Certified Kubernetes Application Developer (CKAD)

Certified Kubernetes Application Developer (CKAD) 1
References 3
Theory 4
  Architecture 4
  Minikube 7
  Pods 10
  Labels 11
  ReplicaSet 11
  Deployment 12
  Namespaces 14
  Service Accounts 14
  RBAC Authorizer 14
  Exercise - Authentication and Authorization 16
  Services 22
    kube-proxy 23
  Deployment 27
  Volume Management 29
  ConfigMaps and Secrets 32
  Ingress 34
Exam Info 41
PODS 45
  Lifecycle phases of Pod 45
Configuration 48
  Consuming configmaps in a pod 49
  Exercise for configmaps 50
Secrets 52

Exercise 53
Security Context 54
Exercise 55
Resource boundary definition 56
Exercise 57
Service accounts 58
Exercise 59
Multi-container pods 60
Observability 61
Exercise 62
Defining a Pod’s Readiness and Liveness Probe 62
Fixing a Misconfigured Pod 64
Pod Design 65
Labels 65
Annotations 66
Exercise 67
Deployments 68
Scaling a Deployment 70
Jobs and CronJobs 72
Services and Networking 74
Service 74
Network Policies 76
State Persistence 78
Volumes 78
Persistent Volumes 78
Shortcuts 80
Exercises 80
Important references for Kubectl 80
Core Concepts 80
Multi-container Pods 82
Pod Design 82
Deployments 84
Jobs 85
CronJob 87
Configurations 88

ConfigMaps 88
SecurityContext 90
Requests and Limits 91
Secrets 91
ServiceAccounts 92
Observability 93
Liveness and readiness probes 93
Logging 94
Debugging 94
Services and Networking 95
State Persistence 98
LFD259 - Kubernetes for Developers Course Notes 100
Kubernetes Architecture 101

References

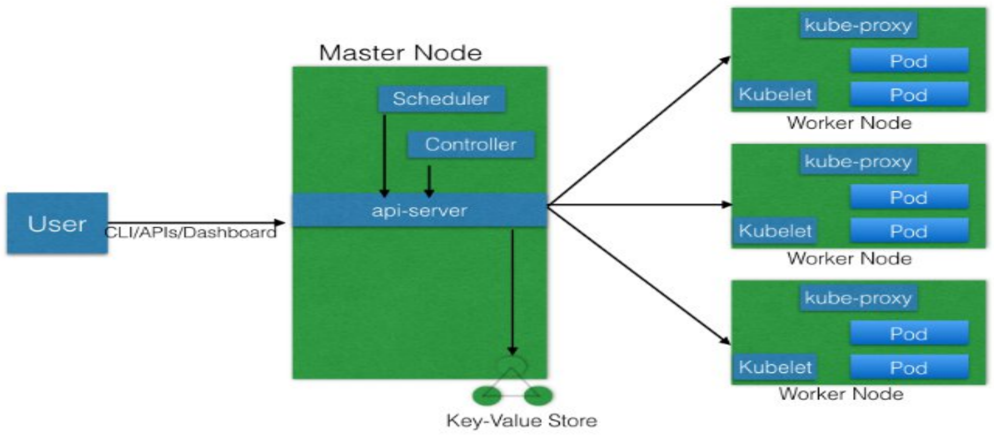
1. https://github.com/bmuschko/ckad-prep - to practice exercises with the below video course.
    a. https://learning.oreilly.com/learning-paths/learning-path-certified/9781492061021 - Video course by Brian Muschko
2. https://kubernetes.io/docs/ - Must practice finding the yaml definitions by searching the docs for various tasks - like creating deployments, services, network policies, etc.
3. https://courses.edx.org/courses/course-v1:LinuxFoundationX+LFS158x+2T2019/course/ (Free) - This is required before you do the paid course from CNCF for CKAD. The theory portion of my notes is excerpted from this.
4. Book: **Kubernetes Up and Running 2nd Edition** by Burns, Beda and Hightower - A more up-to-date book that covers all commands and a quick read compared to the next book.
5. Book: **Kubernetes in Action** by Marko Luska - This is a large book but very readable and said to be one of the best to understand the concepts behind the components of kubernetes.
6. CNCF CKAD Training course (Paid) - https://www.cncf.io/certification/training/
7. Exercises: https://github.com/dgkanatsios/CKAD-exercises - Must do set of exercises. Do it 4-5 times before exam to build speed.
8. CKAD Prep Notes: https://github.com/twajr/ckad-prep-notes
9. https://github.com/cncf/curriculum/blob/master/CKAD_Curriculum_V1.18.pdf

10. https://kubernetes.io/docs/tasks/manage-kubernetes-objects/imperative-command/ - It is very important to save time on exam and imperative commands do that.
11. Some more practice - https://medium.com/bb-tutorials-and-thoughts/practice-enough-with-these-questions-for-the-ckad-exam-2f42d1228552
12. https://kubernetes.io/docs/reference/kubectl/cheatsheet/

Theory

This section is based on Introduction to Kubernetes course by CNCF at edx.org.

Architecture



Kubernetes Architecture

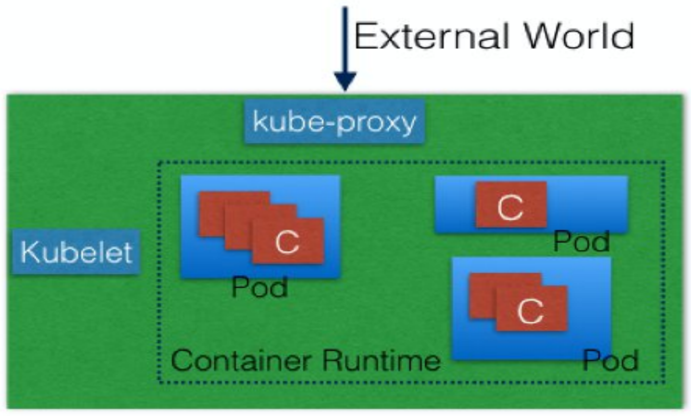
1. Master Nodes - The **master node** provides a running environment for the control plane responsible for managing the state of a Kubernetes cluster, and it is the brain behind all operations inside the cluster. The control plane components are agents with very distinct roles in the cluster's management. To persist the Kubernetes cluster's state, all cluster configuration data is saved to etcd. However, **etcd** is a distributed key-value store which only holds cluster state related data, no client workload data. Master node consists of following components:
    a. API Server - intercepts RESTful calls from users, operators and external agents, then validates and processes them. During processing the API server reads the Kubernetes cluster's current state from the

etcd, and after a call's execution, the resulting state of the Kubernetes cluster is saved in the distributed key-value data store for persistence. The API server is the only master plane component to talk to the etcd data store, both to read and to save Kubernetes cluster state information from/to it - acting as a middle-man interface for any other control plane agent requiring to access the cluster's data store.

b. Scheduler - The role of the **kube-scheduler** is to assign new objects, such as pods, to nodes. During the scheduling process, decisions are made based on current Kubernetes cluster state and new object's requirements. The scheduler obtains from etcd, via the API server, resource usage data for each worker node in the cluster. The scheduler also receives from the API server the new object's requirements which are part of its configuration data. Requirements may include constraints that users and operators set, such as scheduling work on a node labeled with **disk==ssd** key/value pair. The scheduler also takes into account Quality of Service (QoS) requirements, data locality, affinity, anti-affinity, taints, toleration, etc.
c. Controller Managers - Controllers are watch-loops continuously running and comparing the cluster's desired state (provided by objects' configuration data) with its current state (obtained from etcd data store via the API server). In case of a mismatch corrective action is taken in the cluster until its current state matches the desired state.
d. etcd - **etcd** is a distributed key-value data store used to persist a Kubernetes cluster's state. New data is written to the data store only by appending to it, data is never replaced in the data store. Obsolete data is compacted periodically to minimize the size of the data store. Out of all the control plane components, only the API server is able to communicate with the etcd data store. etcd is based on the Raft Consensus Algorithm which allows a collection of machines to work as a coherent group that can survive the failures of some of its members. At any given time, one of the nodes in the group will be the master, and the rest of them will be the followers. Any node can be treated as a master. Its a b+ tree key-value store. It works with curl and provides reliable watch queries. For simulataneous requests to update

a value via kube-apiserver, they are serialized and sent to the etcd DB. The first request updates the value but the following requests to update fail and server returns 409 error (conflict).

2. Worker Node - provide running environment for client applications. Pods - logical collection of contianers that are scheduled together.
    a. container runtime - docker, CRI-O, rkt, containerd
    b. kubelet - The **kubelet** is an agent running on each node and communicates with the control plane components from the master node. It receives Pod definitions, primarily from the API server, and interacts with the container runtime on the node to run containers associated with the Pod. It also monitors the health of the Pod's running containers. Any container runtime that implements CRI can be used by Kubernetes to manage Pods, containers, and container images.
    c. kube-proxy - The **kube-proxy** is the network agent which runs on each node responsible for dynamic updates and maintenance of all networking rules on the node. It abstracts the details of Pods networking and forwards connection requests to Pods.
    d. Addons for DNS, Dashboard, cluster-level monitoring and logging. - **Addons** are cluster features and functionality not yet available in Kubernetes, therefore implemented through 3rd-party pods and services.



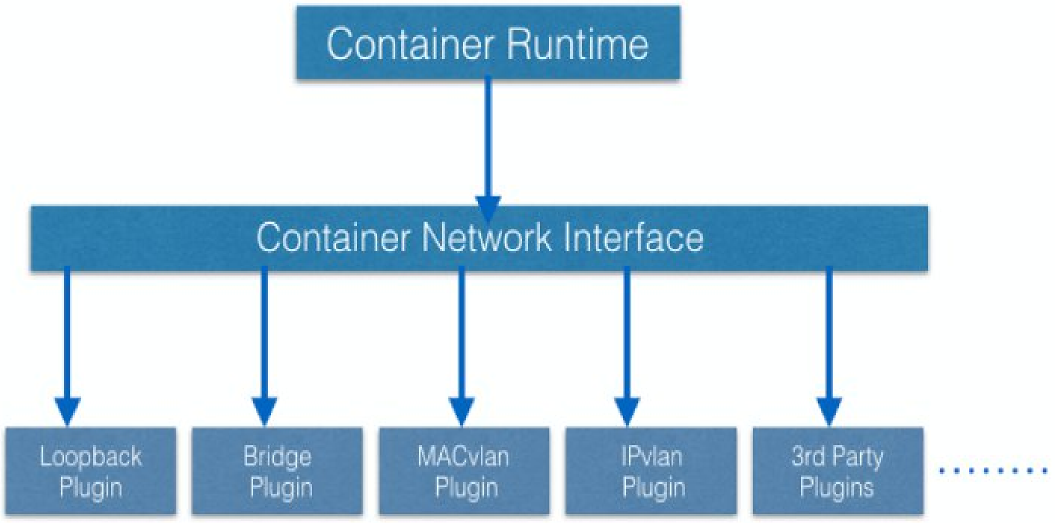
Kubernetes Worker Node

3. Networking

a. Container to container communication inside pods - a container runtime creates an isolated network space for each container it starts. On Linux, that isolated network space is referred to as a **network namespace**. A network namespace is shared across containers, or with the host operating system.

   When a Pod is started, a network namespace is created inside the Pod, and all containers running inside the Pod will share that network namespace so that they can talk to each other via localhost.

b. Pod to Pod on same node and across nodes in a cluster - Each Pod receives an IP Address (IP-per-Pod).



Container Network Interface (CNI)

c. Pod to service within same namespace and across cluster namespaces
d. External to Service communication fro clients to access applications in a cluster - Kubernetes enables external accessibility through **services**, complex constructs which encapsulate networking rules definitions on cluster nodes. By exposing services to the external world with **kube-proxy**, applications become accessible from outside the cluster over a virtual IP.

4. Types of kubernetes installations possible:
   a. All-in-one single-node installation

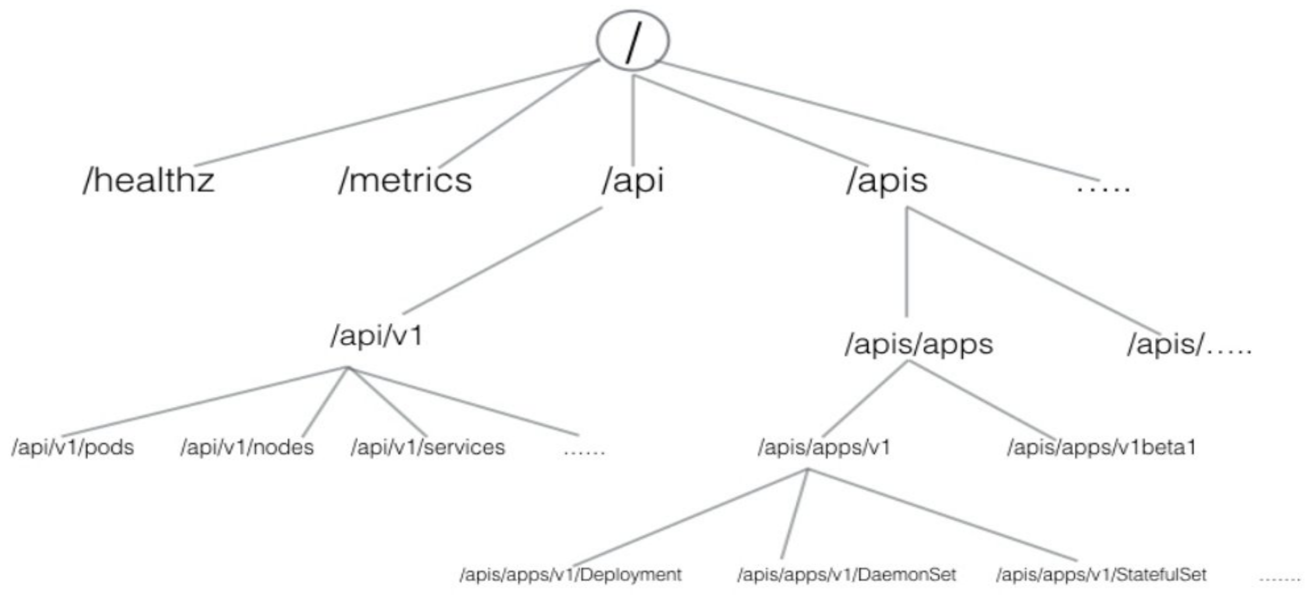
b. single node etcd, single master and multi worker installation
c. single node etcd, multi master and multi worker
d. multi node etcd, multi master and multi worker - most advanced and recommended setup for production workloads.

Minikube

Requires a type-2 hypervisor (KVM or virtualbox) on the node. Spawns a single VM for a single node cluster.

VT-x/AMD-v virtualization must be enabled in BIOS.

Kubernetes API spac



HTTP API Space of Kubernetes

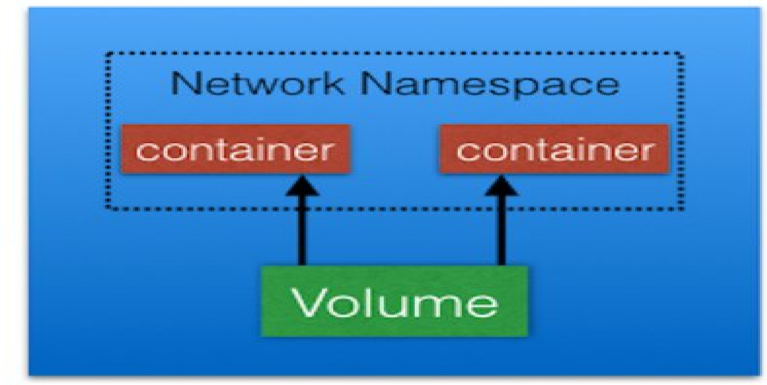
- Core group - /api/v1
- Named Group - /apis/$NAME/$VERSION
- System-wide - /healthz, /logs, /metrics, /ui etc.

To access the Kubernetes cluster, the **kubectl** client needs the master node endpoint and appropriate credentials to be able to interact with the API server running on the master node. While starting Minikube, the startup process creates, by default, a configuration file (~/.kube/config).

```
k config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority: /home/rwatsh/.minikube/ca.crt
    server: https://192.168.39.99:8443
  name: minikube
contexts:
- context:
    cluster: minikube
    user: minikube
  name: minikube
current-context: minikube
kind: Config
preferences: {}
users:
- name: minikube
  user:
    client-certificate: /home/rwatsh/.minikube/client.crt
    client-key: /home/rwatsh/.minikube/client.key

k cluster-info

minikube dashboard

k proxy &

curl http://localhost:8001
curl http://localhost:8001/healthz
curl http://localhost:8001/metrics

TOKEN=$(kubectl describe secret -n kube-system $(kubectl get secrets -n kube-
system | grep default | cut -f1 -d ' ') | grep -E '^token' | cut -f2 -d':' | tr -d '\t' | tr -d "
")
APISERVER=$(kubectl config view | grep https | cut -f 2- -d ":" | tr -d " ")
```

```
curl $APISERVER --header "Authorization: Bearer $TOKEN" --insecure
```

When not using the **kubectl proxy**, we need to authenticate to the API server when sending API requests. We can authenticate by providing a **Bearer Token** when issuing a **curl**, or by providing a set of **keys** and **certificates**.



