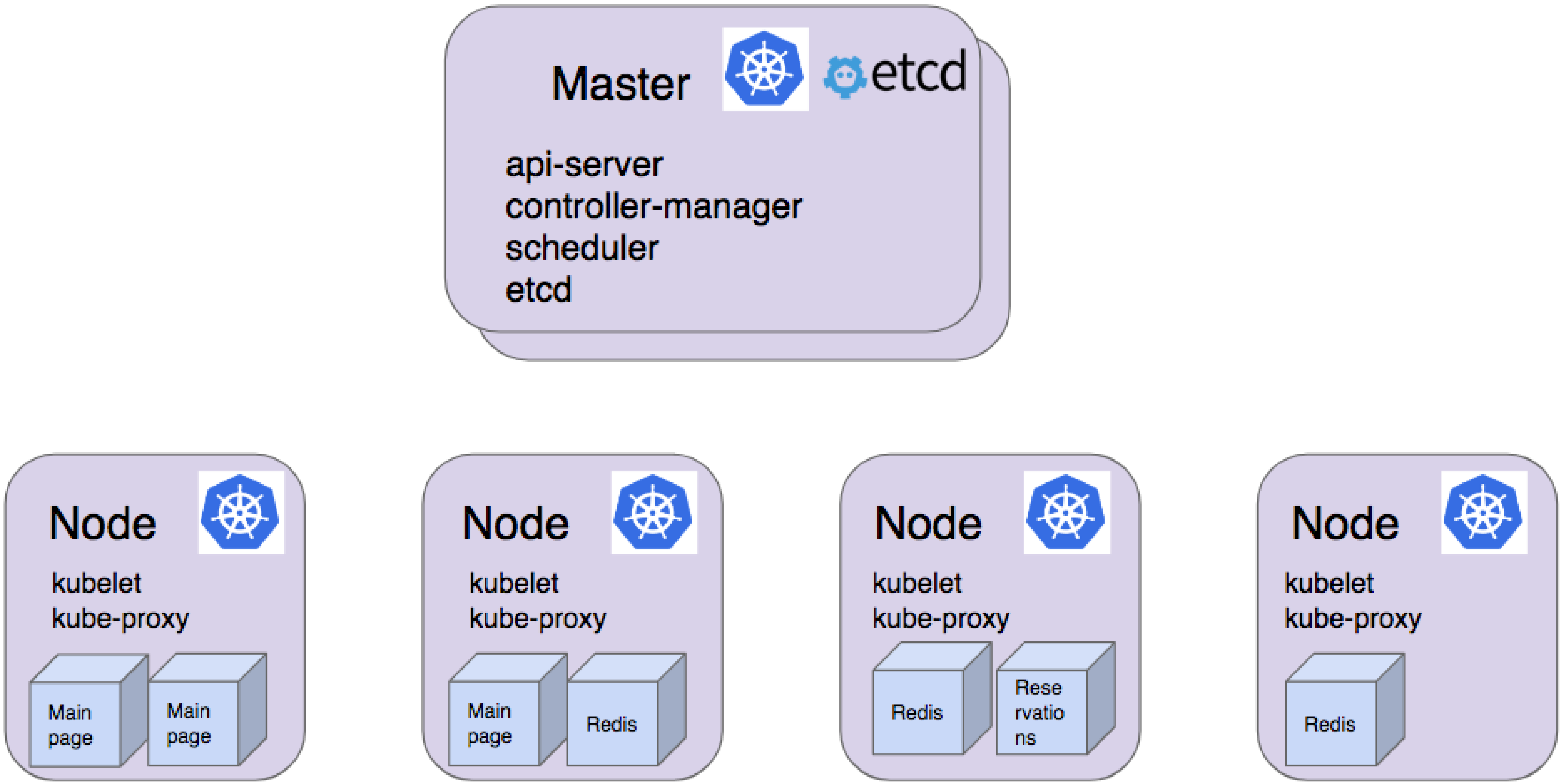
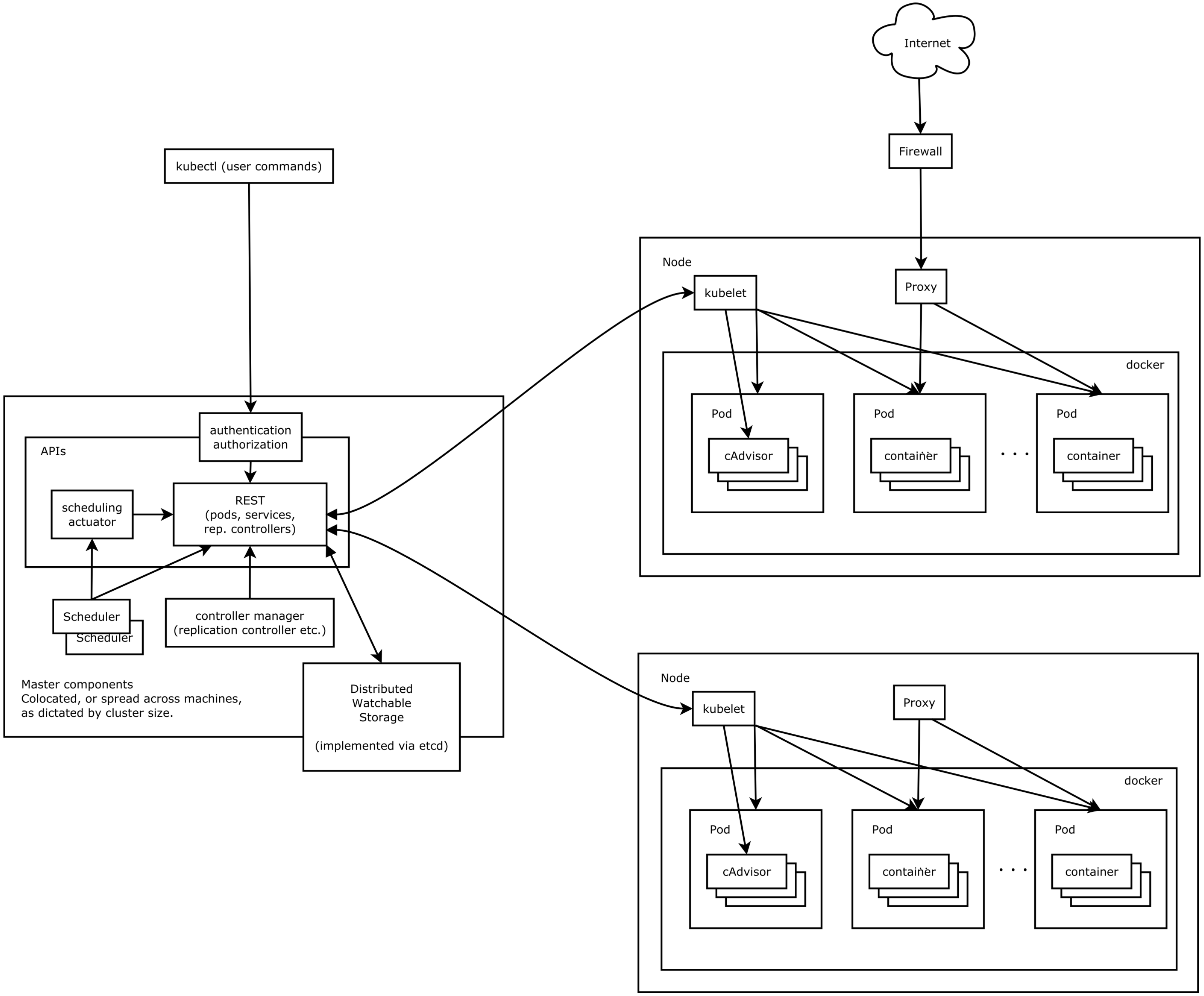
# Kubernetes Clustering

