Managing Containers

Introduction

Your Instructor

- Sander van Vugt
- mail@sandervanvugt.com

Agenda

- 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 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 job title applies to you best?

- sysadmin
- devops
- developer
- security engineer
- DBA
- network engineer
- management
- other (use group chat to specify)

Poll Question 5

Which part of the world are you from?

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

Managing Containers

1. Understanding Containers

Containers Defined

- A container is 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: owns more than 80% of the market and is the de facto standard
  - LXC: Linux native containers
  - Podman: default solution in RHEL 8

Understanding Container Types

- System containers behave like virtual machines. They go through a complete boot process and provide VM-typical services like SSH and logging
- Application containers are used to start just one application. Application containers have become the default
- Orchestration solutions like Kubernetes focus on managing application containers and may become confused while working with system images as they don't have a default application to start

Containers and Linux

- Containers heavily rely upon features provided by the Linux kernel
  - namespaces and chroot
  - Cgroups
- The Linux kernel can be used as the container runtime in a pure Linux-based container environment
  - LXC is the original Linux-based container solution
  - systemd-nspawn offers containers based on systemd
- 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

- There is no such thing as THE container image
- A container image consists of multiple layers that are connected together
- 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 regristries instead
  - Storing in remote registries is common, and the DockerHub registry is the standard (https://hub.docker.com)
  - As an alternative, consider storing in local (private) registries

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

Managing Containers

2. Exploring the Container Landscape

Understanding Container History

- Container is all about running isolated processes in a protected environment
- It started in the 1970's when the chroot system call was introduced
- In 2000 FreeBSD jails were introduced to partition file systems, network addresses and memory
- In, Solaris containers was released in 2004, to combine system resource control and zone-based boundary separation
- In 2007, Control Groups were introduced in the Linux kernel
- In 2008, LXC combined cgroups and Linux namespaces to implement Linux-based containers
- In 2013, Docker started and became successful by offering a complete container ecosystem
- In 2014, Kubernetes started as the default container management platform

Container Standardization - 1

- OCI is Open Containers Initiative and as such a project in LinuxFoundation
- Docker has donated its container format and runtime (runc) to OCI to serve as a solid foundation
- Established in 2015 by with the purpose of creating open industry standards around container formats and runtimes
  - The *Runtime Spec* outlines how to run an "unpacked filesystem bundle" as a container
  - The *Image Specification* outlines how container images should be created
- Because of OCI, different components of the container landscape are highly inter-operable
- Current members include Docker, Microsoft, Red Hat, VMware and many more

Container Standardization - 2

- Cloud Native Computing Foundation (CNCF) is a part of LinuxFoundation promotes the adoption of cloud-native computing
  - Cloud Native Computing is an approach in software development that utlizes cloud computing to build and run scalable applications in mdern, dynamic environments such as public, private and hybrid clouds
  - Technologies such as containers, microservices serverless functions and immutable infrastructure, deployed via declarative code are parts of this style
  - Normally, cloud-native applications are built as a set of microservices that run in containers and may be orchestrated in Kubernetes, and deployed using DevOps and Git
- CNCF hosts the Kubernetes Open Source project

What is a Container Runtime

- The container runtime is the software that executes containers and manages container images
- Docker is the most common container runtime
- Other container runtimes exist, such as containerd, rkt and lxd
- The container runtime can be considered the high-level solution to run containers

Low level container runtimes

- OCI is the Open Containers Initiative
- OCI provides standardization for solutions that work with containers (like orchestration platforms)
- OCI has also defined container runtime specs
- **runc** is the default implementation of OCI runtime specs, **crun** is an implementation of **runc** that is completely written in C
- the OCI runtime provides low level functions for running containers
- The high-level runtime (docker etc.) provides functions that run on top of the low-level runtime

CRI Implementations

- The CRI is an abstraction interface that defines a runtime that runs on top of runc
- Currently, the following CRI implementations exist
  - Dockershim
  - CRI-O
  - Containerd
  - Frakti
  - rktlet
- Functionality in the higher level Docker container runtime is being replaced by lower level container runtimes like CRI-O and containerd

