Kubernetes

A Comprehensive Overview

Kubernetes v1.8

Introduction

Intro - What is Kubernetes?

Kubernetes or **K8s** was a project spun out of Google as a open source next-gen container scheduler designed with the lessons learned from developing and managing Borg and Omega.

Kubernetes was designed from the ground-up as a loosely coupled collection of components centered around deploying, maintaining, and scaling applications.

Intro - What Does Kubernetes do?

Kubernetes is the linux kernel of distributed systems.

It abstracts away the underlying hardware of the nodes and provides a uniform interface for applications to be both deployed and consume the shared pool of resources.

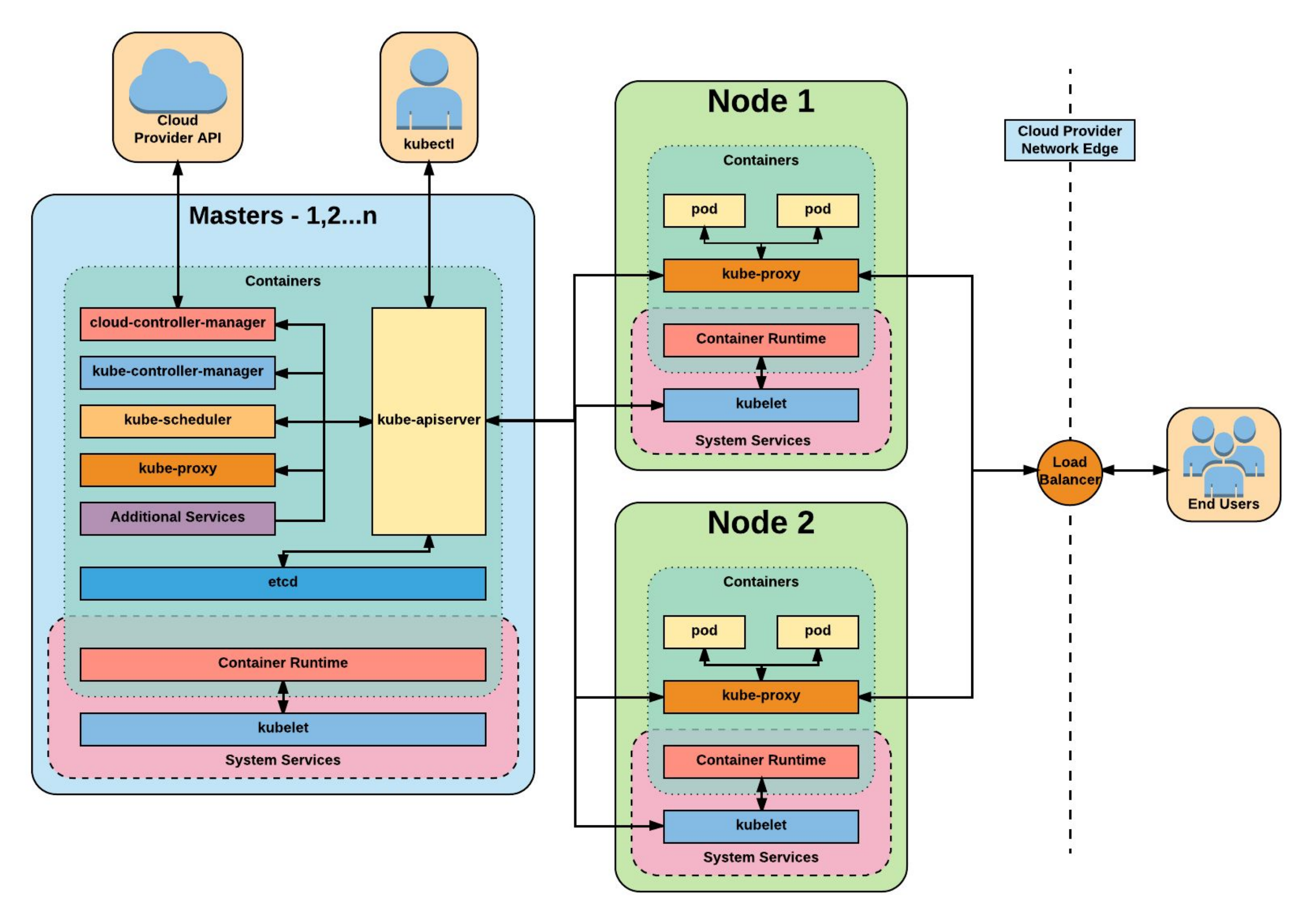
Kubernetes Architecture

Architecture Overview

Masters - Acts as the primary control plane for Kubernetes. Masters are responsible at a minimum for running the API Server, scheduler, and cluster controller. They commonly also manage storing cluster state, cloud-provider specific components and other cluster essential services.

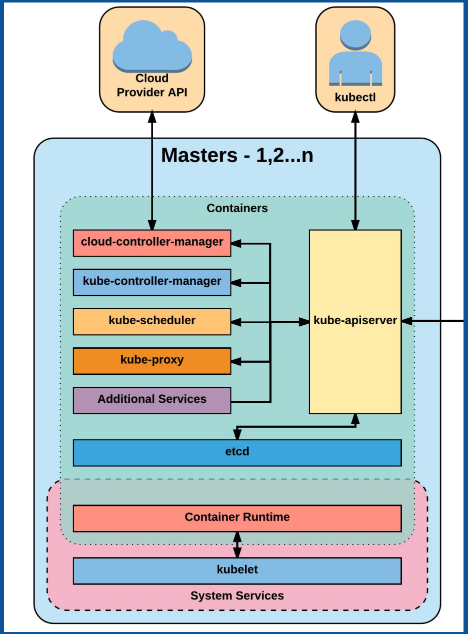
Nodes - Are the ‘workers’ of a Kubernetes cluster. They run a minimal agent that manages the node itself, and are tasked with executing workloads as designated by the master.

Architecture Overview



Master Components

Master Components



- Kube-apiserver
- Etcd
- Kube-controller-manager
- Cloud-controller-manager
- Kube-scheduler

kube-apiserver

The apiserver provides a forward facing REST interface into the kubernetes control plane and datastore. All clients, including nodes, users and other applications interact with kubernetes **strictly** through the API Server.

It is the true core of Kubernetes acting as the gatekeeper to the cluster by handling authentication and authorization, request validation, mutation, and admission control in addition to being the front-end to the backing datastore.

etcd

Etcd acts as the cluster datastore; providing a strong, consistent and highly available key-value store used for persisting cluster state.

kube-controller-manager

The controller-manager is the primary daemon that manages all core component control loops. It monitors the cluster state via the apiserver and steers the cluster towards the desired state.

List of core controllers:

https://github.com/kubernetes/kubernetes/blob/master/cmd/kube-controller-manager/app/controllermanager.go#L332

cloud-controller-manager

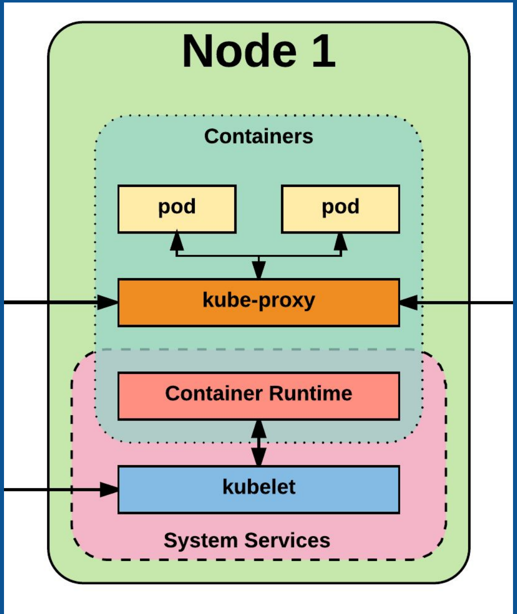
The cloud-controller-manager is a daemon that provides cloud-provider specific knowledge and integration capability into the core control loop of Kubernetes. The controllers include Node, Route, Service, and add an additional controller to handle PersistentVolumeLabels .

kube-scheduler

Kube-scheduler is a verbose policy-rich engine that evaluates workload requirements and attempts to place it on a matching resource. These requirements can include such things as general hardware reqs, affinity, anti-affinity, and other custom resource requirements.

Node Components

Node Components



- Kubelet
- Kube-proxy
- Container runtime engine

kubelet

Acts as the node agent responsible for managing pod lifecycle on its host. Kubelet understands YAML container manifests that it can read from several sources:

- File path
- HTTP Endpoint
- Etcd watch acting on any changes
- HTTP Server mode accepting container manifests over a simple API.

kube-proxy

Manages the network rules on each node and performs connection forwarding or load balancing for Kubernetes cluster services.

Available Proxy Modes:

- Userspace
- iptables
- ipvs (alpha in 1.8)

Container Runtime

With respect to Kubernetes, A container runtime is a CRI (Container Runtime Interface) compatible application that executes and manages containers.

