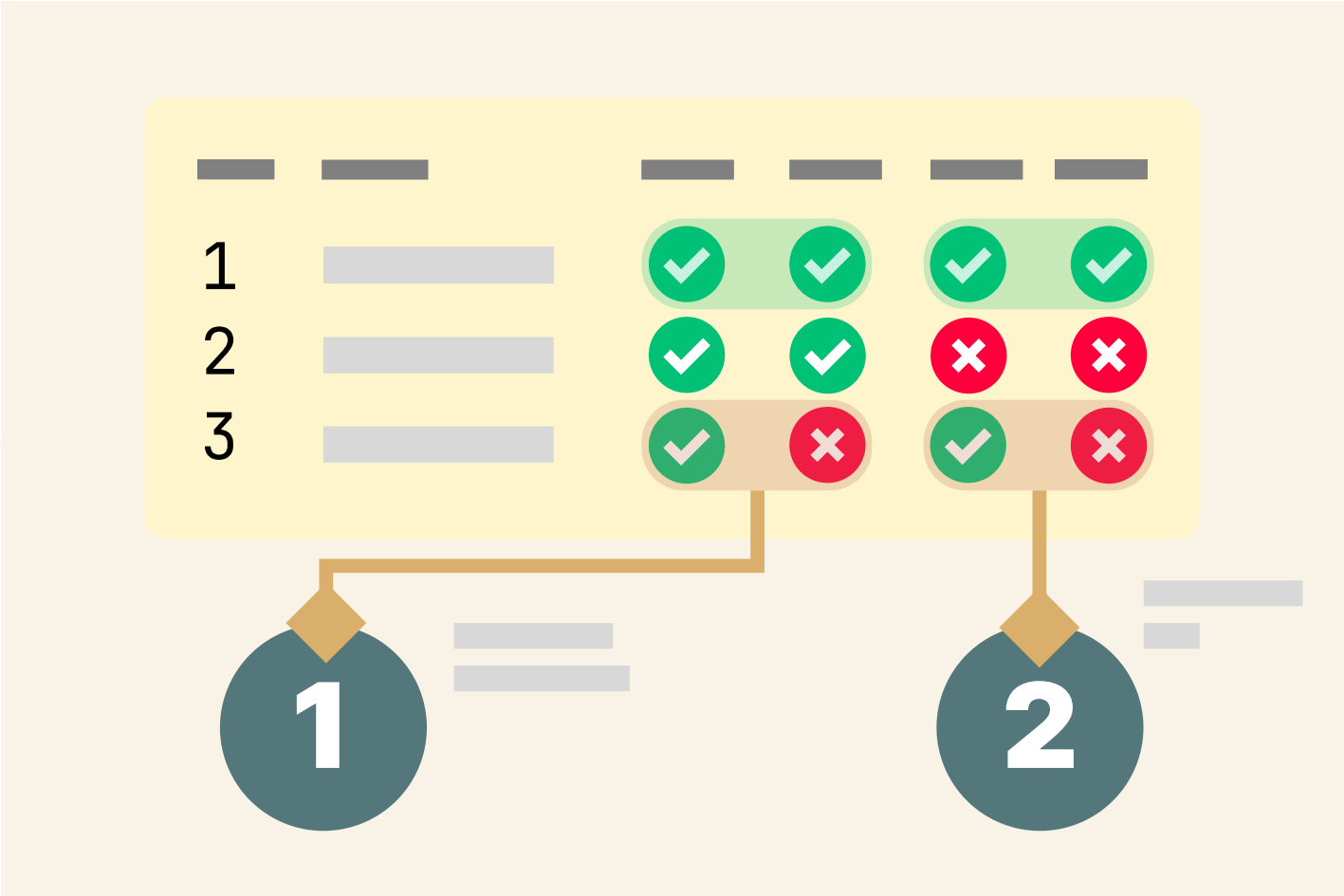
Arthur Chiao

Limiting access to Kubernetes resources with RBAC

MARCH 2022



hentication and authorization in Kubernetes series. More

TL;DR In this article, you will learn how to recreate the Kubernetes RBAC authorization model from scratch and practice the relationships between Roles, ClusterRoles, ServiceAccounts, RoleBindings and ClusterRoleBindings.

As the number of applications and actors increases in your cluster, you might want to review and restrict the actions they can take.

For example, you might want to restrict access to production systems to a handful of individuals.

Or you might want to grant a narrow set of permissions to an operator deployed in the cluster.

The Role-Based Access Control (RBAC) framework in Kubernetes allows you to do just that.

Table of contents

The Kubernetes API

Before discussing RBAC, let's see where the authorization model fits into the picture.

Let's imagine you wish to submit the following Pod to a Kubernetes cluster:

pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
```

```
nk8s/app:1.0.0
```

```
80
```

o the cluster with:

bash

```
yaml
```

When you type `kubectl apply`, a few things happen.

The kubectl binary:

1. Reads the configs from your `KUBECONFIG`.
2. Discovers APIs and objects from the API.
3. Validates the resource client-side *(is there any obvious error?)*.
4. Sends a request with the payload to the `kube-apiserver`.

When the `kube-apiserver` receives the request, it doesn't store it in etcd immediately.

First, it has to verify that the requester is legitimate.

In other words, it has to authenticate the request.

Once authenticated, does the requester have permission to create the resource?

Identity and permission are not the same things.

Just because you have access to the cluster doesn't mean you can create or

nly done with Role-Based Access Control

ontrol (RBAC), you can assign granular
at a user or app can do.

API server executes the following operations

st, authenticate the user.

n fails, reject the request by returning
`401 Unauthorized`.

    2. Otherwise, move on to the next stage.
2. The user is authenticated, but do they have access to the resource?
    1. If they don't, reject the request by returning `403 Forbidden .`
    2. Otherwise, continue.

In this article, you will focus on the authorization part.

Decoupling users and permission with RBAC roles

RBAC is a model designed to grant access to resources based on the roles of individual users within an organization.

To understand how that works, let's take a step back and imagine you had to design an authorization system from scratch.

How could you ensure that a user has write access to a particular resource?

uld involve writing a list with three columns like

```
Resource |
-------- |
  app1   |
  app2   |
  app2   |
```

- Bob has read & write access to `app1` but has no access to `app2`.
- Mo & Alice have only read access to `app2` and have no access to `app1`.

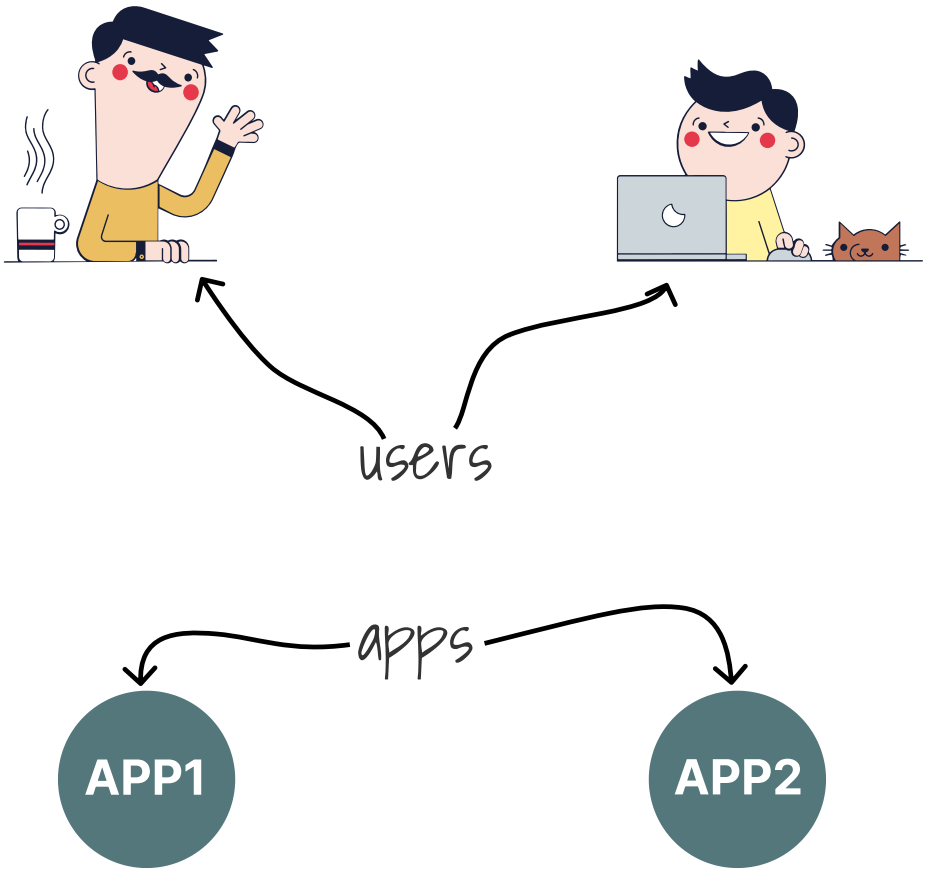
The table works well with a few users and resources but shows some limitations as soon as you start to scale it.

Let's imagine that Mo & Alice are in the same team, and they are granted read access to `app1`.

You will have to add the following entries to your table:

User	Permission	Resource
Bob	read+write	app1
Alice	read	app2
Mo	read	app2
Alice	read	app1
Mo	read	app1

That's great, but it is not evident that Alice and Mo have the same access e same team.



1/4

In a typical authorization system, you have users
sources.

NEXT >

ing a "Team" column to your table, but a better
the relationships:

eric container for permissions: **a role.**

permissions to users, you could include them in
eir role in the organisation.

ink roles to users.

nt.

able, now you have two:

1. In the first table, permissions are mapped to roles.
2. In the second table, roles are linked to identities.

Role	Permission	Resource
admin1	read+write	app1
reviewer	read	app2

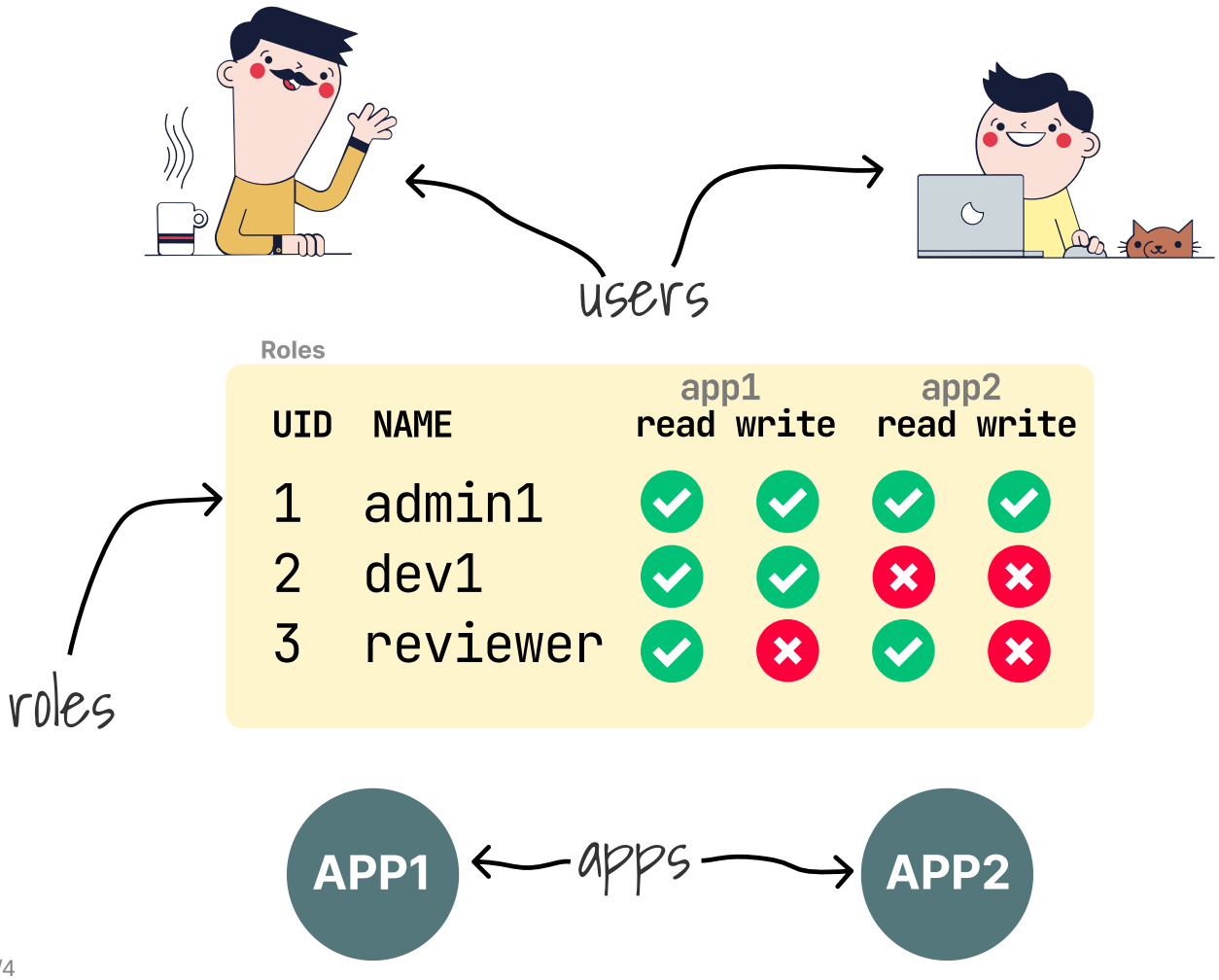
User	Roles
Bob	admin1
Alice	reviewer
Mo	reviewer

What happens when you want Mo to be an admin for app1?

You can add the role to the user like this:

