Agenda

Poll Question

What is your experience with DevOps

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

Poll Question

What is your experience with Kubernetes

- What is Kubernetes?
- None
- Just starting
- Reasonable
- Advanced

Poll Question

Are you using / going to use OpenShift

- yes
- no

Poll Question

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

Course Objectives

- In this course, you will learn about DevOps and GitOps and common 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

Required Course Setup

- To follow along with demos, you need the access to the following:
  - One running Ubuntu (virtual) machine with at least 4 GiB RAM and 2 vCPUs

Day 1 Agenda

- Understanding DevOps
- Understanding GitOps
- Understanding 12 Factor App Development model
- Using Git
- Using CI/CD and Jenkins
- Running Containers in Docker or Podman

Day 2 Agenda

- Managing Container Images
- Using Webhooks to trigger Image Builds from Git Repositories
- Understanding Kubernetes
- Using Kubernetes and OpenShift
- Exploring Basic Kubernetes and OpenShift Skills
- Using Kubernetes for DevOps
- Exposing Applications

Day 3 Agenda

- Configuring Application Storage
- Implementing Decoupling in Kubernetes
- Using Helm Charts, Operators and Custom Resources
- Blue/Green and Canary Deployments
- Using Tekton CI/CD
- Exploring ArgoCD
- Building OpenShift Applications from Git Source Code

Day 1

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 is commonly implemented, using anything from simple GitHub repositories, up to advanced CI/CD-oriented software solutions such as Jenkins and OpenShift

DevOps Key Components

- Different technologies are used to implement devops:
- Configuration as Code
- The DevOps Cycle and pipelines
- Microservices
- The 12-factor application

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

Understanding The DevOps Cycle

This is the framework for this course

- Coding: source code management tools
- Building: continuous integration tools
- Testing: continuous testing tools
- Packaging: packaging tools
- Releasing: release automation
- Configuring: configuration management tools
- Monitoring: applications monitoring

Understanding GitOps

Understanding GitOps

- GitOps uses Git repositories as a single source of truth to deliver infrastructure as code
- It uses CI/CD pipelines to do this in a structured way
  - The code check the CI process
  - The CD process applies security, infrastructure as code and more
- Changes are tracked which makes update and rollback easy
- Common tools in GitOps are
  - Git repositories
  - Kubernetes
  - CI/CD
  - Configuration Management

Understanding 12-Factor Apps

What is the 12-Factor App

- The 12-factor app is a development methodology for building apps that
  - Use declarative formats
  - Offer maximum portability
  - Are suitable for deployment on cloud platforms
  - Enable continuous deployment, which minimizes divergence
  - Allows for easy scaling of applications
- 12-factor app based DevOps explains why orchestrating containerized workloads in Kubernetes is essential
- See 12factor.net for more details

The 12 factors (1)

- I. Codebase: One codebase, tracked in revision control, many deployes: Git, declarative code, Dockerfile
- II. Dependencies: Explicitely declare and isolate dependencies: Kubernetes Probes, init containers
- III. Config: Store config in the environment: ConfigMap
- IV: Backing Services: Treat Backing services as attached resources: Service Resources, pluggable networking
- V: Build, release, run: Strictly separate build and run stages: CI/CD, S2I, Git branches, Helm
- VI: Processes: Execute the app as one or more stateless processes: Microservices, Linux kernel namespaces

The 12 factors (2)

- VII: Port Binding: Export services via port binding: K8s Services, Routes
- VIII: Concurrency: Scale out via the process model: K8s ReplicaSets
- IX: Disposability: Maximize robustness with fast startup and graceful shutdown: K8s probes
- X: Dev/prod parity: Keep development, staging and production as similar as possible: Containers
- XI: Logs: Treat logs as event streams: Logs stored in the orchestration layer
- XII: Admin processes: Run admin/management tasks as one-off processes: Ansible Playbooks, Kubernetes Jobs

Coding: Using Git

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**

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

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 to ensure consistency
- CD automates the software release process based on these frequent changes
- To do so, CD includes automated infrastructure provisioning and deployment

What is a Pipeline?