- Containerd (docker)
- Cri-o
- Rkt
- Kata (formerly clear and hyper)
- Virtlet (VM CRI compatible runtime)

Additional Services

Kube-dns - Provides cluster wide DNS Services. Services are resolvable to *<service>.<namespace>.svc.cluster.local.*

Heapster - Metrics Collector for kubernetes cluster, used by some resources such as the Horizontal Pod Autoscaler. (required for kubedashboard metrics)

Kube-dashboard - A general purpose web based UI for kubernetes.

Networking

Networking - Fundamental Rules

1) All Pods can communicate with all other Pods without NAT
2) All nodes can communicate with all Pods (and vice-versa) without NAT.
3) The IP that a Pod sees itself as is the same IP that others see it as.

Networking - Fundamentals Applied

Containers in a pod exist within the same network namespace and share an IP; allowing for intrapod communication over *localhost.*

Pods are given a cluster unique IP for the duration of its lifecycle, but the pods themselves are fundamentally ephemeral.

Services are given a persistent cluster unique IP that spans the Pods lifecycle.

External Connectivity is generally handed by an integrated cloud provider or other external entity (load balancer)

Networking - CNI

Networking within Kubernetes is plumbed via the Container Network Interface (CNI), an interface between a container runtime and a network implementation plugin.

Compatible CNI Network Plugins:

- Calico
- Cillium
- Contiv
- Contrail
- Flannel
- GCE
- kube-router
- Multus
- OpenVSwitch
- OVN
- Romana
- Weave

Kubernetes Concepts

Kubernetes Concepts - Core

Cluster - A collection of hosts that aggregate their available resources including cpu, ram, disk, and their devices into a usable pool.

Master - The master(s) represent a collection of components that make up the control plane of Kubernetes. These components are responsible for all cluster decisions including both scheduling and responding to cluster events.

Node - A single host, physical or virtual capable of running pods. A node is managed by the master(s), and at a minimum runs both kubelet and kube-proxy to be considered part of the cluster.

Namespace - A logical cluster or environment. Primary method of dividing a cluster or scoping access.

Concepts - Core (cont.)

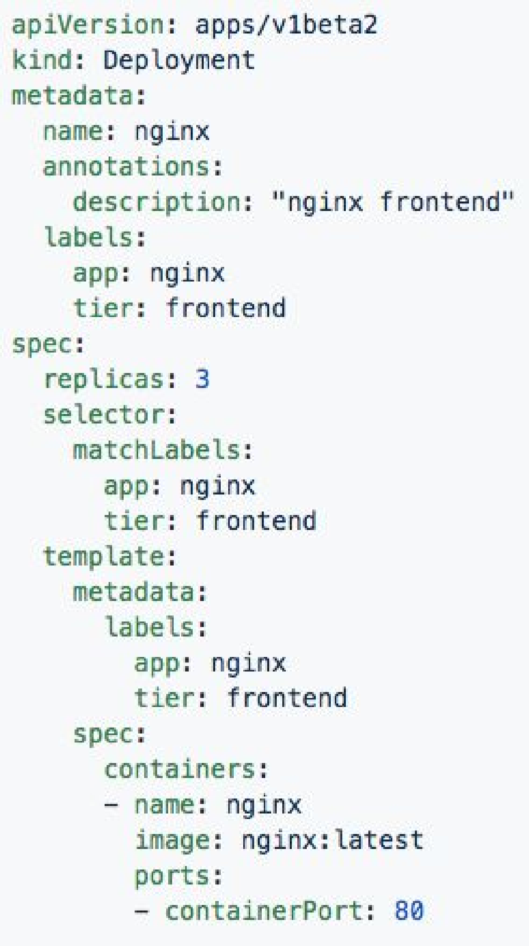
Label - Key-value pairs that are used to **identify**, describe and group together related sets of objects. Labels have a strict syntax and available character set. *

Annotation - Key-value pairs that contain **non-identifying** information or metadata. Annotations do not have the the syntax limitations as labels and can contain structured or unstructured data.

Selector - Selectors use labels to filter or select objects. Both equality-based (=, ==, !=) or simple key-value matching selectors are supported.

* https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#syntax-and-character-set

Labels, and Annotations, and Selectors



```yaml
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx
  annotations:
    description: "nginx frontend"
  labels:
    app: nginx
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
      tier: frontend
  template:
    metadata:
      labels:
        app: nginx
        tier: frontend
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Labels:

app: nginx

tier: frontned

Annotations

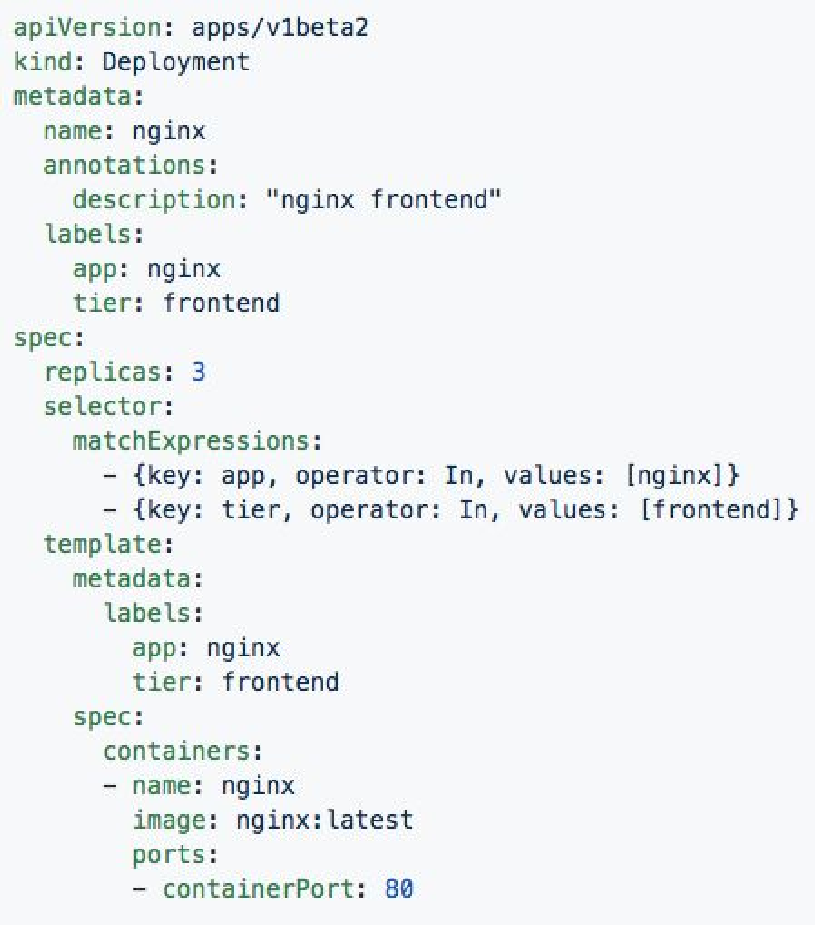
description: “nginx frontend”

Selector:

app: nginx

tier: frontend

Set-based selectors



```yaml
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx
  annotations:
    description: "nginx frontend"
  labels:
    app: nginx
    tier: frontend
spec:
  replicas: 3
  selector:
    matchExpressions:
      - {key: app, operator: In, values: [nginx]}
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: nginx
        tier: frontend
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Valid Operators:

- In
- NotIn
- Exists
- DoesNotExist

Supported Objects with set-based selectors:

- Job
- Deployment
- ReplicaSet
- DaemonSet
- PersistentVolumeClaims

Concepts - Workloads

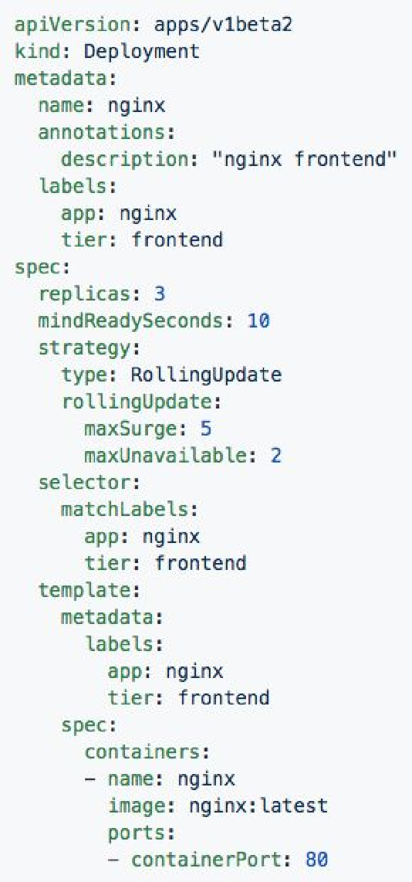
Pod - A pod is the smallest unit of work or management resource within Kubernetes. It is comprised of one or more containers that share their storage, network, and context (namespace, cgroups etc).

