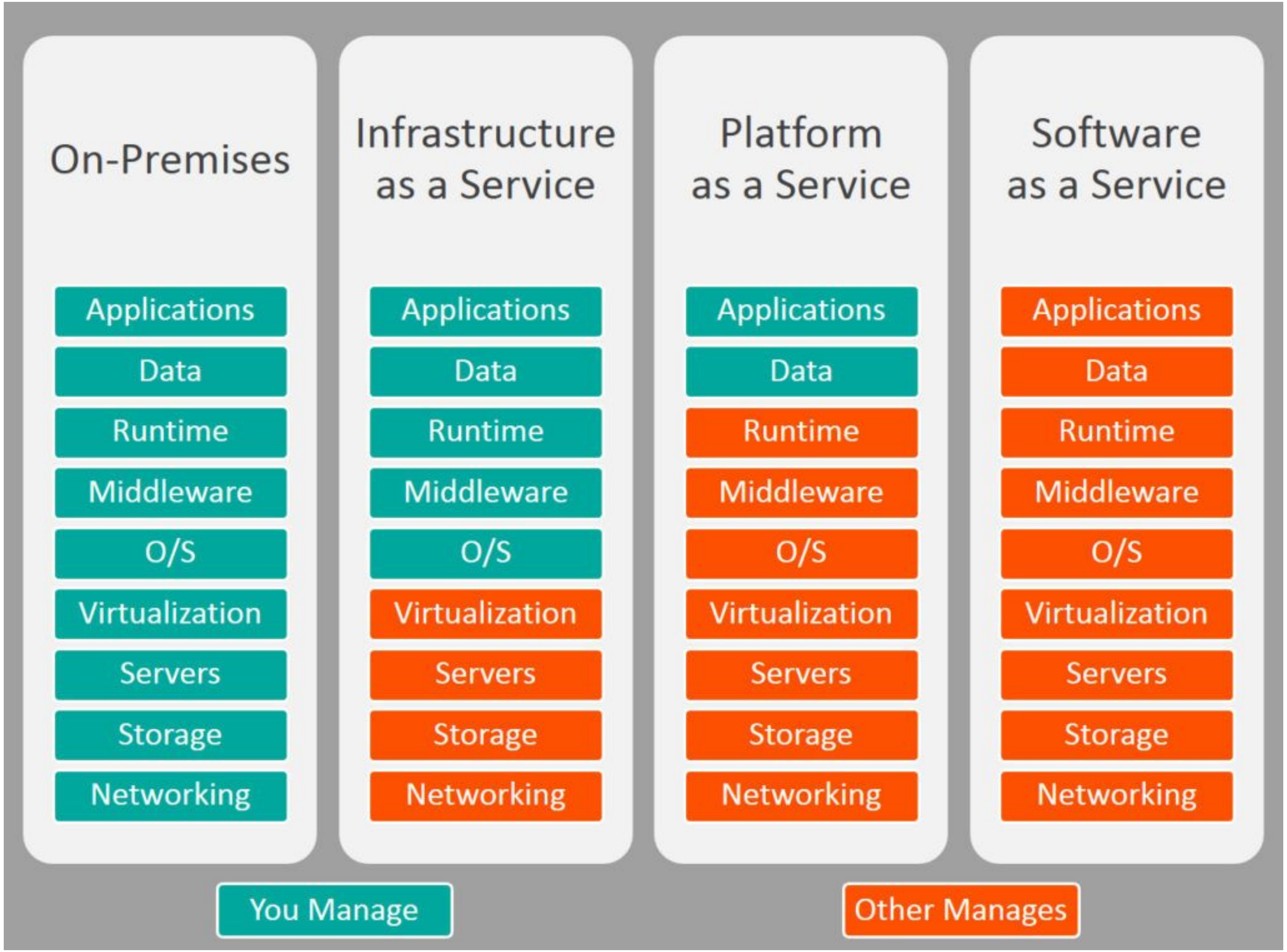
Operating System Version DH

- Ubuntu vs. Redhat vs Arch vs Debian
- Ubuntu 14.04, 16.10, 18.04 ….
- Redhat/CentOS/Fedora 5, 6, 7, 8-beta
- Windows vs Linux
- (conceptually) MacOS vs Windows

Software/Library Related DH

- Unsolvable Library Circular Depends
  - App1 Needs App2
  - App1 Needs library x
  - Library x needs library y
  - App2 is incompatible with library y
- Libraries that break other apps on your computer
- Library Quality
  - Beta Libraries / Alpha-quality Libraries
- Ephemeral Libraries (Don’t want to keep around)
- Multi-User Systems (User x needs...User y needs)

Service Model Implementations


On-Premises
Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
Infrastructure as a Service
Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
Platform as a Service
Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
Software as a Service
Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
You Manage
Other Manages

SaaS Workflows: Do it for me.

