KodeKloud Certified Kubernetes Administrator

01 December 2020

01 - Introduction

Course Introduction:

Structure:

- Lectures
- Demos
- Quizzes
- Practice Questions

Prerequisites:

- Basic knowledge of Docker and Kubernetes
- Good knowledge of YAML File format

Note: This course isn't just for the certification, a number of related topics will be discussed to ensure you can install, configure and troubleshoot your own Kubernetes cluster.

Practice Tests

As with the CKAD certificate, CKA is entirely practical, therefore it's ESSENTIAL that you take part in the practice labs to gain an understanding of the relevant skills to apply in production scenarios.

Certification Details

Kubernetes usage is growing exponentially in the market, therefore the need for certified engineers is also growing.

Obtaining the certification makes you fully certified to design, build and administer highly available Kubernetes clusters.

The exam length is 2 HOURS. As with CKAD, you are allowed access to the official Kubernetes documentation throughout.

02 - Core Concepts

Introduction

This section aims to address the core concepts of cluster architecture, API primitives (pods, replicasets, etc.)

Cluster Architecture

Kubernetes exists to allow the hosting of containerized applications in an automated fashion, allowing communication between the different services associated, and facilitating the creation of however many instances you like.

Master Node - The node that manages, plans, schedules workloads and monitors worker nodes.

Worker Nodes - Nodes that host the containerised applications.

Master node is comprised of multiple tools/clusters, making up the control plane:

- ETCD Cluster - Stores information about worker nodes, such as containers running within.
- Schedulers - Identifies the appropriate nodes that a container should be allocated to depending on metrics such as resource requests, node affinity and selectors etc.
- Controllers - Tools responsible for monitoring and responding to node changes, such as optimizing the number of containers running, responding to faulty nodes etc.
- Kube-API Server - Orchestrates all operations within the cluster, exposes the Kubernetes API, which users use to perform management operations.

To run containers on the master and worker nodes, a standardized runtime environment is required, such as Docker.

On the worker nodes, tools included are:

- Kubelet - An engine on each node that carries out operations based on requests from the master node, occasionally sending statistic reports to the kube-apiserver as part of the monitoring
- Kube-proxy - Service that ensures ingress/egress rules are in place to allow inter-pod and node-node communications

ETCD - Beginners

ETCD Is a distributed reliable key-value store that is simple and fast to operate.

Key-value store stores information ina key-value format, each value is associated with a unique key and stored in a database.

To install and run ETCD:

1. Download and extract the binaries from https://github.com/etcd-io/etcd
2. Run the associated executable ./etcd

This starts a service running on port 2379 by default.

Clients can then be attached to the etcd service to store and retrieve information. A default client included is the etcd control client, a CLI client for etcd; used to store and retrieve key-value-pairs.

To store a kvp, run: `./etcdctl set <key> <value>`

To retrieve a value; run: `./etcdctl get <key>`

For additional information; run: `./etcdctl`

ETCD - Kubernetes

The etcd data store stores information relating to the cluster, such as:

- Nodes
- Pods
- Configs
- Secrets
- Roles

Every piece of information obtained from a kubectl get command is obtained from the etcd server. Additionally, any changes made to the cluster, such as adding nodes, deploying pods or updating resources, the same changes are updated in the etcd server. Changes cannot be confirmed until the etcd server has been updated.

The manner in which etcd is deployed is heavily dependent on your cluster setup. For the purposes of this course, 2 cases will be considered:

- A cluster built from scratch
- A cluster deployed using kubeadm

Setting the cluster up from scratch requires manual downloading and installing the binaries; then configuring ETCD as a service in the master node. There are many options that can be added to this service, most of them relate to certificates, the rest describe configuring etcd as a cluster.

One of the primary options to consider is the flag:

```
--advertise-client-urls https://${INTERNAL_IP}: 2379
```

This defines the address which the ETCD service listens on, by default it's port 2379 on the IP of the server. It is in fact this URL that should be configured on the kube-api server when it attempts communication with the etcd server.

If creating a cluster using kubeadm, the ETCD server is deployed as a pod in the kube-system namespace. The database can then be explored using the etcdctl command within the pod, such as the following; which lists all the keys stored by kubernetes.

```
kubectl exec etcd-master -n kube-system etcdctl get / --prefix -keys-only
```

The data stored on the etcd server adheres to a particular structure. The root is a registry containing various kubernetes resources e.g. pods, replicasets.

In high-availability (HA) environments, multiple master nodes will be present, each containing their own ETCD instance. In this sort of scenario, each instance must be made aware of another, which can be configured by adding the following flag in the etcd service file:

```
--initial-cluster controller-0=https://${CONTROLLER0_IP}:2380,
controller-1=https://${CONTROLLER1_IP}:2380, ....,
controller-N=https://${CONTROLLERN_IP}:2380
```

ETCD - Common Commands (Optional Lecture)

ETCDCTL is the CLI tool used to interact with ETCD.

ETCDCTL can interact with ETCD Server using 2 API versions - Version 2 and Version 3. By default, it's set to use Version 2. Each version has different sets of commands.

For example, ETCDCTL version 2 supports the following commands:

```
etcdctl backup
etcdctl cluster-health
etcdctl mk
etcdctl mkdir
etcdctl set
```

Whereas the commands are different in version 3

```
etcdctl snapshot save
etcdctl endpoint health
etcdctl get
etcdctl put
```

To set the right version of API set the environment variable ETCDCTL_API command

```
export ETCDCTL_API=3
```

When API version is not set, it is assumed to be set to version 2, therefore version 3 commands listed above don't work. And vice versa for when set to version 3.

Apart from that, you must also specify path to certificate files so that ETCDCTL can authenticate to the ETCD API Server. The certificate files are available in the etcd-master at the following path. We discuss more about certificates in the security section of this course. So don't worry if this looks complex:

```
--cacert /etc/kubernetes/pki/etcd/ca.crt
--cert /etc/kubernetes/pki/etcd/server.crt
--key /etc/kubernetes/pki/etcd/server.key
```

So for the commands I showed in the previous video to work you must specify the ETCDCTL API version and path to certificate files. Below is the final form:

```
kubectl exec etcd-master -n kube-system -- sh -c "ETCDCTL_API=3 etcdctl get /
--prefix --keys-only --limit=10 --cacert /etc/kubernetes/pki/etcd/ca.crt --cert
/etc/kubernetes/pki/etcd/server.crt  --key /etc/kubernetes/pki/etcd/server.key"
```

31 December 2020

Kube-API Server

- The primary management component in Kubernetes
- When a kubectl command is ran, it is the Kube-API server that is contacted by the kubectl utility for the desired action
- The kube-api-server authenticates and validates the request, then retrieves and displays the requested information from the etcd server.
- Note: The kubectl command isn't always necessary to set the API server running.
    - Can instead send a post request requesting a resource creation such as:
        - Curl -X POST /api/v1/namespaces/default/pods ... [other]
    - In this scenario, the following steps occur:
        - The request is authenticated
        - The request is validated
        - The API server creates a POD without assigning it to a node
        - Updates the information in the ETCD server and the user to inform of the pod creation.
    - The updated information of the nodeless pod is acknowledged by the scheduler; which monitors the API server continuously
        - The scheduler then identifies the appropriate node to place the pod onto; communicating this back to the API server
    - The API server updates the ETCD server with this information and passes this information to the kubelet in the chosen worker node
    - Kubelet agent creates the pod on the node and instructs the container runtime engine to deploy the chosen application image.
    - Finally, the kubelet updates the API server with the change(s) in status of the resources, which in turn updates the ETCD cluster's information.
- This pattern is loosely followed every time a change is requested within the cluster, with the kube-api server at the centre of it all.
- In short, the Kube-api server is:
    - Responsible for validating requests
    - Retrieving and updating the ETCD data store

        - The API-Server is the only component that interacts directly with the etcd data store
- Other components such as the scheduler and kubelet only use the API server to perform updates in the cluster to their respective areas
- Note: The next point doesn't need to be considered if you bootstrapped your cluster using kubeadm
    - If setting Kubernetes up "The Hard Way", the kube-apiserver is available asa binary in the kubernetes release pages
    - Once downloaded and installed, you can configure it to run as a service on your master node
- Kubernetes architecture consists of a lot of different components working with each other and interacting with each other to know where they are and what they're doing
    - Many different modes of authentication, authorization, encryption and security, leading to many options and parameters being associated with the API server.
- The options within the kube-apiserver's service file will be covered in detail later in the notes, for now, the important ones are mainly certificates, such as:
    - --etcd-certifile=/var/lib/kubernetes/kubernetes.pem
    - --kubelet-certificate-authority=/var/lib/kubernetes/ca.pem
- Each of the components to be considered in this section will have their own associated certificates.
- Note: To specify the location of the etcd servers, add the optional argument:
    - --etcd-servers=https://127.0.0.1:2379
        - Change IP address where appropriate or port, it's via this address the kube-api-server communicates with the etcd server
- Viewing the options of the kube-api server in an existing cluster depends on how the cluster was set up:
    - Kubeadm:
        - The kube-api server is deployed as a pod in the kube-system namespace
        - Options can be viewed within the pod definition file at: /etc/kubernetes/manifests/kube-apiserver.yaml

    - Non-kubeadm setup:
        - Options displayed in kube-apiserver.service file at /etc/systemd/system/kube-apiserver.service
        - Alternatively, use **ps -aux | grep kube-apiserver** to view the process and its associated options

Kube Controller Manager

- The kubernetes component that manages each of the controllers within Kubernetes.
- Each controller has their own set of responsibilities, such as monitoring and responding to changes in kubernetes resources or containers.
    - Continuous monitoring determined by “Watch Status”
    - Responsive actions carried out to “Remediate the situation”
- In terms of Kubernetes:
    - Controllers are processes that continuously monitor the state of various components within the system
    - If any changes occur that negatively affect the system, the controllers work towards bringing the system back to “normal”
- A common example of a controller us the node controller
    - Monitors the status of nodes in the cluster and takes responsive actions to keep it running
- Any actions a controller takes are done via the kube-api server.
- The monitoring period for controllers can be configured and varies, for example the node controller checks the status of the nodes every 5 seconds.
    - Allows frequent and accurate monitoring
    - If the controller cannot communicate with a node after 40 seconds, it’s marked as “Unreachable”
    - If after 5 minutes the node is still unreachable, the controller takes any pods originally on the node and places them on a healthy available one.
- Another example is the replication controller
    - Responsible for monitoring the status of replicasets and ensuring that the desired number of pods are available at all times within the set.
    - If a pod dies, it creates another.
- Many more controllers are found within Kubernetes, such as:

    - Deployments
    - CronJobs
    - Namespace
- All controllers are packaged as part of the Kube-Controller Manager; a single process
- To install and view the Kubernetes Controller Manager. You can download and extract the binary from the Kubernetes release page via wget etc, where you can then run it as a service.
- When running the Kubernetes Controller Manager as a service, you can see that there are a list of customisable options available.
    - Some options that are customizable include node monitor grace period, monitoring period etc.
- You can also use the --controllers flag to configure and view what controllers you're using.
- As with the Kube-API Server, the way you view the options on the Kube-Controller Manager depends on your cluster's setup:
    - Kubeadm:
        - The kube-api server is deployed as a pod in the kube-system namespace
        - Options can be viewed within the pod definition file at: /etc/kubernetes/manifests/kube-controller-manager.yaml
    - Non-kubeadm setup:
        - Options displayed in kube-controller-manager.service file at /etc/systemd/system/kube-apiserver.service
        - Alternatively, use **ps -aux | grep kube-controller-manager** to view the process and its associated options.

Kube Scheduler

- Responsible for scheduling pods on nodes i.e. identifying the best node for objects such as pods and deployments to be placed on.
- It's a common misconception that the scheduler is responsible for actually placing the resources onto the nodes, this is actually Kubelet's responsibility.

- A scheduler is needed to ensure that containers and resources end up on the nodes that can successfully accommodate them based on certain criteria:
    - Resource requirements for pod
    - Resource capacity/quota for nodes
- The scheduler follows the 2-step process to make its decision:
    - Filters nodes that don't fit the resource requirements for the pod/object
    - Uses a priority function to determine which of the remaining nodes is the best fit for the object based on the node's resource capacity, scoring from 0-10.
        - For example, if a pod requiring 10 cpu units could be placed on a node with 12 total units or 16, it's preferable to place it on the 16-unit one as this leaves more space for additional objects to be deployed to the pod.
- The scheduler also utilises other tools such as taints and tolerations, and node selectors/affinity.
- To install the kube-scheduler, extract and run the binary from the release page as a service; under the file kube-scheduler.service, where you can configure the options as per usual.
- As with the Kube-API Server, the way you view the options on the Kube-Scheduler depends on your cluster's setup:
    - Kubeadm:
        - The kube-api server is deployed as a pod in the kube-system namespace
        - Options can be viewed within the pod definition file at: /etc/kubernetes/manifests/kube-scheduler.yaml
    - Non-kubeadm setup:
        - Options displayed in kube-kube-scheduler.service file at /etc/systemd/system/kube-scheduler.service
        - Alternatively, use **ps -aux | grep kube-scheduler** to view the process and its associated options.

Kubelet

- Kubelet registers nodes and other objects within Kubernetes to their required places on a cluster.
- When it receives instructions from the kube-scheduler via the kube-api server to load a container, pod etc on the node, it requests the container runtime (usually Docker), to pull the required image.
- Once the request is made, it continues to periodically monitor the state of the pod and the containers within, reporting its findings to the kube-apiserver.
- When installing the kubelet, it must be noted that if setting up a cluster via kubeadm, the kubelt isn't automatically deployed.
    - This is a KEY difference.
    - You must always manually install the kubelet on your worker nodes.
- To install, download the binary from the release page, from which you can extract and run it as a service under kubelet.service
- The associated options can be viewed by either:
    - **/etc/systemd/system/kubelet.service**
        - Options can be configured within the file
    - **ps -aux | grep kubelet**

02 January 2021

Kube Proxy

- In a cluster, every pod can interact with one another as long as a Pod Networking solution is deployed to the cluster
    - Pod Network - An internal virtual network across all nodes within the cluster, allowing any pod within the cluster to communicate with one another
- There are multiple solutions for deploying a pod network.
    - In one scenario, suppose you have a web application and a database running on two separate nodes.
    - The two instances can communicate with each other via the IP of the respective pods.
- In the example above, the problem arises when the IP of the pods aren't static, to work around this, you can expose the pods across the cluster via a service.