- A pipeline is nothing more than a series of steps that must be performed
- In a generic sense, any series of steps can be executed as a pipeline
- In CI/CD the purpose of the pipeline is to deliver a new version of software
- To do so, the software development life cycle is automated in the pipeline

Understanding CI and its tools

- A version control system (such as Git) allows developers to work with branches which are later merged into the main code
- CI automation is used to watch the repository for changes. Different options exist:
  - Webhooks on Docker image repositories
  - Jenkins
  - Other, such as Tekton or Gitlab actions
- When CI is automated, changes must be verified and that is what happens in a pipeline that goes through different phases

Understanding CD

- At the end of the CI pipeline, there is continuous delivery, which ensures that software is published once it is ready
- A deployable version is created every time the pipeline runs successfully, and that can happen multiple times a day
- Continuous delivery can be enhanced to continuous deployment, where the release is not only published but also deployed without further human intervention

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
- All of these can be automated in a CI/CD pipeline

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
- 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
  - Kubernetes Blue/Green and Canary deployments can be used for this purpose
- After it 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

Understanding CI/CD and Kubernetes

- CI/CD procedures can be integrated in Kubernetes
- It starts with code that is checked out to Git
- From there, software builds into container images can be automated based on webhooks
- Deployment can be automated using tools like Jenkins, ArgoCD, OpenShift or Tekton
- The result is a running containerized application, which will automatically be updated whenever that is needed
- In this procedure Kubernetes is taking care of a smooth zero-downtime application update

Building– testing – packaging: Taking a Jenkins Quick Start

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; in this demo we'll have a look at the Docker plugin

Understanding Jenkins Workflow

- Jenkins can work with different sources, including GitHub projects
- It can be configured with Build Triggers such as webhooks to completely automate the pipeline procedure
- The pipeline script is used to define the different steps in the pipeline in detail
- Plugins such as Docker Pipeline define where the result should be created

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)

Demo: Installing Jenkins on Ubuntu

- **sudo apt 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 update**
- **sudo apt 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

Demo: Installing Docker on Ubuntu

- sudo apt 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 update
- sudo apt 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

Demo: 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
- Use **sudo docker images** to verify an image was created

Demo: 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

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 application
- 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
- Docker still is the main solution for running containers
- Docker Hub is the main registry, hosting more than a million container images

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

RHEL: 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 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**

Day 2

Day 2 Agenda

- Managing Container Images
- Triggering Image Builds from Git Repositories
- Managing Container Storage
- Understanding Kubernetes
- Using Kubernetes and OpenShift
- Exploring Basic Kubernetes and OpenShift Skills

Managing Container Images

Understanding Images

- A container is a running instance of an image
- The image contains application code, language runtime and libraries
- External libraries 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
- Alternatively, private registries can easily be created
- Use Dockerfile/Containerfile 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**

Demo: Managing Container Images

- Explore **https://hub.docker.com**
- **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**

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!

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

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

Publishing on Docker Hub

Demo: Creating an autobuild Repo 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

Demo: Creating an autobuild Repo 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 -u origin master**

Demo: Creating an autobuild Repo 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

Demo: Creating an autobuild Repo 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 (may take a minute to synchronize)
- On GitHub, check Settings > Webhooks for your repo - settings should be automatically added

Demo: Creating an autobuild Repo 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 build has been triggered (should be fast)

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

Understanding Kubernetes

Understanding Kubernetes

- Kubernetes offers enterprise features that are needed in a containerized world
  - Scalability
  - Availability
  - Decoupling between static code and site specific data
  - Persistent external storage
  - The flexibility to be used on premise or in cloud
- Kubernetes is the de facto standard and currently there are no relevant competing products

Options for Using Kubernetes

- Managed in Cloud
- Minikube
- In Docker Desktop
- AiO
- As a distribution: OpenShift (or others)
- If using OpenShift: CodeReady Containers

Installing Kubernetes

- In cloud, managed Kubernetes solutions exist to offer a Kubernetes environment in just a few clicks
- On premise, administrators can build their own Kubernetes cluster using **kubeadm**
- For testing, **minikube** can be used
- In this course we'll use Minikube on Ubuntu
- On Ubuntu, use **minikube-docker-setup.sh**

Getting Started with OpenShift CodeReady Containers

Understanding CodeReady Containers

- CodeReady Containers (CRC) is a free all-in-one OpenShift solution
- You need a free Red Hat developer account
- CodeReady Containers can be installed in different ways
  - On top of your current OS
  - Isolated in a Linux VM
- To prevent having conflicts with other stuff running on your computer, it's recommended to install in an isolated VM
- For other usage options, see here: https://developers.redhat.com/products/codeready-containers/overview