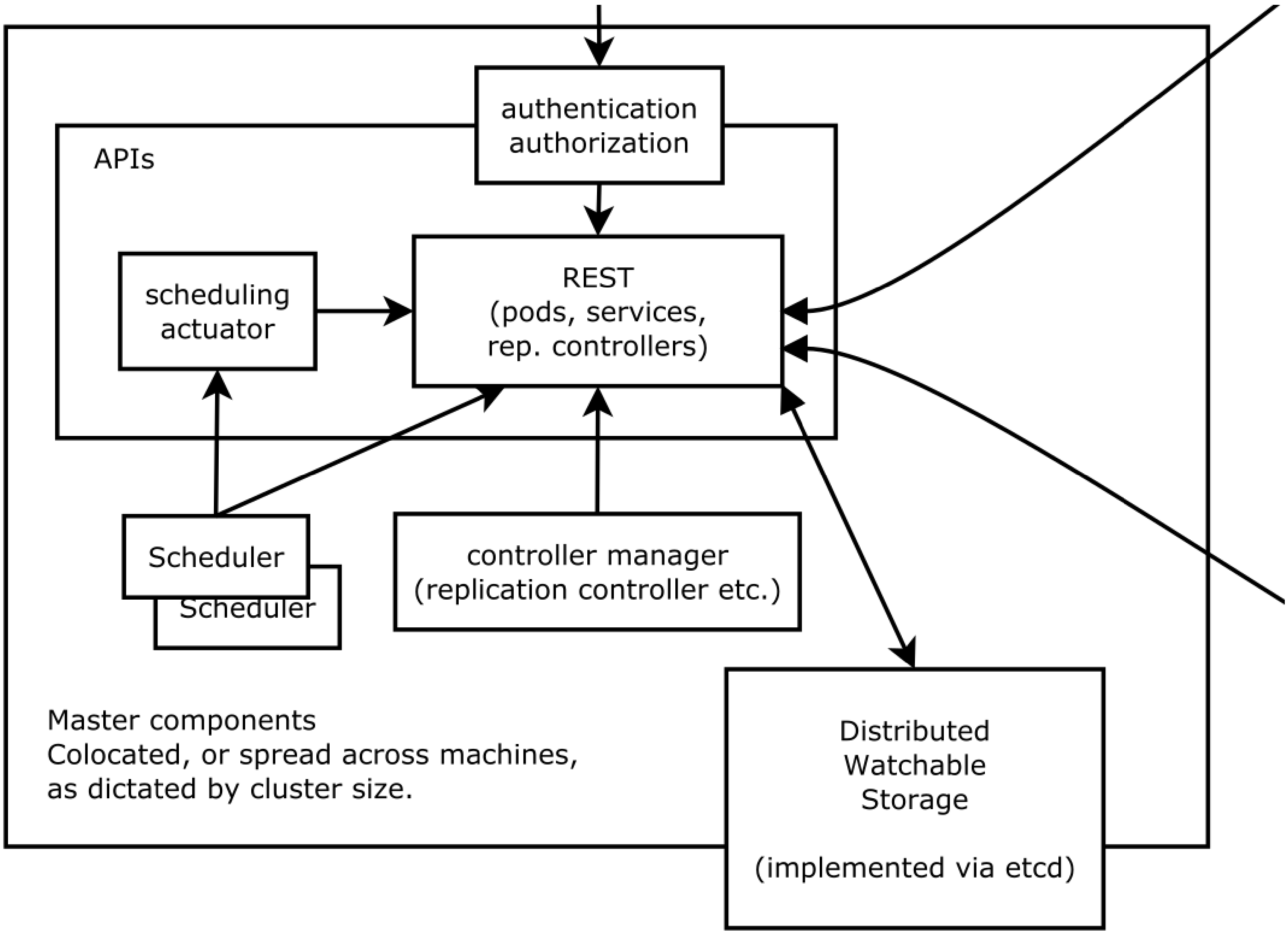
# In this section we will cover

