Identity and Access Management

**[Identity and Access Management](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management" \l "identity-and-access-management)**

[AWS Identity and Access Management (IAM)](https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html) is an AWS service that performs two essential functions: Authentication and Authorization. Authentication involves the verification of a identity whereas authorization governs the actions that can be performed by AWS resources. Within AWS, a resource can be another AWS service, e.g. EC2, or an AWS [principal](https://docs.aws.amazon.com/IAM/latest/UserGuide/intro-structure.html#intro-structure-principal) such as an [IAM User](https://docs.aws.amazon.com/IAM/latest/UserGuide/id.html#id_iam-users) or [Role](https://docs.aws.amazon.com/IAM/latest/UserGuide/id.html#id_iam-roles). The rules governing the actions that a resource is allowed to perform are expressed as IAM [policies](https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html).

**[Controlling Access to Amazon EKS Clusters](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management" \l "controlling-access-to-amazon-eks-clusters)**

The Kubernetes project supports a variety of different strategies to authenticate requests to the kube-apiserver service, e.g. Bearer Tokens, X.509 certificates, OIDC, etc. Amazon EKS currently has native support for [webhook token authentication](https://kubernetes.io/docs/reference/access-authn-authz/authentication/#webhook-token-authentication), [service account tokens](https://kubernetes.io/docs/reference/access-authn-authz/authentication/#service-account-tokens), and as of February 21, 2021, OIDC authentication.

The webhook authentication strategy calls a webhook that verifies bearer tokens. On Amazon EKS, these bearer tokens are generated by the AWS CLI or the [aws-iam-authenticator](https://github.com/kubernetes-sigs/aws-iam-authenticator) client when you run kubectl commands. As you execute commands, the token is passed to the kube-apiserver which forwards it to the authentication webhook. If the request is well-formed, the webhook calls a pre-signed URL embedded in the token's body. This URL validates the request's signature and returns information about the user, e.g. the user's account, Arn, and UserId to the kube-apiserver.

Each token starts with k8s-aws-v1. followed by a base64 encoded string.To manually generate a authentication token, type the following command in a terminal window.

The string, when decoded, should resemble this.

TOKEN\_DATA=$(aws eks get-token --cluster-name eksworkshop-eksctl | jq -r '.status.token')

echo $TOKEN\_DATA

IFS='.' read -r -a array <<< "$TOKEN\_DATA"

echo "${array[1]}"==== | fold -w 4 | sed '$ d' | tr -d '\n' | base64 --decode

The output looks like something like this.

Expand to see Sample Output

<https://sts.us-west-2.amazonaws.com/?Action=GetCallerIdentity&Version=2011-06-15&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=ASIAQAHCJ2QPBFJYF6M5%2F20230228%2Fus-west-2%2Fsts%2Faws4_request&X-Amz-Date=20230228T075742Z&X-Amz-Expires=60&X-Amz-SignedHeaders=host%3Bx-k8s-aws-id&X-Amz-Security-Token=IQoJb3JpZ2luX2VjEMj%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaCXVzLWVhc3QtMSJHMEUCIQDG2ef8SQiWF04CqAz4u3JLQWs%2Bhfk7si8jVKQwbGRfDAIgChTi5my0BugBetYDzW4P%2B8nD6CLiSKzMMjG2bUZ6ankqsgUIcBADGgwwMDA0NzQ2MDA0NzgiDFh6XvFedD%2BwWObAzCqPBeAKIDUDWZN1rvrRwapBl7eFVUJJL%2FWLtnFY%2FNmMl6cb9HgSvP%2BeRZpX%2FLOkhNT2MYSEo%2BPKJ2%2FQf0%2FrGfLOQ1JS52rtKuobXFOTLnB%2BBFoal%2B%2FJmEGqh9z71FRCYHAVEhKST3F2N1JAT7BHV3vyt9QaSz%2BmdreVclJkyFAVqxXxdySVZHtZdnwetJ0YbU9dxI0Neuilu5XBR79owkwJ3cdDgz4tYSzC7JcOmrsvb2IQFLhFlZyWercXZICSYEZpTLwuFNMN4nci7B4mrSTfz5bZheYOQ%2F2vdu3kq1j2CapDrIpgpLYhe1slCSBetLPu6ZdMp8nIsfMwOjtSrqGgrLxKlP2wNdGwUE4EwODXJOM%2FeJHqooGFBpScle%2FLCg9Et4PCrnPaJ7zxltWtcT1l7rWOnlXVTWTC17%2FG3K5qlkWbdJMt40xXGzfdxP5Mf0ykmMQ1%2BfHnsgFoiRDTcHTBAHbgavy27kUI8%2FCUk6WSZQeLKTQplI6GQBAfRtMI38H2Eh1iugsvRWlNaCTHpTLjYZ7vkzFrgqXK6YlulCJ6dwzRqzTrK9UZn8EzEsphuEejxwOJ2Rkc7FSwH1%2BNo4KHJgbe5hVv3xOpWxDqWjvet3c5056MtbJVxhm2N0Z1ZaYYNdEdv%2Bvfkyy7c9Bd6fOsCPIYUHiOMmvGNYrAbqQgp0VC7U4%2FVO1%2BYQ0N6u4Arjns98MVrhOOfcl0%2BMbQkSrs8m6Z3OBq1c%2FYL1BTMKMgTwGpmOzOnmbxna58M%2F6T3ZD2%2F03g%2BcKx7dxipEGaC08rzggJI43EPLD5%2BD%2FfDjxWDCPevBPBztU1HSYLdJFMnXkWeHwzDor75Vit4hgjAWzRrU%2F5bgfHeohclJpLtvHDhTAws9X2nwY6sgHfBDGEhFkvcFmnasbtv%2BuUBWtFBMgSPdrwy0SebbkxhbtPeozsM21w8yPbDQgQWq0ETNVDx1sHrw%2F0eF6GX8NOYsKNM6xD3y7Y7TSLVIcsdnjms6VUsnl1VHMiUsO196%2B8m3NdMYwkMFwxiqbfOsX6YaxavuFxH37%2BDL28rSiMXHdT%2BNc2VJFO2HPMnllfISh1FpfPeKmogjpwreQfEiydSunHrUCwy1qW5FL3YyYpAN89&X-Amz-Signature=4b0e9fc4867a0a9f9c0044e9f167ae2a1e64e192a33e622e578a575098e52092>

