Agenda

Poll Question

What is your experience with DevOps

- What is DevOps
- None
- Just starting
- Reasonable
- Advanced

Poll Question

- Which days are you planning to attend (select all that apply)
- Day 1: DevOps intro
- Day 2: Containers intro
- Day 3: Kubernetes intro
- Day 4: OpenShift intro

Poll Question

Which of the following topics are most interesting for you (choose all that aplly)

- Working with Git
- Understanding DevOps
- Using CI/CD
- Working with Containers
- Kubernetes basics
- Kubernetes intermediate
- OpenShift basics
- OpenShift intermediate

Poll Question

Which of the following topics do you feel already confident with? (select all that apply)

- Working with Git
- Understanding DevOps
- Using CI/CD
- Working with Containers
- Kubernetes basics
- Kubernetes intermediate
- OpenShift basics
- OpenShift intermediate

Poll Question

- Where are you from?
- India
- Asia (not India)
- USA or Canada
- Central America
- South America
- Africa
- Netherlands
- Europe
- Australie/Pacific

WARNING

- Today Feb. 1st is the first time I'm starting this course
- You may see small differences between the course agenda and the course topics list that is published for this course
- Some things will go wrong
- Your feedback is more important than ever!
- Slides in day 2-4 are still being developed and not complete yet

Course Overview

- On day 1, you'll learn about DevOps fundamentals. It has significant amount of lectures, and you'll learn how to work with GitHub and Jenkins, which are essential DevOps tools
- On day 2, we'll explore containers, the preferred way of offering access to appplications in a DevOps world
- On day 3, you'll learn how to work with Kubernetes, the perfect tool to build container based microservices and decouple site-specific information from the code you want to distribute
- On day 4, you'll learn how to work with the OpenShift Kubernetes distribution, because it has a very nice integration of CI/CD by offering its Source to Image (S2I)

Course Objectives

- In this course, you will learn about DevOps and common DevOps solutions
- You will learn how to apply these solutions in Orchestrated Containerized IT environments
- We'll zoom into the specific parts, but in the end the main goal is to bring these parts together, allowing you to make DevOps work more efficient by working with containers

Minimal Software Requirements

- Day 1: a base installation of any Linux distribution as a virtual machine. Recommended: Ubuntu LTS 20.04 Workstation
- Day 2: a base installation of any Linux distribution as a virtual machine
- Day 3: Ubuntu LTS 20.04 Workstation in a VM with at least 4 GiB RAM
- Day 4: Fedora Workstation in a VM with at least 12 GiB RAM
- At the end of each day, next day lab setup instructions are provided

Day 1 Agenda

- Understanding DevOps
- Using Git
- Using CI/CD
- Understanding Microservices
- Using Containers in Microservices
- Getting started with Ansible
- Homework assignment

Day 1 Objectives

- Learn about different DevOps concepts and tools
- Learn about MicroServices fundamentals
- Understand why Containers can be used to bring it all together
- Learn how Container based Microservices are the perfect solution for working the DevOps way

Day 2 Agenda

- Understanding Containers
- Running Containers in Docker or Podman
- Managing Container Images
- Managing Container Storage
- Accessing Container Workloads
- Preparing week 3 Setup and Homework

Day 2 Objectives

- Learn about containers
- Learn how to setup a containerized environment with Ansible

Day 3 Agenda

- Understanding Kubernetes
- Running Applications in Kubernetes
- Exposing Applications
- Configuring Application Storage
- Implementing Decoupling in Kubernetes
- Exploring week 4 setup and homework

Day 3 Objectives

- Learn about Kubernetes Fundamentals
- Learn how to implement Microservices based decoupling using common Kubernetes tools

Day 4 Agenda

- Comparing OpenShift to Kubernetes
- Running Kubernetes applications in OpenShift
- Building OpenShift applications from Git source code
- Configuring OpenShift Authentication and Authorization