Installing CRC in an Isolated VM

- The VM needs the following
  - 12 GB RAM
  - 4 CPU cores
  - 40 GB disk
  - Support for nested virtualization
- Download the tar ball and the pull-secret
- Extract the tarball
- move the **crc** file to /usr/local/bin
- **crc setup**
- **crc start -p pull-secret -m 8192**

Running Applications in Kubernetes

Understanding Kubernetes Resources

- Kubernetes resources are defined in the APIs
- Use **kubectl api-resources** for an overview
- Kubernetes API's are extensible, which means that you can add your own resources

Understanding Kubernetes Key Resources

- Pod: used to run one (or more) containers and volumes
- Deployment: adds scalability and update strategy to pods
- Service: exposes pods for external use
- Persistent Volume Claim: connects to persistent storage
- ConfigMap: used to store site specific data separate from pods

Exploring **kubectl**

- **kubectl** is the main management interface
- Make sure that **bash-completion** is installed for awesome tab completion
- **source <(kubectl completion bash)**
- Explore **kubectl -h** at all levels of **kubectl**

Running Applications in Kubernetes

- **kubectl create deployment** allows you to create a deployment
- **kubectl run** allows you to run individual pods
- Individual pods (aka "naked pods") are unmanaged and should not be used
- **kubectl get pods** will show all running Pods in the current namespace
- **kubectl get all** shows running Pods and related resources in the current namespace
- **kubectl get all -A** shows resources in all namespaces

Troubleshooting Kubernetes Applications

- **kubectl describe pod <podname>** is the first thing to do: it shows events that have been generated while defining the application in the Etcd database
- **kubectl logs** connects to the application STDOUT and can indicate errors while starting application. This only works on running applications
- **kubectl exec -it <podname> -- sh** can be used to open a shell on a running application

Lab: Troubleshooting Kubernetes Applications

- Use **kubectl create deployment --image=busybox** to start a Busybox deployment
- It fails: use the appropriate tools to find out why
- After finding out why it fails, delete the deployment and start it again, this time in a way that it doesn't fail

Day 3

Using Kubernetes for DevOps

Declarative versus Imperative

- In Imperative mode, the administrator uses command with command line options to define Kubernetes resources
- In Declarative mode, Configuration as Code is used by the DevOps engineer to ensure that resources are created in a consistent way throughout the entire environment
- To do so, YAML files are used
- YAML files can be written from scratch (not recommended), or generated: **kubectl create deployment mynginx --image=nginx --replicas=3 --dry-run=client -o yaml > mynginx.yaml**
- For complete documentation: use **kubectl explain <resource>.spec**

Exposing Applications

Understanding Application Access

- Kubernetes applications are running as scaled pods in the pod network
- The pod network is provided by the **kube-apiserver** and not reachable from the outside
- To expose access to applications, **service** resources are used

Demo: Exposing Applications

- **kubectl create deploy mynginx --image=nginx --replicas=3**
- **kubectl get pods -o wide**
- **kubectl expose deploy mynginx --type=NodePort --port=80**

Configuring Application Storage

Understanding K8s Storage Solutions

- Pod storage by nature is ephemeral
- Pods can refer to external storage to make it less ephemeral
- Storage can be decoupled by using Persistent Volume Claim (PVC)
- PVC addresses Persistent Volume
- Persistent Volume can be manually created
- Persistent Volume can be automatically provisioned using StorageClass
- StorageClass provides default storage in specific (cloud) environments
- Check **pv-pvc-pod.yaml** for an example

Implementing Decoupling in Kubernetes

Demo: Running MySQL

- **kubectl run mymysql --image=mysql:latest**
- **kubectl get pods**
- **kubectl describe pod mymysql**
- **kubectl logs mymysql**

Providing Variables to Kubernetes Apps

- In imperative way, the **-e** command line option can be used to provide environment variables to Kubernetes applications
- That's not very DevOps though, and something better is needed
- But let's verify that it works first: **kubectl run newmysql --image=mysql --env=MYSQL_ROOT_PASSWORD=password**
- Notice alternative syntax: **kubectl set env deploy/mysql MYSQL_DABASASE=mydb**

