Common kubectl commands	Kubectl get all
Yaml files format	apiVersion: v1 kind: Pod metadata: spec:
pods	kubectl run nginx --image=nginx apiVersion: v1 kind: Pod metadata: labels: run: nginx2 name: nginx2 spec: containers: - image: nginx name: nginx2 ports: - containerPort: 80 **command**: ["printenv"] **args**: ["HOSTNAME", "KUBERNETES_PORT"] **env**: - **name**: SERVICE_PORT **value**: "80" kubectl run hazelcast --image=hazelcast/hazelcast --port=5701 kubectl run hazelcast --image=hazelcast/hazelcast --env="DNS_DOMAIN=cluster" kubectl run hazelcast --image=hazelcast/hazelcast --labels="app=hazelcast” # Start the nginx pod using the default command, but use custom arguments (arg1 .. argN) for that command. kubectl run nginx --image=nginx -- <arg1> <arg2> ... <argN> # Start the nginx pod using a different command and custom arguments. kubectl run nginx --image=nginx --command -- <cmd> <arg1> ... <argN>
node	And update and maintenance steps Drain node in preparation for maintenance. kubectl drain node01

	`# Mark node "node01" as unschedulable.` ` kubectl cordon node01` `kubectl uncordon node01`
Replica controller	Add theory ReplicaSets are the successors to *ReplicationControllers*. The two serve the same purpose, and behave similarly, except that a ReplicationController does not support **set-based selector requirements.** Equality based Selectors: For example, if we provide the following selectors: **`env = prod`** **`tier != frontend`** Unlike Equality based, Set-based label selectors allow filtering keys according to a set of values. For example, if we provide the following selectors: **`env in (prod, qa)`** **`tier notin (frontend, backend)`** **`Partition`**
replicasets	**A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time.** As such, it is often used to guarantee the availability of a specified number of identical Pods. A ReplicaSet is defined with fields, including a selector that specifies how to identify Pods it can acquire, a number of replicas indicating how many Pods it should be maintaining, and a pod template specifying the data of new Pods it should create to meet the number of replicas criteria. `apiVersion: apps/v1` `kind: ReplicaSet` `metadata:` `  name: replicaset-1` `spec:` `  replicas: 2` `  selector:` `    matchLabels:` `      tier: frontend` `  template:` `    metadata:`

<table>
<tr><td></td><td>

```
      labels:
        tier: frontend
    spec:
      containers:
      - name: nginx
        image: nginx
```

Add theory
</td></tr>
<tr><td></td><td>kubectl scale rs new-replica-set --replicas=5</td></tr>
<tr><td>deployment</td><td>

kubectl create deployment nginx --image=nginx --replicas=4

kubectl scale deployment nginx --replicas=4

```
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:1.14.2
        ports:
        - containerPort: 80
```

```
kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1
--record
```

```
kubectl rollout status deployment/nginx-deployment
```

```
kubectl rollout history deployment.v1.apps/nginx-deployment
```

```
kubectl rollout history deployment.v1.apps/nginx-deployment
--revision=2
```

```
kubectl rollout undo deployment.v1.apps/nginx-deployment
```

```
kubectl rollout undo deployment.v1.apps/nginx-deployment
```

</td></tr>
</table>

	`--to-revision=2` `kubectl rollout pause deployment.v1.apps/nginx-deployment` `kubectl set resources deployment.v1.apps/nginx-deployment -c=nginx --limits=cpu=200m,memory=512Mi` **Strategy** specifies the strategy used to replace old Pods by new ones. `.spec.strategy.type` can be "Recreate" or "RollingUpdate". "RollingUpdate" is the default value. All existing Pods are killed before new ones are created when **`.spec.strategy.type==Recreate`.** The Deployment updates Pods in a rolling update fashion when **`.spec.strategy.type==RollingUpdate`**. You can specify `maxUnavailable` and `maxSurge` to control the rolling update process. Max Unavailable `.spec.strategy.rollingUpdate.maxUnavailable` is an optional field that specifies the maximum number of Pods that can be unavailable during the update process. The value can be an absolute number (for example, 5) or a percentage of desired Pods (for example, 10%). The absolute number is calculated from percentage by rounding down. The value cannot be 0 if `.spec.strategy.rollingUpdate.maxSurge` is 0. The default value is 25%. Max Surge `.spec.strategy.rollingUpdate.maxSurge` is an optional field that specifies the maximum number of Pods that can be created over the desired number of Pods. The value can be an absolute number (for example, 5) or a percentage of desired Pods (for example, 10%). The value cannot be 0 if `MaxUnavailable` is 0. The absolute number is calculated from the percentage by rounding up. The default value is 25%.
namespaces	kubectl get ns kubectl get po -n=research kubect run redis --image=redis -n=finance
services	Types of service: 1. nodePort 2. ClusterIP

