Getting Started with OpenShift

Introduction

Course Agenda

- Understanding OpenShift
- Setting up an OpenShift POC environment
- Running an application in OpenShift
- Understanding OpenShift building blocks
- Using Kubernetes Resources in OpenShift
- Using Source-to-Image to create applications
- Managing Routes
- Managing Storage

Expectations

- This course provides an introduction to OpenShift
- No knowledge of OpenShift is required, Linux, knowledge of containers and Kubernetes is useful

Related Courses

- Managing Containers on Linux
- Kubernetes in 4 Hours

Course Setup and Resources

- Sample files are available on github
- **git clone** https://github.com/sandervanvugt/openshift
- For minishift: A demo machine with the latest version of Fedora Workstation is used. If you want to follow along with the labs, start installing that now with the following specs
  - 8 GiB RAM
  - 40 GiB disk space
  - 2 vCPUs with support for virtualization
- For CRC: Use a demo-machine with CentOS 8.x, 16 GB RAM, 40GB disk space and 4 vCPUs with virtualization support

Poll Question 1

Rate your own OpenShift experience / knowledge

- none
- beginner
- intermediate
- advanced
- expert

Poll Question 2

Rate your own knowledge of Containers and Kubernetes

- none
- beginner
- intermediate
- advanced
- expert

Poll Question 3

Which of the following best describes your job title

- sysadmin
- devops
- developer
- IT support staff
- IT manager
- student
- other

Poll Question 4

Where are you from?

- North America
- Europe
- India
- Asia (not India)
- Africa
- South America
- Australia / Pacific

Getting Started with OpenShift

1. Understanding OpenShift

Understanding OpenShift

- OpenShift Container Platform (OCP) is a Red Hat platform that allows developers to easily build an environment based on source code that is inserted into the system
  - Source code can be fetched from any repository
  - Dockerfiles can be processed easily, and other sources are also supported
- The result is a container that will be orchestrated by the integrated Kubernetes layer

Understanding Pipelines

- OpenShift implements CI/CD Pipelines
- A Continuous Integration / Continuous Development Pipeline allows developers to pusblish code more frequently and more efficiently
    - In Continuous Integration, small modifications to code are inserted frequently
    - The goal of CI is to build, package and test applications
    - Continuous Delivery picks up with CI stops and make sure delivery of applications to the infrastructure is automated

Understanding Containers

- Containers are the modern-day replacement of applications that are installed on servers
- Containers run on top of a container engine, which by itself runs on a host operating system
- The Docker container engine is a common engine, but not the only one: In RHEL 8 containers can run natively on top of the RHEL operating system
- In OpenShift, containers are managed in Pods. A Pod consists of one or more containers that provide a service

Understanding Kubernetes

- OpenShift is a Kubernetes distribution
- Kubernetes brings scalability to working with containers
- It manages a cluster of hosts that run the containers
- Kubernetes gathers containers in Pods and orchestrates the Pod to run in a specific location
- Kubernetes defines specific resource types for managing the environment
- Etcd is the distributed key-value store that Kubernetes uses to store configuration and state of the Kubernetes cluster

Understanding PaaS

- Platform as a Service (PaaS) is a category of cloud computing services that allows customers to develop, run and manage applications without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app (Wikipedia)
- OpenShift is a PaaS solution, that adds different PaaS features to a Kubernetes/Docker environment
    - Remote management
    - multitenancy
    - Security
    - Monitoring
    - Application life-cycle management
    - Auditing

OpenShift and Kubernetes

- OpenShift adds features on top of Kubernetes, but uses the core Kubernetes infrastructure
- OpenShift adds resource types to the Kubernetes environment and stores these in Etcd
- Most OpenShift services are implemented as containers
- OpenShift adds xPaaS, which is the middleware services that can be offered as PaaS, by adding JBoss middleware solutions
- Some Kubernetes resources types are not available in OpenShift

Understanding Kubernetes Resource Types

OpenShift builds on top of the Kubernetes Resources

- Pods: represent a collection of one or more containers that share resources
- Services: used to connect clients to pods, by offering a single IP address and port that is offered in a round-robin fashion
- Deployments: take care of scaling, typically triggered through a replica set
- Persistent Volumes: storage (often networked) that can be accessed from within a pod
- Persistent Volume Claims: used by pods to request access to storage that meets certain criteria

