Cluster Architecture

The architectural concepts behind Kubernetes.

1 - Nodes

Kubernetes runs your workload by placing containers into Pods to run on *Nodes*. A node may be a virtual or physical machine, depending on the cluster. Each node is managed by the control plane and contains the services necessary to run Pods

Typically you have several nodes in a cluster; in a learning or resource-limited environment, you might have only one node.

The components on a node include the kubelet, a container runtime, and the kube-proxy.

Management

There are two main ways to have Nodes added to the API server:

1. The kubelet on a node self-registers to the control plane
2. You (or another human user) manually add a Node object

After you create a Node object, or the kubelet on a node self-registers, the control plane checks whether the new Node object is valid. For example, if you try to create a Node from the following JSON manifest:

```
{
  "kind": "Node",
  "apiVersion": "v1",
  "metadata": {
    "name": "10.240.79.157",
    "labels": {
      "name": "my-first-k8s-node"
    }
  }
}
```

Kubernetes creates a Node object internally (the representation). Kubernetes checks that a kubelet has registered to the API server that matches the `metadata.name` field of the Node. If the node is healthy (i.e. all necessary services are running), then it is eligible to run a Pod. Otherwise, that node is ignored for any cluster activity until it becomes healthy.

> **Note:**
> Kubernetes keeps the object for the invalid Node and continues checking to see whether it becomes healthy.
>
> You, or a controller, must explicitly delete the Node object to stop that health checking.

The name of a Node object must be a valid DNS subdomain name.

Node name uniqueness

The name identifies a Node. Two Nodes cannot have the same name at the same time. Kubernetes also assumes that a resource with the same name is the same object. In case of a Node, it is implicitly assumed that an instance using the same name will have the same state (e.g. network settings, root disk contents). This may lead to inconsistencies if an instance was modified without changing its name. If the Node needs to be replaced or updated significantly, the existing Node object needs to be removed from API server first and re-added after the update.

Self-registration of Nodes

When the kubelet flag `--register-node` is true (the default), the kubelet will attempt to register itself with the API server. This is the preferred pattern, used by most distros.

For self-registration, the kubelet is started with the following options:

- `--kubeconfig` - Path to credentials to authenticate itself to the API server.
- `--cloud-provider` - How to talk to a cloud provider to read metadata about itself.
- `--register-node` - Automatically register with the API server.
- `--register-with-taints` - Register the node with the given list of taints (comma separated `<key>=<value>:<effect>`).

  No-op if `register-node` is false.
- `--node-ip` - IP address of the node.
- `--node-labels` - Labels to add when registering the node in the cluster (see label restrictions enforced by the NodeRestriction admission plugin).
- `--node-status-update-frequency` - Specifies how often kubelet posts node status to master.

When the Node authorization mode and NodeRestriction admission plugin are enabled, kubelets are only authorized to create/modify their own Node resource.

Manual Node administration

You can create and modify Node objects using kubectl.

When you want to create Node objects manually, set the kubelet flag `--register-node=false`.

You can modify Node objects regardless of the setting of `--register-node`. For example, you can set labels on an existing Node or mark it unschedulable.

You can use labels on Nodes in conjunction with node selectors on Pods to control scheduling. For example, you can constrain a Pod to only be eligible to run on a subset of the available nodes.

Marking a node as unschedulable prevents the scheduler from placing new pods onto that Node but does not affect existing Pods on the Node. This is useful as a preparatory step before a node reboot or other maintenance.

To mark a Node unschedulable, run:

```
kubectl cordon $NODENAME
```

> **Note:** Pods that are part of a DaemonSet tolerate being run on an unschedulable Node. DaemonSets typically provide node-local services that should run on the Node even if it is being drained of workload applications.

Node status

A Node's status contains the following information:

- Addresses
- Conditions
- Capacity and Allocatable
- Info

You can use `kubectl` to view a Node's status and other details:

```
kubectl describe node <insert-node-name-here>
```

Each section of the output is described below.

Addresses

The usage of these fields varies depending on your cloud provider or bare metal configuration.