1. Pull docker image for the application (example: wordpress/wordpress:latest)
   a. Image will include base OS and all supporting libraries
2. Mostly no or little configuration
   a. Environment Variables, mostly
3. Mount volume to appropriate place in container when running it
4. Set environment variables to pass to container
5. Run the container
6. Container serves webpages or acts on content from the mounted volume, live.

PaaS Workflows: Do it *with* me.

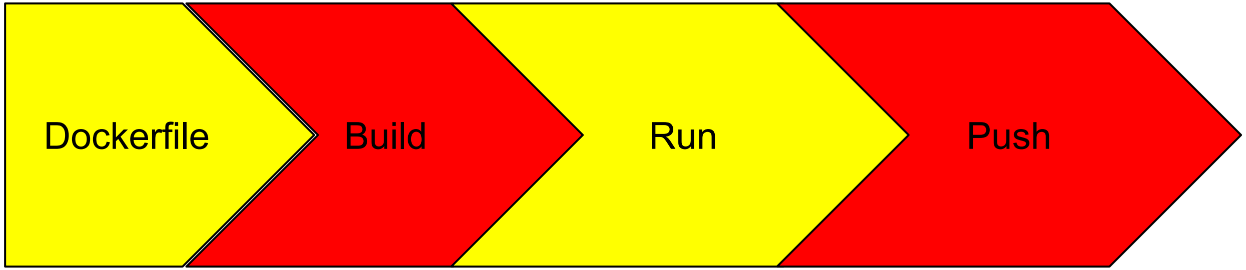
1. Create Dockerfile to extend on existing platform-level container (nginx/nginx:latest)
   a. Add configuration, plugins, extensions and files as-needed
2. Build an Image using Dockerfile. Commit and push to repository or Dockerhub as required.
3. Place customized code, configuration, content files and directories in bind-mountable directory. Or include them within the image for maximum portability
   a. Volume can be updated dynamically
4. Run the container with environment and volumes if necessary
5. Container serves webpages performs activity using customized, site-specific configurations

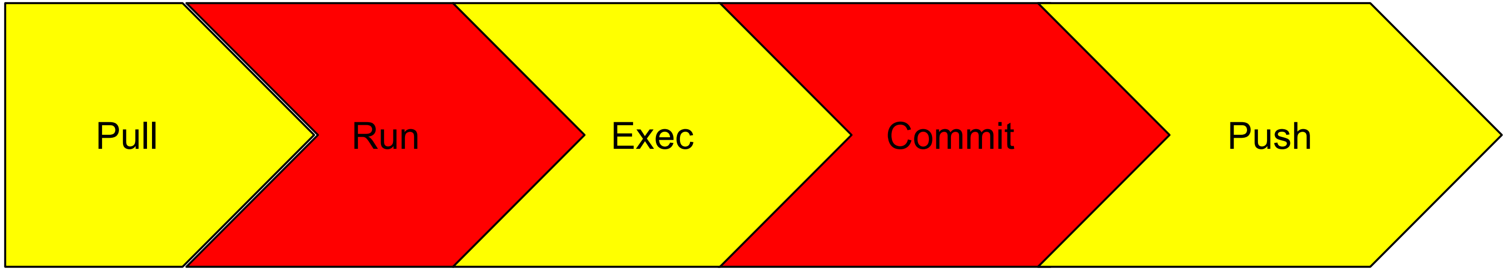
Iaas Workflows: Do it *yourself*.

1. Create Dockerfile to extend on existing, official operating system container
    a. Examples: ubuntu/ubuntu:14.04, ubuntu/ubuntu:latest, centos/centos:6, centos/centos:7.14493
2. Use many RUN, CMD and COPY commands to “build up” a full system image before installing the application needed.
3. Place customized code, subject files and directories in bind-mountable volume. Or include them within the image for maximum portability
    a. Files in volume can be updated dynamically as-needed
4. Run the container with environment and volumes if necessary
5. Use the image and container as a base for developing future derivatives by committing to a local repository

Workflows

Two Workflows: Container Images and Dockerfile





Two Workflows: Container Images and Dockerfile

Container Images

- Natural way to build an “image”
- Can be done interactively, in logical order: install x, install y, install my application
- Tolerant to mistakes
- Only the final product need be saved
- Can be committed, pushed and pulled to container registries
- All-In-One

Dockerfile

- “Makefile” for container images
- Text-files can be edited in any editor
- Version control with any version-control system
- Simple, small, powerful set of commands
- Built layer-at-a-time
- Often has extra files
- Very Tedious (layers need to be rebuilt)

Third Possible Workflow: Separate Dev and Ops

- Keep container development separate repositories or branches
- Single or Multiple bind-mount git repository or repositories for project files
- Very simple container upgrades
- Difficult to test