	3. LoadBalancer Nodeport: creates a service which is accessible via node:port Port = 30,000 - 32,767 ClusterIP: creates virtual IP within a cluster. For internal communication between front end and back end LoadBalancer: provisions a load balancer, distributes load NodePort: `kubectl expose pod nginx --type=NodePort --port=80 --name=nginx-service --dry-run=client -o yaml` `kubectl create service nodeport nginx --tcp=80:80 --node-port=30080 --dry-run=client -o yaml` `apiVersion: v1` `kind: Service` `metadata:` `  name: myapp-service` `spec:` `   type: NodePort` `   ports:` `     - targetPort: 80` `       port: 80` `       nodePort: 30008` `   selector:` `     app: myapp` ClusterIP: default service type `kubectl expose pod redis --port=6379 --name redis-service --dry-run=client -o yaml` `kubectl create service clusterip redis --tcp=6379:6379 --dry-run=client -o yaml` `apiVersion: v1` `kind: Service` `metadata:` `  name: myapp-service` `spec:` `   type: ClusterIP` `   ports:` `     - targetPort: 80` `       port: 80`

	`` selector: app: myapp Headless service ``` LoadBalancer: provisions a load balancer, distributes load ``` apiVersion: v1 kind: Service metadata: name: myapp-service spec: type: LoadBalancer ports: - targetPort: 80 port: 80 nodePort: 30008 selector: app: myapp ```
Ingress	An API object that manages external access to the services in a cluster, typically HTTP. Ingress can provide load balancing, SSL termination, and name-based virtual hosting. An Ingress does not expose arbitrary ports or protocols. Exposing services other than HTTP and HTTPS to the internet typically uses a service of type [Service.Type=NodePort] or [Service.Type=LoadBalancer]. How is Ingress different from a Service? The Kubernetes Service is an abstraction over endpoints (pod-ip:port pairings). The Ingress is an abstraction over Services. This doesn't mean all Ingress controller must route *through* a Service, but rather, that routing, security and auth configuration is represented in the Ingress resource per Service, and not per pod. As long as this configuration is respected, a given Ingress controller is free to route to the DNS name of a Service, the VIP, a NodePort, or directly to the Service's endpoints. A minimal ingress resource example:

	`` apiVersion: networking.k8s.io/v1beta1 kind: Ingress metadata: name: simple-fanout-example annotations: nginx.ingress.kubernetes.io/rewrite-target: / spec: rules: - host: foo.bar.com http: paths: - path: /foo backend: serviceName: service1 servicePort: 4200 - path: /bar backend: serviceName: service2 servicePort: 8080 ``` You can secure an Ingress by specifying a secret that contains a TLS private key and certificate. ``` apiVersion: v1 kind: Secret metadata: name: testsecret-tls namespace: default data: tls.crt: base64 encoded cert ```

	`  tls.key: base64 encoded key` `type: kubernetes.io/tls` ——--------- `apiVersion: networking.k8s.io/v1beta1` `kind: Ingress` `metadata:` `  name: tls-example-ingress` `spec:` `  tls:` `  - hosts:` `    - sslexample.foo.com` `    secretName: testsecret-tls` `  rules:` `    - host: sslexample.foo.com` `      http:` `        paths:` `        - path: /` `          backend:` `            serviceName: service1` `            servicePort: 80`
Ingress Controllers	In order for the Ingress resource to work, the cluster must have an ingress controller running. Unlike other types of controllers which run as part of the `kube-controller-manager` binary, Ingress controllers are not started automatically with a cluster. Use this page to choose the ingress controller implementation that best fits your cluster. Kubernetes as a project supports and maintains AWS, GCE, and nginx ingress controllers.
configmap	A ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line