```
        |
------ |
        |
r       |
min1    |
```

ow decoupling users from permissions with Roles
nistration in large organizations with many users



1/4

When using RBAC, you have users, resources and roles.

NEXT >

ernetes

RBAC model (as well as several other models)
he cluster.

me three concepts explained earlier: identities,

ly different names.

As an example, let's inspect the following YAML definition needed to grant access to Pods, Services, etc.:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: serviceaccount:app1
  namespace: demo-namespace
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: role:viewer
  namespace: demo-namespace
rules:          # Authorization rules for this role
  - apiGroups:  # 1st API group
      - ''      # An empty string designates the core API group.
    resources:
      - services
      - pods
    verbs:
      - get
```

```
  metadata:
    name: rolebinding:app1-viewer
    namespace: demo-namespace
  roleRef:
    apiGroup: rbac.authorization.k8s.io
    kind: Role
    name: role:viewer
  subjects:
    - kind: ServiceAccount
      name: serviceaccount:app1
      namespace: demo-namespace
```

The file is divided into three blocks:

1. **A Service Account** — this is the identity of who is accessing the resources.
2. **A Role** which includes the permission to access the resources.
3. **A RoleBinding** that links the identity (Service Account) to the permissions (Role).

After submitting the definition to the cluster, the application that uses the Service Account is allowed to issue requests to the following endpoints:


```
 resources
espace}/services
espace}/pods

ention provided by cilium.io
spaces/{namespace}/ciliumnetworkpolicies
spaces/{namespace}/ciliumnetworkpolicies/status
```

a lot of details that we've glossed over.

What resources are you granting access to, exactly?

What is a Service Account? Aren't the identities just "Users" in the cluster?

Why does the Role contain a list of Kubernetes objects?

To understand how those work, let's set aside the Kubernetes RBAC model and try to rebuild it from scratch.

We will focus on three elements:

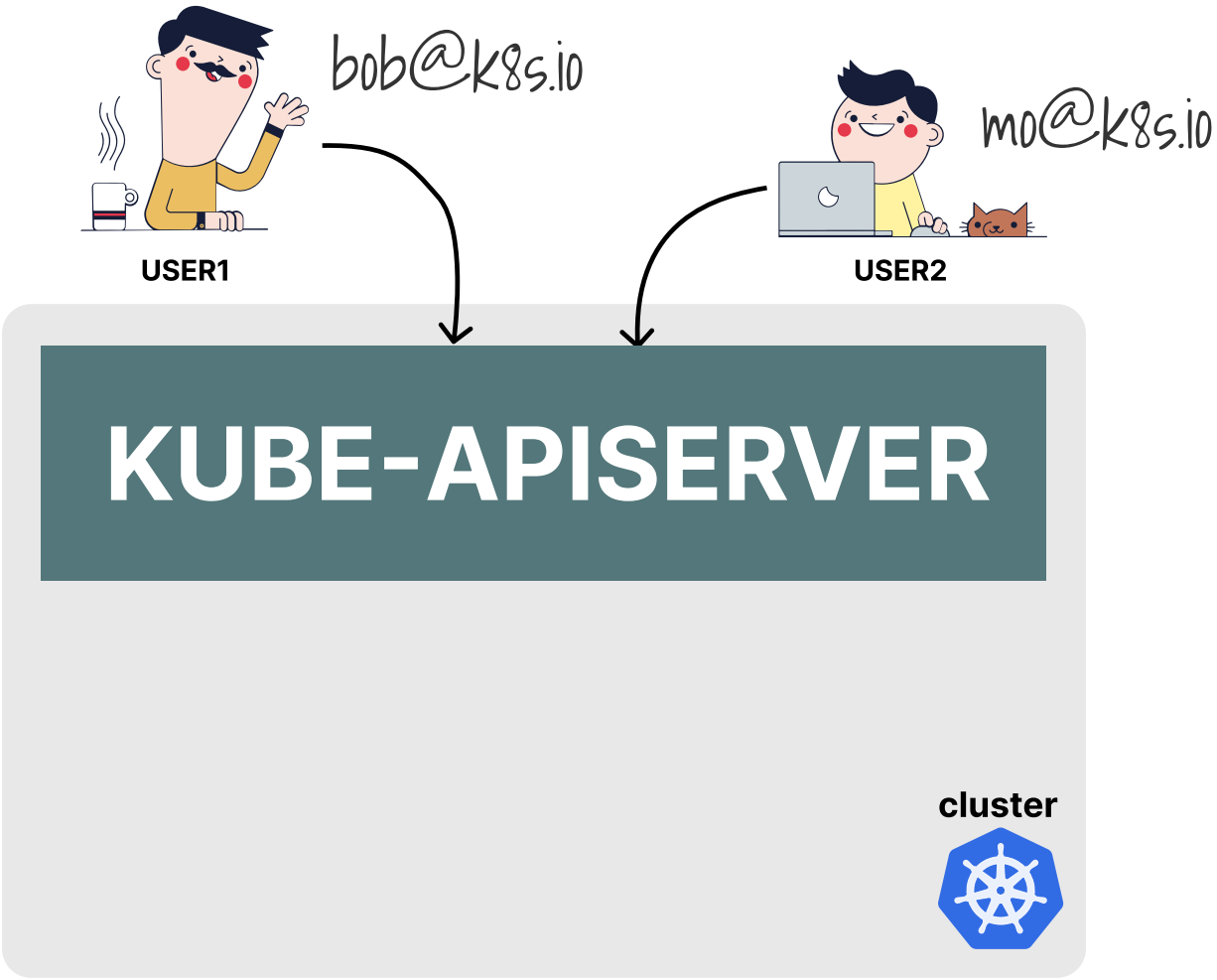
1. Identifying and assigning identities.
2. Granting permissions.
3. Linking identities to permissions.

Let's start.

Assigning identities: humans, bots and groups

ue wishes to log in to the Kubernetes dashboard.

ve an entity for an "account" or a "user", with each me or ID (such as the email address).



User in the cluster?

e objects which represent regular user

a cluster through an API call.

sents a valid certificate signed by the cluster's
considered authenticated.

rnetes assigns the username from the common
ect' of the certificate (e.g., "/CN=bob").

A temporary User info object is created and passed to the authorization (RBAC) module.

Digging into the code reveals that a struct maps all of the details collected from the Authentication module.

```
type User struct {
    name string   // unique for each user
    ...           // other fields
}
```

Note that the `User` is used for **human or processes outside the cluster**.

If you want to identify a process in the cluster, you should use a Service Account instead.

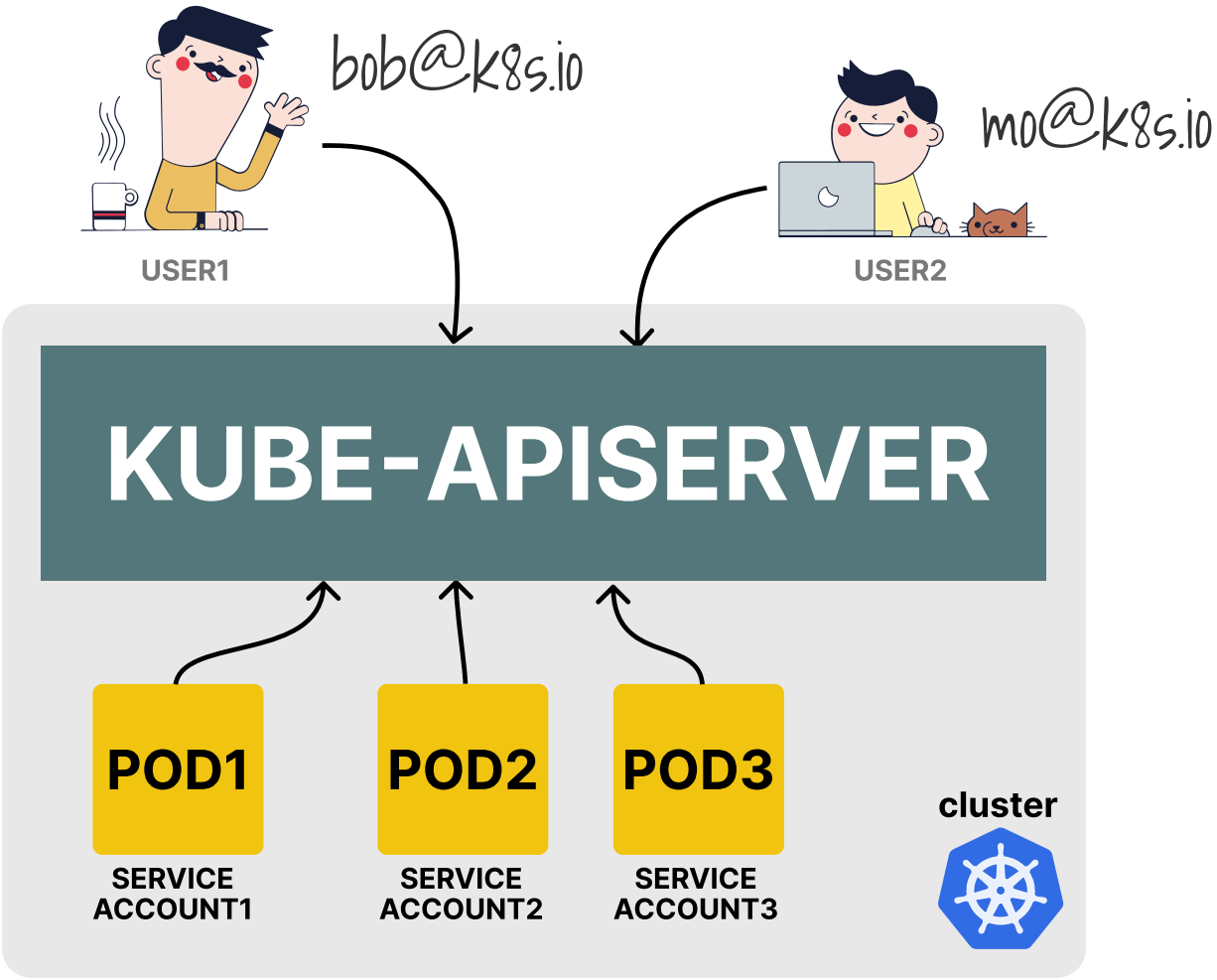
The account is very similar to a regular user, but it's different because Kubernetes manages it.

A Service Account is usually assigned to pods to grant permissions.

For example, we have the following applications accessing resources

- `kubectl` wants to list all pod resources on a specific node.
- The `kube-controller` has to list all the backend endpoints for a

To do so, we define a ServiceAccount (SA).