Day 4 Objectives

- Learn about OpenShift Fundamentals
- Understand how OpenShift brings Microservices based decoupling together with CI/CD

How this course is different

- Topics in this course have overlap with other courses I'm teaching
  - Containers in 4 Hours
  - Kubernetes in 4 Hours
  - Ansible in 4 Hours
  - Getting Started with OpenShift
- This course is different, as its purpose is to learn how to do DevOps using the tools described in these courses
- As such, this course gives an overview of technology explained more in depth in the above mentioned courses
- Consider attending these other courses to fill in some of the details

Day 1

Day 1 Agenda

- Understanding DevOps
- Understanding Microservices
- Using Git
- Using CI/CD
- An Introduction to Jenkins
- Getting Started with Ansible
- Using Containers in Microservices
- Homework assignment

Understanding DevOps

Understanding DevOps

- In DevOps, Developers and Operators work together on implementing new software and updates to software in the most efficient way
- The purpose of DevOps is to reduce the time between committing a change to a system and the change being placed in production
- DevOps is Microservices-oriented by nature, as multiple smaller project are easier to manage than one monolithic project
- In DevOps, CI/CD pipelines are commonly implemented, using anything from simple GitHub repositories, up to advanced CI/CD-oriented software solutions such as Jenkins and OpenShift

Configuration as Code

- In the DevOps way of working, Configuration as code is the common approach
- Complex commands are to be avoided, use manifest files containing the desired configuration instead
- YAML is a common language to create these manifest files
- YAML is used in different DevOps based solutions, including Kubernetes and Ansible

The DevOps Cycle and its Tools

This is the framework for this course

- Coding: source code management tools - Git
- Building: continuous integration tools – Jenkins, OpenShift
- Testing: continuous testing tools – Jenkins, OpenShift
- Packaging: packaging tools – Jenkins, Dockerfile, Docker compose
- Releasing: release automation – Docker, Kubernetes, Openshift
- Configuring: configuration management tools – Ansible, Kubernetes
- Monitoring: applications monitoring – Kubernetes

Understanding Microservices

Understanding Microservices

- Microservices define an application as a collection of loosely coupled services
- Each of these services can be deployed independently
- Each of them is independently developed and maintained
- Microservices components are typically deployed as containers
- Microservices are a replacement of monolithic applications

Microservices benefits

- When broken down in pieces, applications are easier to build and maintain
- Smaller pieces are easier to understand
- Developers can work on applications independently
- Smaller components are easier to scale
- One failing component doesn't necessarily bring down the entire application

Coding: Using Git

Using Git in a Microservices Environment

- Git can be used for version control and cooperation between different developers and teams
- Using Git makes it easy to manage many changes in an effective way
- Different projects in a Microservice can have their own Git repository
- For that reason, Git and Microservices are a perfect match

Using Git

- Git is typically offered as a web service
- GitHub and GitLab are commonly used
- Alternatively, private Git repositories can be used

Understanding Git

- Git is a version control system that makes collaboration easy and effective
- Git works with a repository, which can contain different development branches
- Developers and users can easily upload as well as download new files to and from the Git repository
- To do so, a Git client is needed
- Git clients are available for all operating systems
- Git servers are available online, and can be installed locally as well
- Common online services include GitHub and GitLabs

Git Client and Repository

- The Git repository is where files are uploaded, and shared with other users
- Individual developers have a local copy of the Git repository on their computer and use the Git client to upload and download to and from the repository
- The organization of the Git client lives in the .git directory, which contains several files to maintain the status

Understanding Git Workflow

- To offer the best possible workflow control, A Git repository consists of three trees maintained in the Git-managed directory
  - The *working directory* holds the actual files
  - The *Index* acts as a staging area
  - The *HEAD* points to the last commit that was made

