



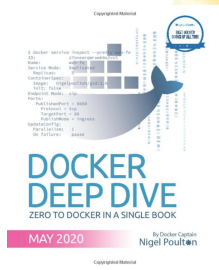
UMD DATA605 - Big Data Systems

Lesson 3.1: DevOps with Docker

Instructor: Dr. GP Saggese, gsaggese@umd.edu

Docker - Resources

- We will use Docker during the class project and most tutorials
- Concepts in the slides
- Class tutorials:
 - [tutorial_docker](#)
 - [tutorial_docker_compose](#)



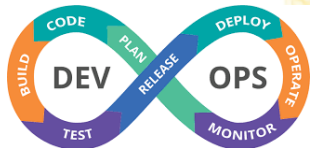
- Web resources:
 - [Docker Tutorial for beginners](#)
 - <https://labs.play-with-docker.com/>
 - <https://training.play-with-docker.com>
 - [A Beginner-Friendly Introduction to Containers, VMs and Docker](#)
 - [Official Docker Getting Started Tutorial](#)
- Mastery:
 - Poulton, [Docker Deep Dive: Zero to Docker in a single book](#), 2020

Application Deployment

- For (almost all) Internet companies, **the application is the business**
 - If the application breaks, the business stops
 - E.g., Amazon, Google, Facebook, online banks, travel sites, ...
- **Problem**
 - Release, deploy, manage, monitor applications
- **Solutions**
 - Before 2000s: “bare-metal era”
 - 2000s-2010s: “virtual machine era”
 - After ~2013: “container era”

DevOps

- **DevOps** = practices combining:
 - Software development (*dev*)
 - IT operations (*ops*)
- **Containers revolutionized DevOps**
 - Enable independence between app development and IT ops
 - One team creates an app
 - Another team deploys and manages apps
 - Foster collaboration and innovation
 - IT: "It doesn't work!"
 - Devs: "What? It works for me"



- Plan
- Code
- Build
- Test
- Release
- Deploy
- Operate
- Monitor

Run on bare metal

- **< 2000s**
 - Running few applications per server (no virtualization)
- **Pros**
 - No virtualization overhead
- **Cons**
 - Insecure due to no application separation
 - Expensive
- **Expensive / low efficiency**
 - New server for each application
 - Hard to spec machines → buy “big and fast servers”
 - Overpowered servers at 5-10% capacity
 - Significant spending during 2000 DotCom boom on machines and networks
- Resurfaced in 2020 with different Cloud Computing use cases
- **Winners:** Cisco, Sun, Microsoft



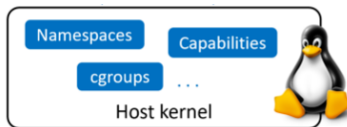
Virtual Machine Era

- **Circa 2000-2010: Virtual Machine**
 - Run multiple OS copies on the same hardware
- **Pros**
 - Run multiple apps safely on one server
 - Use existing servers with spare capacity
- **Cons**
 - Each VM needs an OS (wastes CPU, RAM, disk)
 - Buy OS license
 - Monitor and patch each OS
 - Slow boot times
- **Winners:** VMWare, RedHat, Citrix



Containers Era

- **Circa 2013: Docker becomes ubiquitous**
- **Docker**
 - Didn't invent containers
 - Made containers simple and mainstream
- Linux supported containers
 - Kernel namespaces
 - Control groups
 - Union filesystems
- **Pros**
 - Containers are fast and portable
 - Don't require full OS
 - All run on a single host
 - Reduce OS licensing cost
 - Reduce OS patching and maintenance
- **Cons**
 - CPU overhead
 - Toolchain to learn / use
- **Winners**: AWS, Microsoft Azure, Google (not Docker Inc.)



Serverless Computing

- **Containers run in an OS, OS runs on a host**
 - **Where is the host running?**
 - Local (your laptop)
 - On premise (your computer in a rack)
 - Cloud instance (e.g., AWS EC2)
 - **How is the host running?**
 - On bare-metal server
 - On a virtual machine
 - On a virtual machine running a virtual machine
- **Serverless computing**
 - Application runs without concern for “*how*” or “*where*”
 - E.g., AWS Lambda