	arguments, or as configuration files in a volume. There are four different ways that you can use a ConfigMap to configure a container inside a Pod: 1. Inside a container command and args 2. Environment variables for a container 3. Add a file in read-only volume, for the application to read 4. Write code to run inside the Pod that uses the Kubernetes API to read a ConfigMap <pre>apiVersion: v1 kind: ConfigMap metadata: name: game-demo data: # property-like keys; each key maps to a simple value player_initial_lives: "3" ui_properties_file_name: "user-interface.properties"</pre> **Immutable ConfigMaps** <pre>immutable: true # Create a new configmap named my-config with specified keys instead of file basenames on disk kubectl create configmap my-config --from-file=key1=/path/to/bar/file1.txt --from-file=key2=/path/to/bar/file2.txt # Create a new configmap named my-config with key1=config1 and key2=config2 kubectl create configmap my-config --from-literal=key1=config1 --from-literal=key2=config2 # Create a new configmap named my-config from the key=value pairs in the file kubectl create configmap my-config --from-file=path/to/bar # Create a new configmap named my-config from an env file kubectl create configmap my-config --from-env-file=path/to/bar.env</pre>

<table>
<tr><td></td><td></td></tr>
<tr><td>schedule</td><td>

Manually schedule pod on node01

nodeName is the simplest form of node selection constraint

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
  nodeName: kube-01
```

Label node

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

NodeSelector

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

Taint a node

kubectl taint NODE NAME KEY_1=VAL_1:TAINT_EFFECT_1

kubectl taint nodes foo dedicated=special-user:NoSchedule

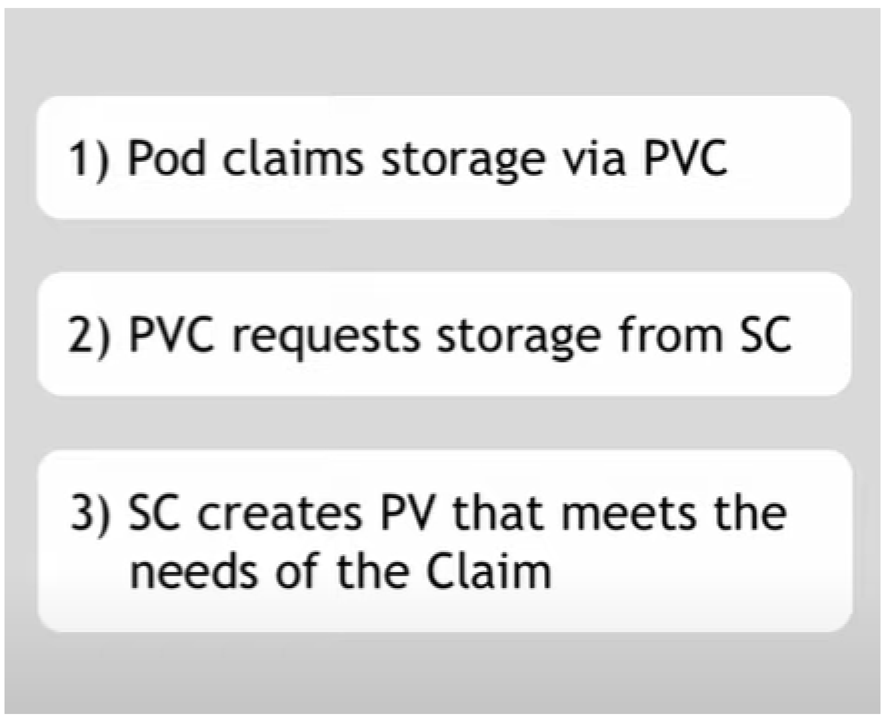
The taint effect must be **NoSchedule, PreferNoSchedule or NoExecute.**

NoSchedule: do not schedule new pods
PreferNoSchedule: prefer not scheduling new pods
NoExecute: remove existing pod that do not tolerate taint and do not schedule new pods

</td></tr>
</table>

	# Remove from node 'foo' all the taints with key 'dedicated' kubectl taint nodes foo dedicated- # Remove from node 'foo' the taint with key 'dedicated' and effect 'NoSchedule' if one exists. kubectl taint nodes foo dedicated:NoSchedule- Node Affinity `nodeSelector` provides a very simple way to constrain pods to nodes with particular labels. The affinity/anti-affinity feature, greatly expands the types of constraints you can express. The key enhancements are 1. The affinity/anti-affinity language is more expressive. The language offers more matching rules besides exact matches created with a logical AND operation; 2. you can indicate that the rule is "soft"/"preference" rather than a hard requirement, so if the scheduler can't satisfy it, the pod will still be scheduled; 3. you can constrain against labels on other pods running on the node (or other topological domain), rather than against labels on the node itself, which allows rules about which pods can and cannot be co-located There are currently two types of node affinity, called `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution`. You can think of them as "hard" and "soft" respectively, `apiVersion: v1` `kind: Pod` `metadata:` `  name: with-node-affinity` `spec:` `  affinity:` `    nodeAffinity:` `      requiredDuringSchedulingIgnoredDuringExecution:` `        nodeSelectorTerms:` `        - matchExpressions:` `          - key: kubernetes.io/e2e-az-name` `            operator: In` `            values:` `            - e2e-az1` `            - e2e-az2` `      preferredDuringSchedulingIgnoredDuringExecution:` `      - weight: 1` `        preference:` `          matchExpressions:`

	<pre> - key: another-node-label-key operator: In values: - another-node-label-value containers: - name: with-node-affinity image: k8s.gcr.io/pause:2.0</pre> Add pod affinity
