KodeKloud Certified Kubernetes Application Developer (CKAD) Notes

horizontal line

# 1.0 - Introduction

* The exam associated with this course is entirely practical
* Roughly 19 questions in 2 hours
* To pass, need to show an understanding of configurations within kubernetes and how to fix any issues that may arise with them
* Kubernetes is supported on any cloud platform and supports hosting advanced and complex applications for any kind of architecture
* For the exam, you are allowed one browser tab to be open, which is for accessing the Kubernetes documentation only

# 2.0 - Core Concepts

## 2.1 - Recap: Kubernetes Architecture

Node:

A physical or virtual machine upon which Kubernetes is installed

Classed as “worker” machines; which can be used to run containers

Note: Nodes can also be classed as “minions”

To avoid applications failing when running on a node, it’s recommended to have multiple nodes running the same app simultaneously

If one node goes down, the app is still available to general users

Cluster:

A set of nodes grouped together, if one fails, the others are still accessible

The use of clusters allows greater load balancing

The Master Node:

A node configured as a “master” over the worker nodes.

Watches over the worker nodes and orchestrates containers running within the cluster

Other responsibilities include:

* Cluster management
* Storage of information regarding cluster members
* Monitor node status
* Manage node workload

Master Node Components:

1. API Server
   1. Acts as the frontend for Kubernetes
   2. Uses, management devices and CLI all interact with the API Server to interact with the cluster
2. etcd server
   1. Key store service
   2. Stores all data used to manage the cluster
   3. Responsible for implementing logs within the cluster; avoiding master-master conflict
3. Scheduler
   1. Distributes work or containers across nodes
   2. Looks for newly created containers and assigns them to nodes
4. Controller
   1. The main orchestrators
   2. Responsible for noticing and responding to object failures within the cluster
5. Container Runtime
   1. The underlying software used to run containers
6. Kubectl
   1. The agent running on each node
   2. Responsible for ensuring containers run on nodes as expected

Master vs Worker Nodes

Worker Nodes host containers, requiring a container runtime software, usually docker

The master node has the Kube API server running on it

Worker nodes have the kubelet agent running to facilitate API server interactions

The key value store is found on the master node, running via the etcd framework

Controller and scheduler also found on the master node; amongst other components

| Master Node | Worker Node |
| --- | --- |
| Kube API Server | Kubelet agent |
| Etcd key value store | Container Runtime |
| Controller |  |
| Scheduler |  |

Kubectl:

Kube Control

Tool used to deploy and manage apps on a kubernetes cluster

Common commands:

Commands:

| Command | Operation |
| --- | --- |
| Kubectl run hello-minikube | Deploys application onto the cluster |
| Kubectl cluster-info | View info regarding the cluster |
| Kubectl get nodes | Display nodes in the cluster |

## 2.2 - Recap: Pods

Containers aren’t deployed directly to the cluster, they are instead encapsulated in pods

A pod is a single instance of the application; said application must be containerised pre-deployment

Suppose a containerised app is running on a single pod in a single node, what happens when the user demand increases?

Cannot have multiple versions of the same container to a pod

A new pod must be created with a new instance of the application

If the user demand increases further but no more pods can be created on the current node, a new node must be created to run the additional instances

In general, pods and containers have a 1-to-1 relationship

Multi-Container Pods:

A single pod CAN have more than 1 container, but they cannot run the same application

There may be a “helper” or “Init” container running alongside the application container; running support processes for the main application; e.g. process user data

When the pod’s created or a new pod is created, the helper container is created alongside

The app and helper container communicate and share resources as they share the same network

Helper containers have a 1 to 1 relationship with the application they’re supporting

Commands:

| Command | Operation |
| --- | --- |
| Kubectl run <container\_name> | Run a container by creating a pod  Add --image= flag to specify container image  Image pulled from Docker Hub |
| Kubectl get pods | List all pods in the current namespace |

## 2.3 - Recap: Pods with YAML

Kubernetes uses YAML files as input for object creation e.g. pods, deployments and services

Any YAML File always contain 4 key fields:

1. apiVersion
   1. The version of the Kubernetes API to be used to create the object
   2. Correct api version must be specified for objects

| Kind | ApiVersion |
| --- | --- |
| Pod | v1 |
| Service | v1 |
| ReplicaSet | apps/v1 |
| Deployment | apps/v1 |

1. Kind
   1. The type of object being created
2. Metadata
   1. Data referring to the specifics of the object being created
   2. Expressed as a dictionary
   3. Labels: a child of metadata
   4. Indents indicate what metadata is related via parent/child links
   5. Use labels for pod differentiation via Key Value Pairs
3. Spec
   1. Specification of the object
   2. Contains additional configuration information for the object
   3. A dictionary
   4. ‘-’ denotes the first item in a dictionary

When creating a pod from a yaml file, run the following:



To view pods:



To view detailed information about a particular pod:



## 2.4 - Creating a Pod with YAML: Demo

Create a pod YAML file can be done with any valid text editor, as long as the file is saved with the .yaml extension.