The token consists of a pre-signed URL that includes an Amazon credential and signature. For additional details see <https://docs.aws.amazon.com/STS/latest/APIReference/API_GetCallerIdentity.html>.

The token has a time to live (TTL) of 15 minutes after which a new token will need to be generated. This is handled automatically when you use a client like kubectl, however, if you're using the Kubernetes dashboard, you will need to generate a new token and re-authenticate each time the token expires.

Once the user's identity has been authenticated by the AWS IAM service, the kube-apiserver reads the aws-auth ConfigMap in the kube-system Namespace to determine the RBAC group to associate with the user. The aws-auth ConfigMap is used to create a static mapping between IAM principals, i.e. IAM Users and Roles, and Kubernetes RBAC groups. RBAC groups can be referenced in Kubernetes RoleBindings or ClusterRoleBindings. They are similar to IAM Roles in that they define a set of actions (verbs) that can be performed against a collection of Kubernetes resources (objects).

Using AWS IAM Groups and Roles to Manage Kubernetes Cluster Access

In this module, we’ll learn about how to simplify access to different parts of the kubernetes clusters depending on AWS IAM Roles.

When an Amazon EKS cluster is created, the IAM entity (user or role) that creates the cluster is permanently added to the Kubernetes RBAC authorization table as the administrator. This entity will be automatically part of the Kubernetes RBAC group called **system:masters** and gets assigned to the [Kubernetes Default ClusterRole](https://kubernetes.io/docs/reference/access-authn-authz/rbac/) called **cluster-admin**. The ClusterRole **cluster-admin** is the most powerful role and allows super-user access to perform any action on any resource. When used in a ClusterRoleBinding, it gives full control over every resource in the cluster and in all namespaces. When used in a RoleBinding, it gives full control over every resource in the role binding's namespace, including the namespace itself.

**WARNING**

It is highly recommended not to add any Kubernetes user to **system:masters** group unless it is necessary

The identity of this entity isn't visible in your cluster configuration. So, it's important to note the entity that created the cluster and make sure that you never delete it.

Initially, only the IAM entity that created the server can make calls to the Kubernetes API server using kubectl. If you use the console to create the cluster, you must ensure that the same IAM credentials are in the AWS SDK credential chain when you run kubectl commands on your cluster. After your cluster is created, you can grant other IAM entities access to your cluster

Let us check the IAM role assigned to the Cloud 9 Instance.

1

2

3

4

C9\_ROLE\_ARN=$(aws sts get-caller-identity --query Arn)

IFS='/' read -r -a array <<< "$C9\_ROLE\_ARN"

C9\_ROLE="${array[1]}"

echo "$C9\_ROLE"

Check Output if you running on your own

Check Output if you running at AWS Event

Note the IAM role eksworkshop-admin or eks-bootstrap-template-ws-Cloud9InstanceRole-V1RKIVUA1ZM0 in the above output is used to create and authenticate the Amazon EKS Cluster.

Let us check if this IAM role is part of the aws-auth config map in kube-system namespace.

1

2

3

4

5

6

7

8

9

DOES\_ROLE\_EXISTS\_IN\_CONFIGMAP=$(kubectl get cm aws-auth -n kube-system -oyaml | grep $C9\_ROLE)

echo $DOES\_ROLE\_EXISTS\_IN\_CONFIGMAP

if [ -z "$DOES\_ROLE\_EXISTS\_IN\_CONFIGMAP" ]

then

echo "$C9\_ROLE doesn't exist in aws-auth config map in kube-system namespace"

else

echo "$C9\_ROLE exists in aws-auth config map in kube-system namespace"

fi

Check output

eks-bootstrap-template-ws-Cloud9InstanceRole-V1RKIVUA1ZM0 doesn't exist in aws-auth configmap in kube-system namespace

Note that the above IAM Role used to EKS cluster doesn't exist in aws-auth configmap, which is expected.

It is recommended to create a dedicated IAM role to create the EKS cluster with Least privileged IAM permissions to be able to perform CRUD operations on the EKS clusters. Howeever, the above IAM Role eks-bootstrap-template-ws-Cloud9InstanceRole-V1RKIVUA1ZM0 has AdministratorAccessIAM permission for the lab purpose.

# Kubernetes Authentication

According to [the official kubernetes docs:](https://kubernetes.io/docs/reference/access-authn-authz/rbac/)

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

The core logical components of RBAC are:

**Entity**  
A group, user, or Service Account (an identity representing an application that wants to execute certain operations (actions) and requires permissions to do so).

