Multi-tenancy models for Kubernetes

@davidopp, 31 May 2017

# Introduction

This doc covers three topics:

1) Various possible multi-tenancy models, and the authorization rules/tenant's view of the cluster that one might want to associate with each of them.

2) What kind of hierarchy we might want to add to Kubernetes, to make it easier to represent policies in a multi-tenant cluster. For example, one possibility is having a "tenant" concept that spans namespaces. Another (not mutually exclusive) is being able to set policies on subsets of pods within a namespace.

3) Virtualization: To what extent do we want to give tenants the illusion that they have their own private cluster.

Much of the doc is based on discussions from Kubernetes mailing lists (which are linked to from the doc), as well as discussions with @deads2k, @liggitt, @smarterclayton, and @thockin.

# Authorization policies for various multi-tenancy models

The chart below proposes what a single tenant’s view of the cluster would be in different multi-tenancy scenarios. (Note that the “SaaS multitenancy” column is a bit different from the others because there the tenants are not interacting directly with the Kubernetes API.)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **tenancy model**  ------------------------>          **can a non-privileged tenant... |  v** | soft multitenancy -- code and API access come from single org where people mostly trust each other (trusted code, trusted deployer); shared or dedicated nodes; example: Google | SaaS multitenancy -- a trusted higher-level control plane interacts with the Kubernetes API to run untrusted code on behalf of untrusted users (untrusted code, trusted deployer); shared or dedicated nodes; [tenants here are users whose code is running, NOT the deployer, which is not sharing the cluster] | CaaS multitenancy model 1 -- untrusted/untrusting users sharing the control plane, users knows it’s a shared control plane, dedicated nodes per tenant (autoscaled); untrusted code, untrusted deployer | CaaS multitenancy model 2 - same as model 1 but with shared nodes; example: OpenShift Online |
| see the set of nodes in the cluster | yes | no (since they can’t access the control plane) | only the ones dedicated to them | no |
| see node properties (capacity, labels, taints, conditions, IP, ...) | yes | no (same reason) | yes (for the nodes dedicated to them) | no |
| use nodeSelector, nodeName, node (anti-)affinity | yes | no (same reason) | yes | no |
| set node properties (e.g. add taints and labels) | any user (dedicated nodes model), cluster admin (shared nodes model) | no (same reason) | yes (for the nodes dedicated to them) | only cluster admin |
| see what’s running on node + node logs | yes | no (same reason) | yes (for the nodes dedicated to them) | no (or only their own things) |
| see other tenants’ API objects | yes | no (same reason) | no | no |
| “default” namespace for non-cluster-admin | shared read/write | read-only | read-only | read-only |
| “kube-system” namespace for non-cluster-admin | read-only | hidden | hidden | hidden |
| see other tenants’ services in DNS | yes | no | no | no |
| send/receive network traffic between tenants without external service | yes | no | no | no |
| see other tenants’ application data | no | no | no | no |
| mutate other tenants’ objects | no | no | no | no |
| create a namespace without worrying about name conflicts | no | no | no | no |
| provision new nodes | only indirectly via cluster autoscaler | only indirectly via cluster autoscaler | yes? (to their set of nodes) + cluster autoscaler | only indirectly via cluster autoscaler |
| nodes run in whose project (on GKE) | user’s (dedicated nodes model) or Google’s (shared nodes model) | Google’s | user’s or Google’s | Google’s |
| control/tune node upgrade, auto-repair, etc. policies | yes (dedicated nodes model), cluster admin (shared nodes model) | no | yes (for their nodes) | no |
| choose arbitrary host ports | yes | no | yes | no |
| choose NodePort | yes | no | yes | no |
| use hostPath, provision local vols. | yes | no | yes | no |

## Supporting a tenant concept that spans namespaces

Namespace-local operations are easy to support in any tenancy model because the addressable universe is limited to the namespace in which the operation is performed, and we assume a namespace is a trust domain. Self-service policies can be installed that control permissions or control allowed operations within that namespace, e.g. ResourceQuota or RoleBinding today, as it’s easy to use RBAC to control who has permission to create and modify such policies.

But in some use cases, the one-namespace-per-tenant model may be too restrictive. For example, a company with a single Kubernetes cluster might want to allow each department to manage a set of namespaces (perhaps one namespace per user or application), while having both department-level and namespace-level rules. (The motivation for this is similar to that for the GCP project/folder/org hierarchy.) Or multiple companies might share a cluster, with each company using a set of namespaces, and it is desired to have company-wide rules as well as per-namespace rules. To satisfy these use cases, we can add a “tenant” concept to Kubernetes. Tenants would still be aware that they are sharing the system, and any particular namespace name could only be used by one tenant, but tenants and namespaces would be given an identifier to allow tenants to easily work with sets of namespaces as a group, and could be isolated on the boundary of this multi-namespace tenancy group in addition to the current isolation boundaries.