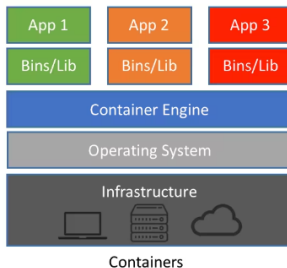
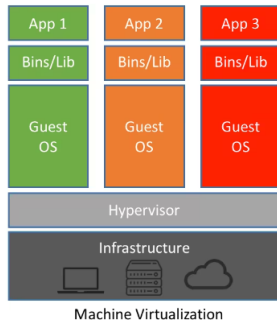
HW vs OS Virtualization

- **Hypervisor performs HW virtualization**

- Carves physical hardware into VMs
- Allocates CPUs, RAM, storage to a VM
- Like having multiple computers
- “Virtual machine tax”
 - Running 3 apps requires 3 VMs
 - Each VM takes time to start
 - Consumes CPU, RAM, storage
 - Needs OS license
 - Requires admins, patching
 - You just want to run 3 apps!

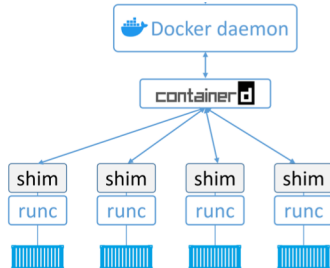
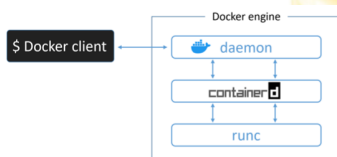
- **Containers perform OS virtualization**

- It's like having multiple OSes



Docker: Client-Server

- Docker relies on a **client-server architecture**
- **Docker client**
 - Command line interface
 - Communicate with server through IPC socket
 - E.g., `/var/run/docker.sock` or IP port
- **Docker engine**
 - Run and manage containers
 - Modular, built from OCI-compliant sub-systems
 - E.g., docker daemon, containerd, runc, plug-ins for networking and storage



Docker Architecture

- **Docker run-time**

- runc: start and stop containers
- containerd
 - Pull images
 - Create containers, volumes, network interfaces

- **Docker engine**

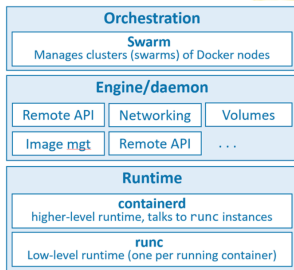
- dockerd
 - Expose remote API
 - Manage images, volumes, networks

- **Docker orchestration**

- docker swarm
- Manage clusters of nodes
- Replaced by Kubernetes

- **Open Container Initiative (OCI)**

- Standardize low-level components of container infrastructure
- E.g., image format, run-time API
- “Death” of Docker



Docker Container

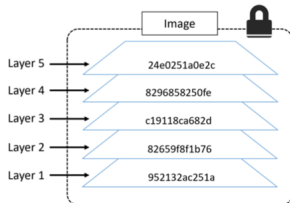
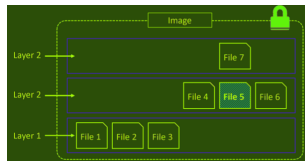
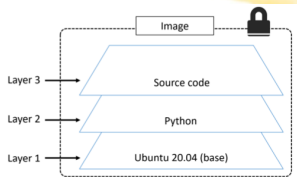
- **Docker Container**
 - Unit of computation
 - Lightweight, stand-alone, executable software package
- **Includes everything needed to run**
 - Code
 - Runtime/system libraries
 - Settings
- **Run-time object**
 - Docker images are build-time objects
 - Like program running (container) vs program code (image)

Docker Image

- **Docker Image**
 - Unit of deployment
- **Contains everything needed to run an app**
 - Application code
 - Application dependencies
 - Minimal OS support
 - File system
- Users can
 - Build images from Dockerfiles
 - Pull pre-built images from a registry
- Multiple layers stacked
 - Typically few 100s MBs

Docker Image Layers

- **Docker image** is a configuration file listing layers and metadata
 - Composed of read-only layers
 - Each layer is independent
 - Each layer comprises many files
- **Docker driver**
 - Stacks layers as a unified filesystem
 - Implements copy-on-write behavior
 - Files from top layers can obscure files from bottom layers
- **Layer hash**
 - Each layer has a hash based on content
 - Layers are pulled and pushed compressed
- **Image hash**
 - Each image has a hash
 - Hash is a function of the config file and layers
 - Image changes generate a new hash