Understanding OpenShift Resource Types

OpenShift builds on top of Kubernetes and adds a few resource types

- Deployment Configurations: similar to deployments in Kubernetes and used to represent a replicated set of pods
- Build Configurations: used to build a container image from application source that is stored on a Git server
- Routes: represent a DNS host name as ingress point for applications and microservices

Jenkins and OpenShift

- Jenkins is a solution to set up continuous integration or delivery environment for almost any combination of languages and source code repositories
- It is used to build and test software projects continuously, making it easier for developers to integrate changes to a project
- The result is a CI/CD pipeline for automated builds of software
- OpenShift integrates with Jenkins
- Note that Jenkins pipelines do need a plugin in OpenShift

Getting Started with OpenShift

2. Setting up a Lab Environment

Note

- Time in this class is very short to set up your own lab environment. You might as well just listen to the lectures and set up your lab environment later

Understanding OpenShift POC options

- OpenShift 3.x runs as a virtual machine in Minishift
- OpenShift 4.x is not supported in Minishift. Use Red Hat CodeReady Containers instead
    - Requires a valid RH subscription
    - Available through the Developer Subscription: developers.redhat.com

Summary: Installing Minishift on Fedora WS

- Install a VM with Fedora Workstation 31
- Enable Hypervisor extensions in the VM layer
- **sudo dnf install libvirt qemu-kvm**
- **sudo usermod –aG libvirt student**
- **sudo curl –L https://github.com/dhiltgen/docker-machine-kvm/releases/download/v0.10.0/docker-machine-driver-kvm-centos7 -o /usr/local/bin/docker-machine-driver-kvm**
- **sudo chmod +x /usr/local/bin/docker-machine-driver-kvm**

Summary: Installing Minishift on Fedora WS

- as student: **wget https://github.com/minishift/minishift/releases/download/v1.34.1/minishift-1.34.1-linux-amd64.tgz**
- **tar zxvf minishift-1.34.1-linux-amd64.tgz**
- **chmod +x minishift-1.34.1-linux-amd64/minishift**
- **sudo mv minishift-1.34.1-linux-amd64/minishift /usr/local/bin**
- Google for openshift-origin-client-tools and download and extract
- **minishift start**
- **oc status** to verify
- (see here: https://docs.okd.io/latest/minishift/getting-started/installing.html for more details)

Alternatively: Code Ready Containers

- On developers.redhat.com you can get access to CodeReady Containers, which is the all-in one Red Hat licensed version of OpenShift 4
  - CRC works on Mac, Windows and Linux
  - Recommended: use a dedicated Linux VM to avoid any conflicts
  - System requirements: 4 vCPUs, 16 GB of RAM (NOT 8!), 35 GB of storage in /home
- OKD 4 can be installed as well, but requires a recommended minimum of 96GB of RAM…
- To work with CRC, download the xz archive, as well as the pull secret from developers.redhat.com

CRC on MacOS

- **crc setup** om MacOS gives a security warning
- Use **sudo xattr -d com.apple.quarantine ~/Downloads/crc-macos-1.13.0-amd64/crc**
- Next, run **crc setup** again it will work
- When prompted for the pull secret, copy paste the entire pull secret crypto string into the installer interface
- Also consult the getting started guide for more details: https://access.redhat.com/documentation/en-us/red_hat_codeready_containers/1.13/html/getting_started_guide/index

CRC on Linux

- CentOS / Red Hat / recent Fedora is supported
- Required packages: **libvirt** and **NetworkManager**
- **crc setup** as non-root to provide initial setup
- **crc start -p pull-secret -m 12244** to start, import the pull secret as well
- **crc console** gives access to the CRC console
- **crc console --credentials** prints credentials
  - Use developer for creating OpenShift applications and project
  - Use kubeadm for creating new users, setting roles and more
- **crc oc-env** prints a command to execute to add the **oc** binary to your path: run it!
- **oc login -u developer** https://api.crc.testing:6443 will log in to the cluster