ReplicationController - Method of managing pod replicas and their lifecycle. Their scheduling, scaling, and deletion.

ReplicaSet - Next Generation ReplicationController. Supports set-based selectors.

Deployment - A declarative method of managing stateless Pods and ReplicaSets. Provides rollback functionality in addition to more granular update control mechanisms.

Deployment

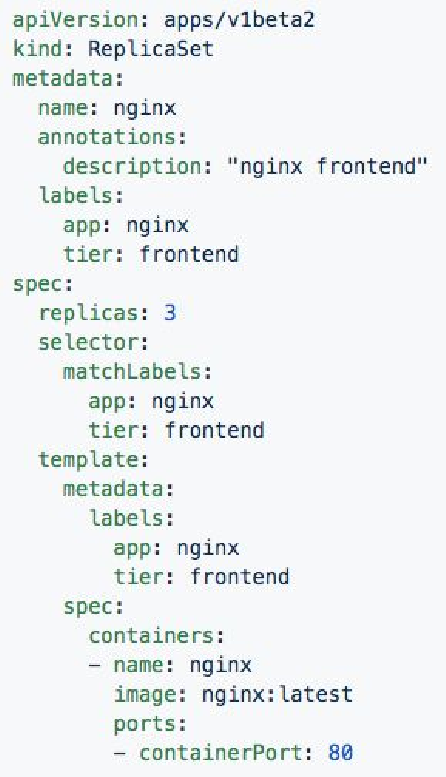


```yaml
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx
  annotations:
    description: "nginx frontend"
  labels:
    app: nginx
    tier: frontend
spec:
  replicas: 3
  mindReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 5
      maxUnavailable: 2
  selector:
    matchLabels:
      app: nginx
      tier: frontend
  template:
    metadata:
      labels:
        app: nginx
        tier: frontend
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Contains configuration of how updates or ‘deployments’ should be managed in addition to the pod template used to generate the ReplicaSet.

ReplicaSet

Generated ReplicaSet from Deployment spec.



```yaml
apiVersion: apps/v1beta2
kind: ReplicaSet
metadata:
  name: nginx
  annotations:
    description: "nginx frontend"
  labels:
    app: nginx
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
      tier: frontend
  template:
    metadata:
      labels:
        app: nginx
        tier: frontend
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

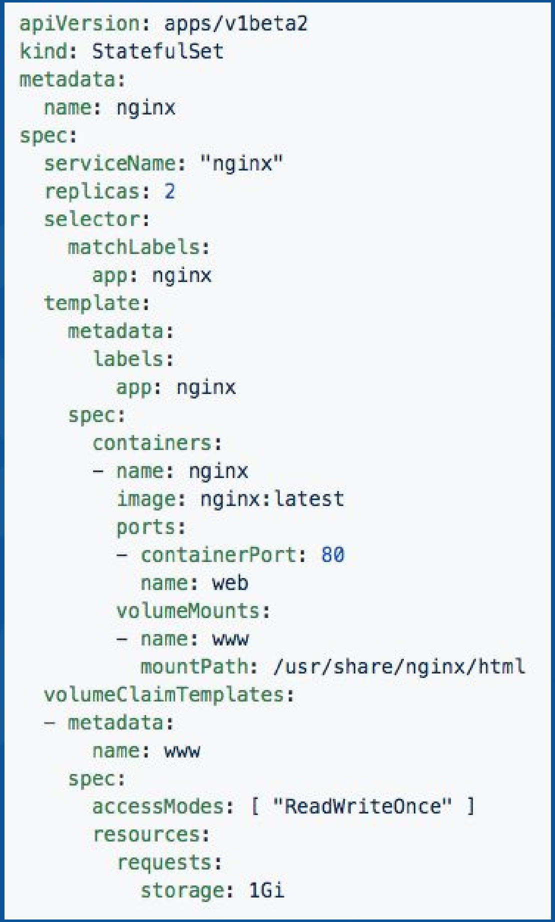
Concepts - Workloads (cont.)

StatefulSet - A controller tailored to managing Pods that must persist or maintain state. Pod identity including hostname, network, and storage will be persisted.

DaemonSet - Ensures that all nodes matching certain criteria will run an instance of a supplied Pod. Ideal for cluster wide services such as log forwarding, or health monitoring.

StatefulSet

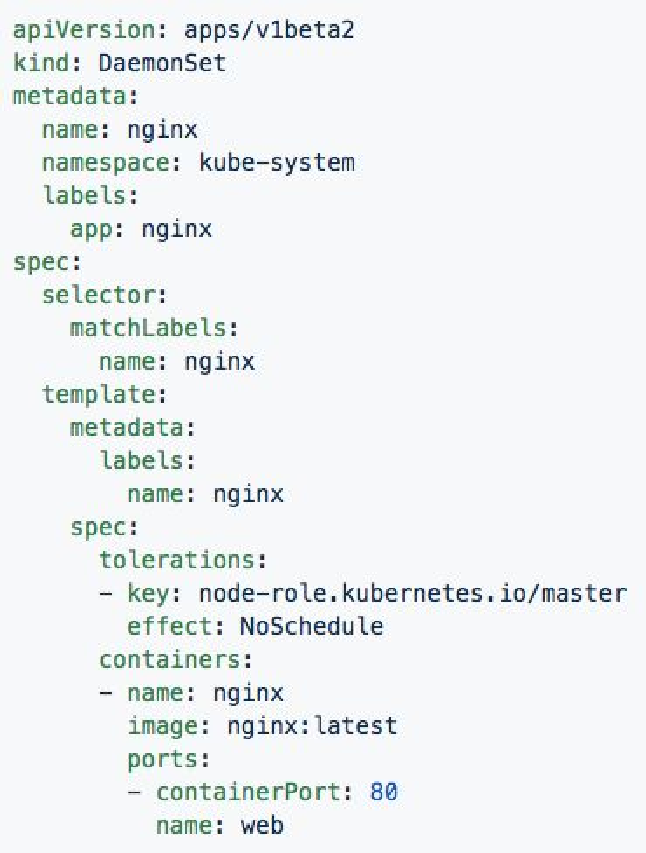
- Attaches to ‘headeless service’ (not shown) *nginx*.
- Pods given unique ordinal names using the pattern *<statefulset name>-<ordinal index>*.
- Creates independent persistent volumes based on the ‘volumeClaimTemplates’.



```yaml
apiVersion: apps/v1beta2
kind: StatefulSet
metadata:
  name: nginx
spec:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

DaemonSet

- Bypasses default scheduler
- Schedules a single instance on every host while adhering to tolerances and taints.



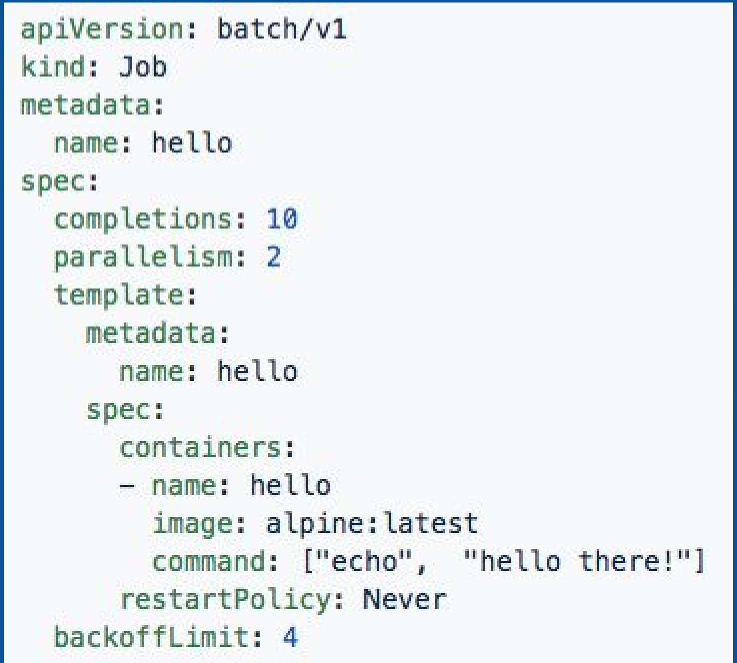
```yaml
apiVersion: apps/v1beta2
kind: DaemonSet
metadata:
  name: nginx
  namespace: kube-system
  labels:
    app: nginx
spec:
  selector:
    matchLabels:
      name: nginx
  template:
    metadata:
      labels:
        name: nginx
    spec:
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
          name: web
```

Concepts - Workloads (cont.)

Job - The job controller ensures one or more pods are executed and successfully terminates. It will do this until it satisfies the completion and/or parallelism condition.

CronJob - An extension of the Job Controller, it provides a method of executing jobs on a cron-like schedule.

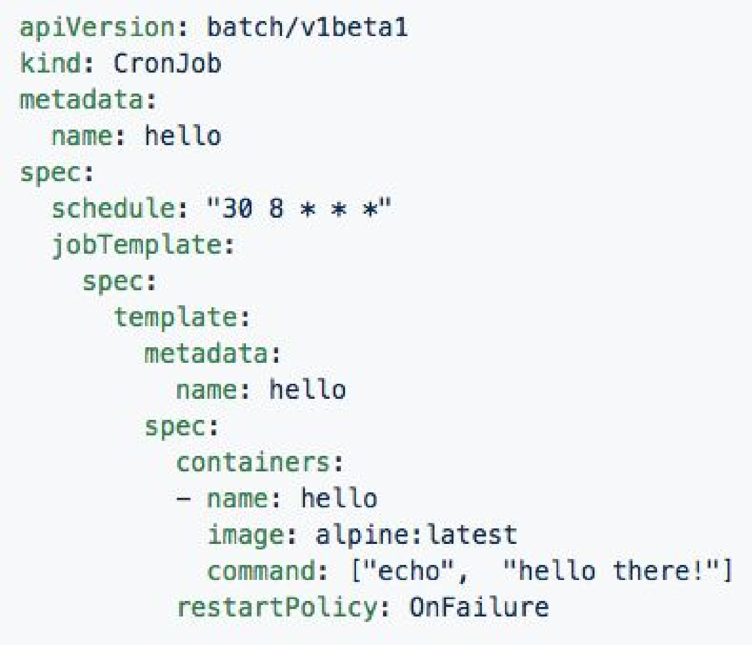
Jobs