In the directory containing the file, the object described by the yaml file can be created:



To verify the container has created, run:



## 2.5 - Practice Test and Solutions

1. How many pods exist on the system
   1. Kubectl get pods
2. Create a new pod with the nginx image
   1. Kubectl run nginx --image=nginx
3. How many pods exist now?
   1. Kubectl get pods
4. What is the image used to create the pod?
   1. Kubectl describe pod <pod-name>
5. What nodes are pods on?
   1. Kubectl get pods -o wide
6. How many containers are part of the pod ‘web app’
   1. Kubectl get pods
   2. Identify requested pod
   3. Check “ready” column
   4. Note: could also use describe command
7. What images are used in the new pod?
   1. Kubectl describe pod <pod-name>
8. What is the state of the container?
   1. Kubectl describe pods <pod-name>
9. Why is the pod in that state?
   1. Reason detailed in the state description
   2. Further details in the events
10. What does the “Ready” column in the output of “kubectl get pods” indicate?
    1. 2 numbers x/y
    2. X = total number of containers ready
    3. Y = total number of containers in pod
11. Delete pod <pod-name>
    1. Kubectl delete pod <pod-name>
12. Create a pod with name <pod-name>, image <image-name> with a yaml file
    1. 
    2. Edit as required and run kubectl create -f <pod-name>.yaml
13. Fix the image to redis: update the pod and use kubectl apply or edit to change the image
    1. Kubectl edit pod <pod-name> -> edit pod as appropriate
    2. OR kubectl set image <kind>/<object-name> <container-name>=<new image>

## 2.6 - Editing Existing Pods

If given a pod definition file, edit it and use it to create a new pod

If not given a pod definition file, can extract it:



Edit as necessary, then delete and recreate the pod

Or use kubectl edit pod <pod-name> and edit the properties required

## 2.7 - Recap: ReplicaSets

Controllers monitor kubernetes objects and respond accordingly

One of the key controllers is the replication controller

Consider a single pod running on an application. If this pod crashes, the app becomes inaccessible. To prevent this, would like to have multiple instances of the same app running simultaneously.

The replication controller allows the running of multiple instances of the same pod in the cluster, leading to high availability.

Note: Even you have a single pod, the replication controller can automatically replace the single pod in the event of failure

Load Balancing and Scaling:

The replication controller exists to create multiple pods and share the load across it

Consider a single pod serving a single user:

If a new user wants to access the service, the replication controller automatically deploys an additional pod(s) to balance the load

If demand exceeds node space, can spread new pods across other nodes within the cluster; this is done automatically by the RC.

Therefore, can see the Replication Controller spans multiple nodes. This helps balance the load across multiple pods on different nodes; allowing scalability of applications.

In terms of a replication controller, 2 names are considered:

1. ReplicationController
2. ReplicaSet

Both types are essentially the same, just use different containers. ReplicaSets is the newer technology for the role of Replication Controller.

To create a Replication Controller, write a yaml in the following format:



The above YAML file now contains two nested definitions, one for the replication controller, the other for the pod to be replicated.

To create the replication controller, run the kubectl create -f command as normal.

To view ReplicationControllers:



To view the pods, use the kubectl get pods command as usual.

ReplicaSet:



Selector helps ReplicaSet define what pods should be replicated. This is needed as replicaset can also manage pods not created alongside the replicaset; as long as their label matches the selector.

The inclusion of the selector is the main difference.

If skipped, the selector assumes the same label provided in the definition file’s metadata.

Labels & Selectors:

Consider a deployment of an application with 3 pods. To create a replication controller or set will ensure that at any given point, 3 pods will be running.

If the pods weren’t created, the replica set will automatically create them and monitor for failures; deploying replacements in the event of failure.

The replicaset knows what pods to monitor via the labels defined in the yaml file. In the matchLabels set, the label entered denotes the pods to be managed by the ReplicaSet.

The template section is required such that the pods can be redeployed in with the correct configuration.