CRC Considerations

- CRC clusters need to be rebuilt every month
- Use **crc cleanup** to delete the old cluster
- Use **crc setup; crc start** to run a new cluster
- Make sure you have the pull-secret when creating a new cluster
- Use **crc console --credentials** to print current credentials
- Use **crc console** to log in on the console
  - Make sure to select **htpasswd_provider** in the password field before logging in as developer!
- Use **crc oc-env** to print the command needed to add the **oc** binary to $PATH and run the command that is printed
- Use **oc login -u developer** https://api.crc.testing:6443 to log in
- Use **oc get co** to verify the availability of operators

Getting Started with OpenShift

3. Running OpenShift

Running OpenShift

- OpenShift is available as a hosted solution in all major public cloud platforms
  - Red Hat OpenShift on AWS
  - Azure Red Hat OpenShift
  - Red Hat OpenShift on Google Cloud
  - and more
- Alternatively, OpenShift can be installed on-premise, running in your own datacenter

OpenShift 4 New Features

Currently, OpenShift 4.5 is the current version, offering the following new features

- Integrated installation in a cluster
- Support of KEDA event-drive containers
- Openshift Service Mesh to improve service availability
- RHEL CoreOS as default platform
- Simple Interface to upgrade clusters
- Cloud Automation for easy deployment in hybrid cloud
- Offers Red Hat Container Storage 4
- Knative integration to build a serverless infrastructure

OpenShift 4 New Features (2)

- Easy Deploying, Managing, and Packaging applications using Kubernetes Operators
- Better integration with Red Hat JBoss middleware
- SELinux labeling of each container with its specific context
- Auto-scaling compute nodes

Managing OpenShift

- The web console is an easy way to get started, but functionality is limited
- The **oc** command can be used from the command line and has rich functionality to work with OpenShift in imperative mode
- YAML (or JSON) files can be used as input files to define OpenShift resources in declarative way
- Using YAML with **oc** is the common way to work with OpenShift

Getting Started with OpenShift

4. Setting up an Application in OpenShift

Exploring OpenShift

- All applications are offered through a project, so start by creating a project first
- The Topology offers different sources for building applications
- In Topology, the OpenShift Catalog provides builder images to build new applications
    - It allows you to connect to specific languages, databases, source codes to build an application
- The OpenShift Projects provide the interface for managing applications
    - Existing applications are listed in the projects

Exploring OpenShift

- Make sure you have logged in as developer (verify in upper right corner and on top of the pane on the left)
- Select Topology for the different options to import applications
- Use From Catalog to use a Builder Image to use one of the images that make it easy to process your source code into a working container image

Demo: Creating an Application in the web interface

- From Catalog, select PHP, version 7.latest
- Provide a name to the application
- Specify the git repository to use
  - https://github.com/openshift/cakephp-ex.git
- Click **Create** to launch, next Close that window
- Now get to **Project**, where you can see the application being built.
- Now, select **Builds** where you can see the actual application
- Further click on the application details to explore what it is doing
- At the end of the build, an image is created and pushed to the OpenShift container registry
- Check success in the Events log
- Check routes, it contains the DNS name to get to it

Demo: Running a Container in the web interface

- To run an application from an existing image, select Topology > Container Image
- Select **Deploy Image** and provide an **image name**
- Click the search symbol to look for that image in the provided registries
- Click **Create** to deploy the application
- Alternatively, refer to a YAML file to create the application (more details later)

Demo: Creating an application with **oc new-project**

- **oc whoami**
- **oc completion**
- **oc new-project mysql**
- **oc new-app --docker-image=mysql:latest --name=mysql-openshift -e MYSQL_USER=myuser -e MYSQL_PASSWORD=password -e MYSQL_DATABASE=mydb -e MYSQL_ROOT_PASSWORD=password**
- **oc status**
- **oc get all**
- **oc get pods -o=wide**
- Log in to the webconsole and see the new application in Project: mysql > Overview

Basic Troubleshooting

- **oc get events** connects to the etcd to show recent events
- **oc logs <podname>** connects to the pod to show an application log
- **oc describe pod <podname>** connects to the etcd to show pod details
- **oc projects** will show all projects, you might be in the wrong project!