Applying the Git Workflow

- The workflow starts by creating new files in the working directory
- When working with Git, the **git add** command is used to add files to the index
- To commit these files to the head, use **git commit -m "commit message"**
- Use **git add origin https://server/reponame** to connect to the remote repository
- To complete the sequence, use **git push origin master**. Replace "master" with the actual branch you want to push changes to

Creating a GitHub Repository

- Create the repository on your GitHub server
- Set your user information
  - **git config --global user.name “Your Name”**
  - **git config --global user.email “you@example.com”**
- Create a local directory that contains a README.md file. This should contain information about the current repository
- Use **git init** to generate the Git repository metadata
- Use **git add <filenames>** to add files to the staging area
- From there, use **git commit -m "commit message"** to commit the files. This will commit the files to HEAD, but not to the remote repository yet
- Use **git remote add origin https://server/reponame**
- Push local files to remote repository: **git push -u origin master**

Understanding GitHub 2FA

- GitHub uses 2 Factor Authentication
- In the GitHub website, use Account > Account Security > Two-factor authentication to setup 2FA
- Check github.com/ateucher/setup-gh-cli-auth-2fa.md for instructions on how to use 2FA in CLI clients

Using Git Repositories

- Use **git clone https://gitserver/reponame** to clone the contents of a remote repository to your computer
- To update the local repository to the latest commit, use **git pull**
- Use **git push** to send local changes back to the Git server (after using **git add** and **git commit** obviously)

Uploading Changed Files

- Modified files need to go through the staging process
- After changing files, use **git status** to see which files have changed
- Next, use **git add** to add these files to the staging area; use **git rm <filename>** to remove files
- Then, commit changes using **git commit -m "minor changes"**
- Synchronize, using **git push origin master**
- From any client, use **git pull** to update the current Git clone

Removing Files from Git Repo's

- While removing files, the Git repository needs to receive specific instructions about the removal
  - **rm vault***
  - **git rm vault***
  - **git commit**
  - **git push**
- Check in the repository, the files will now be removed

Understanding Branches

- Branches are used to develop new features in isolation from the main branch
- The master branch is the default branch, other branches can be manually added
- After completion, merge the branches back to the master

Using Branches

- Use **git checkout -b dev-branch** to create a new branch and start using it
- Use **git push origin dev-branch** to push the new branch to the remote repository
- Use **git checkout master** to switch back to the master
- Use **git merge dev-branch** to merge the dev-branch back into the master
- Delete the branch using **git branch -d dev-branch**

Lab: Using Git

- Got to https://github.com, and create an account if you don't have an account yet
- Create a new Git repository from the website
- From a Linux client, create a local directory with the name of the Git repository
- Use the following commands to put some files in it
  - echo "new git repo" >README.md
  - git init
  - git add *
  - git status
  - git commit -m "first commit"
  - git remote add origin https://github.com/yourname/yourrepo
  - git push -u origin master

Understanding CI/CD

What is CI/CD

- CI/CD is Continuous integration and continuous deliver/continuous deployment
- It's a core Devops element that enforces automation in building, testing and deployment of applications
- The CI/CD pipeline is the backbone of modern DevOps operations
- In CI, all developers merge code changes in a central repository multiple times a day
- CD automates the software release process based on these frequent changes
- To do so, CD includes automated infrastructure provisioning and deployment

Understanding CI/CD pipelines

- The Ci/CD pipeline automates the software delivery process
- It builds code, runs tests (CI) and deploys a new version of the application (CD)
- Pipelines are automated so that errors can be reduced
- Pipelines are a runnable specification of the steps that a developer needs to perform to deliver a new version of a software product
- A CI/CD pipeline can be used as just a procedure that describes how to get from code to running software
- CI/CD pipelines can also be automated using software like Jenkins or OpenShift

Understanding Stages of Software Release

- 1: From source to Git: git push
- 2: From Git to running code: docker build, make
- 3: Testing: smoke test, unit test, integration test
- 4: Deployment: staging, QA, production