```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: hello
spec:
  completions: 10
  parallelism: 2
  template:
    metadata:
      name: hello
    spec:
      containers:
      - name: hello
        image: alpine:latest
        command: ["echo",  "hello there!"]
      restartPolicy: Never
  backoffLimit: 4
```

- Number of pod executions can be controlled via *spec.completions*
- Jobs can be parallelized using *spec.parallelism*
- Jobs and Pods are **NOT** automatically cleaned up after a job has completed.

CronJob



```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "30 8 * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          name: hello
        spec:
          containers:
          - name: hello
            image: alpine:latest
            command: ["echo",  "hello there!"]
          restartPolicy: OnFailure
```

- Adds cron schedule to job template

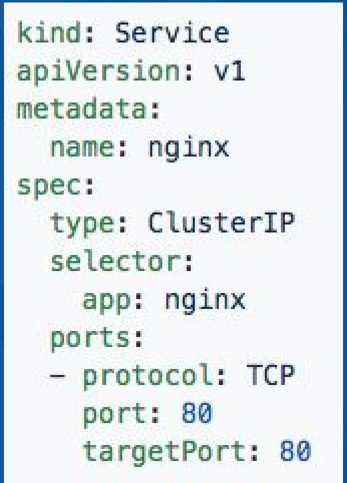
Concepts - Network

Service - Services provide a method of exposing and consuming L4 Pod network accessible resources. They use label selectors to map groups of pods and ports to a cluster-unique virtual IP.

Ingress - An ingress controller is the primary method of exposing a cluster service (usually http) to the outside world. These are load balancers or routers that usually offer SSL termination, name-based virtual hosting etc.

Service

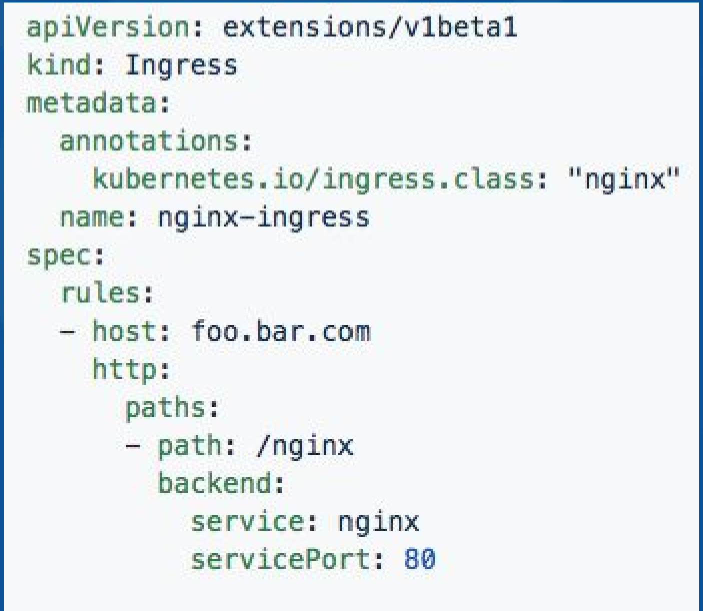
- Acts as the unified method of accessing replicated pods.
- Four major Service Types:
  - CluterIP - Exposes service on a strictly cluster-internal IP (default)
  - NodePort - Service is exposed on each node’s IP on a statically defined port.
  - LoadBalancer - Works in combination with a cloud provider to expose a service outside the cluster on a static external IP.
  - ExternalName - used to references endpoints **OUTSIDE** the cluster by providing a static internally referenced DNS name.



```yaml
kind: Service
apiVersion: v1
metadata:
  name: nginx
spec:
  type: ClusterIP
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

Ingress Controller

- Deployed as a pod to one or more hosts
- Ingress controllers are an external controller with multiple options.
  - Nginx
  - HAproxy
  - Contour
  - Traefik
- Specific features and controller specific configuration is passed through annotations.



```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: "nginx"
  name: nginx-ingress
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /nginx
        backend:
          service: nginx
          servicePort: 80
```

Concepts - Storage

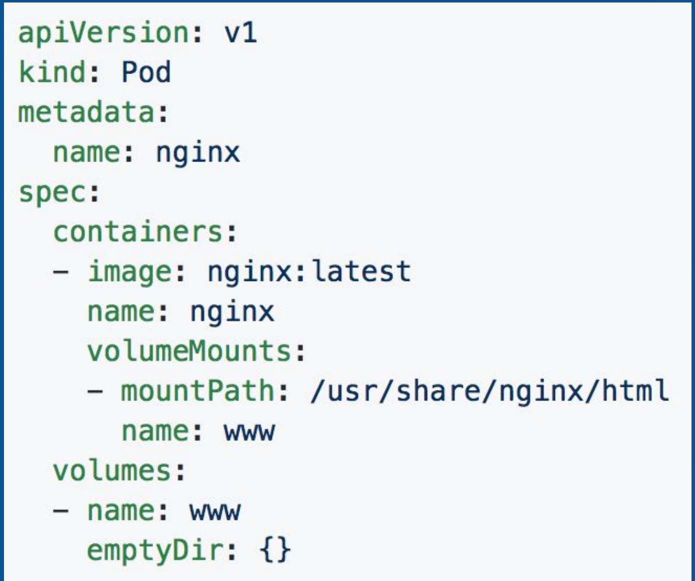
Volume - Storage that is **tied to the Pod Lifecycle**, consumable by one or more containers within the pod.

PersistentVolume - A PersistentVolume (PV) represents a storage resource. PVs are commonly linked to a backing storage resource, NFS, GCEPersistentDisk, RBD etc. and are provisioned ahead of time. Their lifecycle is handled independently from a pod.

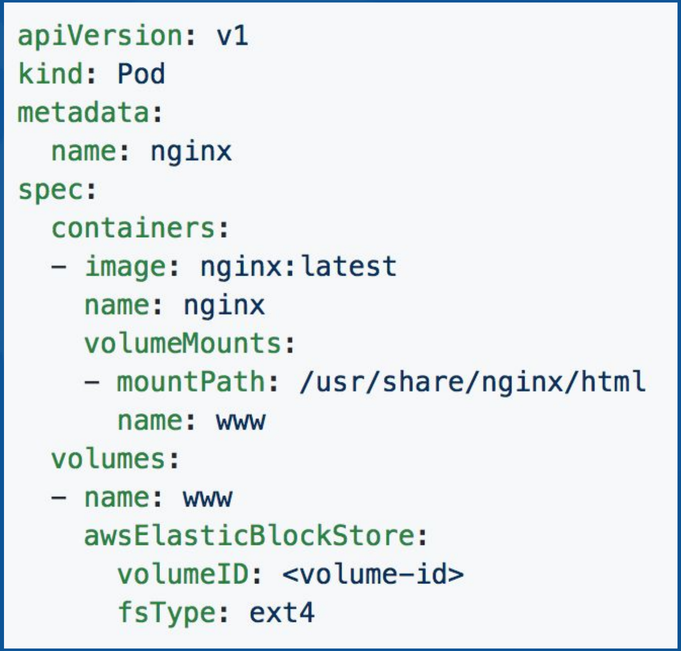
PersistentVolumeClaim - A PersistentVolumeClaim (PVC) is a request for storage that satisfies a set of requirements instead of mapping to a storage resource directly. Commonly used with dynamically provisioned storage.

StorageClass - Storage classes are an abstraction on top of an external storage resource. These will include a provisioner, provisioner configuration parameters as well as a PV reclaimPolicy.

