Kubernetes is a container management technology developed in Google lab to manage containerized applications in different kind of environments such as physical, virtual, and cloud infrastructure. It is an open source system which helps in creating and managing containerization of application.

Kubernetes Registry and Docker Registry

Registries are Kubernetes secrets containing credentials used to authenticate with private Docker registries.

The word "registry" can mean two things, depending on whether it is used to refer to a Docker or Kubernetes registry:

- A Docker registry contains Docker images that you can pull in order to use them in your deployment. The registry is a stateless, scalable server side application that stores and lets you distribute Docker images.
- The Kubernetes registry is an image pull secret that your deployment uses to authenticate with a Docker registry.

Deployments use the Kubernetes registry secret to authenticate with a private Docker registry and then pull a Docker image hosted on it.

Currently, deployments pull the private registry credentials automatically only if the workload is created in the Rancher UI and not when it is created via kubectl.

Creating a Registry

Prerequisites: You must have a private registry available to use.

1. From the Global view, select the project containing the namespace(s) where you want to add a registry.
2. From the main menu, click Resources > Secrets > Registry Credentials.
3. Click Add Registry.
4. Enter a Name for the registry.

   > Note: Kubernetes classifies secrets, certificates, and registries all as secrets, and no two secrets in a project or namespace can have duplicate names. Therefore, to prevent conflicts, your registry must have a unique name among all secrets within your workspace.

5. Select a Scope for the registry. You can either make the registry available for the entire project or a single namespace.
6. Select the website that hosts your private registry. Then enter credentials that authenticate with the registry. For example, if you use DockerHub, provide your DockerHub username and password.
7. Click Save.

Result:

- Your secret is added to the project or namespace, depending on the scope you chose.
- You can view the secret in the Rancher UI from the Resources > Registries view.
- Any workload that you create in the Rancher UI will have the credentials to access the registry if the workload is within the registry's scope.

Kubectl controls the Kubernetes Cluster. It is one of the key components of Kubernetes which runs on the workstation on any machine when the setup is done. It has the capability to manage the nodes in the cluster.

Kubectl commands are used to interact and manage Kubernetes objects and the cluster. In this chapter, we will discuss a few commands used in Kubernetes via kubectl.

kubectl annotate − It updates the annotation on a resource.

$kubectl annotate [--overwrite] (-f FILENAME | TYPE NAME) KEY_1=VAL_1 ...
KEY_N = VAL_N [--resource-version = version]

For example,

kubectl annotate pods tomcat description = 'my frontend'

kubectl api-versions − It prints the supported versions of API on the cluster.

$ kubectl api-version;

kubectl apply − It has the capability to configure a resource by file or stdin.

$ kubectl apply –f <filename>

kubectl attach − This attaches things to the running container.

$ kubectl attach <pod> –c <container>
$ kubectl attach 123456-7890 -c tomcat-conatiner

kubectl autoscale − This is used to auto scale pods which are defined such as Deployment, replica set, Replication Controller.

$ kubectl autoscale (-f FILENAME | TYPE NAME | TYPE/NAME) [--min = MINPODS] --
max = MAXPODS [--cpu-percent = CPU] [flags]
$ kubectl autoscale deployment foo --min = 2 --max = 10

kubectl cluster-info − It displays the cluster Info.

$ kubectl cluster-info

kubectl cluster-info dump − It dumps relevant information regarding cluster for debugging and diagnosis.

$ kubectl cluster-info dump
$ kubectl cluster-info dump --output-directory = /path/to/cluster-state

kubectl config − Modifies the kubeconfig file.

$ kubectl config <SUBCOMMAD>
$ kubectl config –-kubeconfig <String of File name>

kubectl config current-context − It displays the current context.

$ kubectl config current-context
#deploys the current context

kubectl config delete-cluster − Deletes the specified cluster from kubeconfig.