- The service will have its own static IP address, so whenever a pod is to be accessed or communicated with, communications are routed through the service's IP address to the pod it's exposing.
- Note: The service cannot join the pod network as it's not an actual component, more of an abstraction or virtual component.
    - It's not got any interfaces or an actively listening process.
- Despite the note, the service needs to be accessible across the cluster from any node. This is achieved via the Kube-Proxy.
- Kube-Proxy: A process that runs on each node in the kubernetes cluster.
- The process looks for new services continuously, creating the appropriate rules on each node to forward traffic directed to the service to the associated pods.
- To allow the rule creation, the Kube-Proxy uses IPTables rules.
    - Kube-Proxy creates an IP Tables rule on each node within the cluster to forward traffic heading to the specific service to the designated pod; almost like a key-value-pair.
- To install, download the binary from the release page, from which you can extract and run it as a service under kube-proxy.service
- The kubeadm tool deploys kube-proxy as a PODs on each node
    - Kube-Proxy deployed as a Daemon Set, a single POD is always deployed on each node in the cluster.

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

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
        a. The version of the Kubernetes API to be used to create the object
        b. Correct api version must be specified for objects

Kind	ApiVersion
Pod	v1
Service	v1
ReplicaSet	apps/v1

Deployment	apps/v1

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

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

```
Kubectl create -f <pod-definition>.yaml
```

- To view pods:

```
Kubectl create get pods
```

- To view detailed information about a particular pod:

```
Kubectl describe pod <pod-name>
```

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:
    - ReplicationController
    - 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:

```
apiVersion: v1
Kind: ReplicationController
metadata:
        Name: myapp-rc
        Labels:
                App: myapp
                Type: front-end
Spec:
        Template:
                <pod spec>
        Replicas: <replica number>
```

- 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:

```
Kubectl get replicationcontroller
```

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

ReplicaSet:

```
apiVersion: apps/v1
Kind: ReplicaSet
metadata:
        Name: myapp-replicaset
        Labels:
                App: myapp
                Type: front-end
Spec:
        Template:
                <pod spec>
        Replicas: <replica number>
        Selector:
                matchLabels:
                        Type: front-end
```

- 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 the event of failure in with the correct configuration.

Scaling:

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

```
Kubectl replace -f replica-definition.yaml
```

- Alternatively:

```
Kubectl scale --replicas=<number> replicaset-definition.yaml
```

- Or:

```
Kubectl scale --replicas=<x> replicaset/<replicaset-name>
```

- 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

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:

```
apiVersion: apps/v1
kind: Deployment
metadata:
        name: myapp-deployment
        labels:
                app: myapp
                type: frontend
spec:
        template:
                metadata:
                        name: myapp-pod
                        labels:
                                app: myapp
                                type: front-end
                spec:
                        containers:
                        - name: nginx-container
                          Image: nginx
        replicas: 3
        selector:
                matchLabels:
                        type: front-end
```

- 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

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.

Examples:

Creating an nginx pod

kubectl run nginx --image=nginx

Creating an nginx deployment

kubectl create deployment --image=nginx nginx

- 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=client** flag signals to Kubernetes to not physically create the object described, only generate a YAML template that describes the specified object

Examples:

Creating an nginx pod YAML file without deploying the pod:

kubectl run nginx --image=nginx --dry-run -o yaml > nginx-pod.yaml

Creating an nginx deployment:

kubectl create deployment --image=nginx nginx --dry-run -o yaml > nginx-deployment.yaml

Creating an nginx deployment YAML file with a particular number of replicas:

kubectl create deployment --image=nginx nginx --replicas=4 --dry-run=client -o yaml > nginx-deployment.yaml

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:

```
kubectl get pods --namespace=<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>**

```
kubectl create -f <definition>.yaml  --namespace=<namespace>
```

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

```
apiVersion: v1
kind: Namespace
metadata:
        name: dev
```

- From here you can use the **kubectl create** command

- Alternatively you can run the following command:

```
kubectl create namespace <namespace_name>
```

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

```
kubectl config set-context $(kubectl config current-context)
--namespace=<namespace_name>
```

- To view all resources in all namespaces, run:

```
kubectl get all --all-namespaces
```

- 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

```
apiVersion: v1
kind: ResourceQuota
metadata:
        name: compute-quota
        namespace: dev
spec:
        hard:
                pods: "10"
                requests.cpu: "4"
                requests.memory: 5Gi
                limits.cpu: "10"
                limits.memory: 10Gi
```

Services

- Kubernetes services enable inter-component communication both internal and external to your application
- Additionally, the services allow connections to be made from an application to users or other applications.
- Consider an example application which has groups of pods running different aspects of the application, such as one group running the frontend to users, an API backend and connecting to an external data source,
    - It's services that enable connectivity between these groups of pods and for users to be able to access the frontend, communication between the frontend and backend, and for the application to connect to the external data source.
- In short, services enable a loose coupling between microservices offered by applications.

- Suppose we have a pod with a web application running on it. To understand how an external user accesses the web page, one must first understand the existing setup.
    - The Node the pod is running on will have its own IP address e.g. 192.168.1.2
    - The laptop is running on the same network so its IP address is for example 192.168.1.10
    - The internal POD network is in the range of 10.244.0.0; the POD is at 10.244.0.2
- Because the POD network is separate to the Node and Laptop, we cannot access it or ping it.
- It would be possible to SSH into the Kubernetes node and access the PODs webpage via a Curl operation (or use the Node's GUI to access a browser and navigate to the address), but this is troublesome and problematic.
- In reality we would like to remove the need for SSH'ing into the node, and just access the service by accessing the Node's IP address and getting rerouted
- Kubernetes services allows this problem to be solved; it allows the requests made from our laptop to the node to be mapped to the POD and its container.

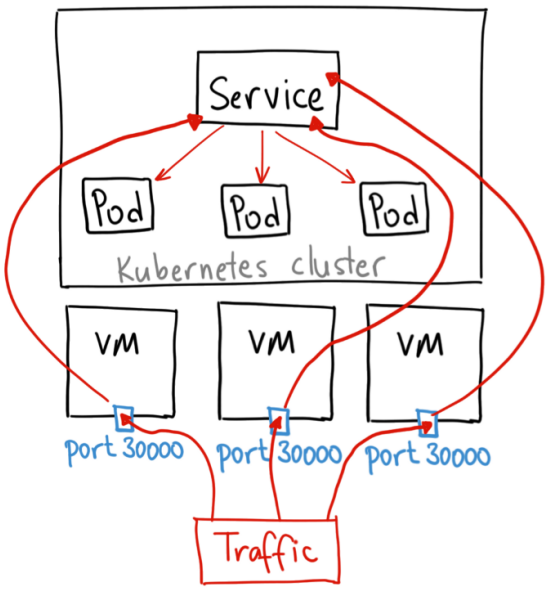
- Services are Kubernetes objects similar to PODs, deployments etc and have multiple use cases

Service Types

- There are multiple types of services available:
    - **NodePort:**
        - A service that listens to a port on the node and forwards requests to that port to a port on an appropriate pod
        - Makes the internal pod accessible via a port on the Node rather than SSH'ing into the Node
    - **ClusterIP:**
        - The service creates a virtual IP within the cluster to enable communication between different services
        - Services that can communicate with each other as a result of this include sets of frontend servers to backend servers
    - **LoadBalancer:**
        - Service provisions a load balancer for our service in supported cloud providers
        - A good example of using a LoadBalancer would be to distribute load across the different web servers of your application's frontend.

NodePort Demonstration

- When working with a NodePort service, there are 3 ports involved:
    - targetPort: The port where the actual web application is running
    - Port: The port on the service itself
    - NodePort: The port on the node used to access the web server externally
        - Defaults to a range between 30000 - 32767



- The service is similar in nature to a virtual server within the node within the cluster
    - It has its own IP address - The cluster IP of the service
- In general, it's better to use a definition file to create a service.

```
apiVersion: v1
kind: Service
metadata:
        name: myapp-service
spec:
        type: NodePort
        ports:
        - targetPort: 80 #assumed to be the same as port if not provided
          port: 80 #The only required port
          nodePort: 30008 #automatically assigned within range if not specified
        selector:
                matchLabels: #add labels associated with pod/object to be exposed
                        app: myapp
                        type: front-end
```

- Under ports, port is the only required entry, the rest are automatically generated if not specified
- As always, use kubectl create -f to create the service from the definition file. To view the service, run:

```
kubectl get services
```

- Running the above command produces a list of all services running within the cluster, as well as information regarding their type, cluster and external IPs, using this information allows users to access the exposed service via a curl command (or web browser) to the specified port and the node's IP address.
- In a production environment, chances are there will be multiple instances of the application running for high availability purposes
    - All pods will share the same labels
    - Via the selector, the service will therefore pick up on all instances and picks the most appropriate to forward a request to via a random algorithm

    - This makes the service a built-in load balancer.
- In the event the pods are running on separate nodes, Kubernetes will automatically configure the service to map the targetport to the same nodeport for the pods associated.
- Regardless of whether a service is running for a:
    - Singular pod
    - Multiple pods on a single node
    - Multiple pods across multiple nodes
- Kubernetes services are created the exact same way, with no additional configuration required.
- The service will automatically update itself, making it one of Kubernetes most flexible and adaptive offerings.

06 January 2021

Services - ClusterIP

- Typical full stack web apps have different pods running different aspects of the application e.g:
    - One set running a Key-Value store (E.g. Redis)
    - One set running a persistent database (E.g. MySQL)
    - One set running the frontend
- These various sets need to communicate with each other for the app to work, such as the web frontend server communicating with the backend servers
- Each pod involved has their own IP address, however these are not static as pods are frequently deleted and recreated; they cannot be relied on for internal communication.
- Additionally, considerations must be made for when one of the pods from a set wants to interact with the pods from another set, how is this decided?
- A Kubernetes Service of type ClusterIP can provide assistance with this by grouping these sets of pods together to provide a single interface for other pods to access the service.
- For this to work, a ClusterIP service should be created for each of the different pod sets i.e. one for the frontend, one for the backend, and so on

- If a request is made to this service, via a single IP address within the cluster, the request is randomly forwarded to one of the pods the service is exposing.
- By creating services like this, microservices-based applications can easily be deployed, with each service scalable without affecting other services involved.
- To create a service like this, you can use either a definition file or the command line:

```
apiVersion: v1
kind: Service
metadata:
        name: backend
spec:
        type: ClusterIP #default type anyways
        ports:
        - targetPort: 80 #the port which the backend is exposed
          port: 80 #The port is where the service is exposed
        selector:
                matchLabels: #used to determine the pods to be exposed by the service
                        app: myapp
                        type: front-end
```

```
Kubectl expose pod/<podname> --type=ClusterIP --options
```

Services - LoadBalancer

- Kubernetes service type that helps balance traffic routing to underlying services and nodes within the cluster.
- Only supported on separate cloud platforms such as GCP, Azure etc
- Unsupported in environments such as Virtualbox, if still used, it basically has the same effect as a service of type NodePort.

Imperative vs Declarative

- Imperative: The use of statements to change a programs state, give a program step-by-step instructions on how to perform a task, specify the “how” to get to the “what”

- Declarative: Writing a program describing an operation by specifying only the end goal, specify the “what” only
- In Kubernetes, this split in programming language can be broken down as:
    - Imperative - Using kubectl commands to perform CRUD operations like scaling and updating images, as well as operations with .yaml definition files.
        - These commands specify the exact commands and how they should be performed.
    - Declarative:
        - Using kubectl apply commands with definition files, Kubernetes will consider the information provided and determine what changes need to be made
- Imperative commands in kubernetes include:
    - Creation
        - Run
        - Create
        - Expose
    - Update Objects
        - Edit
        - Scale
        - Set (image)
- It should be noted that Imperative commands are often “one-time-use” and are limited in functionality, for advanced operations it’s better to work with definition files, and that’s where using the -f <filename> commands are better-used.
- Imperative commands can become taxing as they require prior knowledge of pre-existing configurations, which can result in extensive troubleshooting if unfamiliar.
- For the declarative approach, it’s more recommended to use this when making extensive changes or updates without having to worry about manual troubleshooting or management.

Kubectl Apply

- The apply command takes into consideration 3 aspects when ran:
    - The local configuration file

    - The live object's current definition file
    - The last applied configuration
- If the object doesn't already exist when the command is ran, it's created.
- The live definition includes additional information regarding the status of the object
- When the apply command is ran, the configuration file is converted into a JSON format and stored as the "last applied configuration" as a reference point going forward.
- All three aspects are then compared the next time kubectl apply is ran to determine what changes are to be made.
- To understand why the last applied configuration is needed, suppose a field is deleted like a typed label:
    - If the apply command is ran, the last configuration applied is shown to have a label which isn't present in the local configuration, so it must be removed from the live configuration.
- In reverse i.e. the field is in the local file but not in the last applied configuration, the field is added.

08 January 2021

03 - Scheduling

Manual Scheduling

- When a pod is made available for scheduling, the Scheduler looks at the PODs definition file for the value associated with the field NodeName
- By default, NodeName's value isn't set and is added automatically when scheduling
- The scheduler looks at all pods currently on the system and checks if a value has been added to the NodeName field, any which do not are a candidate for scheduling.
- The scheduler identifies the best candidate for scheduling using its algorithm and schedules the pod onto that Node by adding the node's name to the NodeName field.
- This setting of the NodeName field value binds the Pod to the Node.

- If there is no scheduler to monitor and schedule the nodes, the pods will remain in a pending state.
- Pods can be manually assigned to nodes if a scheduler isn't present.
- This can be done by manually setting the pod's value for NodeName in the definition file.
    - This can only be done before the pod is created for the first time, it cannot be done post-creation for a pre-existing pod
    - To configure, as a child of the pod's Spec, add the field: nodeName: <nodename>
- Alternatively, you can assign a node by creating a binding object definition file to send a post request to the pod binding API; mimicking the scheduler's actions.