- HostName: The hostname as reported by the node's kernel. Can be overridden via the kubelet `--hostname-override` parameter.
- ExternalIP: Typically the IP address of the node that is externally routable (available from outside the cluster).
- InternalIP: Typically the IP address of the node that is routable only within the cluster.

Conditions

The `conditions` field describes the status of all `Running` nodes. Examples of conditions include:

Node Condition	Description
`Ready`	`True` if the node is healthy and ready to accept pods, `False` if the node is not healthy and is not accepting pods, and `Unknown` if the node controller has not heard from the node in the last `node-monitor-grace-period` (default is 40 seconds)
`DiskPressure`	`True` if pressure exists on the disk size--that is, if the disk capacity is low; otherwise `False`
`MemoryPressure`	`True` if pressure exists on the node memory--that is, if the node memory is low; otherwise `False`
`PIDPressure`	`True` if pressure exists on the processes—that is, if there are too many processes on the node; otherwise `False`
`NetworkUnavailable`	`True` if the network for the node is not correctly configured, otherwise `False`

> **Note:** If you use command-line tools to print details of a cordoned Node, the Condition includes `SchedulingDisabled`. `SchedulingDisabled` is not a Condition in the Kubernetes API; instead, cordoned nodes are marked Unschedulable in their spec.

The node condition is represented as a JSON object. For example, the following structure describes a healthy node:

```
"conditions": [
  {
    "type": "Ready",
    "status": "True",
```

```
    "reason": "KubeletReady",
    "message": "kubelet is posting ready status",
    "lastHeartbeatTime": "2019-06-05T18:38:35Z",
    "lastTransitionTime": "2019-06-05T11:41:27Z"
  }
]
```

If the Status of the Ready condition remains `Unknown` or `False` for longer than the `pod-eviction-timeout` (an argument passed to the kube-controller-manager), then all the Pods on the node are scheduled for deletion by the node controller. The default eviction timeout duration is **five minutes**. In some cases when the node is unreachable, the API server is unable to communicate with the kubelet on the node. The decision to delete the pods cannot be communicated to the kubelet until communication with the API server is re-established. In the meantime, the pods that are scheduled for deletion may continue to run on the partitioned node.

The node controller does not force delete pods until it is confirmed that they have stopped running in the cluster. You can see the pods that might be running on an unreachable node as being in the `Terminating` or `Unknown` state. In cases where Kubernetes cannot deduce from the underlying infrastructure if a node has permanently left a cluster, the cluster administrator may need to delete the node object by hand. Deleting the node object from Kubernetes causes all the Pod objects running on the node to be deleted from the API server and frees up their names.

The node lifecycle controller automatically creates taints that represent conditions. The scheduler takes the Node's taints into consideration when assigning a Pod to a Node. Pods can also have tolerations which let them tolerate a Node's taints.

See Taint Nodes by Condition for more details.

Capacity and Allocatable

Describes the resources available on the node: CPU, memory, and the maximum number of pods that can be scheduled onto the node.

The fields in the capacity block indicate the total amount of resources that a Node has. The allocatable block indicates the amount of resources on a Node that is available to be consumed by normal Pods.

You may read more about capacity and allocatable resources while learning how to reserve compute resources on a Node.

Info

Describes general information about the node, such as kernel version, Kubernetes version (kubelet and kube-proxy version), Docker version (if used), and OS name. This information is gathered by Kubelet from the node.

Node controller

The node controller is a Kubernetes control plane component that manages various aspects of nodes.

The node controller has multiple roles in a node's life. The first is assigning a CIDR block to the node when it is registered (if CIDR assignment is turned on).

The second is keeping the node controller's internal list of nodes up to date with the cloud provider's list of available machines. When running in a cloud environment and whenever a node is unhealthy, the node controller asks the cloud provider if the VM for that node is still available. If not, the node controller deletes the node from its list of nodes.