Alternatives for Docker

- CoreOS rkt / Podman: now the standard in RHEL 8
- Mesos Containers: developed by Apache, focusing on performance and often seen in big data applications. Doesn't work without the Mesos framework
- LXC: "Chroot on steroids", the original inspiration for the start of Docker
- LXD: A layer on top of LXC to make LXC easier and more secure, mainly used on Ubuntu
- OpenVZ: founded in 2005, with the main orientation on running system containers
- Containerd: Proposed by CNCF as a standard daemon for running OCI images on any platform

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

Managing Containers

3. Managing Containers

Installing Docker on CentOS 7

- Installing Docker on CentOS 8 is possible, but not supported and known to be problematic. Install on CentOS 7 instead
  - You can do it, but risk is significant that it breaks after updating RHEL 8 software
- On CentOS 7, Docker can be installed from the Docker repositories, or from the CentOS repositories. This procedure shows how to install from the Docker repositories

Installing Docker on CentOS 7

- **sudo yum install –y yum-utils**
- **sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo**
- **sudo yum install docker-ce docker-ce-cli containerd.io**
- **sudo systemctl start docker**
- **sudo docker run hello-world**

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

Installing Docker on Windows and Mac

- There is no native installation of Docker on Windows or Mac
- To use Docker on Windows or Mac, use Docker Desktop as discussed before

Before Getting Started

- 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!

Running Docker Containers

- To run Docker containers, registry access must be available
- By default, images are fetched from docker hub
- Use **docker search** to find the image you need
- Or use the web interface available at https://hub.docker.com
- After finding the image you need, use **docker run** to start it
  - Notice that when starting a system image, parameters must be provided to have it run a default command as well: **docker run –it busybox /bin/sh**
  - When starting an application image, this is not required: **docker run –d nginx**
- Use **exit** or **Ctrl-p,Ctrl-q** to quit/disconnect
- Use **docker ps [--all]** for an overview of currently running containers

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

Verifying Container Availability

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

Getting More Details

- Use **docker inspect** to get details about running 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

Investigating Containers on the Host OS

- On the host OS, a Docker container is just a running process that is managed by one of the Docker components
- Because of Linux kernel namespaces, one container does not have access to running parts of the other containers
- When using **ps aux**, all the running Docker containers show as Linux processes

Managing Containers

4. Using Containers on RHEL 8

Containers in RHEL 8

- Managing Containers is a vital part of RHEL 8
- RHEL itself has core technologies for running containers
  - Cgroups: resource management
  - Namespaces: process isolation
  - SELinux: security
- OpenShift is a dedicated RHEL product for managing and running containers in an enterprise environment

Understanding RHEL 8 Containers

- RHEL 8 does no longer support Docker
- The Docker Community Edition can be used on RHEL 8, but this is not recommended
- RHEL 8 provides tools to work with containers to run directly on top of RHEL 8 in single-node use cases
- That means that NO docker or any other container engine is required! RHEL 8 containers run daemon-less
- RHEL 8 Container functionality is based on project Atomic
- For running containers in enterprise environments, OpenShift is required

Accessing Registries

- Container images must be fetched from a registry
- /etc/containers/registries.conf is used to tell RHEL where to go
- By default all is setup to start working!
- Add your own insecure registries in the [registries.insecure] section
- Note that registry.redhat.io requires authentication, registry.access.redhat.com doesn't need any authentication but is deprecated
  - Registries are processed in order, so use the most successful registry first in your list
  - Include the registry in the pull command to ensure where the image is fetched from: **podman pull registry.redhat.io/rhel8/rhel**
- Use **podman login registry.redhat.io** to authenticate

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
- **runc**: the container engine that provides container run and build features to podman and buildah
- 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
- Install using **yum module install container-tools**
- Optional: **yum install podman-docker** provides docker like syntax in the podman command