```
apiVersion: v1
kind: Binding
metadata:
        name: nginx
target:
        apiVersion: v1
        kind: Node
        name: node02
```

- Once the binding definition file is written, a post request can be sent to the pod's binding API; with the data set to the binding object in a JSON format in a similar vein to:

```
curl --header "Content-Type:application/json" --request POST --data
'{"apiVersion":"v1", "kind": "Binding" …}

http://$SERVER/api/v1/namespaces/default/pods/$PODNAME/binding
```

10 January 2021

Labels and Selectors

- Built-In Kubernetes features used to help distinguish objects of similar nature from one another by grouping them
- Labels are added under the metadata section, where an infinite number of labels can be added to the Kubernetes object in a key value format
- To filter objects with labels, use the kubectl get command and add the flag --selector followed by the key-value pair in the format key=value

```
kubectl get <object> --selector key=value
```

- Selectors are used to link objects to one another, for example, when writing a replica set definition file, use the selector feature in the spec to specify the labels the object should look for in pods to manage.
- The same can be applied to services to help identify what pods or deployments it is exposing.
- Annotations - Used to record data associated with the object for integration purposes e.g. version number, build name etc

Taints & Tolerations

- Used to set restrictions regarding what pods can be scheduled onto a node
- Consider a cluster of 3 nodes with 4 pods preparing for launch:
    - The scheduler will allocate the pods as equally as possible if no restrictions are applied (1 to 1 node each, then the fourth to any of the remaining nodes)
- Suppose now only node 1 has resources available to run a particular application
    - Can apply a taint to node 1: Prevents unwanted pods from being launched on Node 1
    - Need to then apply a tolerant to the pod(s) to be ran only on node 1
        - Node 1 can only run pods that tolerate the taint
- Taints and tolerations allow the scheduler to allocate pods to required nodes
- Note: By default, no taints or tolerations are applied to nodes and pods
- To apply a taint:

```
kubectl taint nodes <nodename> key=value:<taint-effect>
```

- The key=value part of the command above would be the key value pair used for labelling the specific pods
- The taint effect determines what happens to pods that are intolerant to the taint; there are 3 possibilities:
    - NoSchedule - Intolerant pods won't be scheduled AT ALL for this node
    - PreferNoSchedule - Intolerant pods won't be scheduled for the majority of the time on this node, however if there's no other option they will be
    - NoExecute - Intolerant pods won't be scheduled and any existing pods that are intolerant to the taint will be evicted.
- To apply a toleration to a pod, add it to the pod's spec file in a similar form to:

```
tolerations:
- key: app
  operator: "equal"
  value: "blue"
  effect: "NoSchedule"
```

- Note: The values included under tolerations are identical to that of the kubectl command above
- All values need to be in quotes ""

Taints: NoExecute

- Suppose node1 is to be used for a particular application
    - Apply a taint to node 1 with the app name and apply a toleration to the pod running the app
    - Setting the taint effect to NoExecute, all preexisting pods on the node that are intolerant to the taint are stopped and eviced
- Taints and tolerations are only used to restrict pod access to nodes

- As there are no restrictions/taints applied to the other pods, there is a chance the app could still be placed on a different node
- If you wanted a pod to go to a particular node, a concept of node affinity must be used in combination with taints and tolerations
- Note: A taint is automatically applied to the master node such that no pods can be scheduled to it
- To view taints associated with a node, run:

```
kubectl describe node <nodename> | grep/Select-String "Taint"
```

- Note: to untaint a node, run the same command as you would to taint the node, but add a - immediately after the taint effect e.g. NoSchedule-

Node Selectors

- Consider a 3-node cluster, with node 1 having a larger resource configuration
- In this scenario, we would like the task/process requiring more resources to go to the larger node; without restrictions this may not occur
- To solve, can place limitations on pods via the use of node selectors
- To apply a node selector, add the following to a definition file:

```
nodeSelector:
    key: value
```

- Note: For node selectors to work, the node involved must already have the same key value pair as the nodeSelector in its labels, this can be done via:

```
kubectl label nodes <node name> <key>=<value>
```

- When pods are created once the node is labelled and the node selector is added to the pod definition; it should go to the node as desired

Limitations of Node Selectors:

- Node selectors are beneficial for simple allocation tasks, but if more complex situations arise, you need to use Node Affinity
    - An example of where this would apply would be if you wanted a pod to go to either 1 of 2 nodes; this cannot be completed using OR actions

Node Affinity

- Node affinity looks to ensure that pods are hosted on particular nodes
- This can ensure high-resource consumption jobs go to high-resource nodes; similar to the example in the previous section.
- Node affinity allows more complex capabilities regarding pod-node restrictions
- As per usual, you specify the node affinity in the definition file for the pod's spec:

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms: #array
      - matchExpressions:
        - key: size
          operator: In #pod will be placed on any node that matches the values specified
          values:
          - Large
```

- If for example, you wanted the pod to go any of a set number of nodes, for example either nodes where size=large OR size=medium, add - medium or any other applicable values to the list of values under matchExpressions.
- Note: You could also use the NotIn operator in a similar manner to the above to neglect particular values
- If you just want a pod to go to any node with a particular label; regardless of the value, use the Exists operator; this doesn't require any values to be specified
- In the event that a node cannot be allocated due to a label fault, the resulting action is dependent on the node affinity type set.

Node Affinity Types:

- Determine how the scheduler behaves regarding Node Affinity and pod lifecycle stages
- There are two types available:
    - requiredDuringSchedulingIgnoredDuringExecution
    - preferredDuringSchedulingIgnoredDuringExecution
- Other types are to be released, such as requiredDuringSchedulingRequiredDuringExecution
- To understand the types of Node Affinity, it is helpful to break them down by considering the two stages of a pod's lifecycle:
    - DuringScheduling:
    - DuringExecution
- Considering the DuringScheduling aspect, if the node isn't available according to the Node Affinity specification, the resultant action is dependent upon the node Affinity type:
    - Required:
        - The pod must be placed on a node that satisfies the node affinity criteria
        - If no node satisfies the criteria, the pod will not be scheduled at all
        - This type is generally used when the node placement for a pod is crucial
    - Preferred:
        - If the pod placement is less important than the need for actually running the task
        - If a matching node isn't found, the scheduler will ignore the node affinity criteria and place the pod on any node that can house it
- Suppose a pod has been running and a change is made to affect Node Affinity, for example a change in node labels, how the pod responds to this change is determined by the prefix of DuringExecution:
    - Ignored:
        - The pod will continue to run
        - Any changes in NodeAffinity will have no affect once they are scheduled

    - Required:
        - To be introduced
        - When applied, any current pods that don't meet the NodeAffinity requirements when the changes are made will be evicted.

Taints & Tolerations vs Node Affinity

- Consider a 5-cluster setup, with a blue, red, green and two random pods:
    - Blue - To run blue pod
    - Red - To run red pod
    - Green - To run green pod
    - Node1 - To host gray pod
    - Node2 - To host gray pod
- A taint can be applied to each of the coloured nodes to accept their respective pods
- Tolerations applied to pods as appropriate
- There needs to be a taint applied to node1 and node2 as the coloured pods could still be allocated where they're not wanted
- To overcome this problem, Node Affinity could be applied:
    - Label the nodes with the respective colours
    - Pods end up in correct pods via the use of Node Selector
    - There's still a chance that the unwanted pods could still be allocated
- Therefore, a combination of taints and tolerations and node affinity must be used
    - Taints and tolerations applied to prevent unwanted pod placement on nodes
    - Use node affinity to prevent the correct pods from being placed on incorrect nodes

Resource Requirements & Limits

- Consider a 3-node cluster, each node has a set amount of resources available to be consumed by pods:
    - CPU
    - Memory
    - Disk Space
- The Kubernetes Scheduler is responsible for allocating pods to nodes

    - The scheduler takes into account the node's current resources and the resources requested by the pod
    - If no resources are available, the scheduler holds the pod in a pending state
- Kubernetes automatically assumes a pod or container within a pod will require at minimum the following resources:
    - 0.5 CPU units
    - 256Mi of memory
- If the pod or container requires more resources than allocated above, you can configure/specify this in the pod's definition file spec by adding a "resources" property within the containers section for each container

```
containers:
- name:
  image:
  resources:
    requests:
      memory: "1Gi"
      cpu: 1
    limits:
      memory: "2Gi"
      cpu: 2
```

- Within the resources section, you can specify the resources required by the pod at minimum and the limits for resource consumption by the pod so it doesn't affect the node and other pods within
    - Limits must be specified as Docker containers have no limit to the resources they can consume
    - By default, Kubernetes will set the limits for Docker containers so it can only use a maximum of 1 CPU unit and 512 Mebibytes of memory, if the limits need changing, configure the definition file as shown above
- In the event that the CPU request is overloaded for the node repeatedly, the cpu usage is "throttled" to prevent the limit being exceeded
- For memory overload, if the limit is repeatedly exceeded, the pod is terminated

Resource Metrics - CPU

- Can be set from 1m (1 micro to as high as the system allows)

- 1 CPU unit is equivalent to:
    - 1 AWS vCPU
    - 1 GCP Core
    - 1 Azure Core
    - 1 Hyperthread

Resource Metrics - Memory

- Memory can be allocated with any of the following suffixes depending on the application's requirements:

Symbol	Memory Type Name	Numerical Value
G	Gigabyte	1000M
M	Megabyte	1000K
K	Kilobyte	1000 Bytes
Gi	Gigibyte	1024Mi
Mi	Mebibyte	1024Ki
Ki	Kibibyte	1024 Bytes

Note on Default Resource Requirements and Limits:

- Naturally Kubernetes assumes containers request 0.5 units of CPU and 256Mi of memory
- These defaults can be configured to suit for each namespace within the Kubernetes cluster by setting a limitrange, which can be produced via a yaml definition file similar to:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-limit-range #usually keep memory and cpu LimitRange separate
spec:
  limits:
  - default:
      memory: 512Mi
      cpu: 1
    defaultRequest:
      memory: 256Mi
      cpu: 0.5
    type: Container
```

Notes on Pod/Deployment Editing

- When editing an existing pod, only the following aspects can be edited in the spec:
    - Image (for containers and initcontainers)
    - activeDeadlineSeconds
    - Tolerations
- Aspects such as environment variables, service accounts and resource limits cannot be edited easily, but there are ways to do it:
    - Editing the specification:
        - Run kubectl edit pod <podname> and edit the appropriate features
        - When saving to log the changes, if the feature cannot be edited for an existing pod, the changes will be denied
        - A copy of the definition file with the changes will saved to a temporary location, which can be used to recreate the pod with the changes once the current version is deleted
    - Extracting and editing the yaml definition file:
        - Run kubectl get pod <podname> -o yaml > <filename.yaml>
        - Make the desired changes to the yaml file and delete the current version of the pod
        - Create the pod again with the file

- When editing the deployment, any aspect of its underlying pods can be edited as the pod template is a child of the deployment spec
    - When changes are made, the deployment will automatically delete and create new pods to apply the updates as appropriate

DaemonSets

- Daemonsets are similar in nature to replicasets, they provide assistance in the deployment of multiple instances of a pod
- Daemonsets run only one instance of the pod per node
- Whenever a new node is added, the pod is automatically added to the node and vice versa for when the node is removed
- Use cases of Daemonsets include monitoring and logging agents
    - Removes the need for manually deploying one of these pods to any new nodes within the cluster
    - Kubernetes components such as Kube-Proxy could be deployed as a Daemonset as one pod is required per cluster
        - Similar network solutions could also be deployed as a Daemonset
- Daemonsets can be deployed via a definition file, it's similar in structure to that of a Replicaset, with the only difference being the Kind

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: monitoring-daemon
spec:
  selector:
    matchLabels:
      app: monitoring-agent
  template:
    metadata:
      labels:
        app: monitoring-agent
    spec:
      containers:
      - name: monitoring-agent
          image: monitoring-agent
```

- To view daemonsets, use the **kubectl get daemonsets** command
- You can view the details of the daemonset with the kubectl describe command i.e. **kubectl describe daemonset <daemonset name>**
- Prior to Kubernetes v1.12, a Daemonset would work by manually setting the nodename for each pod to be allocated, thus bypassing the scheduler
- Post Kubernetes v1.12, the Daemonset uses the default scheduler and Node Affinity rules discussed previously to allocate the single pod to each node

Static Pods

- Kubelet relies on the kube-apiserver for instructions on what pods to load on its respective node
- The instructions are determined by the kube scheduler which was stored in the etcd server
- Considerations must be made if any of the kube-api server, scheduler and etcd server are not present, as a worst case scenario, suppose none of them are available

- Kubelet is capable of managing a node independently to an extent
    - The only thing kubelet knows to do is to create pods, however in this scenario there's no api server to feed it the instructions based on yaml definition files
    - To work around this, you can configure the kubelet to read the pod definition files from a directory on the server designated to store information about pods
    - Once configured, the Kubelet will periodically check the directory for any new files, where it reads the information and creates pods based on the information provided
- In addition to creating the pods, the kubelet would take actions to ensure the pod remains running, i.e.:
    - if a pod crashes, kubelet will attempt to restart it
    - if any changes are made to any files within the directory, the kubelet will recreate the pod to cause the changes to occur
- Pods created in this manner, without the intervention of the API server or any other aspects of a kubernetes cluster, are **Static Pods**
- Note: Only pods could be created in this manner, objects such as Deployments and Replicasets cannot be created in this manner
- To configure the "Desginated Folder" for the kubelet to look in for pod definition files, add the following option to the kubelet service file kubelet.service; note the directory could be any directory on the system:

```
--pod-manifest-path=/etc/Kubernetes/manifests
```