Pods

Pods

1. one or more containers
2. share same network namespace
3. have access to mount the same external storage (volume)

Pods are ephemeral in nature, and they do not have the capability to self-heal by themselves. That is the reason they are used with controllers which handle Pods' replication, fault tolerance, self-healing, etc. Examples of controllers are Deployments, ReplicaSets, ReplicationControllers, etc. We attach a nested Pod's specification to a controller object using the Pod Template

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.15.11
        ports:
        - containerPort: 80
```

spec - defines the replicas = 3 and label selector

template.spec - defines the pod with matching label

Deployment/ReplicaSet is the controller that this pod is associated with to ensure that there are 3 replicas of the pod at any time.

Labels

Labels are key-value pairs attached to Kubernetes objects (e.g. Pods, ReplicaSets). Labels are used to organize and select a subset of objects, based on the requirements in place. Many objects can have the same Label(s). Labels do not provide uniqueness to objects. Controllers use Labels to logically group together decoupled objects, rather than using objects' names or IDs.

Controllers use Label Selectors to select a subset of objects. Kubernetes supports two types of Selectors:

1. Equality based - =, == or != (both = and == mean the same)
2. Set based - in/not in for values and exists/does not exist for keys. Eg.

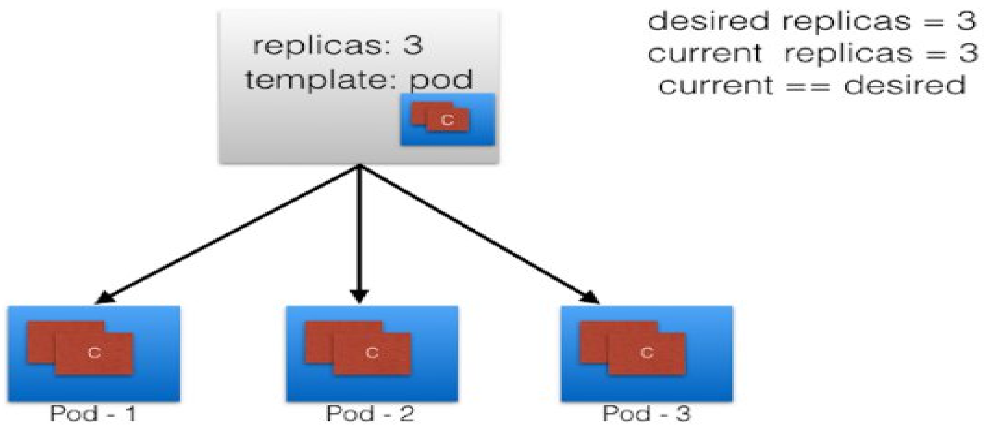
   env in (dev,qa) - objects with either dev or qa value for env.

   !app - select objects with no label key app.

Generally, we don't deploy a Pod independently, as it would not be able to re-start itself if terminated in error. The recommended method is to use some type of replication controllers to create and manage Pods.

The default controller is a Deployment which configures a ReplicaSet to manage Pods' lifecycle.

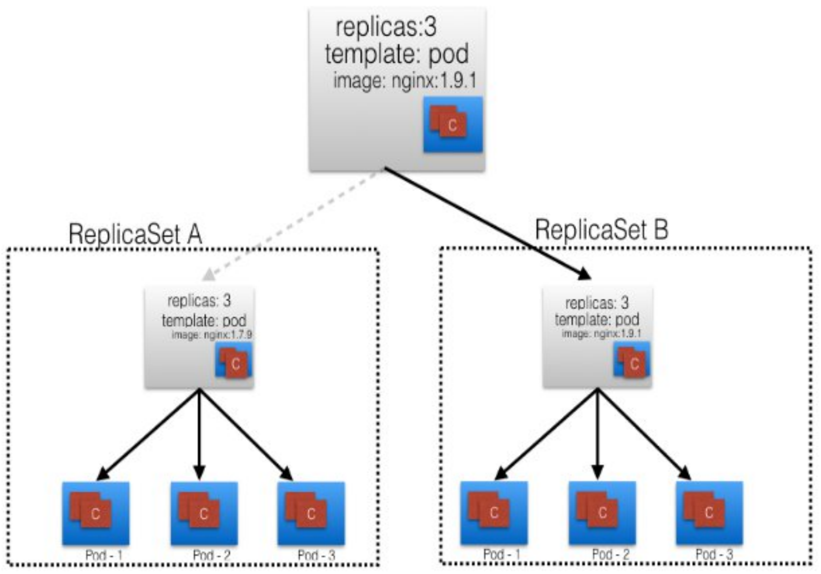
ReplicaSet
A ReplicaSet is the next-generation ReplicationController. ReplicaSets support both equality- and set-based selectors, whereas ReplicationControllers only support equality-based Selectors.



ReplicaSet (Current State and Desired State Are the Same)

ReplicaSets can be used independently as Pod controllers but they only offer a limited set of features. A set of complementary features are provided by Deployments, the recommended controllers for the orchestration of Pods. Deployments manage the creation, deletion, and updates of Pods. A Deployment automatically creates a ReplicaSet, which then creates a Pod. There is no need to manage ReplicaSets and Pods separately, the Deployment will manage them on our behalf.

Deployment



Deployment (ReplicaSet B Created)

Deployment objects provide declarative updates to Pods and ReplicaSets. The DeploymentController is part of the master node's controller manager, and it ensures that the current state always matches the desired state. It allows for seamless application updates and downgrades through **rollouts** and **rollbacks**, and it directly manages its ReplicaSets for application scaling.

Now, in the Deployment, we change the Pods' Template and we update the container image from **nginx:1.7.9** to **nginx:1.9.1**. The **Deployment** triggers a new **ReplicaSet B** for the new container image versioned **1.9.1** and this association represents a new recorded state of the **Deployment**, **Revision 2**. The seamless transition between the two ReplicaSets, from **ReplicaSet A** with 3 Pods versioned **1.7.9** to the new **ReplicaSet B** with 3 new Pods versioned **1.9.1**, or from **Revision 1** to **Revision 2**, is a Deployment **rolling update**.

A **rolling update** is triggered when we update the Pods Template for a deployment. Operations like scaling or labeling the deployment do not trigger a rolling update, thus do not change the Revision number.

Once the rolling update has completed, the **Deployment** will show both **ReplicaSets A** and **B**, where **A** is scaled to 0 (zero) Pods, and **B** is scaled to 3 Pods. This is how the Deployment records its prior state configuration settings, as **Revisions**.

Namespaces

We can partition the cluster into virtual sub-clusters using Namespaces. The names of the resources/objects created inside a Namespace are unique, but not across Namespaces in the cluster.

Default namespaces:

1. kube-system - contains the objects created by the Kubernetes system, mostly the control plane agents.
2. default - contains the objects and resources created by administrators and developers. By default, we connect to the **default** Namespace.
3. kube-public - a special Namespace, which is unsecured and readable by anyone, used for special purposes such as exposing public (non-sensitive) information about the cluster.
4. kube-node-lease - holds node lease objects used for node heartbeat data.

Good practice, however, is to create more Namespaces to virtualize the cluster for users and developer teams. With Resource Quotas, we can divide the cluster resources within Namespaces.

Service Accounts

With Service Account users, in-cluster processes communicate with the API server to perform different operations. Most of the Service Account users are created automatically via the API server, but they can also be created manually. The Service Account users are tied to a given Namespace and mount the respective credentials to communicate with the API server as Secrets.

RBAC Authorizer

We can have different roles that can be attached to subjects like users, service accounts, etc. While creating the roles, we restrict resource access by specific operations, such as **create**, **get**, **update**, **patch**, etc. These operations are referred to as verbs.

2 kinds of roles:

1. **Role** - grant access to resources within a namespace
2. **ClusterRole** - grant access to resources cluster-wide.

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: lfs158
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

it creates a **pod-reader** role, which has access only to read the Pods of **lfs158** Namespace. Once the role is created, we can bind users with *RoleBinding.*

2 kinds of RoleBinding:

1. RoleBinding - bind users to the same namespace as a Role
2. ClusterRoleBinding - bind users to ClusterRole roles.

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: pod-read-access
  namespace: lfs158
subjects:
```

```
- kind: User

  name: student

  apiGroup: rbac.authorization.k8s.io

roleRef:

  kind: Role

  name: pod-reader

  apiGroup: rbac.authorization.k8s.io
```

Gives access to the *student* user to read the Pods of **lfs158** Namespace.

Exercise - Authentication and Authorization

View the content of the **kubectl** client's configuration file, observing the only context **minikube** and the only user **minikube**, created by default:

```
$ kubectl config view

apiVersion: v1
clusters:
- cluster:
    certificate-authority: /home/student/.minikube/ca.crt
    server: https://192.168.99.100:8443
  name: minikube
contexts:
- context:
    cluster: minikube
    user: minikube
  name: minikube
current-context: minikube
kind: Config
preferences: {}
users:
- name: minikube
  user:
    client-certificate: /home/student/.minikube/client.crt
    client-key: /home/student/.minikube/client.key
```

Create **lfs158** namespace:

```
$ kubectl create namespace lfs158
```

```
namespace/lfs158 created
```

Create **rbac** directory and **cd** into it:

```
$ mkdir rbac

$ cd rbac/
```

Create a **private key** for the **student** user with **openssl** tool, then create a **certificate signing request** for the **student** user with **openssl** tool:

```
~/rbac$ openssl genrsa -out student.key 2048

Generating RSA private key, 2048 bit long modulus (2 primes)
.............................................+++++
.......................+++++
e is 65537 (0x010001)

~/rbac$ openssl req -new -key student.key -out student.csr -subj
"/CN=student/O=learner"
```

Create a YAML configuration file for a **certificate signing request** object, and save it with a blank value for the **request** field:

```
~/rbac$ vim signing-request.yaml

apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: student-csr
spec:
  groups:
  - system:authenticated
  request: <assign encoded value from next cat command>
  usages:
  - digital signature
  - key encipherment
  - client auth
```

View the **certificate**, encode it in **base64**, and assign it to the **request** field in the **signing-request.yaml** file:

```
~/rbac$ cat student.csr | base64 | tr -d '\n'
```

```
LS0tLS1CRUd...1QtLS0tLQo=

~/rbac$ vim signing-request.yaml

apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: student-csr
spec:
  groups:
  - system:authenticated
  request: LS0tLS1CRUd...1QtLS0tLQo=
  usages:
  - digital signature
  - key encipherment
  - client auth
```

Create the **certificate signing request** object, then list the certificate signing request objects. It shows a **pending** state:

```
~/rbac$ kubectl create -f signing-request.yaml

certificatesigningrequest.certificates.k8s.io/student-csr created

~/rbac$ kubectl get csr

NAME          AGE   REQUESTOR      CONDITION
student-csr   27s   minikube-user   Pending
```

Approve the **certificate signing request** object, then list the certificate signing request objects again. It shows both **approved** and **issued** states:

```
~/rbac$ kubectl certificate approve student-csr

certificatesigningrequest.certificates.k8s.io/student-csr approved

~/rbac$ kubectl get csr

NAME          AGE   REQUESTOR      CONDITION
student-csr   77s   minikube-user   Approved,Issued
```

Extract the approved **certificate** from the **certificate signing request**, decode it with **base64** and save it as a **certificate file**. Then view the certificate in the newly created certificate file:

```
~/rbac$ kubectl get csr student-csr -o jsonpath='{.status.certificate}' | base64 --decode > student.crt

~/rbac$ cat student.crt

-----BEGIN CERTIFICATE-----
MIIDGzCCA...
...
...NOZRRZBVunTjK7A==
-----END CERTIFICATE-----
```

Configure the **student** user's credentials by assigning the **key** and **certificate**:

```
~/rbac$ kubectl config set-credentials student --client-certificate=student.crt --client-key=student.key

User "student" set.
```

Create a new **context** entry in the **kubectl** client's configuration file for the **student** user, associated with the **lfs158** namespace in the **minikube** cluster:

```
~/rbac$ kubectl config set-context student-context --cluster=minikube --namespace=lfs158 --user=student

Context "student-context" created.
```

View the contents of the **kubectl** client's configuration file again, observing the new **context** entry **student-context**, and the new **user** entry **student**:

```
~/rbac$ kubectl config view

apiVersion: v1
clusters:
- cluster:
    certificate-authority: /home/student/.minikube/ca.crt
    server: https://192.168.99.100:8443
  name: minikube
contexts:
- context:
    cluster: minikube
    user: minikube
  name: minikube
- context:
    cluster: minikube
    namespace: lfs158
    user: student
  name: student-context
current-context: minikube
kind: Config
preferences: {}
users:
- name: minikube
  user:
    client-certificate: /home/student/.minikube/client.crt
    client-key: /home/student/.minikube/client.key
- name: student
  user:
    client-certificate: /home/student/rbac/student.crt
    client-key: /home/student/rbac/student.key
```

While in the default **minikube context**, create a new **deployment** in the **lfs158** namespace:

```
~/rbac$ kubectl -n lfs158 create deployment nginx --image=nginx:alpine

deployment.apps/nginx created
```

From the new **context student-context** try to list pods. The attempt fails because the **student** user has no permissions configured for the **student-context**:

```
~/rbac$ kubectl --context=student-context get pods

Error from server (Forbidden): pods is forbidden: User "student" cannot list resource
"pods" in API group "" in the namespace "lfs158"
```

The following steps will assign a limited set of permissions to the **student** user in the **student-context**.

Create a YAML configuration file for a **pod-reader role** object, which allows only **get, watch, list** actions in the **lfs158** namespace against **pod** objects. Then create the **role** object and list it from the default **minikube context**, but from the **lfs158** namespace:

```
~/rbac$ vim role.yaml

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
```

```
metadata:
  name: pod-reader
  namespace: lfs158
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]

~/rbac$ kubectl create -f role.yaml

role.rbac.authorization.k8s.io/pod-reader created

~/rbac$ kubectl -n lfs158 get roles

NAME         AGE
pod-reader   57s
```

Create a YAML configuration file for a **rolebinding** object, which assigns the permissions of the **pod-reader role** to the **student** user. Then create the **rolebinding** object and list it from the default **minikube context**, but from the **lfs158** namespace:

```
~/rbac$ vim rolebinding.yaml

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-read-access
  namespace: lfs158
subjects:
- kind: User
  name: student
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io

~/rbac$ kubectl create -f rolebinding.yaml

rolebinding.rbac.authorization.k8s.io/pod-read-access created

~/rbac$ kubectl -n lfs158 get rolebindings

NAME              AGE
```

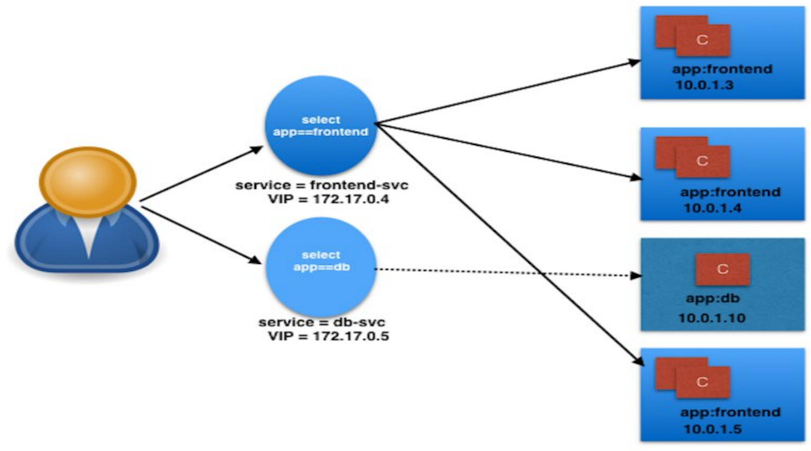
```
pod-read-access   23s
```

Now that we have assigned permissions to the **student** user, we can successfully list **pods** from the new **context student-context**.

```
~/rbac$ kubectl --context=student-context get pods

NAME                   READY   STATUS    RESTARTS   AGE
nginx-77595c695-f2xmd   1/1     Running   0          7m41s
```

Services

Kubernetes provides a higher-level abstraction called Service, which logically groups Pods and defines a policy to access them. This grouping is achieved via **Labels** and **Selectors**.



Grouping of Pods using the Service object

Using the selectors **app==frontend** and **app==db**, we group Pods into two logical sets: one with 3 Pods, and one with a single Pod.

We assign a name to the logical grouping, referred to as a **Service**. In our example, we create two Services, **frontend-svc**, and **db-svc**, and they have the **app==frontend** and the **app==db** Selectors, respectively.

By default, each Service receives an IP address routable only inside the cluster, known as **ClusterIP**. The user/client now connects to a Service via its **ClusterIP**, which forwards traffic to one of the Pods attached to it. A Service provides load balancing by default while selecting the Pods for traffic forwarding.

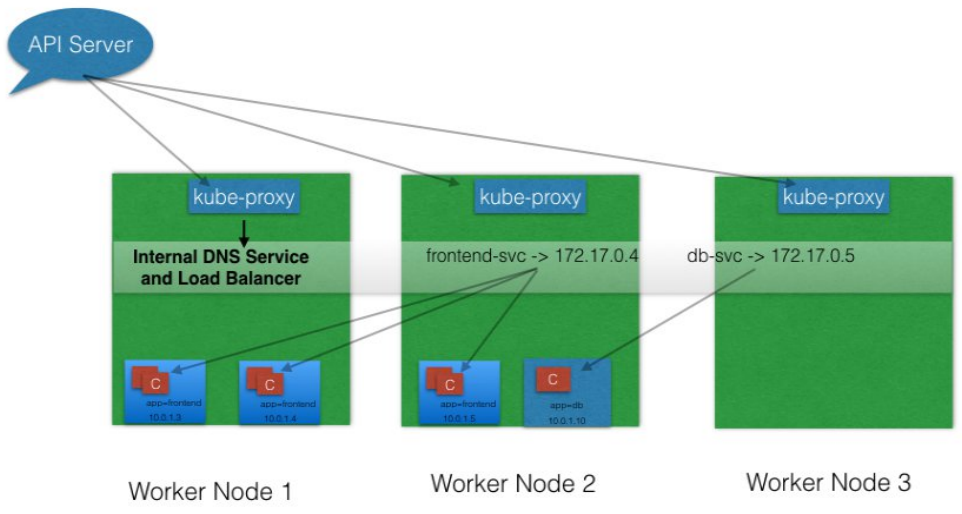
Services can expose single Pods, ReplicaSets, Deployments, DaemonSets, and StatefulSets. While the Service forwards traffic to Pods, we can select the **targetPort** on the Pod which receives the traffic. If targetPort is not defined then it is same as value of service port.

