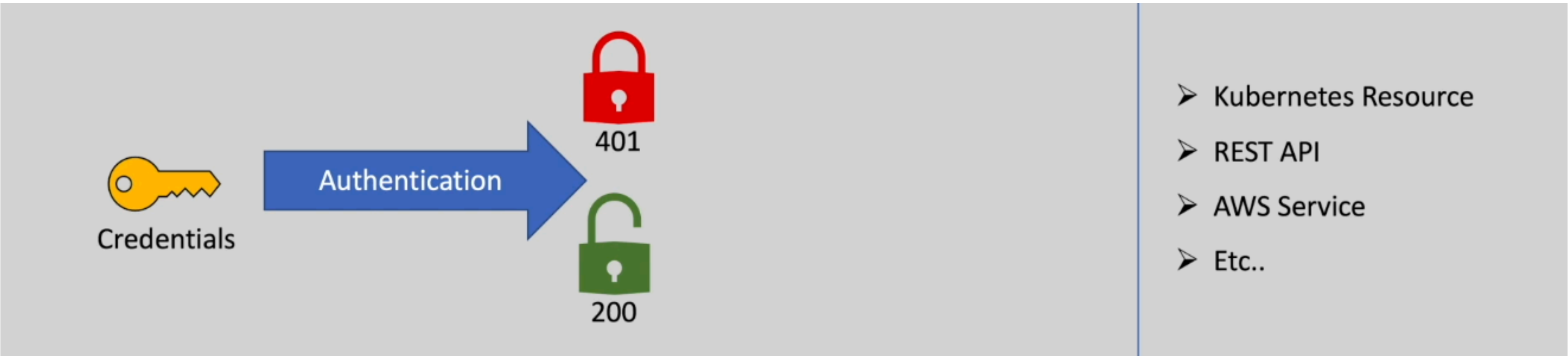
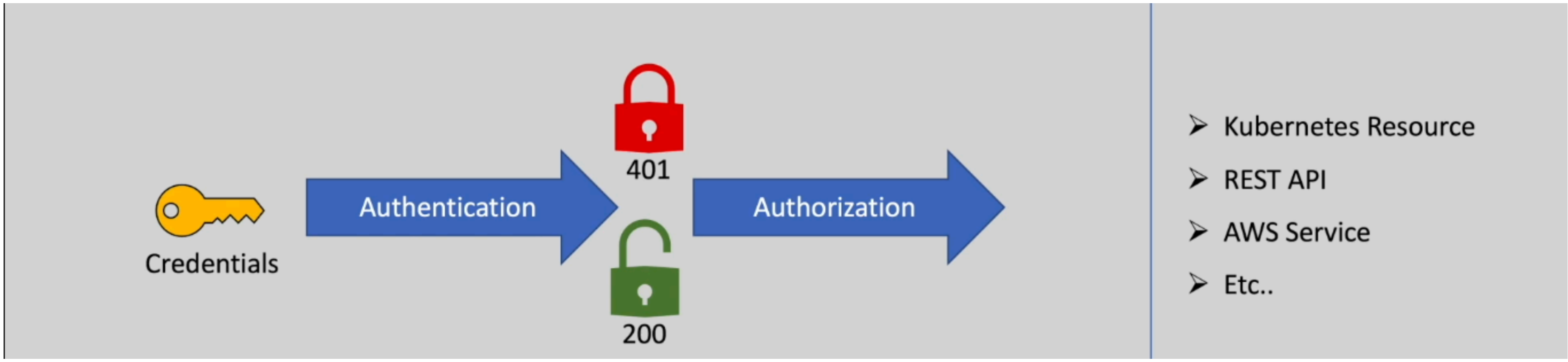
Table of Contents

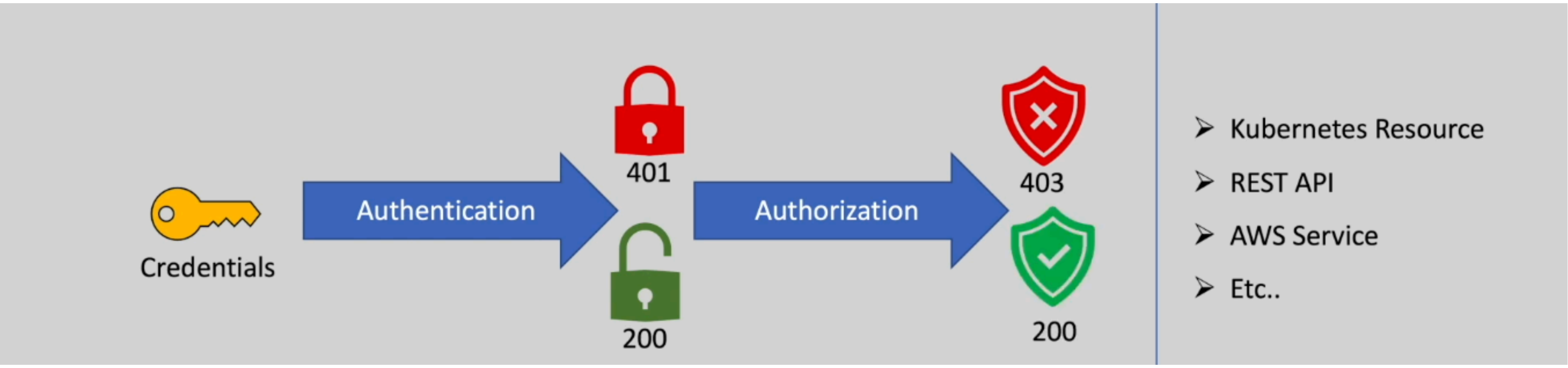
0.0: Authentication & Authorization :



We must be authorized to perform any action. After authentication we will check whether we are authorized to perform any action or not



If we are authorized to perform any action we will get 200 , if not then we will get 403



In summary with authentication we are proving that we are a valid user , and with authorization we will check if we perform any specific task.

There are different models with which we can achieve this authorization .

- RBAC (role based access control)
- ABAC (Attribute based access control)
- Node Authorization

The popular method is RBAC (Role-Based Access Control), where a user is allowed to perform a certain action on a given resource based on their assigned role.

Based on roles amount of access can vary,

Like

Role	Access
Developer	Create, Read , Update
Monitoring	Read
Admin (Can do Anything)	Create, Read, Update, Delete

1.0 : How role based access control works,

Role-based access control (RBAC) is a method of regulating access to computer or network resources based on the roles of individual users within your organization.

RBAC authorization uses the `rbac.authorization.k8s.io` API group to drive authorization decisions, allowing you to dynamically configure policies through the Kubernetes API.

1.2: API objects:

The RBAC API declares four kinds of Kubernetes objects: *Role, ClusterRole, RoleBinding* and *ClusterRoleBinding*. You can describe or amend the RBAC objects using tools such as `kubectl`, just like any other Kubernetes object.

2.0 : Create User :

Generate user access key with openssl

```
vishal@vishalk17:~/vishal/rbac$ openssl genrsa --out vishalk17.key 2048

vishal@vishalk17:~/vishal/rbac$ ls
vishalk17.key

vishal@vishalk17:~/vishal/rbac$ cat vishalk17.key
-----BEGIN PRIVATE KEY-----
MIIEvwIBADANBgkqhkiG9w0BAQEFAASCBKkwggSlAgEAAoIBAQC+ZxOcfOdpW/VF
3gO2cD69R/qf5ypwX2/WpYJSTPixNcQgbRK1DWJmhIUckFNS37dnGWNG4krteb/8
****************************************************************
****************************************************************
```

We have private key generated

Generates a new Certificate Signing Request (CSR) using the specified private key and subject details.

```
vishal@vishalk17:~/vishal/rbac$  openssl req --new --key vishalk17.key --out vishalk17.csr --subj
"/CN=vishalk17/O=dev/O=example.org"

vishal@vishalk17:~/vishal/rbac$ ls
vishalk17.csr  vishalk17.key
```

1. `openssl`: This is the OpenSSL command-line tool, which is used for various cryptography tasks such as creating and managing keys, generating CSRs, and more.

2. `req`: This subcommand is used to create and process certificate requests.
3. `--new`: This option specifies that a new certificate request is to be generated.
4. `--key vishalk17.key`: This option specifies the private key file (`vishalk17.key`) to use for creating the CSR. The private key is needed to sign the request.
5. `--out vishalk17.csr`: This option specifies the output file (`vishalk17.csr`) where the CSR will be saved.
6. `--subj "/CN=vishalk17/O=dev/O=example.org"`: This option allows you to specify the subject (Distinguished Name) of the CSR directly on the command line, bypassing the interactive prompts. The subject includes the following parts:
   - `/CN=vishalk17`: The Common Name (CN) for the certificate, usually the name of the user or the hostname of the server.
   - `/O=dev`: The Organization (O) field, which typically specifies the organization to which the entity belongs.
   - `/O=example.org`: Another Organization (O) field, which may specify a different part of the organization or an organizational unit.

This CSR must be signed by the Certificate authority. We can get the certificate authority details from kubeconfig file

```
vishal@vishalk17:~/vishal/rbac$ ls
vishalk17.csr  vishalk17.key

vishal@vishalk17:~/vishal/rbac$ openssl x509 -req -in vishalk17.csr -CA
/var/snap/microk8s/current/certs/ca.crt -CAkey /var/snap/microk8s/current/certs/ca.key -CAcreateserial
-out vishalk17.crt

Certificate request self-signature ok
subject=CN = vishalk17, O = dev, O = example.org

vishal@vishalk17:~/vishal/rbac$ ls
vishalk17.crt  vishalk17.csr  vishalk17.key
```

1. **Certificate request self-signature ok**:
   - This indicates that the signing operation completed successfully.
2. **subject=CN = vishalk17, O = dev, O = example.org**:
   - This shows the distinguished name (DN) fields from the certificate request (`vishalk17.csr`). Each field in the DN provides identifying information about the certificate holder.