- Alternatively, you could create a yaml file to specify the path the kubelet should look at, i.e. staticPodPath: /etc/Kubernetes/manifests, which can be referenced by adding the --config=/path/to/yaml/file to the service file
    - Note this is the kubeadm approach
- Once static pods are created, they can be viewed by docker ps (can't use kubectl due to no api server)

- It should be noted that even if the api server is present, both static pods and traditional pods can be created
- The api server is made aware of the static pods because when the kubelet is part of a cluster and creates static pods, it creates a mirror object in the kube api server
    - You can read details of the pod but cannot make changes via the kubectl edit command, only via the actual manifest
- Note: the name of the pod is automatically appended with the name of the node it's assigned to
- Because static pods are independent of the Kubernetes control plane, they can be used to deploy the control plane components themselves as pods on a node
    - Install kubelet on all the master nodes
    - Create pod definition files that use docker images of the various control plane components (api server, controller, etcd server etc)
    - Place the definition files in the chosen manifests folder
    - The pods will be deployed via the kubelet functionality
    - Note: By doing this, you don't have to download the associated binaries, configure services or worry about services crashing
- In the event that any of these pods crash, they will automatically be restarted by Kubelet with them being a static pod
- Note: To delete a static pod, you have to delete the yaml file associated with it from the path configured

Static Pods vs Daemonsets

Static Pods	**Daemonsets**
Created via Kubelet	Created by Kube-API server (Daemonset controller)
Used to deploy control plane components as static pods	Used to deploy monitoring agents and logging agents on nodes
Ignored by Kube-Scheduler	

Multiple Schedulers

- The default scheduler has its own algorithm that takes into accounts variables such as taints and tolerations and node affinity to distribute pods across nodes
- In the event that advanced conditions must be met for scheduling, such as placing particular components on specific nodes after performing a task, the default scheduler falls down
- To get around this, Kubernetes allows you to write your own scheduling algorithm to be deployed as the new default scheduler or an additional scheduler
    - Via this, the default scheduler still runs for all usual purposes, but for the particular task, the new scheduler takes over
- You can have as many schedulers as you like for a cluster
- When creating a pod or deployment, you can specify Kubernetes to use a particular scheduler
- When downloading the binary for kube-scheduler, there is an option in he kube-scheduler.service file that can be configured; --scheduler-name
    - Scheduler name is set to default-scheduler if not specified
- To deploy an additional scheduler, you can use the same kube-scheduler binary or use one built by yourself
    - In either case, the two schedulers will run as their own services
    - It goes without saying that the two schedulers should have separate names for differentiation purposes
- If a cluster has been created via the Kubeadm manner, schedulers are deployed via yaml definition files, which you can then use to create additional schedulers by copying the file
    - Note: customise the scheduler name with the --scheduler-name flag
- Note: The leader-elect option should be used when you have multiple copies of the scheduler running on different master nodes
    - Usually observed in a high-availability setup where there are multiple master nodes running the kube-scheduler process
    - If multiple copies of the same scheduler are running on different nodes, only one can be active at a time

    - The leader-elect option helps in choosing a leading scheduler for activities, to get multiple schedulers working, do the following:
        - If you don't have multiple master nodes running, set it to false
        - If you do have multiple masters, set an additional parameter to set a lock object name
            - This differentiates the new custom scheduler from the default during the leader election process
- The custom scheduler can then be created using the definition file and deployed to the kube-system namespace
- From here, pods can be created and configured to be scheduled by a particular scheduler by adding the field schedulerName to its definition file
    - Note: Any pods created in this manner to be scheduled by the custom scheduler will remain in a pending state if the scheduler wasn't configured correctly
- To confirm the correct scheduler picked the pod up, use **kubectl get events**
- To view the logs associated with the scheduler, run:

```
kubectl logs <scheduler-name> --name-space=kube-system
```

Configuring Kubernetes Scheduler

- Schedulers can be configured manually or set up via kubeadm
- Additional schedulers can be created via yaml files, which can then be configured with naming and identifying the "leader" of the schedulers for high-availability setups
- Advanced options are available, but are outside the scope of the course

11 January 2021

04 - Logging & Monitoring

Monitor Cluster Components

- Suppose we want to monitor performance metrics relating to resource consumption across a cluster or at a pod level, where we can analyse it
- There isn't any built-in solution for Kubernetes that satisfied this, but plenty of third-party options like Prometheus and Dynatrace are available
- Heapster was one of the first solutions for this, but is now deprecated
    - Slimmed down version available via Metrics Server
    - One metrics server allowed per cluster
    - It's an in-memory solution, data isn't persisted to system storage
- To generate the metrics, a sub-component of the kublet, known as the container advisor (c-advisor) collects the metrics from the pods and exposes them via the Kubelet API
- If using minikube, the metrics-server can be deployed via: **minikube addons enable metrics-server**
- For all other kubernetes setups, the metrics server is enabled via downloading and applying the yaml files from the associated Github repository
- To collect metrics about a particular object, run: **kubectl top <object>**
    - Objects that can be monitored include pods, nodes and more

Managing Application Logs

- When running a docker container in the background, one can view the associated logs of a container by running **docker logs -f <container ID>**
- For kubernetes, run **kubectl logs -f <pod name>** for a standalone container
    - For a pod with multiple containers, you must specify the container name you want to view, append this to the end of the above command

05 - Application Lifecycle Management

Rolling Updates and Rollbacks

Rollout and Versioning:

- When first creating a deployment, a rollout is triggered
- Rollout is equivalent to a deployment revision in definition

- When future updates occur, a new rollout will occur creating a new deployment revision
- This functionality allows tracking of deployment changes
    - Rollback functionality is therefore available in the event of application failure

Rollout Commands:

- To view rollout status:
    - **kubectl rollout status <deployment name>**
- To view rollout history and versioning:
    - **kubectl rollout history <deployment-name>**

Deployment Strategy:

- There are multiple deployment strategies available, the two main versions are:
    - Recreate:
        - When a new version of an application is ready, tear down all instances currently deployed at once
        - Deploy new versions once “current” version is unavailable
        - Results in significant user downtime
    - Rolling Updates:
        - Destroys current instance and upgrades with a new version one after another (take down one old version, upload a new version, repeat)
        - Leads to higher availability of the application
        - Upgrade appears “seamless”
- To update a deployment, simply make the changes to the definition file and run **kubectl apply -f <deployment-definiton-file>.yaml**
    - This triggers a new rollout
- It should be noted, you could also update the deployment via the CLI only, for example, updating a deployment’s image:
    - **kubectl set image deployment <deployment-name> <image>=<image>:<tag>**
        - This method doesn’t update the YAML file associated with the deployment

- You can view deployment strategies in detail via the **kubectl describe deployment <deployment-name>**
    - For the recreate strategy, it can be seen that during the upgrade process the deployment is scaled from maximum size, to zero, then back again
    - For rolling updates, the pods are scaled individually, one old pod removed, one new pod added, and so on.
- When a new deployment is created, a new replicaset is automatically created, hosting the desired number of replica pods
- During an upgrade a new replica set is created
    - New pods with new application added sequentially
    - At the same time, the new old pods are sequentially taken down
- Once upgraded, if a rollback is required, run: **kubectl rollout undo <deployment>**
- The command **kubectl run <deployment> --image=<image>** will create a deployment
    - Serves as an alternative to using a definition file
    - Required replicasets and pods automatically created in the backend
    - It's still highly recommended to use a definition file for editing and versioning deployments
- Command Summary:
    - To create a deployment from a yaml file:
        - **kubectl create -f <deployment.yaml>**
    - To get a list of all deployments
        - **kubectl get deployments**
    - To update a deployment, run one of the following two:
        - **kubectl apply -f <deployment.yaml>**
        - **Kubectl set image <deployment> <image ID>=<image>:<tag>**
    - To get the status of a deployment rollout:
        - **kubectl rollout status <deployment>**
    - To view the rollout history:
        - **kubectl rollout history <deployment>**
    - To rollback:
        - **kubectl rollout undo <deployment>**

Commands & Arguments - Docker

- Note: This isn't a requirement for the CKAD/CKA curriculum
- Consider a simple scenario:
    - Run an ubuntu image with Docker: **docker run ubuntu**
    - This runs an instance of the Ubuntu image and exits immediately
- If **docker ps -a** is ran to list the containers, it won't appear
    - Due to the fact that containers aren't designed to host an OS
    - They're instead designed to run a specific task/process e.g. host a web server
    - The container will exist as long as the hosted process is active, if the service is stopped or crashes, the container exits
- The Dockerfile sCMD section was set as "bash"
    - This isn't a command, but a CLI instead
    - When the container ran, Docker created a container based on the Ubuntu image and launched bash
- Note: By default, Docker doesn't attach a terminal to a container when it's ran
    - Bash cannot find a terminal
    - Container exits as the process is finished
- To solve a situation like this, you can add container commands to the docker run command, e.g.
    - **docker run ubuntu sleep 5**
- These changes can be made permanent via editing the Docker file either in a:
    - Shell format: **CMD command param1**
    - JSON format: **CMD ["command", "param1"]**
- To build the new image, run: **docker build -t image_name .**
- Run the new image via **docker run <image_name>**
- To use a command but with a different value of parameter to change each time, change the CMD to "ENTRYPOINT" i.e. **ENTRYPOINT ["command"]**
- Any parameters specified on the CLI will automatically be appended to the entrypoint command
- If using entrypoint and a command parameter isn't specified, an error is likely
    - A default value should be specified

- To overcome the problem, use a combination of entrypoints and command in a JSON format i.e.:
    - **ENTRYPOINT [“command”]**
    - **CMD [“parameter”]**
- From this configuration, if no additional parameter is provided, the CMD parameter will be applied
- Any parameter on the CLI will override the CMD parameter
- To override the entrypoint command, run:
    - **docker run --entrypoint <new command> <image name>**

Commands and Arguments - Kubernetes

- Using the previously defined image, one can create a yaml definition file:

```
apiVersion: v1
Kind: Pod
metadata:
        name: pod-name
spec:
        containers:
                name: container-name
                image: container-image
                command: [“new command”]
                args: [“10”]
```

- To add anything to be appended to the docker run command for the container, add **args** to the container spec
- Pod may then be created using the **kubectl create -f** command as per
- To override entrypoint commands, add “command” field to the pod spec
- To summarise, in Kubernetes Pod Specs:
    - command overrides Dockerfile entrypoint commands
    - args override Dockerfile CMD commands
- Note: You cannot edit specifications of a preexisting pod aside from:
    - Containers

- Initcontainers
- activeDeadlineSeconds
- Tolerations

Configure Environment Variables in Applications

- For a given definition file, one can set environment variable via the env field under containers in a pod's spec
- Each environment variable is an array, so each one has its own name, value and is denoted by a - prior to the name field

```
env:
- name: APP_COLOR
  value: blue
```

- Environment variables can also be referenced via two other methods:
  - Configmaps: rather than env, replace with **valueFrom**, and add **configMapKeyRef** underneath
  - Secrets: rather than env, replace with **valueFrom**, and add **secretKeyRef** underneath

Configure ConfigMaps in Applications

- When there are multiple pod definition files, it becomes difficult to manage environmental data
- This can info can be removed from the definition files and managed centrally via ConfigMaps
- ConfigMaps used to pass configuration data as key-value pairs in Kubernetes
- When a pod is created, the configmap's data can be injected into the pod, making the kvps available as environmental variables for the application within the container
- Configuring the ConfigMap involves 2 phases:
  - Create the ConfigMap

    - Inject it to the Pod
- To create, can run either:
    - **kubectl create configmap <configmap name>**
    - **kubectl create -f <configmap-definition>.yaml**
- By using the first command specified above, you can immediately create key-value pairs:

```
kubectl create configmap <configmap-name> --from-literal=<key>=<value> …
--from-literal<key N>=<value N>
```

-

Configure Secrets in Applications

See written CKAD notes section 3.8

Scale Applications

See section on Rolling Updates and Rollbacks

Multi-Container Pods

See section 4.1 of CKAD written notes

Multi Container Pod Design Patterns

Ditto section above

InitContainers

- In multi-container pods, each container will run a process that stays alive for the duration of the pod's lifecycle
- Example: Consider a multi-container pod that runs a web application in one container, and a logging agent in another
    - Both containers are expected to stay running at all times (the log agent runs as long as the web application is running)
    - If either fails, the pod stops

- In some scenarios, would want to run a process that runs to completion in a container e.g. a process that pulls a code or binary from a repository to be used by the main application
    - Processes like this are to be ran only one time when the pod is first created, or to wait for an external service or database to be up and running before the application starts
- Containers like this are initContainers
- InitContainers are configured in the exact same way as regular containers in a pod's spec, they are just a separate section to containers
    - When the POD is first created, the initContainer runs its progress to completion before the main application starts
- Multiple initContainers can be configured in a similar manner to multi-container pods
    - If this is the case, each initContainer will run one at a time
    - In the event any of the initContainers fail, the pod will repeatedly restart until they all succeed

Self Healing Applications

- Self-healing applications are supported through the use of ReplicaSets and Replication Controllers
- ReplicationControllers help in ensuring a pod is recreated automatically when the application within crashes; thus ensuring enough replicas of the application are running at all times
- Additional support to check the health of applications running within pods and take necessary actions when they're unhealthy, this is done through Readiness and Liveness Probes.
- For more information, see section 5.0-5.2 of the written CKAD notes