Working with **podman** - 1

- **podman search rhel8** will look up the RHEL8 image
- **podman pull rhel8/rhel** will download it from the first registry that answers
  - It's not required to pull an image before you can run a container, running the container automatically pulls the image as well
- **podman images** will show all images stored on the local system
- **podman inspect <imagename>** will inspect image contents: it will show you what exactly will happen when you run the container (look for Cmd

Working with **podman** - 2

- **podman run <image-ID>** will run it and execute the default command
  - By default, you will attach to the container and enter the current container shell
  - Use **podman run -d <image-ID>** to detach
- **podman ps** gives a list of currently running containers
- **podman mount <id>** allows you to mount a container on a specific location so that you can further inspect it
- Use the mount point to request an RPM package list:
  - **rpm -qa --root=/var/lib/containers/.../merged**
- **podman rmi** is used to remove images

Demo: Running Containers with **podman**

- **podman run --rm ubuntu:latest cat /etc/os-release** will start the container, run the command and exit, and delete the container afterwards (--rm)
- **podman run --rm --name=mycontainer –it ubuntu:latest /bin/bash** will start the container, run the bash shell, and keep it open
  - **-i** is interactive
  - **-t** opens a terminal
- **podman run --name=mycontainer -v /dev/log:/dev/log --rm ubuntu:latest logger hello** will start the container and map the local /dev/log device to the container /dev/log device. Next, logs a message and exits

Inspecting Containers

- **podman ps** shows all containers actually running
- **podman inspect <id>** shows properties on a container based on its id
  - Note that some properties only apply to containers that are actually running
  - Use **--format** to request values of specific properties: **podman inspect --format='{{.NetworkSettings.IPAddress}}' <id>**
- **podman exec -it <id> /bin/bash** allows you to run a bash shell within a container
  - Note that the command you want to run needs to exist within the container

Understanding **buildah**

- Buildah is a tool that helps in building images
- It is NOT the preferred tool for actually running containers
- Doesn't need a container runtime daemon but uses runx
- Different options to build an image
  - By mounting the root directory of a container and modifying that
  - By using native **buildah** commands
  - From a Dockerfile
  - From scratch
- Using the **buildah** native commands makes it easy to script the entire build process

Using **buildah** Main Commands

- **buildah bud** (build-using-dockerfile) will build an image from a Dockerfile
- **buildah from <imagename>** builds an image from another image
- **buildah from scratch** allows you to build from scratch
- **buildah inspect** shows container or image metadata
- **buildah mount** mounts the container root FS
- **buildah commit** uses updated contents of a container root FS as a FS layer to commit content to a new image
- **buildah rmi/rm** remove image or container
- **buildah unmount** unmounts container

Creating an Image using **buildah** Commands - 1

- **buildah from fedora:latest** creates a new image based on Fedora
- **buildah images** shows that NO image was created - we still have to create it
- **buildah containers** shows that a new buildah-based container was started (fedora-working-container)
- **curl -sSL http://ftpmirror.gnu.org/hello/hello-2.10.tar.gz -o hello-2.10.tar.gz** downloads a file
- **buildah copy fedora-working-container hello-2.10.tar.gz /tmp/hello-2.10.tar.gz**
- **buildah run fedora-working-container dnf install -y tar gzip gcc make automake gettext**
- **buildah run fedora-working-container dnf clean all**

Creating an Image Using **buildah** Commands - 2

- **buildah run fedora-working-container tar xzvf /tmp/hello-2.10.tar.gz -C /opt**
- **buildah config --workingdir /opt/hello-2.10 fedora-working-container**
- **buildah run fedora-working-container ./configure**
- **buildah run fedora-working-container autoreconf**
- **buildah run fedora-working-container make**
- **buildah run fedora-working-container cp hello /usr/local/bin/**
- **buildah run fedora-working-container hello -v**
- **buildah config --entrypoint /usr/local/bin/hello fedora-working-container**
- **buildah commit --format docker fedora-working-container hello:latest**
- **buildah rm fedora-working-container**