Understanding ConfigMaps

- ConfigMaps are used to separate site-specific data from static data in a Pod
  - Variables: **kubectl create cm variables --from-literal=MYSQL_ROOT_PASSWORD=password**
  - Config files: **kubectl create cm myconf --from-file=my.conf**
- Secrets are base64 encoded ConfigMaps
- Adressing the ConfigMap from a Pod depends on the type of ConfigMap
  - Use **envFrom** to address variables
  - Use **volumes** to mount ConfigMaps that contain files

Demo: Using a ConfigMap for Variables

- **kubectl create cm myvars --from-literal=VAR1=goat --from-literal=VAR2=cow**
- **kubectl create -f cm-test-pod.yaml**
- **kubectl logs test-pod**

Demo: Using a ConfigMap for Storage

- **kubectl create cm nginxconf --from-file nginx-custom-config.conf**
- **kubectl create -f nginx-cm.yml**
- Did that work? Fix it!
- **kubectl exec -it nginx-cm -- /bin/bash**
- **cat /etc/nginx/conf.d/default.conf**

Lab: Running MySQL the DevOps way

- Create a ConfigMap that stores all required MySQL variables
- Start a new mysql pod that uses the ConfigMap to ensure the availability of the required variables within the Pod

Using Blue/Green Deployments in Kubernetes and OpenShift

Understanding Blue/Green Deployments

- A blue/green deployment is a way of accomplishing a zero-downtime application upgrade
- The blue deployment is the current application
- The green deployment is the new application
- Once the green deployment is ready, traffic is re-routed to the new application version
- Kubernetes Deployment and Service resources make implementing blue/green deployment in Kubernetes easy

Procedure Overview

- Notice this can be done in multiple ways
- Start with already running deployment and service
- Create new deployment running the new version
- Perform a health check
- If health check passes, update the load balancer and remove old deployment
- if health check fails, stop

Detailed Procedure

- **kubectl create deployment blue-nginx --image=quay.io/bitnami/nginx:1.14 --replicas=3**
- **oc expose deployment blue-nginx --port=80 --name=bgnginx**
- **oc get deploy blue-nginx -o yaml > green-nginx.yaml**
  - Change Image version
  - Change "blue" to "green" throughout
- **oc create -f green-nginx.yaml**
- **oc get pods**
- **oc delete svc bgnginx**
- **oc expose deployment green-nginx --port=80 --name=bgnginx**
- **oc delete deployment blue-nginx**

Using Canary Deployments in Kubernetes

Understanding Canary Deployments

- A Canary Deployment is an update strategy where you first push the update at small scale to see if it works well
- In terms of Kubernetes, you could imagine a deployment that runs 4 replicas
- Next you add a new deployment that uses the same label
- Then you create a service that uses the same selector label for all
- As the service is load balancing, only 1 out of 5 requests would be serviced by the newer version
- And if that doesn't seem to be working, you can easily delete it

Demo (1): Running the Old version

- **kubectl create deploy old-nginx --image=nginx:1.14 --replicas=3 --dry-run=client -o yaml > ~/oldnginx.yaml**
- **vim oldnginx.yaml**
  - set labels: type: canary in deploy metadata as well as pod metadata
- **kubectl create -f oldnginx.yaml**
- **kubectl expose deploy old-nginx --name=oldnginx --port=80 --selector type=canary**
- **kubectl get svc; kubectl get endpoints**
- **minikube ssh; curl <svc-ip-address>** a few times, you'll see all the same

Demo (2): Create a ConfigMap

- **kubectl cp <old-nginx-pod>:/usr/share/nginx/html/index.html index.html**
- **vim index.html**
  - Add a line that uniquely identifies this as the canary pod
- **kubectl create configmap canary --from-file=index.html**
- **kubectl describe cm canary**

Demo (3): Prepare the new version

- **cp oldnginx.yaml canary.yaml**
- **vim canary.yaml**
  - image: nginx:latest
  - replicas: 1
  - :%s/old/new/g
  - Mount the configMap as a volume (see Git repo canary.yaml)
- **kubectl create -f canary.yaml**
- **kubectl get svc; kubectl get endpoints**
- **minikube ssh; curl <service-ip>** and notice different results: this is Canary in action