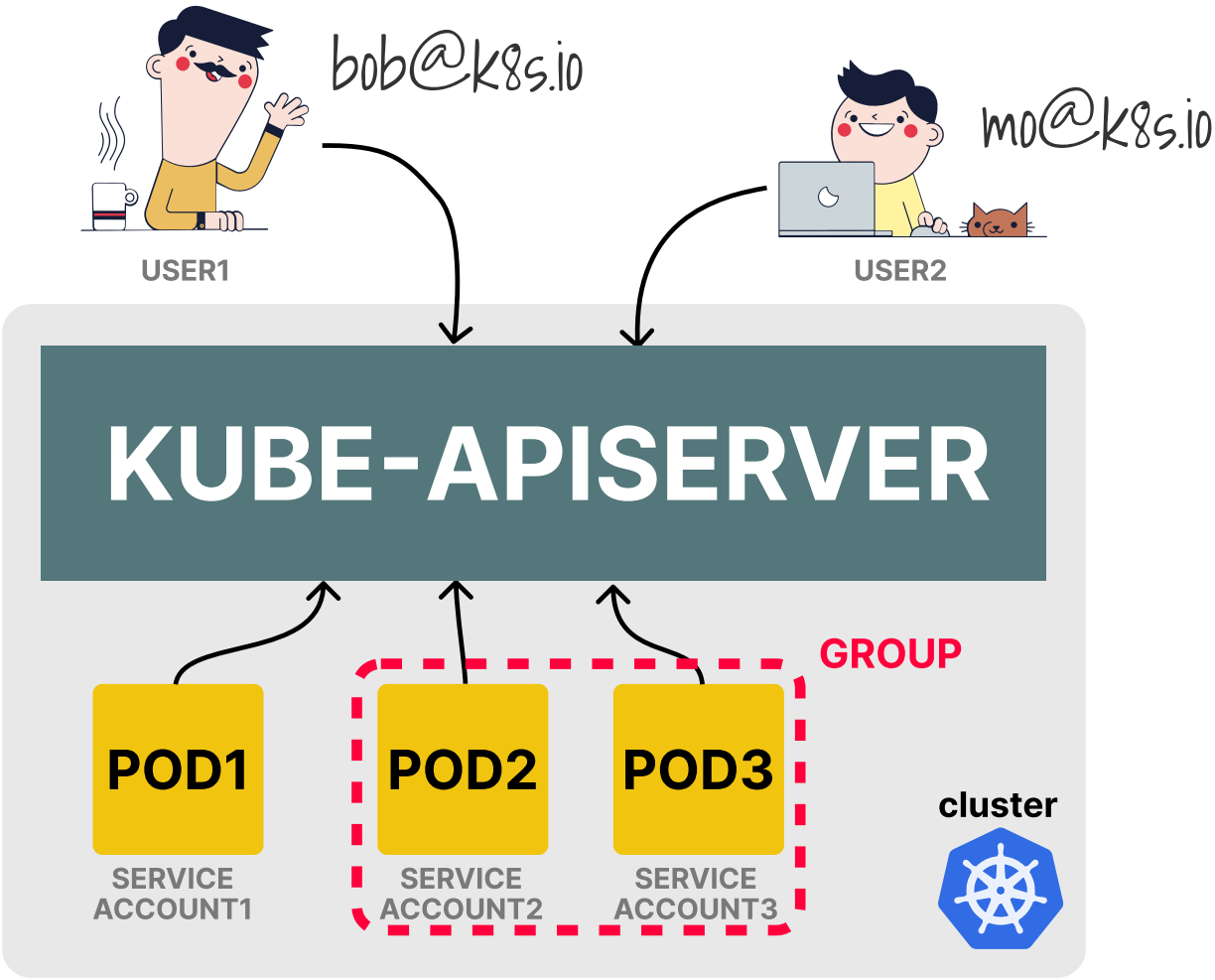
e managed in the cluster, you can create them

service-account.yaml

```
    # arbitrary but unique string
pace
```

ministration, you could also define a group of

s.



sh to reference **all ServiceAccounts in a**

how to access the resources, it's time to discuss

echanism to identify who has access to

or a group of them.

y accessing in the cluster?

Modelling access to resources

In Kubernetes, we are interested in controlling access to resources such as Pods, Services, Endpoints, etc.

Those resources are usually stored in the database (etcd) and accessed via built-in APIs such as:

```
/api/v1/namespaces/{namespace}/pods/{name}
/api/v1/namespaces/{namespace}/pods/{name}/log
/api/v1/namespaces/{namespace}/serviceaccounts/{name}
```

The best way to limit access to those resources is to control how those API endpoints are requested.

You will need two things for that:

e resource.

granted to access the resource (e.g. read-only,

ill use a verb such as `get`, `list`, `create`,

`et`, `list` and `watch` Pods, logs and Services.

esources and permission in a list like this:

```
  resources:
    - /api/v1/namespaces/{namespace}/pods/{name}
    - /api/v1/namespaces/{namespace}/pods/{name}/log
    - /api/v1/namespaces/{namespace}/serviceaccounts/{name}
  verbs:
    - get
    - list
    - watch
```

You could simplify the definition and make it more concise if you notice that:

- The base URL `/api/v1/namespaces/` is common for all. Perhaps we could omit it.
- You could assume that all resources are in the current namespace and drop the `{namespace}` path.

That leads to:

```
  resources:
    - pods
```


ndly, and you can immediately identify what's

Besides APIs for built-in objects such as pods, endpoints, services, etc., Kubernetes also supports API extensions.

For example, when using install the Cilium CNI, the script creates a `CiliumEndpoint` custom resource (CR):

cilium-endpoint.yaml

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: ciliumendpoints.cilium.io
spec:
  group: cilium.io
  names:
    kind: CiliumEndpoint
  scope: Namespaced
  # truncated...
```

Those objects are stored in the cluster and are available through kubectl:

bash

```
points.cilium.io -n demo-namespace
ENTITY ENDPOINT STATE   IPV4

28124  ready            10.6.7.54
24494  ready            10.6.7.94
75701  ready            10.6.4.24
```

be similarly accessed via the Kubernetes API:

```
/apis/cilium.io/v2/namespaces/{namespace}/ciliumendpoints
/apis/cilium.io/v2/namespaces/{namespace}/ciliumendpoints/{name}
```

If you want to map those into a YAML file, you could write the following:

```
resources:
  - ciliumnetworkpolicies
  - ciliumnetworkpolicies/status
verbs:
  - get
```

However, how does Kubernetes know that the resources are custom?

How can it differentiate between APIs that use custom resources and built-in?

Unfortunately, dropping the base URL from the API endpoint wasn't such a good idea.

slight change.

p and use it later to expand the URL for the

```
Group name

es
es/status
```

What about resources such as Pods that don't have a namespaced API?

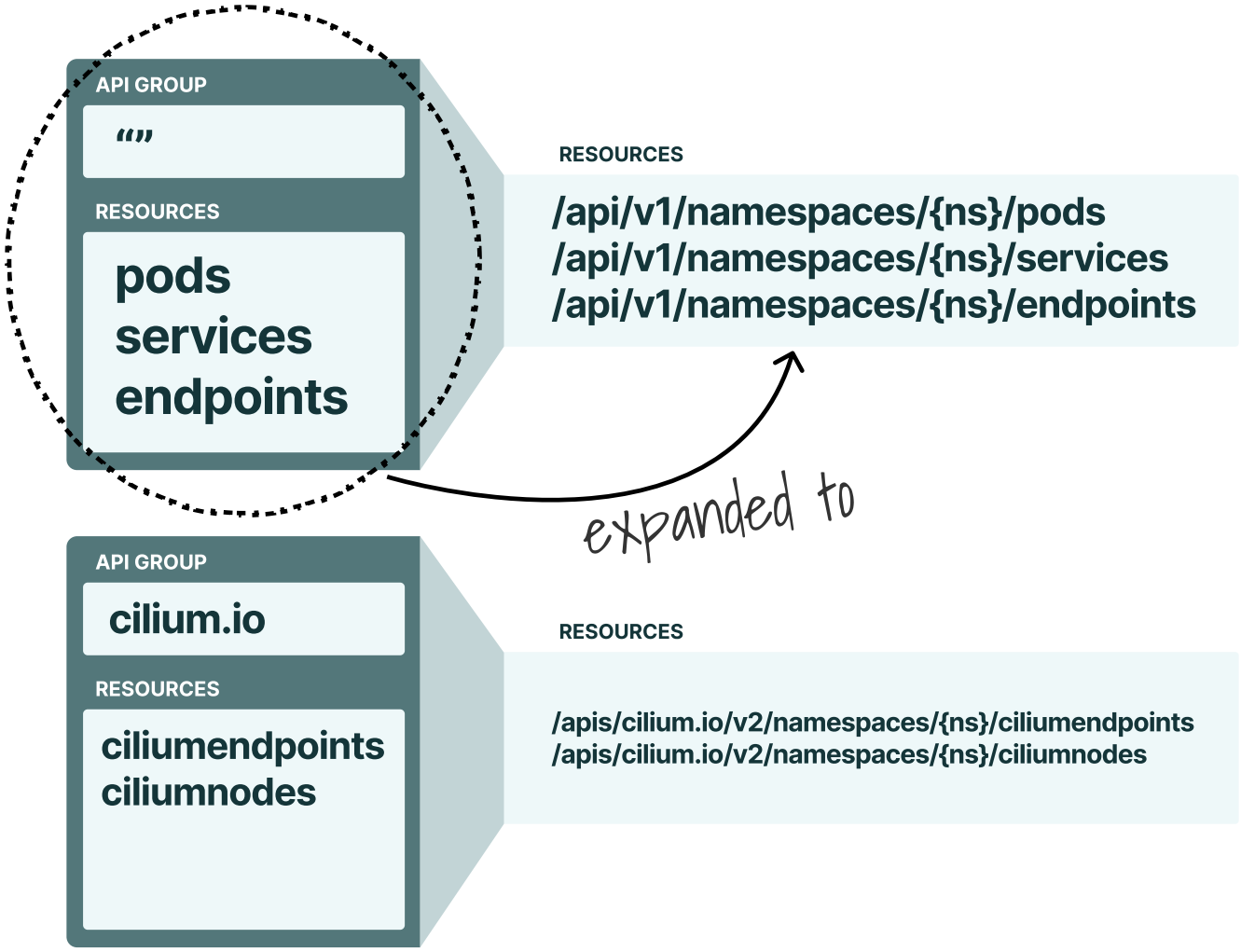
The Kubernetes `""` empty API group is a special group that refers to the built-in objects.

So the previous definition should be expanded to:

```
apiGroups:
  - '' # Built-in objects
resources:
  - pods
  - pods/logs
  - serviceaccounts
verbs:
  - get
  - list
  - watch
```

Kubernetes reads the API group and automatically expands them to:

```
pi/v1/xxx .
group_name}/{apigroup_version}/xxx .
```

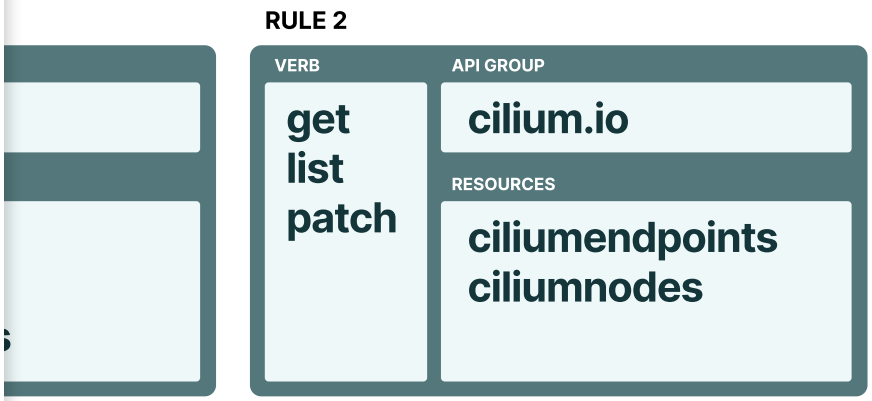


map resources and permissions, it's finally time to
ources together.

of resources and verbs is called a `Rule`, and
ist:

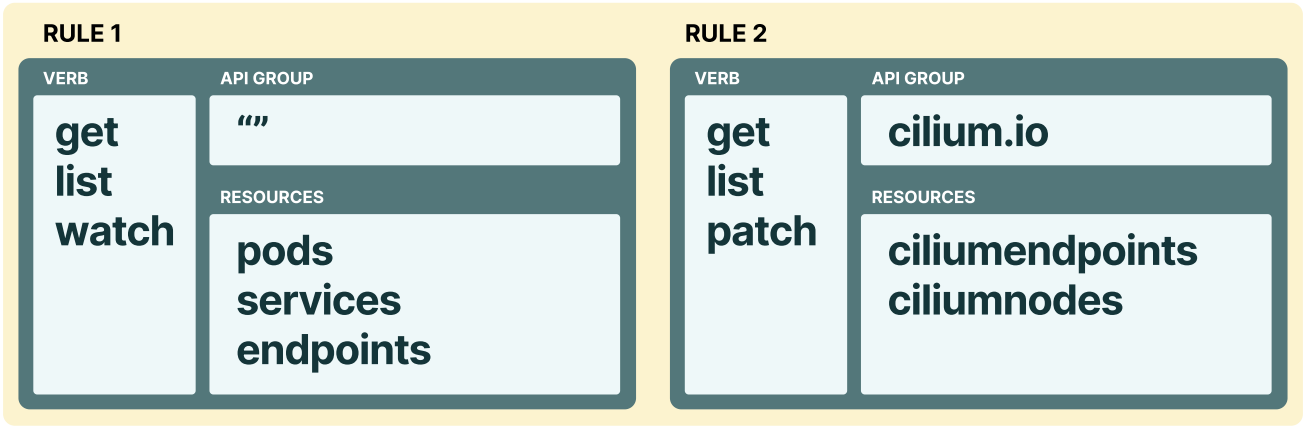
Each rule contains the `apiGroups`, `resources` and `verbs` that you just learned:

```
rules: # Authorization rules
  - apiGroups: # 1st API group
      - '' # An empty string designates the core API group.
    resources:
      - pods
      - pods/logs
      - serviceaccounts
    verbs:
      - get
      - list
      - watch
  - apiGroups: # another API group
      - cilium.io # Custom APIGroup
    resources:
      - ciliumnetworkpolicies
      - ciliumnetworkpolicies/status
    verbs:
      - get
```

A collection of rules has a specific name in Kubernetes, and it is called a Role.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: viewer
rules: # Authorization rules
  - apiGroups: # 1st API group
      - '' # An empty string designates the core API group.
    resources:
      - pods
      - pods/logs
      - serviceaccounts
    verbs:
      - get
      - list
      - watch
  - apiGroups: # another API group
      - cilium.io # Custom APIGroup
    resources:
      - ciliumnetworkpolicies
                              licies/status
```



Excellent!

So far, you modelled:

- Identities with Users, Service Accounts and Groups.
- Permissions to resources with Roles.

The missing part is linking the two.

issions to users

permissions defined in a Role to a User,
o.

mple:

```
ization.k8s.io/v1
kind: RoleBinding
```

```
  metadata:
    name: role-binding-for-app1
  roleRef:
    apiGroup: rbac.authorization.k8s.io
    kind: Role
    name: viewer
  subjects:
    - kind: ServiceAccount
      name: sa-for-app1
      namespace: kube-system
```

The definition has two important fields:

- the `roleRef` that references the `viewer` Role.
- the `subjects` links to the `sa-for-app1` Service Account.

As soon as you submit the resource to the cluster, the application or user using the Service Account will have access to the resources listed in the Role.

If you remove the binding, the app or user will lose access to those resources (but the Role will stay ready to be used by other bindings).

ld is a list that contains `kind`, `name` and

ssary to identify Users from Service Accounts

?

e cluster up into namespaces and limit access to specific accounts.

RoleBindings are placed inside and grant espace.

However, it is possible to mix these two types of resources — you will see how later.

Before we wrap up the theory and start with the practice, let's have a look at a few examples for the `subjects` field:

```
subjects:
  - kind: Group
    name: system:serviceaccounts
    apiGroup: rbac.authorization.k8s.io
    # when the namespace field is not specified, this targets all servic
```

You can also have multiple Groups, Users or Service Accounts as `subjects`:

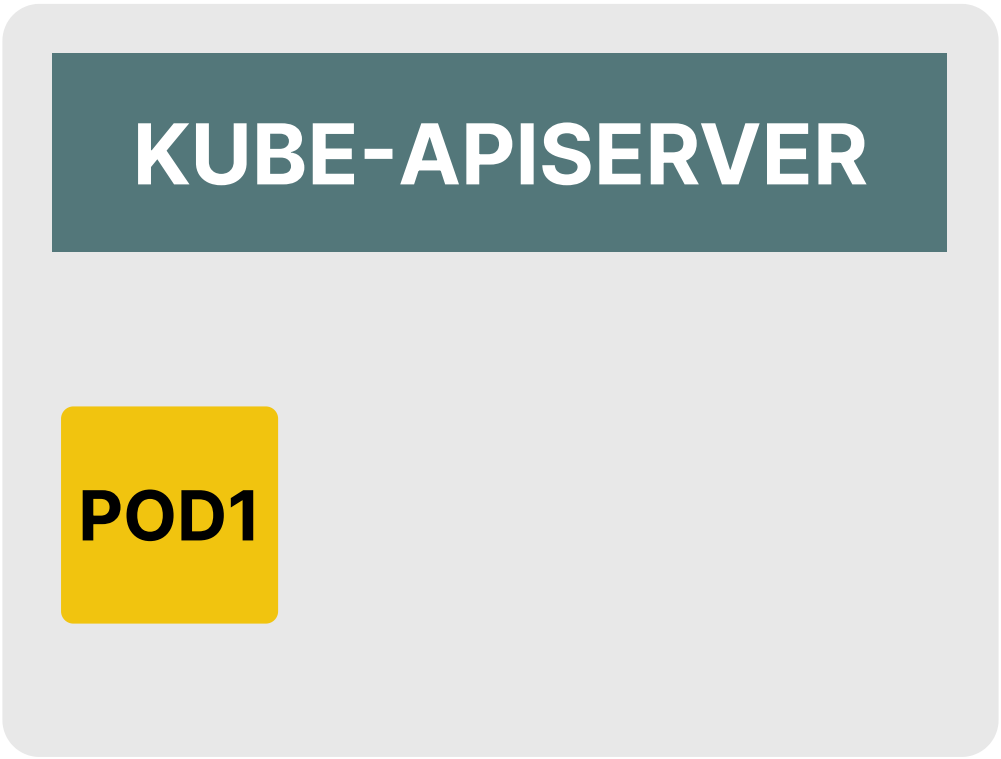
```
subjects:
  - kind: Group
```

```
ticated # for all authenticated users
orization.k8s.io

enticated # for all unauthenticated users
orization.k8s.io
```

ed so far, let's look at how to grant permissions custom resources.

enge: you have an app that needs access to the m.



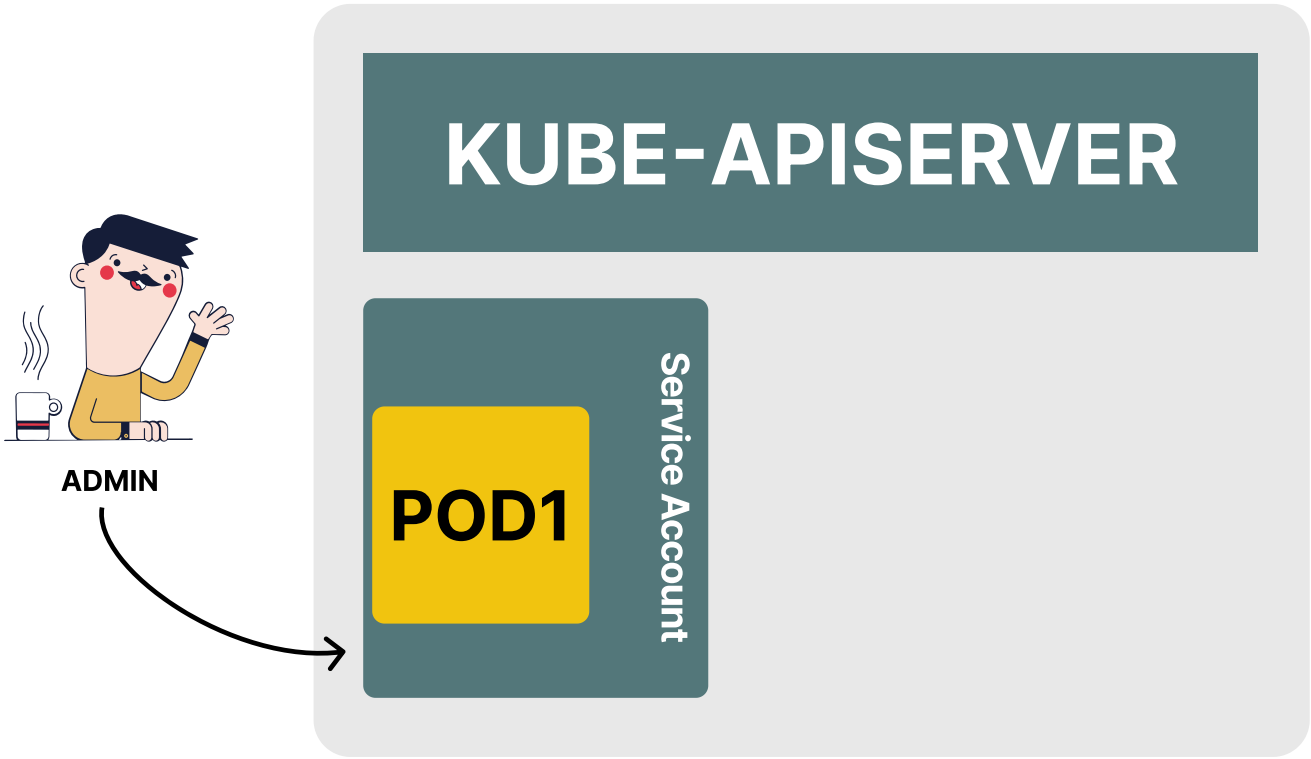
1/2

NEXT >

Imagine having an app deployed in the cluster that needs
Custom Resource through the API.

sions to access those resources?

ole and RoleBinding.



NEXT >

First, you should create an identity for your workload. In
hat means creating a Service Account.

and cluster-wide resources

sources, you learned that the structure of the

```
/api/v1/namespaces/{namespace}/pods/{name}
/api/v1/namespaces/{namespace}/pods/{name}/log
/api/v1/namespaces/{namespace}/serviceaccounts/{name}
```

But what about resources that don't have a namespace, such as Persistent Volumes and Nodes?

Namespaced resources can only be created within a namespace, and the name of that namespace is included in the HTTP path.

If the resource is global, like in the case of a Node, the namespaces name is not present in the HTTP path.

```
/api/v1/nodes/{name}
/api/v1/persistentvolume/{name}
```

Can you add those to a Role?

You can

s any namespace limitation when Roles and
ced.

```
ization.k8s.io/v1
```

```
rules
I group
```

```
    - '' # An empty string designates the core API group.
  resources:
    - persistentvolumes
    - nodes
  verbs:
    - get
    - list
    - watch
```

If you try to submit that definition and link it to a Service Account, you might realize it doesn't work, though.

Persistent Volumes and Nodes are cluster-scoped resources.

However, Roles can grant access to scoped resources to a namespace.

If you'd like to use a Role that applies to the entire cluster, you can use a `ClusterRole` (and the corresponding `ClusterRoleBinding` to assign it a subject).

The previous definition should be changed to:


```
ization.k8s.io/v1


rization rules
PI group
pty string designates the core API group.

es
```

```
    - list
    - watch
```

Notice how the only change is the `kind` property, and everything else stays the same.

You can use ClusterRoles to grant permissions to all resources — for example, all Pods in the cluster.

This functionality isn't restricted to cluster-scoped resources.

Kubernetes ships with a few Roles and ClusterRoles already.

Let's explore them.

bash

```
$ kubectl get roles -n kube-system | grep "^system:"
NAME
system::leader-locking-kube-controller-manager
system::leader-locking-kube-scheduler
system:controller:bootstrap-signer
-provider
-cleaner
```


d to denote that the resource is directly managed
.

lusterRoles and ClusterRoleBindings are labelled
`rapping=rbac-defaults .`

les with:

bash

```
$ kubectl get clusterroles -n kube-system | grep "^system:"
NAME
system:aggregate-to-admin
system:aggregate-to-edit
system:aggregate-to-view
system:discovery
system:kube-apiserver
system:kube-controller-manager
system:kube-dns
system:kube-scheduler
# truncated output...

$ _
```

You can inspect the details for each Role and ClusterRole with:

bash

```
$ kubectl get role <role> -n kube-system -o yaml
# or
$ kubectl get clusterrole <clusterrole> -n kube-system -o yaml _
```


basic building blocks of Kubernetes RBAC.

s with Users, Service Accounts and groups.

ions to resources in a namespace with a Role.

ions to cluster resources with a ClusterRole.