A logical set of a Pod's IP address, along with the **targetPort** is referred to as a **Service endpoint**. For example, the **frontend-svc** Service has 3 endpoints: **10.0.1.3:5000**, **10.0.1.4:5000**, and **10.0.1.5:5000**.

```
kind: Service
apiVersion: v1
metadata:
  name: frontend-svc
spec:
  selector:
    app: frontend
  ports:
  - protocol: TCP
    port: 80
    targetPort: 5000
```

kube-proxy

All worker nodes run a daemon called **kube-proxy**, which watches the API server on the master node for the addition and removal of Services and endpoints. In the example below, for each new Service, on each node, **kube-proxy** configures **iptables** rules to capture the traffic for its ClusterIP and forwards it to one of the Service's endpoints. Therefore any node can receive the external traffic and then route it internally in the cluster based on the **iptables** rules. When the Service is removed, **kube-proxy** removes the corresponding **iptables** rules on all nodes as well.



kube-proxy, Services, and Endpoints

Kubernetes supports two methods for discovering Services:

1. Environment Variables - As soon as the Pod starts on any worker node, the **kubelet** daemon running on that node adds a set of environment variables in the Pod for all active Services.

   ```
   REDIS_MASTER_SERVICE_HOST=172.17.0.6
   REDIS_MASTER_SERVICE_PORT=6379
   REDIS_MASTER_PORT=tcp://172.17.0.6:6379
   REDIS_MASTER_PORT_6379_TCP=tcp://172.17.0.6:6379
   REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
   REDIS_MASTER_PORT_6379_TCP_PORT=6379
   REDIS_MASTER_PORT_6379_TCP_ADDR=172.17.0.6
   ```

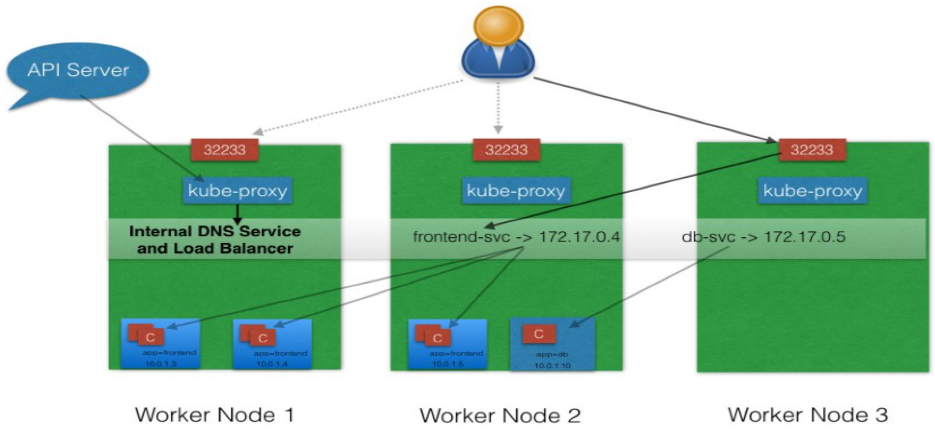
   Note: The above injection of environment variables into pods will only work if the pods are created after the service.

2. DNS - Kubernetes has an add-on for DNS, which creates a DNS record for each Service and its format is **my-svc.my-namespace.svc.cluster.local**. Services within the same Namespace find other Services just by their name. If we add a Service **redis-master** in **my-ns** Namespace, all Pods in the same Namespace lookup the Service just by its name, **redis-master**. Pods from other Namespaces lookup the same Service by adding the respective Namespace as a suffix, such as **redis-master.my-ns.**

This solution to discovering a service is recommended over environment variables.

Access scope is decided by **ServiceType**, which can be configured when creating the Service.

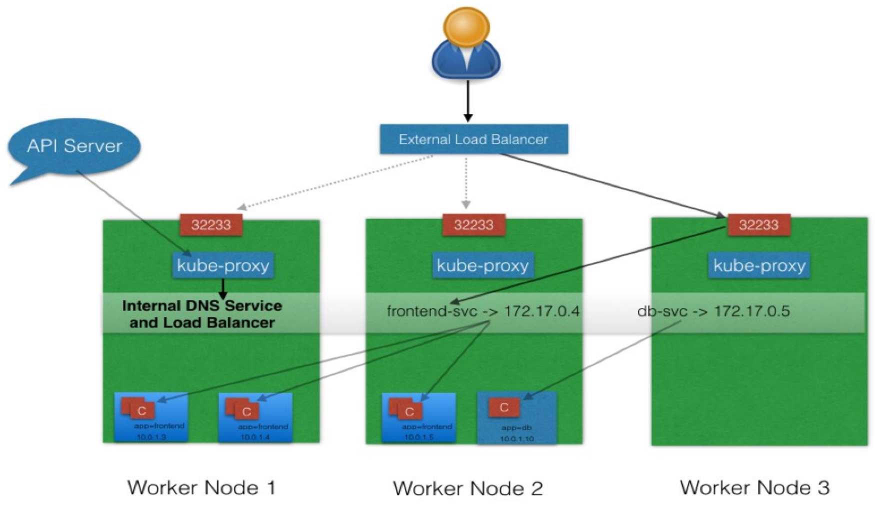
1. ClusterIP - default. Only accessible within cluster.
2. NodePort - With the **NodePort** *ServiceType*, in addition to a ClusterIP, a high-port, dynamically picked from the default range **30000-32767**, is mapped to the respective Service, from all the worker nodes. For example, if the mapped NodePort is **32233** for the service **frontend-svc**, then, if we connect to any worker node on port **32233**, the node would redirect all the traffic to the assigned ClusterIP - **172.17.0.4**. If we prefer a specific high-port number instead, then we can assign that high-port number to the NodePort from the default range.



NodePort

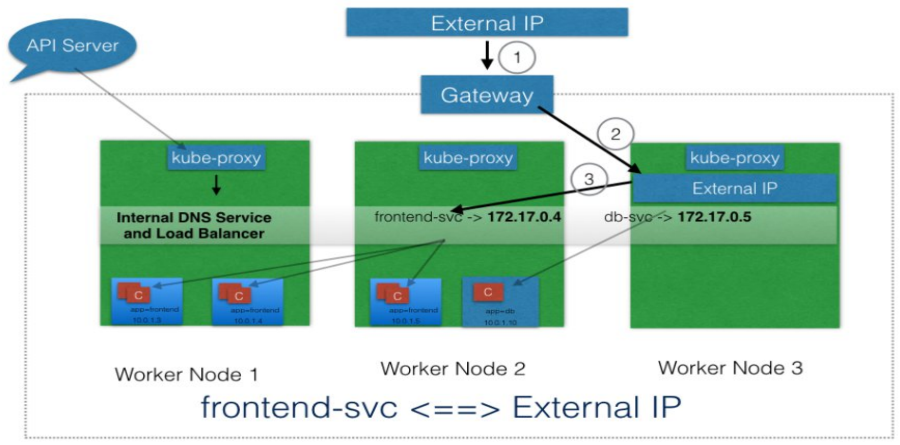
To access multiple applications from the external world, administrators can configure a reverse proxy - an ingress, and define rules that target Services within the cluster.

3. LoadBalancer - NodePort and ClusterIP are created automatically and external LB routes to those Service endpoints. The LB is exposed externally via Cloud Provider’s LB features.



LoadBalancer

4. ExternalIP - A Service can be mapped to an **ExternalIP** address if it can route to one or more of the worker nodes. Traffic that is ingressed into the cluster with the ExternalIP (as destination IP) on the Service port, gets routed to one of the Service endpoints. This type of service requires an external cloud provider. Please note that ExternalIPs are not managed by Kubernetes. The cluster administrator has to configure the routing which maps the ExternalIP address to one of the nodes.



ExternalIP

5. ExternalName - **ExternalName** is a special *ServiceType*, that has no Selectors and does not define any endpoints. When accessed within the cluster, it

returns a **CNAME** record of an externally configured Service. The primary use case of this *ServiceType* is to make externally configured Services like **my-database.example.com** available to applications inside the cluster.

Deployment

When defining both **Readiness** and **Liveness Probes**, it is recommended to allow enough time for the **Readiness Probe** to possibly fail a few times before a pass, and only then check the **Liveness Probe**. If **Readiness** and **Liveness Probes** overlap there may be a risk that the container never reaches ready state.

Liveness probe checks on an application's health, and if the health check fails, **kubelet** restarts the affected container automatically.

Liveness Probes can be set by defining:

- Liveness command

```
apiVersion: v1kind: Pod

metadata:

  labels:

    test: liveness

  name: liveness-execspec:

  containers:

  - name: liveness

    image: k8s.gcr.io/busybox

    args:

    - /bin/sh

    - -c

    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

- Liveness HTTP request

```
livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

- TCP Liveness Probe.

```
livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

Sometimes, applications have to meet certain conditions before they can serve traffic. These conditions include ensuring that the depending service is ready, or acknowledging that a large dataset needs to be loaded, etc. In such cases, we use **Readiness Probes** and wait for a certain condition to occur. Only then, the application can serve traffic.

A Pod with containers that do not report ready status will not receive traffic from Kubernetes Services.

```
readinessProbe:

  exec:

    command:

    - cat

    - /tmp/healthy  initialDelaySeconds: 5

  periodSeconds: 5
```

```
$ minikube service web-service
```

Above command will launch webserver app

Volume Management

All data stored inside a container is deleted if the container crashes. However, the **kubelet** will restart it with a clean slate, which means that it will not have any of the old data.

To overcome this problem, Kubernetes uses Volumes. A Volume is essentially a directory backed by a storage medium. The storage medium, content and access mode are determined by the Volume Type.

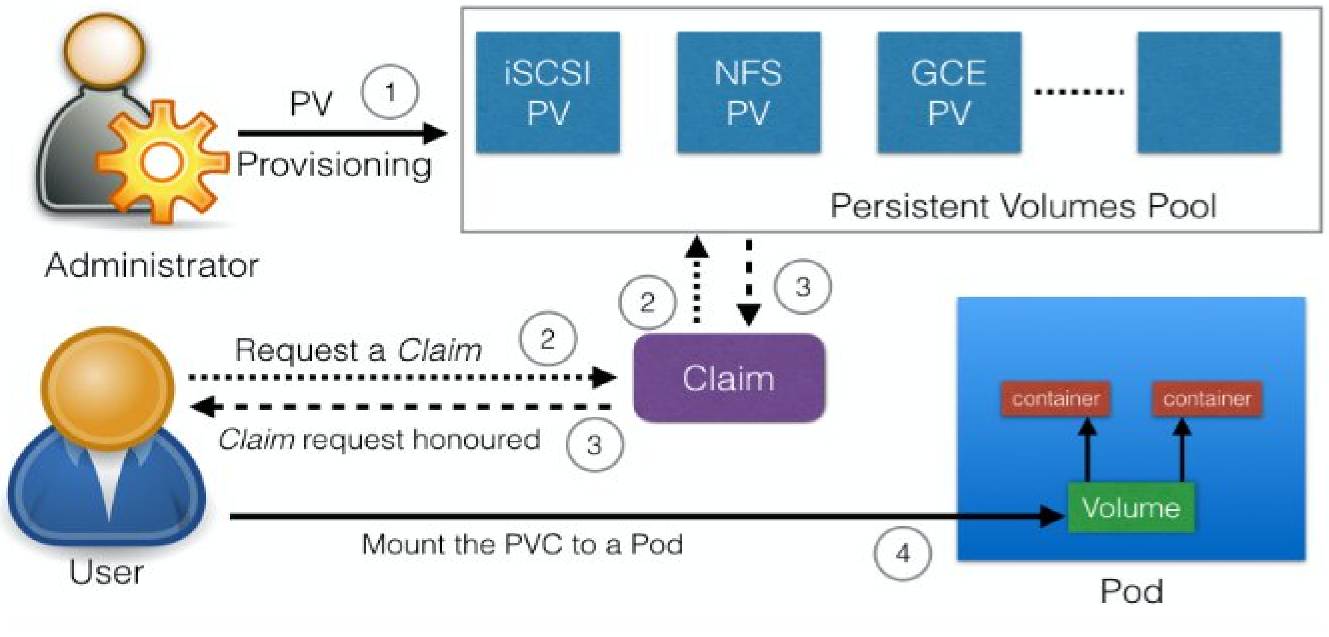
In Kubernetes, a Volume is attached to a Pod and can be shared among the containers of that Pod. The Volume has the same life span as the Pod, and it outlives the containers of the Pod - this allows data to be preserved across container restarts.

A directory which is mounted inside a Pod is backed by the underlying Volume Type. A Volume Type decides the properties of the directory, like size, content, default access modes, etc. Some examples of Volume Types are:

Volume Type	**Description**
emptyDir	An **empty** Volume is created for the Pod as soon as it is scheduled on the worker node. The Volume's life is tightly coupled with the Pod. If the Pod is terminated, the content of **emptyDir** is deleted forever.
hostPath	With the **hostPath** Volume Type, we can share a directory from the host to the Pod. If the Pod is terminated, the content of the Volume is still available on the host.
gcePersistentDisk awsElasticBlockStore azureDisk azureFile	mount GCE persistent disk, aws EBS volume, Azure Data Disk or Azure File Volume
cephfs	With **cephfs**, an existing CephFS volume can be mounted into a Pod. When a Pod terminates, the volume is unmounted and the contents of the volume are preserved.
nfs	mount nfs share
iscsi	mount iscsi share
configMap	config data mounted as files
secret	secrets mounts as files
persistentVolumeClaim	attach a persistent volume to pod via PVC.

To manage the Volume, it uses the PersistentVolume API resource type, and to consume it, it uses the PersistentVolumeClaim API resource type.

A Persistent Volume is a network-attached storage in the cluster, which is provisioned by the administrator.



PersistentVolumeClaim Used In a Pod

PersistentVolumes can be dynamically provisioned based on the StorageClass resource. A StorageClass contains pre-defined provisioners and parameters to create a PersistentVolume. Using PersistentVolumeClaims, a user sends the request for dynamic PV creation, which gets wired to the StorageClass resource.

Volume types that support managing storage using PVs are:

1. GCE Persistent Disk
2. AWS EBS
3. Azure File
4. Azure Disk
5. NFS
6. iSCSI

A **PersistentVolumeClaim (PVC)** is a request for storage by a user. Users request for PersistentVolume resources based on type, access mode, and size. There are three access modes: ReadWriteOnce (read-write by a single node), ReadOnlyMany (read-only by many nodes), and ReadWriteMany (read-write by many nodes). Once a suitable PersistentVolume is found, it is bound to a PersistentVolumeClaim.

After a successful bound, the PersistentVolumeClaim resource can be used in a Pod.

Once a user finishes its work, the attached PersistentVolumes can be released. The underlying PersistentVolumes can then be reclaimed (for an admin to verify and/or aggregate data), deleted (both data and volume are deleted), or recycled for future usage (only data is deleted).

ConfigMaps and Secrets

ConfigMaps allow us to decouple the configuration details from the container image. Using ConfigMaps, we pass configuration data as key-value pairs, which are consumed by Pods or any other system components and controllers, in the form of environment variables, sets of commands and arguments, or volumes. We can create ConfigMaps from literal values, from configuration files, from one or more files or directories.

```
$ kubectl create configmap my-config --from-literal=key1=value1 --from-literal=key2=value2
```

or

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: customer1
data:
  TEXT1: Customer1_Company
  TEXT2: Welcomes You
  COMPANY: Customer1 Company Technology Pct. Ltd.
```

Other way:

```
$ kubectl create configmap permission-config --from-file=<path/to/>permission-reset.properties
```

Inside the pod config map can be used as:

environment vars	
	```
	containers:
	- name: myapp-specific-container
	  image: myapp
	  env:
	  - name: SPECIFIC_ENV_VAR1
	    valueFrom:
	      configMapKeyRef:
	```

	name: config-map-1
	          key: SPECIFIC_DATA
	    - name: SPECIFIC_ENV_VAR2
	      valueFrom:
	        configMapKeyRef:
	          name: config-map-2
	          key: SPECIFIC_INFO

volumes	**...**
	  containers:
	  - name: myapp-vol-container
	    image: myapp
	    volumeMounts:
	    - name: config-volume
	      mountPath: /etc/config
	  volumes:
	  - name: config-volume
	    configMap:
	      name: vol-config-map
	...

It is important to keep in mind that the Secret data is stored as plain text inside **etcd**, therefore administrators must limit access to the API server and **etcd**. A newer feature allows for Secret data to be encrypted at rest while it is stored in **etcd**; a feature which needs to be enabled at the API server level.

```
$ kubectl create secret generic my-password --from-literal=password=mysqlpassword
```

```
$ kubectl create secret generic my-file-password --from-file=password.txt
```

Use secrets inside pods:

environmet vars	**....**
	spec:
	  containers:
	  - image: wordpress:4.7.3-apache
	    name: wordpress
	    env:
	    - name: WORDPRESS_DB_PASSWORD
	      valueFrom:
	        secretKeyRef:
	          name: my-password
	          key: password
	....

secrets as files	**....**
	spec:
	  containers:
	  - image: wordpress:4.7.3-apache

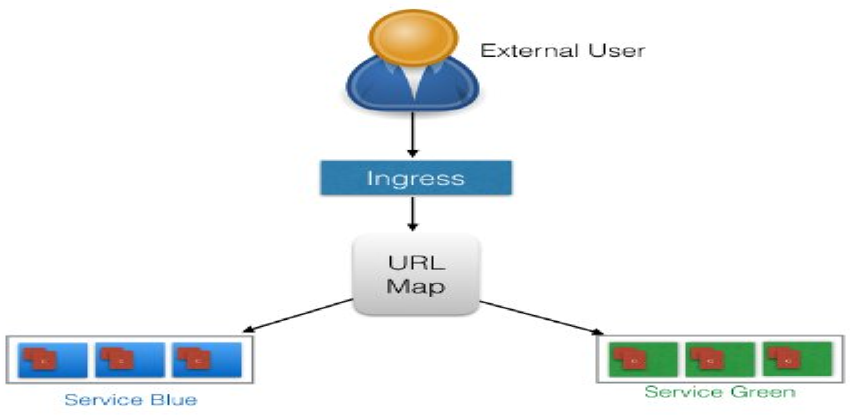
	`    name: wordpress`
	`    volumeMounts:`
	`    - name: secret-volume`
	`      mountPath: "/etc/secret-data"`
	`      readOnly: true`
	`  volumes:`
	`  - name: secret-volume`
	`    secret:`
	`      secretName: my-password`
	`....`

Ingress

"An Ingress is a collection of rules that allow inbound connections to reach the cluster Services."