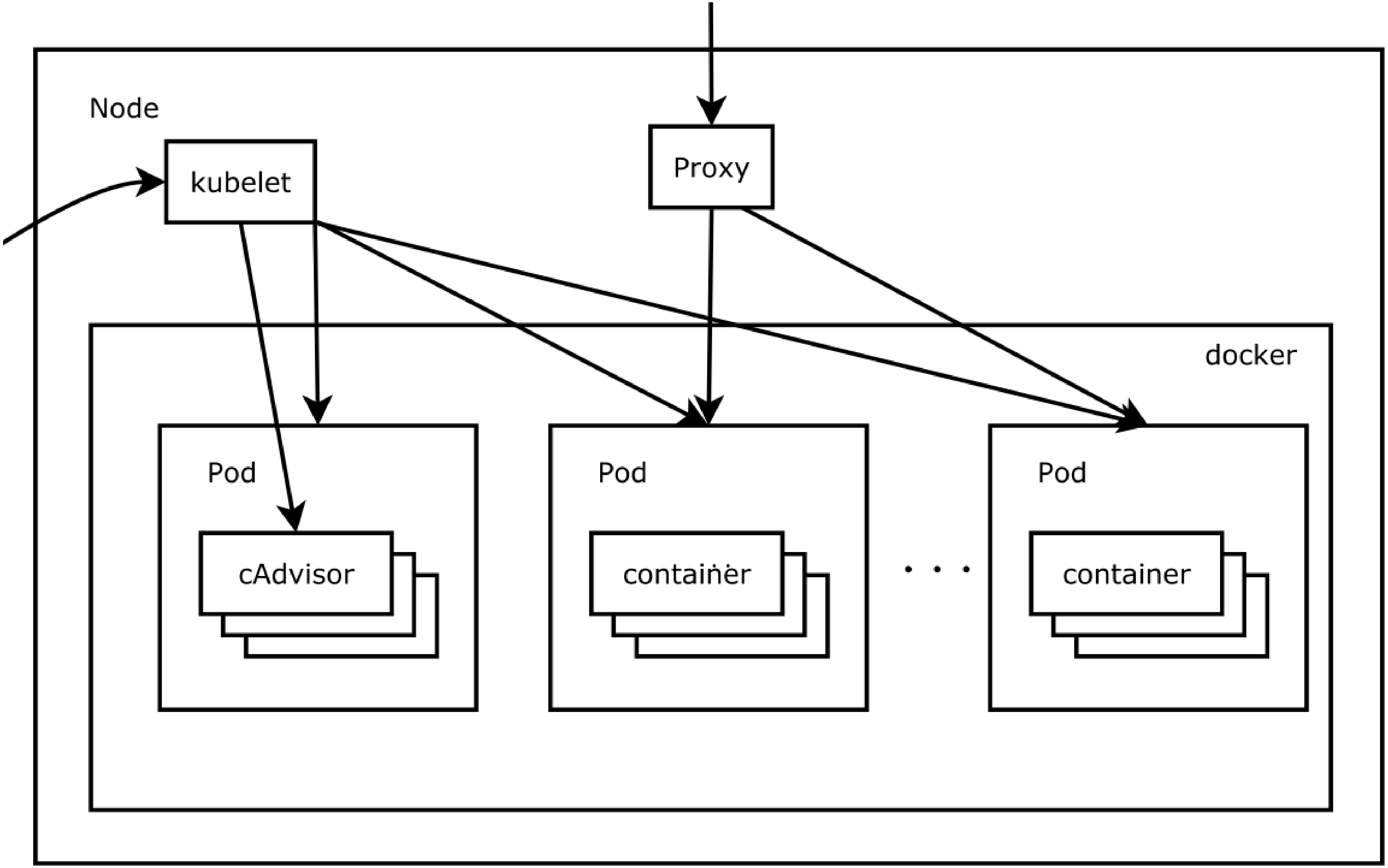
- Cluster Architecture
- High Availability
- Multi Cluster/Region