Breakdown of the DN Fields:

- **CN = vishalk17**:
  - `CN` stands for Common Name. In this context, `vishalk17` is the common name of the entity (e.g., a user, device, or service) that the certificate is issued to.
- **O = dev**:
  - `O` stands for Organization. Here, `dev` represents the name of the organization.
- **O = example.org**:
  - This is another `O` field, which typically represents the organization or a domain name associated with the organization. In this case, `example.org`.

Summary:

The command generated a new certificate (`vishalk17.crt`) signed by your CA, using the information provided in the certificate signing request (`vishalk17.csr`). The `subject` line confirms that the request's distinguished name fields have been successfully incorporated into the signed certificate.

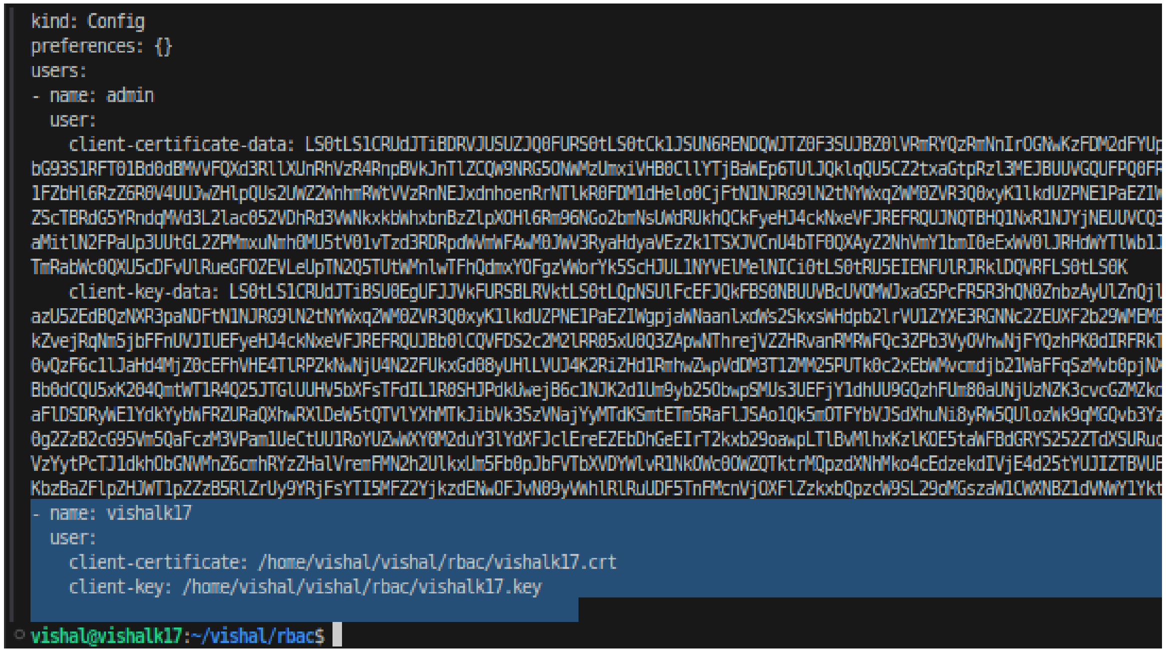
Add vishalk17 User to the Cluster using kubectl set-credential or adding a new user configuration to your Kubernetes configuration file (`kubeconfig`).

```
vishal@vishalk17:~/vishal/rbac$ kubectl config set-credentials vishalk17 --client-certificate=vishalk17.crt
--client-key=vishalk17.key

User "vishalk17" set.
```

Verify in kubeconfig file,

I'm using microk8s kubernetes distribution so , i m going to use command , in your case you might have file called kubeconfig

```
vishal@vishalk17:~/vishal/rbac$ microk8s config
```



```
kind: Config
preferences: {}
users:
- name: admin
  user:
    client-certificate-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUN6RENDQWJTZ0F3SUJBZ0lVRmRYQzRmNnIrOGNwKzFDM2dFYUp
bG93S1RFT01Bd0dBMVVFQXd3RllXUnRhVzR4RnpBVkJnTlZCQW9NRG5ONWMzUmxiVHB0CllYTjBaWEp6TUlJQklqQU5CZ2txaGtpRzl3MEJBUUVGQUFPQ0FP
1FZbHl6RzZ6R0V4UUJwZHlpQUs2UWZ2WnhmRWtVVzRnNEJxdnhoenRrNTlkR0FDM1dHelo0CjFtN1NJRG9lN2tNYWxqZWM0ZVR3Q0xyK1lkdUZPNE1PaEZ1W
ZScTBRdG5YRndqMVd3L2lac052VDhRd3VWNkxkbWhxbnBzZlpXOHl6Rm96NGo2bmNsUWdRUkhQCkFyeHJ4ckNxeVFJREFRQUJNQTBHQ1NxR1NJYjNEUUVCQ3
aMitlN2FPaUp3UUtGL2ZPMmxuNmh0MU5tV01vTzd3RDRpdWVmWFAwM0JWV3RyaHdyaVEzZk1TSXJVCnU4bTF0QXAyZ2NhVmY1bmI0eExwV0lJRHdWYTlWb1J
TmRabWc0QXU5cDFvUlRueGFOZEVLeUpTN2Q5TUtWMnlwTFhQdmxYOFgzVWorYk5ScHJUL1NYVElMelNICi0tLS0tRU5EIENFUlRJRklDQVRFLS0tLS0K
    client-key-data: LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSUlFcEFJQkFBS0NBUUVBcUVOMWJxaG5PcFR5R3hQN0ZnbzAyUlZnQjl
azU5ZEdBQzNXR3paNDFtN1NJRG9lN2tNYWxqZWM0ZVR3Q0xyK1lkdUZPNE1PaEZ1WgpjaWNaanlxdWs2SkxsWHdpb2lrVU1ZYXE3RGNNc2ZEUXF2b29WMEM
kZvejRqNm5jbFFnUVJIUEFyeHJ4ckNxeVFJREFRQUJBb0lCQVFDS2c2M2lRR05xU0Q3ZApwNThrejVZZHRvanRMRWFQc3ZPb3VyOVhwNjFYQzhPK0dIRFRkT
0vQzF6cllJaHd4MjZ0cEFhVHE4TlRPZkNwNjU4N2ZFUkxGd08yUHlLVUJ4K2RiZHd1RmhwZwpVdDM3T1ZMM25PUTk0c2xEbWMvcmdjb21WaFFqSzMvb0pjN
Bb0dCQU5xK204QmtWT1R4Q25JTGlUUHV5bXFsTFdIL1R0SHJPdkUwejB6c1NJK2d1Um9yb250bwpSMUs3UEFjY1dhUU9GQzhFUm80aUNjUzNZK3cvcGZMZk
aFlDSDRyWE1YdkYybWFRZURaQXhwRXlDeW5tQTVLYXhMTkJibVk3SzVNajYyMTdKSmtETm5RaFlJSAo1Qk5mOTFYbVJSdXhuNi8yRW5QUlozWk9qMGQvb3Yz
0g2ZzB2cG95Vm5QaFczM3VPam1UeCtUU1RoYUZwWXY0M2duY3lYdXFJclEreEZEbDhGeEIrT2kxb29oawpLTlBwMlhxKzlKOE5taWFBdGRYS252ZTdXSURu
VzYytPcTJ1dkhObGNVMnZ6cmhRYzZHalVremFMN2h2UlkxUm5Fb0pJbFVTbXVDYWlvR1NkOWc0OWZQTktrMQpzdXNhMko4cEdzekdIVjE4d25tYUJIZTBVUE
KbzBaZFlpZHJWT1pZZzB5RlZrUy9YRjFsYTI5MFZ2YjkzdENwOFJvN09yVWhlRlRuUDF5TnFMcnVjOXFlZzkxbQpzcW9SL29oMGszaW1CWXNBZ1dVMWY1Ykt
- name: vishalk17
  user:
    client-certificate: /home/vishal/vishal/rbac/vishalk17.crt
    client-key: /home/vishal/vishal/rbac/vishalk17.key

vishal@vishalk17:~/vishal/rbac$
```

3.0: Create Context for the user :

Cluster name : microk8s-cluster (This is my cluster name)
Context Name: vishalk17-context
User : vishalk17
Default Namespace to be set : Default

```
vishal@vishalk17:~/vishal/rbac$ kubectl config set-context vishalk17-context --cluster=microk8s-cluster
--namespace=default --user=vishalk17

Context "vishalk17-context" created.
```

`kubectl config set-context`: This command is used to set or modify a context in your kubeconfig file.
`vishalk17-context`: This is the name of the context you're creating or modifying. You can choose any meaningful name for the context.
`--cluster=microk8s-cluster`: This specifies the name of the cluster the context should use. This cluster must be defined in your kubeconfig file.
`--namespace=default`: This specifies the default namespace the context should use. When you don't specify a namespace in your `kubectl` commands, this default namespace will be used.
`--user=vishalk17`: This specifies the name of the user the context should use. This user must be defined in your kubeconfig file.

3.1 : Need for Contexts:

1. Simplifies Switching Between Environments:

Contexts allow you to easily switch between different clusters, namespaces, and users. For example, you might have multiple clusters (e.g., development, staging, production) and you can switch between them using different contexts without needing to reconfigure each time.

2. Facilitates Multi-User and Multi-Cluster Management:

If you're managing multiple clusters or working in different namespaces frequently, contexts help by providing a simple way to switch between these environments. Each context can specify a different cluster, namespace, and user, making it easier to manage access and configurations.

3. Enhances Security:

By using different contexts, you can ensure that you are operating with the correct permissions in the correct environment. This reduces the risk of accidental operations in the wrong cluster or namespace.