To allow the inbound connection to reach the cluster Services, Ingress configures a Layer 7 HTTP/HTTPS load balancer for Services and provides the following:

- TLS (Transport Layer Security)
- Name-based virtual hosting
- Fanout routing
- Loadbalancing
- Custom rules.



Ingress

```
apiVersion: networking.k8s.io/v1beta1

kind: Ingress
```

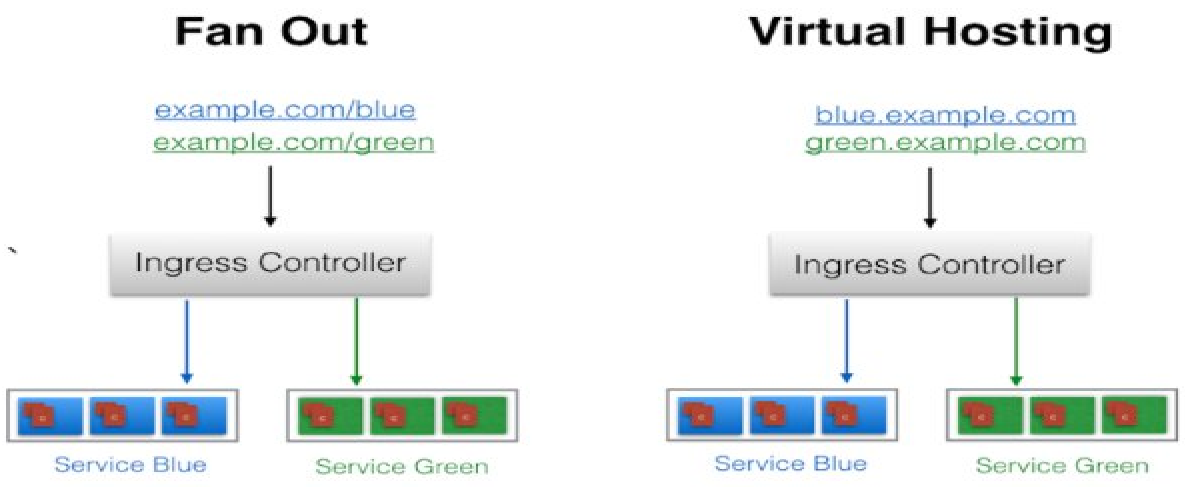
```
metadata:
  name: virtual-host-ingress
  namespace: defaultspec:
  rules:
  - host: blue.example.com
    http:
      paths:
      - backend:
          serviceName: webserver-blue-svc
          servicePort: 80
  - host: green.example.com
    http:
      paths:
      - backend:
          serviceName: webserver-green-svc
          servicePort: 80
```

In the example above, user requests to both **blue.example.com** and **green.example.com** would go to the same Ingress endpoint, and, from there, they would be forwarded to **webserver-blue-svc**, and **webserver-green-svc**, respectively. This is an example of a **Name-Based Virtual Hosting** Ingress rule.

We can also have **Fanout** Ingress rules, when requests to **example.com/blue** and **example.com/green** would be forwarded to **webserver-blue-svc** and **webserver-green-svc**, respectively:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
```

```
metadata:
  name: fan-out-ingress
  namespace: defaultspec:
  rules:
  - host: example.com
    http:
      paths:
      - path: /blue
        backend:
          serviceName: webserver-blue-svc
          servicePort: 80
      - path: /green
        backend:
          serviceName: webserver-green-svc
          servicePort: 80
```



Ingress URL Mapping

An Ingress Controller is an application watching the Master Node's API server for changes in the Ingress resources and updates the Layer 7 Load Balancer accordingly. Kubernetes supports different Ingress Controllers, and, if needed, we can also build our own. GCE L7 Load Balancer Controller and Nginx Ingress Controller are commonly used Ingress Controllers. Other controllers are Istio, Kong, Traefik, etc.

```
$ minikube addons enable ingress
```

To support enterprise-class production workloads, Kubernetes also supports auto-scaling, rolling updates, rollbacks, quota management, authorization through RBAC, package management, network and security policies, etc.

With Annotations, we can attach arbitrary non-identifying metadata to any objects, in a key-value format:

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

Unlike Labels, annotations are not used to identify and select objects.

A Job creates one or more Pods to perform a given task. The Job object takes the responsibility of Pod failures. It makes sure that the given task is completed successfully. Once the task is complete, all the Pods have terminated automatically. Job configuration options include:

- **parallelism** - to set the number of pods allowed to run in parallel;
- **completions** - to set the number of expected completions;
- **activeDeadlineSeconds** - to set the duration of the Job;
- **backoffLimit** - to set the number of retries before Job is marked as failed;
- **ttlSecondsAfterFinished** - to delay the clean up of the finished Jobs.

we can also perform Jobs at scheduled times/dates with CronJobs, where a new Job object is created about once per each execution cycle. The CronJob configuration options include:

- **startingDeadlineSeconds** - to set the deadline to start a Job if scheduled time was missed;
- **concurrencyPolicy** - to *allow* or *forbid* concurrent Jobs or to *replace* old Jobs with new ones.

We can set the following types of quotas per Namespace:

1. Compute resource quota - limit total sum of compute resource that can be requested in a given namespace.
2. Storage resource quota - limit storager resources
3. Object count quota - limit number of objects of a given type (pods, configMaps, PVC, Replication Controllers, Services, Secrets etc)

Autoscaling can be implemented in a Kubernetes cluster via controllers which periodically adjust the number of running objects based on single, multiple, or custom metrics. Types of autoscalars:

1. Horizontal Pod Autoscalar (HPA) - auto adjusts number of replicas based on CPU utilization
2. Vertical Pod autoscalar (VPA) - VPA automatically sets Container resource requirements (CPU and memory) in a Pod and dynamically adjusts them in runtime, based on historical utilization data, current resource availability and real-time events.
3. Cluster Autoscalar - auto resizes the cluster if there insufficient resource available for new pods expecting to be scheduled or when there are underutilized nodes in cluster.

In cases when we need to collect monitoring data from all nodes, or to run a storage daemon on all nodes, then we need a specific type of Pod running on all nodes at all times. A DaemonSet is the object that allows us to do just that. The **kube-proxy** agent running as a Pod on every single node in the cluster is managed by a **DaemonSet**.

A newer feature of the DaemonSet resource allows for its Pods to be scheduled only on specific nodes by configuring **nodeSelectors** and node **affinity** rules. Similar to Deployment resources, DaemonSets support rolling updates and rollbacks.

The StatefulSet controller is used for stateful applications which require a unique identity, such as name, network identifications, strict ordering, etc. For example, **MySQL cluster**, **etcd cluster**.

The **StatefulSet** controller provides identity and guaranteed ordering of deployment and scaling to Pods. Similar to Deployments, StatefulSets use ReplicaSets as intermediary Pod controllers and support rolling updates and rollbacks.

In Kubernetes, a **resource** is an API endpoint which stores a collection of API objects. For example, a Pod resource contains all the Pod objects.

Although in most cases existing Kubernetes resources are sufficient to fulfill our requirements, we can also create new resources using **custom resources**. With custom resources, we don't have to modify the Kubernetes source.

Custom resources are dynamic in nature, and they can appear and disappear in an already running cluster at any time.

To make a resource declarative, we must create and install a **custom controller**, which can interpret the resource structure and perform the required actions. Custom controllers can be deployed and managed in an already running cluster.

There are two ways to add custom resources:

- Custom Resource Definitions (CRDs)
  This is the easiest way to add custom resources and it does not require any programming knowledge. However, building the custom controller would require some programming.
- API Aggregation
  For more fine-grained control, we can write API Aggregators. They are

subordinate API servers which sit behind the primary API server. The primary API server acts as a proxy for all incoming API requests - it handles the ones based on its capabilities and proxies over the other requests meant for the subordinate API servers.

To deploy an application, we use different Kubernetes manifests, such as Deployments, Services, Volume Claims, Ingress, etc. Sometimes, it can be tiresome to deploy them one by one. We can bundle all those manifests after templatizing them into a well-defined format, along with other metadata. Such a bundle is referred to as *Chart*. These Charts can then be served via repositories, such as those that we have for **rpm** and **deb** packages.

Helm is a package manager (analogous to **yum** and **apt** for Linux) for Kubernetes, which can install/update/delete those Charts in the Kubernetes cluster.

Helm has two components:

- A client called *helm*, which runs on your user's workstation
- A server called *tiller*, which runs inside your Kubernetes cluster.

The client *helm* connects to the server *tiller* to manage Charts.

https://github.com/helm/charts - has charts submitted to kubernetes

Security Contexts allow us to set Discretionary Access Control for object access permissions, privileged running, capabilities, security labels, etc. However, their effect is limited to the individual Pods and Containers where such context configuration settings are incorporated in the **spec** section.

In order to apply security settings to multiple Pods and Containers cluster-wide, we can define Pod Security Policies. They allow more fine-grained security settings to control the usage of the host namespace, host networking and ports, file system groups, usage of volume types, enforce Container user and group ID, root privilege escalation, etc.

Network Policies are sets of rules which define how Pods are allowed to talk to other Pods and resources inside and outside the cluster. Pods not covered by any **Network Policy** will continue to receive unrestricted traffic from any endpoint.

Network Policies are very similar to typical Firewalls. They are designed to protect mostly assets located inside the Firewall but can restrict outgoing traffic as well based on sets of rules and policies.

The **Network Policy** API resource specifies **podSelectors**, *Ingress* and/or *Egress* **policyTypes**, and rules based on source and destination **ipBlocks** and **ports**. As a good practice, it is recommended to define a default deny policy to block all traffic to and from the Namespace, and then define sets of rules for specific traffic to be allowed in and out of the Namespace.

In Kubernetes, we can collect logs from different cluster components, objects, nodes, etc. Unfortunately, however, Kubernetes does not provide cluster-wide logging by default, therefore third party tools are required to centralize and aggregate cluster logs. The most common way to collect the logs is using Elasticsearch, which uses fluentd with custom configuration as an agent on the nodes. **fluentd** is an open source data collector, which is also part of CNCF.

Exam Info

Tools to know: YAML, BASH and VI
Prepare a cheat sheet of commands used for both docker and kubectl
Using kubernetes.io/docs is allowed in exam.

To have kubectl explain sections of yaml file:

```
 kubectl explain pods.spec
```

Exam Duration: 19 problems in 2 hrs - getting 17 or so done in the time is ok

Add the below to bashrc or come up with your own shortcuts to save time typing during the exam:

```
alias kp='kubectl get pods'
alias ks='kubectl get services'
alias kn='kubectl get ns'

function kdesc() {
        kubectl describe pod $1
}
function kexec() {
        kubectl exec $1 -it -- /bin/sh
}


function kdelete() {
        kubectl delete pod $1 --grace-period=0 --force
}
function krun() {
        kubectl run $1 --image=$2 --restart=Never --dry-run -o yaml
}
function kapply() {
        kubectl apply -f $1
}
function khelp() {
 echo 'kp ks kn kexec kdelete krun kapply kdesc'
}

function klogs() {
 kubectl logs $1 -f
}
```

To run commands on specific cluster:

```
kubectl config set-context <context-of-question> --namespace <namespace of
question>
```

Some commands have shortcuts:
For eg. use ns for namespace:

kubectl get ns
Same for persistentvolumeclaim use pvc instead.

kubectl describe pvc claim
For service use svc:
kubectl get svc

Kubenetes object full name	**Alias**
pod	po
service	svc
deployment	deploy
namespace	ns
persistentvolume	pv
persistentvolumeclaim	pvc
cronjob	cj
serviceaccount	sa
endpoint	ep

Dont wait for graceful deletion of objects if some mistake is done...
during exam just kill the object (use --force and 0 as grace period)

```
kubectl delete pod nginx --grace-period=0 --force
```

-- make a shortcut for it

Practice bash scripting:
1. Write if conditions
2. Write loop constructs

```
while true; do echo $(date) >> ~/tmp/date.txt; sleep 5; done;
```

Practice with kubectl -- this is the key to cracking the exam.

General structure of a kubernetes object in YAML:

- API version,
- Kind - Pod, Deployment, Quota etc.
- Metadata - Label
- Spec - Desired state
- Status - actual state

Create object from command line:
1. kubectl
2. yaml - start with yaml and use it to create object

2 ways to create objects:

1. imperative way: Fast but no track record,
kubectl create namespace ckad
kubectl run nginx --image=nginx --restart=Never -n ckad
kubectl edit pod/nginx -n ckad

Note: For exam, creating objects imperatively is faster (kubectl create/run)

2. Declarative way: create yaml and apply
there is track record of who changed, and how changes progressed.
yaml config can be version controlled.

vi nginx-pod.yaml
kubectl create -f nginx-pod.yaml

3. Hybrid approach.
Generate yaml with kubectl (in dry-run mode so object is not created) but make further edits to the basic object YAML
and then apply declaratively.

cat nginx-pod.yaml

```
----
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: nginx
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Never
```

```
status: {}
----

Edit the generated yaml file then create object declaratively.
kubectl apply -f nginx-pod.yaml
```

PODS
Single container pods are common.
Pod creation flow:
kubectl > API Server on master > etcd (in-memory DB) >scheduler (master) > kubelet (worker) > POD is created > docker container(s) is run within POD.

Lifecycle phases of Pod

1. Pending - Pod accepted.
2. Running - Pod has running container and can service request.
3. Succeeded - Finished execution successfully.
4. Failed - Finished execution with failure.
5. Unknown - unknown state

```
kubectl describe pods nginx -- gives the current status of pod

kubectl get pods nginx -o yaml -- gets the staus of pods as yaml
```

Injecting Environment Vars or running commands can be done by editing the yaml spec section for pod yaml.

```
kubectl exec nginx -it -- /bin/sh  (will shell into Pod)
```

created bashrc function ks to shell into named pod

Exercise:
https://github.com/bmuschko/ckad-prep/blob/master/1-core-concepts.md#creating-a-pod-and-inspecting-it

Create the namespace ckad-prep.

>>
```
k create ns ckad-prep
```

In the namespace ckad-prep create a new Pod named mypod with the image nginx:2.3.5. Expose the port 80.
>>
```
k run mypod --image=nginx:2.3.5 --port=80 -n ckad-prep --restart=Never
pod/mypod created
```

Identify the issue with creating the container. Write down the root cause of issue in a file named pod-error.txt.
>>
```
k describe pods/mypod -n ckad-prep
Warning  Failed     58s (x4 over 2m22s)  kubelet, minikube  Failed to pull image
"nginx:2.3.5": rpc error: code = Unknown desc = Error response from daemon: manifest
for nginx:2.3.5 not found: manifest unknown: manifest unknown
```

Change the image of the Pod to nginx:1.15.12.
>>
```
k edit pods/mypod -n ckad-prep
```

List the Pod and ensure that the container is running.
>>
```
k get pods -n ckad-prep
NAME    READY   STATUS    RESTARTS   AGE
mypod   1/1     Running   0          4m37s
```

Log into the container and run the ls command. Write down the output. Log out of the container.
>>
```
k exec pods/mypod -n ckad-prep -it -- /bin/bash
root@mypod:/# ls
bin   dev  home  lib64 mnt  proc  run srv  tmp  var
boot  etc  lib media opt  root  sbin  sys  usr
```

Retrieve the IP address of the Pod mypod.
>>
```
k get pods/mypod -n ckad-prep -o yaml | grep hostIP
  hostIP: 192.168.39.99
podIP: 172.17.0.7
```

or

```
k get pods -n ckad-prep -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP           NODE       NOMINATED
NODE   READINESS GATES
mypod   1/1     Running   0          11m   172.17.0.7   minikube   <none>           <none>
```

Run a temporary Pod using the image busybox, shell into it and run a wget command against the nginx Pod using port 80.

```
>>
$ k run busybox --image=busybox -n ckad-prep --restart=Never -it --rm -- /bin/sh

If you don't see a command prompt, try pressing enter.
/ # wget -O- 172.17.0.7:80
Connecting to 172.17.0.7:80 (172.17.0.7:80)
writing to stdout
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

```
-                    100% |********************************|   612  0:00:00 ETA
written to stdout
/ # exit
pod "busybox" deleted
```

Render the logs of Pod mypod.

>>

```
kubectl logs mypod -n ckad-prep
```

Delete the Pod and the namespace.

>>

```
$ k delete pods/mypod -n ckad-prep
pod "mypod" deleted
$ k delete ns ckad-prep
namespace "ckad-prep" deleted
```

Configuration

Configuration data - define configMap or secret
ComfigMaps let us decouple the config data from the pod definition so they can be shared between different pods.

inject runtime configuration - environment variables or reference configMap objects.
Secrets - build on top of configMap; values are encoded in base64

Imperative approach of creating config maps:

```
k create configmap db-config --from-literal=db=staging

# Single file with environment variables
k create configmap db-config --from-env-file=config.env

# File or directory
k create configmap db-config --from-file=config.txt
```

Declarative approach:
config map yaml example:

```
apiVersion: v1
data:
  db: staging
  user: jdoe
kind: ConfigMap
metadata:
  name: db-config
  namespace: default
```

o/p abbreviated.
The above yaml can be used with apply to create configmap declaratively.