The referring object may be namespaced, as in NetworkPolicy, or non-namespaced, as in (OpenShift) ClusterResourceQuota. In either case there is the question of how to refer to a target object (or set of objects), given that it/they may be in any of the tenant’s namespaces. The main choices are

1. explicit namespace name(s), plus object name/selector
   1. pros: simple, explicit, secure (modulo recycling of namespace names to new tenants, which can be worked around by using namespace name + UUID)
   2. cons: tedious to manage, must be updated when namespaces are added/removed, limited to namespace
2. field selector over namespace field, plus object name/selector
   1. pros: secure (with same caveat as above)
   2. cons: limited to namespace
3. namespace selector (selector over labels on namespaces) plus object name/selector
   1. pros: flexible
   2. cons: any tenant who can modify their own namespace’s labels can insert themselves into any selected set
4. object selector
   1. pros: flexible
   2. cons: any tenant who can modify their own objects’ labels can insert themselves into any selected set

There is more discussion of this general topic in [this thread](https://groups.google.com/d/msg/kubernetes-sig-network/4Q1Xk1Jm4-M/UyNIW53XBAAJ).

Diving more into the “any tenant who can modify their own namespace’s labels can insert themselves into any selected set” con, here are two examples.

* If NetworkPolicy selected the allowed sending pods using a selector, anyone who added a label that matched the selector could make themselves an allowed sender.
* In a quota policy that identifies the covered pods using a selector, a tenant could DoS another tenant, and get free quota, by setting the appropriate label

There are several ways around this. (Note that we describe these assuming approach (3) i.e. namespace selector, but they also apply to approach (4), which has the same problem.)

* Allow only cluster administrators to set labels on namespaces
* Reserve a portion of the label keyspace as “secure labels” and allow only cluster administrators to set labels in that keyspace
* Conversely, reserve a portion of the label keyspace as mutable for each user (e.g. $USERNAME/\*, plus unprefixed keys) and make the rest immutable
* Enforce no special rules in code, but announce a design pattern wherein label values used for these kinds of rules should be large integers that cannot be easily guessed. Tell users to use this pattern if they want to protect themselves from other tenants. For example, a tenant might use “foo.com/tenant-id = <some random 256-bit value>” They would then include this key/value pair in the label selector when they want to select pods between namespaces or from a global rule (or it could be automatically added). OpenShift has a variant of this in ClusterResourceQuota, wherein the system automatically attaches a “openshift.io/requester: <user-name>” annotation to namespaces when they are created, and then ClusterResourceQuota can aggregate across namespaces from a single requester.

The above prevents a malicious tenant from inserting itself as part of some other tenant’s cross-namespace rule. If we want to protect against the opposite, i.e. a malicious tenant referring to other tenants’ namespaces, then we can use an admission controller to ensure a tenant-id is always part of the selector. But it’s not clear there is actually a danger here, since tenants can only harm themselves by being overly permissive in selecting pods in their policies. And tenants can use RBAC to ensure that the referrer has permissions on the referred-to objects.

When tenants use multiple namespaces, they lose the organizational (as in easy-to-keep-track-of-your-objects) benefits of a single namespace because now their objects are spread across namespaces. We’d like to make it easy for them to find their namespaces and objects. For example, we’d like them to be able to LIST a non-namespaced kind and see only their own objects. OpenShift has a “project” concept; projects map 1:1 with namespaces but a user can LIST projects and get back only “their” namespaces, with the system applying the rule that the user must have “get” permission on a namespace to see it. If all non-namespaced objects that belong to a tenant had a tenant-id label attached to them *and* the system required a tenant-id label selector on all LIST operations then this wouldn’t be necessary. (Note that some non-namespaced objects don’t logically belong to a tenant, for example a ClusterRole that a cluster admin wants to use in multiple namespaces, or a PodSecurityPolicy that the cluster admin wants to apply to all pods in the cluster, or a non-dedicated node.)

Along the same lines, when tenants use multiple namespaces, it would be nice for them to be able to write RBAC clusterroles that apply to all of their objects of some kind. For this it has been [suggested](https://github.com/kubernetes/kubernetes/issues/40403#issue-203014457) to allow a label selector over something like the tenant-id as part of an RBAC clusterrole definition, which would make it easy, for example, to grant “get” permission to all of a tenant’s namespaces in one fell swoop (assuming namespaces are labeled with tenant-id), which fits well with the pattern in the previous paragraph. This label-selector-in-RBAC-rules feature would also allow us to write rules like “ingress controller X should only be able to view secrets that the secret’s owner has explicitly labeled for consumption by that ingress controller.” This could be combined with an RBAC inverted index to make it efficient for the system to identify/list all the objects the tenant has permissions for.

