KUBERNETES OBJECTS

- Kubernetes objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster.
- Specifically, they can describe:
  - What containerized applications are running (and on which nodes)
  - The resources available to those applications
  - The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance.

OBJECT SPEC AND STATUS:

- Every Kubernetes object includes **two nested object fields** that govern the object's configuration: **object spec & object status.**
- For **objects** that have **a spec,** you have to set this when you create the object, providing a description of the characteristics you want the resource to have: its desired state.
- The **status** describes the current state of the object, supplied and updated by the Kubernetes system and its components.

DESCRIBING A KUBERNETES OBJECT:

- When you use the Kubernetes API to create the object (either directly or via kubectl), that API request must include that information as JSON in the request body.
- Most often, you provide the information to kubectl in a **.yaml** file. kubectl converts the information to JSON when making the API request.

REQUIRED FIELDS:

- In the .yaml file for the Kubernetes object you want to create, you'll need to set values for the following fields:
  - **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 you desire for the object

$kubectl apply -f https://k8s.io/examples/application/deployment.yaml

IMPERATIVE COMMANDS:

- When using imperative commands, a user operates directly on live objects in a cluster. The user provides operations to the kubectl command as arguments or flags.

Run an instance of the nginx container by creating a Deployment object:

$kubectl create deployment nginx --image nginx

Delete the objects defined in two configuration files:

$kubectl delete -f nginx.yaml -f redis.yaml

Update the objects defined in a configuration file by overwriting the live configuration:

$kubectl replace -f nginx.yaml

OBJECT NAMES AND IDS:

- Each object in your cluster has a Name that is unique for that type of resource.
- Every Kubernetes object also has a UID that is unique across your whole cluster.

NAMES:

- A client-provided string that refers to an object in a resource URL, such as /api/v1/pods/somename.
- Four types of commonly used name constraints for resources.

DNS SUBDOMAIN NAMES:

- Most resource types require a name that can be used as a DNS subdomain name as defined in RFC 1123.
- This means the name must:
  - contain no more than 253 characters
  - contain only lowercase alphanumeric characters, '-' or '.'
  - start with an alphanumeric character
  - end with an alphanumeric character

RFC 1123 LABEL NAMES:

- Some resource types require their names to follow the DNS label standard as defined in RFC 1123.
- This means the name must:
  - contain at most 63 characters
  - contain only lowercase alphanumeric characters or '-'
  - start with an alphanumeric character
  - end with an alphanumeric character

RFC 1035 LABEL NAMES:

- Some resource types require their names to follow the DNS label standard as defined in RFC 1035.
- This means the name must:
  - contain at most 63 characters
  - contain only lowercase alphanumeric characters or '-'
  - start with an alphabetic character
  - end with an alphanumeric character

PATH SEGMENT NAMES:

- Some resource types require their names to be able to be safely encoded as a path segment. In other words, the name may not be "." or ".." and the name may not contain "/" or "%".

Manifest for a Pod named nginx-demo:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-demo
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

NAMESPACES:

- In Kubernetes, namespaces provide a mechanism for isolating groups of resources within a single cluster.
- Names of resources need to be unique within a namespace, but not across namespaces.
- Namespace-based scoping is applicable only for namespace objects (e.g., Deployments, Services, etc) and not for cluster-wide objects (e.g., StorageClass, Nodes, PersistentVolumes, etc).
- Namespaces cannot be nested inside one another and each Kubernetes resource can only be in one namespace.
- Namespaces are a way to divide cluster resources between multiple users.

KUBERNETES 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. 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 holds Lease objects associated with each node. Node leases allow the kubelet to send heartbeats so that the control plane can detect node failure.

List the current namespaces in a cluster using:

$kubectl get namespace

Create a new Namespace:

$kubectl create namespace dev-ns

$kubectl get namespace

To change namespace:

#kubectl config set-context --current --namespace=dev-ns

To set the namespace for a current request, use the --namespace flag:

$kubectl run nginx --image=nginx --namespace=<insert-namespace-here>

$kubectl get pods --namespace=<insert-namespace-name-here>

You can permanently save the namespace for all subsequent kubectl commands in that context:

$kubectl config set-context --current --namespace=<insert-namespace-here>

$kubectl config view --minify | grep namespace:

To list all namespaces pods:

$kubectl get all --all-namespaces (or) $kubectl get all -A

To list specific namespaes:

#kubectl get all -n dev-ns [to get pods in dev-namespace]

To change namespace:

#kubectl config set-context --current --namespace=dev-ns

#kubectl get pods

To delete a name space:

#kubectl delete namespaces dev-ns

#kubectl get namespaces

LABELS AND SELECTORS:

LABELS:

- Labels are key/value pairs that are attached to objects, such as pods.
- It is used to specify identifying attributes of objects that are meaningful and relevant.
- Labels can be used to organize and to select subsets of objects.

```
"metadata": {
  "labels": {
    "key1" : "value1",
    "key2" : "value2"
  }
}
```

Example Labels:

```
"release" : "stable" , "release" : "canary"

"environment" : "dev" , "environment" : "qa" , "environment" : "production"

"tier" : "frontend" , "tier" : "backend" , "tier" : "cache"

"partition" : "customerA" , "partition" : "customerB"

"track" : "daily" , "track" : "weekly"
```

Configuration file for a Pod that has two labels environment: production and app: nginx:

```
apiVersion: v1
kind: Pod
metadata:
  name: label-demo
  labels:
    environment: production
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

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.
- It supports two types of selectors: **equality-based and set-based.**

EQUALITY-BASED REQUIREMENT:

- Equality- or inequality-based requirements allow filtering by label keys and values. Matching objects must satisfy all of the specified label constraints, though they may have additional labels as well.
- Three kinds of operators are admitted **=, ==, !=** .

```
environment = production
tier != frontend
```

$kubectl get po

$kubectl get pods --show-labels

$kubectl get pods -l environment=production

$kubectl get pods -l 'environment in (production)'

$kubectl get pods -l 'environment in (production, development)'

$kubectl get pods -l environment!=production

SET-BASED REQUIREMENT:

- 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).

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

$kubectl get pods -l environment=production,tier=frontend
$kubectl get pods -l 'environment in (production),tier in (frontend)'
$kubectl get pods -l 'environment in (production, qa)'
$kubectl get pods -l 'environment,environment notin (frontend)'

ANNOTATIONS:

- Annotations are also key-value pairs for connecting non-identifying metadata with objects.
- These are not used to identify and select objects.

```
"metadata": {
  "annotations": {
    "key1" : "value1",
    "key2" : "value2"
  }
}
```

NOTE:

- The keys and the values in the map must be strings. In other words, you cannot use numeric, boolean, list or other types for either the keys or the values

  The configuration file for a Pod that has the annotation imageregistry: https://hub.docker.com/:

```
apiVersion: v1
kind: Pod
metadata:
  name: annotations-demo
  annotations:
    imageregistry: "https://hub.docker.com/"
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

$kubectl get po

$kubectl describe pod annotations-demo