Working with PODS and Kubectl Commands

- Create out first pod with kubectl
- Basic Kubectl Commands
- Inspecting Kubernetes Objects using kubectl
- Imperative vs Declarative Commands
- YAML Syntax
- Name and Metadata
- Labels and Label Selectors
- Annotations
- Kubernetes Namespace

Working with Kubernetes Pods

A Kubernetes Pod is a group of one or more Containers, tied together for the purposes of administration and networking.

To get existing list of Kubernetes Object including Deployment, Replicaset, Pods and Services.

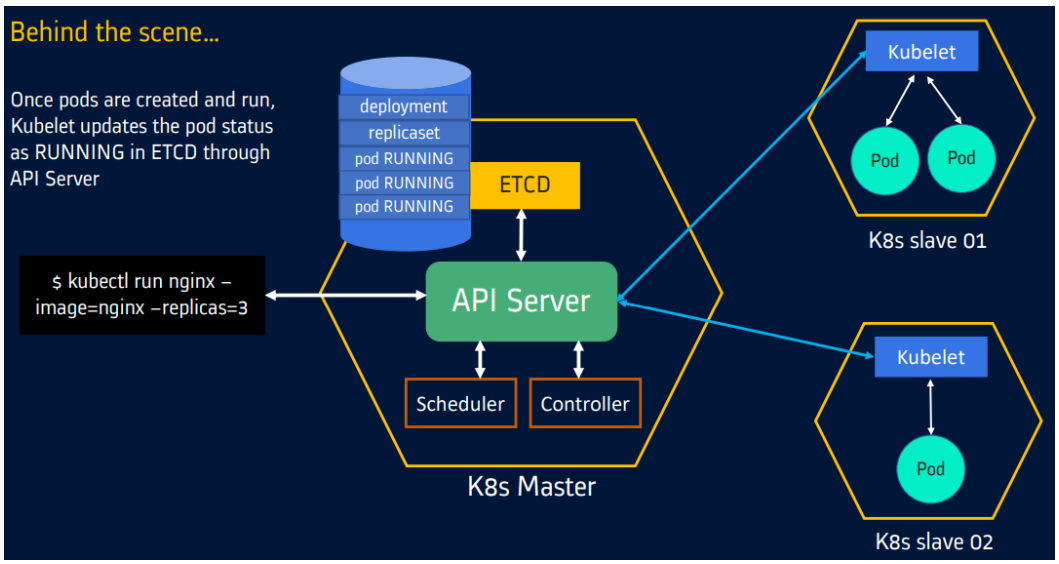
```
kubectl get all
```

Creation of [Naked] Pod:

```
kubectl run mynginx --image nginx --port 80
```

Note that mynginx is pod name

Pod States: **Pending -> Creating -> Running**



View an object

There are several commands for printing information about an object:

- **get: Prints basic information about matching objects. Use get -h to see a list of options.**

- **describe**: Prints aggregated detailed information about matching objects.
- **logs**: Prints the stdout and stderr for a container running in a Pod

To get the list of all pods

```
kubectl get pods
kubectl get pod/mynginx -o wide
kubectl get pod mynginx -o wide
```

Note: Use the -o wide option to see the internal IP assigned to the pod, as well as NOMINATED NODE

Viewing Log of a particular Pod creation

```
kubectl logs mynginx -c mynginx
```

To get the current state of the Pods (similar to docker inspect and also includes events)

```
kubectl describe pod/mynginx
```

Accessing the NGINX server in container:

Start a New Shell Process in Kubernetes Pod container.

```
kubectl exec -it mynginx -c mynginx -- sh
# curl localhost
# exit
```

Accessing the NGINX server using IP address of Pod from container of another pod

```
kubectl get pods -o wide #note the IP address of Pod
kubectl run -it myclient --image=ubuntu -- bash
# apt update
# apt install curl
# curl http://<IPOfPod>
# exit
```

Accessing the NGINX server from host machine:

Syntax: kubectl **port-forward** pod <PODName> <LOCALPORT>:<CONTAINERPORT>

```
kubectl port-forward mynginx 8080:80
```

In New Terminal Window:

```
curl http://localhost:8080
```

To delete pod

```
kubectl delete pod/mynginx
```

Declarative Commands

Imperatively

- Involves using any of the verb-based commands like kubectl run, kubectl create, kubectl expose, kubectl delete, kubectl scale and kubectl edit
- Suitable for testing and interactive experimentation.

Declaratively

- Objects are written in YAML files and deployed using kubectl create or kubectl apply
- Best suited for production environments

YAML Document:

```
apiVersion: v1
kind: Pod
metadata:
 name: mynginx
 labels:
  app: web
spec:
 containers:
 - name: nginx-con
  image: nginx
  ports:
  - containerPort: 80
```

Required Fields

In the .yaml file, for the Kubernetes object we want to create, we need to set values for the following fields (also known as **root keys**).