The third is monitoring the nodes' health. The node controller is responsible for:

- Updating the NodeReady condition of NodeStatus to ConditionUnknown when a node becomes unreachable, as the node controller stops receiving heartbeats for some reason

such as the node being down.
- Evicting all the pods from the node using graceful termination if the node continues to be unreachable. The default timeouts are 40s to start reporting ConditionUnknown and 5m after that to start evicting pods.

The node controller checks the state of each node every `--node-monitor-period` seconds.

Heartbeats

Heartbeats, sent by Kubernetes nodes, help determine the availability of a node.

There are two forms of heartbeats: updates of `NodeStatus` and the Lease object. Each Node has an associated Lease object in the `kube-node-lease` namespace. Lease is a lightweight resource, which improves the performance of the node heartbeats as the cluster scales.

The kubelet is responsible for creating and updating the `NodeStatus` and a Lease object.

- The kubelet updates the `NodeStatus` either when there is change in status or if there has been no update for a configured interval. The default interval for `NodeStatus` updates is 5 minutes, which is much longer than the 40 second default timeout for unreachable nodes.
- The kubelet creates and then updates its Lease object every 10 seconds (the default update interval). Lease updates occur independently from the `NodeStatus` updates. If the Lease update fails, the kubelet retries with exponential backoff starting at 200 milliseconds and capped at 7 seconds.

Reliability

In most cases, the node controller limits the eviction rate to `--node-eviction-rate` (default 0.1) per second, meaning it won't evict pods from more than 1 node per 10 seconds.

The node eviction behavior changes when a node in a given availability zone becomes unhealthy. The node controller checks what percentage of nodes in the zone are unhealthy (NodeReady condition is ConditionUnknown or ConditionFalse) at the same time:

- If the fraction of unhealthy nodes is at least `--unhealthy-zone-threshold` (default 0.55), then the eviction rate is reduced.
- If the cluster is small (i.e. has less than or equal to `--large-cluster-size-threshold` nodes - default 50), then evictions are stopped.
- Otherwise, the eviction rate is reduced to `--secondary-node-eviction-rate` (default 0.01) per second.

The reason these policies are implemented per availability zone is because one availability zone might become partitioned from the master while the others remain connected. If your cluster does not span multiple cloud provider availability zones, then there is only one availability zone (i.e. the whole cluster).

A key reason for spreading your nodes across availability zones is so that the workload can be shifted to healthy zones when one entire zone goes down. Therefore, if all nodes in a zone are unhealthy, then the node controller evicts at the normal rate of `--node-eviction-rate`. The corner case is when all zones are completely unhealthy (i.e. there are no healthy nodes in the cluster). In such a case, the node controller assumes that there is some problem with master connectivity and stops all evictions until some connectivity is restored.

The node controller is also responsible for evicting pods running on nodes with `NoExecute` taints, unless those pods tolerate that taint. The node controller also adds taints corresponding to node problems like node unreachable or not ready. This means that the scheduler won't place Pods onto unhealthy nodes.

> **Caution:** `kubectl cordon` marks a node as 'unschedulable', which has the side effect of the service controller removing the node from any LoadBalancer node target lists it was previously eligible for, effectively removing incoming load balancer traffic from the cordoned node(s).

Node capacity

Node objects track information about the Node's resource capacity: for example, the amount of memory available and the number of CPUs. Nodes that self register report their capacity during registration. If you manually add a Node, then you need to set the node's capacity information when you add it.

The Kubernetes scheduler ensures that there are enough resources for all the Pods on a Node. The scheduler checks that the sum of the requests of containers on the node is no greater than the node's capacity. That sum of requests includes all containers managed by the kubelet, but excludes any containers started directly by the container runtime, and also excludes any processes running outside of the kubelet's control.

> **Note:** If you want to explicitly reserve resources for non-Pod processes, see reserve resources for system daemons.

Node topology

FEATURE STATE: `Kubernetes v1.16 [alpha]`

If you have enabled the `TopologyManager` feature gate, then the kubelet can use topology hints when making resource assignment decisions. See Control Topology Management Policies on a Node for more information.