secret	A Secret can contain user credentials required by pods to access a database. Managing Secrets using kubectl: <pre>kubectl create secret generic db-user-pass \ --from-literal=username=devuser \ --from-literal=password='S!B*d$zDsb='</pre> Create a Secret with files <pre>echo -n 'admin' > ./username.txt echo -n '1f2d1e2e67df' > ./password.txt kubectl create secret generic db-user-pass \ --from-file=./username.txt \ --from-file=./password.txt</pre> Managing Secrets using Configuration File: <pre>echo -n 'admin' | base64 echo -n '1f2d1e2e67df' | base64 apiVersion: v1 kind: Secret metadata: name: mysecret type: Opaque data: username: YWRtaW4= password: MWYyZDFlMmU2N2Rm</pre> Types of Secret: 1. **Opaque**: arbitrary user-defined data, Opaque is the default Secret type if omitted from a Secret configuration file. 2. **.io/service-account-token:** service account token 3. **kubernetes.io/dockercfg:** serialized ~/.dockercfg file for storing docker-username and passwords 4. **kubernetes.io/basic-auth:** credentials for basic authentication, Secret must

	contain the following two keys username and password 5. **kubernetes.io/ssh-auth:** credentials for SSH authentication 6. **kubernetes.io/tls:** data for a TLS client or server 7. **bootstrap.kubernetes.io/token:** bootstrap token data
Volumes	A `PersistentVolume` (PV) is a piece of storage in the cluster that has been provisioned by an administrator. A `PersistentVolumeClaim` (PVC) is a request for storage by a user. It is similar to a pod. Pods consume node resources and PVCs consume PV resources. While `PersistentVolumeClaims` allow a user to consume abstract storage resources, it is common that users need `PersistentVolumes`with varying properties, such as performance, for different problems. Cluster administrators need to be able to offer a variety of `PersistentVolumes` that differ in more ways than just size and access modes, without exposing users to the details of how those volumes are implemented. For these needs there is the `StorageClass` resource.  Pod > Persistent volume claim > storage class > PersistentVolume
Volumes	**Types of Volumes** awsElasticBlockStore

	`` aws ec2 create-volume --availability-zone=eu-west-1a --size=10 --volume-type=gp2 apiVersion: v1 kind: Pod metadata: name: test-ebs spec: containers: - image: k8s.gcr.io/test-webserver name: test-container volumeMounts: - mountPath: /test-ebs name: test-volume volumes: - name: test-volume # This AWS EBS volume must already exist. awsElasticBlockStore: volumeID: "<volume id>" fsType: ext4 ``` ### configMap A ConfigMap provides a way to inject configuration data into pods. The data stored in a ConfigMap can be referenced in a volume of type `configMap` and then consumed by containerized applications running in a pod. ``` apiVersion: v1 kind: Pod metadata: name: configmap-pod spec: containers: - name: test image: busybox volumeMounts: - name: config-vol mountPath: /etc/config volumes: - name: config-vol configMap: name: log-config items: - key: log_level path: log_level ``` ### emptyDir An `emptyDir` volume is first created when a Pod is assigned to a node, and

	exists as long as that Pod is running on that node. As the name says, the `emptyDir` volume is initially empty. All containers in the Pod can read and write the same files in the `emptyDir` volume, though that volume can be mounted at the same or different paths in each container. When a Pod is removed from a node for any reason, the data in the `emptyDir` is deleted permanently. ```apiVersion: v1 kind: Pod metadata: name: test-pd spec: containers: - image: k8s.gcr.io/test-webserver name: test-container volumeMounts: - mountPath: /cache name: cache-volume volumes: - name: cache-volume emptyDir: {}``` **hostPath** A `hostPath` volume mounts a file or directory from the host node's filesystem into your Pod. This is not something that most Pods will need, but it offers a powerful escape hatch for some applications. hostPath configuration example ```apiVersion: v1 kind: Pod metadata: name: test-pd spec: containers: - image: k8s.gcr.io/test-webserver name: test-container volumeMounts: - mountPath: /test-pd name: test-volume volumes: - name: test-volume hostPath: # directory location on host path: /data # this field is optional type: Directory```

<table>
<tr>
<td></td>
<td>

local

A `local` volume represents a mounted local storage device such as a disk, partition or directory.

A `local` volume represents a mounted local storage device such as a disk, partition or directory.

Local volumes can only be used as a statically created PersistentVolume. Dynamic provisioning is not supported.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
spec:
  capacity:
    storage: 100Gi
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/disks/ssd1
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - example-node
```