$ kubectl config delete-cluster <Cluster Name>

kubectl config delete-context − Deletes a specified context from kubeconfig.

$ kubectl config delete-context <Context Name>

kubectl config get-clusters − Displays cluster defined in the kubeconfig.

$ kubectl config get-cluster
$ kubectl config get-cluster <Cluser Name>

kubectl config get-contexts − Describes one or many contexts.

$ kubectl config get-context <Context Name>

kubectl config set-cluster − Sets the cluster entry in Kubernetes.

$ kubectl config set-cluster NAME [--server = server] [--certificateauthority = path/to/certificate/authority] [--insecure-skip-tls-verify = true]

kubectl config set-context − Sets a context entry in kubernetes entrypoint.

$ kubectl config set-context NAME [--cluster = cluster_nickname] [--user = user_nickname] [--namespace = namespace]
$ kubectl config set-context prod –user = vipin-mishra

kubectl config set-credentials − Sets a user entry in kubeconfig.

$ kubectl config set-credentials cluster-admin --username = vipin --password = uXFGweU9l35qcif

kubectl config set − Sets an individual value in kubeconfig file.

$ kubectl config set PROPERTY_NAME PROPERTY_VALUE

kubectl config unset − It unsets a specific component in kubectl.

$ kubectl config unset PROPERTY_NAME PROPERTY_VALUE

kubectl config use-context − Sets the current context in kubectl file.

$ kubectl config use-context <Context Name>

kubectl config view

$ kubectl config view
$ kubectl config view –o jsonpath='{.users[?(@.name == "e2e")].user.password}'

kubectl cp − Copy files and directories to and from containers.

$ kubectl cp <Files from source> <Files to Destinatiion>
$ kubectl cp /tmp/foo <some-pod>:/tmp/bar -c <specific-container>

kubectl create − To create resource by filename of or stdin. To do this, JSON or YAML formats are accepted.

$ kubectl create –f <File Name>
$ cat <file name> | kubectl create –f -

In the same way, we can create multiple things as listed using the create command along with kubectl.

- deployment
- namespace
- quota
- secret docker-registry
- secret
- secret generic
- secret tls
- serviceaccount
- service clusterip
- service loadbalancer
- service nodeport

kubectl delete − Deletes resources by file name, stdin, resource and names.

$ kubectl delete –f ([-f FILENAME] | TYPE [(NAME | -l label | --all)])

kubectl describe − Describes any particular resource in kubernetes. Shows details of resource or a group of resources.

$ kubectl describe <type> <type name>
$ kubectl describe pod tomcat

kubectl drain − This is used to drain a node for maintenance purpose. It prepares the node for maintenance. This will mark the node as unavailable so that it should not be assigned with a new container which will be created.

$ kubectl drain tomcat –force

kubectl edit − It is used to end the resources on the server. This allows to directly edit a resource which one can receive via the command line tool.

$ kubectl edit <Resource/Name | File Name)
Ex.
$ kubectl edit rc/tomcat

kubectl exec − This helps to execute a command in the container.

$ kubectl exec POD <-c CONTAINER > -- COMMAND < args...>
$ kubectl exec tomcat 123-5-456 date

kubectl expose − This is used to expose the Kubernetes objects such as pod, replication controller, and service as a new Kubernetes service. This has the capability to expose it via a running container or from a yaml file.

$ kubectl expose (-f FILENAME | TYPE NAME) [--port=port] [--protocol = TCP|UDP]
[--target-port = number-or-name] [--name = name] [--external-ip = external-ip-ofservice]
[--type = type]
$ kubectl expose rc tomcat –-port=80 –target-port = 30000
$ kubectl expose –f tomcat.yaml –port = 80 –target-port =

kubectl get − This command is capable of fetching data on the cluster about the Kubernetes resources.