**Resource**  
A Pod, Service, or Secret that the entity wants to access using the certain operations.

**Role and ClusterRole**  
An RBAC Role or ClusterRole contains rules that represent a set of permissions. Permissions are purely additive (there are no "deny" rules).

A Role always sets permissions within a particular Namespace; when you create a Role, you have to specify the Namespace it belongs in.

ClusterRole, by contrast, is a non-namespaced resource. The resources have different names (Role and ClusterRole) because a Kubernetes object always has to be either namespaced or not namespaced; it can't be both.

**RoleBinding and ClusterRoleBinding**  
A RoleBinding grants the permissions defined in a role to a user or set of users. It holds a list of subjects (users, groups, or Service Accounts), and a reference to the role being granted. A RoleBinding grants permissions within a specific Namespace whereas a ClusterRoleBinding grants that access cluster-wide.

A RoleBinding may reference any Rolein the same Namespace. Alternatively, a RoleBindingcan reference a ClusterRole and bind that ClusterRole to the 'Namespaceof theRoleBinding. If you want to bind a ClusterRole to all the Namespaces in your cluster, you use a ClusterRoleBinding.

**Namespace**

Namespaces are an excellent way of creating security boundaries, they also provide a unique scope for object names as the Namespace name implies. They are intended to be used in multi-tenant environments to create virtual kubernetes clusters on the same physical cluster.

If you have different teams which needs different kind of cluster access, it would be difficult to manually add or remove access for each Amazon EKS clusters you want them to give or remove access from.

We can leverage on AWS [IAM Groups](https://docs.aws.amazon.com/IAM/latest/UserGuide/id_groups.html) to easily add or remove users and give them permission to whole cluster, or just part of it depending on which groups they belong to.

In this lesson, we will create **3 IAM roles** that we will map to **3 IAM groups**.

Create AWS IAM Roles

Let us create 3 least privileged IAM Roles

We are going to create 3 roles:

* a **k8sAdmin** role which will have **admin** rights in our Amazon EKS cluster
* a **k8sDev** role which will give access to the **developers** namespace in our Amazon EKS cluster
* a **k8sInteg** role which will give access to the **integration** namespace in our Amazon EKS cluster

Set below environment variables

1

2

export ACCOUNT\_ID=$(aws sts get-caller-identity --output text --query Account)

export AWS\_REGION=$(curl -s 169.254.169.254/latest/dynamic/instance-identity/document | jq -r '.region')

Create the IAM Roles:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

POLICY=$(echo -n '{"Version":"2012-10-17","Statement":[{"Effect":"Allow","Principal":{"AWS":"arn:aws:iam::'; echo -n "$ACCOUNT\_ID"; echo -n ':root"},"Action":"sts:AssumeRole","Condition":{}}]}')

export IAM\_ROLE="k8sAdmin"

export ROLE\_DESCRIPTION="Kubernetes administrator role (for AWS IAM Authenticator for Kubernetes)."

export IAM\_ROLE\_ARN=$(aws iam get-role --role-name $IAM\_ROLE | jq -r '.Role.Arn')

if [ -z "$IAM\_ROLE\_ARN" ]

then

IAM\_ROLE\_ARN=$(aws iam create-role \

--role-name $IAM\_ROLE \

--description "$ROLE\_DESCRIPTION" \

--assume-role-policy-document "$POLICY" \

--output text \

--query 'Role.Arn')

echo "IAM Role ${IAM\_ROLE} created. IAM\_ROLE\_ARN=$IAM\_ROLE\_ARN"

else

echo "IAM Role ${IAM\_ROLE} already exist..."

fi

export IAM\_ROLE="k8sDev"

export ROLE\_DESCRIPTION="Kubernetes developer role (for AWS IAM Authenticator for Kubernetes)."

export IAM\_ROLE\_ARN=$(aws iam get-role --role-name $IAM\_ROLE | jq -r '.Role.Arn')

if [ -z "$IAM\_ROLE\_ARN" ]

then

IAM\_ROLE\_ARN=$(aws iam create-role \

--role-name $IAM\_ROLE \

--description "$ROLE\_DESCRIPTION" \

--assume-role-policy-document "$POLICY" \

--output text \

--query 'Role.Arn')

echo "IAM Role ${IAM\_ROLE} created. IAM\_ROLE\_ARN=$IAM\_ROLE\_ARN"

else

echo "IAM Role ${IAM\_ROLE} already exist..."

fi

export IAM\_ROLE="k8sInteg"

export ROLE\_DESCRIPTION="Kubernetes role for integration namespace in quick cluster."

export IAM\_ROLE\_ARN=$(aws iam get-role --role-name $IAM\_ROLE | jq -r '.Role.Arn')

if [ -z "$IAM\_ROLE\_ARN" ]

then

IAM\_ROLE\_ARN=$(aws iam create-role \

--role-name $IAM\_ROLE \

--description "$ROLE\_DESCRIPTION" \

--assume-role-policy-document "$POLICY" \

--output text \

--query 'Role.Arn')

echo "IAM Role ${IAM\_ROLE} created. IAM\_ROLE\_ARN=$IAM\_ROLE\_ARN"

else

echo "IAM Role ${IAM\_ROLE} already exist..."