ClusterRoles to subjects.

There's only one missing topic left to explore: a few unusual edge cases of RBAC.

Making sense of Roles, RoleBindings, ClusterRoles, and ClusterBindings

At a high level, Roles and RoleBindings are placed inside and grant access to a specific namespace, while ClusterRoles and ClusterRoleBindings do not belong to a namespace and grant access across the entire cluster.

However, it is possible to mix these two types of resources.

For example, what happens when a RoleBinding links an account to a ClusterRole?

Let's explore this next with some hands-on practice.

Let's start by creating a local cluster with minikube:


```
bash
```

```
ted the docker driver
ane node in cluster
...
tainer (CPUs=2, Memory=4096MB) ...
s v1.22.3 on Docker 20.10.8 ...
icates and keys ...
l plane ...
rules ...
s components...
o/k8s-minikube/storage-provisioner:v5
```

```
🌟  Enabled addons: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use the cluster and "default" nam

$ _
```

To start, create four namespaces:

bash

```
$ kubectl create namespace test
namespace/test created
$ kubectl create namespace test2
namespace/test2 created
$ kubectl create namespace test3
namespace/test3 created
$ kubectl create namespace test4
namespace/test4 created

$ _
```

And finally, create a Service Account in the `test` namespace:

service-account.yaml

e with:

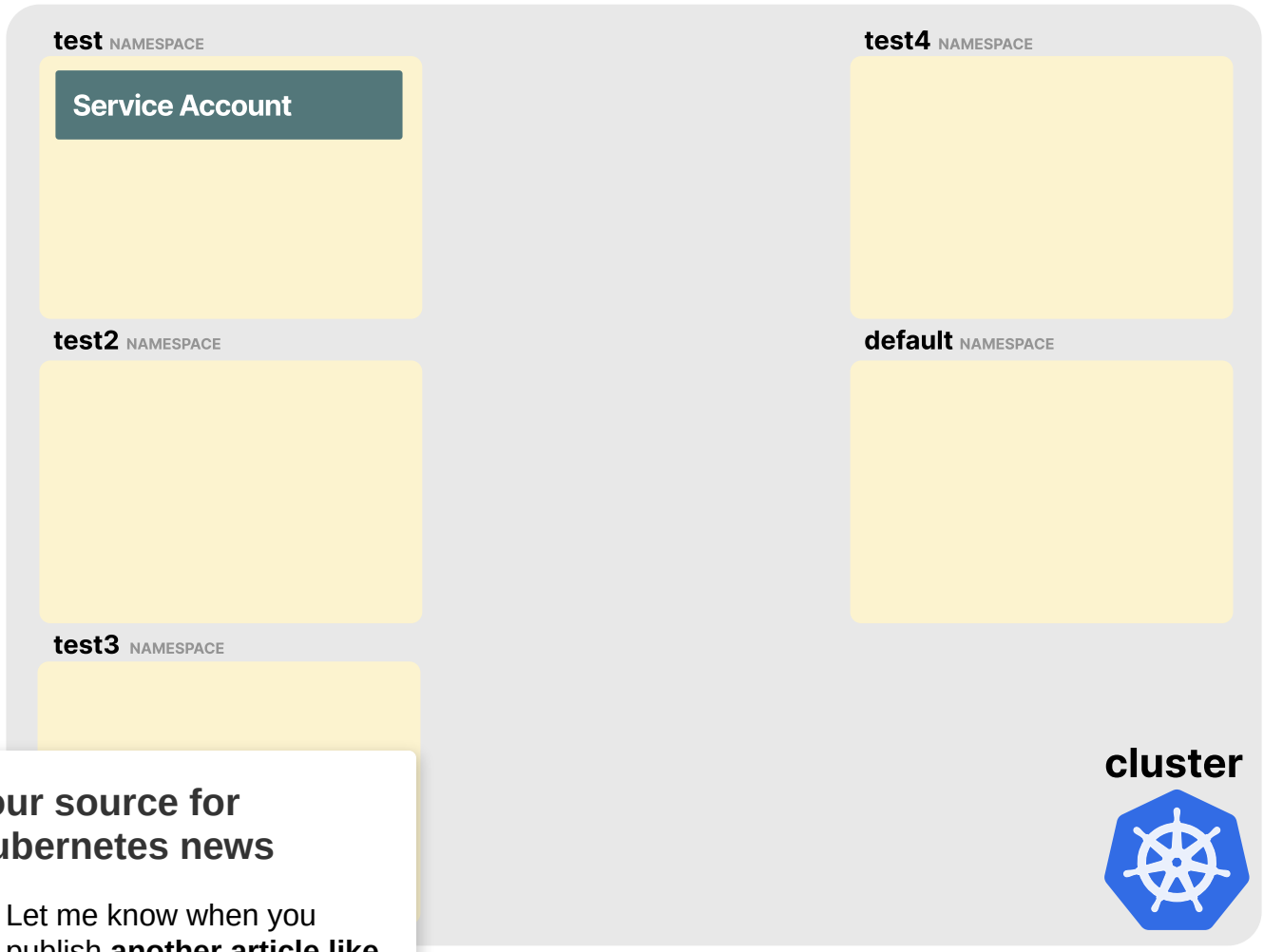
bash

```
$ kubectl apply -f service-account.yaml
```

```
serviceaccount/myaccount created

$ _
```

At this point, your cluster should look like this:



ole and RoleBinding in the ace

Let's start with creating a Role and a RoleBinding to grant the Service Account access to the `test` namespace:

scenario1.yaml

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: testadmin
  namespace: test
rules:
  - apiGroups: ['*']
    resources: ['*']
    verbs: ['*']
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: testadminbinding
  namespace: test
subjects:
  - kind: ServiceAccount
    name: myaccount
    namespace: test
```


`ization.k8s.io`

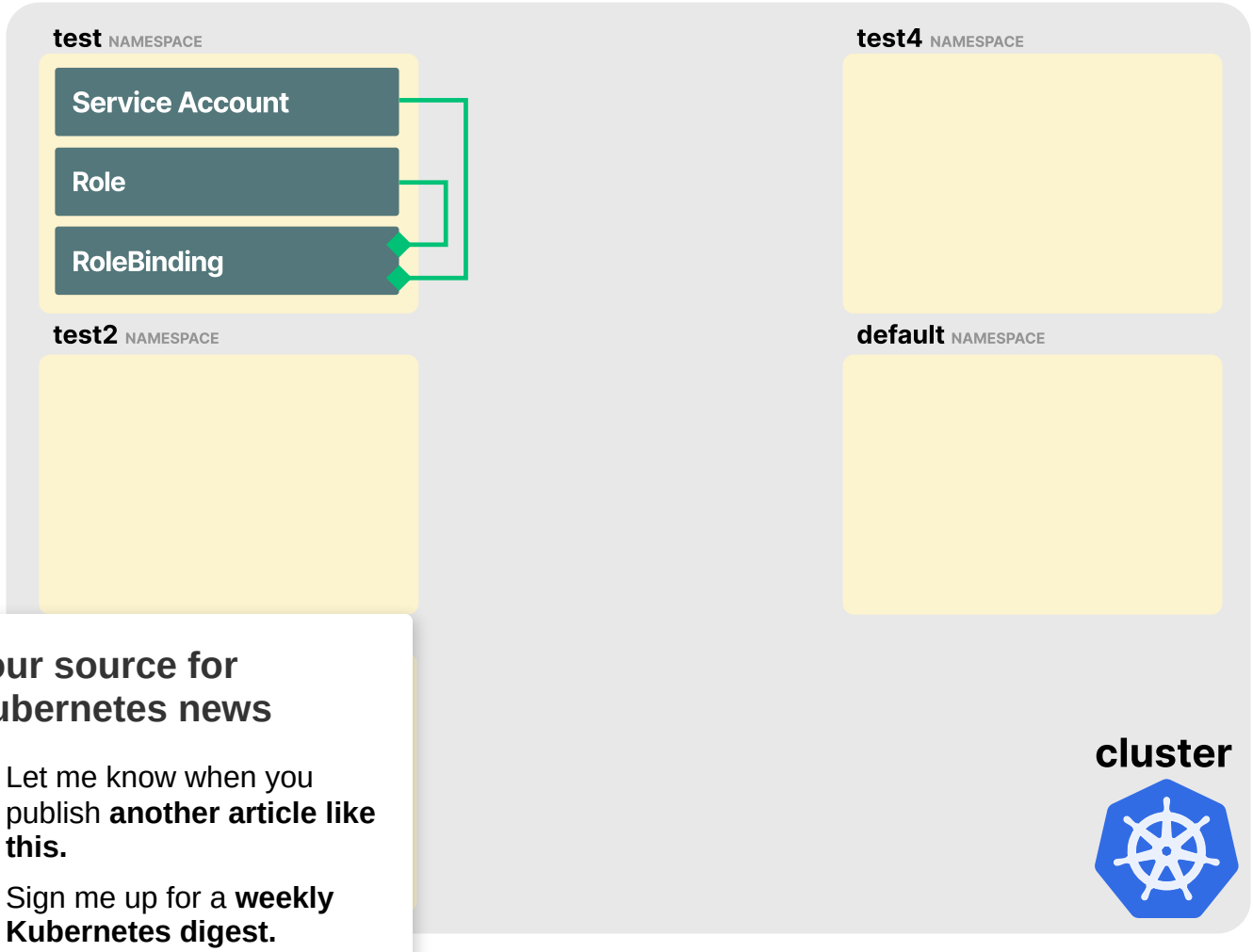
`e` with:

bash

```
$ kubectl apply -f scenario1.yaml
role.rbac.authorization.k8s.io/testadmin created
rolebinding.rbac.authorization.k8s.io/testadminbinding created

$ _
```

Your cluster looks like this:



Account, Role, and RoleBinding) are in the `test`

all resources, and the RoleBinding links the

ole.

How do you test that the Service Account has access to the resources?

You can combine two features of kubectl:

1. User-impersonation with `kubectl <verb> <resource> --as=jenkins .`
2. Verifying API access with `kubectl auth can-i <verb> <resource> .`

> Please note that your user should have the `impersonate` verb as permission for this to work.

To issue a request as the `myaccount` Service Account and check if you can list Pod in the namespace, you can issue the following command:

bash