12 January 2021

06 - Cluster Maintenance

OS Upgrades

- Suppose you have a cluster with a few nodes and pods serving applications; what happens if one of these nodes goes down?
    - Associated pods are rendered inaccessible
    - Depending on the deployment method of these PODs, users may be impacted
- If multiple replicas of the pod are spread across the cluster, users are uninterrupted as it's still accessible
    - Any pods running ONLY on that node however will experience downtime
- Kubernetes will automatically try and restart the node
    - If it comes back on immediately, kubectl restarts and the pods restart
    - If after 5 mins and it's not back online, Kubernetes considers the pods as dead and terminates them from the node
        - If part of a replicaset, the pods will be recreated on other nodes
- The time it takes for a pod to come back online is the pod eviction timeout
    - Can be set on the controller manager via: **kube-controller-manager --pod-eviction-timeout=xmys**
        - X,y = integer values
- If the node comes back online after the timeout it restarts as a blank node, any pods that were on it and not part of a replicaset will remain "gone"
- Therefore, if maintenance is required on a node that is likely to come back within 5 minutes, and workloads on it are also available on other nodes, it's fine for it to be temporarily taken down for upgrades
    - There is no guarantee that it'll reboot within the 5 minutes
- Nodes can be "drained", a process where they are gracefully terminated and deployed on other nodes
    - Done so via: **kubectl drain <node name>**
    - Node cordoned and made unschedulable
- To uncordon node: **kubectl uncordon <nodename>**
- To mark the node as unschedulable, run: **kubectl cordon <nodename>**
    - Doesn't terminate any preexisting pods, just stops any more from being scheduled
- Note: May need the flag **--ignore-daemonsets** and or **--force**

Kubernetes Software Versions

- When installing a kubernetes cluster, a specific version of kubernetes is installed alongside
- Can be viewed via **kubectl get nodes** in the version column
- Release versions follow the process **major.minor.patch**
- Kubernetes is regularly updated with new minor versions every couple of months
- Alpha and beta versions also available
- Alpha - Features disabled by default, likely to be buggy
- Beta - Code tested, new features enabled by default
- Stable release - Code tested, bugs fixed
- Kubernetes releases found in a tarball file in Github; contains all required executables of the same version
- Note: Some components within the control plane will not have the same version numbers and are released as separate files; ETCD cluster and CoreDNS servers being the main examples

13 January 2021

Cluster Upgrade Process

- The kubernetes components don't all have to be at the same versions
    - No component should be at a version higher than the kube-api server
    - If Kube-API Server is version X (a minor release), then the following ranges apply for the other components for support level:
        - Controller manager: X-1
        - Kube-Scheduler: X-1
        - Kubelet: X-2
        - Kube-Proxy: X-2
        - Kubectl: X-1 - X+1
- At any point, Kubernetes only supports the 3 most recent minor releases e.g. 1.19 - 1.17
- It's better to upgrade iteratively over minor processes e.g. 1.17 - 1.18 and so on

- Upgrade process = Cluster-Dependent
- If on a cloud provider, built-in functionality available
- If on kubeadm/manually created cluster, must use commands:
    - **kubeadm upgrade plan**
    - **kubeadm upgrade apply**
- Cluster upgrades involve two steps:
    - Upgrade the master node
        - All management components go down temporarily during the processes
        - Doesn't impact the current node workloads (only if you try to do anything with them)
    - Upgrade the worker nodes
        - Can be done all at once - Results in downtime
        - Can be done iteratively - Minimal downtime by draining nodes as they get upgraded one after another
        - Could also add new nodes with the most recent software versions
            - Proves especially inconvenient when on a cloud provider
- Upgrading via Kubeadm:
    - kubeadm upgrade plan
        - Lists latest versions available
        - Components that must be upgraded manually
        - Command to upgrade kubeadm
    - Note: kubeadm itself must be upgraded first: **apt-get upgrade -y kubeadm=major.minor-patch_min-patch_max**
- Check upgrade success based on CLI output and kubectl get nodes
- If Kubelet is running on Master node, this must be upgraded next the master node and restart the service:
    - **Apt-get upgrade -y kubelet=1.12.0-00**
    - **systemctl restart kubelet**
- Upgrading the worker nodes:
    - Use the drain command to stop and transfer the current workloads to other nodes, then upgrade the following for each node (ssh into each one):
        - Kubeadm

        - Kubelet
        - Node config: kubeadm upgrade node config --kubelet-version major.minor.patch
    - Restart the service: **systemctl restart kubelet**
- Make sure to uncordon each node after each upgrade!

Backup and Restore Methods

- It's good practice to save resource configuration definition files
- Kube-api server can be used to query all resources to get yaml files for each
- E.g. kubectl get all --all-namespaces -o yaml > filename.yaml
- The etcd cluster stores information about the state of the cluster e.g. what nodes are on it and what applications are they running
- When configuring the etcd, you can configure the data directory for the etcd data store via the --data-dir flag
- You can take a snapshot of the etcd database using the etcdctl utility
- To restore the cluster from the backup:
    - Service kube-apiserver stop
    - Etcdctl snapshot restore snapshot.db --options
    - New data store directory created
    - The etcd service file must then be reconfigured for the new cluster token and data directory
    - Reload the daemon and restart the service
- Backup candidates:
    - Kube-API Server query - Generally the more common method
    - ETCD Server

Disaster Recovery with ETCD in Kubernetes:

Assuming ETCDCTL is installed, use it to take a snapshot, make sure to specify the flags, which can all be found via examining the etcd pod and ARE MANDATORY for authentication:

```
ETCDCTL_API=3 etcdctl --endpoints=https://[127.0.0.1]:2379
--cacert=/etc/kubernetes/pki/etcd/ca.crt \
```

```
     --cert=/etc/kubernetes/pki/etcd/server.crt
--key=/etc/kubernetes/pki/etcd/server.key \
     snapshot save PATH/TO/BACKUP/BACKUP.db
```

Suppose disaster happens:

Restore the snapshot to a new folder:

```
ETCDCTL_API=3 etcdctl  --data-dir /var/lib/etcd-from-backup \

     snapshot restore PATH/TO/BACKUP/BACKUP.db
```

Update the etcd pod's volume hostpath and mount paths for etcd-data to be /var/lib/etcd-from-backup etc as appropriate by updating the yaml file at `/etc/kubernetes/manifests/etcd.yaml`

The etcd pod should automatically restart once this update is done, bringing back the pods stored in the backup along with it. (Use **watch "docker ps | grep etcd"** to track)

Working with ETCDCTL

- For backup and restore purposes, make sure to set the ETCDCTL API to 3: **export ETCDCTL_API=3**
- For taking a snapshot of the etcd cluster: **etcdctl snapshot save -h** and keep a note of the mandatory global options.
- For a TLS-Enabled ETCD Database, the following are mandatory:
    - --cacert
    - --cert
    - --endpoints[IP:PORT]
    - --key
- Use the snapshot restore option for backup: **etcdctl snapshot restore -h**
    - Note options available and apply as appropriate

07 - Security

Kubernetes Security Primitives

- Controlling access to API Server is the top priority

- Need to define who can access the API Server through and what they can do
- Could use any of:
    - Files
    - Certificates
    - External authentication providers
    - Service Accounts
- By default, all pods within a cluster can access one another
- This can be restricted via the introduction of network policies

Authentication

- Authentication Mechanisms Available in Kubernetes for the API Server:
    - Static Password File
    - Static Token File
    - Certificates
    - Identity Services
- Suppose you have the user details in a file, you can pass this file as an option for authentication in the kube-apiserver.service file adding the flag:
  --basic-auth-file=user-details.csv
    - Restart the service after this change is done
- If the cluster is setup using Kubeadm, edit the yaml file and add the same option
    - Kuberntes will automatically update the apiserver once the change is made
- To authenticate using the credentials specified in this manner run a curl command similar to:

  curl -v -k https://master-node-ip:6443/api/v1/pods -u "username:password"

- Could also have a token file, specify using --token-auth-file=user-details.csv
- Note: These are not recommended authentication mechanisms
    - Should consider a volume mount when providing the auth file in a kubeadm setup
    - Setup RBAC for new users
- Could also setup RBAC using YAML files to create rolebindings for each user

17 January 2021

TLS Basics

- Certificates used to guarantee trust between two parties communicating with one another, leading to a secure and encrypted connection
- Data involved in transmission should be encrypted via the use of encryption keys
- Encryption Methods:
    - Symmetric: Same key used for encryption and decryption
    - Asymmetric encryption: A public and private key are used for encryption and decryption specifically
        - Private key can only be used for decryption
- SSH Assymetric Encryption: run **ssh-keygen**
    - Creates **id_rsa** and **id_rsa.pub** (public and private keys)
    - Servers can be secured by adding public key to authorized key file at **~/.ssh/authorized_keys**
    - Access to the server is then allowed via ssh -i id_rsa username@server
    - For the same user, can copy the public key to any other servers
- To securely transfer the key to the server, use asymmetric encryption
- Can generate keys with: **openssl genrsa -out <name>.key 1024**
    - Can create public variant with: **openssl rsa -in <name>.key -pubout > <name>.pem**
- When the user first accesses the web server via HTTPS, they get the public key from the server
    - Hacker also gets a copy of it
- The users browser encrypted the symmetric key using the public key
    - Hacker gets copy
- Server uses private key to decrypt symmetric key
    - Hacker doesn't have access to the private key, and therefore cannot encrypt it.
- For the hacker to gain access, they would have to create a similar website and route your requests
    - As part of this, the hacker would have to create a certificate
    - In general, certificates must be signed by a proper authority
    - Any fake certificates made by hackers must be self-signed

        - Web browsers have built-in functionalities to verify if a connection is secure
- To ensure certificates are valid, the Certificate Authorities (CAs) must sign and validate the certs.
    - Examples: Symanteg, Digicert
- To validate a certificate, one must first generate a certificate validation request to be sent to the CA: **openssl req -new -key <name>.key -out <<name>.csr -subk "/C=US/ST=CA/O=MyOrg, Inc./CN=mybank.com"**
- CAs have a variety of techniques to ensure that the domain is owned by you
- How does the browser know what certificates are valid? CAs have a series of public and private keys built in to the web browser, the public key is then used for communication between the browser and CA to validate the certificates
- Note: The above are described for public domain websites
- For private websites, such as internal organisation websites, private CAs are generally required and can be installed on all instances of the web browser within the organisation
- Note:
    - Certificates with a public key are named with the extension .crt or .pem, with the prefix of whatever it is being communicated with
    - Private keys will have the extension of either **.key** or **-key.pem**

TLS In Kubernetes

- In the previous section, three types of certificates were discussed, for the purposes of discussing them in Kubernetes, how they're referred to will change:
    - Public and Private Keys used to secure connectivity between the likes of web browsers and servers: **Server Certificates**
    - Certificate Authority Public and Private Keys for signing and validating certificates: **Root Certificates**
    - Servers can request a client to verify themselves: **Client Certificates**
- Note:
    - Certificates with a public key are named with the extension .crt or .pem, with the prefix of whatever it is being communicated with
    - Private keys will have the extension of either **.key** or **-key.pem**

- All communication within a Kubernetes cluster must be secure, be it pods interacting with one another, services with their associated clients, or accessing the APIServer using the Kubectl utility
- Secure TLS communication is a requirement
- Therefore, it is required that the following are implemented:
- Server Certificates for Servers
    - Client Certificates for Clients
- **Server Components:**
    - Kube-API Server
        - Exposes an HTTPS service that other components and external users use to manage the Kubernetes cluster
        - Requires certificates and a key pair
            - apiserver.crt and apiserver.key
    - ETCD Server
        - Stores all information about the cluster
        - Requires a certificate and key pair
            - Etcdserver.crt and apiserver.key
    - Kubelet server:
        - Exposes HHTPS API Endpoint that the Kube-API Server uses to interact with the worker nodes
- **Client Certificates:**
    - All of the following require access to the Kube-API Server
    - Admin user
        - Requires certificate and key pair to authenticate to the API Server
        - admin.crt and admin.key
    - Scheduler
        - Client to the Kube-APIServer for object scheduling pods etc
            - scheduler.crt and scheduler.key
    - Kube-Controller:
        - controller-manager.crt and controller-manager.key
    - Kube-Proxy
        - kube-proxy.crt and kube-proxy.key

- Note: The API Server is the only component that communicates with the ETCD server, which views the API server as a client
    - The API server can use the same keys as before for serving itself as a service OR a new pair of certificates can be generated specifically for ETCD Server Authentication
    - The same principle applies for the API Server connecting to the Kubelet service
- To verify the certificates, a CA is required. Kubernetes requires at least 1 CA to be present; which has its own certificate and key (**ca.crt** and **ca.key**)

TLS In Kubernetes - Certificate Creation

- Tools available for certificate creation include:
    - EASYRCA
    - OPENSSL - The more common one
    - CFSSL
- **Steps - Server Certificates: CA Example**
    - Generate the keys: **openssl genrsa -out ca.key 2048**
        - The number "2048" in the above command indicates the size of the private key. You can choose one of five sizes: 512, 758, 1024, 1536 or 2048 (these numbers represent bits). The larger sizes offer greater security, but this is offset by a penalty in CPU performance. We recommend the best practice size of 1024.
    - Generate certificate signing request: **openssl req -new -key ca.key -subj "/CN=KUBERNETES-CA" -out ca.csr**
    - Sign certificates: **openssl x509 -req -in ca.csr -signkey ca.key -out ca.crt**
- **Client Certificate Generation Steps: Admin User Example**
    - Generate the keys: **openssl genrsa -out admin.key 2048**
    - Generate certificate signing request: **openssl req -new -key admin.key -subj "/CN=kube-admin" -out admin.csr**
    - Sign the certificate: **openssl x509 -req -in admin.csr -CAkey ca.key -out admin.crt**
        - The CA key pair is used for signing the new certificate, thus proving its validity

- When the admin user attempts to authenticate to the cluster, it is the certificate admin.crt that will be used for this
- For non-admin users, need to add group details to the certificate signing request to signal this.
    - Group called SYSTEM:MASTERS has administrative privileges, to specify this for an admin user, the signing request should be like: **openssl req -new -key admin.key -subj "/CN=kube-admin/O=system:masters" -out admin.csr**
- The same procedure would be followed for client certificates for the Scheduler, Controller-Manager and Kubelet
- The certificates generated could be applied in different scenarios:
    - Could use the certificate instead of usernames and password in a REST API call to the api server (via a curl request).
        - To do so, specify the key and the certs involved as options following the request e.g. --key admin.key --cert admin.crt and --cacert ca.crt
    - Alternatively, all the parameters could be moved to the kube config yaml file, which acts as a centralized location to reference the certificates
- Note: For each of the Kubernetes components to verify one another, they need a copy of the CA's root certificate
- **Server Certificate Example: ETCD Server**
    - As ETCD server can be deployed as a cluster across multiple servers, you must secure communication between the cluster members, or peers as they're commonly known as.
    - Once generated, specify the certificates when starting the server
    - In the etcd yaml file, there are options to specify the peer certificates
- **Server Certificate Example: API Server:**
    - Same procedure involved, but due to the nature of the API Server, with essentially every operation running via it, and it being referred to by multiple names, requires a lot more information included; requiring an openssl config file
        - Using the config file, specify DNS and IP aliases for the component
        - When generating the signing request, you can reference the config file by appending: --config <config name>.cnf to the signing request command