fi

Check Output

1

2

3

IAM Role k8sAdmin created. IAM\_ROLE\_ARN=arn:aws:iam::ACCOUNT\_ID:role/k8sAdmin

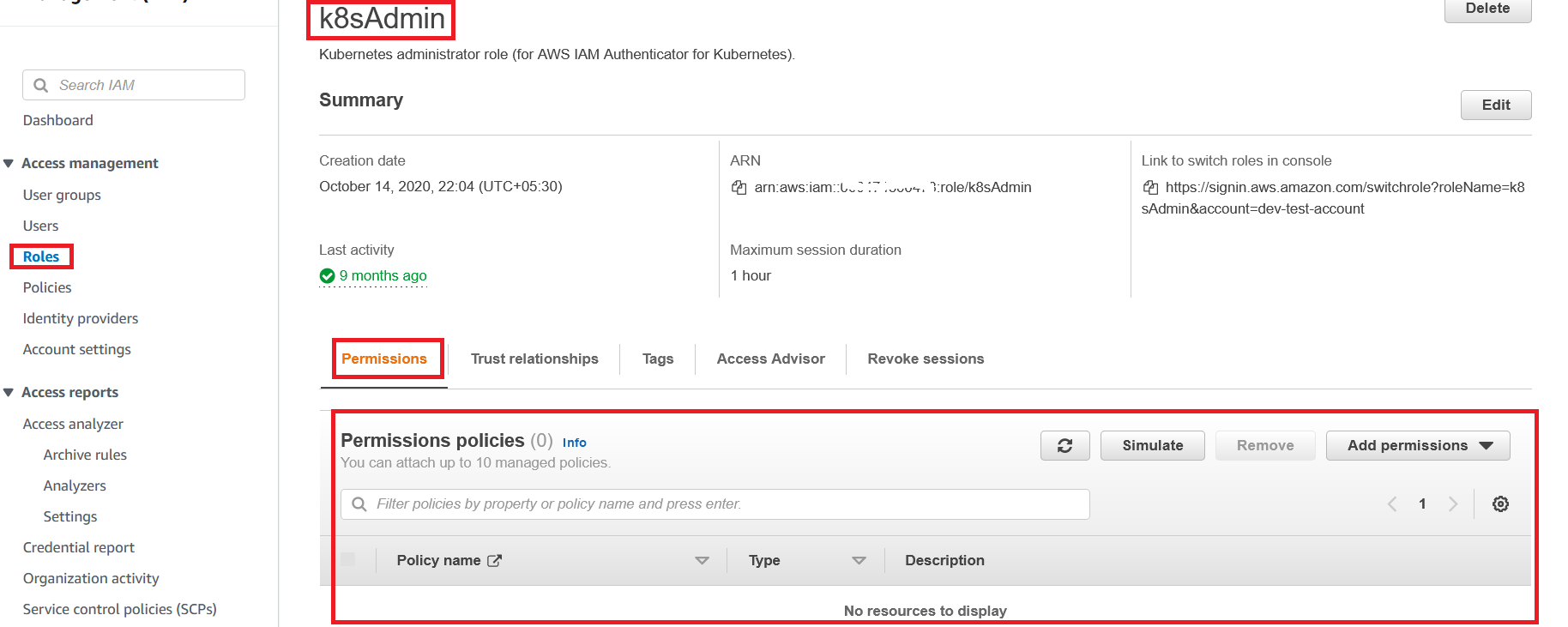
IAM Role k8sDev created. IAM\_ROLE\_ARN=arn:aws:iam::ACCOUNT\_ID:role/k8sDev

IAM Role k8sInteg created. IAM\_ROLE\_ARN=arn:aws:iam::ACCOUNT\_ID:role/k8sInteg

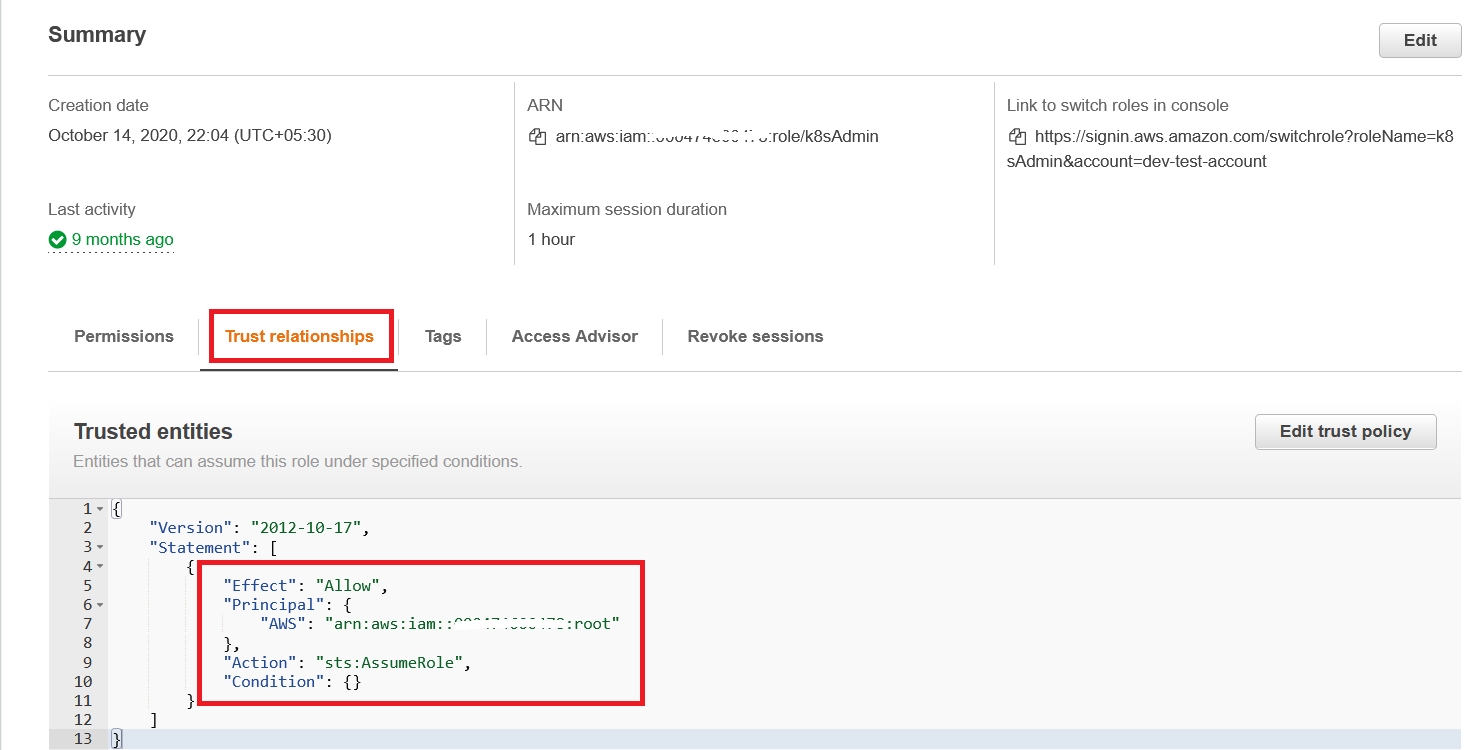
*In this example, the assume-role-policy allows the root account to assume the role. We are going to allow specific groups to also be able to assume those roles. Check the*[official documentation](https://docs.aws.amazon.com/eks/latest/userguide/iam-roles-for-service-accounts-technical-overview.html)*for more information.*