Volumes

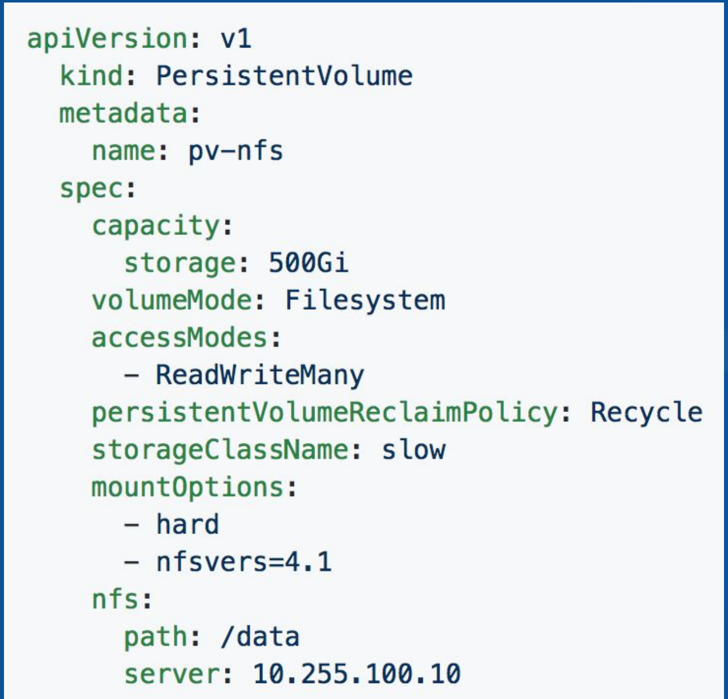


```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:latest
    name: nginx
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: www
  volumes:
  - name: www
    emptyDir: {}
```



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:latest
    name: nginx
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: www
  volumes:
  - name: www
    awsElasticBlockStore:
      volumeID: <volume-id>
      fsType: ext4
```

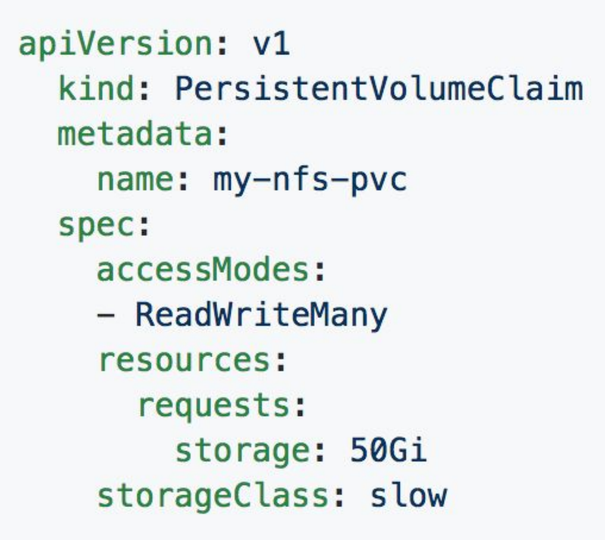
Persistent Volumes



```yaml
apiVersion: v1
  kind: PersistentVolume
  metadata:
    name: pv-nfs
  spec:
    capacity:
      storage: 500Gi
    volumeMode: Filesystem
    accessModes:
      - ReadWriteMany
    persistentVolumeReclaimPolicy: Recycle
    storageClassName: slow
    mountOptions:
      - hard
      - nfsvers=4.1
    nfs:
      path: /data
      server: 10.255.100.10
```

- PVs are a cluster-wide resource
- Not directly consumable by a Pod
- PV Parameters:
  - Capacity
  - accessModes
    - ReadOnlyMany (ROX)
    - ReadWriteOnce (RWO)
    - ReadWriteMany (RWX)
  - persistentVolumeReclaimPolicy
    - Retain
    - Recycle
    - Delete
  - StorageClass

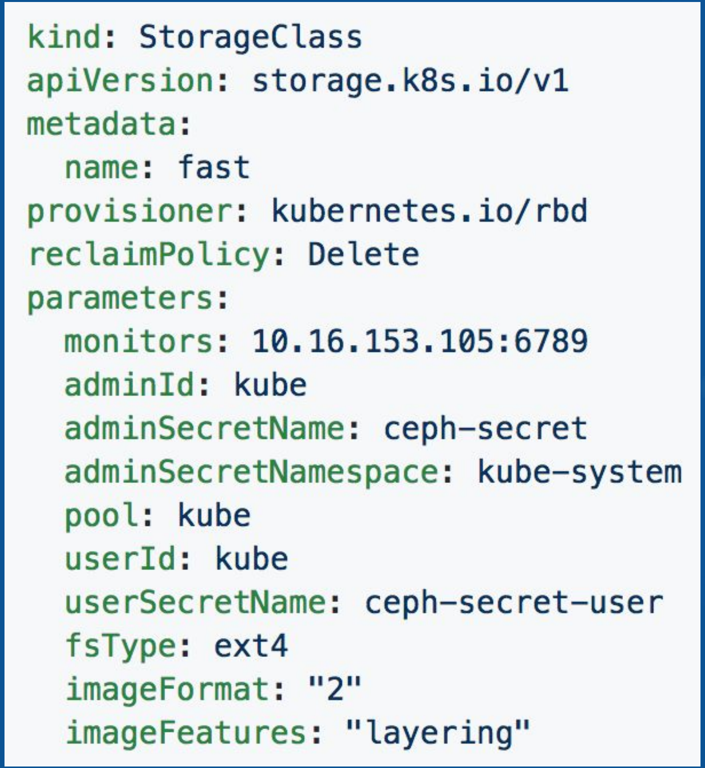
Persistent Volume Claims



```yaml
apiVersion: v1
 kind: PersistentVolumeClaim
 metadata:
  name: my-nfs-pvc
 spec:
  accessModes:
  - ReadWriteMany
  resources:
   requests:
    storage: 50Gi
  storageClass: slow
```

- PVCs are scoped to namespaces
- Supports accessModes like PVs
- Uses resource request model similar to Pods
- Claims will consume storage from matching PVs or StorageClasses based on *storageClass* and selectors.

Storage Classes



```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast
provisioner: kubernetes.io/rbd
reclaimPolicy: Delete
parameters:
  monitors: 10.16.153.105:6789
  adminId: kube
  adminSecretName: ceph-secret
  adminSecretNamespace: kube-system
  pool: kube
  userId: kube
  userSecretName: ceph-secret-user
  fsType: ext4
  imageFormat: "2"
  imageFeatures: "layering"
```

- Uses an external system defined by the provisioner to dynamically consume and allocate storage.
- Storage Class Fields
  - Provisioner
  - Parameters
  - reclaimPolicy

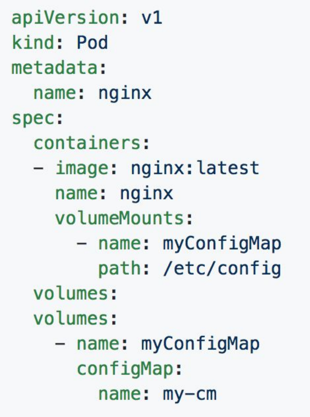
Concepts - Configuration

ConfigMap - Externalized data stored within kubernetes that can be referenced as a commandline argument, environment variable, or injected as a file into a volume mount. Ideal for separating containerized application from configuration.

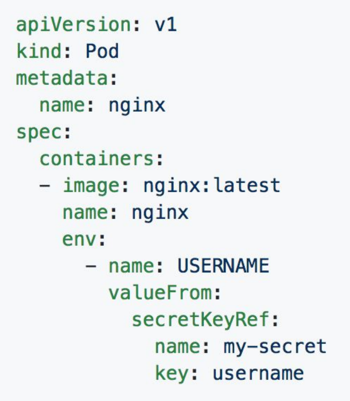
Secret - Functionally identical to ConfigMaps, but stored encoded as base64, and encrypted at rest (if configured).

ConfigMaps and Secrets

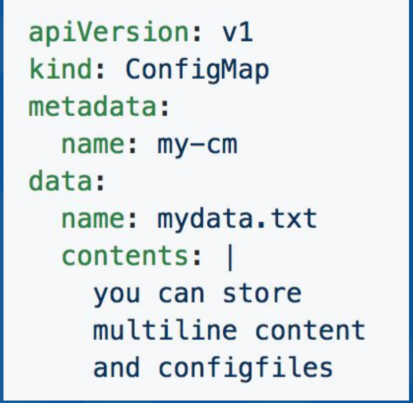
- Can be used in Pod Config:
  - Injected as a file
  - Passed as an environment variable
  - Used as a container command (requires passing as env var)



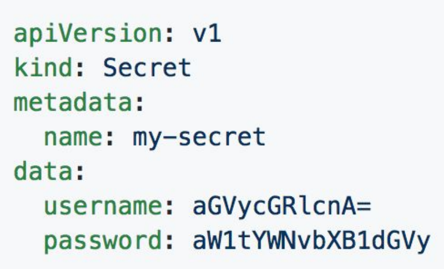
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:latest
    name: nginx
    volumeMounts:
      - name: myConfigMap
        path: /etc/config
  volumes:
  volumes:
    - name: myConfigMap
      configMap:
        name: my-cm
```



