# Container in 4 Hours

## Introduction

# Your Instructor

- Sander van Vugt
- mail@sandervanvugt.com

# Agenda

- Understanding Containers
- Using a Container Platform
- Managing Docker and Podman Containers
- Performing Daily Container Management
- Managing Container Images
- Managing Container Storage
- Managing Container Networking

# Poll Question 1

Which of the following topics are most interesting for you? (Choose 3)

- Understanding Containers
- Exploring the Container Landscape
- Managing Containers
- Using Containers on RHEL 8
- Performing Daily Container Management
- Managing Container Images
- Managing Container Storage
- Managing Container Networking

# Poll Question 2

Rate your knowledge/experience about Containers

- none
- minimal
- just attended a basic course
- some working experience
- good working experience
- lots of working experience
- expert
- guru

# Poll Question 3

What is your main container operating system platform?

- Windows
- MacOS
- Red Hat / CentOS / Oracle / Fedora Linux
- Ubuntu / Mint Linux
- Other Linux
- Cloud

# Poll Question 4

Which part of the world are you from?

- Europe
- Netherlands
- Africa
- North/Central America
- South America
- India
- Asia
- Australia/Pacific

# Containers in 4 hours

1. Understanding Containers

# Containers Defined

- A container is a "box" that contains an application with all of its dependencies included
- Containers always start an application, they don't sit down and wait for you to tell them what to do
- To run a container, a container runtime is needed. This is the layer between the host operating system and the container itself
- Different solutions exist to run containers
  - Docker: still the most commonly used solution
  - LXC: Linux native containers
  - Podman: default solution in RHEL 8
- Most solutions are OCI compliant

# Understanding Container History

- Containers started as chroot directories, and have been around for a long time
- Docker kickstarted the adoption of containers in 2013/2014
- Docker originally was based on LXC, and became a huge success because it added the Docker Registry

# Understanding Container Use Types

- System containers are used as the foundation to build your own application containers. They are not a replacement for a virtual machine
- Application containers are used to start just one application. Application containers are the standard
- To run multiple connected containers, you need to create a microservice. Use docker-compose or Kubernetes Pods to do this in an efficient way

# Containers and Linux

- Containers heavily rely upon features provided by the Linux kernel
  - namespaces and chroot
  - Cgroups
- The Linux features are available on other platforms as well
  - Docker Desktop provides an easy way to run containers on Windows or Mac
  - DockerMicrosoftProvider makes it possible to run containers on top of Windows servers

# Containers and Images

- A container is a running instance of an image
- The image contains the application code, language runtime and libraries
- External libraries such as libc are typically provided by the host operating system, but in a container is included in the images
- The container image is a read-only instance of the application that just needs to be started
- While starting a container, it adds a writable layer on the top to store any changes that are made while working with the container

# Understanding Container Images

- Container images are typically shared through public registries, or by sharing mechanisms to build them easily, such as Dockerfile
- The Docker container image format has become the de facto standard image format
- Open Container Initiative (OCI) has standardized the Docker container image format

# Understanding Registries

- To work with containers, you'll need to take care of distribution of images
- Manual distribution using images in tar balls is possible, but NOT recommended
- Use registries instead
  - Storing in remote registries is common, and the DockerHub registry is very common (https://hub.docker.com)
  - When using Red Hat, have a look at quay.io
  - As an alternative, consider storing in local (private) registries

# Understanding Container Runtimes

- The container runtime is a specific part of the container engine, and two of them are commonly used:
  - CRI-O: Red Hat
  - containerd: originates from Docker
- The container runtime is taking care of specific tasks
  - Provide the mount point
  - Communicate with the kernel
  - Set up cgroups and more

# Understanding runC

- runC is a lightweight universal container runtime
- It is the default runtime defined by the Open Containers Initiative
- It is the specific part that focusses on creating containers
- As such, it's included in CRI-O as well as containerd
- Where CRI-O and containerd are adding features, like container lifecycle management and container image management

# Understanding Container Orchestration

- To build microservices using containers, additional datacenter features are needed
  - Flexible and scalable networking
  - Scalable and flexible storage
  - Methods to connect containers together
  - Additional services for cluster-wide monitoring of service availability
- To provide for all of these, an orchestration platform is needed
- Kubernetes is the standard orchestration platform
- Almost all other orchestration platforms are based on Kubernetes
  - OpenShift
  - Rancher