Because the above roles are only used to authenticate within our Amazon EKS cluster, they don't need to have AWS permissions. We will only use them to allow some IAM groups to assume this role in order to have access to our EKS cluster.

Let's go to the AWS IAM Console and check one of the above IAM Role and see that there are no IAM permissions attached to the Role.



And also let's see trust policy of the IAM Role that allows the root account to assume the role, which means any IAM principal (user or role) can now assume the role.



Create AWS IAM Groups

In this section let's create 3 IAM groups and attach IAM permission policy on these IAM groups to be assume the IAM roles created earlier for Kubernetes role.

We want to have different IAM users which will be added to specific IAM groups in order to have different rights in the kubernetes cluster.

We will define 3 groups:

* **k8sAdmin** - users from this group will have admin rights on the kubernetes cluster
* **k8sDev** - users from this group will have full access only in the development namespace of the cluster
* **k8sInteg** - users from this group will have access to integration namespace.

*In fact, users from****k8sDev****and****k8sInteg****groups will only have access to namespaces where we will define kubernetes RBAC access for their associated kubernetes role. We'll see this but first, let's create the groups.*

**[Create k8sAdmin IAM Group](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/3-create-iam-groups" \l "create-k8sadmin-iam-group)**

The **k8sAdmin** Group will be allowed to assume the **k8sAdmin** IAM Role.

1

2

3

4

5

6

7

8

9

10

IAM\_GROUP="k8sAdmin"

export IAM\_GROUP\_ARN=$(aws iam get-group --group-name $IAM\_GROUP | jq -r '.Group.Arn')

if [ -z "$IAM\_GROUP\_ARN" ]

then

IAM\_GROUP\_ARN=$(aws iam create-group --group-name $IAM\_GROUP | jq -r '.Group.Arn')

echo "IAM Group ${IAM\_GROUP} created. IAM\_GROUP\_ARN=$IAM\_GROUP\_ARN"

else

echo "IAM Group ${IAM\_GROUP} already exist..."

fi

Check Output

Let's add a Policy on our group which will allow users from this group to assume our k8sAdmin Role:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

ADMIN\_GROUP\_POLICY=$(echo -n '{

"Version": "2012-10-17",

"Statement": [

{

"Sid": "AllowAssumeOrganizationAccountRole",

"Effect": "Allow",

"Action": "sts:AssumeRole",

"Resource": "arn:aws:iam::'; echo -n "$ACCOUNT\_ID"; echo -n ':role/k8sAdmin"

}

]

}')

echo ADMIN\_GROUP\_POLICY=$ADMIN\_GROUP\_POLICY

aws iam put-group-policy \

--group-name k8sAdmin \

--policy-name k8sAdmin-policy \

--policy-document "$ADMIN\_GROUP\_POLICY"

**[Create k8sDev IAM Group](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/3-create-iam-groups" \l "create-k8sdev-iam-group)**

The **k8sDev** Group will be allowed to assume the **k8sDev** IAM Role.