Activating the Newer Version

- After verifying the newer version works successfully, you need to finalize the procedure
- Use **kubectl get deploy** to verify the names of the old and the new deployment
- **kubectl delete deploy** to delete the old deployment
- Use **kubectl scale** to scale the canary deployment up to the desired number of replicas

Using Helm Charts

Understanding Helm

- Helm is used to streamline installing and managing Kubernetes applications
- Helm consists of the **helm** tool, which needs to be installed, and a chart
- A chart is a Helm package, which contains the following:
  - A description of the package
  - One or more templates containing Kubernetes manifest files
- Charts can be stored locally, or accessed from remote Helm repositories

Demo: Installing the Helm Binary

- Fetch the binary from web page https://github.com/helm/helm/releases; check for the latest release!
- **tar xvf helm-xxxx.tar.gz**
- **sudo mv linux-amd64/helm /usr/local/bin**
- **helm version**

Getting Access to Helm Charts

- The main site for finding Helm charts, is through https://artifacthub.io
- This is a major way for finding repository names
- Search for specific software here, and run the commands to install it; for instance, to run the Kubernetes Dashboard:
  - **helm repo add kubernetes-dashboard https://kubernetes.github.io/dashboard/**
  - **helm install kubernetes-dashboard kubernetes-dashboard/kubernetes-dashboard**
- Note: in Minikube, use **minikube addons enable dashboard** for easier installation

Demo: Managing Helm Repositories

- **helm repo add bitnami https://charts.bitnami.com/bitnami**
- **helm repo list**
- **helm search repo bitnami**
- **helm repo update**

Installing Helm Charts

- After adding repositories, use **helm repo update** to ensure access to the most up-to-date information
- Use **helm install** to install the chart with default parameters
- After installation, use **helm list** to list currently installed charts
- Optionally, use **helm delete** to remove currently installed charts

Demo: Installing Helm Charts

- **helm install bitnami/mysql --generate-name**
- **kubectl get all**
- **helm show chart bitnami/mysql**
- **helm show all bitnami/mysql**
- **helm list**
- **helm status mysql-xxxx**

Customizing Before Installing

- A Helm chart consists of templates to which specific values are applied
- The values are stored in the values.yaml file, within the helm chart
- The easiest way to modify these values, is by first using **helm pull** to fetch a local copy of the helm chart
- Next use your favorite editor on **chartname/values.yaml** to change any values

Demo: Customizing Before Install

- **helm show values bitnami/nginx**
- **helm pull bitnami/nginx**
- **tar xvf nginx-xxxx**
- **vim nginx/values.yaml**
- **helm template --debug nginx**
- **helm install -f nginx/values.yaml my-nginx nginx/**

Using Custom Resources and Operators

Understanding Custom Resources

- Custom Resource Definitions (CRDs) allow users to add custom resources to clusters
- Doing so allows anything to be integrated in a cloud-native environment
- The CRD allows users to add resources in a very easy way
  - The resources are added as extension to the original Kubernetes API server
  - No programming skills required
- The alternative way to build custom resources, is via API integration
  - This will build a custom API server
  - Programming skills are required

Creating Custom Resources

- Creating Custom Resources using CRDs is a two-step procedure
- First, you'll need to define the resource, using the CustomResourceDefinition API kind
- After defining the resource, it can be added through its own API resource

Lab: Creating Custom Resources

- **cat crd-object.yaml**
- **kubectl create -f crd-object.yaml**
- **kubectl api-resources | grep backup**
- **cat crd-backup.yaml**
- **kubectl create -f crd-backup.yaml**
- **kubectl get backups**

Understanding Operators and Controllers

- Operators are custom applications, based on Custom Resource Definitions
- Operators can be seen as a way of packaging, running and managing applications in Kubernetes
- Operators are based on Controllers, which are Kubernetes components that continuously operate dynamic systems
- The Controller loop is the essence of any Controllers
- The Kubernetes Controller manager runs a reconciliation loop, which continuously observes the current state, compares it to the desired state, and adjusts it when necessary
- Operators are application-specific Controllers