# Architecture




Internet
Firewall
kubectl (user commands)
Node
kubelet
Proxy
docker
Pod
cAdvisor
Pod
containėr
. . .
Pod
container
authentication
authorization
APIs
scheduling
actuator
REST
(pods, services,
rep. controllers)
Scheduler
Scheduler
controller manager
(replication controller etc.)
Distributed
Watchable
Storage
(implemented via etcd)
Master components
Colocated, or spread across machines,
as dictated by cluster size.
Node
kubelet
Proxy
docker
Pod
cAdvisor
Pod
containėr
. . .
Pod
container


authentication
authorization
APIs
scheduling
actuator
REST
(pods, services,
rep. controllers)
Scheduler
Scheduler
controller manager
(replication controller etc.)
Master components
Colocated, or spread across machines,
as dictated by cluster size.
Distributed
Watchable
Storage
(implemented via etcd)


Node
kubelet
Proxy
docker
Pod
cAdvisor
Pod
container
. . .
Pod
container

# Master

- api-server
  - Provides outbound Kubernetes REST API
  - Validates requests
  - Saves cluster state to etcd

# Master

- controller-manager
    - Runs "control loops"
    - Regulates the state of the system
    - Watches cluster state through the api-server
    - Changes current state towards the desired state
        - e.g. checks correct number of pods running

## Master

- Scheduler
    - Selects node on which to run a pod

# Master

- etcd
  - Distributed, consistent key-value store for shared configuration and service discovery

# Node

- kubelet
  - Agent that runs on each node
  - Takes a set of `PodSpecs` from API server
  - Starts containers to fulfill specs
  - Exposes monitoring data

# Node

- kube-proxy
  - Implements service endpoints (virtual IPs)
  - IPTables

# High Availability And Multi Region

# Highly Available

Why do we want our systems to be Highly Available?

"Everything fails, all the time"

-Werner Vogels

What does HA even mean?

*High availability is a characteristic of a system, which aims to ensure an agreed level of operational performance, usually uptime, for a higher than normal period.*