4. Improves Efficiency:

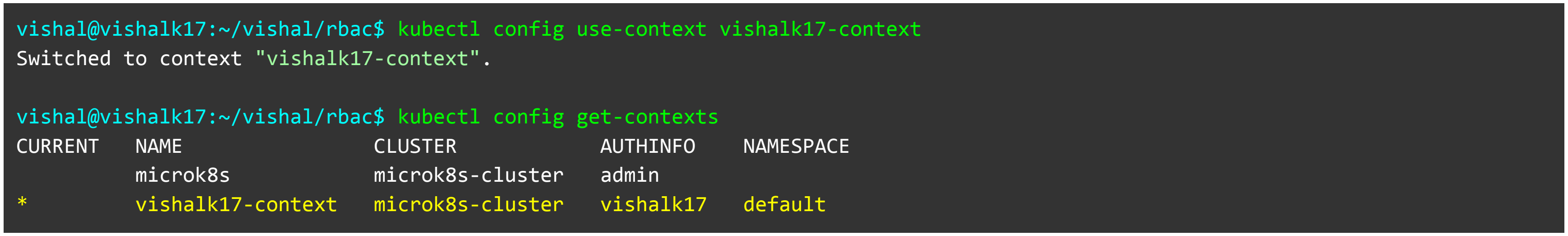
Contexts save time and reduce errors by eliminating the need to specify the cluster, namespace, and user for each `kubectl` command. Once a context is set, these parameters are automatically used for all commands, streamlining your workflow.

3.2 Verify the context :

```
vishal@vishalk17:~/vishal/rbac$ kubectl config  get-contexts
CURRENT   NAME                CLUSTER            AUTHINFO    NAMESPACE
*         microk8s            microk8s-cluster   admin
          vishalk17-context   microk8s-cluster   vishalk17   default
```

* indicates we are working with `microk8s` `context` Currently

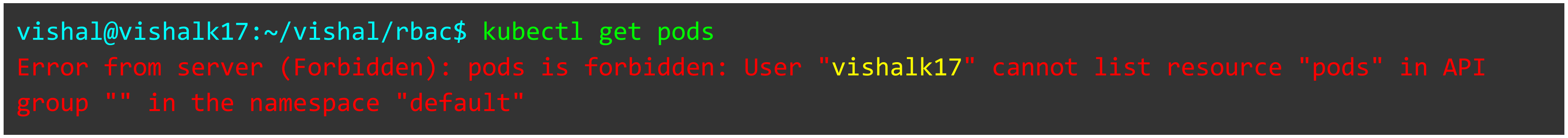
3.3: Switch the context



```
vishal@vishalk17:~/vishal/rbac$ kubectl config use-context vishalk17-context
Switched to context "vishalk17-context".

vishal@vishalk17:~/vishal/rbac$ kubectl config get-contexts
CURRENT   NAME                CLUSTER           AUTHINFO    NAMESPACE
          microk8s            microk8s-cluster  admin
*         vishalk17-context   microk8s-cluster  vishalk17   default
```

Now whatever we are sending request to the kubernetes cluster will goes on behalf of vishalk17 user,

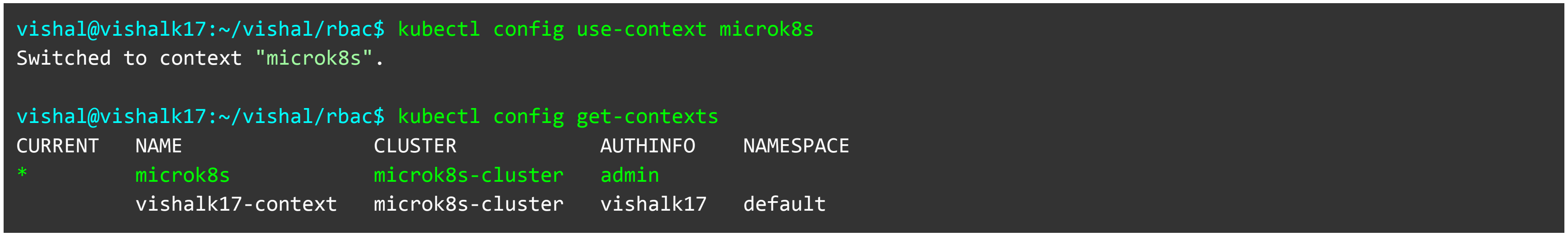
Let get list of pods



```
vishal@vishalk17:~/vishal/rbac$ kubectl get pods
Error from server (Forbidden): pods is forbidden: User "vishalk17" cannot list resource "pods" in API
group "" in the namespace "default"
```

As you can see this vishalk17 user forbidden to list down the pods, we have just created the user, by default it doesnt have any permission to access our cluster resources . It is valid user but not authorized to perform any action.

As an administrator we should give some permission to vishalk17 user, now switch back to default user , in my case it is microk8s which has access to everything



```
vishal@vishalk17:~/vishal/rbac$ kubectl config use-context microk8s
Switched to context "microk8s".

vishal@vishalk17:~/vishal/rbac$ kubectl config get-contexts
CURRENT   NAME                CLUSTER           AUTHINFO    NAMESPACE
*         microk8s            microk8s-cluster  admin
          vishalk17-context   microk8s-cluster  vishalk17   default
```

4.0: Role and Role Bindings:

Role: Defines a set of permissions within a namespace.

RoleBinding: Associates a Role with users, groups, or ServiceAccounts to grant permissions within that namespace.

4.1: Role:

Sample Role.yaml file

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

apiVersion: rbac.authorization.k8s.io/v1

- Specifies the API version for Role-based access control (RBAC). The `rbac.authorization.k8s.io/v1` is the stable version for RBAC resources.

kind: Role

- Defines the kind of resource. In this case, it's a `Role`, which is a set of permissions within a specific namespace.

metadata:

- **namespace: default**
  - Specifies the namespace where the role is applied. Here, it's the `default` namespace.
- **name: pod-reader**
  - Sets the name of the role. This role is named `pod-reader`.

rules:

- A list of rules that define the permissions included in this role.

apiGroups: [""]

- Indicates the API group for the resources. An empty string `[""]` signifies the core API group (core/v1), which includes fundamental resources like pods, services, etc.

resources: ["pods"]

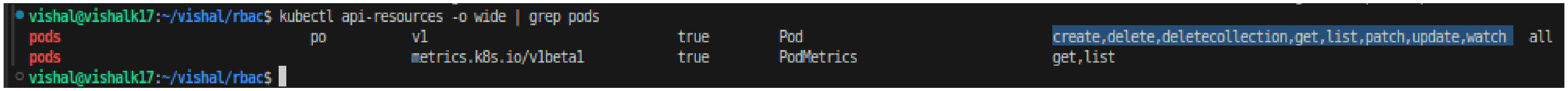
- Specifies the resources this role can access. In this case, it's `pods`.

verbs: ["get", "watch", "list"]

- Defines the actions (verbs) that are allowed on the specified resources.
  - **get**: Allows reading the details of a pod.
  - **watch**: Allows watching for changes to pods.
  - **list**: Allows listing all pods.

To see what kind of verbs we can give for particular user, run following command

```
kubectl api-resources -o wide | grep pods
```



```
vishal@vishalk17:~/vishal/rbac$ kubectl api-resources -o wide | grep pods
pods                    po          v1                        true        Pod                              create,delete,deletecollection,get,list,patch,update,watch   all
pods                                metrics.k8s.io/v1beta1    true        PodMetrics                       get,list
vishal@vishalk17:~/vishal/rbac$
```

So, for pod we can use any of these verbs.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

Apply role.yaml

```
vishal@vishalk17:~/vishal/rbac$ vi role.yaml
vishal@vishalk17:~/vishal/rbac$ kubectl apply -f role.yaml
role.rbac.authorization.k8s.io/pod-reader created
```

Verify:

```
vishal@vishalk17:~/vishal/rbac$ kubectl get role
NAME         CREATED AT
pod-reader   2024-06-19T21:15:30Z
```

4.2: Role Binding :

Now **we have a user and role ready.** Now how will we connect these two so the user will have the permission ?

How Role Binding Works:

1. **Roles:**
   - A Role is a set of permissions that define what actions a user, group, or service account can perform on specific resources within a namespace.
   - ClusterRoles are similar but grant permissions across the entire cluster, not just within a single namespace.
2. **Subjects:**
   - Subjects are the entities that you want to grant permissions to. This can include users, groups, or service accounts.
3. **Role Binding:**
   - A RoleBinding is the object that connects a Role (or ClusterRole) to one or more Subjects. It essentially says, "These Subjects are allowed to perform the actions defined in this Role (or ClusterRole)."
   - RoleBindings are namespace-scoped, meaning they grant permissions within a specific namespace.
   - ClusterRoleBindings grant permissions cluster-wide and can reference either ClusterRoles or Roles.

Create **rolebinding.yaml** file :

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding  # Declares this resource as a RoleBinding
metadata:
  name: pod-reader-binding  # The name of this RoleBinding
  namespace: default  # The namespace where this RoleBinding is created
subjects:  # Specifies the users, groups, or service accounts this RoleBinding applies to
- kind: User # The kind of subject (in this case, a User)
  name: vishalk17  # The name of the user being granted access
  apiGroup: rbac.authorization.k8s.io
roleRef:  # References the role being bound by this RoleBinding
  kind: Role  # The kind of role being referenced
  name: pod-reader  # The name of the role being bound (must match the name of an existing Role)
  apiGroup: rbac.authorization.k8s.io
```

- Under `metadata`, the `name` field specifies the name of the RoleBinding, and `namespace` indicates in which namespace it is created.
- The `subjects` section lists the users, groups, or service accounts that the role applies to. In this case, it is a user named `vishalk17`.
- The `roleRef` section references the role that this binding applies to. It specifies the kind of role (`Role`), the name of the role (`pod-reader`), and the API group for RBAC.