Source Stage

- Source code ends up in a repository
- Developers need to use **git push** or something to get their software into the repository
- The pipeline run is triggered by the source code repository

Build Stage

- The source code is converted into a runnable instance
- Source code written in C, Go or Java needs to be compiled
- Cloud-native software is deployed by using container images
- Failure to pass the build stage indicates there's a fundamental problem in either the code or the generic CI/CD configuration

Test Stage

- Automated testing is used to validate code correctness and product behavior
- Automated tests should be written by the developers
- Smoke tests are quick sanity checks
- End-to-end tests should test the entire system from the user point of view
- Typically, test suites are used
- Failure in this stage will expose problems that the developers didn't foresee while writing their code

Deploy Stage

- In deployment, the software is first deployed in a beta or staging environment
- After is passes the beta environment successfully, it can be pushed to the production environment for end users
- Deployment can be a continuous process, where different parts of a microservice are deployed individually and can automatically be approved and commited to the master branch for production

Benefits of using pipelines

- Developers can focus on writing code and monitoring behavior of their code in production
- QA have access to the latest version of the system at any time
- Product updates are easy
- Logs of all changes are always available
- Rolling back to a previous version is easy
- Feedback can be provided fast

Building– testing – packaging: Taking a Jenkins Quick Start

Using Jenkins Pipelines

- Jenkins is a very common open source tool to manage pipelines
- Jenkins pipelines offer four states of continuous delivery
  - Build
  - Deploy
  - Test
  - Release

Benefits of using Jenkins pipelines

- CI/CD is defined in a Jenkinsfile that can be scanned into Source Code Management
- It supports complex pipelines with conditional loop, forks, parallel execution and more
- It can resume from previously saved checkpoints
- Many plugins are available

Understanding Jenkinsfile

- Jenkinsfile stores the whole process as code
  - Jenkinsfile can be written in Groovy DSL syntax (scripted approach)
  - Jenkinsfile can be generated by a tool (declarative approach)

Installing Jenkins on Ubuntu

- **sudo apt-get install openjdk-11-jdk**
- **wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -**
- **sudo sh -c 'echo deb https://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'**
- **sudo apt-get update**
- **sudo apt-get install jenkins**
- read password: **sudo cat /var/lib/jenkins/secrets/initialAdminPassword**
- Access Jenkins at **http://localhost:8080**, skip over initial user creation
- Install suggested plugins

Understanding Jenkins Working

- Jenkins is going to run jobs on the computer that hosts Jenkins
- If the Jenkinsfile needs the Docker agent to run a Job, the Docker software must be installed on the host computer
- Also, the **jenkins** user must be a member of the **docker** group, so that this user has sufficient permissions to run the jobs

Installing Docker on Ubuntu

- 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
- sudo usermod -aG docker jenkins
- sudo systemctl restart jenkins

Understanding Jenkinsfile syntax

- **pipeline** is a mandatory block in the Jenkinsfile that defines all stages that need to be processed
- **node** is a system that runs a workflow
- **agent** defines which agent should process the CI/CD. **docker** is a common agent, and can be specified to run a specific image that runs the commands in the Jenkinsfile
- **stages** defines the different levels that should be processed: (build, test, qa, deploy, monitor)
- **steps** defines the steps in the individual stages

Using Jenkins

- Login in: http://localhost:8080
- Select Manage Jenkins > Manage Plugins.
- Select Available > Docker and Docker Pipelines plugins and install it

Creating your First Pipeline

- Select Dasboard > New Item
- Enter an item name: myfirstpipeline
- Select **Pipeline**, click **OK**
- Select **Pipeline**, set **Definition** to **Pipeline Script**
- In the script code, manually copy contents of https://github.com/sandervanvugt/devopsinfourweeks/firstpipeline
- Click **Apply**, **Save**
- In the menu on the left, select **Build Now**, it should run successfully
- Click the build time and date, from there select **Console Output** to see console output

Lab: Running a Pipeline

- In https://github.com/sandervanvugt/devopsinfourweeks, you'll find the file **secondpipeline**, which contains a pipeline script. Build a pipeline in Jenkins based on this file and verify that it is successful
- Notice that step two prompts for input. Click the step 2 field to view the prompt and provide your input

Configuration Management: Using Ansible in DevOps

What is Ansible?