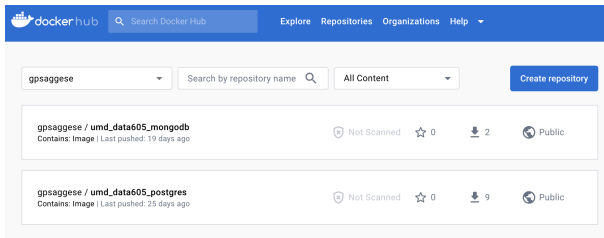
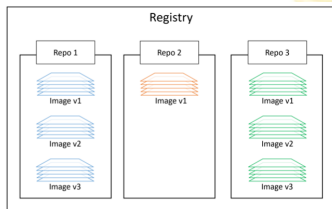


Docker: Container Data

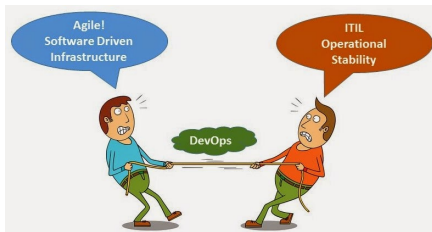
- A container has access to different data
- **Container storage**
 - Copy-on-write layer in the image
 - Ephemeral (temporary data)
 - Data persists until the container is killed
 - Stopping or pausing a container doesn't lose data
 - Containers are immutable
 - Avoid writing persistent data into containers
- **Bind-mount a local dir**
 - Mount a local dir to a dir inside a container
- **Docker volumes**
 - Volumes exist separately from the container
 - E.g., store Postgres DB content
 - State is permanent across container invocations
 - Shareable across containers

Docker Repos

- **Docker Repo (Registry)**
 - E.g., DockerHub, AWS ECR
 - Store Docker images
 - `<registry>/<repo>:<tag>`
 - E.g., `docker.io/alpine:latest`
 - Some repos are vetted by Docker
 - Unofficial repos shouldn't be trusted
 - E.g., <https://hub.docker.com/>



Devops = Devs + Ops



- **Devs**

- Implement app
 - Python, virtual env
- Containerize app
 - Create Dockerfile
 - Instructions to build image
- Build image
- Run app as container
- Test locally

- **Ops**

- Download container images
 - Filesystem, application, dependencies
- Start / destroy containers
- Reproduce issues easily
 - "Here is the log"
 - Run command line
 - Deploy on test system and debug

Containerizing an App

- **Containerizing an app** means creating a container with your app inside
- Develop application code with dependencies
 - Install dependencies
 - Inside a container
 - Inside a virtual env
- Create a Dockerfile describing:
 - App
 - Dependencies
 - How to run it
- Build image with `docker image build`
- (Optional) Push image to Docker image registry
- Run/test container from image
- Distribute app as a container (no installation required)

Building a Container

- **Dockerfile**
 - Describe how to create a container
- **Build context**
 - `docker build -t web:latest .` where `.` is the build context
 - Send directory containing the application to Docker engine to build the application
 - Typically the Dockerfile is in the root directory of the build context

Dockerfile: Example

```
FROM python:3.8-slim-buster
```

```
LABEL maintainer="gsaggese@umd.edu"
```

```
WORKDIR /app
```

```
COPY requirements.txt requirements.txt
```

```
RUN pip3 install -r requirements.txt
```

```
COPY . .
```

```
CMD ["python3", "-m", "flask", "run", "--host=0.0.0.0"]
```

Docker: Commands

- Show all the available images

```
> docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
counter_app-web-fe	latest	4bf6439418a1	17 minutes ago	54.7MB
...				

- Show a particular image

```
> docker images counter_app-web-fe
```

counter_app-web-fe	latest	4bf6439418a1	17 minutes ago	54.7MB
...				

- Delete an image

```
> docker rmi ...
```

- Show the running containers

```
> docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
505541bcfe8b	counter_app-web-fe	"python app.py"	7 minutes ago	Up 7 minutes	0.0.0.0:5001->5000/tcp	counter_app
c1889540cfd2	redis:alpine	"docker-entrypoint.sh"	7 minutes ago	Up 7 minutes	6379/tcp	counter_app

Docker: Commands

- Show running containers

```
> docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
281d654f6b8d	counter_app-web-fe	"python app.py"	5 minutes ago	Up 5 minutes
0.0.0.0:5001->5000/tcp	counter_app-web-fe-1			
de55ae4104da	redis:alpine	"docker-entrypoint.s..."	5 minutes ago	Up 5 minutes
6379/tcp	counter_app-redis-1			

- Show volumes and networks