    - The location of all certificates are passed into the exec start file for the api server or the service's configuration file, specifically:
        - Etcd:
            - CA File
            - ETCD Certificate
            - ETCD Private Key File
        - Kubelet
            - CA File
            - Client Certificate
            - Client Key
        - Client CA
        - API Server Cert and Private Key
- Kubelet Server:
    - Key-Certificate pair required for each worker node
    - Named after each node
    - Must be referenced in the kubelet config file for each node, specifically:
        - Client CA file
        - TLS Certificate file (kubelet-node01.crt for example)
        - TLS Private Key File (kubelet-node01.key for example)
    - Client certificates used to authenticated to the Kube API Server
        - Naming convention should be system:node:nodename
        - Nodes must also be added to **system:nodes** group for associated privileges

View Certificate Details

- The generation of certificates depends on the cluster setup
    - If setup manually, all certificates would have to be generated manually in a similar manner to that of the previous sections
        - Components deployed as native services in this manner
    - If setup using a tool such as kubeadm, this is all pre-generated
        - Components deployed as pods in this manner
- For Kubeadm clusters:
    - Component found in **/etc/kubernetes/manifests/** folder

        - Certificate file paths located within component's yaml files
        - Example: **apiserver.crt**
        - Use **openssl x509 -in /path/to/.crt file -text -noout**
    - Can check the certificate details such as name, alternate names, issuer and expiration dates
- Note: Additional details available in the documentation for certificates
- Use **kubectl logs <podname>** on kubeadm if any issues are found with the components
- If kubectl is unavailable, use Docker to get the logs of the associated container:
    - Run **docker ps - a** to identify the container ID
    - View the logs via **docker logs <container ID>**

Certificates API

- All certificates have an expiration date, whenever the expiry happens, keys and certificates must be re-generated
- As discussed, the signing of the certificates is handled by the CA Server
- The CA server in reality is just a pair of key and certificate files generated
- Whoever has access to these files can sign any certificate for the Kubernetes environment, create as many users they want and set their permissions
- Based on the previous point, it goes without saying these files need to be protected
    - Place the files on a fully secure server
    - The server that securely hosts these files becomes the "CA Server"
    - Any time you want to sign a certificate, it is the CA server that must be logged onto/communicated with
- For smaller clusters, it's common for the CA server to actually be the master node
    - The same applies for a kubeadm cluster, which creates a CA pair of files and stores that on the master node
- As clusters grow in users, it becomes important to automate the signing of certificate requests and renewing expired certificates; this is handled via the Certificates API
    - When a certificate needs signing, a Certificate Signing Request is sent to Kubernetes directly via an API call

    - Instead of an admin logging onto the node and manually signing the certificate, they create a Kubernetes API object called CertificateSigningRequest
    - Once the API object is created, any requests like this can be seen by administrators across the cluster
    - From here, the request can be reviewed and approved using kubectl, the resultant certificate can then be extracted and shared with the user
- Steps:
    - User generates key: openssl genrsa -out <keyname>.key 2048
    - User generates certificate signing request and sends to administrator: openssl req -new -key <key>.name -subk "/CN=name" -out name.csr
    - Admin receives request and creates the API object using a manifest file, where the spec file includes the following:
        - Groups - To set the permissions for the user
        - Usages - What is the user able to do with keys with this certificate to be signed?
        - Request - The certificate signing request associated with the user, which must be encoded in base64 language first i.e. **cat cert.crt | base64**
        - Admins across the cluster can view certificate requests via: **kubectl get csr**
        - If all's right with the csr, any admin can approve the request with: **kubectl certificate approve <name>**
        - You can view the CSR in a YAML form, like any Kubernetes object by appending **-o yaml** to the **kubectl get** command
            - Note: The certificate will still be in base64 code, so run: **echo "CODED CERTIFICATE" | base64 --decode**
- Note: The controller manager is responsible for all operations associated with approval and management of CSR
- The controller manager's YAML file has options where you can specify the key and certificate to be used when signing certificate requests:
    - **--cluster-signing-cert-file**
    - **--cluster-signing-key-file**

19 January 2021

KubeConfig

- Files containing information for different cluster configurations, such as:
    - --server
    - --client-key
    - --client-certificate
    - --certificate-authority
- The existence of this file removes the need to specify the option in the CLI
- File located at **$HOME/.kube/config**
- KubeConfig Files contain 3 sections:
    - Clusters - Any cluster that the user has access to, local or cloud-based
    - Users - User accounts that have access to the clusters defined in the previous section, each with their own privileges
    - Contexts - A merging of clusters and users, they define which user account can access which cluster
- These config files do not involve creating new users, it's simply configuring what existing users, given their current privileges, can access what cluster
- Removes the need to specify the user certificates and server addresses in each kubectl command
    - --server spec listed under clusters
    - User keys and certificates listed in Users section
    - Context created to specify that the user "MyKubeAdmin" is the user that is used to access the cluster "MyKubeCluster"
- Config file defined in YAML file
    - ApiVersion = v1
    - Kind = Config
    - Spec includes the three sections defined previously, all of which are arrays
    - Under clusters: specify the cluster name, the certificate authority associated and the server address
    - Under users, specify username and associated key(s) and certificate(s)
    - Under contexts:

        - Name format: username@clustername
        - Under context specify cluster name and users
    - Repeat for all clusters and users associated
- The file is automatically read by the kubectl utility
- Use current-context field in the yaml file to set the current context
- CLI Commands:
    - View current config file being used: **kubectl config view**
        - Default file automatically used if not specified
        - To view non-default config files, append: --kubeconfig=/path/to/file
    - To update current context: kubectl config use-context <context-name>
    - Other commands available via **kubectl config -h**
    - Default namespaces for particular contexts can be added also
- Note: for certificates in the config file, use the full path to specify the location
    - Alternatively use certificate-authority-data to list certificate in base64 format

API Groups

- API Server accessible at master node IP address at port 6443
    - To get the version, append /version to a curl request to the above IP address
    - To get a list of pods, append /api/v1/pods
- Kubernetes' API is split into multiple groups depending on the group's purpose such as
    - /api - core functionalities e.g. pods, namespaces, secrets
    - /version - viewing the version of the cluster
    - /metrics - used for monitoring cluster health
    - /logs - for integration with 3rd-party logging applications
    - /apis - named functionalities added to kubernetes over time such as deployments, replicasets, extensions
        - Each group has a version, resources, and actions associated with them
    - /healthz - used for monitoring cluster health
- Use curl http://localhost:6443 -k to view the api groups, then append the group and grep name to see the subgroups within

- Note: Need to provide certificates to access the api server or use **kubectl proxy** to view
- Note: kubectl proxy is not the same as kube proxy, the former is an http proxy service to access the api server

Authorization

- When adding users, need to ensure their access levels are sufficiently configured, so they cannot make any unwanted changes to the cluster
- This applies to any physical users, like developers, or virtual users like applications e.g. Jenkins
- Additional measures must be taken when sharing clusters with organizations or teams, so that they are restricted to their specific namespaces
- Authorization mechanisms available are:
    - Node-based
    - Attribute-Based
    - Rule-Based
    - WebHook-based
- Node-Based:
    - Requests to the kube-apiserver via users and the kubelet are handled via the Node Authorizer
    - Kubelets should be part of the system:nodes group
    - Any requests coming from a user with the name system-node and is aprt of the system nodes group is authorized and granted access to the apiserver
- ABAC - Attribute-Based
    - For users wanting to access the cluster, you should create a policy in a JSON format to determine what privileges the user gets, such as namespace access, resource management and access, etc
    - Repeat for each users
    - Each policy must be edited manually for changes to be made, the kube apiserver must be restarted to make the changes take effect
- RBAC
    - Instead of associating each user with a set of permissions, can create a role which outlines a particular set of permissions

    - Assign users to the role
    - If any changes are to be made, it is just the role configuration that needs to be changed
- Webhook
    - Use of third-party tools to help with authorization
    - If any requests are made to say the APIserver, the third party can verify if the request is valid or not
- Note: Additional authorization methods are available:
    - AlwaysAllow - Allows all requests without checks
    - AlwaysDeny - Denies all requests without checks
- Authorizations set by --authorization option in the apiserver's .service or .yaml file
- Can set modes for multiple-phase authorization, use --authorization-mode and list the authorization methods

RBAC

- To create a role, create a YAML file
- Spec replaced with rules
    - Covers apiGroups, resources and verbs
- Multiple rules added by - apiGroups for each
    - Create the role using **kubectl create -f**
- To link the user to the role, need to create a **Role Binding**
- Under metadata:
    - Specify subjects - Users to be affected by the rolebinding, their associated apiGroup for authorization
    - RoleRef - The role to be linked to the subject
- To view roles: kubectl get roles
- To view rolebindings: kubectl get rolebindings
- To get additional details: kubectl describe role/rolebinding <name>
- To check access level: **kubectl auth can-i <command/activity>**
- To check if a particular user can do an activity, append **--as <username>**
- To check if an activity can be done via a user in a particular namespace, append **--namespace <namespace>**

- Note: Can restrict access to particular resources by adding resourceNames: [“resource1”, “resource2”, ...] to the role yaml file

Cluster Roles and Role Bindings

- Roles and role bindings are created for particular namespaces and control access to resources in that particular namespace
- By default, roles and role bindings are applied to the default namespace
- In general, resources such as pods, replicasets are namespaced
- Cluster-scoped resources are resources that cannot be associated to any particular namespace, such as:
    - Persistentvolumes
    - Nodes
- To switch view namespaced/cluster-scoped resources: **kubectl api-resources --namespaced=TRUE/FALSE**
- To authorize users to cluster-scoped resources, use cluster-roles and cluster-rolebindings
    - Could be used to configure node management across a cluster etc
- Cluster roles and role bindings are configured in the exact same manner as roles and rolebindings; the only difference is the kind
- Note: Cluster roles and rolebindings can be applied to namespaced resources

Image Security

- Dockers image naming convention follows:
    - <REGISTRY>/<USER/ACCOUNT>/<IMAGE/REPOSITORY>
    - Registry set to docker.io by default
    - Private registries available for users wanting to store private images via platforms like Azure and AWS
    - To login to a private registry: docker login <private-registry name>
    - To pull an image from a private registry:
        - Image name: <private registry name>/<repo>/<image name>
    - To handle the login to the private registry, need to create a secret for the credentials:

- **kubectl create secret docker-registry <secret-name> --docker-server=<server name> --docker-username=<USERNAME> --docker-password=<password> --docker-email=<user@org.com>**
- This secret can then be referenced in the pod spec as a sibling of containers, listing imagePullSecret, then - name: <secret name> underneath

Security Contexts

- When running docker containers, can specify security standards such as the ID of the user to run the container
- The same security standards can be applied to pods and their associated containers
- Configurations applied a pod level will apply to all containers within
- Any container-level security will override pod-level security
- To add security contexts, add securityContext to either or both the POD and Container specs; where user IDs and capabilities can be set

23 January 2021

Network Policy

- See CKAD section 7.4

Developing Network Policies

- When developing network policies for pods, always consider communication from the pods perspective
- PolicyTypes Available:
  - Ingress - Incoming traffic
  - Egress - Outgoing traffic
    - Both can be implemented if desired
- Each ingress rule has a from and ports field:
  - From describes the pods which the pod affected by the policy can accept ingress communication
  - For additional specification, can use podSelector and namespaceSelector or ipBlock to specify particular IP addresses

    - Each rule start denoted by -
    - Ports - Network ports communication can be received from
- For egress rules, the only difference is “from” is replaced with “to”

08 - Storage

Storage in Docker

For storage in Docker, must consider both Storage and Volume Drivers.