Consuming configmaps in a pod
1. injected as environment variables
2. mounted as volume

In definition for pod we can add reference to configmap:

```
kind: Pod
metadata:
  name: backend
spec:
  containers:
    - image: nginx
      name: backend
      envFrom:
        - configMapRef:
            name: db-config
```

We can verify the configmap did get used by the pod's container:
k exec nginx -it -- env

The env command will dump all environment variables in the container which should include the ones injected from configMap db-config.

ConfigMap being mounted as volume:

```
kind: Pod
metadata:
  name: backend
spec:
  containers:
    - image: nginx
      name: backend
      volumeMounts:
```

```
    - name: config-volume
      mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: db-config

$ k exec backend -it -- /bin/sh
$ ls /etc/config
db
username
$ cat /etc/config/db
staging
```

Each configMap key becomes a file in the mounted volume within the container.
The value of the env variable key is the content of the file.

Exercise for configmaps

Configuring a Pod to Use a ConfigMap

Create a new file named config.txt with the following environment variables as key/value pairs on each line.
>> vi config.txt

DB_URL equates to localhost:3306
DB_USERNAME equates to postgres

Create a new ConfigMap named db-config from that file.
>>

```
$ k create configmap db-config --from-env-file=config.txt
configmap/db-config created
```

Create a Pod named backend that uses the environment variables from the ConfigMap and runs the container with the image nginx.
>>
Use the tip to generate yaml file in dry-run mode for pod and then edit it to reference configmap and apply.

```
$ k run backend --image=nginx --restart=Never --dry-run -o yaml > backend.yaml
$ cat backend.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: backend
  name: backend
spec:
  containers:
  - image: nginx
    name: backend
    envFrom:
      - configMapRef:
          name: db-config
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}
$ k apply -f backend.yaml
```

Shell into the Pod and print out the created environment variables. You should find DB_URL and DB_USERNAME with their appropriate values.

>>

```
$ k exec backend -it -- /bin/sh
# env
DB_URL=localhost:3306
KUBERNETES_SERVICE_PORT=443
KUBERNETES_PORT=tcp://10.96.0.1:443
HOSTNAME=backend
HOME=/root
PKG_RELEASE=1~buster
DB_USERNAME=postgres
TERM=xterm
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
NGINX_VERSION=1.17.6
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
KUBERNETES_PORT_443_TCP_PORT=443
NJS_VERSION=0.3.7
KUBERNETES_PORT_443_TCP_PROTO=tcp
```

```
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_SERVICE_HOST=10.96.0.1
PWD=/
#
```

Secrets
Imperative approach:

```
k create secret generic db-creds --from-env-file=secret.env
k create secret generic db-creds --from-file=ssh-private-key=~/.ssh/id_rsa
```

Declarative approach:

```
echo -n "val" | base64
dmFs

Put the encoded val to secrets yaml:
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  pwd: dmFs

Then create the secrets declaratively using above yaml.
k apply -f secrets.yaml

Mounting secrets as volume in pod yaml:
apiVersion: v1
kind: Pod
metadata:
  name: backend
spec:
  containers:
    - name: backend
      image: nginx
      volumeMounts:
        - name: secret-volume
          mountPath: /etc/secret
  volumes:
    - name: secret-volume
```

```
      secret:
        secretName: mysecret

k exec backend -it -- /bin/sh
# ls /etc/secrets
pwd
# cat pwd
val

The value is in plain text inside the container.
```

Exercise

Configuring a Pod to Use a Secret

Create a new Secret named db-credentials with the key/value pair db-password=passwd.

>>

```
k create secret generic db-credentials --from-literal=db-password=passwd
```

Create a Pod named backend that defines uses the Secret as environment variable named DB_PASSWORD and runs the container with the image nginx.

>>

```
$ k run backend --image=nginx -o yaml --restart=Never --dry-run > backend.yaml
$ cat backend.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: backend
  name: backend
spec:
  containers:
  - image: nginx
    name: backend
    env:
      - name: DB_PASSWORD
        valueFrom:
          secretKeyRef:
            name: db-credentials
            key: db-password
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}
```

Shell into the Pod and print out the created environment variables. You should find DB_PASSWORD variable.

```
>>
k exec backend -it -- /bin/sh
# env
KUBERNETES_SERVICE_PORT=443
KUBERNETES_PORT=tcp://10.96.0.1:443
HOSTNAME=backend
HOME=/root
PKG_RELEASE=1~buster
TERM=xterm
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
NGINX_VERSION=1.17.6
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
KUBERNETES_PORT_443_TCP_PORT=443
NJS_VERSION=0.3.7
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_SERVICE_PORT_HTTPS=443
DB_PASSWORD=passwd
KUBERNETES_SERVICE_HOST=10.96.0.1
PWD=/
#
```

Security Context
Used to define access control or privileges for containers.

```
apiVersion: v1
kind: Pod
metadata:
  name: secured-pod
spec:
  securityContext: -- defined at pod level (applies to all containers in the pod)
    runAsUser: 1000
  containers:
  - securityContext: -- defined at container level
    runAsGroup: 3000
```

Exercise
Creating a Security Context for a Pod

Create a Pod named secured that uses the image nginx for a single container.
Mount an emptyDir volume to the directory /data/app.
Files created on the volume should use the filesystem group ID 3000.
>>

```
$ k run secured --image=nginx --restart=Never --dry-run -o yaml > secured.yaml
$ cat secured.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: secured
  name: secured
spec:
  securityContext:
    fsGroup: 3000
  volumes:
  - name: sec-ctx-vol
    emptyDir: {}
  containers:
  - image: nginx
    name: secured
    volumeMounts:
    - name: sec-ctx-vol
      mountPath: /data/app
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}

$ k create -f secured.yaml
pod/secured created
```

Get a shell to the running container and create a new file named logs.txt in the directory /data/app. List the contents of the directory and write them down.

```
>>
s k exec secured -it -- /bin/sh
# cd /data/app
# touch log.txt
# ls -al
-rw-r--r-- 1 root 3000    0 Dec 18 06:18 log.txt
# exit
```

Resource boundary definition
# of pods, cpu and memory usage per namespace.

Declarative creation of resource quota boundaries:

```
kind: ResourceQuota
metadata:
  name: app
spec:
  hard:
    pods: "2"
    requests.cpu: "2"
    requests.memory: 500m

k create -f rq.yaml -n ckad-prep
k describe quota -n rq-demo
Name:            app
Namespace:       rq-demo
Resource         Used  Hard
--------         ----  ----
pods             1     2
requests.cpu     500m  2
requests.memory  100m  500m
```

This will **create the resource quota boundary for the given namespace**.
It is also required to specify resource quota limits at the container level in pod definition yaml if the namespace has resource quota boundaries defined.

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - image: nginx
    name: mypod
    resources:
      requests:
        cpu: "0.5"
        memory: "200m"
```

We can define both upper and lower limits. Above example defines the lower limit only. So the pod definition means it requires at least 0.5 cpu and 200m of memory to execute.

Exercise

Defining a Pod’s Resource Requirements
Create a resource quota named apps under the namespace rq-demo using the following YAML definition in the file rq.yaml.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: app
spec:
  hard:
    pods: "2"
    requests.cpu: "2"
    requests.memory: 500m
```

```
>>
k create ns rq-demo
namespace/rq-demo created
$ k create -f rq.yaml -n rq-demo
resourcequota/app created
rwatsh@rwatsh-ThinkPad-T490:~/work/learn_k8s$ k get quota -n rq-demo
NAME   CREATED AT
app    2019-12-18T06:30:16Z
```

Create a new Pod that exceeds the limits of the resource quota requirements. Write down the error message.

```
cat mypod.YAML:
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: mypod
  name: mypod
spec:
  containers:
  - image: nginx
    name: mypod
    resources:
      requests:
        memory: "10Gi"
        cpu: "500m"
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}
```

```
$ k create -f mypod.yaml -n rq-demo
Error from server (Forbidden): error when creating "mypod.yaml": pods "mypod" is forbidden: exceeded quota: app, requested: pods=1,requests.memory=10Gi, used: pods=2,requests.memory=0, limited: pods=2,requests.memory=500m
```

Change the request limits to fulfill the requirements to ensure that the Pod could be created successfully. Write down the output of the command that renders the used amount of resources for the namespace.

```
rwatsh@rwatsh-ThinkPad-T490:~/work/learn_k8s$ k create -f mypod.yaml -n rq-demo
pod/mypod created
```

Service accounts

The account name to be used for making api requests.

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: app
spec:
  serviceAccountName: myserviceaccount
```

Exercise

Using a Service Account

Create a new service account named backend-team.

```
>>
$ k create serviceaccount backend-team
serviceaccount/backend-team created
rwatsh@rwatsh-ThinkPad-T490:~/work/learn_k8s$ k get serviceaccount
NAME           SECRETS   AGE
backend-team   1         7s
default        1         6d2h
```

Print out the token for the service account in YAML format.

```
>>
s$ k get serviceaccount backend-team -o yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: "2019-12-18T06:36:26Z"
  name: backend-team
  namespace: default
  resourceVersion: "157835"
  selfLink: /api/v1/namespaces/default/serviceaccounts/backend-team
  uid: 03a2d4cb-e269-46dd-9722-3d967984b567
secrets:
- name: backend-team-token-2qpvs
```

Create a Pod named backend that uses the image nginx and the identity backend-team for running processes.

```
>>
$ k run backend --image=nginx --serviceaccount=backend-team --restart=Never
pod/backend created
```

Get a shell to the running container and print out the token of the service account.

```
k exec backend -it -- /bin/sh
cd /var/run/secrets/kubernetes.io/serviceaccount
cat token
```

Multi-container pods

1. Shared container lifecycle - like sharing volume between containers.

Example pod definition with multiple containers:

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-container
spec:
  containers:
  - image: nginx
    name: container1
  - image: nginx
    name: container2
```

Dump logs of container c1:

```
k logs busybox --container=c1
```

Login to c2 container:

```
k exec busybox -it -c c2 -- /bin/sh
```

Design patterns:

1. **Sidecar pattern** - o/p logs from main container which sidecar will process and do something.
2. **Init containers** -

some initialization logic before main app container starts.
one or many init containers in a multi container pod - seed data, file permissions etc
first init container is successful then following init container will run.
init containers will be re-run on of pod so they should be made idempotent.

3. **Adapter** - used for cases where another service accessing this service requires a special tranformation in the o/p that it understands. Adapter can keep the logic for adapting the o/p of app's o/p for the service decoupled from the app's logic.

4. **Ambassador** - proxy between app container and external service

Observability

http get requests or bash command or tcp connection to define these probes.

Readiness probe: is application ready to serve request (is it done with initialization)
We dont want to send any traffic to the pod unless it is ready.

Example of pod with readiness probe:

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: web-app
    image: eshop:4.6.3
    readinessProbe:
      httpGet:
        path: /
        port: 8080
      initialDelaySeconds: 10
      periodSeconds: 5
```

  initialDelaySeconds - is initial delay before the first readiness check is done.
  periodSeconds - is delay between readiness checks.

Liveness probe: If the pod is healthy and responsive.

Example pod definition with liveness probe.

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: web-app
    image: eshop:4.6.3
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 10
      periodSeconds: 5
```

Kubernetes (kubelet) can restart the pod when liveness probe detects pod is unhealthy.

Exercise

Defining a Pod’s Readiness and Liveness Probe

- Create a new Pod named hello with the image bonomat/nodejs-hello-world that exposes the port 3000. Provide the name nodejs-port for the container port.
  kubectl run hello --image=bonomat/nodejs-hello-world --restart=Never --port=3000 -o yaml --dry-run > pod.yaml
- Add a Readiness Probe that checks the URL path / on the port referenced with the name nodejs-port after a 2 seconds delay. You do not have to define the period interval.
- Add a Liveness Probe that verifies that the app is up and running every 8 seconds by checking the URL path / on the port referenced with the name nodejs-port. The probe should start with a 5 seconds delay.

```
apiVersion: v1kind: Podmetadata:

 creationTimestamp: null

 labels:

  run: hello

 name: hellospec:

 containers:
 - image: bonomat/nodejs-hello-world

  name: hello

  ports:

  - name: nodejs-port

    containerPort: 3000

  readinessProbe:

    httpGet:

      path: /

      port: nodejs-port

    initialDelaySeconds: 2

  livenessProbe:

    httpGet:

      path: /

      port: nodejs-port

    initialDelaySeconds: 5

    periodSeconds: 8

  resources: {}

 dnsPolicy: ClusterFirst

 restartPolicy: Neverstatus: {}
```

- Shell into container and curl localhost:3000. Write down the output. Exit the container.
- Retrieve the logs from the container. Write down the output.

```
 $ kubectl create -f pod.yamlpod/hello created
$ kubectl exec hello -it -- /bin/sh
/ # curl localhost:3000<!DOCTYPE html><html><head>
        <title>NodeJS Docker Hello World</title>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link href="http://cdn.bootcss.com/bootstrap/3.3.2/css/bootstrap.min.css" rel="stylesheet">
        <link href="/stylesheets/styles.css" rel="stylesheet"></head><body>
        <div class="container">
                <div class="well well-sm">
```

```
                        <h2>This is just a hello world message</h2>
                        <img a href="./cage.jpg"/>
                        <img src="src/cage.jpg" alt="Smiley face" width="640">
                </div>
        </div></body></html>
/ # exit

$ kubectl logs pod/hello
Magic happens on port 3000
```

Fixing a Misconfigured Pod

1. Create a new Pod with the following YAML.

```
apiVersion: v1kind: Podmetadata:
 creationTimestamp: null
 labels:
   run: failing-pod
 name: failing-podspec:
 containers:
 - args:
   - /bin/sh
   - -c
   - while true; do echo $(date) >> ~/tmp/curr-date.txt; sleep
     5; done;
   image: busybox
   name: failing-pod
   resources: {}
 dnsPolicy: ClusterFirst
 restartPolicy: Neverstatus: {}
```

```
$ kubectl create -f pod.yaml
```

1. Check the Pod's status. Do you see any issue?

```
$ kubectl logs failing-podUnable to write file!
/bin/sh: 1: cannot create /root/tmp/curr-date.txt: Directory nonexistent
Unable to write file!
/bin/sh: 1: cannot create /root/tmp/curr-date.txt: Directory nonexistent
Unable to write file!
/bin/sh: 1: cannot create /root/tmp/curr-date.txt: Directory nonexistent
```

1. Follow the logs of the running container and identify an issue.

```
$ kubectl logs failing-podUnable to write file!
/bin/sh: 1: cannot create /root/tmp/curr-date.txt: Directory nonexistent
Unable to write file!
/bin/sh: 1: cannot create /root/tmp/curr-date.txt: Directory nonexistent
Unable to write file!
/bin/sh: 1: cannot create /root/tmp/curr-date.txt: Directory nonexistent
```

1. Fix the issue by shelling into the container. After resolving the issue the current date should be written to a file. Render the output.

```
$ kubectl exec failing-pod -it -- /bin/sh
/ # mkdir -p ~/tmp

/ # cd ~/tmp

/ # ls -ltotal 4
-rw-r--r-- 1 root root 112 May  9 23:52 curr-date.txt

/ # cat ~/tmp/curr-date.txt
Thu May 9 23:59:11 UTC 2019/ # exit
```

Pod Design

Labels

determine selection of an object assigned in metadata section of the object.

```
apiVersion: v1
kind: Pod
metadata:
    name: pod1
    labels:
        tier: backend
        env: prod
        app: myapp
spec:
....
```

kubectl get pods —show-labels

kubectl get pods -l run=backend

where, -l is shortform of —selector.

Labels can be used to filter the o/p of kubectl get.

1. Equality based
   a. =

   kubeclt get pods -l tier=frontend,env=dev (works like AND operation; returns pods with labels tier=frontend and env=dev)

   kubctl get pods -l version (returns all pods with version as label key)

2. Set based

   kubectl get pods -l 'tier in (frontend,backend),env=dev' (gets pods in set frontend or backend and is dev environment)

soec.selector can be used select an object within a resource definition of some advanced resource types like Service or Job.