```
vishal@vishalk17:~/vishal/rbac$ ls
rolebinding.yaml  role.yaml  vishalk17.crt  vishalk17.csr  vishalk17.key

vishal@vishalk17:~/vishal/rbac$ kubectl apply -f rolebinding.yaml
rolebinding.rbac.authorization.k8s.io/pod-reader-binding created
```

Now, User vishalk17 has the permission of : `["get", "watch", "list"]` with ref to role.yaml

Verify:

Apply Sample deployment file (**sample_deployment.yaml**):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
```

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

```
vishal@vishalk17:~/vishal/rbac$ ls
rolebinding.yaml  role.yaml  sample_deployment.yaml  vishalk17.crt  vishalk17.csr  vishalk17.key

vishal@vishalk17:~/vishal/rbac$ kubectl apply -f sample_deployment.yaml
deployment.apps/nginx-deployment created
service/nginx-service created
```

Switch the context:

```
vishal@vishalk17:~/vishal/rbac$ kubectl config use-context vishalk17-context
Switched to context "vishalk17-context".

vishal@vishalk17:~/vishal/rbac$ kubectl config get-contexts
CURRENT   NAME                CLUSTER           AUTHINFO    NAMESPACE
          microk8s            microk8s-cluster  admin
*         vishalk17-context   microk8s-cluster  vishalk17   default
```

Test A : List the pods , Services in same namespace

Note: User vishalk17 has the permission of `["get", "watch", "list"]` , `resources: ["pods"]` with ref to role.yaml

```
vishal@vishalk17:~/vishal/rbac$ kubectl get pods -n default
NAME                               READY   STATUS    RESTARTS   AGE
nginx-deployment-576c6b7b6-r4rnq   1/1     Running   0          4m32s

vishal@vishalk17:~/vishal/rbac$ kubectl get svc -n default
Error from server (Forbidden): services is forbidden: User "vishalk17" cannot list resource "services" in API group "" in the namespace "default"
```

Why `kubectl get pods -n default` Works

- The command `kubectl get pods -n default` attempts to list all pods in the `default` namespace.
- Since the Role `pod-reader` grants the `get`, `watch`, and `list` verbs for the `pods` resource in the `default` namespace, the user `vishalk17` has the necessary permissions to execute this command.
- Therefore, the command succeeds and lists the pods.

Why `kubectl get svc -n default` Does Not Work

- The command `kubectl get svc -n default` attempts to list all services in the `default` namespace.
- The Role `pod-reader` does not grant any permissions for the `services` resource; it only grants permissions for the `pods` resource.
- Since the user `vishalk17` does not have any Role that grants permissions to `get`, `watch`, or `list` services, the attempt to list services fails.
- Kubernetes returns a "forbidden" error because the user `vishalk17` lacks the necessary permissions to list services.

Test B : Get list of pods in default and kube-system namespace

```
vishal@vishalk17:~/vishal/rbac$ kubectl get pods -n default
NAME                               READY   STATUS    RESTARTS   AGE
nginx-deployment-576c6b7b6-r4rnq   1/1     Running   0          34m

vishal@vishalk17:~/vishal/rbac$ kubectl get pods -n kube-system
Error from server (Forbidden): pods is forbidden: User "vishalk17" cannot list resource "pods" in API
group "" in the namespace "kube-system"
```

Resolving the Issue

To allow user `vishalk17` to list pods in both the `default` and `kube-system` namespaces, you need to create a Role and RoleBinding for each namespace

or

Use a ClusterRole and ClusterRoleBinding for cluster-wide permissions.

5.0: Cluster Role & ClusterRoleBinding :

Concept	Scope	Applies to	Grants permissions to	Example
Role	Namespace	Resources within a namespace	Users or service accounts within a namespace	`pod-reader` Role in `default` namespace grants permission to read pods
RoleBinding	Namespace	Users or service accounts within a namespace	Binds a Role to a user or service account within a namespace	`pod-reader-binding` binds `pod-reader` Role to user `vishalk17` in `default` namespace
ClusterRole	Cluster-wide	Resources across the entire cluster	Users or service accounts across the entire cluster	`cluster-admin` ClusterRole grants permission to create, update, and delete resources across the cluster
ClusterRoleBinding	Cluster-wide	Users or service accounts across the entire cluster	Binds a ClusterRole to a user or service account across the entire cluster	`cluster-admin-binding` binds `cluster-admin` ClusterRole to user `admin` across the entire cluster

5.1: When to use each:

- Use `Role` and `RoleBinding` when you want to grant permissions to a user or service account within a single namespace.
- Use `ClusterRole` and `ClusterRoleBinding` when you want to grant permissions to a user or service account across the entire cluster.

ClusterRole.yaml :

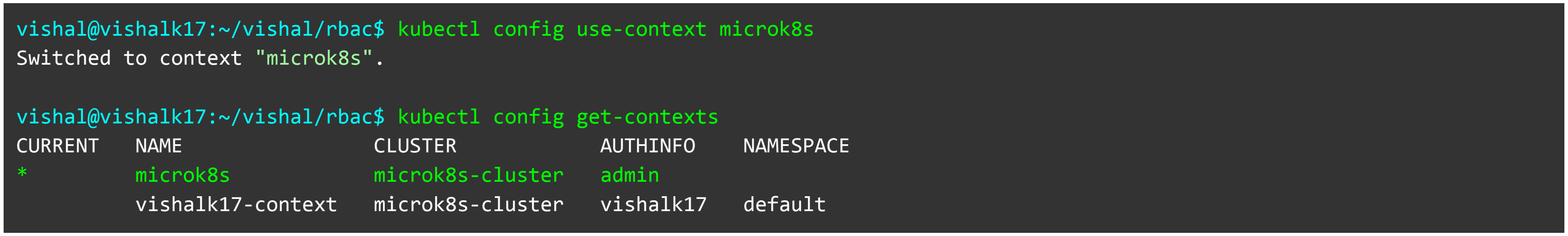
```
apiVersion: rbac.authorization.k8s.io/v1    #    Specifies the API version for RBAC
kind: ClusterRole                           #    Declares this resource as a ClusterRole
metadata:
  name: cluster-pod-reader                  #    The name of this ClusterRole
rules:                                      #    Specifies the rules/permissions of this ClusterRole
- apiGroups: [""]                           #    "" indicates the core API group
  resources: ["pods", "pods/log", "services"] #    Specifies the resources (e.g., pods)
  verbs: ["get", "list", "watch"]           #    Specifies the actions/permissions (e.g., get, list, watch)
```

ClusterRoleBinding.yaml :

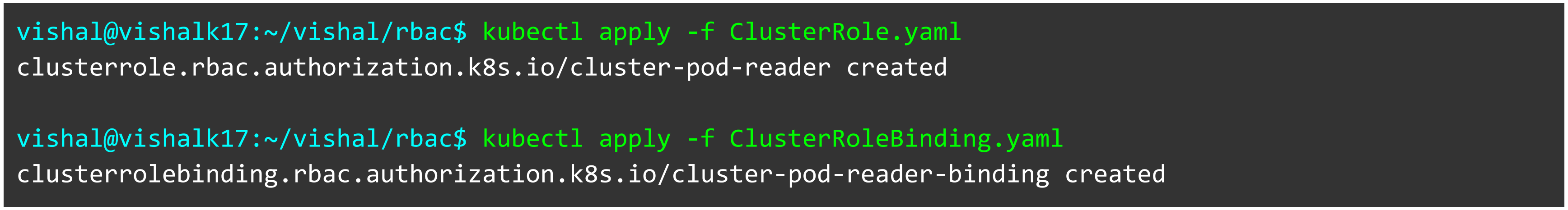
```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding   #    Declares this resource as a ClusterRoleBinding
metadata:
  name: cluster-pod-reader-binding   #    The name of this ClusterRoleBinding
subjects:  #    Specifies the users, groups, or service accounts this ClusterRoleBinding applies to
- kind: User
  name: vishalk17   #    The name of the user being granted access
  apiGroup: rbac.authorization.k8s.io
roleRef:   #    References the ClusterRole being bound by this ClusterRoleBinding
  kind: ClusterRole
  name: cluster-pod-reader  #    The name of the ClusterRole being bound (must match the name of an existing ClusterRole)
  apiGroup: rbac.authorization.k8s.io
```

Test A : Apply Cluster Role and ClusterRoleBinding and check whether user able to access all the namespaces with ref to permission Assigned in ClusterRole.yaml

Now switch back to default user , in my case it is microk8s which has access to everything



```
vishal@vishalk17:~/vishal/rbac$ kubectl config use-context microk8s
Switched to context "microk8s".

vishal@vishalk17:~/vishal/rbac$ kubectl config get-contexts
CURRENT   NAME                CLUSTER            AUTHINFO    NAMESPACE
*         microk8s            microk8s-cluster   admin
          vishalk17-context   microk8s-cluster   vishalk17   default
```