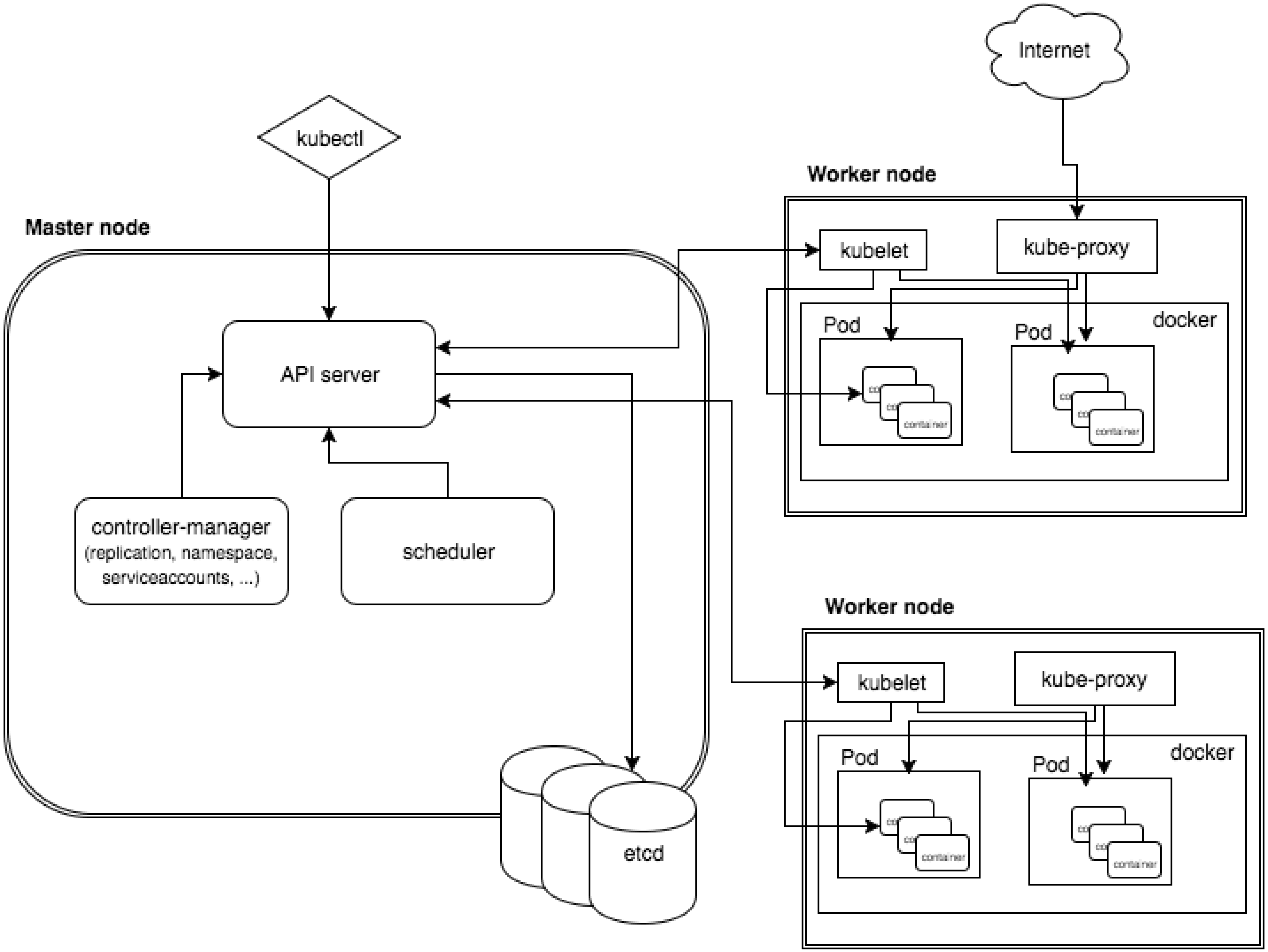
-Wikipedia

(This sounds a bit vague)

When we talk about systems which are Highly Available we mean that there should be no **single point of failure**

What are the potential single points of failure in a Kubernetes cluster?