4. Understanding OpenShift Building Blocks

Understanding Projects

- A project is an isolated environment that groups resources that belong together
- A project is a Kubernetes namespace that contains all services running in the OpenShift application and works as a strictly separated environment
- Users will see their projects only
- Log in as kube_admin to see all projects (or Kubernetes namespaces)
- Type **oc config get-contexts** to see all current projects (all users) and **oc projects** to see your current projects (your account)
- After logging in, you'll see which projects you have access to
- Use **oc project myproject** to switch to a different project
- Resources will always be specific to a specific project

Understanding Namespaces

- In Linux, Namespaces implement isolation at the Linux kernel level and are available at different levels
  - Linux namespaces provide strict isolation at a container level
- A Kubernetes namespace is based on Linux namespace and is a group of isolated resources that behaves as a cluster, in OpenShift we call this a project
- Because of using namespaces, strictly isolated environments can be implemented
- In OpenShift, namespaces are addressed using Projects: use **oc project <projectname>** to switch

Understanding Pods

- An application is defined in a run-time image
- A container is a running instance of an image
- The pod is the minimal entity that is managed by OpenShift
- Pods typically run one container, and often define a volume that can be used for storage by that container also
- Don't run "naked Pods", run Deployments instead to offer features like scalability and update strategy

Deployment

To run Pods, you'll start a Deployment, as these add useful features to the Pods

- The ReplicaSet takes care of replication of pods and is a part of the Deployment
- Update strategy is also a part of the deployment
  - Rolling update
  - Recreate
  - Custom
- In OpenShift 3.x, Deployment Config was used instead of Deployment

Replicaset

- The Replicaset takes care of running multiple instances of a Pod
- Replicaset is a part of the Deployment
- Replicaset uses labels and selectors to track availability of Pods
- Every pod by default has a label
  - Manual labels can be set as well
- The Replicaset uses a selector to specify which labels should be used
  - Use **oc get pods --show-labels** to show the labels that OS has automatically added
  - Use **oc describe rc <name>** to see the current selector that is used
- Use **oc get all --selector key=value** to show all resources matching a specific label

Services and Routes

- If you look at the overview tab in OS, you see available applications, including the URL you need to access the application
  - Use the cluster IP to connect to the service
  - Or use the hostname that has been generated (in nip.io)
- On a replicated application, there's a load balancer behind that to decide which pod to connect to
- The service takes care of load balancing, and gives one identity
- The *route* is what gives a published URL, and what allows access to the application from outside the cluster
  - Route on K8s is based on the ingress controller but needs additional configuration

5. Understanding K8S Resources in an OpenShift Environment

Using YAML Files to Define Resources

- YAML is the most common way for defining resources in OpenShift
- When defining YAML files, four components are required:
  - apiVersion
  - kind
  - metadata
  - spec
- Other properties depend on the kind you want to create
- Get an overview of all available Kinds using **oc api-resources**

OpenShift and Kubernetes Resources

- Some run OpenShift for its Kubernetes functionality
- OpenShift is a Kubernetes distribution, but it adds specific resources by adding APIs
- Use **oc api-resources** to get an overview of current API objects: it will clearly show which objects are Kubernetes, and which are specific to OpenShift
- Use **oc explain** to get more information about the use of specific API objects (Kubernetes objects only)
- **oc explain** output provides useful information for writing YAML files

Creating a (failing) application using YAML files

Warning: this demo will fail

- **git clone https://github.com/dockersamples/example-voting-app**
- **cd example-voting-app**
- **cat kube-deployment.yml**
- **oc apply -f kube-deployment.yml**
- **oc get pods** - some are crashing
- Use **oc logs <podname>** to see what's happening
- Use **oc describe pod <podname>** for more details
- Check the YAML file and see what is wrong
- Tip: what is the PersistentVolume doing?