Scaling:

To scale a Kubernetes deployment, can update number in the yaml file and run:



Alternatively:



Or:



Note: the final method will NOT update the yaml file.

| Command | Description |
| --- | --- |
| Kubectl create -f definition.yaml | Creates a replicaset or any object in kubernetes |
| Kubectl get replicasets | List replicasets |
| Kubectl delete replicaset <replicaset-name> | Delete replicaset and underlying pods |
| Kubectl replace -f replicaset-definition.yaml | Replace or update the replicaset |
| Kubectl scale --replicas=x -f replicasdefinition.yaml | Scale the deployment to change the object’s replicas in the yaml file |

## 2.8 - ReplicaSets Test and Solutions

1. How many pods in the current namespace?
   1. Kubectl get pods
2. How many replicasets?
   1. Kubectl get replicasets
3. How about now?
   1. “ “
4. How many desired pods?
   1. “ “ and analyse output
5. What’s the image used in the pods?
   1. Kubectl describe replicaset <replicaset name>
   2. Look in the pod template
6. How many ready pods?
   1. Kubectl get replicasets
   2. Analyse outputs
7. Why are the pods not ready?
   1. Kubectl describe replicaset
      1. Analyse status and events
   2. Kubectl get pods
      1. Analyse outputs
      2. Describes individual pod and analyse events
      3. Note the image doesn’t exist
8. Delete any of the pods
   1. Kubectl delete pod <pod-name>
9. How many pods exist now?
   1. Kubectl get pods
10. Why couldn’t the pod get deleted?
    1. Replicaset ensures the desired number of pods are always available
11. Create a replicaset using a yaml file
    1. Kubectl create -f <def.yaml>
    2. Failure> -> Open yaml file and troubleshoot
    3. Run create -f command again
12. Fix the issue and create a replicaset
    1. Attempt creation
    2. Note: Invalid value, selector doesn’t match template labels
    3. Make selector matchLabel match that of template
    4. Kubectl create -f <deployment>.yaml
13. Delete the replicaset
    1. Kubectl delete replicaset <replicaset-name>
14. Fix the original replicaset to use a certain image:

## 2.9 - Recap: Deployments

* When deploying an application in a production environment, such as a web server, the following requirements need to be met:
  + Many instances need to be running simultaneously (meets user demand)
  + Need to be able to upgrade instances seamlessly one-after-another i.e. rolling updates
  + Need to be able to roll back or undo any changes that lead to a fault to the last working iteration
  + In the event you want to make multiple changes to the environment, such as upgrading WebServer versions, modifying resource allocation etc, need to pause the environment to make the changes, then resume them to apply all the changes together
* All these requirements can be met by using Kubernetes deployments
* **Deployments:**
  + Kubernetes objects higher in the hierarchy than ReplicaSets
  + Provides abilities to:
    - Upgrade underlying instances seamlessly
    - Utilise rolling update capabilities
    - Rollback changes in the event of failures
    - Pause and resume environments to allow changes to be made/take place
* To create a deployment, you need a definition file:



* Note: This is very similar to how a ReplicaSet’s definition file looks, it’s only the Kind value that is different.

| Command | Description |
| --- | --- |
| Kubectl create -f <deployment-definition>.yaml | Create Kubernetes deployment from yaml file |
| Kubectl get deployments | View deployments in Kubernetes.  Note: when deployments are created, associated replicasets and pods are automatically created, and can be viewed using similar kubectl get commands |
| Kubectl get all | Display all kubernetes objects within the cluster |

## 2.10 - Tips for Good Practice

* When using the CLI, it can become difficult to create and edit the YAML files associated with objects in Kubernetes
* A quick alternative is to copy and paste a template file for the designated object and edit it as required, in Linux Distributions this can be done via:
  + CTRL+Insert = Copy
  + SHIFT+Insert= = Paste
* Alternatively, the **kubectl run** command can be used to generate a YAML template which can be easily modified, though in some cases you can get away with using kubectl run without creating a new YAML file, such as the following examples.



* In cases where a YAML file is needed, one can add the **--dry-run** flag to the **kubectl run** command and direct its output to a YAML file
* The **--dry-run** flag signals to Kubernetes to not physically create the object described, only generate a YAML template that describes the specified object



## 2.11 - Namespaces

* A namespace called **default** is automatically created when a cluster is created
* They serve to isolate the cluster resources such that they aren’t accidentally manipulated.
* For example, when developing an application, one can have a namespace for “Development” and “Production” to keep the associated resources isolated and intact.
* Each namespace has its own policies, detailing user access, control etc.
* Resource limits may also be set for namespaces
* **Note:** Kubernetes objects associated with internal components such as the Kube-proxy are created under the **kube-system** namespace
* **Note:** At cluster startup, a third namespace is created called **kube-public**, this contains all the resources that should be made available to all users.
* For practice purposes, you shouldn't need to worry about additional namespaces, however when performing more complex actions, it’s better to utilise namespaces to isolate resources.