- Docker file system setup at /var/lib/docker
    - Contains data relating to containers, images, volumes, etc.
- To ensure data in a container is stored, create a persistent volume:
    - **Docker volume create <volume>**
    - The volume can then be mounted into a container: **docker run -v data_volume:/path/to/volume <container>**
    - Note: if a volume hasn’t been already created before this run command, docker will automatically create a volume of that name at the path specified
- For mounting a particular folder to a container, replace <data_volume> or whatever named with the full path to the folder you want to mount
- Alternative command: --mount type=<type>,source=<source>,target=<container volume path> container
- Operations like this, maintaining a layered architecture etc. is handled by storage drivers such as AUFS, BTRFS, Overlay2, Device Mapper
- Docker chooses the best storage driver given the OS and application

Volume Driver Plugins in Docker

- Default volume driver plugin = local
- Alternatives available include:
    - Azure File Storage
    - GCE-Docker
    - VMware vSphere Storage
    - Convoy

- To specify the volume driver, append --volume-driver <drivername> to the docker run command

Container Storage Interface (CSI)

- CRI = Container Runtime Interface
    - Configures how Kubernetes interacts with container runtimes, such as Docker
- CNI - Container Network Interface
    - Sets predefined standards for networking solutions to work with Kubernetes
- CSI - Container Storage Interface
    - Sets standards for storage drivers to be able to work with kubernetes
    - Examples include Amazon EBS, Portworx
- All of the above allow any container orchestration to work with drivers available

Volumes

See section 8.1 of CKAD

Persistent Volumes

See section 8.2 of CKAD

Persistent Volume Claims

See section 8.3 of CKAD

Storage Class

- Allows definition of a provisioner, so that storage can automatically be provisioned and attached to pods when a claim is made
- Make storage classes using yaml files to define a particular storage class
- To use the storage class, specify it in the pvc definition file
- Still creates a PV, BUT doesn't require a definition file
- Provisioners available include:
    - AWS
    - GCP

    - Azure
    - ScaleIO
- Additional configuration options available for different provisioners, so you could have multiple storage classes per provisioners

24 January 2021

09 - Networking

Prerequisite - Switching Routing

- To connect two hosts to one another, need to connect them to a switch, which creates a network connection
- Need an interface on the host, viewable via **ip link**
- Assign the system with IP addresses of the same network: ip addr add <IP> <namespace> <networkname>
- For systems on other networks, need a router for inter-switch communication
    - Has an IP address for each network that it can communicate with
- Gateway - Setup to help route requests to a particular location, view via **route**
    - Add via ip route add <IP> via <IP>
- For the internet, can set default gateway so any requests to a network outside of the current can be sent to the internet - **ip route add default via <Router IP>**
- If multiple routers, entries required for each to setup gateway
- Ip route add <IP> via <IP>
- To check connection - **ping <IP>**
- Whether data is forwarded is defined by **/proc/sys/net/ipv4/ip_forward** (set to 1 by default)
- **ip link** - List and modify interfaces on the host
- **Ip addr** - see ip addresses assigned to interfaces described in ip link
- **Ip addr add** - Add IP addresses to interface
- **Note: Any changes made via these commands don't persist beyond a restart, to ensure they do, edit the /etc/network/interfaces file**
- **Ip route (or just route)** - View routing table
- **Ip route add** - add entries into the ip routing table

Prerequisite - DNS

- Used to assign text names to ip addresses, saving the need to remember manual ip addresses
- Can assign names in /etc/hosts, write the IP and name in key-value pairs
- Note: No checks would be done by the system when done in this manner to check the hostname
- As environments grow, modifying the /etc/hosts becomes impossible
- Moved to Domain Name Server for centralized management
    - Host will point to the DNS server to resolve any names unknown to them
- For any changes that need to be made, just the one change needs to be made in the DNS server, all hosts will register it
- Note: custom entries can still be added in the /etc/hosts file, though this is better for local networking
- If both the DNS and the /etc/hosts file contains the same IP address for an entry, it looks in the /etc/hosts file first, then DNS, taking whichever one comes first
- Record types:
    - A - Domain Name - IP address
    - AAAA - Domain name to full Address
    - CNAME - 1-to-1 name mapping for the same IP
- Dig - tool to test DNS resolution (dig <DNS NAME>)

Prerequisite - CoreDNS

We are given a server dedicated as the DNS server, and a set of Ips to configure as entries in the server. There are many DNS server solutions out there, in this lecture we will focus on a particular one – CoreDNS.

So how do you get core dns? CoreDNS binaries can be downloaded from their Github releases page or as a docker image. Let's go the traditional route. Download the binary using curl or wget. And extract it. You get the coredns executable.

Run the executable to start a DNS server. It by default listens on port 53, which is the default port for a DNS server.

Now we haven't specified the IP to hostname mappings. For that you need to provide some configurations. There are multiple ways to do that. We will look at one. First we put all of the entries into the DNS servers /etc/hosts file.

And then we configure CoreDNS to use that file. CoreDNS loads it's configuration from a file named Corefile. Here is a simple configuration that instructs CoreDNS to fetch the IP to hostname mappings from the file /etc/hosts. When the DNS server is run, it now picks the Ips and names from the /etc/hosts file on the server.

CoreDNS also supports other ways of configuring DNS entries through plugins. We will look at the plugin that it uses for Kubernetes in a later section.

Prerequisite - Network Namespaces

- Used to implement network isolation
- Resources within a namespace can only access other resources within their namespace
- Want containers to remain isolated when running a process; run in a namespace
    - Underlying host sees all processes associated with other containers
- Creating a new namespace: ip netns add <namespace name>
- To view namespaces: ip netns
- To execute a command in a namespace: ip netns exec <namespace> <command>

While testing the Network Namespaces, if you come across issues where you can't ping one namespace from the other, make sure you set the NETMASK while setting IP Address. ie: 192.168.1.10/24

ip -n red addr add 192.168.1.10/24 dev veth-red

Another thing to check is FirewallD/IP Table rules. Either add rules to IP Tables to allow traffic from one namespace to another. Or disable IP Tables all together (Only in a learning environment).

Prerequisite - Docker Networking

- Run a docker container without attaching it to a network (specified by none parameter) - **docker run --network none <containername>**
- Attach a container to a host's network - docker run --network host <containername>
    - For whatever the port the container runs on, it will be available on the same port at the hosts IP address (localhost)
- Setup a private internal network which the docker host and containers attach themselves to: docker run <containername>
- When docker's installed it creates a default network called bridge (when viewed by docker) and docker0 (when viewed via ip link)
- Whenever docker run <containername> is ran, it creates its own private namespace (viewable via ip netns) and docker inspect <namespace>
- Port mapping:
    - For a container within the private network on the host, only the containers within the network can view it
    - To allow external access, docker provides a port mapping option: appending -p <hostport>:<containerport> to the docker run command

Prerequisite - CNI

- A single program encompassing all the steps required to setup a particular network type, for example bridge add <container ns> /path/
- CNI Defines a set of standards that define how programs should be developed to solve and perform network operations with containers
- Any variants developed in line with the CNI are plugins
- Docker doesn't use CNI, instead adopting CNM (container network model)
    - Can't use certain CNI plugins with Docker instantly, instead would have to create a none network container, then manually configure CNI features

Cluster Networking

- Each node within a cluster must have at least 1 interface connected to a network
- Each node's interface must have an IP address configured
- Hosts must each have a unique hostname and unique MAC address
    - Particularly important if cloning VMs
- Ports need to be opened:
- APIServer (Master Node) - Port 6443
- Kubelet (Master and Worker) - Port 10250
- Kube-Scheduler (Master) - Port 10251
- Kube-Controller-Manager - Port 10252
- ETCD - Port 2379
- Note: 2380 In addition for the case where there are multiple master nodes (allows ETCD Clients to communicate with each otHER

Note re CNI and CKA Exam

In the upcoming labs, we will work with Network Addons. This includes installing a network plugin in the cluster. While we have used weave-net as an example, please bear in mind that you can use any of the plugins which are described here:

https://kubernetes.io/docs/concepts/cluster-administration/addons/

https://kubernetes.io/docs/concepts/cluster-administration/networking/#how-to-implement-the-kubernetes-networking-model

In the CKA exam, for a question that requires you to deploy a network addon, unless specifically directed, you may use any of the solutions described in the link above.

However, the documentation currently does not contain a direct reference to the exact command to be used to deploy a third party network addon.

The links above redirect to third party/ vendor sites or GitHub repositories which cannot be used in the exam. This has been intentionally done to keep the content in the Kubernetes documentation vendor-neutral.

At this moment in time, there is still one place within the documentation where you can find the exact command to deploy weave network addon:

https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/#steps-for-the-first-control-plane-node (step 2)

Pod Networking

- Inter-pod communication is hugely important in a fully operational environment for Kubernetes
- At the time of writing, there is no built-in solution in Kubernetes for this, but the requirements have been clearly identified:
    - Each pod should have an IP address
    - Each pod should be able to communicate with every other pod on the same node
    - Every pod should be able to communicate with pods on other nodes without NAT
- Solutions available include weaveworks, VMware etc

Example: Configuring Pod Networking

Consider a cluster containing 3 identical nodes. The nodes are part of an external network and have IP addresses in the 192.168.1 series (11, 12 and 13).

When containers are created on pods, Kubernetes creates network namespaces for each of them, to enable communication between containers, can create a bridge network and attach the containers to it. Running **ip link add v-net-0 type bridge** on each node, then bring them up with **ip link set dev v-net-0 up**

IP addresses can then be assigned to each of the bridge interfaces of networks. In this case, suppose we want each personal network to be on its own subnet (/24). The Ip address for the bridge interface can be set from here via **ip addr add 10.244.1.1/24 dev v-net-0** etc

The remaining steps can be summarised in a script that is to be ran every time a new container is created.

#create veth pair

ip link add ….

#attach veth pair

Ip link set ….

Ip link set …..

#assign IP address

Ip -n <namespace> addr add ….

Ip -n <namespace> addr add ….

#bring up the interface

Ip -n <namespace> link set

This script is run for the second container involved in the pair, with its respective information applied; allowing the two containers to communicate with one another. The script is then copied and run on the other nodes; assigning IP addresses and connecting their containers to their own internal networks.

This solves the first problem, all pods get their own IP address and can communicate with each other within their own nodes. To extend communication across nodes in the cluster, create an ip route to each nodes' internal network via the nodes' IP address i.e. on each node, run: **ip route add <pod network ip> via <node IP>**

For larger, more complex network, it's better to configure these routes via a central router, which is then used as a default gateway. Additionally, we don't have to run the script manually for each pod, this can automatically be done via the CNI as it sets out predefined standards and how the script/operations look. The script needs a bit of tweaking to consider container creation and deletion.

--cni-conf-dir=/etc/cni/net.d

--cni-bin-dir=/etc/cni/bin

./net-script.sh add <container> <namespace>

27 January 2021

CNI in Kubernetes

- CNI Defines the best practices and standards that should be followed when networking containers and the container runtime
- Responsibilities include:
    - Creating namespaces
    - Identifying the network a container should attach to
    - Invoke the associated network plugin (bridge) when a container is added and deleted
    - Maintain the network configuration in a JSON format
- CNI Must be invoked by the Kubernetes component responsible for container creation, therefore its configuration is determined by the kubelet server
- Configuration parameters for CNI in Kubelet (kubelet.service):
    - --network-plugin=cni
    - --cni-bin-dir=/opt/cni/bin/
    - --cni-conf-dir=/etc/cni/net.d/
- Can see these options by viewing the kubelet process
- CNI bin contains associated network plugins e.g. bridge, flannel
- Conf dir contains config files to determine the most suitable one
    - If multiple files, considerations made in alphabetical order
- IPAM section in conf considers subnets, IPs and routes etc

CNI Weave

- Becomes important when significant numbers of routes are available
- Deploys an agent or service on each node and communicates with other nodes agent
- Weave creates its own inter-node network, each agent knows the configuration and location of each node on the network, helping route the packages from one node to another, which can then be sent to the correct pod

- Weave can be deployed manually as a daemonset or via pods
- Kubectl apply -f "https://cloud.weave.works/k8s...
- Weave peers deployed as daemonsets

IP Address Management - Weave

- How are the virtual networks, nodes and pods assigned IPs?
- How do you avoid duplicate IPs
- The CNI plugin is responsible for assigning IPs
- To manage the IPs, Kubernetes isn't bothered how they're managed
    - Could do it via referencing a list
    - CNI comes with 2 built in plugins to leverage this, the host_local plugin or dynamic
- CNI conf has sections determining the plugins, routes and subnet used
- Various network solutions have different approaches
- Weave by default allocates the range 10.32.0.0/12 => 10.32.0.0 - 10.47.0.0, ~1 million IPs available, each node gets a subrange of this range defined

Service Networking

- Don't want to make each pod communicate with one another, can use services to leverage this
- Each service runs at a particular IP address and is accessible by any pod from any node, they aren't bound to a particular node
- ClusterIP = Service exposed to particular cluster only
- NodePort = Runs on a particular port on all nodes, with its own IP
- How are these services allocated IP addresses, made available to users, etc
- Each kubelet server watches for cluster changes via the api-server, each time a pod is to be created, it creates the pod and invokes the CNI plugin to configure the networking for it
- Kube-proxy watches for any changes, any time a new service is created, it's invoked, however these are virtual objects.
- Services are assigned an IP from a predefined range, associated forwarding rules are assigned to it via the kube-proxy

- The kube-proxy creates the forwarding rules via:
    - Listening on a port for each service and proxies connections to pods (userspace)
    - Creating ipvs rules
    - Use IP tables (default setting)
- Proxy mode configured via: kube-proxy --proxy-mode <proxy mode>
- When a service is created, kubernetes will assign an IP address to it, the range is set by the kube-api server option **--service-cluster-ip-range ipNet**
    - By default, set to 10.0.0.0/24
- Network ranges for services, pods etc. should never overlap as this causes conflicts
- iptables -L -t net | grep <service>
    - Displays rules created by kube-proxy for service

28 January 2021

DNS in Kubernetes

- Consider a 3-node cluster with pods and services
- Nodenames and IP addresses stored in DNS server
- Want to consider cluster-specific DNS
- Kubernetes by default deploys a Cluster-DNS server
- Manual setup required otherwise
- Consider 2 pods with one running as a service, each pod being on different nodes
- Kubernetes DNS service creates a DNS record for any service created (name and IP address)
- If in same namespace, just need curl http://service
- In different namespaces: curl http://service.namespace
- For each namespace, the kubernetes service creates a subdomain, with further subdomains for services
- The full hierarchy would follow:
    - Cluster.local (root domain)
    - Svc
    - Namespace
    - Service name

- So to access the fill service, can run: curl http://service.namespace.svc.cluster.local
- Note: DNS records aren't created for pods by default, though this can be enabled (next section)
- Once enabled, records are created for pods in the DNS server, however the pod name is rewritten, replacing the dots in the pods IP address with dashes
- Pod can then be accessed via: curl https://<pod hostname>.namespace.pod.cluster.local

CoreDNS in Kubernetes

- How does Kubernetes implement DNS?
- Could add entries into /etc/hosts file -> not suitable for large scale
- Move to central dns server and specify the nameserver located at /etc/resolv.conf
- This works for services, but for pods it works differently
- Pod hostnames are the pod IP addresses rewritten with - instead of .
- Recommended DNS server = CoreDNS
- DNS Server deployed within the cluster as a pod in the kube-system namespace
- Deployed as a replicaset
- Runs the coredns executable
- Requires a config file named Corefile at /etc/coredns/
    - Details numerous plugins for handling errors, monitoring metrics, etc
- Cluster.local defined by kubernetes plugin
    - Options here determine whether pods have records
    - Set pods insecure -> pods secure
- Coredns config deployed as configmap -> edit this to make changes
- Kube-DNS deployed as a service by default, IP address configured as the nameserver of the pod
- IP address to look at for DNS server configured via Kubelet.