secret

A `secret` volume is used to pass sensitive information, such as passwords, to Pods.

</td>
</tr>
<tr>
<td>Persistent Volumes</td>
<td>A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes.
A PersistentVolumeClaim (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory).</td>
</tr>
</table>

Lifecycle of a volume and claim

Provisioning

There are two ways PVs may be provisioned: statically or dynamically.

Static

A cluster administrator creates a number of PVs.

Dynamic

When none of the static PVs the administrator created match a user's PersistentVolumeClaim, the cluster may try to dynamically provision a volume specially for the PVC. This provisioning is based on StorageClasses: the PVC must request a storage class and the administrator must have created and configured that class for dynamic provisioning to occur.

Binding

A user creates, or in the case of dynamic provisioning, has already created, a PersistentVolumeClaim with a specific amount of storage requested and with certain access modes. A control loop in the master watches for new PVCs, finds a matching PV (if possible), and binds them together. If a PV was dynamically provisioned for a new PVC, the loop will always bind that PV to the PVC.

Using

Pods use claims as volumes. The cluster inspects the claim to find the bound volume and mounts that volume for a Pod. For volumes that support multiple access modes, the user specifies which mode is desired when using their claim as a volume in a Pod.

Reclaiming

Retain

Delete

Recycle

If supported by the underlying volume plugin, the `Recycle` reclaim policy performs a basic scrub (`rm -rf /thevolume/*`) on the volume and makes it available again for a new claim.

Storage Classes	A StorageClass provides a way for administrators to describe the "classes" of storage they offer. Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators. `apiVersion: storage.k8s.io/v1` `kind: StorageClass` `metadata:` `  name: standard` `provisioner: kubernetes.io/aws-ebs` `parameters:` `  type: gp2` `reclaimPolicy: Retain` `allowVolumeExpansion: true` `mountOptions:` `  - debug` `volumeBindingMode: Immediate` Reclaim Policy `Delete` or `Retain`
Authentication	
RBAC Authorization	https://kubernetes.io/docs/reference/access-authn-authz/rbac/
Role and ClusterRole	If you want to define a role within a namespace, use a Role; if you want to define a role cluster-wide, use a ClusterRole. Here's an example Role in the "default" namespace that can be used to grant read access to pods: **kubectl create role pod-reader --verb=get --verb=list --verb=watch --resource=pods** **# Create a Role named "pod-reader" with ResourceName specified** **kubectl create role pod-reader --verb=get --resource=pods --resource-name=readablepod --resource-name=anotherpod** `apiVersion: rbac.authorization.k8s.io/v1` `kind: Role` `metadata:`

	`` namespace: default name: pod-reader rules: - apiGroups: [""] # "" indicates the core API group resources: ["pods"] verbs: ["get", "watch", "list"] ``` Here is an example of a ClusterRole that can be used to grant read access to secrets in any particular namespace, or across all namespaces (depending on how it is bound): root# k get roles -A NAMESPACE NAME CREATED AT blue developer 2021-12-14T06:25:27Z kube-public kubeadm:bootstrap-signer-clusterinfo 2021-12-14T06:16:46Z kube-public system:controller:bootstrap-signer 2021-12-14T06:16:44Z kube-system extension-apiserver-authentication-reader 2021-12-14T06:16:44Z ``` apiVersion: rbac.authorization.k8s.io/v1 kind: ClusterRole metadata: # "namespace" omitted since ClusterRoles are not namespaced name: secret-reader rules: - apiGroups: [""] # # at the HTTP level, the name of the resource for accessing Secret # objects is "secrets" resources: ["secrets"] verbs: ["get", "watch", "list"] ```
RoleBinding and ClusterRoleBinding	A role binding grants the permissions defined in a role to a user or set of users. It holds a list of *subjects* (users, groups, or service accounts), and a reference to the role being granted. A RoleBinding grants permissions within a specific namespace whereas a ClusterRoleBinding grants that access cluster-wide. ``` apiVersion: rbac.authorization.k8s.io/v1 # This role binding allows "jane" to read pods in the "default" namespace. # You need to already have a Role named "pod-reader" in that namespace. ```