Interacting with the pod from the CLI

- **oc projects**
- **oc get pods**
- **oc get pods -o wide** # shows IP addresses
- **oc describe pod <podname>**
- **oc api-resources**
- **oc -it exec <podname> sh** / oc rsh <podname> can be used as alternative
- **oc attach <podname>** attaches to the container
- **oc logs <podname>**

Getting Started with OpenShift

6. Managing Routes

Understanding Routes

- Routes are OpenShift resources that allow for network access to pods from users and applications outside the OpenShift instance
- A route connects a public IP address and DNS name to the internal service IP address
- OpenShift uses a HAProxy based shared router service to implement the route
- Routes directly link to the service resource names and don't need any selectors
- Routes are an alternative to Kubernetes Ingress

Creating Routes

- The **oc new-app** command does NOT create routes but it does create services
  - The command cannot know if you intend to expose the route externally or not
  - Note that applications created from the web console will have a route to expose
- Use **oc expose service myapp --name myapp** to expose the service myapp as the route myapp
- Use **oc create** to create a route using a JSON or YAML resource definition file
- When exposing a service, the DNS name is auto-generated as *routename-projectname.defaultDNSdomain*
  - The subdomain is defined in the master-config.yaml file that contains the OpenShift configuration settings
- Use **oc get route** for an overview

7. Using Source-to-Image to create applications

Understanding S2I

- To create an Image, a Dockerfile could be used
- Source 2 Image (S2I) takes application source code from a source control repository (such as Git) and builds a base container based on that to run the application
  - While doing so, the image is pushed to the OpenShift registry
- Using S2I allows developers to build running containers without the need to know anything about the specific OS platform
- S2I also makes it easy to patch: after updating the application code a new image is generated

Understanding Images and Image Streams

- OpenShift works with Image Streams
- An Image Stream is a collection of different versions that exist of an image
- An Image is a runtime template that contains all data that is needed to run a container
  - This includes metadata that describes image needs and capabilities
- Images in an image stream are identified by a tag, and can be specified as such
  - image=nginx:1.14

Understanding Image Types

- *Builder Image* provide specific programming language information that is required to create an application
  - Builder images are stored in the OpenShift namespace and available to anybody
  - **oc get is –n openshift**
- The resulting *custom runtime image* is used to run your own application
  - Custom images are stored in the OpenShift internal registry, and are accessible from the current project
  - **oc get ns**

Understanding the S2I Flow

- To build an image based on source code, a builder image is required and is used as a runtime environment
  - Base builder image such as Python and Ruby are included
  - Custom images can be created
- When either the application source code or the builder image gets updated, a new container image can be created if web hooks are configured
- Applications need to be updated after a change of either the application code, or the builder image
- Applications are built against image streams, which are resources that name specific container images with image stream tags
- The base builder images may be obtained from a trusted repository, or can be self-built

Demo: Building an Application - 1

1. **oc -o yaml new-app php~https://github.com/sandervanvugt/simpleapp --name=simple1 > s2i.yaml**
2. **view s2i.yaml**
3. **oc new-app php~https://github.com/sandervanvugt/simpleapp**
4. **oc status**
5. **oc get builds**
6. **oc get pods**

Understanding BuildConfig vs. Deployment

- The BuildConfig pod creates images in OpenShift and pushes them to the internal Docker registry
  - **oc get builds** will show them
- The Deployment is responsible for deploying pods in OpenShift. To do so, it may use the images created by BuildConfig in the internal Docker registry
  - **oc get deployments** will show them

Troubleshooting: Cleaning Up

- **oc delete all -l app=simple1** will delete everything using that label
- **oc delete all --all**

Getting Started with OpenShift

7. Allocating Storage

Pearson

Understanding OpenShift Storage

- To store data, in the Pod Spec a volume needs to be created
- OpenShift uses Kubernetes persistent volumes to provide decoupling for the storage for pods
- In persistent storage, data is stored external to the Pod, so if the Pods shut down, the data is still available
- Persistent storage is typically some kind of networked storage, provided by the OpenShift administrator
- Persistent volumes are objects that exist independent of any Pod
- Developers create a persistent volume claim (PVC) that requests access to persistent storage without the need to know anything about the underlying infrastructure