- **apiVersion** - Which version of the Kubernetes API you're using to create this object
- **kind** - What kind of object you want to create
- **metadata** - Data that helps uniquely identify the object, including a name string, UID, and optional namespace
- **spec** - What state we desire for the object

```
kubectl apply -f pod.yaml #Declarative object configuration
```

Using YAML for K8s definitions gives you a number of advantages, including:

- **Convenience:** You'll no longer have to add all of your parameters to the command line
- **Maintenance:** YAML files can be added to source control, so you can track changes.
- **Flexibility:** You'll be able to create much more complex structures using YAML than you can on the command line

To get YAML from exisiting objects (detailed live configuration):

```
kubectl get pod/mynginx -o yaml
```

You can also use Dry Run to generate YAML

```
kubectl run mynginx1 --image=nginx -o yaml --dry-run=server > pod-server.yaml
kubectl run mynginx1 --image=nginx -o yaml --dry-run=client > pod-client.yaml
```

Change and apply configuration file:

In pod.yaml: update Image to **nginx:1.16.1**

```
kubectl apply -f pod.yaml -f pod2.yaml -f pod3.yaml
```

Create the objects defined in a configuration files in folder

```
kubectl apply -f <folder-name>/ -R
```

Add the **-R** flag to recursively process directories.

To Get the difference in Deployed Object and New YAML

```
kubectl diff -f pod.yml
```

Note: Install DiffUtils for Windows on local machine and restarting the machine everything works.

To get properties of Kubernetes objects

- kubectl **explain** pods
- kubectl **explain** pods.spec
- kubectl **explain** pods.spec.containers.image

Kubectl create and replace:

- **kubectl create**: Will create the resource only if its not existing.
- **kubectl replace**: Replaces a existing resource. If replacing an existing resource, the complete resource spec must be provided.

Try the following commands:

```
kubectl create -f pod.yaml
kubectl edit pod mynginx
# OR
kubectl get pod mynginx -o yaml > pod-existing-state.yaml
# Update the pod image or Label
kubectl replace -f pod-existing-state.yaml
```

Note: Replace works only on Server Side YAML.

Delete the objects defined in two configuration files:

```
kubectl delete -f pod.yaml
```

Labels and Selectors

- *Labels* are **key/value pairs** that are attached to objects, such as pods.
- Labels are intended to be used to specify **identifying attributes** of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system.
- If labels are not mentioned while deploying k8s objects using imperative commands, the label is auto set as **app: <object-name>**
- Labels can be used to organize and to select subsets of objects. Labels can be attached to objects at creation time and subsequently added and modified at any time. Each object can have a set of key/value labels defined.
- Keys can be 63 chars or less and Values can be 253 chars of less.

```
apiVersion: v1
kind: Pod
metadata:
 name: mynginx
 labels:
  environment: production
  tier: frontend
spec:
 containers:
 - name: nginx
  image: nginx:1.14.2
  ports:
  - containerPort: 80
```

kubectl apply -f pod.yaml

kubectl get pod --show-labels

To set labels for a Pod that has two labels **environment: production and tier: frontend**

```
kubectl label pod mynginx environment=production tier=fontend
kubectl get pod --show-labels
kubectl label pod mynginx tier=backend --overwrite
kubectl get pod --show-labels
kubectl label pod mynginx tier-
kubectl get pod --show-labels
```

Label Selectors

- Unlike names and UIDs, labels do not provide uniqueness.
- In general, we expect many objects to carry the same label(s).
- Via a ***label selector***, the client/user can identify a set of objects. The label selector is the core grouping primitive in Kubernetes.

The API currently supports two types of selectors: ***equality-based*** **and** ***set-based***

1. ***Equality-*** **or** ***inequality-based*** *(=, !=)* requirements allow filtering by label keys and values. Matching objects must satisfy all of the specified label constraints
2. ***Set-based*** label requirements allow filtering keys according to a set of values. Three kinds of operators are supported: in,notin and exists (only the key identifier).

```
tier in (frontend, backend)
environment notin (production, qa)
partition
!partition
```

Filtering based on labels:

```
kubectl get pods --show-labels
kubectl get pods --selector environment=production,tier=frontend
kubectl get pods -l environment=production,tier!=backend
kubectl get pods -l "environment in (production, development),tier in (frontend, backend)"
```

Some Kubernetes objects, such as services and replicationcontrollers, also use label selectors to specify sets of other resources, such as pods.

```
selector:
  component: redis
```

Newer resources, such as Deployment, ReplicaSet, Job and DaemonSet, support *set-based* requirements as well.

```
selector:
  matchLabels:
    component: redis
OR
selector:
  matchExpressions:
    - {key: tier, operator: In, values: [frontend, backend]}
    - {key: environment, operator: NotIn, values: [production, staging]}
```

Annotations

- Used to add additional information about your cluster resources.
- Mostly used by people or tooling to make decisions.
- Saves you from having to write integrations to retrieve data from external data sources.
- Non-hierarchical, key/value pair
- Can't be used to query/select Pods or other resources
- Data is used for "other" purposes
- Keys can be up to 63 characters Values can be up to 256 chars

```
kubectl annotate pod nginx-pod owner=Sandeep age=47
kubectl annotate pod nginx-pod owner=Sandeep Soni --overwrite
```

OR

```
metadata:
  name: nginx-pod
  annotations:
    owner: sandeep
    age: 47
```

Kubernetes Namespaces

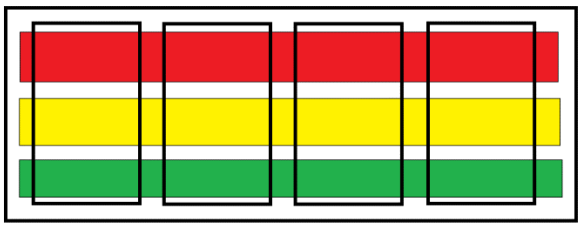
Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.