```
> docker volume ls
```

DRIVER	VOLUME NAME
local	counter_app_counter-vol

```
> docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
b4c1976d7c27	bridge	bridge	local
33ff702253b3	counter-app_counter-net	bridge	local

Docker: Delete State

- Commands:

```
> docker container ls
> docker container rm $(docker container ls -q)

> docker images
> docker rmi $(docker images -q)

> docker volume ls
> docker volume rm $(docker volume ls -q)

> docker network ls
> docker network rm $(docker network ls -q)
```

Docker Tutorial

- [Docker tutorial](#)

Docker Compose

- **Manage multi-container apps running on a single node**
 - Describe app in a single *declarative* YAML file
 - Avoid long Docker command scripts
 - Compose interacts with Docker API to achieve the requested state
 - Examples:
 - Client app and Postgres DB
 - Microservices: Web front-end, Ordering, Back-end DB
 - In 2020, Docker Compose became an open standard for “code-to-cloud” process
- **Manage multi-container apps running on multiple hosts**
 - Docker Stacks / Swarm
 - Kubernetes

Docker Compose: Tutorial Example

- Default name for a Compose file is `docker-compose.yml`
 - Specify `-f` for custom filenames
- **Top-level keys:**
 - `version:`
 - Mandatory first line for API version
 - Use latest version, typically 3 or higher
 - `services:`
 - Define microservices
 - `networks:`
 - Create new networks
 - Default bridge network connects containers on the same Docker host
 - `volumes:`
 - Create new volumes
- **Key in services** describes a “service” in terms of container
 - **Inner keys** specify Docker run command params

```
version: "3.8"
services:
  web-fe:
    build: .
    command: python app.py
    ports:
      - target: 5000
        published: 5001
    networks:
      - counter-net
    volumes:
      - type: volume
        source: counter-vol
        target: /code
  redis:
    image: "redis:alpine"
    networks:
      counter-net:
networks:
  counter-net:
volumes:
  counter-vol:
```

Docker Compose: Commands

```
> docker compose --help
```

```
Usage: docker compose [OPTIONS] COMMAND
```

```
Options:
```

--env-file string	Specify an alternate environment file.
-f, --file stringArray	Compose configuration files
-p, --project-name string	Project name

```
Commands:
```

build	Build or rebuild services
convert	Converts the compose file to platform's canonical format
cp	Copy files/folders between a service container and the local filesystem
create	Creates containers for a service.
down	Stop and remove containers, networks
events	Receive real time events from containers.
exec	Execute a command in a running container.
images	List images used by the created containers
kill	Force stop service containers.
logs	View output from containers
ls	List running compose projects
pause	Pause services
port	Print the public port for a port binding.
ps	List containers
pull	Pull service images
push	Push service images
restart	Restart containers
rm	Removes stopped service containers
run	Run a one-off command on a service.
start	Start services
stop	Stop services
top	Display the running processes
unpause	Unpause services
up	Create and start containers
version	Show the Docker Compose version information



Docker Compose: Commands

- Build the containers for the services
`> docker compose build`
- Pull the needed images for the services
`> docker compose pull`
- Show the running services
`> docker compose ps`
- Show the status of the service
`> docker compose ls`
- Bring up the entire service
`> docker compose up`
- Rebuild after trying out some changes in dockerfile/compose file
`> docker-compose up --build --force-recreate`

Docker Compose: Commands

- Show the processes inside each container

```
> docker compose top
```

```
counter_app-redis-1
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
999	49590	49549	0	10:40	?	00:00:02	redis-server *:6379

```
counter_app-web-fe-1
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	49614	49574	0	10:40	?	00:00:00	python app.py
root	49734	49614	1	10:40	?	00:00:08	/usr/local/bin/python /

Docker Compose: Commands

- Build the containers for the services

```
> docker compose down
[+] Running 3/2
Container counter_app-redis-1    Removed
Container counter_app-web-fe-1   Removed
Network counter_app_counter-net  Removed
```

- Shutdown service removing the volume (i.e., resetting state)

```
> docker-compose down -v
```

- Shutdown service removing images and volume

```
> docker-compose down -v --rmi all
```

Docker Compose: Tutorial

- Example taken from <https://github.com/nigelpoulton/counter-app>
- [tutorial_docker_compose](#)

```
> cd tutorials/tutorial_docker_compose  
> vi tutorial_docker_compose.md
```