Supported Volume Types

- Use **oc explain pod.spec.volumes** to get an overview of volume types
- NFS
- GlusterFS
- OpenStack Cinder
- Ceph RBD
- AWS Elastic Block Store
- GCE Persistent Disk
- Azure Disk and Azure File
- VMware vSphere
- iSCSI
- Fibre Channel

Persistent Volume Access Modes

The access mode defines how and how many nodes can access the storage

- ReadWriteOnce: a single node has read/write access
- ReadWriteMany: multiple nodes can mount the volume in read/write mode
- ReadOnlyMany: the volume can be mounted read-only by many nodes

Determining Persistent Storage Access

- The storage access type in a PVC is matched to volumes offering similar access modes
- So if the PVC needs ReadWriteOnce, it looks for a PV that offers this class (and which may offer other classes as well)
- Optionally, the PVC may request a specific storage class, using the **storageClassName** attribute. In that case, the PVC is matched to PV's that have the same storageClassName set
- The PVC is NOT connected to any specific PV in any way
- The Pod has a connection to the PersistentVolumeClaim, NOT to the Persistent Volume

Creating PVs and PVC Resources

Objects need to be created in the right order

- First, the PersistentVolumes need to be created
- Next, the PersistentVolumeClaims are created
- Finally, the Pods are configured to use a specific PVC

Using NFS for Persistent Volumes

- Mapping between containers and UIDs on an NFS Server doesn't work as container UIDs are randomly generated
- To use an NFS share as OpenShift PV, it must match the following requirements
  - Owned by the **nfsnobody** user and group
  - Permission mode set to 700
  - Exported using the **all_squash** option
  - Consider using the **async** export option for faster handling of storage requests

Step 1: Setting up the NFS Server

1. yum install nfs-utils
2. mkdir /storage
3. Useradd nfsnobody
4. chown nfsnobody.nfsnobody /storage
5. chmod 700 /storage
6. echo "/storage *(rw,async,all_squash) >> /etc/exports"
7. systemctl enable --now nfs-server
8. firewall-cmd --add-service mountd --permanent
9. firewall-cmd --add-service rpc-bind --permanent
10. Firewall-cmd --reload

Step 2: Creating the Persistent Volume

- Create a YAML file with the following content

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv
spec:
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  nfs:
    path: /data
    server: 172.17.0.1
    readOnly: false
```

Step 3: Adding the Persistent Volume

- **oc login -u system:admin**
- **oc create -f nfs-pv.yaml**
- **oc get pv | grep nfs**
- **oc describe pv nfs-pv**

Step 4: Creating a PVC

- Create a YAML file with the following content

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: nfs-pv-claim
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 100Mi
```

Step 5: Creating the PVC

- **oc create -f nfs-pvc.yaml**
- **oc describe pvc nfs-pv-claim**
- **oc get pv | grep nfs** # look for the "bound" state

Step 6: Creating the Pod

- Create a YAML file to create the pod
- Use **oc create -f nfs-pv-pod.yaml** to create the pod
- Use **oc describe pod nfs-pv-pod** to verify
- Check the Volumes section, also check Events
- Use **oc logs pod nfs-pv-pod** to check its logs

Step 6: Pod Sample code

```
kind: Pod
apiVersion: v1
metadata:
  name: nfs-pv-pod
spec:
 volumes:
   - name: nfs-pv
     persistentVolumeClaim:
       claimName: nfs-pv-claim
 containers:
   - name: nfs-client1
     image: toccoag/openshift-nginx
     ports:
       - containerPort: 8081
         name: "http-server1"
     volumeMounts:
       - mountPath: "/nfsshare"
         name: nfs-pv
     resources: {}
   - name: nfs-client2
     image: toccoag/openshift-nginx
     ports:
       - containerPort: 8082
         name: "http-server2"
     volumeMounts:
       - mountPath: "/nfsshare"
         name: nfs-pv
     resources: {}
```

Step 7: Verifying Current Configuration

- oc describe pod <podname>
- oc get pvc
- oc logs <podname>
- oc exec <podname> -it -- sh
  - mount | grep mysql
- oc logs pod/nfs-pv-pod -c nfs-client1