```
$ kubectl auth can-i get pods -n test --as=system:serviceaccount:test:my
yes

$ _
```


…and:

…ary to query the authorization model (RBAC).

… and `resource .`

…space where you want to issue the command.

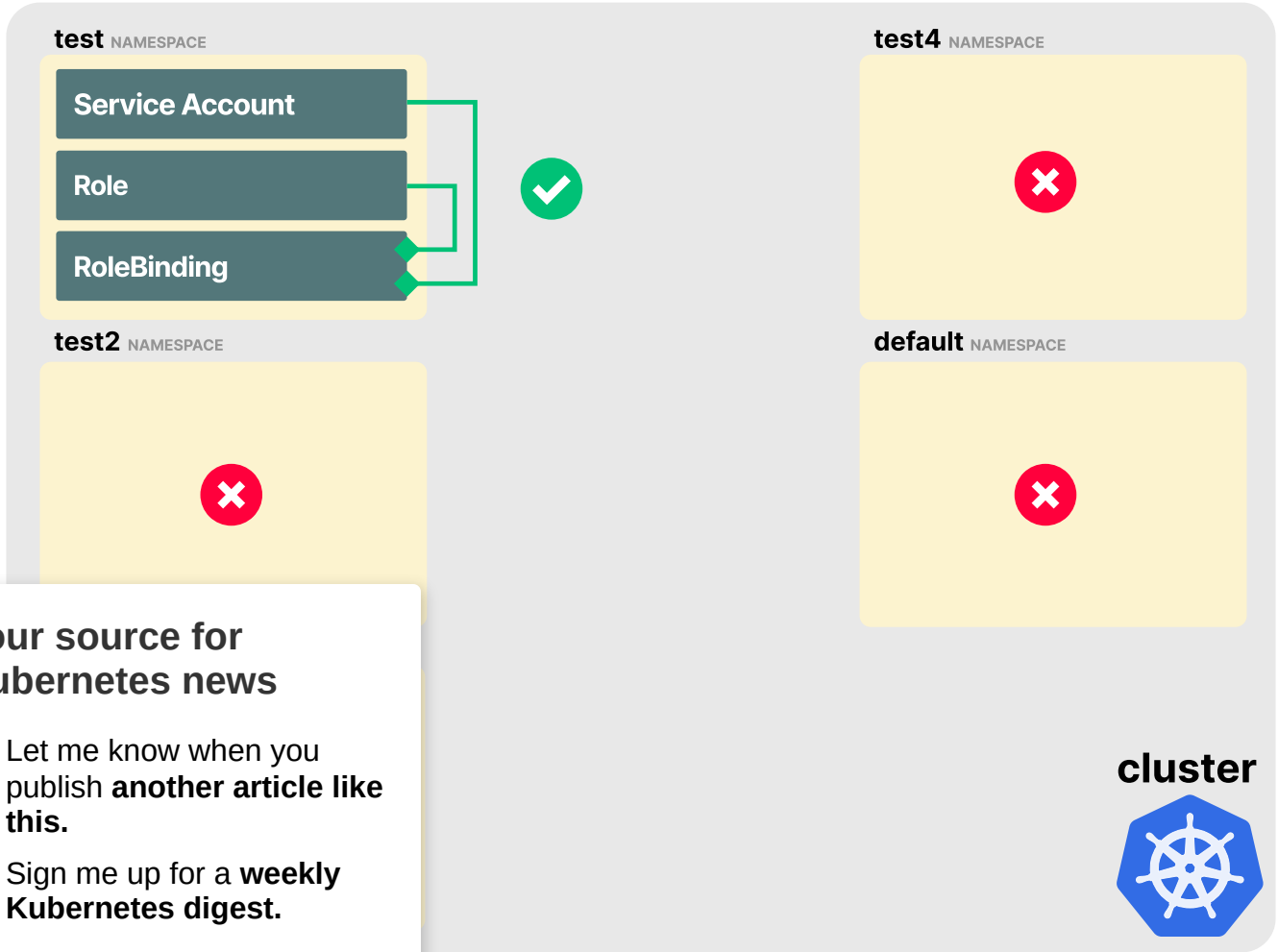
…`ccount:test:myaccount` is used to impersonate the …count.

…eeds some extra hints to identify the Service

…oken down to:

```
--as=system:serviceaccount:{namespace}:{service-account-name}
     ^^^^^^^^^^^^^^^^^^^^^
     This should always be included for Service Accounts.
```

With this Role+ServiceAccount+RoleBindings combination, you can access all resources in the `test` namespace.



mplex example.

Scenario 2: Role and RoleBinding in a different namespace

Let's create a new Role and RoleBinding in the `test2` namespace.

Notice how the RoleBinding links the role from `test2` and the service account from `test`:

scenario2.yaml

```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: test2
  name: testadmin
rules:
  - apiGroups: ['*']
    resources: ['*']
    verbs: ['*']
```

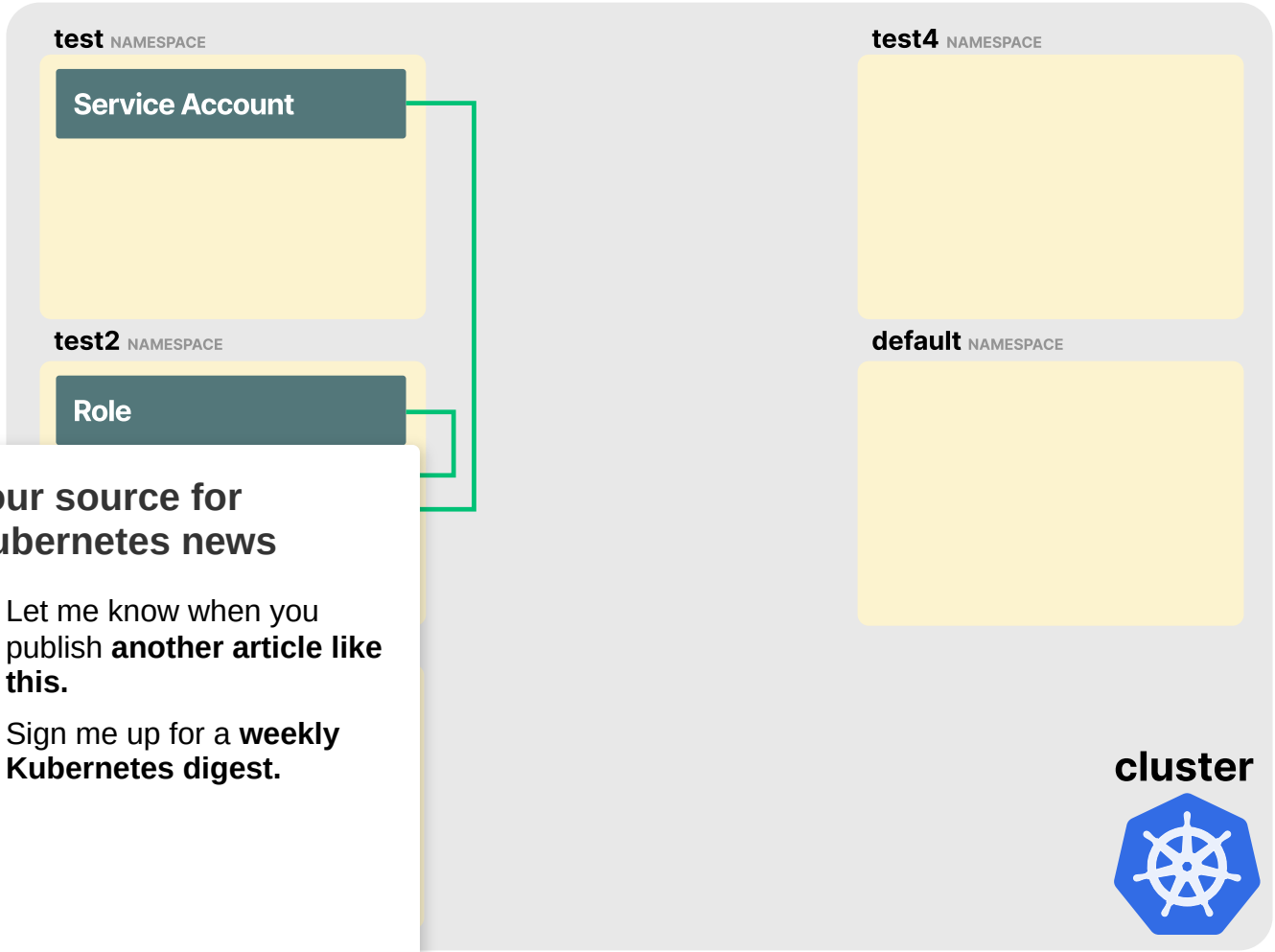
You can submit the resource with:

bash

```
$ kubectl apply -f scenario2.yaml
role.rbac.authorization.k8s.io/testadmin created
rolebinding.rbac.authorization.k8s.io/testadminbinding created

$ _
```

Your cluster looks like this:



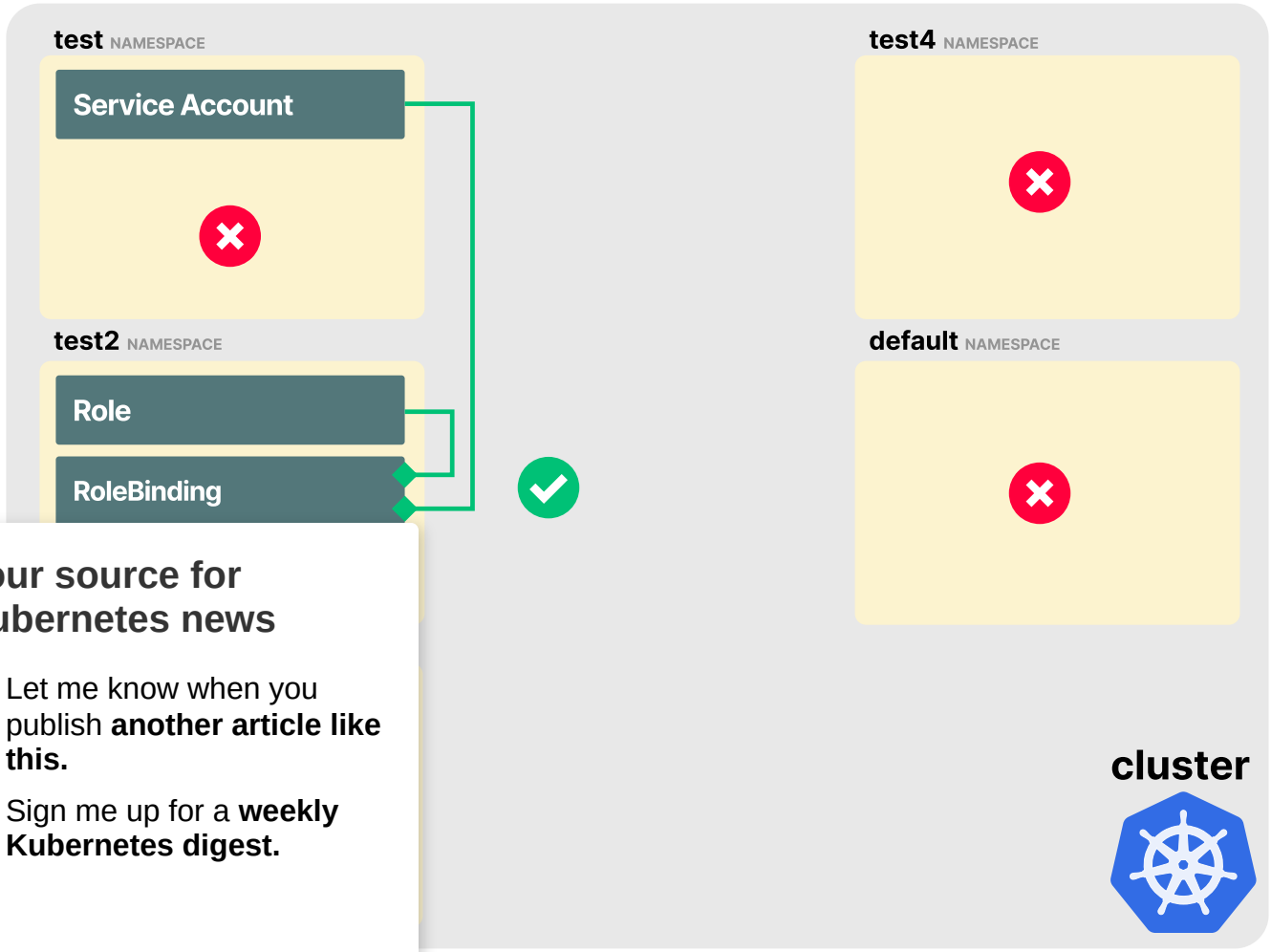
ount located in `test` has access to the

bash