Docker Technical Overview

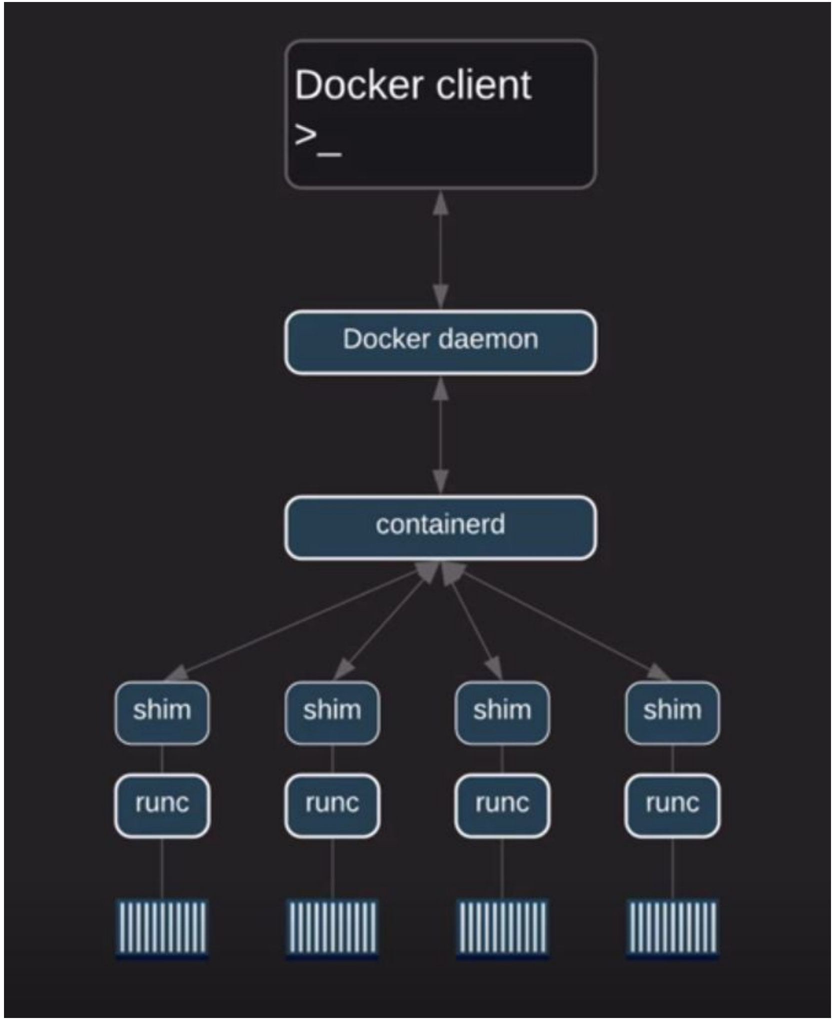
Definitions

- **Docker Image** : On-disk, read-only, layered, post-build container file. Never altered - only added to layer-by-layer later through commits.
- **Container:** In-memory, read/write representation of an on-disk image. Cannot write back without a commit. Commits create new layers on image. Every “run” is a new container. 50 runs,, 50 containers, same image.
- **Entrypoint Process:** (instead of init) The single process that a container environment is built to support. **MUST NOT** run in “the background”. If process exits, the container exits.
- **Container Tags:** Metadata describing the repo(s), name and version. OR short, site-relevant descriptions
- **Commit:** Taking in-memory image and applying changes to a layer of the locally cached base (FROM) image, resulting in a new, local image waiting to be committed to a repository. (similar to git).

- **Repository:** Name of the project that encompasses versioned or specialized containers. Also used to describe the repository where commits are stored/uploaded to

Technical Components of Docker

- The Docker Machine
  - dockerd
  - runc
  - containerd
  - shim
- Metadata
  - Names
  - Tags
  - IDs
  - inspect
- Logging
- Networking
  - Types: NAT (bridge), host-based
  - Ports
- Environment Variables
- Storage
  - Bind Mounts
  - Docker Volumes



- dockerd (Docker Daemon)
  - Uses gRPC (CRUD API) to Accept CLI and Instructs containerd
- containerd
  - Starts, stops, pauses, deletes
  - Forks one runc for each container
- shim
  - Used to Enable Many Containers to be Run at One Time (Limitation of LXC)
  - Becomes Parent for Container AFTER runc Exits
  - STDIN/STDOUT
- runc
  - OCI (Open Container Initiative) Specifications
  - Forked from containerd
  - Keeps container alive if daemon needs to be restarted
  - Interfaces with Kernel to acquire cgroups