1

2

3

4

5

6

7

8

9

10

IAM\_GROUP="k8sDev"

export IAM\_GROUP\_ARN=$(aws iam get-group --group-name $IAM\_GROUP | jq -r '.Group.Arn')

if [ -z "$IAM\_GROUP\_ARN" ]

then

IAM\_GROUP\_ARN=$(aws iam create-group --group-name $IAM\_GROUP | jq -r '.Group.Arn')

echo "IAM Group ${IAM\_GROUP} created. IAM\_GROUP\_ARN=$IAM\_GROUP\_ARN"

else

echo "IAM Group ${IAM\_GROUP} already exist..."

fi

Check Output

Let's add a Policy on our group which will allow users from this group to assume our k8sDev Role:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

DEV\_GROUP\_POLICY=$(echo -n '{

"Version": "2012-10-17",

"Statement": [

{

"Sid": "AllowAssumeOrganizationAccountRole",

"Effect": "Allow",

"Action": "sts:AssumeRole",

"Resource": "arn:aws:iam::'; echo -n "$ACCOUNT\_ID"; echo -n ':role/k8sDev"

}

]

}')

echo DEV\_GROUP\_POLICY=$DEV\_GROUP\_POLICY

aws iam put-group-policy \

--group-name k8sDev \

--policy-name k8sDev-policy \

--policy-document "$DEV\_GROUP\_POLICY"

**[Create k8sInteg IAM Group](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/3-create-iam-groups" \l "create-k8sinteg-iam-group)**

1

2

3

4

5

6

7

8

9

10

IAM\_GROUP="k8sInteg"

export IAM\_GROUP\_ARN=$(aws iam get-group --group-name $IAM\_GROUP | jq -r '.Group.Arn')

if [ -z "$IAM\_GROUP\_ARN" ]

then

IAM\_GROUP\_ARN=$(aws iam create-group --group-name $IAM\_GROUP | jq -r '.Group.Arn')

echo "IAM Group ${IAM\_GROUP} created. IAM\_GROUP\_ARN=$IAM\_GROUP\_ARN"

else

echo "IAM Group ${IAM\_GROUP} already exist..."

fi

Check Output

Let's add a Policy on our group which will allow users from this group to assume our k8sInteg Role:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

INTEG\_GROUP\_POLICY=$(echo -n '{

"Version": "2012-10-17",

"Statement": [

{

"Sid": "AllowAssumeOrganizationAccountRole",

"Effect": "Allow",

"Action": "sts:AssumeRole",

"Resource": "arn:aws:iam::'; echo -n "$ACCOUNT\_ID"; echo -n ':role/k8sInteg"

}

]

}')

echo INTEG\_GROUP\_POLICY=$INTEG\_GROUP\_POLICY

aws iam put-group-policy \

--group-name k8sInteg \

--policy-name k8sInteg-policy \

--policy-document "$INTEG\_GROUP\_POLICY"

You now should have your 3 groups

1

aws iam list-groups

The output will look like below.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

{

"Groups": [

{

"Path": "/",

"GroupName": "k8sAdmin",

"GroupId": "AGPAZRV3OHPJZGT2JKVDV",

"Arn": "arn:aws:iam::ACCOUNT\_ID:group/k8sAdmin",

"CreateDate": "2020-04-07T13:32:52Z"

},

{

"Path": "/",

"GroupName": "k8sDev",

"GroupId": "AGPAZRV3OHPJUOBR375KI",

"Arn": "arn:aws:iam::ACCOUNT\_ID:group/k8sDev",

"CreateDate": "2020-04-07T13:33:15Z"

},

{

"Path": "/",

"GroupName": "k8sInteg",

"GroupId": "AGPAZRV3OHPJR6GM6PFDG",

"Arn": "arn:aws:iam::ACCOUNT\_ID:group/k8sInteg",

"CreateDate": "2020-04-07T13:33:25Z"

}

]

}

# Create AWS IAM Users

In order to test our scenarios, we will create 3 users, one for each groups we created :

1

2

3

4

5

6

7

8

9

10

11

12

IAM\_USERS=("PaulAdmin" "JeanDev" "PierreInteg")

for IAM\_USER in ${IAM\_USERS[@]}; do

export IAM\_USER\_ARN=$(aws iam get-user --user-name $IAM\_USER | jq -r '.User.Arn')

if [ -z "$IAM\_USER\_ARN" ]

then

IAM\_USER\_ARN=$(aws iam create-user --user-name $IAM\_USER | jq -r '.User.Arn')

echo "IAM User ${IAM\_USER} created. IAM\_USER\_ARN=$IAM\_USER\_ARN"

else

echo "IAM User ${IAM\_USER} already exist..."

fi

done

Check Output

1

2

3

IAM User PaulAdmin created. IAM\_USER\_ARN=arn:aws:iam::ACCOUNT\_ID:user/PaulAdmin

IAM User JeanDev created. IAM\_USER\_ARN=arn:aws:iam::ACCOUNT\_ID:user/JeanDev

IAM User PierreInteg created. IAM\_USER\_ARN=arn:aws:iam::ACCOUNT\_ID:user/PierreInteg

Add users to associated groups:

1

2

3

aws iam add-user-to-group --group-name k8sAdmin --user-name PaulAdmin

aws iam add-user-to-group --group-name k8sDev --user-name JeanDev

aws iam add-user-to-group --group-name k8sInteg --user-name PierreInteg

Check users are correctly added in their groups:

1

2

3

aws iam get-group --group-name k8sAdmin

aws iam get-group --group-name k8sDev

aws iam get-group --group-name k8sInteg

Check Output

**Note** For the sake of simplicity, in this chapter, we will save credentials to a file to make it easy to toggle back and forth between users. Never do this in production or with credentials that have priviledged access; It is not a security best practice to store credentials on the filesystem.