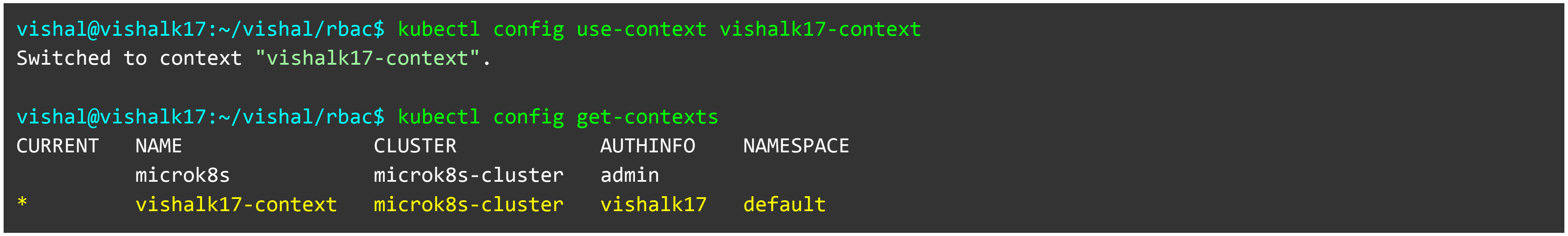


```
vishal@vishalk17:~/vishal/rbac$ kubectl apply -f ClusterRole.yaml
clusterrole.rbac.authorization.k8s.io/cluster-pod-reader created

vishal@vishalk17:~/vishal/rbac$ kubectl apply -f ClusterRoleBinding.yaml
clusterrolebinding.rbac.authorization.k8s.io/cluster-pod-reader-binding created
```

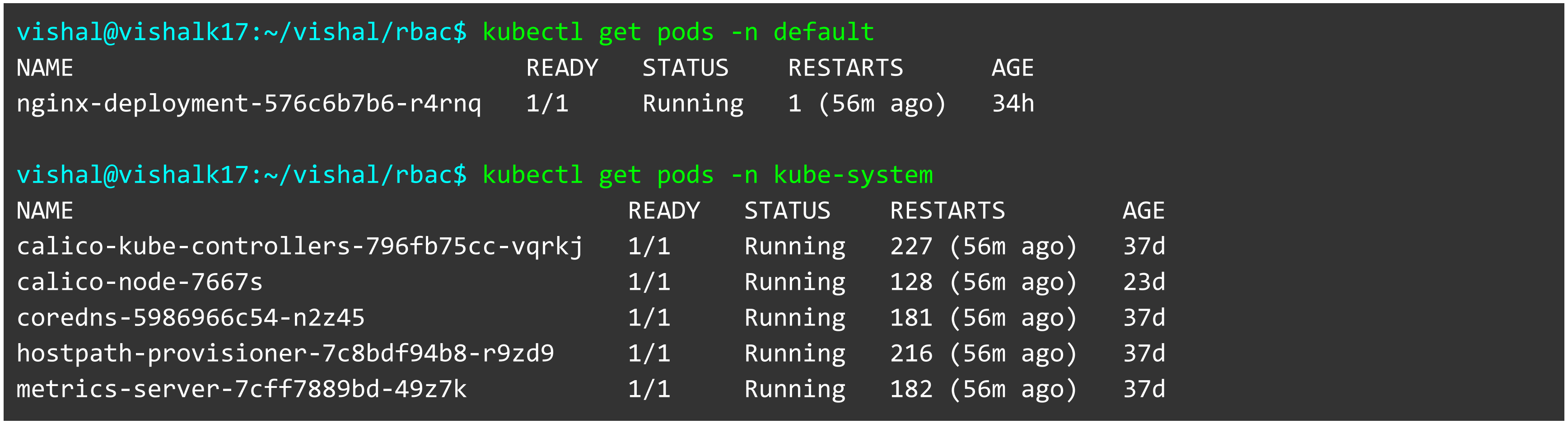
Switch the context:

Now switch back to vishalk17 user , and check whether this service account or user we say is able to access all the namespaces or not with ref. To clusterRole :

```
  resources: ["pods", "pods/log", "services"]
  verbs: ["get", "list", "watch"]
```



```
vishal@vishalk17:~/vishal/rbac$ kubectl config use-context vishalk17-context
Switched to context "vishalk17-context".

vishal@vishalk17:~/vishal/rbac$ kubectl config get-contexts
CURRENT   NAME                CLUSTER           AUTHINFO    NAMESPACE
          microk8s            microk8s-cluster  admin
*         vishalk17-context   microk8s-cluster  vishalk17   default
```



```
vishal@vishalk17:~/vishal/rbac$ kubectl get pods -n default
NAME                                READY   STATUS    RESTARTS      AGE
nginx-deployment-576c6b7b6-r4rnq    1/1     Running   1 (56m ago)   34h

vishal@vishalk17:~/vishal/rbac$ kubectl get pods -n kube-system
NAME                                       READY   STATUS    RESTARTS        AGE
calico-kube-controllers-796fb75cc-vqrkj    1/1     Running   227 (56m ago)   37d
calico-node-7667s                          1/1     Running   128 (56m ago)   23d
coredns-5986966c54-n2z45                   1/1     Running   181 (56m ago)   37d
hostpath-provisioner-7c8bdf94b8-r9zd9      1/1     Running   216 (56m ago)   37d
metrics-server-7cff7889bd-49z7k            1/1     Running   182 (56m ago)   37d
```

Conclusion: I am able to access the resources mentioned in the `ClusterRole.yaml` cluster-wide (in all namespaces).

6.0 : Managing Permissions with Group for a Large Number of Users in a Cluster :

Challenge: Handling Permissions for Numerous Users

- Managing permissions for hundreds of users individually in a `ClusterRoleBinding` is tedious and inefficient.

Solution: Use Groups for Permission Management

- Instead of assigning permissions to individual users, assign permissions to groups. Users within the group inherit these permissions automatically.

Assigning Permissions to Groups

- Create a `ClusterRole` that specifies the permissions required.
- Create a `ClusterRoleBinding` that assigns the `ClusterRole` to a group.

Adding Users to Groups

- Ensure users are added to the appropriate group.
- All users in the group will inherit the permissions specified in the `ClusterRole`.

Advantages of Using Groups

- Simplifies the process of managing permissions.
- Reduces the risk of errors when assigning permissions.
- Scales efficiently as the number of users grows.

Example: Configuring Permissions for a Group

- Create a `ClusterRole` that defines the permissions for the group.
- Create a `ClusterRoleBinding` that binds the `ClusterRole` to the group.

User Group Membership

- Ensure that users are added to the correct groups to receive the necessary permissions.

Test A: Assigning Permissions to Multiple Users via Group and Verification

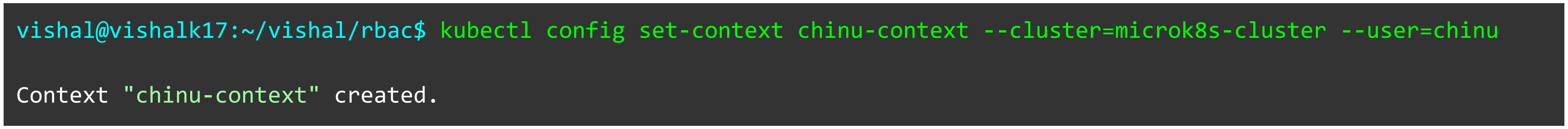
We already have one user vishalk17 , lets create another user chinu.

Create one more user : (I kept private key same vishalk17.key , you can create new one if you like to !!)

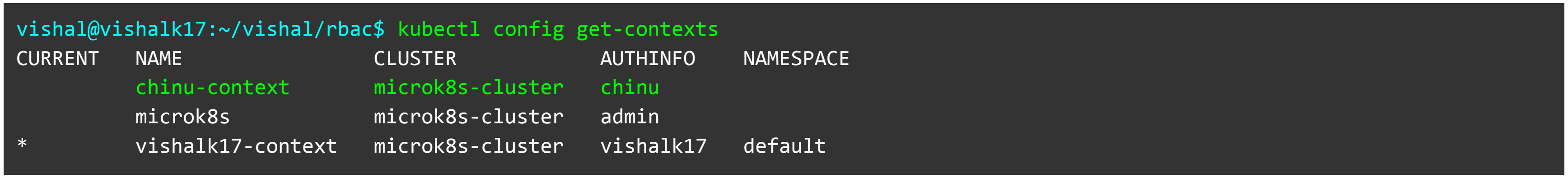
```
vishal@vishalk17:~/vishal/rbac$ ls
ClusterRoleBinding.yaml  ClusterRole.yaml  rolebinding.yaml  role.yaml  sample_deployment.yaml  vishalk17.crt
vishalk17.csr  vishalk17.key

vishal@vishalk17:~/vishal/rbac$ openssl req --new --key vishalk17.key --out chinu.csr --subj
"/CN=chinu/O=dev/O=example.org"

vishal@vishalk17:~/vishal/rbac$ openssl x509 -req -in chinu.csr -CA /var/snap/microk8s/current/certs/ca.crt -CAkey
/var/snap/microk8s/current/certs/ca.key -CAcreateserial -out chinu.crt

Certificate request self-signature ok
subject=CN = chinu, O = dev, O = example.org   // added to dev group just like vishalk17 user we created last time

vishal@vishalk17:~/vishal/rbac$ ls | grep chinu
chinu.crt
Chinu.csr

vishal@vishalk17:~/vishal/rbac$ kubectl config set-credentials chinu --client-certificate=chinu.crt
--client-key=vishalk17.key

User "chinu" set.
```

Create Context for user “chinu” (For Testing purpose):

Cluster name : microk8s-cluster (This is my cluster name)
Context Name: chinu-context
User : chinu