Docker Internal Storage

- Docker/Container engine supports multiple “Storage Drivers”
  - AuFS
  - OverlayFS
  - UnionFS
- OverlayFS
  - Actual Files in a Directory for Layers - mounted as a single “disk”
- Copy-On-Write
- /var/lib/docker/containers
  - Prime Images for Containers
- /var/lib/docker/overlay2
  - Many, Many Layters
  - Overlays
  - Actual files are accessible here per layer

Container External Storage: volumes or binds

Volumes

- Managed by docker
- Maintained on local disk
- Persistent across container restarts
- Available as ordinary files which can be backed up using standard backup applications
- docker volume create <name>

Bind Mounts

- Local directories “bound” to directories inside of the container
- Files and directories under local directory are updated in real-time with bind mounts
- Persistent across container restarts
- Good for keeping configuration files on storage systems with RAID, snapshots and backups
- Example: docker -b mydir:/container/dir run <container>

Container Networking

- Full networking - everything you would “expect”
- Containers have network interfaces like regular computers and VMs
- Containers have IP addresses assigned to them by their hosts’ dockerd
- When docker installs, the host gets a virtual switch for docker to connect its interfaces to
- In a Dockerfile, the EXPOSE keyword will make a port available to be published. *This does not automatically make it accessible from outside the host*
- “-p” or “-P” **must** be used to access the exposed port. If “-P”, port is randomized
- Most common and default model is NAT port forwarding on the vswitch but there are other types: macvlan, host, none, etc

Environment Variables

- Used to configure Dockerhub container applications internally
- Variables given on the command line are used within the container
- Set during container run or exec
- -e VARIABLE=value
- Dockerfiles: ENV

Metadata: Names

- Name on Dockerhub (tag) vs local name
- Names are not tags
- Image names are NOT tied to container names, but often used to
- Names may be the same as tags
- If names are not specified they will be automatically generated during “docker exec” or “docker run”.
- Container names are a combination of <adjective>_<scientist>

Metadata: Docker Tags

{repository} / [project] : {tag}

Examples:

- busybox/busybox:latest
- busybox/busybox
- busybox
- busybox/busybox:3.2.1
- busybox/busybox:my-cool-branch

Metadata: Inspect

docker inspect [NAME|ID]

- Displays insane amount of information about running container
- Expected to be “grepped” or searched with “--format”
- JSON format
- Example: `docker inspect --format='{{range .NetworkSettings.Networks}}{{.MacAddress}}{{end}}' <container>`

Dockerfiles

- “Makefiles for containers”
- Simple core language with many optional keywords
- Every RUN is a new layer
- Should be able to create a complete and independent image which should be identical to what is committed to an image registry
  - Changes *can*, technically, be made to images in-memory and committed from there, however it is good practice to ensure that changes made to in-memory containers be replicated to the Dockerfiles
- Must be named “Dockerfile”

Dockerfile Example

```
FROM ubuntu:18.04
MAINTAINER Curtis E Combs Jr <curtis.combs@moffitt.org>
RUN apt-get update
RUN apt-get install -y nginx
COPY nginx.conf /etc/nginx/nginx.com
COPY index.html /usr/share/nginx/html/
EXPOSE 80
ENTRYPOINT [“/usr/sbin/nginx”]
```

docker build -t myrepo/myweb:latest .

Dockerfiles - Most Common Commands

Dockerfile Instructions

- **FROM** - Derive your container from another container hosted on Dockerhub
- **RUN** - Run a command required to build the container
- **ENV** - Sets environment variables
- **EXPOSE** - Exposes any network ports
- **ENTRYPOINT** - The central process

Container Building

- **docker build .** - Build environment from Dockerfile in current dir
- **docker pull** - Pull container from repository
- **docker commit** - Commit container changes to image
- **docker push** - Push container to repository defined in tag

Enough Theory:
Let’s make some containers

docker run -ti --rm r-base

(docker run -ti --rm dockerhub.io/r-base:latest)

```
docker run -ti --rm r-base:3.1.3
```

docker image ls

```
docker run --name myweb -p 80:8080 -d nginx
```

docker inspect myweb

(docker inspect myweb | grep IPAddress) http://localhost:8080

(http://<IPAddress>:8080)

docker container ls

docker container stop myweb

docker container ls

docker container ls -a

```
docker run --name myweb -p 80:8080 -d nginx
```

docker container ls

```
docker exec -ti myweb /bin/bash
```

(Install things. Write some files. Alter the configuration. Customize it.)

docker commit myweb myrepo:latest

docker push myrepo/myweb:latest

docker container stop myweb

docker container ls -a

docker volume create portainer_data

docker run -d -p 9000:9000 --name portainer --restart always -v /var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data portainer/portainer docker inspect portainer

(docker inspect portainer | grep IPAddress) http://localhost:9000

(http://<IPAddress>:9000)

Thank you!

https://docs.docker.com/get-started/
https://hub.docker.com
https://www.katacoda.com/courses/docker