One could imagine that when a cluster admin attaches a tenant-id label to a Node, it automatically makes that node dedicated to that tenant.

In this section we have proposed adding a “tenant” concept that users would be aware of. We can’t easily inject the tenant concept into pod and service DNS names ($ip.$namespace.pod.cluster.local for pods, $service.$namespace.svc.cluster.local for services) without serious backward compatibility issues. But if namespaces were tagged with a tenant-id, and the DNS server had a way to associate a query with the tenant-id of the tenant making the query, then we could use split-horizon DNS to ensure that a tenant only sees their own pods and services in DNS. Of course, in some cases tenants may want to connect to services offered by other tenants. Perhaps something like the Service Catalog and Service Broker would be useful here; discovering and connecting to another tenant’s service in the same cluster would be like discovering and connecting to any external service on the Internet, and an external service would be declared to expose the service (perhaps with some internal optimization that notices it’s in the same cluster, so doesn’t create an external loadbalancer).

In the model we have described, pods and services are not visible between tenants (via the API or DNS). This isolation can be made even stronger by giving each tenant a private virtual L3 network with a private RFC-1918 address space for their pods and services, rather than sharing these within the cluster.

## Rules within namespaces

The previous section proposed adding label selectors to RBAC clusterroles and policy rules in general, to allow rules that span namespaces. Once we have that, we can also write rules that apply to subsets of objects within a namespace (or subsets of objects within multiple namespaces). For example, a use case that has been [described](https://groups.google.com/d/msg/kubernetes-users/GPaGOGxCDD8/Ec4EPG5tAgAJ) is one namespace per deployment environment (dev, staging, prod), and then multiple microservice-based applications with their own administrators within each namespace. In that case you may want to give a permission like “only owners of application X can exec into application X pods” or “only owners of service Y can view the logs of the pods from the service” or to set aggregate quota per application. This is not possible today since the granularity for a Role and ResourceQuota is the namespace. But if we allow policies to specify label selectors, then they can apply their rules to subsets of a namespace, e.g. all the pods of a particular application. DNS names work as-is for this use case, since the client wants to find other applications in the same namespace

## Do namespaces matter for RBAC and general policies?

At this point one might wonder whether namespaces should play any part in RBAC and policies in general. Obviously we can’t get rid of namespaces, but if we force users to apply a “namespace” label to every object (or even create a virtual one from the actual namespace where the object is created), then we could stop using them for RBAC and general policies -- all rules would be “top-level” (non-namespaced) and select based on label selectors; applicability of a rule would always be determined by a label selector. The label selector might select all objects in multiple namespaces, all objects in one namespace, a subset of objects within one namespace, or even a subset of objects across multiple namespaces. Without the “namespace” label on every object, we can still get a similar effect but would have to separately select namespaces (via a label selector over labels on namespaces) and objects (via a label selector over labels on the objects). We could theoretically flatten hierarchical policies into this single level if we could come up with sensible rules for each type of policy (e.g. what is the meaning of multiple labels on the same pod for the same policy, due to policy at multiple levels of the hierarchy).

## Delegation and security

The issues in the previous section relate to the general topic of “delegated/organizational multi-tenancy” which means an administrative model with one or more cluster-level admins who control global policies, and then any number of tenants who can create namespaces and create rules that apply within and among the namespaces they created. It is already possible to set global rules and per-namespace rules, so this is really about how to set rules that span a subset of namespaces. Adding label selectors to RBAC clusterroles could address this.

But once labels, whether on namespaces or objects, are security-critical, it becomes important to control who can set them and to make sure the implications of different values are easily understood. For safety and sanity, it would probably be necessary to reserve certain labels (like tenant-id in previous examples) or a label namespace (e.g. xxx.kubernetes.io/\*) for security-critical labels. We could handle access control on these label values using specialized admission controllers with their own configurations. It’s less clear how to extend RBAC to handle it, short of something like adding an /applySecureLabel subresource to all objects, which would be the only way to change the set of secure labels on an object, and then setting an RBAC rule on that subresource. While that approach is more complicated, it is probably easier to audit.