```
$ kubectl auth can-i get pods -n test2 --as=system:serviceaccount:test:m
yes

$ _
```

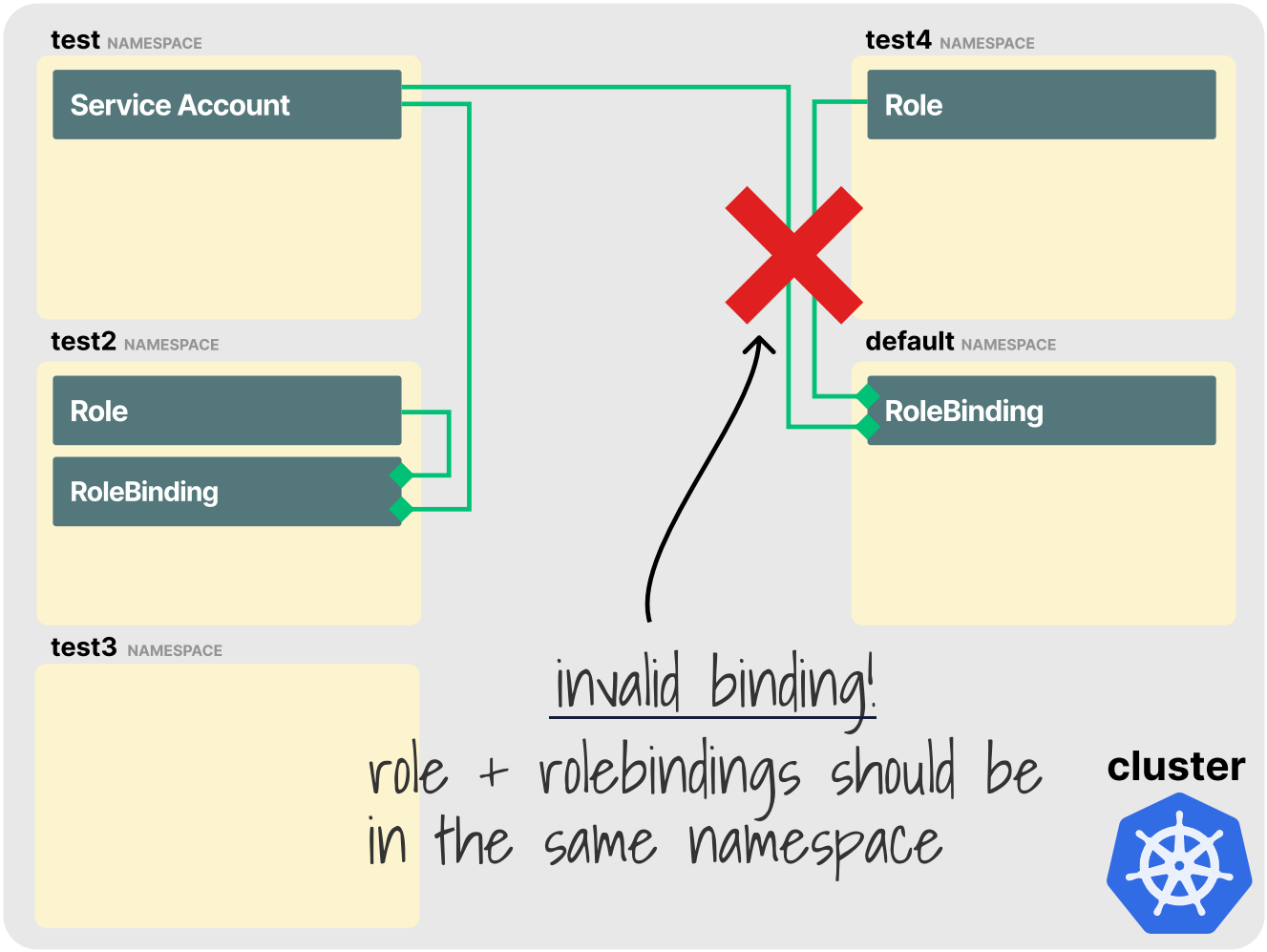
This works, granting the Service Account access to resources outside of the namespace it was created.



`oleRef` property in the RoleBinding does not have

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: testadminbinding
  namespace: test2
subjects:
  - kind: ServiceAccount
    name: myaccount
    namespace: test
roleRef:
  kind: Role
  name: testadmin
  apiGroup: rbac.authorization.k8s.io
```

The implication is that a RoleBinding can only reference a Role in the same namespace.



sing a ClusterRole with a

les do not belong to a namespace.

e does not scope permissions to a single

ole is linked to a Service Account via a

le permissions only apply to the namespace in

which the RoleBinding was created.

Let's have a look at an example.

Create a RoleBinding in namespace `test3` and link the Service Account to the ClusterRole `cluster-admin`:

> `cluster-admin` is one of those built-in ClusterRoles in Kubernetes.

scenario3.yaml

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: testadminbinding
  namespace: test3
subjects:
  - kind: ServiceAccount
    name: myaccount
    namespace: test
roleRef:
  kind: ClusterRole
```



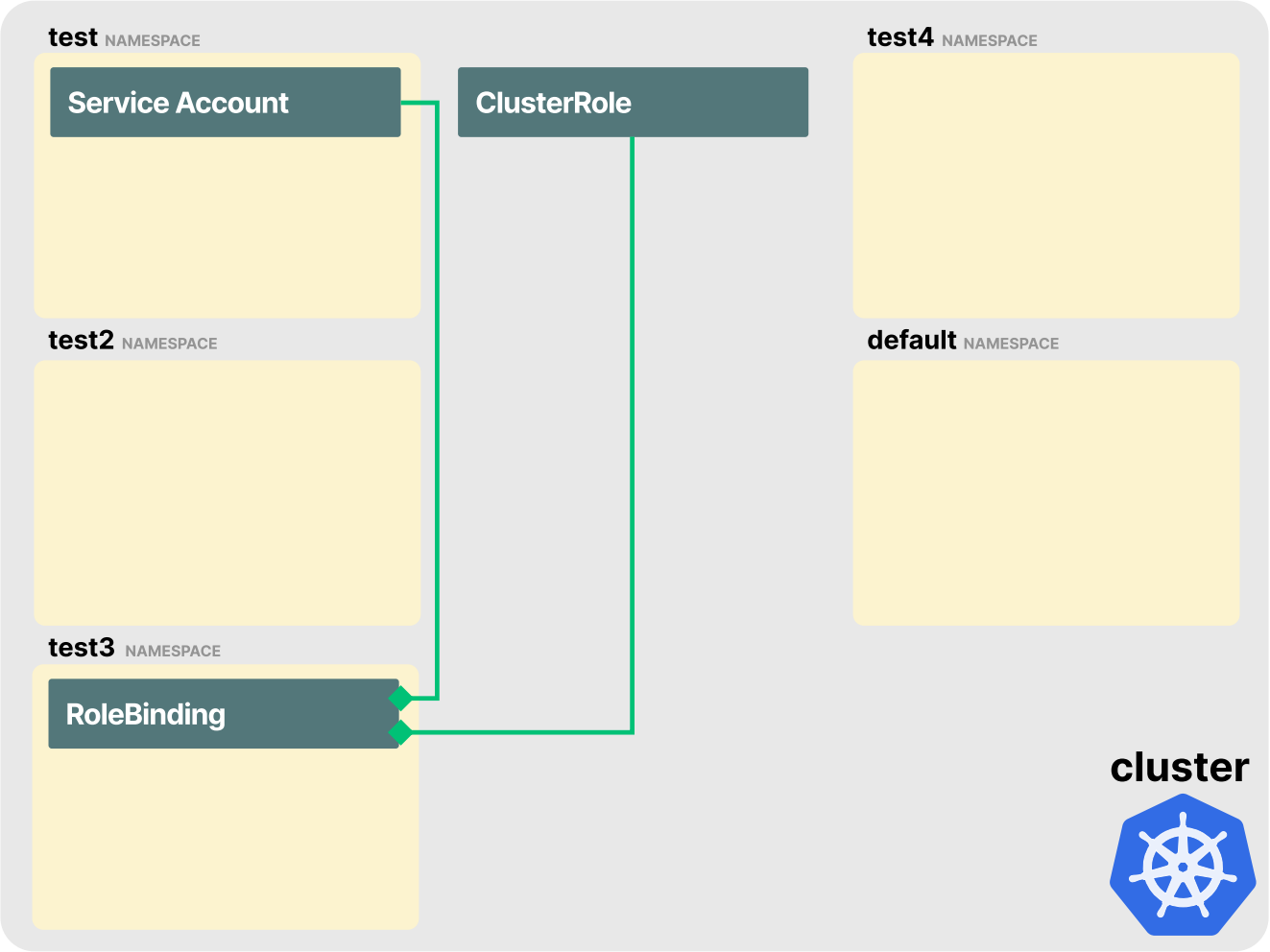
```
ization.k8s.io
```

e with:

bash

```
ario3.yaml
ization.k8s.io/testadminbinding created
```

Your cluster looks like this:



ount located in `test` has access to the

bash

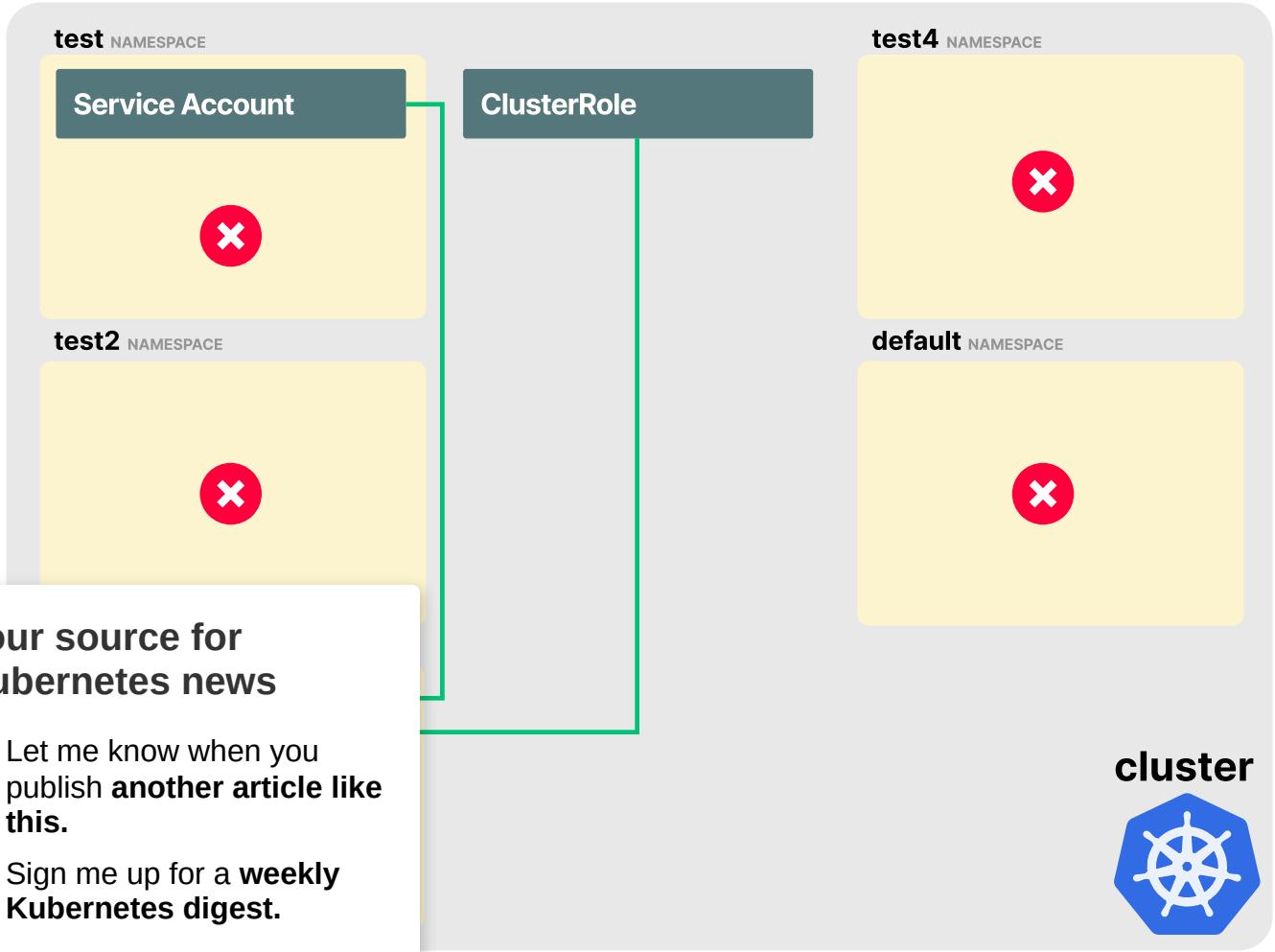
```
t pods -n test3 --as=system:serviceaccount:test:m
```

s to other namespaces:

bash

```
$ kubectl auth can-i get pods -n test4 --as=system:serviceaccount:test:m
no
$ kubectl auth can-i get pods --as=system:serviceaccount:test:myaccount
no

$ _
```



use a RoleBindings to link a Service Account to a
le behaves as if it were a regular Role.

o the current namespace where the RoleBinding
is located.

Scenario 4: Granting cluster-wide access with ClusterRole and ClusterRoleBinding

In this last scenario, you'll create a ClusterRoleBinding to link the ClusterRole to the Service Account:

scenario4.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: testadminclusterbinding
subjects:
  - kind: ServiceAccount
    name: myaccount
    namespace: test
roleRef:
  kind: ClusterRole
  name: cluster-admin
  ization.k8s.io
```

ace field on the `roleRef` again.