2. Using a Container Platform

# Choosing a Container Platform

- To test container in isolation from other parts of your OS, install Docker in a Linux virtual machine
- Use Docker Desktop on top of your main computer OS to work with an integrated platform on MacOS or Windows
- Use Podman/CRI-o when using RHEL 8 and alike
- Do NOT install Docker on RHEL 8 and later as it's not supported

# Understanding Docker Desktop

- To take a Docker test drive, you need a working environment
- Docker Desktop provides an easy way to get started and take a Docker test drive
- Just download the Docker Desktop application on your current host OS, install it and run it
- Next, create a Docker Id and log in
- Now use **docker run hello-world** to run your first Docker application
- See here for specific instructions:
  - Windows: https://docs.docker.com/docker-for-windows/install/
  - MacOS: https://docs.docker.com/docker-for-mac/install/

# Understanding Docker Desktop

- Docker Desktop uses host OS virtualization features to run a Docker virtual machine, which you don't have to manage as a virtual machine
- To interface with it, just access the Docker Desktop application to manage everything you need
- And use the **docker** CLI to run Docker commands from the command line
- This is different from running other solutions like Docker Toolbox, which will give you a full Linux based Docker VM (called Docker Machine), the **docker** CLI and a VirtualBox virtualization platform
- The most significant limitation to Docker Desktop is that it cannot route traffic to containers, so you cannot directly access an exposed port

# Installing Docker on Ubuntu

- sudo apt-get update
- sudo apt-get install apt-transport-https ca-certificates curl gnupg-agent software-properties-common
- curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
- sudo apt-key fingerprint 0EBFCD88
- sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
- sudo apt-get update
- sudo apt-get install docker-ce docker-ce-cli containerd.io
- sudo docker run hello-world

# Understanding RHEL 8 Containers

- RHEL 8 does no longer support Docker
- RHEL 8 provides tools to work with containers to run directly on top of RHEL 8 in single-node use cases
- Use **sudo dnf module install container-tools** to install all you need
- **sudo dnf install podman-docker** provides docker like syntax in the podman command
- That means that NO docker or any other container engine is required! RHEL 8 containers run daemon-less
- For running containers in enterprise environments, OpenShift is required

# RHEL 8 Container Tools Overview

- **podman**: direct management of pods and container images
- **buildah**: for building, pushing, and signing container images
- **skopeo**: for copying, inspecting, and signing images
- These tools are compatible with the Open Container Initiative and therefore can be used to manage Linux containers on top of any OCI-compatible container engine

# Understanding Rootless Containers

- A rootless container runs with limited user privileges
- A root container runs with root privileges
- Rootless containers do not get an IP address
- Rootless containers cannot bind to a privileged port
- Rootless containers have limited access to the filesystem
- Rootless containers are the default in OpenShift

# Containers in 4 Hours

3. Managing Containers

# Before Getting Started with Docker

- Users have to be a member of the **docker** system group in order to communicate with the Docker daemon and start and manage containers
  - Use **sudo usermod –aG docker $(USERID)**
- Do NOT run containers as root!
- When using **podman** on Red Hat, rootless containers can be used

# Finding Images

- To run Docker containers, registry access must be available
- By default, images are fetched from docker hub
- On RHEL, registries are specified in /etc/containers/registries.conf
- Use **docker search** to find the image you need
- Or use the web interface available at https://hub.docker.com

# Running Containers

- Remember: containers are fancy applications
  - **docker run centos** will run the centos:latest image, start its default application and immediately exit
  - **docker run -it centos bash** will run the centos:latest image, start bash, and open an interactive terminal
- Managing foreground and background state
  - **docker run -it centos bash** will run the container in the foreground
    - Use Ctrl-p, Ctrl-q to disconnect
    - Use **exit** to quit the main application
  - **docker run -d nginx** will run the container in the background
  - **docker attach container-name** will attach to the running container if it was started with **-d**
- Notice that since docker version 1.13, **docker container run** instead of **docker run** is the preferred command

# Verifying Container Availability

- **docker ps** gives an overview of containers currently running
- **docker ps -a** also gives an overview of containers that have been running successfully

# Stopping Containers

- A container stops when its primary application stops
- Use **docker stop** to send SIGTERM to a container
- Use **docker kill** to send SIGKILL to a container
- After stopping a container, it does not disappear
- Use **docker rm** to permanently remove it