Examples of selector for equality based and set based selections.

```
apiVersion: v1
kind: Service
metdata:
   name: app-service
..
spec:
...
    selector:
         tier: frontend
         env: dev
```

```
apiVersion: batch/v1
kind: Job
metadata:
     name: myjob
spec:
....
      selector:
           matchLabels:
                  version: v2.1
            matchExpressions:
                    - (key: tier, operator: in values: [frontend, backend])
```

Annotations

are not queryable but are similar to labels

```
apiVersion: v1
kind: Pod
metadata:
    name: mypod
    annotations:
        commit: 866a8dc
        author: rwatsh
        branch: 'work/bugfix'
spec:
....
```

kubectl describe pods mypod — will o/p Annotations

Exercise

Defining and Querying Labels and Annotations

1. Create three different Pods with the names frontend, backend and database that use the image nginx.

```
$ kubectl run frontend --image=nginx --restart=Never --labels=env=prod,team=shiny
pod/frontend created
$ kubectl run backend --image=nginx --restart=Never --labels=env=prod,team=legacy,app=v1.2.4
pod/backend created
$ kubectl run database --image=nginx --restart=Never --labels=env=prod,team=storage
pod/database created
```

2. Declare labels for those Pods as follows:

frontend: env=prod, team=shiny

backend: env=prod, team=legacy, app=v1.2.4

database: env=prod, team=storage

3. Declare annotations for those Pods as follows:

frontend: contact=John Doe, commit=2d3mg3

backend: contact=Mary Harris

```
$ k edit pod frontend

apiVersion: v1kind: Podmetadata:
  annotations:
    commit: 2d3mg3
    contact: John Doe
  name: frontend
```

4. Render the list of all Pods and their labels.

```
$ kubectl get pods —show-labels
NAME                 READY   STATUS     RESTARTS   AGE    LABELS
backend               1/1     Running     0         7d7h   app=v1.2.4,env=prod,team=legacy
frontend             1/1     Running     0         7d7h   env=prod,team=shiny
```

5. Use label selectors on the command line to query for all production Pods that belong to the teams shiny and legacy.

```
kp -l 'team in (shiny,legacy)',env=prod
NAME      READY   STATUS    RESTARTS   AGE    LABELS
backend    1/1     Running   0          7d7h   app=v1.2.4,env=prod,team=legacy
```

```
frontend   1/1     Running   0          7d7h   env=prod,team=shiny
```

6. Remove the label env from the backend Pod and rerun the selection.

```
kp -l 'team in (shiny,legacy)',env=prod
NAME       READY   STATUS    RESTARTS   AGE    LABELS
frontend   1/1     Running   0          7d7h   env=prod,team=shiny
```

7. Render the surrounding 3 lines of YAML of all Pods that have annotations.

```
k get pods -o yaml | grep -C 3 'annotations:'
- apiVersion: v1
  kind: Pod
  metadata:
    annotations:
      contact: Mary Harris
    creationTimestamp: "2020-01-05T00:07:08Z"
    labels:
--
```

Deployments

Deployment manages creating **Pods** by means **of** ReplicaSets. What it boils down to is that **Deployment** will create **Pods** with spec taken from the template. It is rather unlikely that you will ever need to create **Pods** directly **for** a production use-case

Scaling and replication features for a set of Pods

Deployment > ReplicaSet > Pods 1..n

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

Edit the deploy.yaml

kubectl create -f deploy.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: my-deploy
  name: my-deploy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-deploy
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: my-deploy
    spec:
      containers:
      - image: nginx
        name: nginx
        resources: {}
status: {}
```

k describe deploy my-deploy

k get replicasets

Replicasets are automatically created by deployment and not meant to be modified.

Deployments enable rolling updates. There is no downtime as each replica is upgraded to new version.

```
kubectl rollout history deploy my-deploy
deployment.apps/my-deploy
REVISION  CHANGE-CAUSE
1         <none>
```

Above command can be used to check the history of rolling updates performed for the deployment.

To know what changed in a certain revision:

k rollout history deploy my-deploy —revision=1

To rollback to older version:

1. **Go to last version**

k rollout undo deployments my-deploy

k rollout status deployments my-deploy

2. Go to a specific previous revision:

k rollout undo deploy my-deploy —to-revision=1

Scaling a Deployment

Manual Scaling

k scale deploy my-deploy —replicas=4

Auto scaling

Based on CPU utilization threshold

k autoscale deploy my-deploy —cpu-percent=70 —min=1 —max=10

k get hpa my-deploy

where hpa is short for “Horizaontal pod autoscaler”

Exercise:

1. Create a Deployment named deploy with 3 replicas. The Pods should use the nginx image and the name nginx. The Deployment uses the label tier=backend. The Pods should use the label app=v1.

```
k create deployment deploy --image=nginx --dry-run -o yaml > deploy.yaml
vi deploy.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    tier: backend
  name: deploy
spec:
  replicas: 3
  selector:
    matchLabels:
      app: v1
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: v1
    spec:
      containers:
      - image: nginx
        name: nginx
        resources: {}
status: {}
```

```
k create -f deploy.yaml
```

2. List the Deployment and ensure that the correct number of replicas is running.

```
k get deploy
```

3. Update the image to nginx: latest.

```
k set image deployment deploy nginx=nginx:latest
deployment.apps/deploy image updated
```

4. Verify that the change has been rolled out to all replicas.

```
k rollout history deploy deploy
deployment.apps/deploy
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

5. Scale the Deployment to 5 replicas.

```
k scale deployments deploy --replicas=5
deployment.apps/deploy scaled
```

6. Have a look at the Deployment rollout history.

```
k rollout history deploy deploy
deployment.apps/deploy
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

7. Revert the Deployment to revision 1.

```
k rollout undo deployment deploy --to-revision=1
deployment.apps/deploy rolled back

k rollout history deploy deploy
deployment.apps/deploy
REVISION  CHANGE-CAUSE
2         <none>
3         <none>

k rollout history deploy deploy --revision=3
deployment.apps/deploy with revision #3
Pod Template:
  Labels:       app=v1
           pod-template-hash=5cd9f9d486
  Containers:
   nginx:
    Image:       nginx
    Port:        <none>
    Host Port:  <none>
    Environment:       <none>
    Mounts:     <none>
  Volumes:     <none>
```

8. Ensure that the Pods use the image nginx.

Jobs and CronJobs

Pods - run apps for ever (infinitive process)

Jobs - run a one-time process

CronJob - Periodically run process

Imperative way to create a job:

```
k run counter —image=nginx —restart=OnFailure — /bin/sh -c ‘counter=0; while [ $counter -lt
3 ]; do counter=$((counter+1));  echo “$counter”; sleep 3; done;’
```

Declarative way:

```
k create job counter —image=nginx —dry-run -o yaml — /bin/sh -c ‘counter=0; while [ $counter -
lt 3 ]; do counter=$((counter+1));  echo “$counter”; sleep 3; done;’ > job.yaml
```

Edit job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  creationTimestamp: null
  name: counter
spec:
  template:
    metadata:
      creationTimestamp: null
    spec:
      containers:
      - command:
        - /bin/sh
        - -c
        - counter=0; while [ $counter -lt 3 ]; do counter=$((counter+1));  echo "$counter";
          sleep 3; done;
        image: nginx
        name: counter
        resources: {}
      restartPolicy: Never
status: {}
```

k create -f job.yaml

k get jobs

k describe job counter

Types of Jobs

1. Non-parallel - completes as soon as its pod terminates succesfully
2. Parallel with fixed completion count -
3. Parallel with work queue -

Restart policies:

1. Never - creates a new pod on failure
2. OnFailure - restarts the container within the same pod but does not restart the pod

cronjobs - schedule tasks

Exercise:

1. Create a CronJob named current-date that runs every minute and executes the shell command ”Current date: ${date}”.

```
k run current-date --schedule="* * * * *" --restart=OnFailure --image=nginx -- /bin/sh -c
'echo "Current dae: $(date)"'
kubectl run --generator=cronjob/v1beta1 is DEPRECATED and will be removed in a
future version. Use kubectl run --generator=run-pod/v1 or kubectl create instead.
cronjob.batch/current-date created
```

2. Watch the jobs as they are being scheduled.

```
k get cronjobs -w
NAME          SCHEDULE    SUSPEND   ACTIVE   LAST SCHEDULE   AGE
current-date   * * * * *   False     1        5s              86s
```

3. Identify one of the Pods that ran the CronJob and render the logs.

```
k get pods —show-labels
current-date-1578817020-6khgt   0/1     Completed   0          51s     controller-
```

```
uid=27021d8f-5d4e-4b66-a0c8-f05c65ab3f85,job-name=current-date-
1578817020,run=current-date
```

4. Determine the number of successful executions the CronJob will keep in its history.

```
kubectl get cronjobs current-date -o yaml | grep successfulJobsHistoryLimit:
  successfulJobsHistoryLimit: 3
```

5. Delete the Job.

```
k delete cronjob current-date
cronjob.batch "current-date" deleted
```

Services and Networking

Service

Exercise:

1. Create a deployment named myapp that creates 2 replicas for Pods with the image nginx. Expose the container port 80.

```
k run my-app --image=nginx --restart=Always --replicas=2 --port=80
kubectl run --generator=deployment/apps.v1 is DEPRECATED and will be removed in a
future version. Use kubectl run --generator=run-pod/v1 or kubectl create instead.
deployment.apps/my-app created

k get deploy
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
my-app    2/2     2            2           23s
```

2. Expose the Pods so that requests can be made against the service from inside of the cluster.

```
➜  learn_k8s k expose deploy my-app --target-port=80
service/my-app exposed
➜  learn_k8s k get svc
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP   31d
my-app       ClusterIP   10.96.137.100   <none>        80/TCP    7s
```

3. Create a temporary Pods using the image busybox and run a wget command against the IP of the service.

```
k run tmp --image=busybox --restart=Never -it --rm -- wget -O- 10.96.137.100:80
Connecting to 10.96.137.100:80 (10.96.137.100:80)
writing to stdout
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
-                    100% |********************************|   612  0:00:00 ETA
written to stdout
pod "tmp" deleted
```

4. Change the service type so that the Pods can be reached from outside of the cluster.

```
➜  learn_k8s k edit svc my-app
service/my-app edited
...
spec:
  type: NodePort
...

➜  learn_k8s k get svc
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP        31d
```

```
my-app       NodePort    10.96.137.100   <none>        80:30288/TCP   3m45s
```

5. Run a wget command against the service from outside of the cluster.

6. (Optional) Can you expose the Pods as a service without a deployment?

Network Policies

Exercise:

Let's assume we are working on an application stack that defines three different layers: a frontend, a backend and a database. Each of the layers runs in a Pod. You can find the definition in the YAML file app-stack.yaml. The application needs to run in the namespace app-stack.

```
kind: PodapiVersion: v1metadata:
  name: frontend
  namespace: app-stack
  labels:
    app: todo
    tier: frontendspec:
  containers:
    - name: frontend
      image: nginx

---

kind: PodapiVersion: v1metadata:
  name: backend
  namespace: app-stack
  labels:
    app: todo
    tier: backendspec:
  containers:
    - name: backend
      image: nginx

---

kind: PodapiVersion: v1metadata:
  name: database
  namespace: app-stack
  labels:
    app: todo
    tier: databasespec:
  containers:
    - name: database
      image: mysql
      env:
      - name: MYSQL_ROOT_PASSWORD
        value: example
```

1. Create the required namespace.

```
k create ns app-stack
kubens app-stack
```

2. Copy the Pod definition to the file app-stack.yaml and create all three Pods. Notice that the namespace has already been defined in the YAML definition.

```
k apply -f app-stack.yaml
pod/frontend created
pod/backend created
pod/database created
```

3. Create a network policy in the YAML file app-stack-network-policy.yaml.

```
k create -f app-stack-network-policy.yaml
networkpolicy.networking.k8s.io/app-stack-network-policy created

cat app-stack-network-policy.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: app-stack-network-policy
  namespace: app-stack
spec:
  podSelector:
    matchLabels:
      app: todo
      tier: database
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: todo
          tier: backend
    ports:
    - protocol: TCP
      port: 3306
```

4. The network policy should allow incoming traffic from the backend to the database but disallow incoming traffic from the frontend.
5. Incoming traffic to the database should only be allowed on TCP port 3306 and no other port.

```
k get networkpolicy
NAME                       POD-SELECTOR             AGE
app-stack-network-policy   app=todo,tier=database   89s
```

State Persistence

Volumes

Persistent Volumes

1. Create a Persistent Volume named pv, access mode ReadWriteMany, storage class name shared, 512MB of storage capacityv and the host path /data/config.

```
apiVersion: v1kind: PersistentVolumemetadata:
  name: pvspec:
  capacity:
    storage: 512m
  accessModes:
    - ReadWriteMany
  storageClassName: shared
  hostPath:
    path: /data/config

k create -f pv.yaml
persistentvolume/pv created

k get pv
NAME   CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM   STORAGECLASS
REASON   AGE
pv     512m       RWX            Retain           Available           shared                64s
```

2. Create a Persistent Volume Claim named pvc that requests the Persistent Volume in step 1. The claim should request 256MB. Ensure that the Persistent Volume Claim is properly bound after its creation.

```
kind: PersistentVolumeClaimapiVersion: v1metadata:
  name: pvcspec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 256m
  storageClassName: shared

k create -f pvc.yaml
persistentvolumeclaim/pvc created

 k get pvc
NAME   STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pvc    Bound    pv       512m       RWX            shared         33s

k get pv
NAME   CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM           STORAGECLASS
REASON   AGE
pv     512m       RWX            Retain           Bound    app-stack/pvc   shared                3m55s
```

3. Mount the Persistent Volume Claim from a new Pod named app with the path /var/app/config. The Pod uses the image nginx.

```
➜  learn_k8s k create -f app.yaml
pod/app created
➜  learn_k8s cat app.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: app
  name: app
spec:
  containers:
  - image: nginx
    name: app
    volumeMounts:
      - mountPath: "/var/app/config"
        name: configpvc
    resources: {}
  volumes:
    - name: configpvc
      persistentVolumeClaim:
        claimName: pvc
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}
```

4. Check the events of the Pod after starting it to ensure that the Persistent Volume was mounted properly.

```
k describe pod app

k exec app -it — /bin/sh
# ls /var/app/config
/var/app/config
```

Shortcuts

alias k=kubectl

k config set-context <context-name> —namespace=<namespace>

where, context will select the cluster and namespace will select the namespace within that cluster.

To delete objects forcefully -

k delete pod nginx —grace-period=0 —force

Ingress vs LB:
Ingress can front multiple services and has the feature of path based routing.
LB can be used for a single service hence it is costlier to use for multiple services as we need an LB per service.

Exercises

Important references for Kubectl

https://kubernetes.io/docs/reference/kubectl/cheatsheet/

Core Concepts

https://github.com/dgkanatsios/CKAD-exercises/blob/master/a.core_concepts.md

- Tasks -> Accessing Multiple Clusters
- Tasks -> Accessing Cluster with API
- Tasks -> Port Forwarding
- Tasks -> Shell to Running Container (exec)

```
k create ns mynamespace

k run nginx —image=nginx —restart=Never -n mynamespace
k delete po nginx —grace-period=0 —force -n mynamespace

k run nginx —image=nginx —restart=Never  -n mynamespace -o yaml —dryn-run >
nginx.yaml
k create -f nginx.yaml -n mynamespace

k run busybox —image=busybox —restart=Never -n mynamespace -it —rm — env
k delete po busybox

k run busybox —image=busybox —restart=Never -o yaml —dry-run — env > busybox.yaml
k apply -f busybox.yaml -n mynamespace
k logs busybox -n mynamespace


k create ns myns -o yaml —dry-run

# https://kubernetes.io/docs/concepts/policy/resource-quotas/
k create quota myrq —hard=cpu=1,memory=1G,pods=2 —dry-run -o yaml

k get po -A
or k get po —all-namespaces

k run nginx —image=nginx —restart=Never -n mynamespace —port=80

# https://kubernetes.io/docs/reference/kubectl/cheatsheet/
# k set image pod <podName> <containerName>=<imageName>:<tag>
k set image pod nginx nginx=nginx:1.7.1
k describe po nginx -n mynamespace
k get po nginx -n mynamespace -o json | jq '.spec.containers[].image'


k get po nginx -n mynamespace -o wide

k run temp --image=busybox --restart=Never -it --rm -- wget -O - http://172.17.0.15:80

# Pod yaml without cluster-specific info can be retrieved using export
k get po nginx -n mynamespace —export

# Get logs from pod's previous instance
k logs nginx -n mynamespace -p

k exec nginx -n mynamespace -it — /bin/sh

k run busybox —image=busybox -n mynamespace —restart=Never -it —rm — echo "hello
world"

k run nginx --image=nginx --restart=Never -n mynamespace --env=var1=val1 -it —rm —- env
k exec nginx -n mynamespace -it — env
```

Multi-container Pods

https://github.com/dgkanatsios/CKAD-exercises/blob/master/b.multi_container_pods.md

```
k run busybox --image=busybox --restart=Never -o yaml --dry-run -- /bin/sh -c  "echo hello;
sleep 3600" > multicont.yaml

cat multicont.yaml

apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: busybox
  name: busybox
spec:
  containers:
  - args:
    - /bin/sh
    - -c
    - echo hello; sleep 3600
    image: busybox
    name: busybox1
    resources: {}
  - args:
    - /bin/sh
    - -c
    - echo hello; sleep 3600
    image: busybox
    name: busybox2
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}

k create -f multicont.yaml

k exec busybox -it -c busybox2 -- ls
```

Pod Design

https://github.com/dgkanatsios/CKAD-exercises/blob/master/c.pod_design.md

- Concepts -> Assign Pods to Nodes - Selectors

- Concepts -> Labels and Selectors
- Tasks -> ReplicaSet Rolling Updates
- Concepts -> Deployments, Rollouts, and Rollbacks

```
k run nginx1 --image=nginx --restart=Never -n mynamespace --labels=app=v1
k run nginx2 --image=nginx --restart=Never -n mynamespace --labels=app=v1
k run nginx3 --image=nginx --restart=Never -n mynamespace --labels=app=v1

k get po -n mynamespace —show-labels

k edit po nginx2 -n mynamespace
# Edit the label app=v2

k get po -n mynamespace -l=app

k get po -n mynamespace -l=app=v2

# Delete label app from all 3 pods
k label po nginx1 nginx2 nginx3 app-

# Check pod spec help
k explain po.spec

k run podnodeselector --image=busybox --restart=Never -o yaml --dry-run >
podnodeselector.yaml

cat podnodeselector.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: podnodeselector
  name: podnodeselector
spec:
  containers:
  - image: busybox
    name: podnodeselector
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Never
  nodeSelector:
    accelerator: nvidia-tesla-p100
status: {}


k create -f podnodeselector.yaml

k get po
```

```
NAME             READY   STATUS          RESTARTS   AGE     LABELS
podnodeselector   0/1     **Pending**         0          5s      run=podnodeselector

k describe po podnodeselector

Events:
...
Warning  FailedScheduling  <unknown>  default-scheduler  0/1 nodes are available: 1 node(s)
didn't match node selector.
...

# Add annotation
k annotate po nginx1 nginx2 nginx3 description='my description' -n mynamespace

k get po nginx1 -n mynamespace -o jsonpath='{.metadata.annotations}'

# Delete annotation
k annotate po nginx1 nginx2 nginx3 description- -n mynamespace

# Delete all pods
k delete po nginx1 nginx2 nginx3 -n mynamespace
```