$ kubectl get [(-o|--output=)json|yaml|wide|custom-columns=...|custom-columnsfile=...|
go-template=...|go-template-file=...|jsonpath=...|jsonpath-file=...]
(TYPE [NAME | -l label] | TYPE/NAME ...) [flags]

For example,

$ kubectl get pod <pod name>
$ kubectl get service <Service name>

kubectl logs − They are used to get the logs of the container in a pod. Printing the logs can be defining the container name in the pod. If the POD has only one container there is no need to define its name.

$ kubectl logs [-f] [-p] POD [-c CONTAINER]
Example
$ kubectl logs tomcat.
$ kubectl logs –p –c tomcat.8

kubectl port-forward − They are used to forward one or more local port to pods.

$ kubectl port-forward POD [LOCAL_PORT:]REMOTE_PORT
[...[LOCAL_PORT_N:]REMOTE_PORT_N]
$ kubectl port-forward tomcat 3000 4000
$ kubectl port-forward tomcat 3000:5000

kubectl replace − Capable of replacing a resource by file name or stdin.

$ kubectl replace -f FILENAME
$ kubectl replace –f tomcat.yml
$ cat tomcat.yml | kubectl replace –f -

kubectl rolling-update − Performs a rolling update on a replication controller. Replaces the specified replication controller with a new replication controller by updating a POD at a time.

$ kubectl rolling-update OLD_CONTROLLER_NAME ([NEW_CONTROLLER_NAME] --
image = NEW_CONTAINER_IMAGE | -f NEW_CONTROLLER_SPEC)
$ kubectl rolling-update frontend-v1 –f freontend-v2.yaml

kubectl rollout − It is capable of managing the rollout of deployment.

$ Kubectl rollout <Sub Command>

$ kubectl rollout undo deployment/tomcat

Apart from the above, we can perform multiple tasks using the rollout such as −

- rollout history
- rollout pause
- rollout resume
- rollout status
- rollout undo

kubectl run − Run command has the capability to run an image on the Kubernetes cluster.

$ kubectl run NAME --image = image [--env = "key = value"] [--port = port] [--
replicas = replicas] [--dry-run = bool] [--overrides = inline-json] [--command] --
[COMMAND] [args...]
$ kubectl run tomcat --image = tomcat:7.0
$ kubectl run tomcat –-image = tomcat:7.0 –port = 5000

kubectl scale − It will scale the size of Kubernetes Deployments, ReplicaSet, Replication Controller, or job.

$ kubectl scale [--resource-version = version] [--current-replicas = count] --
replicas = COUNT (-f FILENAME | TYPE NAME)
$ kubectl scale –-replica = 3 rs/tomcat
$ kubectl scale –replica = 3 tomcat.yaml

kubectl set image − It updates the image of a pod template.

$ kubectl set image (-f FILENAME | TYPE NAME)
CONTAINER_NAME_1 = CONTAINER_IMAGE_1 ... CONTAINER_NAME_N =
CONTAINER_IMAGE_N
$ kubectl set image deployment/tomcat busybox = busybox ngnix = ngnix:1.9.1
$ kubectl set image deployments, rc tomcat = tomcat6.0 --all

kubectl set resources − It is used to set the content of the resource. It updates resource/limits on object with pod template.

$ kubectl set resources (-f FILENAME | TYPE NAME) ([--limits = LIMITS & --
requests = REQUESTS]
$ kubectl set resources deployment tomcat -c = tomcat --
limits = cpu = 200m,memory = 512Mi

kubectl top node − It displays CPU/Memory/Storage usage. The top command allows you to see the resource consumption for nodes.

$ kubectl top node [node Name]

JFrogArtifactory tool for Artifact Management

The JFrog Platform has been designed to provide developers and administrators with a seamless DevOps experience across all JFrog products.

Main Features and Functionality
Hybrid and Multi-Cloud Environments

You can host Artifactory on your own infrastructure, in the Cloud or use the SaaS solution providing maximum flexibility and choice.