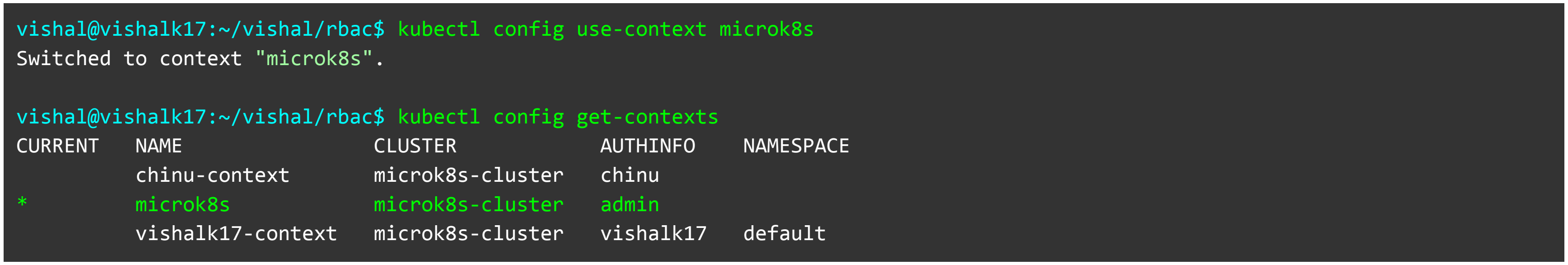


```
vishal@vishalk17:~/vishal/rbac$ kubectl config set-context chinu-context --cluster=microk8s-cluster --user=chinu

Context "chinu-context" created.
```



```
vishal@vishalk17:~/vishal/rbac$ kubectl config get-contexts
CURRENT   NAME                CLUSTER            AUTHINFO    NAMESPACE
          chinu-context       microk8s-cluster   chinu
          microk8s            microk8s-cluster   admin
*         vishalk17-context   microk8s-cluster   vishalk17   default
```

Switch back to microk8s context where we have admin access,



```
vishal@vishalk17:~/vishal/rbac$ kubectl config use-context microk8s
Switched to context "microk8s".

vishal@vishalk17:~/vishal/rbac$ kubectl config get-contexts
CURRENT   NAME                CLUSTER            AUTHINFO    NAMESPACE
          chinu-context       microk8s-cluster   chinu
*         microk8s            microk8s-cluster   admin
          vishalk17-context   microk8s-cluster   vishalk17   default
```

Vishalk17 user and chinu users both are in the same group that is dev hence we can add a group in ClusterRole.yaml instead of user.

For ref.

```
subject=CN = chinu, O = dev, O = example.org
subject=CN = vishalk17, O = dev, O = example.org
```

ClusterRole-dev-group.yaml :

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: dev-cluster-role
rules:
- apiGroups: [""]
  resources: ["pods", "services", "deployments", "configmaps"]
  verbs: ["get", "list", "watch", "create", "update", "delete"]
```

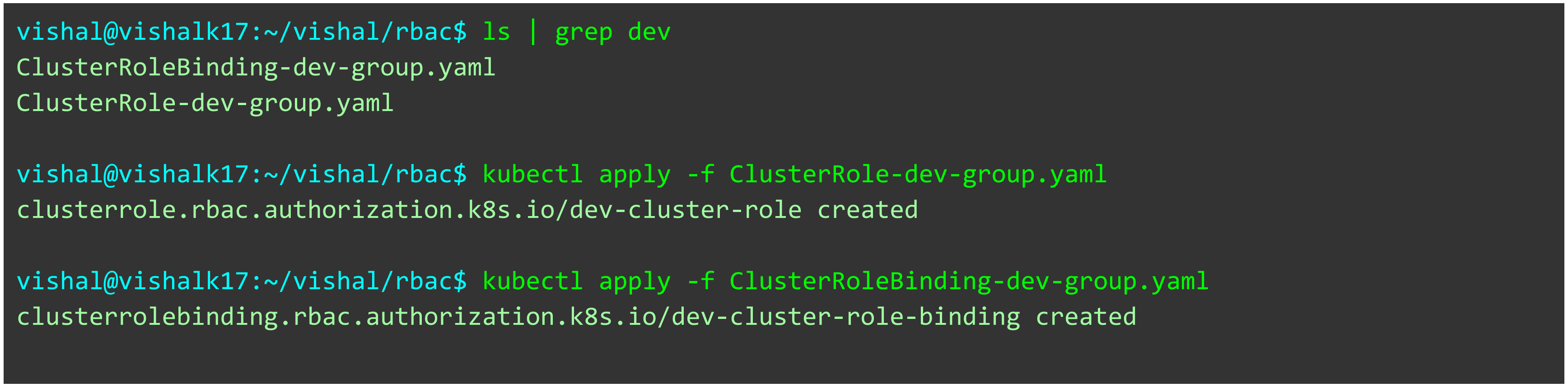
ClusterRoleBinding-dev-group.yaml :

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: dev-cluster-role-binding
subjects:
- kind: Group
  name: dev           # Name of the group
  apiGroup: rbac.authorization.k8s.io
```

```
roleRef:
  kind: ClusterRole
  name: dev-cluster-role   # Name of the ClusterRole defined above
  apiGroup: rbac.authorization.k8s.io
```

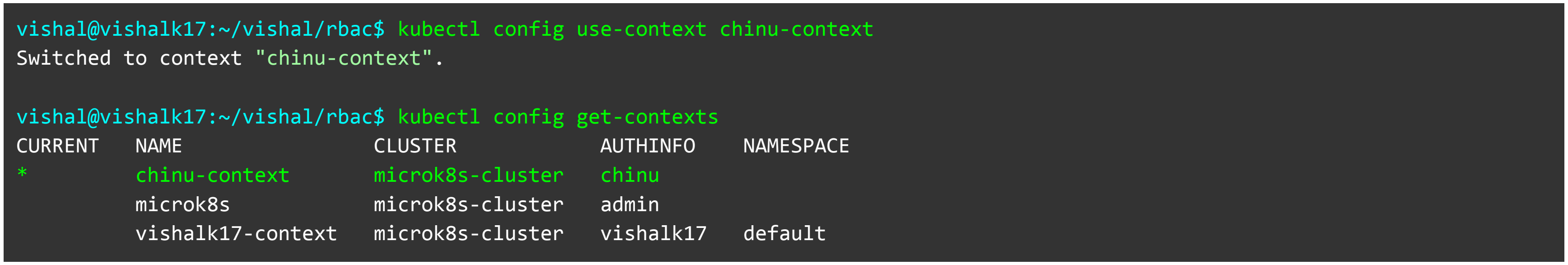
Clear up all our previous mess ,Let's start fresh

```
vishal@vishalk17:~/vishal/rbac$ ls
chinu.crt  ClusterRoleBinding-dev-group.yaml  ClusterRole-dev-group.yaml  rolebinding.yaml  sample_deployment.yaml
vishalk17.csr chinu.csr  ClusterRoleBinding.yaml            ClusterRole.yaml            role.yaml
vishalk17.crt vishalk17.key

vishal@vishalk17:~/vishal/rbac$ kubectl delete -f ./
clusterrole.rbac.authorization.k8s.io "dev-cluster-role" deleted
clusterrole.rbac.authorization.k8s.io "cluster-pod-reader" deleted
clusterrolebinding.rbac.authorization.k8s.io "dev-cluster-role-binding" deleted
clusterrolebinding.rbac.authorization.k8s.io "cluster-pod-reader-binding" deleted
role.rbac.authorization.k8s.io "pod-reader" deleted
rolebinding.rbac.authorization.k8s.io "pod-reader-binding" deleted
deployment.apps "nginx-deployment" deleted
service "nginx-service" deleted
```



```
vishal@vishalk17:~/vishal/rbac$ ls | grep dev
ClusterRoleBinding-dev-group.yaml
ClusterRole-dev-group.yaml

vishal@vishalk17:~/vishal/rbac$ kubectl apply -f ClusterRole-dev-group.yaml
clusterrole.rbac.authorization.k8s.io/dev-cluster-role created

vishal@vishalk17:~/vishal/rbac$ kubectl apply -f ClusterRoleBinding-dev-group.yaml
clusterrolebinding.rbac.authorization.k8s.io/dev-cluster-role-binding created
```

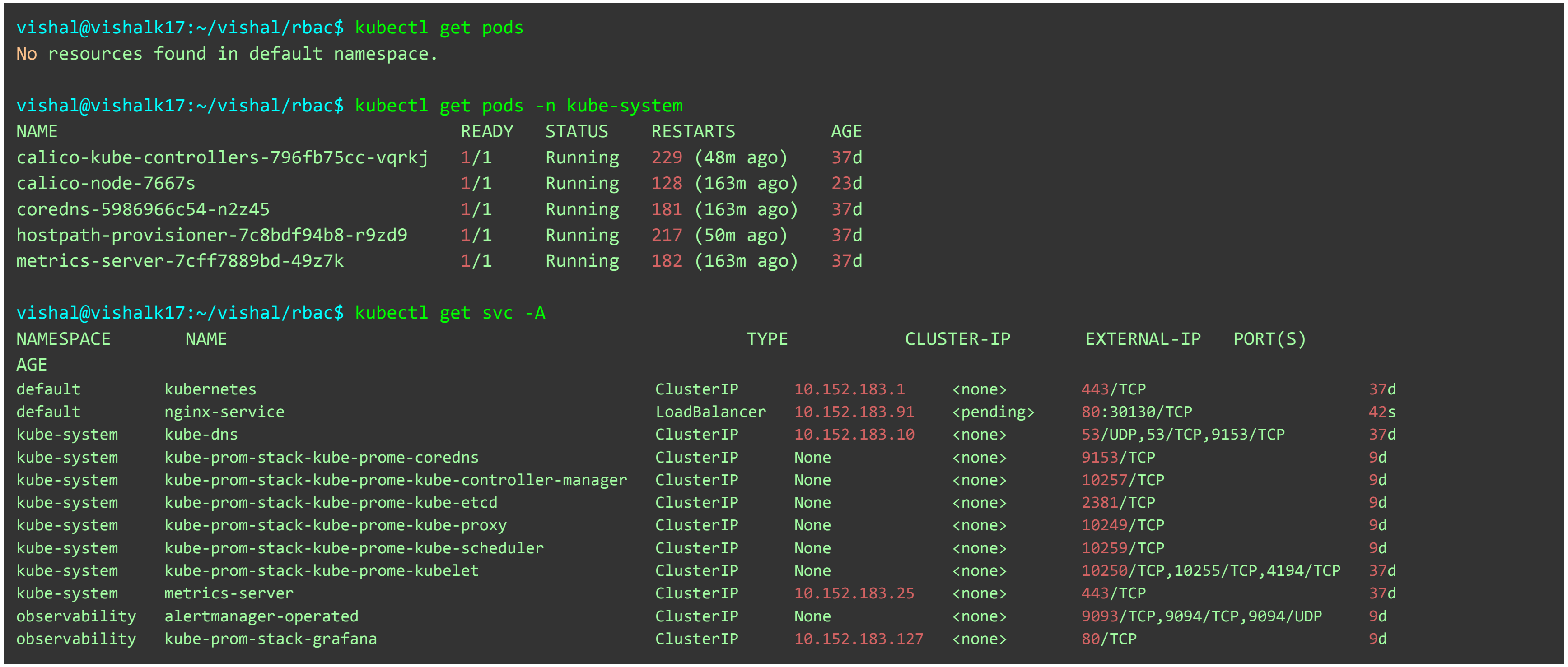
Switch to chinu-context ,



```
vishal@vishalk17:~/vishal/rbac$ kubectl config use-context chinu-context
Switched to context "chinu-context".

vishal@vishalk17:~/vishal/rbac$ kubectl config get-contexts
CURRENT   NAME                CLUSTER            AUTHINFO    NAMESPACE
*         chinu-context       microk8s-cluster   chinu
          microk8s            microk8s-cluster   admin
          vishalk17-context   microk8s-cluster   vishalk17   default
```