- Ansible is a Configuration Management tool
- It can be used to manage Linux, Windows, Network Devices, Cloud, Docker and more
- The Control node runs the Ansible software, which is based on Python
- The Control node reaches out to the managed nodes to compare the current state with the desired state
- Desired state is defined in Playbooks, that are written in YAML

Why is Ansible DevOps?

- Ansible is Configuration as Code

Setting up a simple Ansible Environment

- On control hosts
  - Use CentOS 8.x
  - Enable EPEL repository
  - Enable host name resolving for all managed nodes
  - Generate SSH keys and copy over to managed hosts
  - Install Ansible software
  - Create an inventory file
- On managed hosts
  - Ensure Python is installed
  - Enable (key-based) SSH access
  - Make sure you have a user with (passwordless) sudo privileges

Lab: Setting up Ansible

- On the Ubuntu 20.04 LTS managed hosts
  - **sudo apt-install openssh-server**
- On the CentOS 8.x control host
  - **sudo dnf install epel-release**
  - **sudo dnf install –y ansible**
  - **sudo sh –c 'echo <your.ip.addr.ess> ubuntu.example.com ubuntu >> /etc/hosts'**
  - **ssh-keygen**
  - **ssh-copy-id ubuntu**
  - **echo ubuntu >> inventory**
  - **ansible ubuntu –m ping –i inventory –u student**

Using Ad-Hoc Commands

- Ansible provides 3000+ different modules
- Modules provide specific functionality and run as Python scripts on managed nodes
- Use **ansible-doc -l** for a list of all modules
- Modules can be used in ad-hoc commands:
  - **ansible ubuntu -i inventory -u student -b -K -m user -a "name=linda"**
  - **ansible ubuntu -i inventory -u student -b -K -m package -a "name=nmap"**

Using ansible.cfg

- While using Ansible commands, command line options can be used to provide further details
- Alternatively, use ansible.cfg to provide some standard values
- An example ansible.cfg is in the Git repository at https://github.com/sandervanvugt/devopsinfourweeks

Using Playbooks

- Playbooks provide a DevOps way for working with Ansible
- In a playbook the desired state is defined in YAML
- The **ansible-playbook** command is used to compare the current state of the managed machine with the desired state, and if they don't match the desired state is implemented
- **ansible-playbook -i inventory -u student -K my-playbook.yaml**

Day 1 Homework Assignment

Day 1 Homework

- Next week, we're going to work with Docker Containers managed by Ansible. Setup a CentOS 8.x based Ansible control node to manage an Ubuntu 20.04 LTS workstation according to the instructions in todays session so that we're ready to deploy Docker using Ansible. Further instructions on how to do this are provided next week.

Day 2

Day 2 Agenda

- Understanding Containers
- Using Ansible to Setup a Docker Environment
- Running Containers in Docker or Podman
- Managing Container Images
- Uploading Images to Docker Hub
- Managing Container Storage
- Accessing Container Workloads
- Using Docker Compose
- Preparing week 3 Setup and Homework

Poll Question

Have you attended last weeks class or seen its recording?

- yes
- no

Poll Question

Are you planning to attend the next days in this class?

- Yes
- Only day 3: Kubernetes
- Only day 4: OpenShift

Poll Question

How would you rate your own knowledge about containers?