	`` kind: RoleBinding metadata: name: read-pods namespace: default subjects: # You can specify more than one "subject" - kind: User name: jane # "name" is case sensitive apiGroup: rbac.authorization.k8s.io roleRef: # "roleRef" specifies the binding to a Role / ClusterRole kind: Role #this must be Role or ClusterRole name: pod-reader # this must match the name of the Role or ClusterRole you wish to bind to apiGroup: rbac.authorization.k8s.io ``` A RoleBinding can also reference a ClusterRole to grant the permissions defined in that ClusterRole to resources inside the RoleBinding's namespace. This kind of reference lets you define a set of common roles across your cluster, then reuse them within multiple namespaces. ClusterRoleBinding example ``` apiVersion: rbac.authorization.k8s.io/v1 # This cluster role binding allows anyone in the "manager" group to read secrets in any namespace. kind: ClusterRoleBinding metadata: name: read-secrets-global subjects: - kind: Group name: manager # Name is case sensitive apiGroup: rbac.authorization.k8s.io roleRef: kind: ClusterRole name: secret-reader apiGroup: rbac.authorization.k8s.io ``` https://kubernetes.io/docs/reference/access-authn-authz/rbac/
	root# kubectl auth can-i get pods --as dev-user no
Network Policies	The entities that a Pod can communicate with are identified through a combination of the following 3 identifiers: Other pods that are allowed (exception: a pod cannot block access to itself)

	Namespaces that are allowed IP blocks (exception: traffic to and from the node where a Pod is running is always allowed, regardless of the IP address of the Pod or the node) `apiVersion: networking.k8s.io/v1` `kind: NetworkPolicy` `metadata:` `  name: test-network-policy` `  namespace: default` `spec:` `  podSelector:` `    matchLabels:` `      role: db` `  policyTypes:` `  - Ingress` `  - Egress` `  ingress:` `  - from:` `    - ipBlock:` `        cidr: 172.17.0.0/16` `        except:` `        - 172.17.1.0/24` `    - namespaceSelector:` `        matchLabels:` `          project: myproject` `    - podSelector:` `        matchLabels:` `          role: frontend` `    ports:` `    - protocol: TCP` `      port: 6379` `  egress:` `  - to:` `    - ipBlock:` `        cidr: 10.0.0.0/24` `    ports:` `    - protocol: TCP` `      port: 5978` Behavior of to and from selectors There are four kinds of selectors that can be specified in an ingress from section or egress to section: podSelector: This selects particular Pods in the same namespace as the NetworkPolicy which should be allowed as ingress sources or egress destinations. namespaceSelector: This selects particular namespaces for which all Pods should be allowed as ingress sources or egress destinations. namespaceSelector and podSelector: A single to/from entry that specifies both namespaceSelector and podSelector selects particular Pods within particular namespaces. Be careful to use correct YAML syntax; this policy:

```yaml
 ...
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          user: alice
      podSelector:
        matchLabels:
          role: client
  ...
```

Default policies

Default deny all ingress traffic

```yaml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

Allow all ingress traffic

```yaml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all-ingress
spec:
  podSelector: {}
  ingress:
  - {}
  policyTypes:
  - Ingress
```

Default deny all egress traffic

```yaml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-egress
spec:
  podSelector: {}
  policyTypes:
  - Egress
```

Allow all egress traffic

```yaml
---
apiVersion: networking.k8s.io/v1
```

	kind: NetworkPolicy metadata: name: allow-all-egress spec: podSelector: {} egress: - {} policyTypes: - Egress Default deny all ingress and all egress traffic --- apiVersion: networking.k8s.io/v1 kind: NetworkPolicy metadata: name: default-deny-all spec: podSelector: {} policyTypes: - Ingress - Egress
Resource Quota	