Ingress

See CKAD Section 7.4

10 - Design and Install a Kubernetes Cluster

Note: The following is not required for the exam, it is merely good reference points.

Design a Kubernetes Cluster

- Considerations must be made when designing a cluster, such as:
    - Purpose
    - Workload to be hosted
    - Cloud or On-Prem
- For production clusters, should consider a high availability setup
    - Kubeadm or GCP or other supported platforms
    - Multiple nodes and master nodes
    - For cloud hosted kubernetes clusters, resource requirements are predefined
    - For on-prem, could use the figures from previous for reference
    - Kops - A popular tool for deploying Kubernetes on AWS
- Minimum of 4 nodes required (1 master, 3 workers)
- Nodes must use the Linux x64 OS
- Could separate ETCD cluster to own node, separate from the master

Choosing Kubernetes Infrastructure

- For Kubernetes on a laptop, multiple options available:
    - For Linux, could install binaries - Tedious, but worth it
    - For Windows, no native support available, Hyper-V, Virtualbox etc required to run Linux in a virtualised manner
- Minikube creates a single-node cluster easily, good for beginners practicing
- Kubeadm - Can be used to quickly deploy a multi-node cluster, though the host must be configured beforehand
- Turnkey solutions - Solutions where VMs required are manually configured and maintained
    - OpenShift as an example - Built on top of Kubernetes and easily integratable with CI/CD

    - CloudFoundry - Helps deploy and manage highly available kubernetes clusters
    - VMWare cloud PKS
    - Vagrant - Providers scripts to deploy kubernetes clusters on different cloud providers
- Hosted Solutions (Managed) - Kubernetes-as-a-service solution
    - VMs maintained and provisioned by cloud provider
    - Examples:
        - Google Container Engine (GKE)
        - Openshift Online
        - Azure Kubernetes Service
        - Amazon Elastic Container Service for Kubernetes
- For the purposes of education, Virtualbox is probably the best place to start

Configure High Availability

- As long as the worker nodes are available and nothing is going wrong, the applications on worker nodes will run even if the master node is unavailable
- Multiple master nodes are recommended for high-availability clusters; even if one master node goes down, it's all good.
- For multiple master nodes, it's better to have a load balancer to split traffic between the two api servers, nginx, ha proxy are all good examples
    - Other components like scheduler and kube-controller can't run at the same time across multiple master nodes
    - Can leverage a leader-elect approach for an active-standby approach
        - If one receives the request first, it becomes the leader-elect
        - Configure using the following options:
            - --leader-elect true
            - --leader-elect-lease-duration xs (how long does the non-leader wait until attempting to become the leader again)
            - --leader-elect-renew-deadline xs (interval between acting master attempting to renew a leadership slot before it stops leading (must be equal or less than lease duration)

        - --leader-elect-retry-period xs (The duration the clients should wait between attempting acquisition and renewal of a leadership)
- If ETCD is part of the master nodes: Stacked topology
    - Easy setup and management
    - Fewer servers involved
    - Poses a risk when failures occur
- External ETCD Topology - ETCD setup on a separate node
    - Less risky
    - Harder to setup
- Where the etcd is setup can be determined by the --etcd-servers option on the apiservers configuration

30 January 2021

ETCD In High Availability

- ETCD Previously deployed as one server, but can run multiple instances, each containing the same data as a fault-tolerant measure
- To allow this, ETCD needs to ensure that all instances are consistent in terms of what data they store, such that you can write to and read data from any of the instances
- In the event multiple write requests come in to an ETCD, the leader-elect processes the write request, which then transfers a copy to the other nodes
- Leaders decided using RAFT algorithm, using random timers for initiating requests
    - The first to finish the request becomes the leader
    - Sends out continuous notifications from then on saying “i’m continuing as leader”
    - If no notifications received e.g. the leader goes down, reelection occurs
- A write will only be considered if the transfer is completed to the majority of the Nodes or the Quorum
- Quorum = N/2 + 1; where N = Node number (for .5 numbers, round down)
- Quorum = Minimum number of nodes in an N-node cluster that need to be running for a cluster to operate as expected.

- Fault Tolerance = Instances Number - Quorum
- Odd number of master nodes recommended
- To install etcd, download the binaries from the Github repo and configure the certificates (See previous sections) and configure in etcd.service
- Etcdctl can be used to backup the data
- ETCDCTL_API default version = 2, 3 COMMONLY USED
- For ETCD, the following number of nodes are recommended: 3, 5 or 7

11 - Install “Kubernetes the kubeadm way”

1. Decide on configuration (Master vs worker)
2. Install Container Runtime (Docker)
3. Install Kubeadm
4. Initialise the master node
5. POD network setup
6. Join worker nodes to the master node

Resources

The vagrant file used in the next video is available here:

https://github.com/kodekloudhub/certified-kubernetes-administrator-course

Here's the link to the documentation:

https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/

Deploy with Kubeadm - Provision VMs with Vagrant

Using links above, and ensuring Virtualbox and vagrant are installed, clone the repo and run vagrant up

This will pull the images for the vms defined and kubernetes, creating a kubernetes master node and 2 workers

To access any, vagrant ssh <vm name> (can run commands to test things)

Check status with vagrant status

Demo: Deploy with Kubeadm

Ssh into kubemaster: vagrant ssh kubemaster

Check br_netfilter deployed: lsmod | grep br_netfilter so that iptables can see bridged traffic

Load the kernel module: sudo modprobe br_netfilter

Check for success by repeating the previous lsmod command

Repeat for each node

Lets IPtables see bridged traffic for each node

```
cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
br_netfilter
EOF

cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
sudo sysctl --system
```

Install container runtime i.e. Docker (follow instructions in documentation)

Ensure root/sudo privileges when doing the above: sudo -i

Install kubeadm, kubelet and kubectl (See Documentation)

Note: If not using docker as container runtime, would need to set cgroup

Now need to create the cluster itself

Initialise controlplane/masternode: kubadm init --pod-network-cidrs <range> --apiserver-advertise-address <masternode address>

Need to install pod network addon and setup pod network

Need to configure API server advertisement address so the worker nodes can work with it

Check ip address with ifconfig

kubeadm init --pod-network-cidr 10.244.0.0/16 --apiserver-advertise-address=192.168.56.2

Once complete, instructions provided on how to start using the cluster, including:

Change ownership of the kube config to the original user

Deploy a pod network to the cluster (gonna use weaveworks, follow instructions in documentation)

Join worker nodes to the cluster by running x command as root on each worker node

31 January 2021

13 - Troubleshooting

Application Failure

- Consider a 2-tier application, composed of a database and web server
- It's good practice to draw out the usual data flow for the application to help visualise the failure
- E.g. suppose a user is complaining about accessibility issues
- Check accessibility from the frontend first:
    - Check if the web server is accessible on the IP of the node port using curl:
        - Curl http://<service-IP>:node-port
    - Check the service to see if it has discovered the endpoints for the web pod
        - Kubectl describe service <Service-name>

    - If the service didn't find the endpoints for the pod, check the service-to-pod-discovery by comparing the selectors configured on the service and the pod respectively
    - Check the pod to ensure in a running state: kubectl get pods
        - Check running, restarts, logs and events
    - Repeat for db service and pods
    - Additional tips available via the Kubernetes documentation

Control Plane Failure

- Check node status
- Check pod status on nodes
- Check kube-system pods
- Check services: service <servicename> status
- Check logs for each pod: kubectl logs <podname> -n kube-system

Worker Node Failure

- Check node status and details (Describe)
    - See lastheartbeat field for last online time
- Check resource consumptions: kubectl top nodes
- Check the kubelet status: service kubelet status
- Check certs: openssl x500 -in /var/lib/kubelet/<cert> -text
    - See if they're issued by the correct ca

Networking Failure

<To be added at a later date>

06 February 2021

JSONPath Course:

Introduction to YAML

- Ansible playbooks format similar to XML and JSON

- Used to express data
- Data in YAML files, at its most basic form is a series of Key-Value pairs, separated by a colon
- For an array:

Array title

- entry1

- entry2

- entry3

- For a dictionary:
    - A set of properties grouped together under 1 item

Item:

Property1: value1

Property2: value2

- Note: 2 spaces determine what properties come under what iteM
- Can have lists containing dictionaries containing lists
- To store information about different properties of a single item, use a dictionary
- If the properties need further segregation, can use dictionaries within dictionary items
- For multiple items of the same type, use an array
- To store information for multiple items, expand each array item to a the dictionary for each

Key Notes:

- Dictionary = Unordered
- Lists/Arrays = Ordered
- Lines beginning with # = comment

Introduction to JSON Path

- YAML vs JSON:
    - Data can be expressed via both
    - To segregate data, methods differ:
        - YAML - Indentations and -'s
        - JSON - Indentation and {} for dictionaries, [] for array items
- JSON data can be queried via JSON Path
- To select an item, specify it as <item>
- For a dictionary property: <item>.<property>, repeat as appropriate
- Note: Anything within {} is a dictionary
- The top level dictionary, which isn't named, is denoted by a $
- Query looks like: $.item.property etc
- Any output from a JSON Path query is in an [] array
- To query an array/list, use square brackets to reference the position, with positions starting at [0].
    - E.g. 1st element: $.[0]
- For dictionaries in lists, combine the query use for all
- For criteria: $.[CRITERIA] e.g.
    - $.[?(@ > 40)]
    - In this case, ?() signals to use a filter, the @ symbol signifies "each item"
    - Queries could be used to accommodate for changing positions
    - E.g. $.car.wheels[?(@.location == "rear-right")].model
        - Finds entry that satisfies the criteria

JSON Path - Wildcard

- Denoted by *, meaning "any", can be used to retrieve all/any properties of a particular dictionary
- Can swap * as a value when referencing an array position i.e. [*]

JSON Path - Advanced List Queries

- To get all names in an array's particular range, add [x:y], where x is the first element's position, y is the end position of the range +1
- To iterate over a step, insert: [x:y:z], where z is the step rate

- To get the last item: [-1:0]

Advanced Kubectl Commands - Kubectl and JSON Path

- Prerequisites:
    - JSONPath for beginners
    - JSONPath Practice tests - Kodekloud: General use and for Kubernetes
- Why JSON Path?
    - Large data sets involved with production-grade clusters
        - 100s of nodes,
        - 1000s of PODS, Deployments, ReplicaSets etc
    - More often than not, will want to quickly print information for large numbers of resources and particular information
- Kubectl commands invokes the APIServer, which obtains the information requested in a JSON format and is redisplayed ina readable format by Kubectl
    - Particularly noticeable in kubectl get commands
    - For additional information, add -o wide flag
- Example:
    - Suppose we want to see the following:
        - CPU count
        - Taints and tolerations
        - Pod name and images
    - There is no built-in kubectl command for this, but we can use kubectl and JSON path in combination to get the particular fields
- To use JSON Path in Kubectl, consider the 4 steps:
    - Identify the kubectl command required e.g. kubectl get pods
    - Familiarize yourself with the JSON format output: add the -o json flag
    - From the JSON output, figure out the custom query you'd want to apply, e.g. for container images of a particular pod: .items[*].spec.containers[*].image
    - Combine the kubectl command with the JSON query i.e. kubectl get pods -o=jsonpath='{JSON_PATH_QUERY}'
- Note: For multiple queries, within the ' ', encompass each query by {}
    - To format this, use any of the following:

        - {“\n”} - New line (Add in between queries)
        - {“\t”} - Tab
- Can also loop through ranges:
    - Use ‘{range .items[*]} {Queries} {end}’
- Can print custom columns via the is of
  -o=custom-columns=<COLUMN_NAME>:<JSON PATH>, <COLUMN>:<JSON PATH>
    - Recommended to view full query first then forming the JSON query
- Use --sort-by property where necessary e.g. --sort-by=.metadata.name

Mock Exam Notes:

ETCD Backup:

USE KUBERNETES DOCUMENTATION (WHAT YOU INITIALLY WENT FOR)

ETCDCTL_API=3 etctctl version

Navigate to /etc/etc/kubernetes/manifests

Check etcd yaml and find command for:

- Endpoints
- Cacert
- Cert
- Key

Command: ETCDCTL_API=3 etcdctl --endpoints=<> --cacert=<> --cert=<> --key=<> snapshot save <filepath>

Use-PV Question:

- Create persistentvolumeclaim
    - 10Mi works
    - Ensure correct access mode
    - No storage class needs to be specified
- Specify pvc and volume mount as required

Record Annotations

kubectl run .... --record

Kubectl set image ..... --record

Kubectl rollout history and status where appropriate

CSR

Use manage TLS certificates task in kubernetes documentation

Create using spec provided in YAML file

Encode .csr using |base64 and format as appropriate

Create the CSR

Approve the CSR

Create role with appropriate spec via kubectl create; separate verbs by commas

Create rolebinding -> developer-role-ninding --role=developer user=john ns = developer with kubectl create

check permissions with kubectl auth can-i ……--options

Nginx-Resolver:

- Just use port 80
- Type=clusterIp obvs
- Test DNS lookup with busybox pod: --rm -it -- nslookup <service> (pod IP later.namespace.pod)
- Record as appropriate

Kubectl expose pod nginx-resolver --port=80 --target-port=80

Kubectl describe svc -> gets the IP and endpoint

Kubectl run test-nslookup --image=busybox:1.28 --rm -it -- nslookup nginx-resolver-service > /root/nginx.svc

Kubectl run test-nslookup --image=busybox:1.28 --rm-it -- nslookup <pod IP address> > filepath

Mock Exam 3

PVViewer

Kubectl create serviceaccount

Kubectl create clusterrole -<name> --resource=persistentvolumes --verb=list

Kubectl create clusterrolebinding pvviewer-role-binding --clusterrole=<clusterrole> --serviceaccount=namespace.serviceaccount

Multi-pod

Name: name

Value: alpha/beta

Beta command = sleep 4800

KubeConfig:

Run kubectl cluster-info --kubeconfig=

Analyse output

Edit API Server port to 6443

Replicas

Edit kube-controller yaml file and edit typos