- 0
- 1
- 2
- 3
- 4
- 5

Understanding Containers

Understanding Containers

- A container is a running instance of a container image that is fetched from a registry
- An image is like a smartphone App that is downloaded from the AppStore
- It's a fancy way of running an application, which includes all that is required to run the applicatoin
- A container is NOT a virtual machine
- Containers run on top of a Linux kernel, and depend on two important kernel features
  - Cgroups
  - Namespaces

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 was based on LXC, a Linux native container alternative that had been around a bit longer

Understanding Container Solutions

- Containers run on top of a container engine
- Different Container engines are provided by different solutions
- Some of the main solutions are:
  - Docker
  - Podman
  - LXC/LXD
  - systemd-nspawn

Understanding Container 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

Using Ansible to Setup a Docker Environment

Demo: Using Ansible to Setup Docker

- Make sure you have setup the Ubuntu 20.04 workstation for management by Ansible
- Use **ansible-playbook -u student -K -i inventory ansible-ubuntu.yml** to set up the Ubuntu host
- On Ubuntu, log out and log in as your user **student**
- Use **docker run hello-world**

Running Containers in Docker and Podman

Podman or Docker?

- Red Hat has changed from Docker to Podman as the default container stack in RHEL 8
- Docker is no longer supported in RHEL 8 and related distributions
- Even if you can install Docker on top of RHEL 8, you shouldn't do it as it will probably break with the next software update
- Podman is highly compatible with Docker
- By default, Podman runs rootless containers, which have no IP address and cannot bind to privileged ports
- Both Docker as Podman are based on OCI standards
- For optimal compatibility, install the **podman-docker** package

Demo: Running Containers

- **docker run ubuntu**
- **docker ps**
- **docker ps -a**
- **docker run -d nginx**
- **docker ps**
- **docker run -it ubuntu --name ubuntu sh; Ctrl-p, Ctrl-q**
- **docker inspect ubuntu**
- **docker rm ubuntu**
- **docker run --name webserver --memory="128m" -d -p 8080:80 nginx**
- **curl localhost:8080**

Managing Container Images

Understanding Images

- A container is a running instance of an image
- The image contains application code, language runtime and libraries
- External libraties such as libc are typically provided by the host operating system, but in container is included in the image
- While starting a container it adds a writable layer on the top to store any changes that are made while working with the container
- These changes are ephemeral
- Container images are highly compatible, and either defined in Docker or in OCI format

Getting Container Images

- Containers are normally fetched from registries
- Public registries such as https://hub.docker.com are available
- Red hat offers https://quay.io as a registry with more advanced CI features offered
- Alternatively, private registries can easily be created
- Use Dockerfile to create custom images

Fetching Images from Registries

- By default, Docker fetches containers from Docker Hub
- In Podman, the /etc/containers/registries.conf file is used to specify registry location
- Alternatively, the complete path to an image can be used to fetch it from a specific registry: **docker pull localhost:5000/fedora:latest**

Understanding Image Tags

- Normally, different versions of images are available
- If nothing is specified, the latest version is pulled
- Use tags to pull a different version: **docker pull nginx:1.14**

Demo: Managing Container Images

- Explore **https://hub.docker.io**
- **docker search mariadb** will search for the mariadb image
- **docker pull mariadb**
- **docker images**
- **docker inspect mariadb**
- **docker image history mariadb**
- **docker image rm mariadb**

Creating a Private Registry

Running a Private Registry

- A private registry can easily be implemented by running the **registry** image as a container
- Expose the registry on local port 5000
- Tag the image with the hostname and port in the first part of the tag, to ensure that Docker interprets it as the location of the registry: **docker tag fedora localhost:5000/fedora**
- Consider using a dedicated volume for storing images in your private registry: **docker run -d -p 5000:5000 --restart=always --name registry -v /mnt/registry:/var/lib/registry registry:2**

Running a Private Registry for External Use

- To provide services for external users, a private registry should be configured with TLS certificates for secure access
- Assuming that a CA-signed certificate key pair is available as /certs/my.key and /certs/my.crt, use the following command to start the secured registry:
- **docker run -d --restart=always --name registry -v "$(pwd)"/certs:/certs -e REGISTRY_HTTP_ADDR=0.0.0.0:443 -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/my.crt -e REGISTRY_HTTP_TLS_KEY=/certs/my.key -p 443:443 registry:2**

Demo: Running a local 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

Using Dockerfile

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
- Tip: images on hub.docker.com have a link to Dockerfile. Read it to understand how an image is build using Dockerfile!