Retrieve Access Keys for our fake users:

1

2

3

aws iam create-access-key --user-name PaulAdmin | tee /tmp/PaulAdmin.json

aws iam create-access-key --user-name JeanDev | tee /tmp/JeanDev.json

aws iam create-access-key --user-name PierreInteg | tee /tmp/PierreInteg.json

Check Output

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

{

"AccessKey": {

"UserName": "PaulAdmin",

"AccessKeyId": "XXXXXXXXX",

"Status": "Active",

"SecretAccessKey": "XXXXXXXX",

"CreateDate": "2023-03-14T09:52:30+00:00"

}

}

{

"AccessKey": {

"UserName": "JeanDev",

"AccessKeyId": "XXXXXXX",

"Status": "Active",

"SecretAccessKey": "XXXXXXXX",

"CreateDate": "2023-03-14T09:52:31+00:00"

}

}

{

"AccessKey": {

"UserName": "PierreInteg",

"AccessKeyId": "XXXXXXX",

"Status": "Active",

"SecretAccessKey": "XXXXXXX",

"CreateDate": "2023-03-14T09:52:32+00:00"

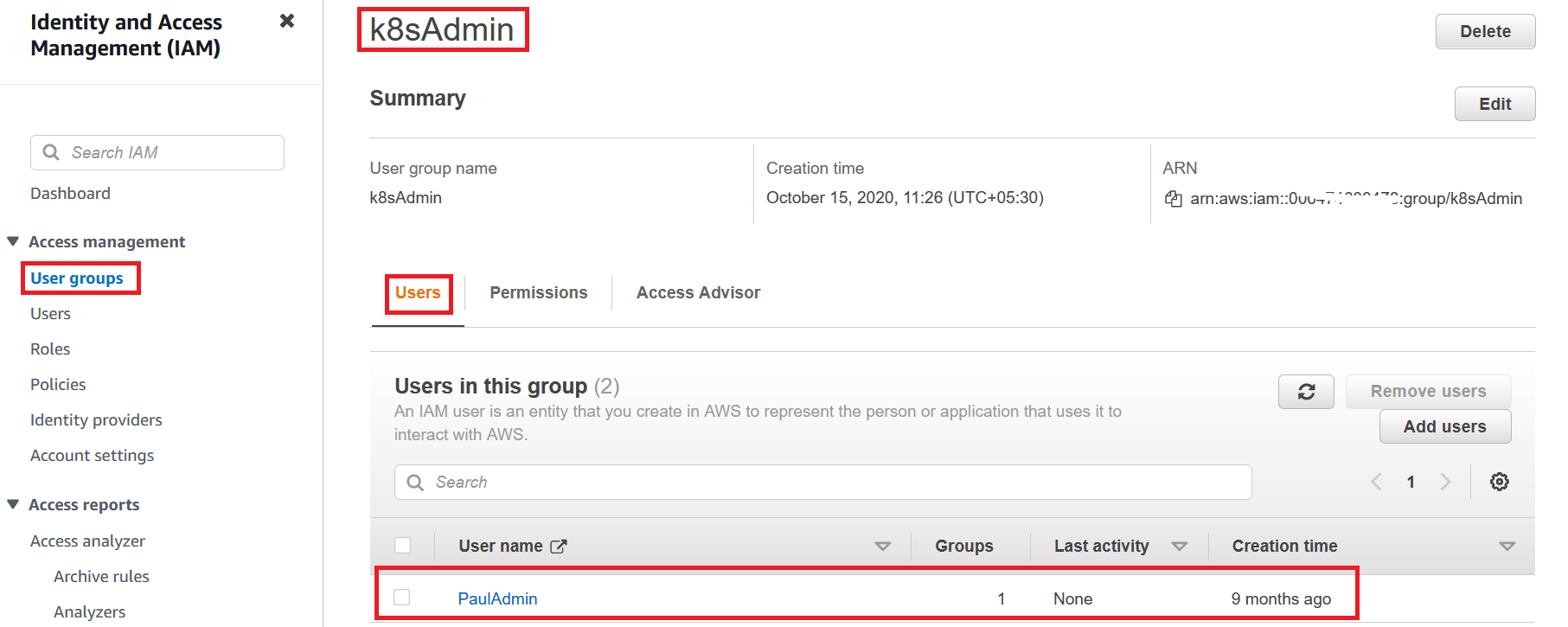
}

}

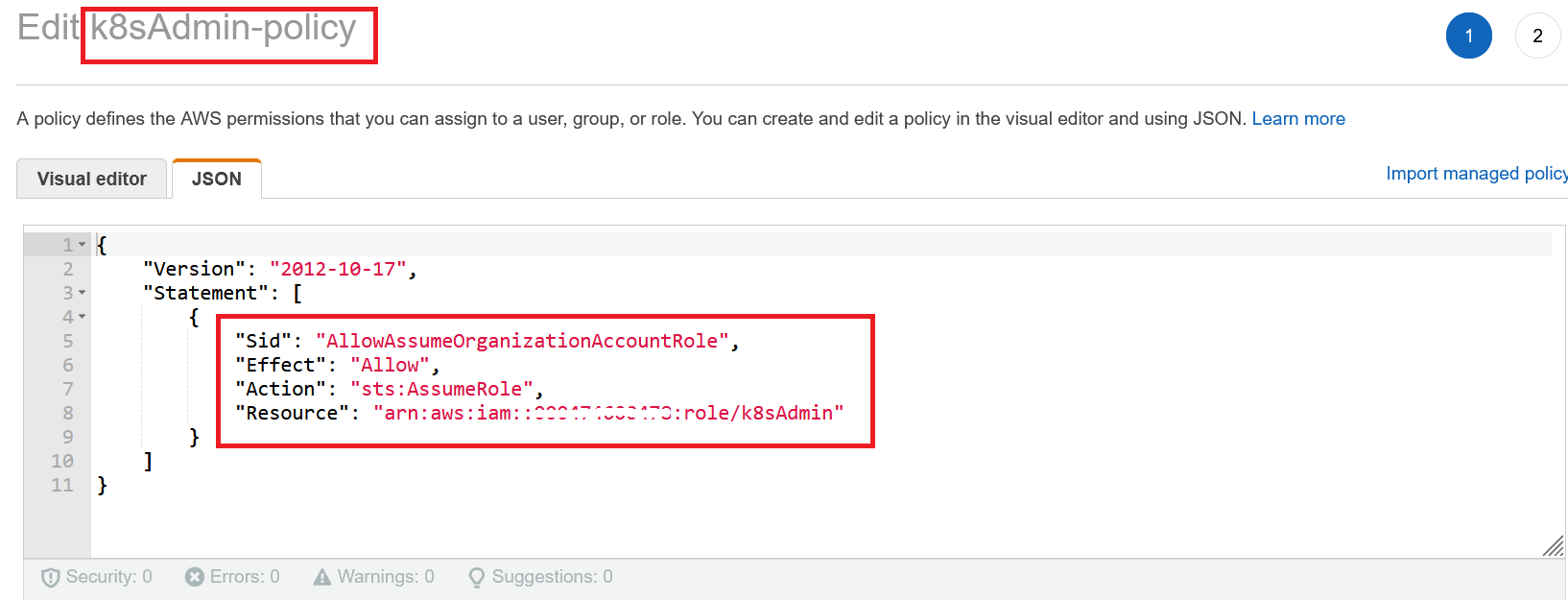
Recap:

* **PaulAdmin** is in the **k8sAdmin** group and will be able to assume the **k8sAdmin** role.
* **JeanDev** is in **k8sDev** Group and will be able to assume IAM role **k8sDev**
* **PierreInteg** is in **k8sInteg** group and will be able to assume IAM role **k8sInteg**

Let's go to the [AWS IAM Console](https://console.aws.amazon.com/iamv2/home#/home) and check one of the above IAM Groups and see that there are IAM users part of the group.



And also let's see trust policy of the IAM Group that allows users from this group to assume an IAM Role:



# Configure Kubernetes RBAC Roles

So far we created **IAM Roles/Groups/Users** in AWS. In this section, let's create corresponding Kubernetes RBAC objects.

#### [Create kubernetes namespaces](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/5-configure-k8s-rbac" \l "create-kubernetes-namespaces)

* **development** namespace will be accessible for IAM users from **k8sDev** group
* **integration** namespace will be accessible for IAM users from **k8sInteg** group

1

2

kubectl create namespace integration

kubectl create namespace development

Expand for Output

1

2

namespace/integration created

namespace/development created

#### [Configuring access to development namespace](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/5-configure-k8s-rbac" \l "configuring-access-to-development-namespace)

We create a kubernetes role and rolebinding in the development namespace giving full access to the kubernetes user **dev-user**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

cat << EOF | kubectl apply -f - -n development

kind: Role