Universal Binary Repository Manager

Artifactory offers a universal solution supporting all major package formats including Alpine, Maven, Gradle, Docker, Cargo, Conda, Conan, Debian, Go, Helm, Vagrant, YUM, P2, Ivy, NuGet, PHP, NPM, RubyGems, PyPI, Bower, CocoaPods, GitLFS, Opkg, SBT, Swift, Terraform and more. For more information, see Package Management.

Extensive Metadata

Artifactory provides full metadata for all major package formats for both artifacts and folders. These include metadata that originates with the package itself, custom metadata added by users such as searchable properties and metadata that is automatically generated by tools such as build information and more.

Artifactory as Your Kubernetes Registry

Artifactory allows you to deploy containerized microservices to the Kubernetes cluster as it serves as a universal repository manager for all your CI/CD needs, regardless of where they are running in your organization. Once you check in your App package, you can proceed to propagate and perform the build, test, promote and finally deploy to Kubernetes.

Massively Scalable

Supports a variety of enterprise-scale storage capabilities including S3 Object Storage, Google Cloud Storage, Azure Blob Storage and Filestore Sharding providing unlimited scalability, disaster recovery, and unmatched stability and reliability. Accommodates large load bursts with no compromise to performance. Increase capacity to any degree with horizontal server scalability to serve any number of concurrent users, build servers and interactions.

Replication

Artifactory’s unique set of replication capabilities ensures locality in any network topology and for any development methodology. Considering the requirements for establishing your specific distributed pipelines and collaboration, you will have several alternatives to choose from. These include both push and pull replication topologies, remote repositories, and different scheduling strategies such as on-demand, on-schedule or event-based replication. For more information, see Replicator.

High Availability

Full active/active HA solution with live failover and non-disruptive production upgrades. For more information, see High Availability.

Advanced CI Server integration with Build Tools

JFrog Artifactory supports build integration whether you are running builds on one of the common CI servers in use today, on cloud-based CI servers or standalone without a CI server. Integration of Artifactory into your build ecosystem provides important information that supports fully reproducible builds through visibility of artifacts deployed, dependencies and information on the build environment.

Artifactory provides visibility into your builds through the metadata it attaches to each artifact. In this way, you can trace your container images back to their source, so you always know what’s in your builds. For more information, see Build Integration.

Custom API-Driven Automation

Artifactory exposes an extensive REST API that provides access to its features anywhere in the development cycle. Through the API you can manage builds, repositories and artifacts, you can perform searches, apply configurations, perform maintenance tasks and more.

Helm Charts

Helm is widely known as "the package manager for Kubernetes". Although it presents itself like this, its scope goes way beyond that of a simple package manager. However, let's start at the beginning.

Helm is an open-source project which was originally created by DeisLabs and donated to CNCF, which now maintains it. The original goal of Helm was to provide users with a better way to manage all the Kubernetes YAML files we create on Kubernetes projects.

The path Helm took to solve this issue was to create Helm Charts. Each chart is a bundle with one or more Kubernetes manifests – a chart can have child charts and dependent charts as well.

This means that Helm installs the whole dependency tree of a project if you run the install command for the top-level chart. You just a single command to install your entire application, instead of listing the files to install via kubectl.

Charts allow you to version your manifest files too, just like we do with Node.js or any other package. This lets you install specific chart versions, which means keeping specific configurations for your infrastructure in the form of code.
Helm also keeps a release history of all deployed charts, so you can go back to a previous release if something went wrong.

Helm supports Kubernetes natively, which means you don't have to write any complex syntax files or anything to start using Helm. Just drop your template files into a new chart and you're good to go.

But why should we use it? Managing application manifests can be easily done with a few combinations of commands.

Why Should You Use Helm?

Helm really shines where Kubernetes didn't go. For instance, templating. The scope of the Kubernetes project is to deal with your containers for you, not your template files.

This makes it overly difficult to create truly generic files to be used across a large team or a large organization with many different parameters that need to be set for each file.