# Kubernetes Architecture



# HA Kubernetes

## Master

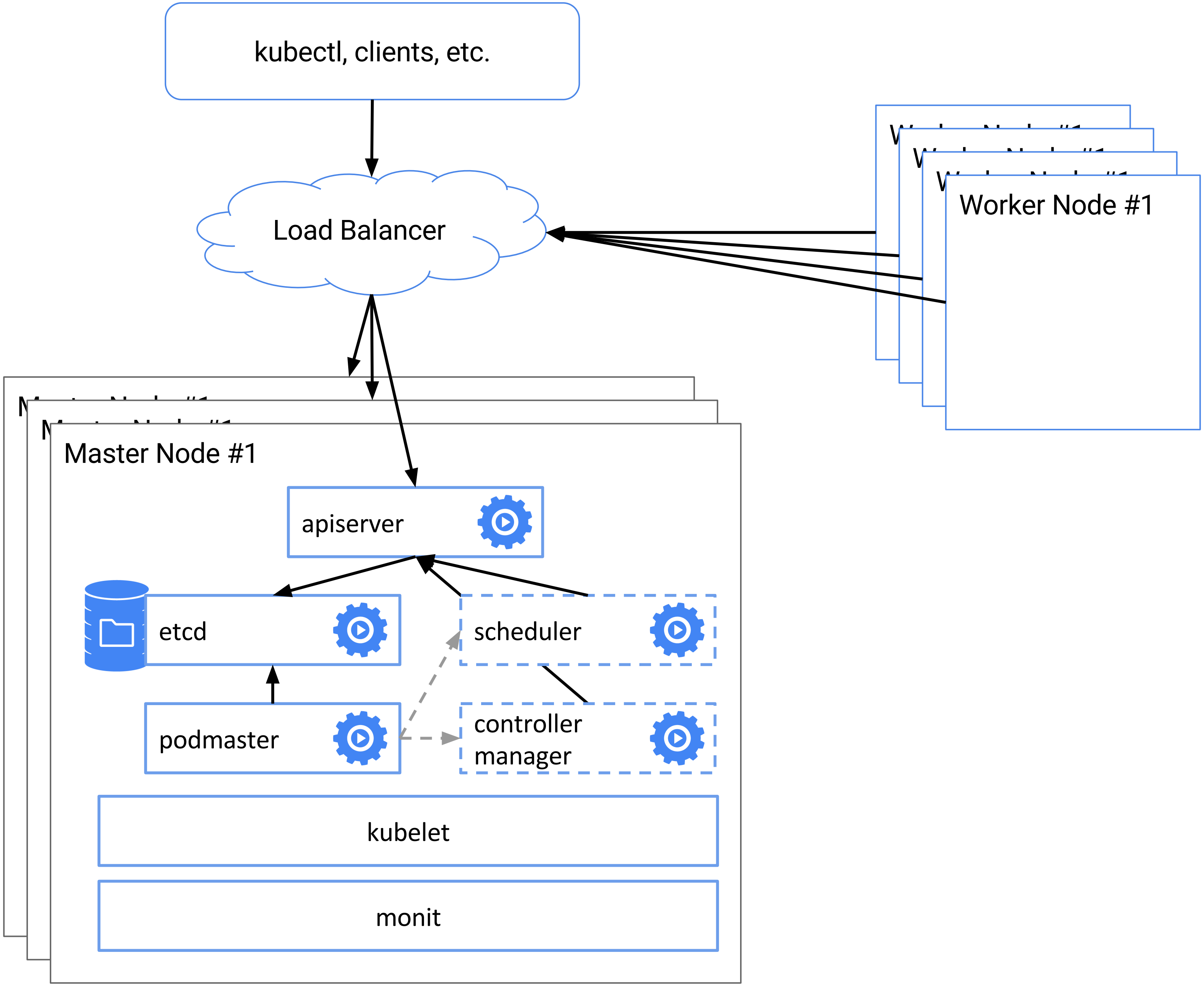
- API Server
- Controller Manager*
- Scheduler*

*Will perform leader election (given the `--leader-elect` flag)

# HA Kubernetes

## Etcd

- Etcd maintains the state of our cluster. This is crucial to maintain high availability.
- We can accomplish by creating a (minimum) 3 node Etcd cluster.
- We can also use persistent storage (e.g. on a cloud provider) to ensure no data is lost.


kubectl, clients, etc.
Load Balancer
Worker Node #1
Master Node #1
apiserver
etcd
scheduler
podmaster
controller manager
kubelet
monit

# To co-locate Masters and Etcd nodes?

This is a trade-off between managing/paying for more instances vs isolation.

# Load Balancing the API Server

- Need to handle master failure
- Kube-proxies need to point to API Server
- kubectl (or any other integration as well)

Common HA Setup on the Cloud

Spread across 3 regions

# How to setup a Highly Available Kubernetes cluster?

# Easy :)

```
$ gcloud container clusters create my-first-cluster
```

# Depends on your Infrastructure

- Google Kubernetes Engine
- Amazon EKS
- Azure Container Service
- OpenShift Origin
- Giant Swarm
- Kubermatic
- Tectonic by CoreOS
- IBM Cloud Container Service
- Kubespray
- Kops
- Kube-Up
- Kubeadm
- Kubicorn

# Depends on your Needs

- Budget (time and money)
- Are you running in the cloud or onprem?
- Do you have dedicated infra/ops team?
- What are your security requirements?
- Do you like to do things the hard way :)?

# Role-based Access Control (RBAC)

# A bit about Roles

Role-based access control (RBAC) is a common approach to managing users’ access to resources or operations.

Permissions specify exactly which resources and actions can be accessed.

The basic principle is: instead of separately managing the permissions of each user, permissions are given to roles, which are then assigned to users, or better - groups of users.