Understanding Operators and Controllers

- Operators can be added to Kubernetes by developing them yourself
- Operators are also available from community websites
- A common registry for operators is found at operatorhub.io (which is rather OpenShift oriented)
- Many solutions from the Kubernetes ecosystem are provided as operators
  - Prometheus: a monitoring and alerting solution
  - Tigera: the operator that manages the calico network plugin
  - Jaeger: used for tracing transactions between distributed services

Lab: Installing the Calico Plugin

- minikube stop; minikube delete
- minikube start --network-plugin=cni --extra-config=kubeadm.pod-network-cidr=10.10.0.0/16
- kubectl create -f https://docs.projectcalico.org/manifests/tigera-operator.yaml
- kubectl api-resources | grep tigera
- kubectl get pods -n tigera-operator tigera-operator-xxx-yyy
- wget https://docs.projectcalico.org/manifests/custom-resources.yaml
- sed -i -e s/192.168.0.0/10.10.0.0/g custom-resources.yaml
- kubectl get installation -o yaml
- kubectl get pods -n calico-system

Using Tekton CI/CD

Understanding Tekton

- Tekton is a Kubernetes native open source framework for creating CI/CD systems
- It helps in automating many things using CI/CD pipelines
  - Create and deploy images
  - Manage version control
  - Implementing blue/green deployments and canary deployments
- Tekton is part of the CD foundation, which is a part of Linux Foundation
- For more information, see https://tekton.dev

Getting Started with Tekton

- Tekton integrates with Kubernetes through Custom Resource Definitions
- Main Tekton resources are managed using **kubectl**
- It also has its own CLI

Understanding Tekton Resources

- A **Task** consists of a series of steps, where each steps runs a specific tool
  - A step is an operation in the CI/CD workflow
  - A task is run as a Kubernetes Pod
  - Output of a previous step can be used as input for the next Step
- A **Pipeline** is a series of tasks, where the output of a previous task can be used as input for the next task
- A **TaskRun** instantiates (runs) a specific task
- A **PipelineRun** instantiates a specific pipeline
- Tekton Triggers are an optional component that can automatically trigger a pipeline when a specific element occurs

Demo: Running Tasks

- See also: https://tekton.dev/docs/getting-started/tasks/
- Use **kubectl apply -f https://storage.googleapis.com/tekton-releases/pipeline/latest/release.yaml** to install Tekton CRDs
- **kubectl api-resources | grep -i tekton**
- **kubectl get pods -n tekton-pipelines --watch**
- **cat hello-world.yaml**
- **kubectl apply -f hello-world.yaml**
- **cat hello-world-run.yaml**
- **kubectl apply -f hello-world-run.yaml**
- **kubectl get taskrun hello-task-run**
- **kubectl logs --selector=tekton.dev/taskRun=hello-task-run**

Demo: Install Tekton CLI

- see https://tekton.dev/docs/cli/
- **sudo apt update; sudo apt install -y gnupg**
- **sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 3EFE0E0A2F2F60AA**
- **echo "deb http://ppa.launchpad.net/tektoncd/cli/ubuntu eoan main"|sudo tee /etc/apt/sources.list.d/tektoncd-ubuntu-cli.list**
- **sudo apt update && sudo apt install -y tektoncd-cli**

Demo: Running a Pipeline

- see https://tekton.dev/docs/getting-started/pipelines/
- **cat goodbye-world.yaml**
- **kubectl apply -f goodbye-world.yaml**
- **cat hello-goodbye-pipeline.yaml**
- **kubectl apply -f hello-goodbye-pipeline.yaml**
- **cat hello-goodbye-pipeline-run.yaml**
- **kubectl apply -f hello-goodbye-pipeline.yaml**
- **tkn pipelinerun logs hello-goodbye-run -f -n default**
- For additional code examples see here: https://tekton.dev/docs/pipelines/tasks/#code-examples

Understanding Tekton Triggers

- Tekton Triggers can automatically execute TaskRuns and PipelineRuns based on detected events
- To use Tekton Triggers, you need to install the Tekton Triggers extension on the current cluster
- Using Triggers allows you to automatically execute a PipelineRun when a git push is detected, and much more
- Installing Triggers: https://tekton.dev/docs/triggers/install/
- Getting started with Triggers: https://github.com/tektoncd/triggers/blob/main/docs/getting-started/README.md

Exploring ArgoCD

Understanding ArgoCD

- ArgoCD integrates to Kubernetes to synchronize applications in Git
- It comes with its CLI or web based management interface that allows you to manage application state