Deployments

```
k create deployment nginxdeploy --image=nginx:1.7.8 --dry-run -o yaml > nginxdeploy.yaml
cat nginxdeploy.yaml

kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: nginxdeploy
  name: nginxdeploy
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginxdeploy
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: nginxdeploy
    spec:
      containers:
      - image: nginx:1.7.8
        name: nginx
        ports:
        - containerPort: 80
        resources: {}
status: {}

k create -f nginxdeploy.yaml
k get deploy nginxdeploy
k get rs -l app=nginxdeploy
k get po -l app=nginxdeploy
k rollout status deploy nginxdeploy

k set image deploy nginxdeploy nginx=nginx1.7.9
k rollout history deploy nginxdeploy
k get deploy nginxdeploy
k get rs
k get po
k rollout history deploy nginxdeploy --revision=2

k rollout undo deploy nginxdeploy
k rollout history deploy nginxdeploy --revision=3

k set image deploy nginxdeploy nginx=nginx1.91
k get deploy nginxdeploy -o yaml
k rollout status deploy nginxdeploy



k rollout undo deploy nginxdeploy --to-revision=2
k rollout history deploy nginxdeploy --revision=5

k scale deploy nginxdeploy --replicas=5
k describe deploy nginxdeploy

k autoscale deploy nginxdeploy --min=5 --max=10 --cpu-percent=80
k rollout pause deploy nginxdeploy
k set image deploy nginxdeploy nginx=nginx1.9.1
# There is no change in the revision history
k rollout history deploy nginxdeploy

k rollout resume deploy nginxdeploy
k rollout history deploy nginxdeploy
# The nginx:1.9.1 image is applied.

k delete deploy nginxdeploy
```

Jobs

```
k create job job1 —image=perl — perl -Mbignum=bpi -wle ‘print bpi(2000)’
k get jobs -w
```

```
k delete job job1

k create job hellojob —image=busybox — /bin/sh -c ‘echo hello;sleep 30;echo world;’
# get the pod for the job
k get po
# get the logs for the job’s pod
k logs hellojob-w8gnq -f
k get jobs job/hellojob

k delete job hellojob

# Run a job with timeout - spec.activeDeadlineSeconds
kubectl create job busybox --image=busybox --dry-run -o yaml -- /bin/sh -c 'while true; do echo hello; sleep 10;done' > job.yaml
vi job.yaml

apiVersion: batch/v1
kind: Job
metadata:
  creationTimestamp: null
  name: hellojob
spec:
  activeDeadlineSeconds: 30
  template:
    metadata:
      creationTimestamp: null
    spec:
      containers:
      - command:
        - /bin/sh
        - -c
        - while true; do echo hello; sleep 10;done
        image: busybox
        name: hellojob
        resources: {}
      restartPolicy: Never
status: {}

# Run a job 5 times - spec.completions = 5
k create job hellojob --image=busybox --dry-run -o yaml  -- /bin/sh -c  'echo hello;sleep 10;echo world' > hellojob.yaml

cat hellojob.yaml
apiVersion: batch/v1
kind: Job
metadata:
  creationTimestamp: null
  name: hellojob
spec:
  completions: 5
  template:
    metadata:
      creationTimestamp: null
    spec:
      containers:
      - command:
        - /bin/sh
        - -c
        - echo hello;sleep 10;echo world
        image: busybox
        name: hellojob
        resources: {}
      restartPolicy: Never
status: {}

k create -f hellojob.yaml
k get job

# To make the jobs run in parallel - spec.parallelism = 5
cat hellojob.yaml
apiVersion: batch/v1
kind: Job
metadata:
  creationTimestamp: null
  name: hellojob
spec:
  parallelism : 5
  template:
    metadata:
      creationTimestamp: null
    spec:
      containers:
      - command:
        - /bin/sh
        - -c
        - echo hello;sleep 10;echo world
        image: busybox
        name: hellojob
        resources: {}
      restartPolicy: Never
status: {}

k create -f hellojob.yaml
```

CronJob

- Tasks -> Automated Tasks with Cron Jobs
- Tasks -> Parallel Jobs with Expansions
- Tasks -> Course Parallel Processing with a Work Queue
- Tasks -> Fine Parallel Processsing with a Work Queue

```
k create cronjob cronjob1 —image=busybox — schedule=”*/1 * * * *” — /bin/sh -c ‘date; echo
Hello from the Kubernetes cluster’
k get cj
k get job -w
k delete cj cronjob1
```

Configurations

- Task -> Config Maps
- Task -> Security Contexts
- Tasks -> Assigning Memory Resources to Pods
- Tasks -> Assigning CPU Resources to Pods
- Tasks -> Pod QOS
- Tasks -> Credentials using Secrets
- Tasks -> Project Volume w/Secrets
- Tasks -> Setting Service Account

ConfigMaps

```
k create configmap config --from-literal=foo=lala --from-literal=foo2=lolo
k get cm
k describe cm config

k create cm config2 --from-file=conf.txt
k describe cm config2

k create cm config3 --from-env-file=config.env
k describe cm config3

kubectl create cm configmap4 --from-file=special=config4.txt

k run cmpod --image=nginx --restart=Never --dry-run -o yaml > cmpod.yaml
cat cmpod.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: cmpod
  name: cmpod
spec:
  containers:
```

```
  - image: nginx
    name: cmpod
    resources: {}
    env:
        # Define the environment variable
        - name: option
          valueFrom:
            configMapKeyRef:
              # The ConfigMap containing the value you want to assign to
SPECIAL_LEVEL_KEY
              name: options
              # Specify the key associated with the value
              key: var5
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}

k create -f cmpod.yaml
k exec cmpod -it — /bin/sh -c ‘env | grep option’

k create cm anotherone --from-literal=var6=val6 --from-literal=var7=val7
k run cmpod2 --image=nginx --restart=Never --dry-run -o yaml > cmpod2.yaml
cat cmpod2.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: cmpod2
  name: cmpod2
spec:
  containers:
  - image: nginx
    name: cmpod2
    resources: {}
    envFrom:
      - configMapRef:
          name: anotherone
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}

k create -f cmpod2.yaml
k exec cmpod2 -it -- env

k create configmap cmvolume --from-literal=var8=val8 --from-literal=var9=val9
k run cmpod3 --image=nginx --restart=Never --dry-run -o yaml > cmpod3.yaml
cat cmpod3.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: cmpod3
  name: cmpod3
spec:
  containers:
  - image: nginx
    name: cmpod3
    resources: {}
    volumeMounts:
      - name: config-volume
        mountPath: /etc/lala
  volumes:
    - name: config-volume
      configMap:
        # Provide the name of the ConfigMap containing the files you want
        # to add to the container
        name: cmvolume
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}

k create -f cmpod3.yaml
k exec cmpod3 -it -- /bin/sh -c 'cd /etc/lala; ls'
```

SecurityContext

https://kubernetes.io/docs/tasks/configure-pod-container/security-context/

```
k run nginxpod --image=nginx --restart=Never --dry-run -o yaml > nginxpod.yaml
cat nginxpod.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: nginxpod
  name: nginxpod
spec:
  securityContext:
    runAsUser: 101
  containers:
  - image: nginx
    name: nginxpod
    securityContext:
        capabilities:
                add: ["NET_ADMIN", "SYS_TIME"]
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}
```

Requests and Limits

```
kubectl run nginx --image=nginx --restart=Never --requests='cpu=100m,memory=256Mi' --
limits='cpu=200m,memory=512Mi'
```

Secrets

```
k create secret generic mysecret --from-literal=password=mypass

echo -n admin > username
k create secret generic mysecret2 --from-file=username

k get secret mysecret2 -o yaml
k get secret mysecret2 -o jsonpath='{.data.username}{"\n"}'

k run nginx --image=nginx --restart=Never --dry-run -o yaml > nginxsecret.yaml
cat nginxsecret.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: nginx
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    resources: {}
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret2
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}

k create -f nginxsecret.yaml
k exec nginx -it -- /bin/sh
```

```
# ls /etc/foo
username
# cat /etc/foo/username
admin

cat nginxsecretenv.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: nginx
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    resources: {}
    env:
      - name: USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret2
            key: username
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}

k create -f nginxsecretenv.yaml
k exec nginx -it -- /bin/sh -c 'env | grep USERNAME'
```

ServiceAccounts

```
k get sa -A
k create sa myuser
```

Imperative:

```
k run nginx --image=nginx --restart=Never --serviceaccount=myuser
```

Declarative:

```
k run nginx --image=nginx --restart=Never --dry-run -o yaml > mynginx.yaml
cat mynginx.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: nginx
  name: nginx
```

```
spec:
  serviceAccountName: myuser
  containers:
  - image: nginx
    name: nginx
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}

k create -f mynginx.yaml

k get pod nginx -o jsonpath='{.spec.serviceAccountName}{"\n"}'
```

Observability

- [Tasks -> App Introspection and Debugging]
- [Tasks - Liveness and Readiness Probes]
- [Tasks -> Debugging Pods]
- [Tasks -> Troubleshooting Applications]
- [Tasks -> Debugging Services]
- [Tasks -> Debugging Services Locally]
- [Tasks -> Core Metrics Pipeline]

Liveness and readiness probes

https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes

```
k run nginx --image=nginx --restart=Never --dry-run -o yaml > nginxliveness.yaml
cat nginxliveness.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: nginx
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    resources: {}
    livenessProbe:
      exec:
        command:
        - ls
      initialDelaySeconds: 5
      periodSeconds: 10
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}

k create -f nginxliveness.yaml
k describe pod nginx | grep -i liveness


cat nginxreadiness.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: nginx
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    resources: {}
    readinessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 10
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}

k create -f nginxreadiness.yaml
k describe pod nginx | grep -i readiness
```

Logging

```
k run busybox --image=busybox --restart=Never -- /bin/sh -c  'i=0; while true; do echo "$i:
$(date)"; i=$((i+1)); sleep 1; done'

k logs busybox -f
```

Debugging

```
k run busybox --image=busybox --restart=Never -- /bin/sh -c  'ls /notexist'
k logs pod/busybox -f
ls: /notexist: No such file or directory

k delete pod busybox --grace-period=0 --force
```

Services and Networking

https://kubernetes.io/docs/concepts/services-networking/connect-applications-service/#exposing-pods-to-the-cluster

https://kubernetes.io/docs/concepts/services-networking/service/#defining-a-service

https://kubernetes.io/docs/tasks/administer-cluster/declare-network-policy/

https://kubernetes.io/docs/concepts/services-networking/network-policies/

- Concepts -> Connecting Apps with Services
- Tasks -> Declare Network Policy

```
k run nginx --image=nginx --restart=Never --port=80 —expose
k get svc
k get ep

k run busybox --image=busybox --restart=Never -it --rm  -- /bin/sh -c 'wget -O-
http://172.17.0.13:80'

k edit svc nginx
# Change type: ClustetIP to NodePort
k get svc # should now show NodePort

minikube ip
192.168.39.99

k run busybox --image=busybox --restart=Never -it --rm  -- /bin/sh -c 'wget -O-
http://192.168.39.99:30638'

k delete svc nginx
k delete pod nginx —grace-period=0 —force

# Create deployment, expose it as service (ClusterIP type)
k create deployment foo --image=dgkanatsios/simpleapp --dry-run -o yaml >
simpleappdeploy.yaml
cat simpleappdeploy.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: foo
  name: foo
spec:
  replicas: 3
  selector:
    matchLabels:
      app: foo
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: foo
    spec:
      containers:
      - image: dgkanatsios/simpleapp
        name: simpleapp
        resources: {}
        ports:
        - containerPort: 8080
status: {}

k get po -o wide # to get the pod IP
k run busybox --image=busybox --restart=Never -it --rm  -- /bin/sh -c 'wget -O-
http://172.17.0.2:8080'

# Expose the deployment as a service so we can access the 3 replicas via a service pod IP

k expose deploy foo --target-port=8080 --port=6262

k get ep
NAME       ENDPOINTS                                            AGE
foo        172.17.0.13:8080,172.17.0.14:8080,172.17.0.15:8080   47s

k get svc
NAME       TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)       AGE
foo        ClusterIP  10.96.111.111   <none>        6262/TCP      58s

k run busybox --image=busybox --restart=Never -it --rm  -- /bin/sh -c 'wget -O-
http://10.96.111.111:6262'
k delete svc foo
k delete deploy foo

k create deployment mynginxdeployment --image=nginx --dry-run -o yaml >
mynginxdeploy.yaml
cat mynginxdeploy.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: mynginxdeployment
  name: mynginxdeployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: mynginxdeployment
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: mynginxdeployment
    spec:
      containers:
      - image: nginx
        name: nginx
        resources: {}
        ports:
        - containerPort: 80
status: {}
k create -f mynginxdeploy.yaml
k expose deploy mynginxdeployment --port=80
cat networkpolicy.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: mynginxdeployment-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
       app: mynginxdeployment
  ingress:
  - from:
      - podSelector:
          matchLabels:
            access: 'true'

k create -f networkpolicy.yaml
k run busybox --image=busybox --restart=Never -it --rm  -- /bin/sh -c 'wget -O-
http://mynginxdeployment:80'
k delete deploy mynginxdeployment
k delete svc mynginxdeployment
```

State Persistence

- Concepts -> Persistent Volumes
- Tasks -> Configuring PVCs

https://kubernetes.io/docs/concepts/storage/volumes/

```
k run busybox --image=busybox --restart=Never --dry-run -o yaml  -- /bin/sh -c 'sleep 3600' >
busyboxmulti.yaml
cat busyboxmulti.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: busybox
  name: busybox
spec:
  volumes:
  - name: cache-volume
    emptyDir: {}
  containers:
  - args:
    - /bin/sh
    - -c
    - sleep 3600
    image: busybox
    name: busybox1
    resources: {}
    volumeMounts:
    - mountPath: /etc/foo
      name: cache-volume
  - args:
    - /bin/sh
    - -c
    - sleep 3600
    image: busybox
    name: busybox2
    resources: {}
    volumeMounts:
    - mountPath: /etc/foo
      name: cache-volume
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}
k create -f busyboxmulti.yaml
k exec busybox -it -c busybox1 -- /bin/sh
# cat /etc/passwd | cut -f 1 -d ':' > /etc/foo/passwd
# exit
k exec busybox -it -c busybox2 -- /bin/sh
/ # cat /etc/foo/passwd
```

```
root
daemon
bin
sys
sync
mail
www-data
operator
nobody
/ # exit
```

https://kubernetes.io/docs/tasks/configure-pod-container/configure-persistent-volume-storage/#create-a-persistentvolume

```
cat pv.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: myvolume
  labels:
    type: local
spec:
  storageClassName: normal
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
    - ReadWriteMany
  hostPath:
    path: "/etc/foo"
k create -f pv.yaml
k get pv
```

```
cat pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mypvc
spec:
  storageClassName: normal
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 4Gi

k create -f pvc.yaml
k get pvc
```

Notes:

1. Pod created with -it —rm will show the o/p of any command immediately after the execution and once completed the pod is deleted. If pod is created from its yaml or without —rm the pod will execute the command and show with Completed status. It will not be deleted and to inspect the o/p one can run kubectl logs <podName> command.

LFD259 - Kubernetes for Developers Course Notes

Objectives:

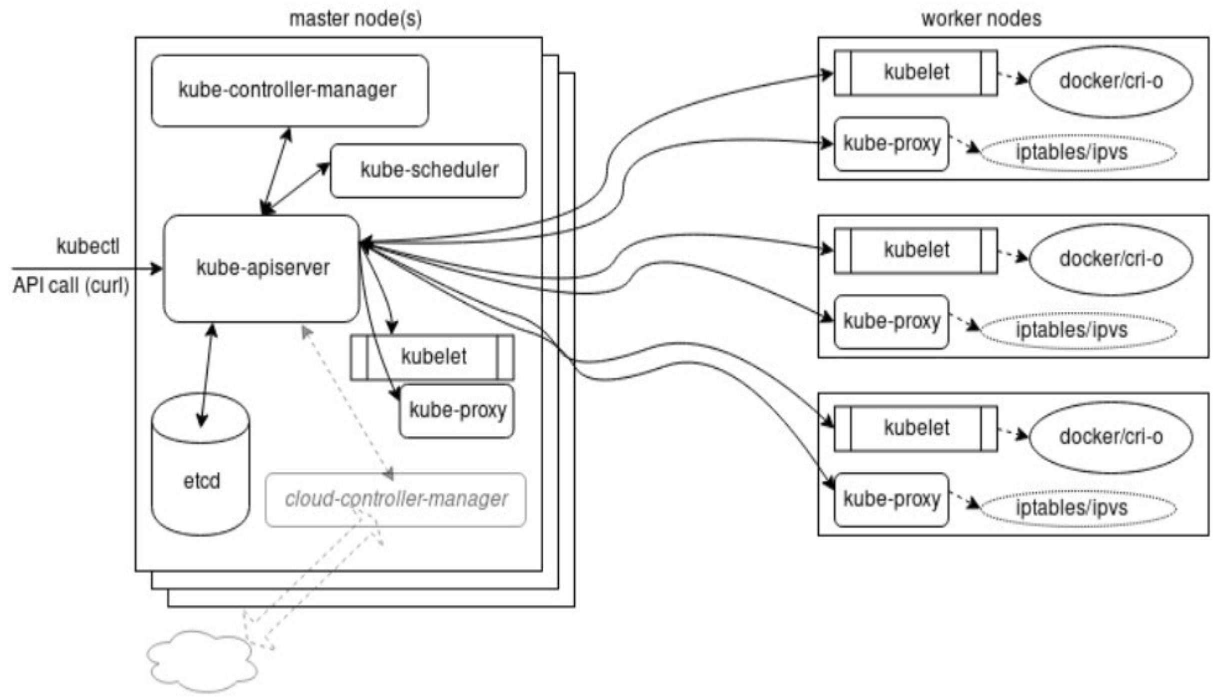
– Containerize apps
– configure deployment with ConfigMaps, Secrets, and SecurityContexts
– multi-container pod design
– probes for pod health
– update and rollback an app
– implement services and NetworkPolicies
– Use PersistentVolumeClaims for state persistence.