Now verify,

Whether chinu user has all the permission or not with ref to clusterrole in which we assigned permission to dev group:



```
vishal@vishalk17:~/vishal/rbac$ kubectl get pods
No resources found in default namespace.

vishal@vishalk17:~/vishal/rbac$ kubectl get pods -n kube-system
NAME                                      READY   STATUS    RESTARTS         AGE
calico-kube-controllers-796fb75cc-vqrkj   1/1     Running   229 (48m ago)    37d
calico-node-7667s                         1/1     Running   128 (163m ago)   23d
coredns-5986966c54-n2z45                  1/1     Running   181 (163m ago)   37d
hostpath-provisioner-7c8bdf94b8-r9zd9     1/1     Running   217 (50m ago)    37d
metrics-server-7cff7889bd-49z7k           1/1     Running   182 (163m ago)   37d

vishal@vishalk17:~/vishal/rbac$ kubectl get svc -A
NAMESPACE       NAME                                                 TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
default         kubernetes                                           ClusterIP      10.152.183.1     <none>        443/TCP                         37d
default         nginx-service                                        LoadBalancer   10.152.183.91    <pending>     80:30130/TCP                    42s
kube-system     kube-dns                                             ClusterIP      10.152.183.10    <none>        53/UDP,53/TCP,9153/TCP          37d
kube-system     kube-prom-stack-kube-prome-coredns                   ClusterIP      None             <none>        9153/TCP                        9d
kube-system     kube-prom-stack-kube-prome-kube-controller-manager   ClusterIP      None             <none>        10257/TCP                       9d
kube-system     kube-prom-stack-kube-prome-kube-etcd                 ClusterIP      None             <none>        2381/TCP                        9d
kube-system     kube-prom-stack-kube-prome-kube-proxy                ClusterIP      None             <none>        10249/TCP                       9d
kube-system     kube-prom-stack-kube-prome-kube-scheduler            ClusterIP      None             <none>        10259/TCP                       9d
kube-system     kube-prom-stack-kube-prome-kubelet                   ClusterIP      None             <none>        10250/TCP,10255/TCP,4194/TCP    37d
kube-system     metrics-server                                       ClusterIP      10.152.183.25    <none>        443/TCP                         37d
observability   alertmanager-operated                                ClusterIP      None             <none>        9093/TCP,9094/TCP,9094/UDP      9d
observability   kube-prom-stack-grafana                              ClusterIP      10.152.183.127   <none>        80/TCP                          9d
```

Conclusion: We have assigned permissions to the `dev` group using the `ClusterRole-dev-group.yaml`. Both vishalk17 and chinu belong to the same group, `dev`, and therefore both have the same permissions accordingly.

7.0: ServiceAccounts:

What are Service Accounts in Kubernetes?

In Kubernetes, a Service Account is a special type of account designed for processes running *inside* your cluster (like Pods). Think of it as the identity card for applications running within your Kubernetes environment.

- **Non-Human:** Service Accounts are not meant for human users.
- **Namespace-Specific:** Each Service Account exists within a specific namespace (a way to organize your Kubernetes resources).
- **Automatic Authentication:** When a Pod uses a Service Account, it automatically gets a token to authenticate with the Kubernetes API server. This lets your application interact with the cluster securely.
- **Permissions Management:** They are essential for controlling what actions your pods are allowed to take within the Kubernetes API (e.g., creating other pods, accessing secrets).

Default Service Account

Every Kubernetes namespace automatically gets a default Service Account named "default". If you don't specify a Service Account when creating a Pod, it'll use this default one. However, for better security and control, it's usually a good practice to create and use specific Service Accounts for different applications.

Use Cases for Kubernetes Service Accounts

- **Pod-to-API Server Communication:** The most common use case is to let your Pods securely talk to the Kubernetes API server to do things like:
  - Get information about other resources (like Services)
  - Create or modify resources
- **Access Control:** You can fine-tune permissions on a Service Account to limit what actions it can take within the cluster (more on this below).
- **Auditing:** Service Accounts provide a way to track which applications are performing which actions within your cluster.

Key Points about Service Accounts

1. **Automatic Creation:** A default Service Account is created automatically in each namespace when you set up a Kubernetes cluster.
2. **Namespace-Bound:** Each Service Account is tied to a specific namespace.
3. **Token-Based Authentication:** Kubernetes issues a token to each Service Account, which pods use to authenticate themselves when interacting with the Kubernetes API.
4. **Role-Based Access Control (RBAC):** You combine Service Accounts with Kubernetes RBAC to define the permissions for pods.
    - **Roles:** Define sets of permissions.
    - **RoleBindings:** Connect Roles to Service Accounts.
5. **Security Best Practice:** Create separate Service Accounts for different applications or services within your cluster to follow the principle of least privilege.
6. **Secrets Access:** Service Accounts can be used to grant access to Kubernetes Secrets, which are a way to store sensitive information like passwords and API keys.

Notes:

ServiceAccounts are namespaced resources, which means they are created within a specific namespace and can only be used within that namespace. Here are the key points regarding ServiceAccounts and their usage across namespaces:

- **Namespace Scope**: Each ServiceAccount is scoped to a specific namespace. This means a ServiceAccount created in one namespace cannot be directly referenced or used in another namespace.

- **Usage in Pods**: When you define a `serviceAccountName` in a Pod spec, Kubernetes looks for that ServiceAccount within the same namespace where the Pod is being created. For example, if you create a Pod in the `default` namespace and specify `serviceAccountName: test-sa`, Kubernetes will look for `test-sa` within the `default` namespace.

- **Multiple ServiceAccounts**: You can create multiple ServiceAccounts with the same name in different namespaces. Each instance of the ServiceAccount is unique within its respective namespace and operates independently of ServiceAccounts with the same name in other namespaces.

7.1: Create & Apply yaml files for service Account :

cluster-role-test-sa.yaml :

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
 name: test-sa-clusterrole
rules:
 - apiGroups: [""]
   resources: ["pods"]
   verbs: ["get", "list", "watch"]
```

cluster-role-test-sa.yaml :

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
 name: test-clusterrolebinding
subjects:
 - kind: ServiceAccount
   name: test-sa    # ServiceAccount Name
   namespace: test  # Where ServiceAccount name : test-sa exit
roleRef:
 kind: ClusterRole
 name: test-sa-clusterrole    # Match with clusterRole name
 apiGroup: rbac.authorization.k8s.io
```

test-serviceAccount.yaml:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: test-sa # ServiceAccount Name
  namespace: test   # namespace where it is going to create
```

```
vishal@vishalk17:~/vishal/rbac/serviceAccounts$ ls
clusterRoleBinding-test-sa.yaml  cluster-role-test-sa.yaml  sample-pod.yaml  test-serviceAccount.yaml

vishal@vishalk17:~/vishal/rbac/serviceAccounts$ kubectl create ns test
namespace/test created

vishal@vishalk17:~/vishal/rbac/serviceAccounts$ kubectl get sa -n test
NAME      SECRETS   AGE
default   0         17s

vishal@vishalk17:~/vishal/rbac/serviceAccounts$ kubectl apply  -f cluster-role-test-sa.yaml
clusterrole.rbac.authorization.k8s.io/test-sa-clusterrole created

vishal@vishalk17:~/vishal/rbac/serviceAccounts$ kubectl apply  -f clusterRoleBinding-test-sa.yaml
clusterrolebinding.rbac.authorization.k8s.io/test-clusterrolebinding created

vishal@vishalk17:~/vishal/rbac/serviceAccounts$ kubectl apply  -f test-serviceAccount.yaml
serviceaccount/test-sa created

vishal@vishalk17:~/vishal/rbac/serviceAccounts$ kubectl get sa -n test
NAME      SECRETS   AGE
default   0         5h7m
test-sa   0         17s   // service account with name test-sa created in test namespace
```

7.2: How to check permission for serviceAccount / user / Group:

```
Check if the user can get pods in the test namespace:

kubectl auth can-i <verb> <resource> --as=<user-name> -n <namespace>

Check if the user associated with the group can get pods in the test namespace:

kubectl auth can-i <verb> <resource> --as=<user-name> --as-group=<group-name> -n <namespace>

Check if a specific service account can get pods in the test namespace:

kubectl auth can-i get pods --as=system:serviceaccount:<created-in-namespace-name>:<service-account-name> -n
<namespace>