Graceful Node Shutdown

FEATURE STATE: `Kubernetes v1.20 [alpha]`

If you have enabled the `GracefulNodeShutdown` feature gate, then the kubelet attempts to detect the node system shutdown and terminates pods running on the node. Kubelet ensures that pods follow the normal pod termination process during the node shutdown.

When the `GracefulNodeShutdown` feature gate is enabled, kubelet uses systemd inhibitor locks to delay the node shutdown with a given duration. During a shutdown, kubelet terminates pods in two phases:

1. Terminate regular pods running on the node.
2. Terminate critical pods running on the node.

Graceful Node Shutdown feature is configured with two `KubeletConfiguration` options:

- `ShutdownGracePeriod`:
  - Specifies the total duration that the node should delay the shutdown by. This is the total grace period for pod termination for both regular and critical pods.
- `ShutdownGracePeriodCriticalPods`:
  - Specifies the duration used to terminate critical pods during a node shutdown. This should be less than `ShutdownGracePeriod`.

For example, if `ShutdownGracePeriod=30s`, and `ShutdownGracePeriodCriticalPods=10s`, kubelet will delay the node shutdown by 30 seconds. During the shutdown, the first 20 (30-10) seconds would be reserved for gracefully terminating normal pods, and the last 10 seconds would be reserved for terminating critical pods.

What's next

- Learn about the components that make up a node.
- Read the API definition for Node.
- Read the Node section of the architecture design document.
- Read about taints and tolerations.

2 - Control Plane-Node Communication

This document catalogs the communication paths between the control plane (apiserver) and the Kubernetes cluster. The intent is to allow users to customize their installation to harden the network configuration such that the cluster can be run on an untrusted network (or on fully public IPs on a cloud provider).

Node to Control Plane

Kubernetes has a "hub-and-spoke" API pattern. All API usage from nodes (or the pods they run) terminates at the apiserver. None of the other control plane components are designed to expose remote services. The apiserver is configured to listen for remote connections on a secure HTTPS port (typically 443) with one or more forms of client authentication enabled. One or more forms of authorization should be enabled, especially if anonymous requests or service account tokens are allowed.

Nodes should be provisioned with the public root certificate for the cluster such that they can connect securely to the apiserver along with valid client credentials. A good approach is that the client credentials provided to the kubelet are in the form of a client certificate. See kubelet TLS bootstrapping for automated provisioning of kubelet client certificates.

Pods that wish to connect to the apiserver can do so securely by leveraging a service account so that Kubernetes will automatically inject the public root certificate and a valid bearer token into the pod when it is instantiated. The `kubernetes` service (in `default` namespace) is configured with a virtual IP address that is redirected (via kube-proxy) to the HTTPS endpoint on the apiserver.

The control plane components also communicate with the cluster apiserver over the secure port.

As a result, the default operating mode for connections from the nodes and pods running on the nodes to the control plane is secured by default and can run over untrusted and/or public networks.

Control Plane to node

There are two primary communication paths from the control plane (apiserver) to the nodes. The first is from the apiserver to the kubelet process which runs on each node in the cluster. The second is from the apiserver to any node, pod, or service through the apiserver's proxy functionality.

apiserver to kubelet

The connections from the apiserver to the kubelet are used for:

- Fetching logs for pods.
- Attaching (through kubectl) to running pods.
- Providing the kubelet's port-forwarding functionality.

These connections terminate at the kubelet's HTTPS endpoint. By default, the apiserver does not verify the kubelet's serving certificate, which makes the connection subject to man-in-the-middle attacks and **unsafe** to run over untrusted and/or public networks.

To verify this connection, use the `--kubelet-certificate-authority` flag to provide the apiserver with a root certificate bundle to use to verify the kubelet's serving certificate.

If that is not possible, use SSH tunneling between the apiserver and kubelet if required to avoid connecting over an untrusted or public network.

Finally, Kubelet authentication and/or authorization should be enabled to secure the kubelet API.

apiserver to nodes, pods, and services

The connections from the apiserver to a node, pod, or service default to plain HTTP connections and are therefore neither authenticated nor encrypted. They can be run over a secure HTTPS connection by prefixing `https:` to the node, pod, or service name in the API URL, but they will not validate the certificate provided by the HTTPS endpoint nor provide client credentials. So while the connection will be encrypted, it will not provide any guarantees of integrity. These connections **are not currently safe** to run over untrusted or public networks.