# Roles Bundle Permissions

- Managing permissions per user can be a tedious task when many users are involved.
- As users are added to the system, maintaining user permissions becomes harder and more prone to errors.
- Incorrect assignment of permissions can block users’ access to required systems, or worse - allow unauthorized users to access restricted areas or perform risky operations.

- A regular user can only perform a limited number of actions (e.g. get, watch, list).
- A closer look into these user actions can reveal that some actions tend to go together e.g. checking logs.
- Once roles are identified and assigned to each user, permissions can then be assigned to roles, instead of users.

Managing the permissions of a small number of roles is a much easier task.

# Basic concepts

**Rule**: grants permission

- Applies to resource types
- Grants verbs (create, edit, view, delete)
- (Cluster)Role
  - Cluster wide / within a namespace
  - List of rules
- (Cluster)RoleBinding
  - Connects (Cluster)Role to User
  - Both human & service account

# API overview

The RBAC API declares four top-level types which will be covered in this section:

- Role
- ClusterRole
- RoleBinding
- ClusterRoleBinding

# Role

A `Role` contains rules that represent a set of permissions. Permissions are additive (there are no “deny” rules).

A `Role` can be defined within a namespace, or cluster-wide (`Role` vs `ClusterRole`)

Here’s an example `Role` in the “default” namespace that can be used to grant read access to pods:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

# Cluster Role

- A `ClusterRole` can be used to grant the same permissions as a `Role`, but because they are cluster-scoped, they can also be used to grant access to:
- cluster-scoped resources (like nodes)
- namespaced resources (like pods) across all namespaces

# Cluster Role

The following ClusterRole can be used to grant read access to secrets in any particular namespace, or across all namespaces