RoleBinding cannot identify a Role to link to amespaces, and ClusterRoleBindings (along with ence) are not namespaced.

e with:

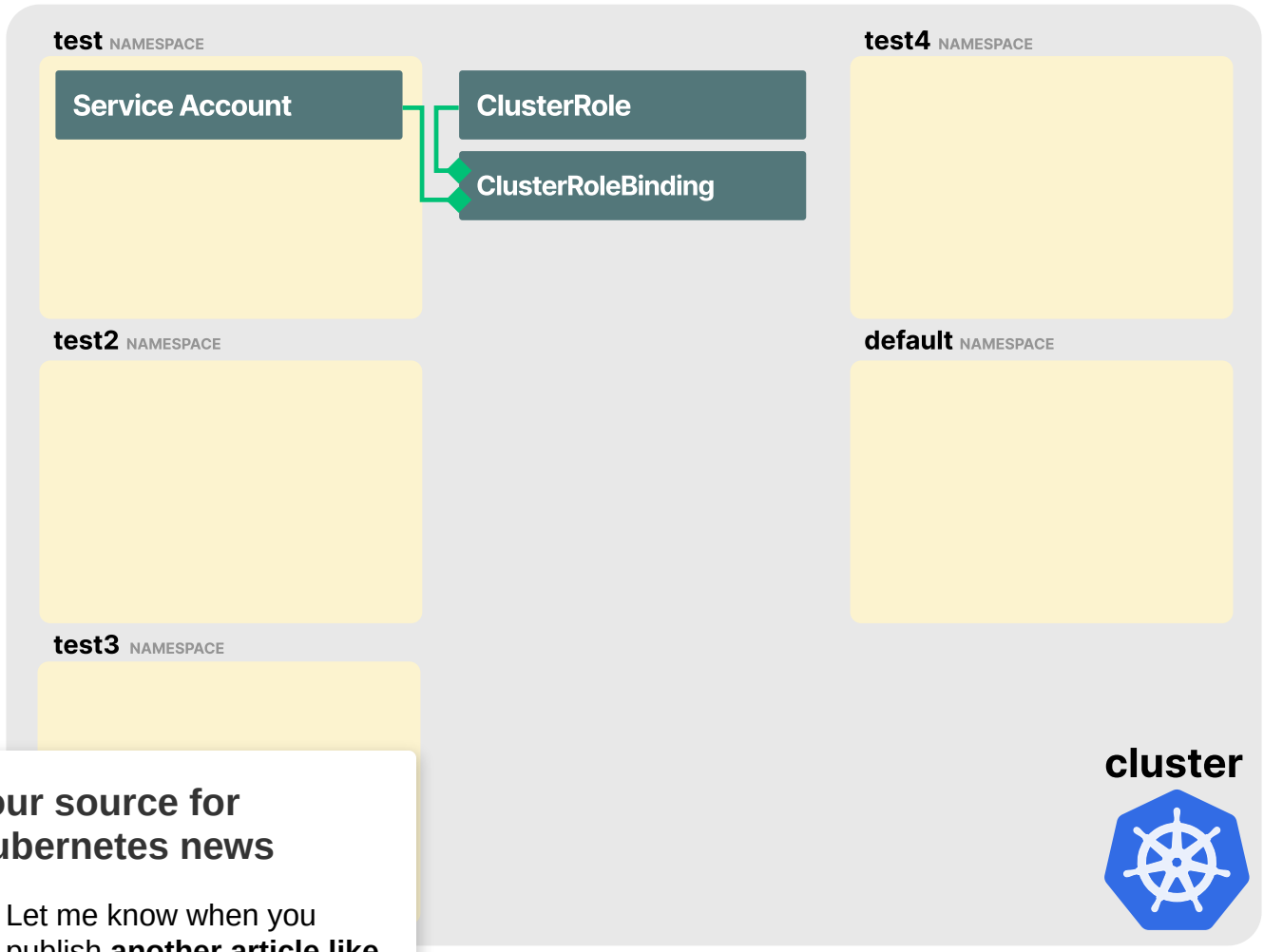
bash

ario3.yaml

```
rolebinding.rbac.authorization.k8s.io/testadminbinding created

$ _
```

Your cluster looks like this:



usterRole nor the ClusterRole binding defined any
ccount now has access to everything:

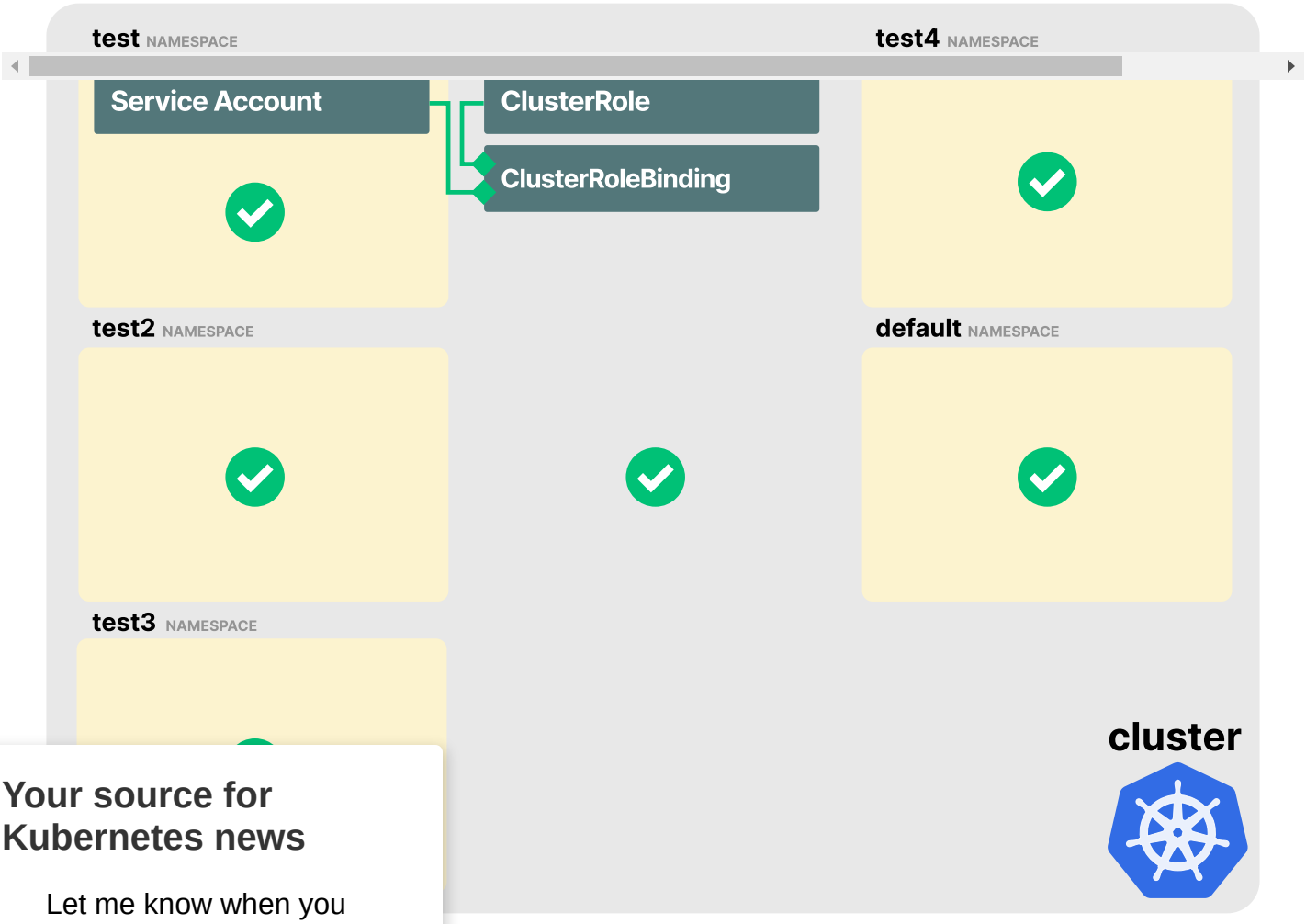
```
bash

t pods -n test4 --as=system:serviceaccount:test:m

t namespaces --as=system:serviceaccount:test:myac
```

```
Warning: resource 'namespaces' is not namespace scoped
yes

$ _
```



can observe some behaviours and limitations of

gs must exist in the same namespace.

t in separate namespaces to Service Accounts.

ClusterRoles, but they only grant access to the

eBinding.

- ClusterRoleBindings link accounts to ClusterRoles and grant access across all resources.
- ClusterRoleBindings can not reference Roles.

Perhaps the most interesting implication here is that a ClusterRole can define common permissions expressed in a single namespace when referenced by a RoleBinding.

This removes the need to have duplicated roles in many namespaces.

Bonus #1: Make RBAC policies more concise

The typical `rules` section of a Role or ClusterRole looks like this:

```
rules:
```

However, the above configurations can be re-written using the following format:

```
- apiGroups: ['']
  resources: ['services', 'endpoints', 'namespaces']
  verbs: ['get', 'list', 'watch', 'create', 'delete']
```

The alternative notation reduces the number of lines significantly and is more concise.

However, Kubernetes still manages the content as a YAML list when you retrieve it from the database.

So every time you get the Role, the array will be rendered into a list:

bash

```
$ kubectl get role pod-reader -o yaml
apiVersion: rbac.authorization.k8s.io/v1
```

Bonus #2: Using Service Account to create Kubernetes accounts

Service Accounts are usually created automatically by the API server and associated with pods running in the cluster.

Three separate components fulfil this task:

1. A **ServiceAccount admission controller** that injects the Service Account property in the Pod definition.
2. A **Token controller** that creates a companion Secret object.
3. A **ServiceAccount controller** creates the default Service Account in every namespace.

Service Accounts can be used outside the cluster to create identities for users or long-standing jobs that wish to talk to the Kubernetes API.

To manually create a Service Account, you can issue the following


```
                bash

eaccount demo-sa
created

counts demo-sa -o yaml

5126654"
selfLink: /api/v1/namespaces/default/serviceaccounts/demo-sa
```

```
  uid: 01b2a3f9-a373-6e74-b3ae-d89f6c0e321f
secrets:
- name: demo-sa-token-hrfq2

$ _
```

You might notice a `secrets` field at the end of the Service Account YAML definition.

What is that?

Every time you create a Service Account, Kubernetes creates a Secret.

The Secret holds the token for the Service Account, and you can use that token to call the Kubernetes API.

It also includes the public Certificate Authority (CA) of the API server:

bash

```
$ kubectl get secret demo-sa-token-hrfq2 -o yaml
apiVersion: v1
```

```
CA BASE64 ENCODED)
==
BASE64 ENCODED)
```

```
.
vice-account-token
```


that can be used as a bearer token to be-apiserver.

Usually, these **secrets are mounted into pods** for accessing the API server but can be used from outside the cluster.

Summary

RBAC in Kubernetes is the mechanism that enables you to configure fine-grained and specific sets of permissions that define how a given user, or group of users, can interact with any Kubernetes object in the cluster or a particular cluster namespace.

In this article, you learned:

- How RBAC decouples permissions from users with a more flexible model.
- How RBAC integrates with the Kubernetes API.
- How to identify subjects for RBAC with Users, Service Accounts and Groups.
- How to map Resources into Rules using Verbs and API groups.
- How to group rules into Roles and link those roles to identities using

en Roles, RoleBindings, ClusterRoles and

- Corporate training
- Public workshops
- Sponsorships
- Consulting

TOOLS

- Kubernetes instance calculator
- Troubleshooting flow chart
- Kubernetes production best practices
- See all tools

RESEARCH

- Ingress controller comparison
- Managed services comparison
- Resource allocations in Kubernetes nodes
- See all research

OUR NETWORK

- KubeFM
- Kube Events
- Kube Careers
- Kubernetes Architect
- Kubesploit
- K3s Daily