**Domain Name Service (DNS)**

* For objects communicating within their namespace, they simply refer to the other object as their name.
* For example, suppose a pod wants to connect to the database service within the namespace called “**db-service**”: **mysql.connect(“db-service”)**
* Now suppose the **db-service** is in a separate environment/namespace, in this case for the dev environment, you append the name of the namespace to the service being referenced i.e.:
  + **mysql.connect(“db-service.dev.svc.cluster.local”)**
* In general, follow the format: **“service-name.namespace.svc.cluster.local”**
* This format is used as when the service is created, a DNS entry is automatically added in this format.
* Note:
  + The **cluster.local** part is the default domain name for the cluster
  + svc = Subdomain for services
* To view pods in a particular namespace:



* When creating a pod/resource from a definition file and want to create it in a particular namespace, you can add a similar flag to the command or specify the namespace in the metadata of the object in the form **namespace: <namespace>**



* To create a namespace, you can use a definition file:



* From here you can use the **kubectl create** command
* Alternatively you can run the following command:



* Suppose you’re working on one namespace and wish to switch to work in a different namespace:



* To view all resources in all namespaces, run:



* To specify and limit the resource consumption from namespaces, you can create Resource Quota(s) via definition files.
  + Resources that can be configured include:
    - Pod numbers
    - Cpu units
    - Memory



# 3.0 - Configuration

## 3.1 - Prerequisites: Commands and Arguments in Docker

**Note: This isn’t a requirement for the CKAD exam curriculum, it’s just a “nice to have” piece of information.**

Consider a simple scenario:

* Want to run a docker container via an Ubuntu image
* The container will run an instance of the Ubuntu image and exit immediately

How does this pan out?

* Can see the docker container isn’t running with the command docker ps -a
* This is because containers aren’t designed to host operating systems
* Containers exist to run a specific task or process e.g host a web server, database, etc
* The container will continue to exist as long as the hosted process is active
  + If the process stops or crashes, etc, the container will exit

For the DockerFile for the Ubuntu container, the CMD section was set as “bash”

* This isn’t a command, but a CLI instead!
* When the container ran, docker simply created a container based on the Ubuntu image and launched bash
* In general, Docker doesn’t attach a terminal to a container when it’s ran
  + Bash cannot find a terminal, therefore the container exits as the process is finished

To overcome this, can append commands to the docker run command:

**docker run ubuntu sleep 5**

These changes can be made permanent via Dockerfile either in a Shell or JSON format:

**Shell: CMD command parameter1**

**JSON: CMD [“command”, “parameter1”]**

To build new image and run:

docker build -t image\_name .

docker run <image name>

To use the command but with a value of a parameter subject to changing each time, change CMD to “ENTRYPOINT” i.e.

**ENTRYPOINT [“command”]** or **ENTRYPOINT command** depending on JSON or Shell format.

Any parameters specified on the CLI will automatically be appended to the entrypoint command.

If using an entrypoint and a command parameter isn’t specified, an error message is likely. A default value should be provided to overcome this. This could be done using CMD and ENTRYPOINT in combination.

ENTRYPOINT [“command”]

CMD [“parameter”]

* From this configuration, if no additional parameter is provided, the CMD parameter will be provided
* Any parameter on the CLI will override the CMD parameter

To overwrite the entrypoint command:

docker run --entrypoint <newcommand> <image name>

Note: In general, entrypoint and cmd values should be expressed in a JSON format.

## 3.2 - Commands and Arguments in Kubernetes

## 3.3 - Environmental Variables

## 3.4 - ConfigMaps

## 3.5 - Secrets

## 3.6 - Docker Security

## 3.7 - Security Contexts

## 3.8 - Service Account

## 3.9 - Resource Requirements

## 3.10 - Taints and Tolerations

## 3.11 - NodeSelectors

## 3.12 - Node Affinity

## 3.13 - Taints & Tolerations vs Node Affinity

# 4.0 - Multi-Container Pods

# 5.0 - Observability

## 5.1 - Readiness and Liveness Probes

## 5.2 - Liveness Probes

## 5.3 - Container Logging

## 5.4 - Monitor and Debug Applications

# 6.0 - Pod Design

## 6.1 - Labels, Selectors and Annotations

## 6.2 - Rolling Updates & Rollbacks in Deployments

## 6.3 - Jobs

## 6.4 - CronJobs

# 7.0 - Services and Networking

## 7.1 - Services

## 7.2 - Services - ClsusterIP

## 7.3 - Ingress Networking

## 7.4 - Network Policies

# 8.0 - State Persistence

## 8.1 - Volumes

## 8.2 - Persistent Volumes

## 8.3 - Persistent Volume Claims