Examples :
kubectl auth can-i get pods --as=<user-name> -n <namespace>
kubectl auth can-i get svc --as=<user-name> -n <namespace>
```

`Note:` Above ServiceAccount is associated with a ClusterRole through a ClusterRoleBinding, it inherits cluster-wide permissions defined by that ClusterRole.

Notes:

- **Scope of Permissions**: The permissions granted to a ServiceAccount via a ClusterRoleBinding are indeed cluster-wide unless restricted by other RBAC configurations.
- **RBAC Best Practices**: It's important to follow RBAC best practices by granting the minimum necessary permissions required for your applications or users to operate effectively. Avoid over-privileged roles like cluster-admin unless absolutely necessary.

By correctly configuring ClusterRoleBindings for ServiceAccounts, you ensure that Kubernetes resources can operate with the appropriate level of access control across your cluster.

Test A: Check ServiceAccount : “ test-sa ” has all necessary permission or not, assigned in test-sa-clusterrole.yaml

Notes :

```
Check if a specific service account can get pods in the test namespace:

kubectl auth can-i <verb> <resource> --as=system:serviceaccount:<created-in-namespace-name>:<service-account-name>
-n <namespace>
```

```
Our service account created in namespace: test
Service Account Name:  test-sa
```

Verify:

```
vishal@vishalk17:~/vishal/rbac/serviceAccounts$ kubectl auth can-i get pods --as=system:serviceaccount:test:test-sa -n
kube-system
yes

vishal@vishalk17:~/vishal/rbac/serviceAccounts$ kubectl auth can-i get pods --as=system:serviceaccount:test:test-sa -n test
yes
```

Conclusion:

- The test-sa ServiceAccount has been correctly associated with sufficient permissions through appropriate ClusterRoleBindings or RoleBindings.
- It has permissions (yes) to perform get pods operations in both the kube-system and test namespaces.

Test B: Create a pod using admin user and then try to access it using service account test-sa

sample-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
 name: vishalk17-pod
 Namespace: test  # test-sa is created in test namespace thus we can’t create this pod in another ns
spec:
 serviceAccountName: test-sa  # Assign the service account here  test-sa in test namespace
 containers:
   - name: kubectl-container
     image: bitnami/kubectl
     command: ["sleep"]
     args: ["1800"]  # 1800 seconds = 30 minutes
```

`serviceAccountName: test-sa` : This field specifies the ServiceAccount (`test-sa`) to be used by the Pod.
Namespace: test : test-sa is created in test namespace thus we can’t create this pod in another service account due to service account associated with this deployment

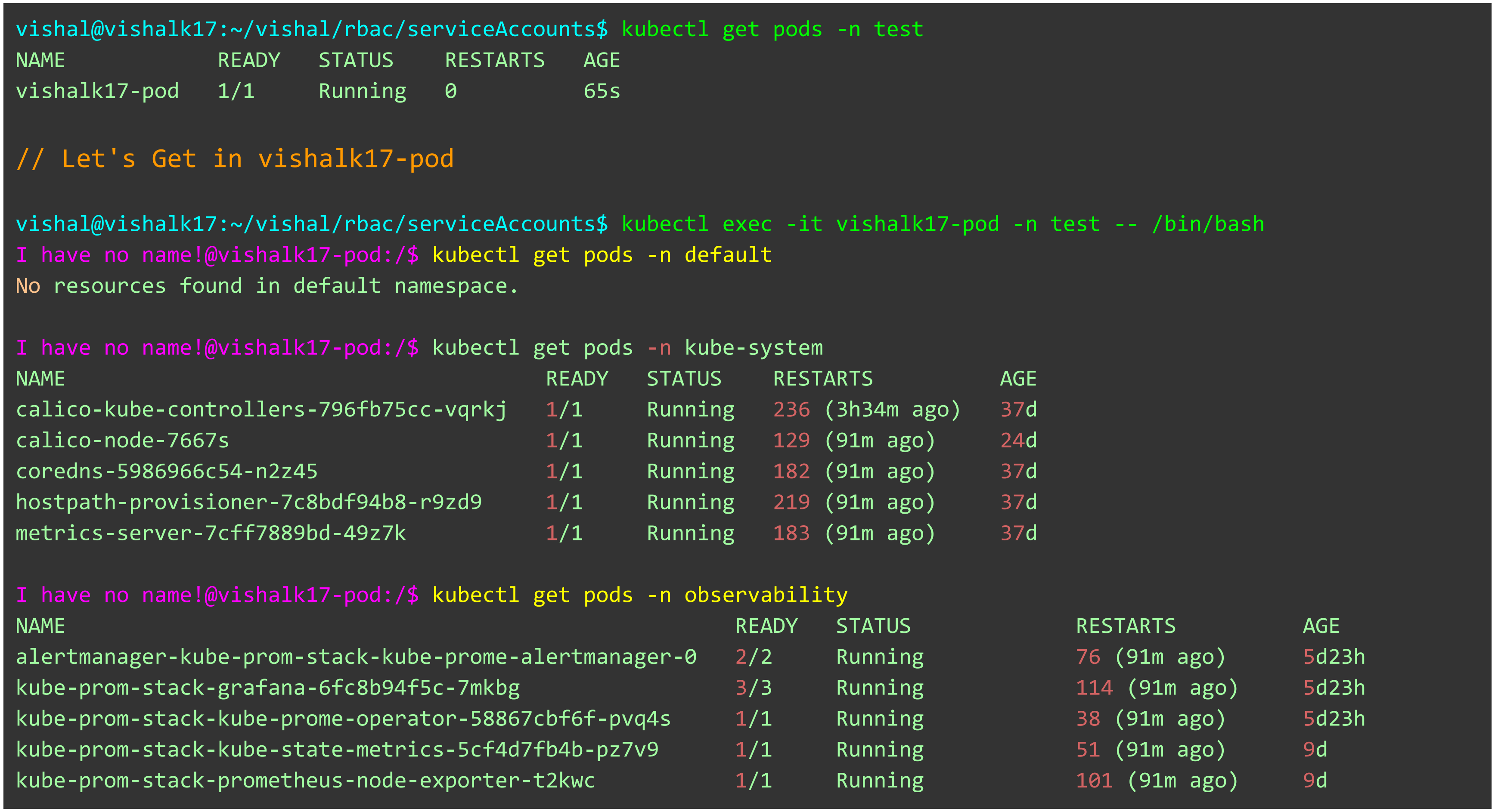
Now switch back to default user , in my case it is microk8s which has access to everything

```
vishal@vishalk17:~/vishal/rbac$ kubectl config use-context microk8s
Switched to context "microk8s".

vishal@vishalk17:~/vishal/rbac$ kubectl config get-contexts
CURRENT   NAME                CLUSTER            AUTHINFO     NAMESPACE
          chinu-context       microk8s-cluster   chinu
*         microk8s            microk8s-cluster   admin
          vishalk17-context   microk8s-cluster   vishalk17    default
```

```
vishal@vishalk17:~/vishal/rbac/serviceAccounts$ kubectl apply -f sample-pod.yaml
pod/vishalk17-pod created

vishal@vishalk17:~/vishal/rbac/serviceAccounts$ kubectl get pods -n test
NAME            READY   STATUS    RESTARTS   AGE
vishalk17-pod   1/1     Running   0          65s
```

Since the Pod `vishalk17-pod` is created in the `test` namespace and associated with the `test-sa` ServiceAccount, which has cluster-wide permissions referenced in `cluster-role-test-sa.yaml`, the pod `vishalk17-pod` has cluster-wide permissions.

Verify:



```
vishal@vishalk17:~/vishal/rbac/serviceAccounts$ kubectl get pods -n test
NAME             READY   STATUS    RESTARTS   AGE
vishalk17-pod    1/1     Running   0          65s

// Let's Get in vishalk17-pod

vishal@vishalk17:~/vishal/rbac/serviceAccounts$ kubectl exec -it vishalk17-pod -n test -- /bin/bash
I have no name!@vishalk17-pod:/$ kubectl get pods -n default
No resources found in default namespace.

I have no name!@vishalk17-pod:/$ kubectl get pods -n kube-system
NAME                                      READY   STATUS    RESTARTS         AGE
calico-kube-controllers-796fb75cc-vqrkj   1/1     Running   236 (3h34m ago)  37d
calico-node-7667s                         1/1     Running   129 (91m ago)    24d
coredns-5986966c54-n2z45                  1/1     Running   182 (91m ago)    37d
hostpath-provisioner-7c8bdf94b8-r9zd9     1/1     Running   219 (91m ago)    37d
metrics-server-7cff7889bd-49z7k           1/1     Running   183 (91m ago)    37d

I have no name!@vishalk17-pod:/$ kubectl get pods -n observability
NAME                                                   READY   STATUS             RESTARTS        AGE
alertmanager-kube-prom-stack-kube-prome-alertmanager-0 2/2     Running            76 (91m ago)    5d23h
kube-prom-stack-grafana-6fc8b94f5c-7mkbg               3/3     Running            114 (91m ago)   5d23h
kube-prom-stack-kube-prome-operator-58867cbf6f-pvq4s   1/1     Running            38 (91m ago)    5d23h
kube-prom-stack-kube-state-metrics-5cf4d7fb4b-pz7v9    1/1     Running            51 (91m ago)    9d
kube-prom-stack-prometheus-node-exporter-t2kwc         1/1     Running            101 (91m ago)   9d
```

Conclusion :

- A Pod must be created in the same namespace where the ServiceAccount has been created; otherwise, the Pod will not be created.
- A Pod created with an associated ServiceAccount inherits all the permissions associated with that ServiceAccount.

References & SourceCode :

Source Code : https://github.com/vishalk17/devops/tree/main/kubernetes/2-after-k8s-learn/rbac