Regardless of how we set policies, we need some kind of hierarchy so that, for example

* a cluster admin can override policies of organizational admins, and organizational admins can override policies of namespace admins
* organizational admins can refine policies set by cluster admins, and namespace admins can refine policies set by organizational admins (by “refine” we mean fill in more detail, for example narrowing the scope of a broad policy, or suballocating a portion of resource quota allocated to the higher level).

Alternatively we could have RBAC rules that limit who can set a particular kind of policy, such that only users at one level of the hierarchy would be allowed to set that particular kind of policy.

## Virtual clusters

In the discussion thus far (including all four columns of the chart presented earlier), we’ve assumed a system wherein tenants are aware that there are other tenants in the system, even though they can’t “see” them directly. For example, this is why the “create a namespace without worrying about name conflicts” row in the table says “no” even for the CaaS cases. We can provide an additional level of isolation by supporting the concept of a “virtual private cluster,” in which each tenant is given the illusion of their own completely private Kubernetes cluster even though it is shared with other tenants. (Note that this provides only slightly improved security; it’s mostly beneficial from a usability standpoint.) This has sometimes been called the “namespace of namespaces” model but there are many ways to implement it.

The virtual private cluster model is useful, for example, for someone who wants to sell Kubernetes as a Service without spinning up a separate cluster for each tenant. Note that this is not directly related to [self-hosting](https://buganizer.corp.google.com/issues/20688695), which is about running Kubernetes control plane components on Kubernetes. In self-hosting each user has an independent control plane; the components for those control planes just happen to be managed by (a separate) Kubernetes (cluster).

There are various resources that can be virtualized, with different degrees of difficulty and different benefits for the tenant:

* control plane: tenant can create arbitrary namespaces and will never be aware of what other namespaces other tenants might have created. This includes namespace for services and therefore we assume DNS names are virtualized as well.
* nodes, e.g. tenant can pick node name, set any taint or label on a node (except taints and labels reserved by the system at startup time), has access to full capacity/allocatable as exposed by the node, use arbitrary host ports, use hostPath, etc.
* L3 of the network, e.g. tenant can pick subnet for pods and services and arbitrary routing policies
* L2 of the network (create virtual L2 domain for a set of nodes)

Of course the implementation for any of these doesn’t *have* to involve virtualization, if we’re willing to sacrifice sharing. For example, “node” virtualization can be trivially achieved by giving tenants dedicated nodes, and DNS virtualization can be achieved by running a DNS per tenant. In the rest of this section we’ll assume sharing.

In terms of how to actually implement the virtual private cluster concept, we can look at each of the resources mentioned above:

* control plane
  + It’s presumably infeasible to add hierarchical namespaces to Kubernetes at this point, though that would be the cleanest solution to the problem. But there are other less elegant options, that meet the requirement of providing each tenant with a seemingly private single level of namespaces. There are two scenarios we need to handle:
    - Namespaced objects. The system could continue to store namespaced objects in namespaces, rewriting the namespace name on the way into and out of storage based on the tenant’s identity. For example, in etcd, namespace “foo” would be stored with the namespace name “${tenant-name}-foo.” This is obviously not trivial to implement and requires tight integration with the identity mechanism. But the fact that the namespace name has been rewritten can presumably be hidden from everything above the storage layer, minimizing changes to other parts of the system. Note that this includes the default namespace, which would be available read-write to every tenant as a separate copy.
    - Non-namespaced objects: These are more difficult to handle than namespaced objects. Storing them is not the problem; we could create “shadow” versions of these objects in a system namespace, prefixing the object name with the tenant name, or create a shadow namespace per tenant for storing these objects with their original names. The challenge is how the logic that uses these objects would be modified. How would we manage multiple Node objects per physical node, one per tenant? Or multiple clusterroles and clusterrolebindings? How would Events work? StorageClasses? ClusterResourceQuota? Logging and metrics? It is probably possible, but would require changes to layers of the system above storage.
  + DNS: If services and pods are virtualized in the API, then pods and services in DNS also need to be virtualized.
* node
  + It is hard to imagine virtualizing the node in Kubernetes itself. The only practical option here is a dedicated nodes per tenant. Of course the node might be a VM, as when running Kubernetes on cloud providers today.
* L3 of the network
  + Tenants can pick the subnet for their pods and services, and create arbitrary routing policies. They can create arbitrary application-specific networks, and services can be connected to one or more of these networks.
* L2 of the network
  + Isolated L2 domains, per tenant or even finer-grained. Or something like that.

## Addendum: Alternative Model

An alternative to trying to encode the tenancy hierarchy in the Kubernetes API objects, is to assume policies and hierarchy have been represented in an external directory system like LDAP or Active Directory, and implement authorization checks and policy checks using an admission controller that calls out to that other system. (Of course, we can also implement both models.)