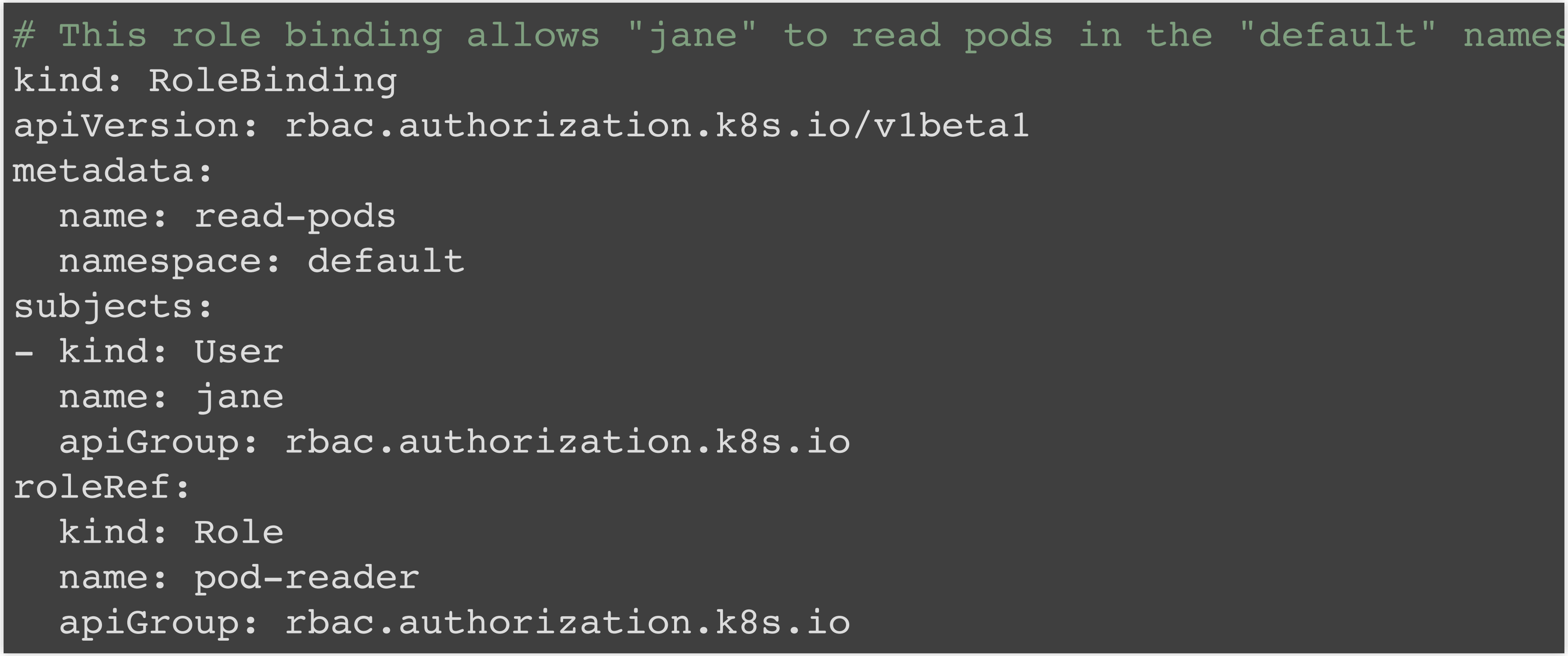
```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

# RoleBinding

A role binding grants the permissions defined in a role to a user

Permissions can be granted within a namespace with a `RoleBinding`, or cluster-wide with a `ClusterRoleBinding`.

A `RoleBinding` may reference a `Role` in the same namespace. The following `RoleBinding` grants the “pod-reader” role to the user “jane” within the “default” namespace.



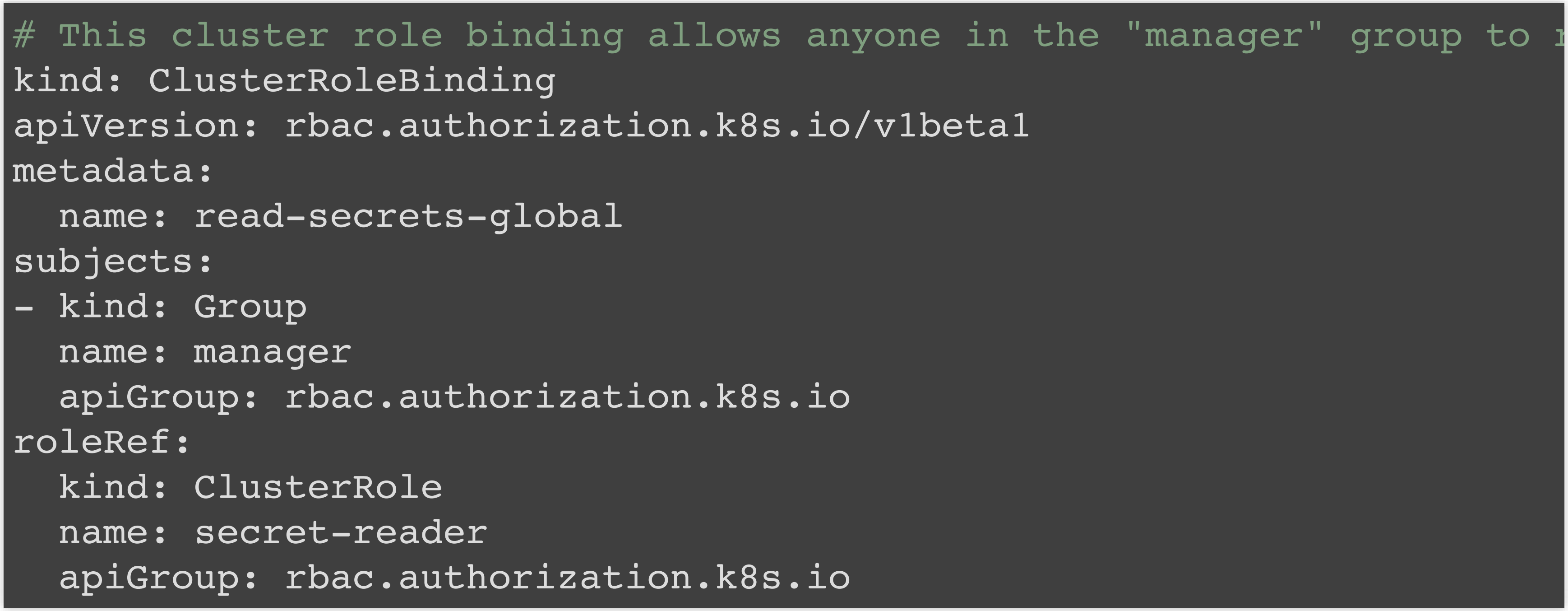
```
# This role binding allows "jane" to read pods in the "default" names
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

In this example, even though the following RoleBinding refers to a `ClusterRole`, **dave** will only be able read secrets in the **development** namespace (the namespace of the RoleBinding).

```
# This role binding allows "dave" to read secrets in the "development
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: read-secrets
  namespace: development # This only grants permissions within the "d
subjects:
- kind: User
  name: dave
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

# Cluster Role Binding

A `ClusterRoleBinding` may be used to grant permission at the cluster level and in all namespaces. The following `ClusterRoleBinding` allows any user in the group “manager” to read secrets in any namespace.



```
# This cluster role binding allows anyone in the "manager" group to r
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: read-secrets-global
subjects:
- kind: Group
  name: manager
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

Next up Monitoring...