# Getting More Details

- Use **docker inspect** to get details about containers
- Use **docker logs** to get access to the primary application STDOUT
- Use **docker stats** for a Linux **top**-like interface about real-time container statistics

# Containers in 4 Hours

4. Performing Daily Container Management

# Publishing Ports

- By default, container applications are accessible from inside the container only
- To make it accessible, you'll need to publish a port
- **docker container run --name web_server -d -p 8080:80 nginx** runs the nginx image, and configures port 8080 on the docker host to port forward to port 80 in the container
- You cannot publish a port on a container that is already running

# Understanding Memory Limitations

- As containers are just Linux processes, by default they'll have full access to the host system resources
- The Linux kernel provides cgroups to put a limit to this
- **docker run -d -p 8081:80 --memory="128m" nginx** sets a hard memory limit
- **docker stats**

# Understanding CPU Limitations

- Docker inherits the Linux kernel Cgroups notion of CPU Shares
- If not specified, all containers get a CPU shares weight of 1024
- When starting a container, a relative weight expressed in CPU shares can be specified
    - **docker run -d --rm -c 512 --cpus 4 busybox dd if=/dev/zero of=/dev/null** will run the container on 4 CPUs, but with relative CPU shares set to 50% of available CPU resources
- Containers can also be pinned to a specific cpu using **--cpuset-cpus**
    - **docker run -d --rm --cpuset-cpus=0,2 --cpus 2 busybox dd=/dev/zero of=/dev/null** will run the container on cores 0 and 2 only
- To test: use different CPU shares on two containers and force them to run on the same CPU

# Containers in 4 Hours

5. Managing Container Images

# Understanding Container Images

- Images are what a container is started from
- Base container images are available at hub.docker.com
- Users can upload images to Docker Hub
- Go to hub.docker.com to search for images
- Or use **docker search** to search for images
- Using the web page is convenient, you can see much information including the Docker file used to build this image

# Creating Images

- Roughly, there are three approaches to creating an image
- Using a running container: a container is started and modifications are applied to the container. From the container, **docker** commands are used to write modifications
- Using a Dockerfile: a Dockerfile contains instructions for building an image. Each instruction adds a new layer to the image, which offers more control which files are added to which layer
- Use **buildah** on RHEL to create an image by executing commands within the image

# Understanding Dockerfile

- Dockerfile is a way to automate container builds
- Containerfile on RHEL is the same
- It contains all instructions required to build a container image
- So instead of distributing images, you could just distribute the Dockerfile
- Use **docker build .** to build the container image based on the Dockerfile in the current directory
- Images will be stored on your local system, but you can direct the image to be stored in a repository

# Demo: Using a Dockerfile

- Dockerfile demo is in https://github.com/sandervanvugt/containers/dockerfile
- Use **docker build -t nmap .** to run it from the current directory
- Tip: use **docker build --no-cache -t nmap .** to ensure the complete procedure is performed again
- Next, use **docker run nmap** to run it
- For troubleshooting: **docker run -it nmap /bin/bash**
  - WILL NOT WORK

# Understanding ENTRYPOINT

- ENTRYPOINT is the default command to be processed
- If not specified, **/bin/sh -c** is executed as the default command
- Arguments to the ENTRYPOINT command should be specified separately using CMD
  - ENTRYPOINT ["command"]
  - CMD ["arg1","arg2"]
- If the default command is specified using CMD instead of ENTRYPOINT, the command is executed as an argument to the default entrypoint **sh -c** which can give unexpected results
- If the arguments to the command are specified within the ENTRYPOINT, then they cannot be overwritten from the command line

# Demo: Using **docker commit**

- **docker run -d --name newnginx nginx**
- **docker ps**
- **docker exec -it newnginx sh**
- **echo hello >> /tmp/testfile**
- **Ctrl-p, Ctrl-q**
- **docker diff newnginx**
- **docker commit newnginx**

# Using **docker save**

- After making changes to a container, you can save it to an image
- Use **docker commit** to do so
  - **docker commit -m "custom web server" -a "Sander van Vugt" myapache myapache**
  - Use **docker images** to verify
- Next, use **docker save -o myapache.tar myapache** and transport it to anywhere you'd like
- From another system, use **docker load -i myapache.tar** to import it as an image
- Next, use **docker run myapache** to run it on that system