SSH tunnels

Kubernetes supports SSH tunnels to protect the control plane to nodes communication paths. In this configuration, the apiserver initiates an SSH tunnel to each node in the cluster (connecting to the ssh server listening on port 22) and passes all traffic destined for a kubelet, node, pod, or service through the tunnel. This tunnel ensures that the traffic is not exposed outside of the network in which the nodes are running.

SSH tunnels are currently deprecated, so you shouldn't opt to use them unless you know what you are doing. The Konnectivity service is a replacement for this communication channel.

Konnectivity service

FEATURE STATE: `Kubernetes v1.18 [beta]`

As a replacement to the SSH tunnels, the Konnectivity service provides TCP level proxy for the control plane to cluster communication. The Konnectivity service consists of two parts: the Konnectivity server in the control plane network and the Konnectivity agents in the nodes network. The Konnectivity agents initiate connections to the Konnectivity server and maintain the network connections. After enabling the Konnectivity service, all control plane to nodes traffic goes through these connections.

Follow the Konnectivity service task to set up the Konnectivity service in your cluster.

3 - Controllers

In robotics and automation, a *control loop* is a non-terminating loop that regulates the state of a system.

Here is one example of a control loop: a thermostat in a room.

When you set the temperature, that's telling the thermostat about your *desired state*. The actual room temperature is the *current state*. The thermostat acts to bring the current state closer to the desired state, by turning equipment on or off.

In Kubernetes, controllers are control loops that watch the state of your cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state.

Controller pattern

A controller tracks at least one Kubernetes resource type. These objects have a spec field that represents the desired state. The controller(s) for that resource are responsible for making the current state come closer to that desired state.

The controller might carry the action out itself; more commonly, in Kubernetes, a controller will send messages to the API server that have useful side effects. You'll see examples of this below.

Control via API server

The Job controller is an example of a Kubernetes built-in controller. Built-in controllers manage state by interacting with the cluster API server.

Job is a Kubernetes resource that runs a Pod, or perhaps several Pods, to carry out a task and then stop.

(Once scheduled, Pod objects become part of the desired state for a kubelet).

When the Job controller sees a new task it makes sure that, somewhere in your cluster, the kubelets on a set of Nodes are running the right number of Pods to get the work done. The Job controller does not run any Pods or containers itself. Instead, the Job controller tells the API server to create or remove Pods. Other components in the control plane act on the new information (there are new Pods to schedule and run), and eventually the work is done.

After you create a new Job, the desired state is for that Job to be completed. The Job controller makes the current state for that Job be nearer to your desired state: creating Pods that do the work you wanted for that Job, so that the Job is closer to completion.

Controllers also update the objects that configure them. For example: once the work is done for a Job, the Job controller updates that Job object to mark it `Finished`.

(This is a bit like how some thermostats turn a light off to indicate that your room is now at the temperature you set).

Direct control

By contrast with Job, some controllers need to make changes to things outside of your cluster.

For example, if you use a control loop to make sure there are enough Nodes in your cluster, then that controller needs something outside the current cluster to set up new Nodes when needed.

Controllers that interact with external state find their desired state from the API server, then communicate directly with an external system to bring the current state closer in line.

(There actually is a controller that horizontally scales the nodes in your cluster.)

The important point here is that the controller makes some change to bring about your desired state, and then reports current state back to your cluster's API server. Other control loops can observe that reported data and take their own actions.

In the thermostat example, if the room is very cold then a different controller might also turn on a frost protection heater. With Kubernetes clusters, the control plane indirectly works with IP address management tools, storage services, cloud provider APIs, and other services by extending Kubernetes to implement that.

Desired versus current state

Kubernetes takes a cloud-native view of systems, and is able to handle constant change.

