OPEN SHIFT DOCUMENTATION:

The core technology used as the basis of OpenShift includes Docker-based containers and orchestration via the Kubernetes system.

Given that the core of the platform is based on containers, we often refer to OpenShift as a container application platform in that it is a platform designed for the development and deployment of containers.

NOTE:

The core technology used as the basis of OpenShift includes Docker-based containers and orchestration via the Kubernetes system. Given that the core of the platform is based on containers, we often refer to OpenShift as a container application platform in that it is a platform designed for the development and deployment of containers.

PROBLEMS SOLVED:

• Load balancing a set of containers with or without session affinity

• Mounting persistent storage inside of the containers

• Placement and scheduling of containers on the infrastructure

• Rolling deployments and other operational considerations that traditional developers are not experts on.

Why Kubernetes?

While Docker gets us a portable lightweight runtime, it lacks features for supporting n-tier applications.

We searched high and low for a great orchestration and scheduling system and made an early bet on the Kubernetes project Google started.

NOTE: \*\*\*\* Red Hat and the OpenShift team were key contributors to Kubernetes because of the value we see in the project to help with the scheduling, orchestration, and running of Docker-based containers for production workloads.

OPENSHIFT

OpenShift is a layer on top of Docker and Kubernetes that makes it accessible and easy for the developer to create applications and a platform that is a dream for operators to deploy containers on for both development and production workloads.

Interacting with OpenShift

The primary grouping concept in Kubernetes is the namespace. Namespaces, as the term suggests, provide a scope for names. More specifically, namespaces provide the scope named resources that describe your application and how it should be deployed. Namespaces are also a way to divide cluster resources between multiple uses. That being said, there is no security between namespaces in Kubernetes; if you are a “user” in a Kubernetes cluster, you can see all the different namespaces and the resources defined in them.

Access is controlled through an authentication and authorization model based on users and groups. Projects in OpenShift therefore provide the walls between namespaces, ensuring that users, or applications, can only see and access what they are allowed to.

This enables multitenant use of an OpenShift cluster with access privileges determined by the identity of the user or the team they belong to. Users can be assigned to multiple groups and can inherit permissions depending on group membership. Groups often map to teams or functional units within a company, such as developers, QA, or production.

Note:

The most basic unit in OpenShift are pods. A pod is one or more containers guaranteed to be running on the same host. The containers within a pod share a unique IP address. They can communicate with each other via the “localhost” and also all share any volumes (persistent storage). The containers themselves are started from an image, which in our case is a Docker image.

When scaled up, an application will have more than one copy of itself, and each copy will have its own local state. Each copy corresponds to a different instance of a pod with the pods being managed by the replication controller. As each pod has a unique IP, we need an easy way to address the set of all pods as a whole. This is where a service comes into play. The service gets its own IP and a DNS name. When making a connection to a service, OpenShift will automatically route the connection to one of the pods associated with that service

Although a service has a DNS name, it is still only accessible within the OpenShift cluster and is not accessible externally. To make a service externally accessible, a route needs to be created. Creating a route automatically sets up haproxy or a hardwarebased router, with an externally addressable DNS name, exposing the service and load-balancing inbound traffic across the pods

Commands for command line Openshift:

Openshift login: oc login <openshift-url>

create a new project: oc new-project <projectname>

Windows: oc login https://10.2.2.2:8443 --insecure-skip-tls-verify=true (self-signed certificate for the serve)

creating a new project called insultapp with different display name:

1. oc new-project insultapp --display-name="Elizabethan Insult Application"

we need to do is create a deployment that contains our application code. We can do this with the oc new-app command while specifying the image we want to use as well as the repository we want to use for the build.

1. oc new-app wildfly:latest~https://github.com/gshipley/book-insultapp.git --name='insults'

To expose a route, issue the following command:

1. oc expose service <app-name or namespace>

To get routes:

* oc get routes --🡪 will give results

We can start a new build using the `oc tool` with the following command:

* oc start-build insults

trigger a new deployment by running the oc deploy command:

* $ oc deploy << NameSpace>> --latest

You can watch the progress as the containers are started and stopped using the oc get pods command:

* $ oc get pods –watch

To get a list of all the deployments that have been made, you can use the following:

* $ oc describe dc/insults <NameSpace or app Name>

You can roll back to this version using the oc rollback command:

* $ oc rollback <<insults -- Namespace>> --to-version=10

#12 rolled back to insults-10

POSTGRESSQL:

Adding the environment variables to DeploymentConfig instead of directly in the running pod ensures that any new pod will be started with the variables it needs to connect.

You could certainly just hard code these values in your application code but that is not a best practice! This can be done using the following command:

* oc env dc insults -e POSTGRESQL\_USER=insult -e PGPASSWORD=insult POSTGRESQL\_DATABASE=<db-name>

Import schema into to DB:

Once you have the terminal open, you can import that schema with the following command: psql -h $POSTGRESQL\_SERVICE\_HOST -p $POSTGRESQL\_SERVICE\_PORT \ -U $POSTGRESQL\_USER $POSTGRESQL\_DATABASE < <db\_name>.sql