Using Dockerfile Instructions

- FROM: identifies the base image to use. This must be the first instruction in Dockerfile
- MAINTAINER: the author of the image
- RUN: executes a command while building the container, it is executed before the container is run and changes what is in the resulting container
- CMD: specifies a command to run when the container starts
- EXPOSE: exposes container ports on the container host
- ENV: sets environment variables that are passed to the CMD
- ADD: copies files from the host to the container. By default files are copied from the Dockerfile directory
- ENTRYPOINT: specifies a command to run when the container starts

Using Dockerfile Instructions

- VOLUME: specifies the name of a volume that should be mounted from the host into the container
- USER: identifies the user that should run tasks for building this container, use for services to run as a specific user
- WORKDIR: set the current working directory for commands that are running from the container

Understanding ENTRYPOINT and CMD

- Both ENTRYPOINT and CMD specify a command to run when the container starts
- CMD specifies the command that should be run by default after starting the container. You may override that, using **docker run mycontainer othercommand**
- ENTRYPOINT can be overridden as well, but it's more work: you need **docker run --entrypoint mycommand mycontainer** to override the default command that is started
- Best practice is to use ENTRYPOINT in situations where you wouldn't expect this default command to be overridden

ENTRYPOINT and CMD Syntax

- Commands in ENTRYPOINT and COMMAND can be specified in different ways
- The most common way is the Exec form, which is shaped as **<instruction> ["executable", "arg1", "arg2"]**
- The alternative is to use Shell form, which is shaped as **<instruction> <command>**
- While shell form seems easier to use, it runs <command> as an argument to /bin/sh, which may lead to confusion

Demo: Using a Dockerfile

- Dockerfile demo is in https://github.com/devopsinfourweeks/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 if you need to run 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!

Building Images in **podman**

- **podman build** allows you to build images based on Dockerfile
- Alternatively, use **buildah**
- **buildah build** is doing the exact same thing
- **buildah from scratch** allows you to build images from scratch, using script-like style
- See also: https://developers.redhat.com/blog/2019/02/21/podman-and-buildah-for-docker-users/

Lab: Working with Dockerfile

- Create a Dockerfile that deploys an httpd web server that is based on the latest Fedora container image. Use a sample file index.html which contains the text "hello world" and copy this file to the /var/www/html directory. Ensure that the following packages are installed: nginx curl wget
- Use the Dockerfile to generate the image and test its working

Using Tags

- Consider using tags on custom images you create
- Without setting a tag, the default tag :latest is used
- Use **docker tag** to manually set a tag: **docker tag myapache:1.0**
- Consider using meaningfull tags **docker tag myapache:testing; docker tag myapache:production**

Using **docker commit**

- **docker commit** allows you to easily save changes to a container in an image
- The container image keeps its original metadata
  - Use on a running container
  - **docker commit -m my-change running_containername my-container**
  - **docker images**
- **docker push my-container**

Managing Private Registries

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 run -d -p 5000:5000 --restart=always --name registry registry:2**
- **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

Publishing on Docker Hub

Creating an autobuild Repository on Docker Hub

- Access https://github.com
- If required, create an account and log in
- Click **Create Repository**
- Enter the name of the new repo; e.g. **devops** and set to **Public**
- Check **Settings** > **Webhooks**. Don't change anything, but check again later

Creating an autobuild Repository on Docker Hub

- On a Linux console, create the local repository
  - **mkdir devops**
  - **echo "hello" >> README.md**
  - **cat > Dockerfile <<EOF**
    FROM busybox
    CMD echo "Hello world!"
    EOF
  - **git init**
  - **git add ***
  - **git commit -m "initial commit"**
  - **git remote add origin https://github.com/yourname/devops.git**
  - **git push -o origin master**