Your cluster could be changing at any point as work happens and control loops automatically fix failures. This means that, potentially, your cluster never reaches a stable state.

As long as the controllers for your cluster are running and able to make useful changes, it doesn't matter if the overall state is stable or not.

Design

As a tenet of its design, Kubernetes uses lots of controllers that each manage a particular aspect of cluster state. Most commonly, a particular control loop (controller) uses one kind of resource as its desired state, and has a different kind of resource that it manages to make that desired state happen. For example, a controller for Jobs tracks Job objects (to discover new work) and Pod objects (to run the Jobs, and then to see when the work is finished). In this case something else creates the Jobs, whereas the Job controller creates Pods.

It's useful to have simple controllers rather than one, monolithic set of control loops that are interlinked. Controllers can fail, so Kubernetes is designed to allow for that.

> **Note:**
> There can be several controllers that create or update the same kind of object. Behind the scenes, Kubernetes controllers make sure that they only pay attention to the resources linked to their controlling resource.
>
> For example, you can have Deployments and Jobs; these both create Pods. The Job controller does not delete the Pods that your Deployment created, because there is information (labels) the controllers can use to tell those Pods apart.

Ways of running controllers

Kubernetes comes with a set of built-in controllers that run inside the kube-controller-manager. These built-in controllers provide important core behaviors.

The Deployment controller and Job controller are examples of controllers that come as part of Kubernetes itself ("built-in" controllers). Kubernetes lets you run a resilient control plane, so that if any of the built-in controllers were to fail, another part of the control plane will take over the work.

You can find controllers that run outside the control plane, to extend Kubernetes. Or, if you want, you can write a new controller yourself. You can run your own controller as a set of Pods, or externally to Kubernetes. What fits best will depend on what that particular controller does.

What's next

- Read about the Kubernetes control plane
- Discover some of the basic Kubernetes objects
- Learn more about the Kubernetes API
- If you want to write your own controller, see Extension Patterns in Extending Kubernetes.

4 - Cloud Controller Manager

FEATURE STATE: `Kubernetes v1.11 [beta]`

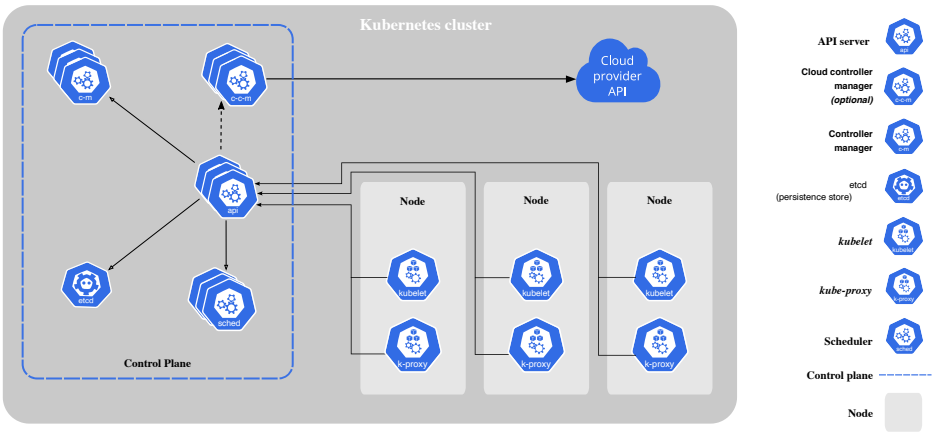
Cloud infrastructure technologies let you run Kubernetes on public, private, and hybrid clouds. Kubernetes believes in automated, API-driven infrastructure without tight coupling between components.

The cloud-controller-manager is a Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster.

By decoupling the interoperability logic between Kubernetes and the underlying cloud infrastructure, the cloud-controller-manager component enables cloud providers to release features at a different pace compared to the main Kubernetes project.

The cloud-controller-manager is structured using a plugin mechanism that allows different cloud providers to integrate their platforms with Kubernetes.

Design



The cloud controller manager runs in the control plane as a replicated set of processes (usually, these are containers in Pods). Each cloud-controller-manager implements multiple controllers in a single process.