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:latest
    name: nginx
    env:
      - name: USERNAME
        valueFrom:
          secretKeyRef:
            name: my-secret
            key: username
```



```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-cm
data:
  name: mydata.txt
  contents: |
    you can store
    multiline content
    and configfiles
```



```yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
data:
  username: aGVycGRlcnA=
  password: aW1tYWNvbXB1dGVy
```

Concepts - Auth and Identity (RBAC)

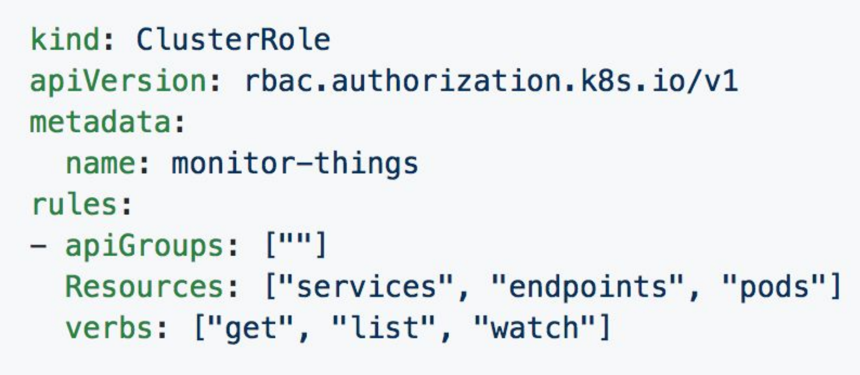
[Cluster]Role - Roles contain rules that act as a set of permissions that apply verbs like "get", "list", "watch" etc over resources that are scoped to apiGroups. Roles are scoped to namespaces, and ClusterRoles are applied cluster-wide.

[Cluster]RoleBinding - Grant the permissions as defined in a [Cluster]Role to one or more "subjects" which can be a user, group, or service account.

ServiceAccount- ServiceAccounts provide a consumable identity for pods or external services that interact with the cluster directly and are scoped to namespaces.

[Cluster]Role

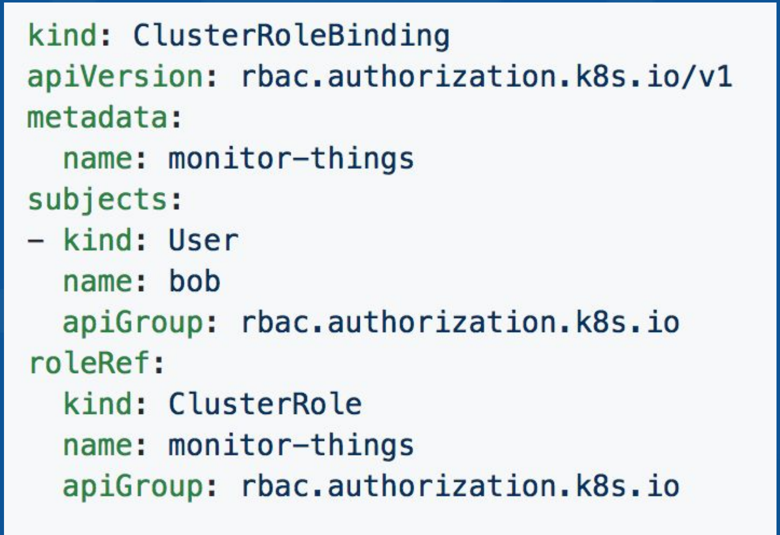
- Permissions translate to url path. With “” defaulting to core group.
- Resources act as items the role should be granted access to.
- Verbs are the actions the role can perform on the referenced resources.