Prep work for the course: **Setup a 2 VM (1 master + 1 worker) k8s cluster.**

I used https://github.com/oracle/vagrant-boxes/tree/master/OLCNE. Pre-requisites are virtualbox and vagrant, then just clone the git repo and do a vagrant up.

This sets up a k8s cluster with 1 master and 2 workers by default running Oracle Enterprise Linux 7.8 with k8s 1.17.4 at the time of this writing.

Kubernetes Architecture



Kubernetes Architecture

Every node (including master) in k8s cluster runs 2 processes:

– Kubelet - it creates the containers by calling into the container runtime and watches over the resources on the node
– kube-proxy -sets up the iptables rules to expose the container ports over the network.

A Pod can contain typically one container that runs an application and other containers in the pod support the primary application. All containers of a Pod share,

– same IP address
– storage access
– namespace

The master node runs:

– Kube-apiserver - front-end for all client requests (internal and external). Only one to access etcd DB. Each API call goes through 3 steps:
    - o Authentication
    - o Authorization
    - o several admission controllers.
– etcd - B+ tree key/value store for cluster metadata. Rather than finding and changing an entry value, they are always appended to the end. Previous copies of the data are then marked for removal by a compaction process. Works with curl and provides reliable

watch queries. Simulataneous requests to update a value - only first gets honored and the following requests will be returned 409 (conflict) error code.

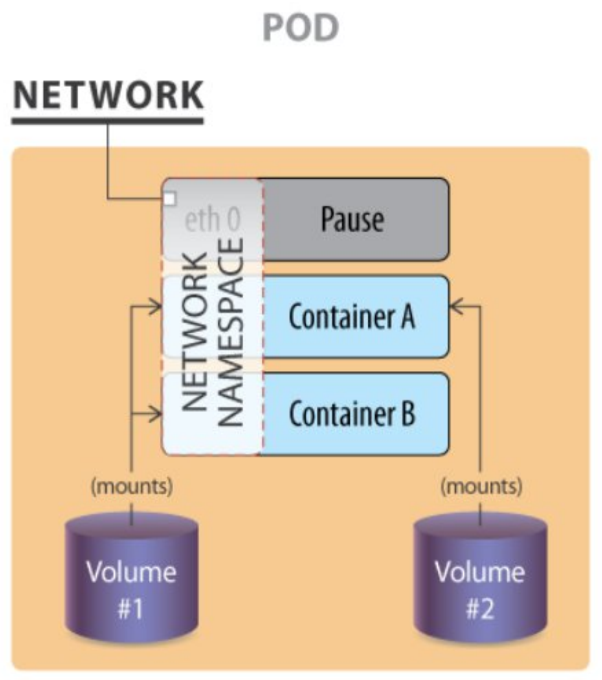
- kube-scheduler - finds the right node for Pod placement. A Pod can be bound to a node as well using node-selectors. Refer https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/scheduler.go
- kube-controller-manager - series of watch-loops that query the kube-apiserver for an object's state and modify the object until its declared state matches current state. Example controllers are:
  - Deployment - deploys a replicaset controller
  - ReplicaSet - deploys all Pod replicas, restarts the unhealthy Pods and watches over number of replicas to match declared number
  - Job - runs single task in a Pod, with retries on failure, timeout if job gets stuck and terminates the Pod when the task is done. It can also run multiple replicas of the Pod in parallel.
  - CronJob - runs a task on given schedule.
  - DarmonSet - ensures a single Pod is deloyed on each worker node - typically for logging or metrics collection.
  - StatefulSet - deploys Pods in a particular order, such that following Pods are only deployed if the previous Pods report a ready status.
  - endpoints
  - namespace
- cloud-controller-manager - interacts with agents outside of the cloud.

https://12factor.net/ - the 12 factor app principles for writing SaaS applications that can be scaled and deployed in cloud with ease.

Worker nodes:

- kubelet
- kube-proxy
- docker or cri-o
- supervisord - to monitor kubelet and dockerd (start them if thet fail)
- fluentd - for cluster-wide logging (Kubernetes natively does not yet have this).

Pods:



Pod = colocated containers + data volumes

All containers in the pod share same network namespace (hence have same IP address). Within the pod containers can communicate to each other using the loopback interface, IPC or write to files on a shared filesystem.

The *pause* container is only used to retrieve the namespaces and IP addresses.

kubectl run newpod —image=nginx —restart=Never

or

kubectl run newpod —image=nginx —generator=run-pod/v1

or

kubectl create -f newpod.yaml

kubect delete po newpod

or

kubectl delete -f newpod.yaml

Services:

Each service is a microservice handling a particular bit of traffic such as a single NodePort or a LoadBalancer to distribute inbound requests among many pods. It also handles access policies for inbound requests, useful for resource control, as well as for security.

uses selector to know which objects to connect. 2 types of selectors:

1. Equality based - filters by label keys and values.
    a. =
    b. ==
    c. !=
2. Set based - filters by a set of values:
    a. in
    b. notin
    c. exists

E.g. status notin (dev, test, maint)

Networking

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

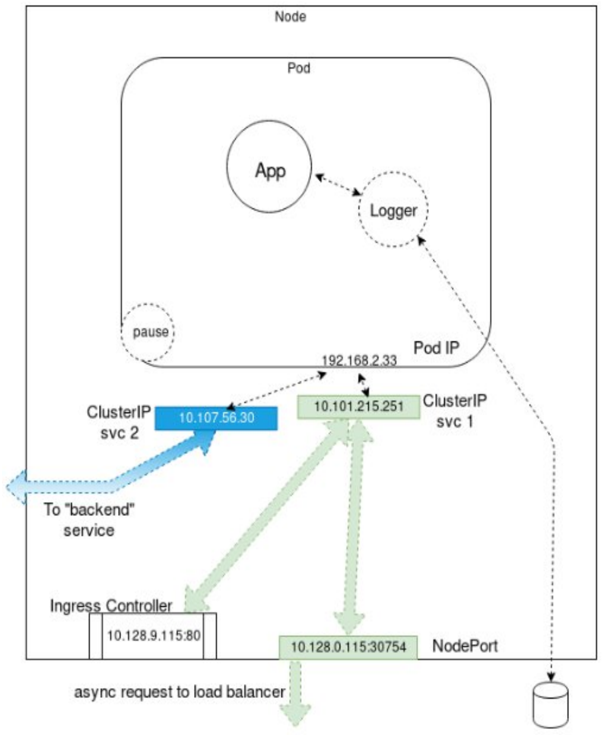
https://github.com/containernetworking/cni - CNI

Main Networking Requirements:

1. All pods can communicate with each other across nodes.
2. All nodes can communicate with all pods.
3. No NAT

3 main challenges to solve in a container orchestration system are:

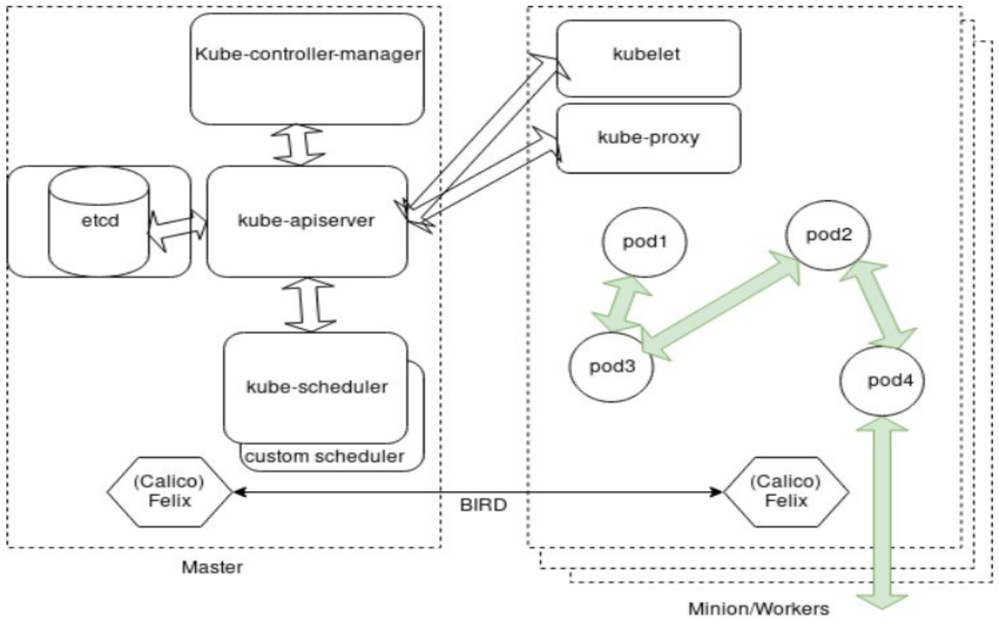
1. container-to-container communication within a Pod
2. Pod-to-pod communication
3. External-to-Pod communications.



Logger (sidecar container).

svc1 connects traffic via ingress controller or Load Balancer (using NodePort)

svc2 connects traffic internally to another Pod in the cluster.



Calico pods running on each node in the cluster uses BGP to route traffic for every Pod (L3 routed networking).

https://www.projectcalico.org/calico-networking-for-kubernetes/

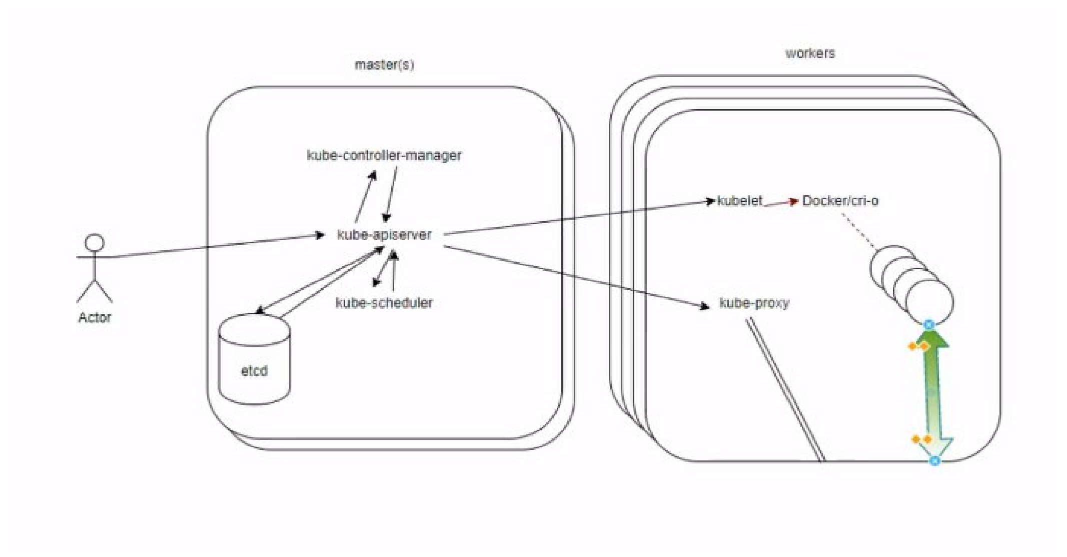
https://speakerdeck.com/thockin/illustrated-guide-to-kubernetes-networking Tim Hockin's An Illustrated Guide to Kubernetes Networking

API Call flow:

User (sends a request say to create a pod via kubectl) > Request goes to kube-apiserver > API server checks in etcd if such a pod exists > API server also checks the auth/authz for the request > API Server then sends the request to kube-controller-manager which will then return a response to API server to schedule a pod creation > API server then calls the kube-scheduler > Kube-scheduler then checks which worker node to send the request to > kube-scheduler then calls the kube-apiserver > kube-apiserver calls the kubelet on that worker node > kubelet then calls the container engine to create a pod (Docker or cri-o) > kube-apiserver also calls the kube-proxy to then setup the firewall rules for the new pod

kube-apiserver is the only one that has access to etcd

kube-controller-manager is the brain of the cluster



Further know-how materials:

https://www.gcppodcast.com/post/episode-46-borg-and-k8s-with-john-wilkes/ - talks about history of Borg and K8s.

Lab2.3 - Creating a basic pod

```
apiVersion: v1
kind: Pod
metadata:
  name: basicpod
  labels:
    type: webserver
spec:
  containers:
  - name: webcont
    image: nginx
    ports:
    - containerPort: 80
```

Service that exposes pod port 80 within the cluster:

```
apiVersion: v1
kind: Service
metadata:
  name: basicservice
spec:
  selector:
    type: webserver
  ports:
  - protocol: TCP
    port: 80
```

We can access the service at its cluster IP.

```
$ k get svc
NAME           TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
basicservice   NodePort   10.102.164.113   <none>        80:32028/TCP   7m2s

$ curl http://10.102.164.113
```

Service that exposes the pod port 80 external to the cluster using NodePort of the worker node on which the pod is located:

```
apiVersion: v1
kind: Service
metadata:
  name: basicservice
spec:
  selector:
    type: webserver
  type: NodePort
  ports:
  - protocol: TCP
    port: 80
```

Now we can access the service using its cluster IP within the cluster or from outside the cluster using the node IP and node port (32028).

```
$ curl http://10.102.164.113

$ curl http://192.168.99.101:32028
```

Lab 2.4 - Multi-container pods

```
apiVersion: v1
kind: Pod
metadata:
  name: basicpod
  labels:
    type: webserver
spec:
  containers:
  - name: webcont
    image: nginx
    ports:
    - containerPort: 80
  - name: fdlogger
    image: fluent/fluentd
```

Lab 2.5 - Create a simple deployment

```
k get deploy,rs,po,svc,ep

k create deployment firstpod --image=nginx

[vagrant@master1 s_02]$ k get deployment,pod
NAME                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/firstpod   1/1     1            1           19s

NAME                            READY   STATUS    RESTARTS   AGE
pod/firstpod-847fccb594-4lnv4   1/1     Running   0          19s
```

CRI - container runtime interface: to allow easy integration of container runtimes with kubelet.

rkt - https://cri-o.io/ has been archived as CRI-O enables any OCI compliant container runtime to be used with K8s. CRI-O implements k8s CRI. It supports runc and Kata Container runtimes but any other runtime can be plugged in. It is lightweight replacement for Docker or rkt as container runtimes. Docker runtime also supports additional swarm orchestration which does not get used when running k8s.

Containerizing an App:

1. make the app stateless if possible
2. single build artifact (like a java fat jar that includes all its dependencies or a go app)
3. remove any environmental configs (use configMaps and secrets instead)

4. buildah - https://github.com/containers/buildah - creates OCI images (can use a Docketfile optionally)
5. podman - https://github.com/containers/libpod - a replacement for docker container run command (provides LCM of a container - create, start, stop, update).

Multi-container pods:

1. share same IP and network namespace
2. share same volume
3. Patterns:
    a. Ambassador - a container in pod will be responsible for communication with a resource outside the cluster. For e.g. embed a proxy like envoy instead of using one provided by the cluster.
    b. Adapter - a container in pod modifies the data that the primary container generates.
    c. Sidecar - a container in pod which helps provide a service not available in primary container. For e.g. Logging containers (may use fluentd)

Probes in deployment:

1. readinessProbe: container will not accept any traffic until the readinessProbe returns healthy state. Types of readinessProbe are:
    a. exec - uses shell script to test readiness. Exit code = 0 is healthy.
    b. httpGet - http code 200- 399 is healthy.
    c. tcpSocket - attempt to connect to a tcp port.
    d.
2. livenessProbe: continually check the health of a container. If container fails a probe, it is terminated and a new container will be spawned.

https://kompose.io/ - converts docker compose file to kubernetes deployment yaml

kompose convert -f docker-compose.yaml -o mydeploy.yaml

Using kompose we can create a local docker registry as a pod in k8s cluster.

Security

1. **admission control** system - inspect and modify requests and accept or deny the request.
2. Network policies - can be used to define ingress rules to restrict ingress traffic between pods in different namespaces.
3. 3 steps to access k8s API:
    a. Authenticate
    b. Authorization
    c. Admission control

Exam Tips

1. Alias k=kubectl already exists. Just use it.
2. There is a set context command at the beginning of each question which is used to switch between clusters. Questions are grouped by those using same clusters appear contiguously. This is to minimize the need to switch between clusters.
3. Resource names, commands etc can be copied from the question by clicking.
4. Question and web terminal are seen side-by-side.
5. Some resources in the queston may be pre-created in the cluster and question will call this out that we don't need to create them and they already exist.
6. On Mac, copy/paste works from/to web terminal using Cmd + C/V as expected.
7. Speed in the exam matters alot. There are 19 questions and 2 hrs. So approx 6 mins per question. I failed my first attempt as i did not manage time well.
8. Practice matters alot .. even if in the real-world we will be able to figure things out in a reasonably short time by looking up in the docs. CLI help and by trying out the command but in the exam we have very limited time and hence need to know when we are giving too much time to a particular quesstion.
9. Questions have different weightages - some are harder and hence have more weightage than simpler ones.

Practice Questions

1. Get just the name column from listing of all pods.

   ```
   kubectl get po -o=name
   ```

2. Write a job which runs every second and stops in 17 seconds.

   Use spec.activeDeadlineSeconds = 17 in job spec.