Creating an autobuild Repository on Docker Hub

- Access https://hub.docker.com
- If required, create an account and log in
- Click **Create Repository**
- Enter the name of the new repo; e.g. **devops** and set to **Public**
- Under Build Settings, click Connected and enter your GitHib "organization" as well as a repository

Creating an autobuild Repository on Docker Hub

- Still from hub.docker.com: Add a Build Rule that sets the following:
  - Source: Branch
  - Source: master
  - Docker Tag: latest
  - Dockerfile location: Dockerfile
  - Build Context: /
- Check Builds > Build Activity to see progress
- Once the build is completed successfully, from a terminal user **docker pull yourname/devops:latest** to pull this latest image
- On GitHub, check Settings > Webhooks for your repo

Creating an autobuild Repository on Docker Hub

- From the Git repo on the Linux console: edit the Dockerfile and add the following line: MAINTAINER yourname your@mailaddress.com
- git status
- git add *
- git commit -m "minor update"
- git push -u origin master
- From Docker hub: Check your repository > Builds > Build Activity. You'll see that a new automatic buold has been triggered

Managing Container Storage

Configuring Storage

- To work with storage, bind mounts and volumes can be used
- A bind mount provides access to a directory on the Docker host
- Volumes exist outside of the container spec, and as such outlive the container lifetime
- Volumes offer an option to use other storage types as well
- Within Docker-CE, volume types are limited to **local** and **nfs**
- In Kubernetes or Docker Swarm more useful storage types are provided

Demo: Using 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

Demo: Using an NFS-based Volume -1

- **docker volume create --driver local --opt type=nfs --opt o=addr=127.0.0.1,rw --opt device=:/nfsdata nfsvol**
- **docker volume ls**
- **docker volume inspect nfsvol**
- **docker run -it --name nfstest --rm --mount source=nfsvol,target=/data nginx:latest /bin/sh**
- **touch /data/myfile; exit**
- **ls /nfsdata**

Lab: Managing Volumes

- Use **docker volume create myvol** to create a volume that uses local storage as its backend
- Inspect the volume to see what it's doing with files that are created
- Mount this volume in an nginx container in the directory /data. Create a file in this directory and verify this file is created on the local volume storage backend

Using Docker Compose

Understanding Docker Compose

- Docker Compose uses the declarative approach to start Docker containers, or Microservices consisting of multiple Docker containers
- The YAML file is used to include parameters that are normally used on the command line while starting a Docker container
- To use it, create a **docker-compose.yml** file in a directory, and from that directory run the **docker-compose up -d** command
- Use **docker-compose down** to remove the container

Demo: Bringing up a Simple Nginx Server

- Use the simple-nginx/docker-compose.yml file from https://github.com/sandervanvugt/devopsinfourweeks
- **cd simple-nginx**
- **docker-compose up -d**
- **docker ps**

Demo: Bringing up a Microservice

- Use the wordpress-mysql/docker-compose.yml file from https://github.com/sandervanvugt/devopsinfourweeks
- **cd wordpress-mysql**
- **docker-compose up -d**
- **docker ps**

Lab: Using Docker Compose

- Start an nginx container, and copy the /etc/nginx/conf.d/default.conf configuration file to the local directory ~/nginx-conf/
- Use Docker compose to deploy an application that runs Nginx. Expose the application on ports 80 and 443 and mount the configuration file by using a volume that exposes the ~/nginx-conf directory

Homework

Day 2 homework

- For next weeks session, prepare a CentOS 7.x virtual machine that has at least 4GiB RAM, 20 GiB disk and 2 CPUs. Install this machine with the minimal installation server pattern so that it is ready for installation of a Kubernetes AiO server. Further instructions are provided next week.