# Understanding Tags

- Tags are used to specify information about a specific image version
- Tags are aliases to the image ID and will show when using **docker images**
- Tags are typically set when building the image:
  - **docker build -t username/imagename:tagname**
  - For private use, the **username** part is optional, when pushing it to a public registry it is mandatory
- Alternatively, tags can be set using **docker tag**
  - **docker tag source-image{:tag] targetimage[:tag]**
- If no tag is applied, the tag **:latest** is automatically set
  - **:latest** always points to the latest version of an image
- Target image repositories can also be specified in the Docker tag

# Tagging Images

- Tags allow you to assign multiple names to images
  - A common tag is "latest", which allows you to run the latest
- In Centos, all images are in one repository, and tags are used to specify which specific image you want: centos:7.6
- Manually tag images: **docker tag myapache  myapache:1.0**
  - Next, using **docker images | grep myapache** will show the same image listed twice, as 1.0 and as latest
- Tags can also be used to identify the target registry
  - **docker tag myapache localhost:5000/myapache:1.0**

# Using Meaningful Tags

- If no tags are specified, the tag **:latest** is automatically added
- Consider using meaningful tags, including version number as well as intended use (like **:prod** and **:test**)

# Using a Private Registry

- **docker run -d -p 5000:5000 --restart=always --name registry registry:latest**
- **sudo ufw allow 5000/tcp**
- **docker pull fedora**
- **docker images**
- **docker tag fedora:latest localhost:5000/myfedora** (the tag is required to push it to your own image registry)
- **docker push localhost:5000/myfedora**
- **docker rmi fedora**; also remove the image based on the tag you've just created
- **docker exec -it registry sh; find . -name "myfedora"**
- **docker pull localhost:5000/myfedora** downloads it again from your own local registry

# Containers in 4 Hours

6. Managing Container Storage

# Understanding Container Storage

- Container storage by nature is ephemeral, which means that it lasts for a very short time and nothing is done to guarantee its persistency
- When files are written, they are written to a writable filesystem layer that is added to the container image
- Notice that as a result, storage is tightly connected to the host machine
- To work with storage in containers in a more persistent and flexible way, permanent storage solutions must be used
- NFS is a common persistent storage type in non orchestrated environments
- Advanced persistent storage solutions only exist in orchestration solutions

# Understanding Storage Solutions

- One solution is to use a bind mount to a file system on the host OS: the storage is managed by the host OS in this case
- Another solution is to connect to external (SAN or cloud-based) persistent storage solutions
- To disconnect storage from the containers using it, Volumes are used, and specific drivers can specify which volume type to connect to

# Understanding Bind Mounts

- Bind Mount storage is based on Linux bind mounts
- The container mounts a directory or file from the host OS into the container
- If the host directory doesn't yet exist, it will be created, but only if the **-v** option is used
- The host OS still fully controls access to the file
- Docker commands cannot be used to manage the bind mount
- The **-v** option as well as the **--mount** option can be used to create the bind mount
  - **-v** is the old option, which combined multiple arguments in one field
  - **--mount** is newer and more verbose

# Cases for Using Bind Mounts

Bind mounts work when the host computer contains the files that need to be accessible in the containers

- Configuration files
- Access to source code
- Log files

# Creating a Bind Mount

- Using **--mount**
  - **mkdir bind1; docker run --rm -d --name=bind1 --mount type=bind,source="$(pwd)"/bind1,target=/app nginx:latest**
- Using **-v**
  - **docker run --rm -dit --name=bind2 -v "$(pwd)"/bind2:/app nginx:latest**
- Use **docker inspect <containername>** to verify
- Use **docker exec -it bind1 sh** to open a shell in the container and check

# Benefits of using Volumes

Volumes are the preferred way to work with persistent data as the volume survives the container lifetime

- Multiple containers can get simultaneous access to the volumes
- Data can be stored externally
- Volumes can be used to transition data from one host to another
- Volumes are supported for Linux as well as Windows containers
- Volumes live outside of the container and for that reason don't increase container size
- Volumes use drivers to specify how storage is accessed. Enterprise-grade drivers are available in Docker Swarm - not stand alone