Demo: Installing ArgoCD

- See also https://argo-cd.readthedocs.io/en/stable/getting_started/
- **kubectl create namespace argocd**
- **kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml**
- **kubectl get all -n argocd**
- Download latest version of the **argocd** binary from here: **https://github.com/argoproj/argo-cd/releases/tag/v2.3.4**
- **sudo mv ~/Downloads/argocd-linux-amd64 /usr/local/bin/argocd**

Demo: Accessing ArgoCD

- **kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d; echo**
- **kubectl port-forward svc/argocd-server -n argocd 8080:443**
- **argocd login localhost:8080**
  - **username: admin**
  - **password: as printed above**
- **argocd account update-password**

Demo: Synchronizing an Application

- **argocd app create guestbook --repo https://github.com/argoproj/argocd-example-apps.git --path guestbook --dest-server https://kubernetes.default.svc --dest-namespace default**
- **argocd app get guestbook**
- **argocd app list**
- **argocd app sync guestbook**
- **kubectl get all**

Comparing OpenShift to Kubernetes

Understanding OpenShift

- OpenShift is a Kubernetes distribution!
- Expressed in main functionality, OpenShift is a Kubernetes distribution where developer options are integrated in an automated way
  - Source 2 Image
  - Pipelines (currently tech preview)
  - More developed authentication and RBAC
- OpenShift adds more operators than vanilla Kubernetes
- OpenShift adds many extensions to the Kubernetes APIs

Running Kubernetes Applications in OpenShift

Running Applications in OpenShift

- Applications can be managed like in Kubernetes
- OpenShift adds easier to use interfaces as well
  - **oc new-app --docker-image=mariadb**
  - **oc set -h**
  - **oc adm -h**
- Managing a running environment is very similar
  - **oc get all**
  - **oc logs**
  - **oc describe**
  - **oc explain**
- etc.

Building OpenShift Applications from Git Source

Understanding S2I

- S2i allows you to run an application directly from source code
- **oc new-app** allows you to work with S2i directly
- S2i connects source code to an S2i image stream builder image to create a temporary builder pod that writes an application image to the internal image registry
- Based on this custom image, a deployment is created
- S2i takes away the need for the developer to know anything about Dockerfile and related items
- S2i also allows for continuous patching as updates can be triggered using web hooks

Understanding S2i Image Stream

- The image stream is offered by the internal image repository to provide different versions of images
- Use **oc get is -n openshift** for a list
- Image streams are managed by Red Hat through OpenShift. If a new version of the image becomes available, it will automatically trigger a new build of application code
- Custom image streams can also be integrated

Understanding S2i Resources

- ImageStream: defines the interpreter needed to create the custom image
- BuildConfig: defines all that is needed to convert source code into an image (Git repo, imagestream)
- DeploymentConfig/Deployment: defines how to run the container in the cluster; contains the Pod template that refers to the custom built image
- Service: defines how the application running in the deployment is exposed

Performing the S2i process

- **oc new-app php~https://github.com/sandervanvugt/simpleapp --name=simple-app**
- **oc get is -n openshift**
- **oc get builds**: allows for monitoring the build process
- **oc get buildconfig**: shows the buildconfig used
- **oc get deployment**: shows the resulting deployment

Summary: Container Based Devops

Summary

- Kubernetes and OpenShift are awesome tools for DevOps
- The CI/CD part is filled in by working with container images that are easily updated
- The MicroServices approach is implemented by using multiple containers and connect these using variables
- The variables are easily decoupled using Kubernetes ConfigMaps and Secrets
- DevOps deployment strategies such as Blue/green and Canary deployment are easily implemented using Kubernetes
- OpenShift is adding S2I to make the CI/CD part even easier

Further Learning

Related Live Courses

- Containers:
  - Containers in 4 Hours: Docker and Podman
- Kubernetes
  - Kubernetes in 4 Hours
  - Hands-on Kubernetes Deployment in 3 Weeks
  - CKAD Crash Course
  - CKA Crash Course
  - Building Microservices with Containers, Kubernetes and Istio
- Recorded Courses
  - Getting Started with Kubernetes
  - Getting Started with Containers
  - Certified Kubernetes Application Developer