Managing Containers

5. Performing Daily Container Management

Searching for Images

- Before running a container, you need to search the right image
  - **docker search ubuntu** will show lots of images
  - **docker search --filter "is-official=true" ubuntu** will only show official images
  - **docker search --filter=stars=30 --filter=is-automated=true centos** will also include appreciation
- Based on what you've found, you can either **pull** the image, or directly **run** the container
  - **docker pull centos:centos6**
  - **docker pull --all-tags centos** (will download a LOT)
  - **docker run --rm centos:latest** will pull and run the latest centos version (and stop immediately) after which the container is removed
  - **docker images** will show all images locally available

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 -dit centos bash** 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

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

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 run -d -p 8082:80 --memory-reservation="256m" nginx** sets a soft limit, which will only be enforced if a memory shortage exists

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 -it --rm -c 512 mycontainer --cpu 4** 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**
  - **docker run -it --rm --cpuset=0,2 mycontainer --cpu 2** 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

Managing Containers

6. Managing Container Images

Understanding Container Images

- Images are what a container is started from
- Base container images are available at hub.docker.io
- 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

Understanding Container Image Layers

- Docker images are made up of a series of filesystem layers
- Each layer adds, removes or modifies files from the preceding layer in the filesystem
- This is called an overlay filesystem, and different overlay filesystems exist, like aufs, overlay and overlay2
- By using these different image layers, and pointing to other image layers in a smart way, it's easy to build container images that have support for multiple versions of vital components
- Apart from the different layers, container images typically have a container configuration file that provides instructions on how to run the container

Exploring Image Layers

- **docker image ls** shows images stored
- **docker history <imageID>** or **docker history image:tag** shows the different layers in the image
- Notice that each image adds an image layer!

Creating Images

- Roughly, there are two 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

Understanding Dockerfile

- Dockerfile is a way to automate container builds
- 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/sandertest
- 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 only work if you've installed bash!

Managing Images with **docker commit**

- 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**)

Creating Private Registries

- On CentOS
  - **yum install docker-distribution**
  - **systemctl enable --now docker-distribution**
  - Config is in /etc/docker-distribution/registry/config.yml
  - Registry service listens on port 5000, open it in the firewall
    - **sudo firewall-cmd --add-port 5000/tcp**
- On Ubuntu
  - **docker run -d -p 5000:5000 --restart=always --name registry registry:2**

Demo: Using Your Own Registry

- **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 pull localhost:5000/myfedora** downloads it again from your own local registry

Managing Containers

7. 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
- 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 int his case
- Another solution is to connect to external (SAN or cloud-based) persistent storage solutions
- To connect to external storage, Volumes are used, and specific drivers can specify which volume type to connect to
- For temporary data, tmpfs can be used

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 -dit --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

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

Using Volumes Procedure Overview

- Recommended but not needed: create the volume before using it in a container
  - Note that **docker run -d --name=voltest3 --rm --mount source=nginxvol,destination=/usr/share/nginx/html nginx:latest** will work also
  - Create volumes before using them to make it easier to manage

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 -I; 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
- sdo 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

Managing Containers

8. 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

Connecting Containers on the Bridge Network - 1

- Type **docker network ls** to see default networking
- Start two containers on the default network
  - **docker run -dit --name alpine1 alpine ash**
  - **docker run -dit --name alpine2 alpine ash**
- Verify the containers are started
  - **docker container ls**
- Check what is currently happening on the Network Bridge and notice the IP addresses of the containers
  - **docker network inspect bridge**
- View the container perspective on current networking
  - **docker attach alpine1; ip addr show**

Connecting Containers on the Bridge Network - 2

- Verify connectivity (from within the container)
  - **ping 172.17.0.3**
- Use Ctrl-p, Ctrl-q to detach from the alpine container
- Stop and remove the containers
  - **docker container stop alpine1 alpine2**
  - **docker container rm alpine1 alpine2**

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

Managing Containers

Further Learning

Poll Question 6

Which of the following topics has been covered too much (choose all that apply)

- 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
- None, all was fine

Poll Question 7

Which of the following topics should get more attention (choose all that apply)

- 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
- None, all was fine

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

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)