# Demo: Working with Volumes

- **docker volume create myvol** creates a simple volume that uses the local file system as the storage backend
- **docker volume ls** will show the volume
- **docker volume inspect myvol** shows the properties of the volume
- **docker run -it --name voltest --rm --mount source=myvol,target=/data nginx:latest /bin/sh** will run a container and attach to the running volume
- From the container, use **cp /etc/hosts /data; touch /data/testfile; ctrl-p, ctrl-q**
- **sudo ls /var/lib/docker/volumes/myvol/_data/**
- **docker run -it --name voltest2 --rm --mount source=myvol,target=/data nginx:latest /bin/sh**
- From the second container: **ls /data; touch /data/newfile; ctrl-p, ctrl-q**

# Understanding Multi-container Volume Access

- To just create a file in a volume, nothing special is needed and the volume can be accessed from multiple containers at the same time
- To simultaneously access files on volumes from multiple containers, a special driver is needed
- Recommended: use the **readonly** mount option to protect from file locking problems
  - **docker run -it --name voltest3 --rm --mount source=myvol,target=/data,readonly nginx:latest /bin/sh**
- Enterprise-grade drivers are available in Docker Swarm, or are provided through Kubernetes
- For non-orchestrator use, consider using the local driver NFS type

# Demo: Configuring an NFS-based Volume - 1

- sudo apt install nfs-server nfs-common
- sudo mkdir /nfsdata
- sudo vim /etc/exports
  - /nfsdata *(rw,no_root_squash)
- sudo chown nobody:nogroup /nfsdata
- sudo systemctl restart nfs-kernel-server
- showmount -e localhost
- docker volume create --driver local --opt type=nfs --opt o=addr=192.168.4.163,rw --opt device=:/nfsdata nfsvol
- docker volume ls
- docker volume inspect nfsvol

# 7. Managing Container Networking

# Understanding Docker Networking

- Container networking is pluggable and uses drivers
- Default drivers provide core networking
  - bridge: the default networking, allowing applications in standalone containers to communicate
  - host: removes network isolation between host and containers and allows containers to use the host network directly. In Docker, only available in swarm
  - overlay: in Swarm, allows different Docker daemons to be connected using a software defined network. Allows standalone containers on different Docker hosts to communicate
  - macvlan: assigns a MAC address to a container, making it appear as a physical device on the network. Excellent for legacy applications
  - none: completely disables networking
  - plugins: uses third-party plugins, typically seen in orchestration software

# Understanding Bridge Networking

- Bridge networking is the default: a container network is created on internal IP address 172.17.0.0/16
- Containers will get an IP address in that range when started
- Additional bridge networks can be created
- When creating additional bridge networks, automatic service discovery is added, so that new containers can be reached by name
- There is no traffic between different bridge networks because of namespaces that provide strict isolation
- You cannot create routes from one bridge network to another bridge network, and that is by design

# Creating a Custom Bridge - 1

- Create a custom network
  - **docker network create –driver bridge alpine-net**
  - **docker network ls**
  - **docker network inspect alpine-net**
- Start containers on a specific network. Notice that while starting, a container can be connected to one network only. If it needs to be on two networks, you'll have to do that later
  - **docker run -dit --name alpine1 --network alpine-net alpine ash**
  - **docker run -dit --name alpine2 --network alpine-net alpine ash**
  - **docker run -dit --name alpine3 alpine ash**
  - **docker run -dit --name alpine4 --network alpine-net alpine ash**
  - **docker network connect bridge alpine4**

# Creating a Custom Bridge - 2

- Verify correct working
  - **docker container ls**
  - **docker network inspect bridge**
  - **docker network inspect alpine-net**
- Verify automatic service discovery, which is enabled on user defined networks
  - **docker container attach alpine1; ping alpine4**
- But notice this doesn't work on the default bridge
  - (still from alpine1) **ping alpine3**
- There's no routing either:
  - (still from alpine1) ping 172.17.0.2
- But all containers can reach out to the external network

# Containers in 4 Hours

## Further Learning

# Live Courses

- Building Microservices with Containers
- Kubernetes in 4 Hours
- Getting Started with OpenShift
- Certified Kubernetes Application Developer (CKAD) Crash Course
- Certified Kubernetes Administrator (CKA) Crash Course
- EX180 Crash Course
- Container Based Devops in 4 Weeks

# Recorded Courses

- Hands-on Kubernetes
- Getting Started with Kubernetes LiveLessons 2nd Edition
- Red Hat OpenShift Fundamentals 3/ed
- Certified Kubernetes Application Developer (CKAD)
- Certified Kubernetes Administrator (CKA)
- Modern Container-based DevOps