> **Note:** You can also run the cloud controller manager as a Kubernetes addon rather than as part of the control plane.

Cloud controller manager functions

The controllers inside the cloud controller manager include:

Node controller

The node controller is responsible for creating Node objects when new servers are created in your cloud infrastructure. The node controller obtains information about the hosts running inside your tenancy with the cloud provider. The node controller performs the following functions:

1. Initialize a Node object for each server that the controller discovers through the cloud provider API.
2. Annotating and labelling the Node object with cloud-specific information, such as the region the node is deployed into and the resources (CPU, memory, etc) that it has available.
3. Obtain the node's hostname and network addresses.

4. Verifying the node's health. In case a node becomes unresponsive, this controller checks with your cloud provider's API to see if the server has been deactivated / deleted / terminated. If the node has been deleted from the cloud, the controller deletes the Node object from your Kubernetes cluster.

Some cloud provider implementations split this into a node controller and a separate node lifecycle controller.

Route controller

The route controller is responsible for configuring routes in the cloud appropriately so that containers on different nodes in your Kubernetes cluster can communicate with each other.

Depending on the cloud provider, the route controller might also allocate blocks of IP addresses for the Pod network.

Service controller

Services integrate with cloud infrastructure components such as managed load balancers, IP addresses, network packet filtering, and target health checking. The service controller interacts with your cloud provider's APIs to set up load balancers and other infrastructure components when you declare a Service resource that requires them.

Authorization

This section breaks down the access that the cloud controller managers requires on various API objects, in order to perform its operations.

Node controller

The Node controller only works with Node objects. It requires full access to read and modify Node objects.

`v1/Node`:

- Get
- List
- Create
- Update
- Patch
- Watch
- Delete

Route controller

The route controller listens to Node object creation and configures routes appropriately. It requires Get access to Node objects.

`v1/Node`:

- Get

Service controller

The service controller listens to Service object Create, Update and Delete events and then configures Endpoints for those Services appropriately.

To access Services, it requires List, and Watch access. To update Services, it requires Patch and Update access.

To set up Endpoints resources for the Services, it requires access to Create, List, Get, Watch, and Update.

`v1/Service`:

- List
- Get
- Watch
- Patch
- Update

Others

The implementation of the core of the cloud controller manager requires access to create Event objects, and to ensure secure operation, it requires access to create ServiceAccounts.

`v1/Event`:

- Create
- Patch
- Update

`v1/ServiceAccount`:

- Create

The RBAC ClusterRole for the cloud controller manager looks like:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cloud-controller-manager
rules:
- apiGroups:
  - ""
  resources:
  - events
  verbs:
  - create
  - patch
  - update
- apiGroups:
  - ""
  resources:
  - nodes
  verbs:
  - '*'
- apiGroups:
  - ""
  resources:
  - nodes/status
  verbs:
  - patch
- apiGroups:
  - ""
  resources:
  - services
  verbs:
  - list
  - patch
  - update
  - watch
- apiGroups:
  - ""
  resources:
  - serviceaccounts
  verbs:
  - create
- apiGroups:
  - ""
  resources:
```

```
  - persistentvolumes
  verbs:
  - get
  - list
  - update
  - watch
- apiGroups:
  - ""
  resources:
  - endpoints
  verbs:
  - create
  - get
  - list
  - watch
  - update
```

What's next

Cloud Controller Manager Administration has instructions on running and managing the cloud controller manager.

Want to know how to implement your own cloud controller manager, or extend an existing project?

The cloud controller manager uses Go interfaces to allow implementations from any cloud to be plugged in. Specifically, it uses the `CloudProvider` interface defined in `cloud.go` from kubernetes/cloud-provider.

The implementation of the shared controllers highlighted in this document (Node, Route, and Service), and some scaffolding along with the shared cloudprovider interface, is part of the Kubernetes core. Implementations specific to cloud providers are outside the core of Kubernetes and implement the `CloudProvider` interface.

For more information about developing plugins, see Developing Cloud Controller Manager.