Namespaces provide a scope for names. Name of a resource has to be unique within a namespace, but not across namespaces.

Provide a boundary for security and resource control.

Namespaces **can not be nested** inside one another

Kubernetes resource can only be in one namespace.



When to use:

- Namespaces are intended for use in environments with many users spread across multiple teams or projects.
- We create new namespaces when we need to add new features to the cluster e.g. **dashboard**, **ingress**.

Note: In future versions of Kubernetes, objects in the same namespace will have the same access control policies by default.

It is not necessary to use multiple namespaces just to separate slightly different resources, such as different versions of the same software. We can use labels to distinguish resources within the same namespace.

Listing the namespaces in a cluster:

```
kubectl get namespace
```

Kubernetes starts with four initial namespaces:

- **default**: The default namespace for objects with no other namespace
- **kube-system**: The namespace for objects created by the Kubernetes system
- **kube-public**: This namespace is created automatically and is **readable by all users** (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster.
- **kube-node-lease**: This namespace for the lease objects associated with each node which improves the performance of the node heartbeats as the cluster scales.

To get list of specified namspace

```
kubectl get all -n kube-system
kubectl get pods -n non-existing-namespace #Doesn't report error
kubectl get pods --all-namespaces
kubectl get pods -A
```

Setting the namespace for a request

To set the namespace for a request, we use the --namespace flag.

```
kubectl create namespace demo-namespace
kubectl run nginx --image=nginx --namespace=demo-namespace
kubectl get pods -n demo-namespace
```

We can explicitly create a namespace and deploy resources to it using namespace field in manifest or --namespace arg on kubectl

To set namespace in YAML file

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mynginx
  namespace: demo-namespace
  labels:
    environment: production
    tier: frontend
spec:
  containers:
  - name: nginx
    image: nginx:1.18.0
    ports:
    - containerPort: 80
```

kubectl create namespace **demo-namespace**

kubectl apply -f pod.yaml - # Use this if YAML has namespace mentioned

OR

kubectl apply -f pod.yaml -n demo-namespace – Use this if YAML doesn't have mention of namespace.

Usecase of Namespace for Creating App in Different Environments:

```
apiVersion: v1
kind: Pod
metadata:
  name: mynginx
  labels:
    tier: frontend
spec:
  containers:
  - name: nginx
    image: nginx:1.18.0
    ports:
    - containerPort: 80
```

kubectl create namespace **development**

kubectl create namespace **testing**

kubectl create namespace **production**

kubectl apply -f pod.yaml -n **development**

kubectl apply -f pod.yaml -n **testing**

kubectl apply -f pod.yaml -n **production**

To set the default namespace:

We can permanently save the namespace for all subsequent kubectl commands in that context.

```
kubectl config set-context --current --namespace=demo-namespace
kubectl get all #Lists all pods under demo-namespace
kubectl config view --minify
kubectl config set-context --current --namespace=default
```

API Groups

Core	Named API Groups
Pod	apps - **Deployment**
Node	storage.k8s.io - **StorageClass**
Namespace	rbac.authorization.k8s.io - **Role**

Services PersistentVolume PersistentVolumeClaim	

To List API Resources from the API Server.

kubectl api-resources

kubectl api-resources **--api-group**=apps

kubectl api-versions

To see which Kubernetes resources are and aren't in a namespace:

kubectl api-resources --namespaced=true

kubectl api-resources --namespaced=false

API Resource Location (API Paths) Core API (Legacy)

- Cluster-scoped resources: http://apiserver:port/api/$VERSION/$RESOURCE_TYPE/$RESOURCE_NAME
- Namespace-scoped resources: http://apiserver:port/api/$VERSION/**namespaces/$NAMESPACE**/$RESOURCE_TYPE/$RESOURCE_NAME
- https://kubernetes.docker.internal:6443/api/v1/namespaces/default/pods

API Groups

- Cluster-scoped resources: http://apiserver:port/apis/$GROUPNAME/$VERSION/$RESOURCE_TYPE/$RESOURCE_NAME
- Namespace-scoped resources: http://apiserver:port/apis/$GROUPNAME/$VERSION/**namespaces/$NAMESPACE**/$RESOURCE_TYPE/$RESOURCE_NAME
- https://kubernetes.docker.internal:6443/apis/apps/v1/namespaces/default/deployments

Examples:

If you add -v 7 or -v 6 to the command, you get verbose logs that show you all the **API requests**

kubectl get deployments -v 6

https://kubernetes.docker.internal:6443/apis/apps/v1/namespaces/default/deployments/depname