```yaml
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: monitor-things
rules:
- apiGroups: [""]
  Resources: ["services", "endpoints", "pods"]
  verbs: ["get", "list", "watch"]
```

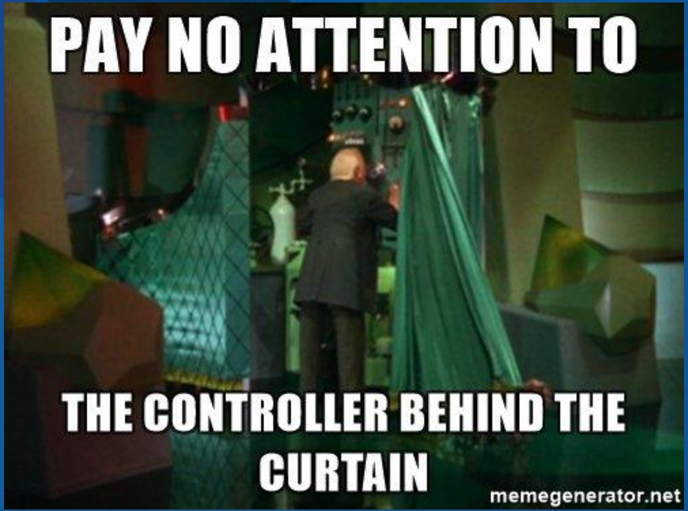
[Cluster]RoleBinding

- Can reference multiple subjects
- Subjects can be of kind:
  - User
  - Group
  - ServiceAccount
- roleRef targets a single role only.



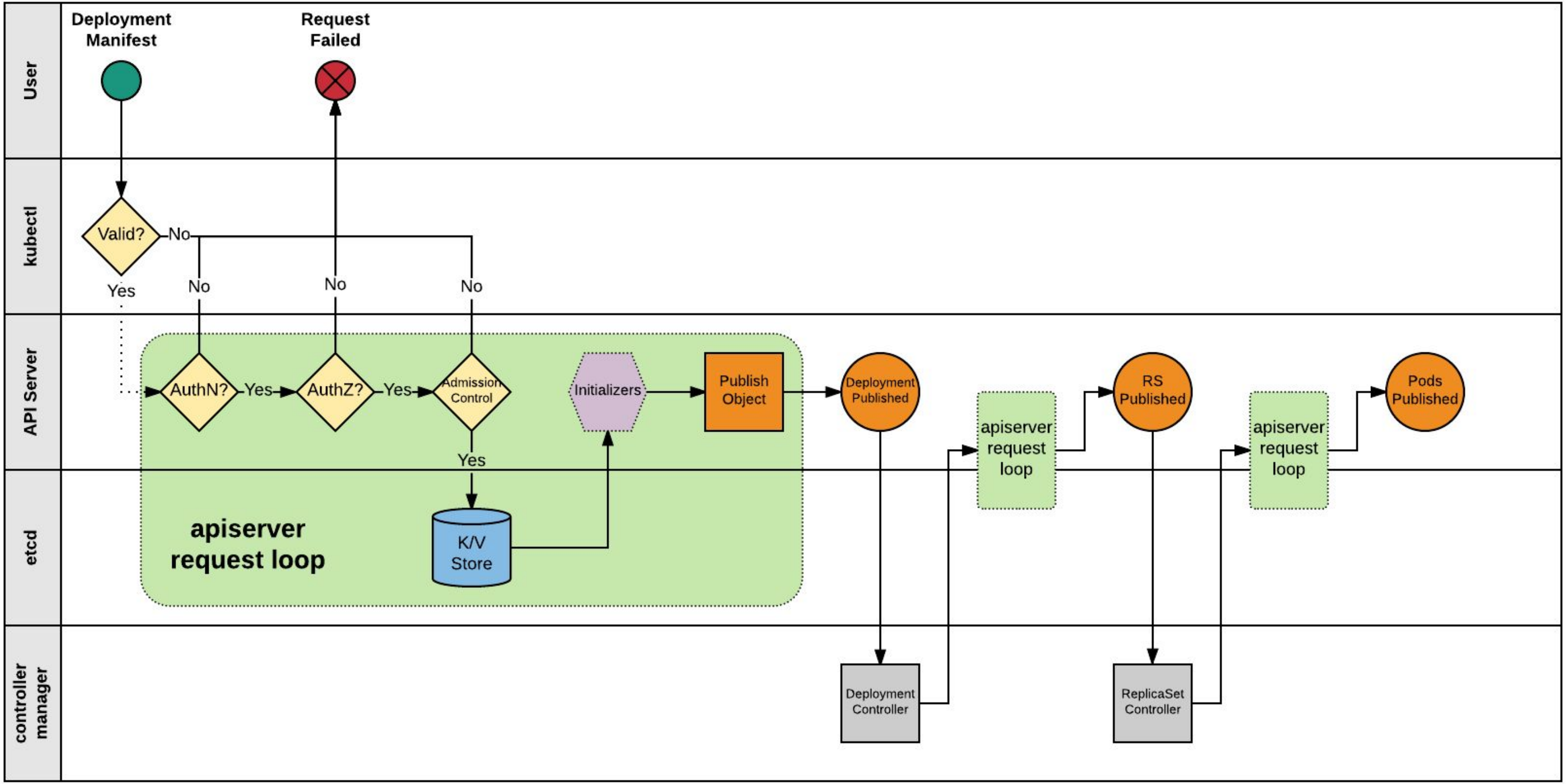
```yaml
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: monitor-things
subjects:
- kind: User
  name: bob
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: monitor-things
  apiGroup: rbac.authorization.k8s.io
```

Behind The Scenes



Behind The Scenes

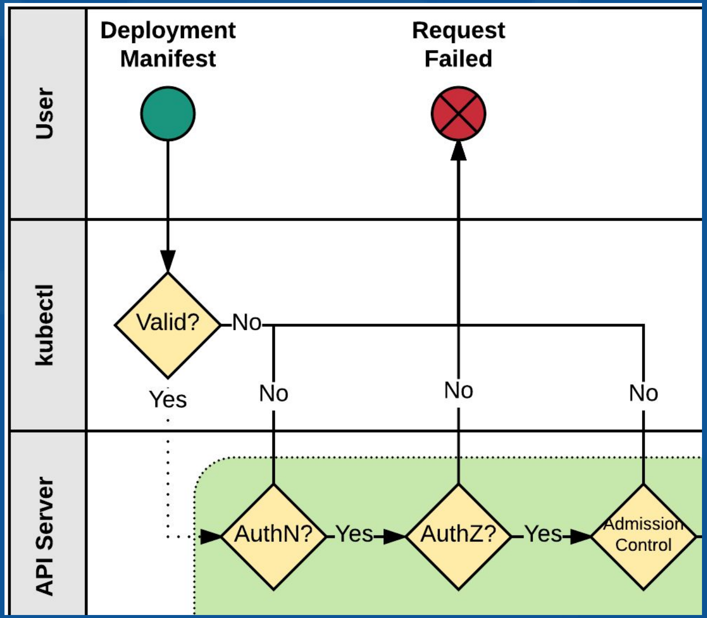
Deployment From Beginning to End


User
kubectl
API Server
etcd
controller manager
Deployment Manifest
Request Failed
Valid?
No
Yes
No
No
No
AuthN?
Yes
AuthZ?
Yes
Admission Control
Yes
K/V Store
Initializers
Publish Object
apiserver request loop
Deployment Published
Deployment Controller
apiserver request loop
RS Published
ReplicaSet Controller
apiserver request loop
Pods Published

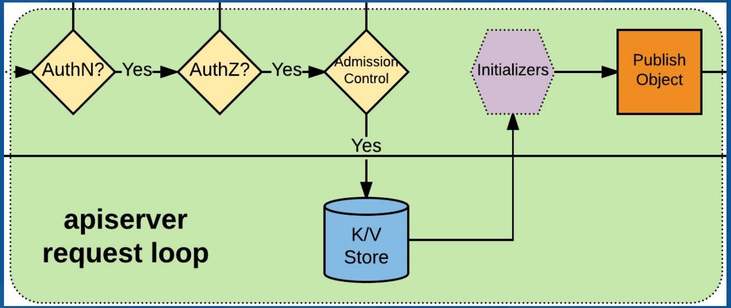
Kubectl

1) Kubectl performs client side validation on manifest (linting).

2) Manifest is prepared and serialized creating a JSON payload.



APIserver Request Loop



3) Kubectl authenticates to apiserver via x509, jwt, http auth proxy, other plugins, or http-basic auth.

4) Authorization iterates over available AuthZ sources: Node, ABAC, RBAC, or webhook.

5) AdmissionControl checks resource quotas, other security related checks etc.

6) Request is stored in etcd.

7) Initializers are given opportunity to mutate request before the object is published.

8) Request is published on apiserver.

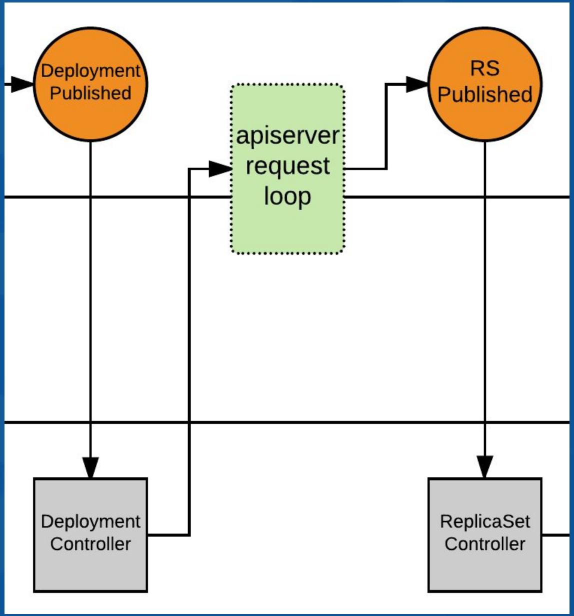
Deployment Controller

9) Deployment Controller is notified of the new Deployment via callback.

10) Deployment Controller evaluates cluster state and reconciles the desired vs current state and forms a request for the new ReplicaSet.

11) apiserver request loop evaluates Deployment Controller request.

12) ReplicaSet is published.



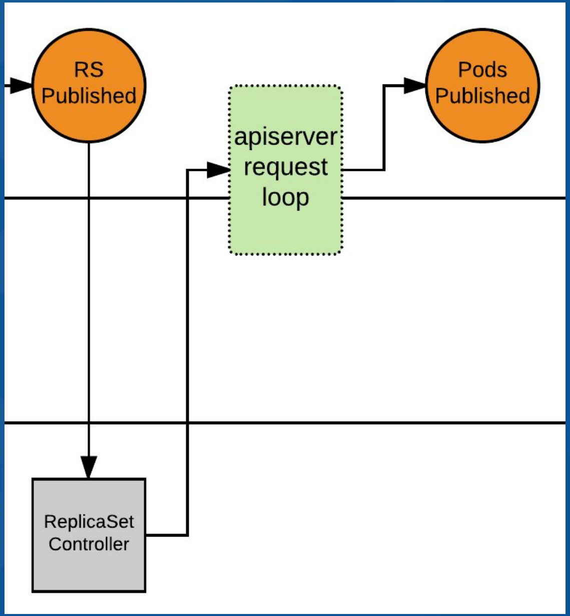
ReplicaSet Controller

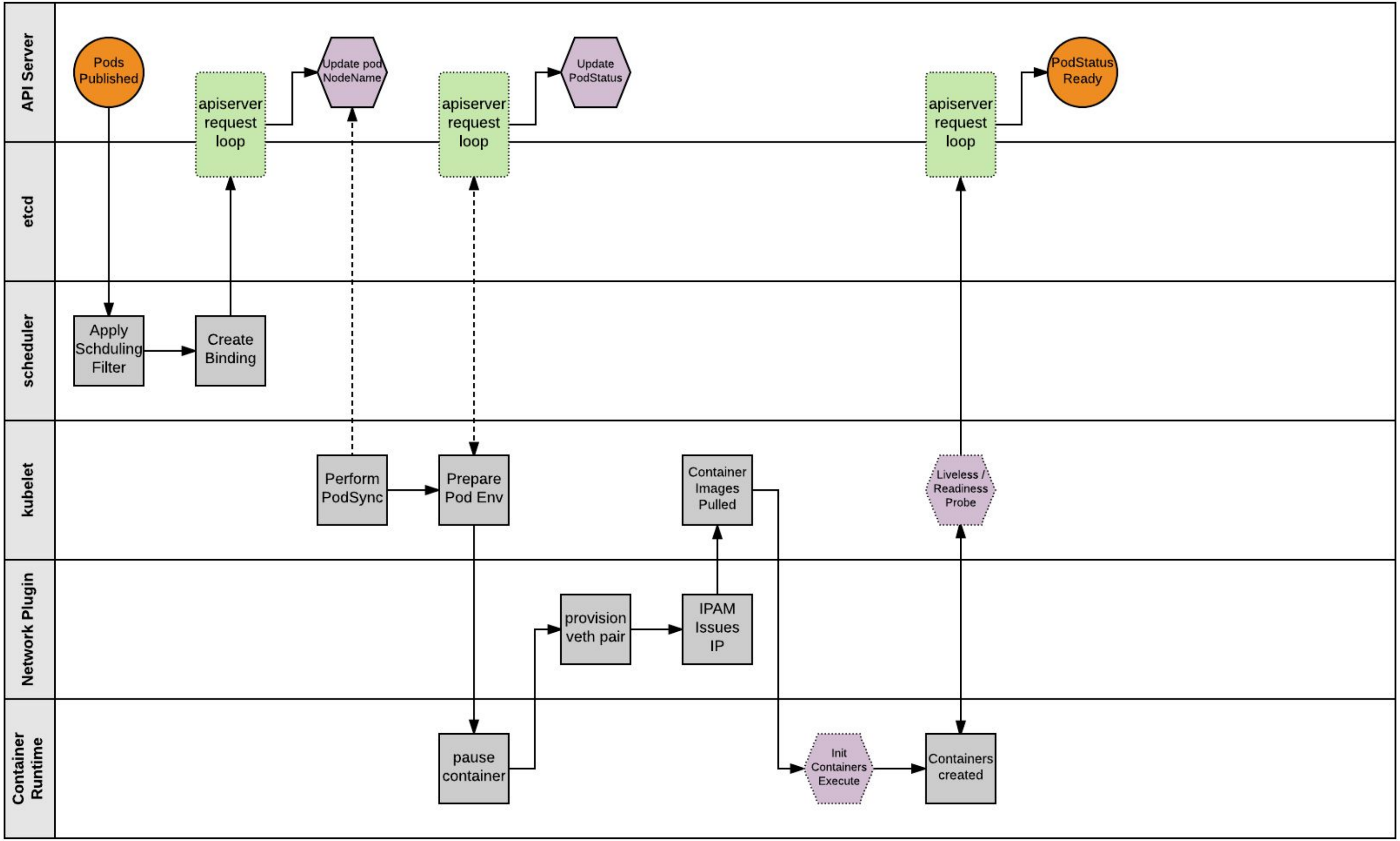
13) ReplicaSet Controller is notified of the new ReplicaSet via callback.

14) ReplicaSet Controller evaluates cluster state and reconciles the desired vs current state and forms a request for the desired amount of pods.

15) apiserver request loop evaluates ReplicaSet Controller request.

16) Pods published, and enter 'Pending' phase.





API Server

etcd

scheduler

kubelet

Network Plugin

Container Runtime

Pods Published

Apply Schduling Filter

Create Binding

apiserver request loop

Update pod NodeName

Perform PodSync

Prepare Pod Env

apiserver request loop

Update PodStatus

pause container

provision veth pair

IPAM Issues IP

Container Images Pulled

Init Containers Execute

Containers created

Liveless / Readiness Probe

apiserver request loop

PodStatus Ready

Scheduler

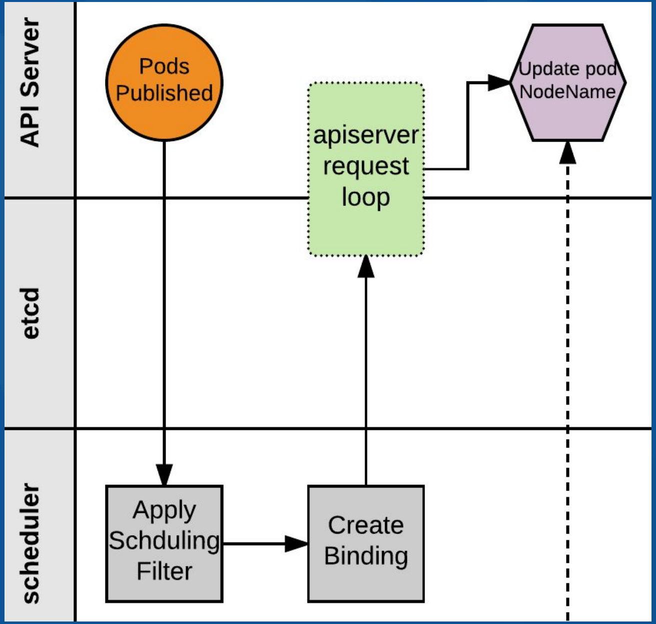
17) Scheduler monitors published pods with no 'NodeName' assigned.

18) Applies scheduling rules and filters to find a suitable node to host the Pod.

19) Scheduler creates a binding of Pod to Node and POSTs to apiserver.

20) apiserver request loop evaluates POST request.

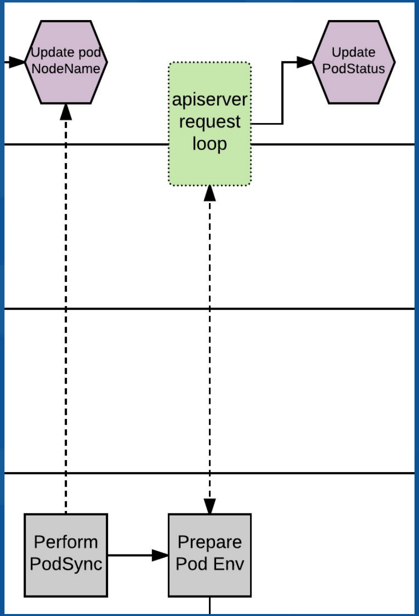
21) Pod status is updated with node binding and sets status to 'PodScheduled'.



Kubelet - PodSync

22) The kubelet daemon on every node polls the apiserver filtering for pods matching its own ‘NodeName’; checking its current state with the desired state published through the apiserver.

23) Kubelet will then move through a series of internal processes to prepare the pod environment. This includes pulling secrets, provisioning storage, applying AppArmor profiles and other various scaffolding. During this period, it will asynchronously be POST’ing the ‘PodStatus’ to the apiserver through the standard apiserver request loop.

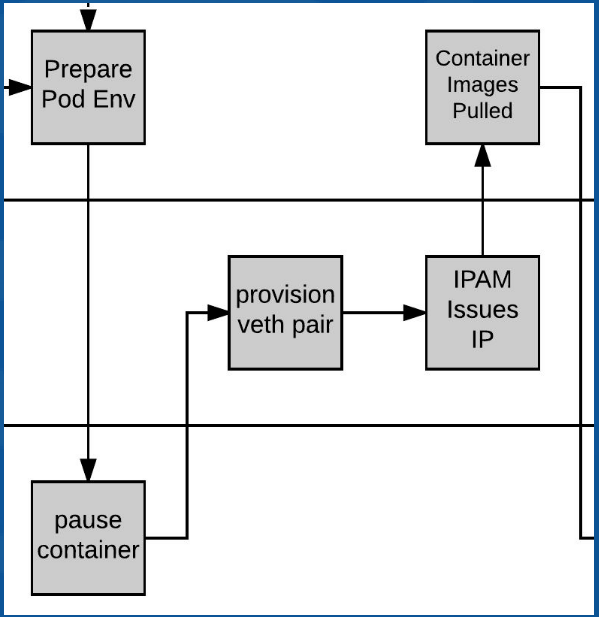


Pause and Plumbing

24) Kubelet then provisions a ‘pause’ container via the CRI (Container Runtime Interface). The pause container acts as the parent container for the Pod.

25) The network is plumbed to the Pod via the CNI (Container Network Interface), creating a veth pair attached to the pause container and to a container bridge (cbr0).

26) IPAM handled by the CNI plugin assigns an IP to the pause container.

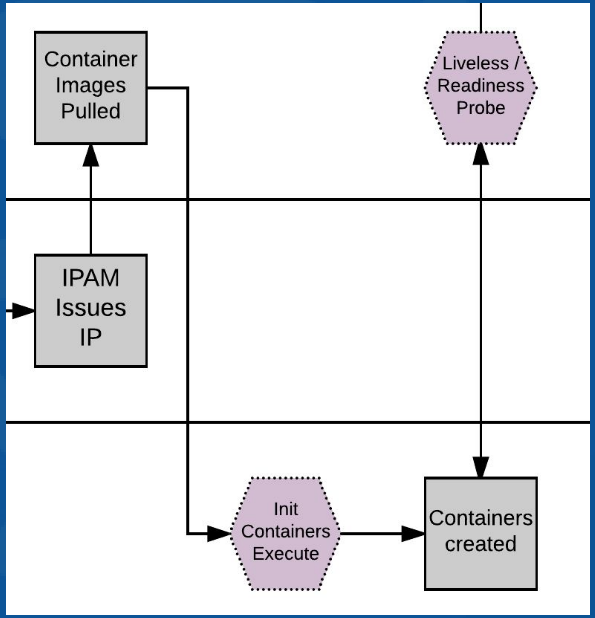


Kublet - Create Containers

24) Kubelet pulls the container Images.

25) Kubelet first creates and starts any init containers.

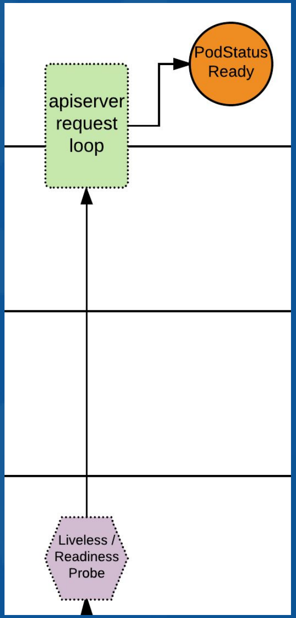
26) Once the optional init containers complete, the primary pod containers are started.



Pod Status

27) If there are any liveless/readiness probes, these are executed before the PodStatus is updated.

28) If all complete successfully, PodStatus is set to ready and the container has started successfully.



The Pod is Deployed!

Questions?