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: dev-role

rules:

- apiGroups:

- ""

- "apps"

- "batch"

- "extensions"

resources:

- "configmaps"

- "cronjobs"

- "deployments"

- "events"

- "ingresses"

- "jobs"

- "pods"

- "pods/attach"

- "pods/exec"

- "pods/log"

- "pods/portforward"

- "secrets"

- "services"

verbs:

- "create"

- "delete"

- "describe"

- "get"

- "list"

- "patch"

- "update"

---

kind: RoleBinding

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: dev-role-binding

subjects:

- kind: User

name: dev-user

roleRef:

kind: Role

name: dev-role

apiGroup: rbac.authorization.k8s.io

EOF

Expand for Output

1

2

role.rbac.authorization.k8s.io/dev-role created

rolebinding.rbac.authorization.k8s.io/dev-role-binding created

The role we define will give full access to everything in that namespace. It is a Role, and not a ClusterRole, so it is going to be applied only in the **development** namespace.

#### [Configuring access to integration namespace](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/5-configure-k8s-rbac" \l "configuring-access-to-integration-namespace)

We create a kubernetes role and rolebinding in the integration namespace for full access with the kubernetes user **integ-user**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

cat << EOF | kubectl apply -f - -n integration

kind: Role

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: integ-role

rules:

- apiGroups:

- ""

- "apps"

- "batch"

- "extensions"

resources:

- "configmaps"

- "cronjobs"

- "deployments"

- "events"

- "ingresses"

- "jobs"

- "pods"

- "pods/attach"

- "pods/exec"

- "pods/log"

- "pods/portforward"

- "secrets"

- "services"

verbs:

- "create"

- "delete"

- "describe"

- "get"

- "list"

- "patch"

- "update"

---

kind: RoleBinding

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: integ-role-binding

subjects:

- kind: User

name: integ-user

roleRef:

kind: Role

name: integ-role

apiGroup: rbac.authorization.k8s.io

EOF

Expand for Output

1

2

role.rbac.authorization.k8s.io/integ-role created

rolebinding.rbac.authorization.k8s.io/integ-role-binding created

The role we define will give full access to everything in that namespace. It is a Role, and not a ClusterRole, so it is going to be applied only in the **integration** namespace.

# Configure aws-auth configmap

In this section, we will configure aws-auth configmap for mapping between IAM Role(i.e. Kubernetes User) to Kubernetes RBAC Role.

#### [Gives Access to our IAM Roles to Amazon EKS Cluster](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/6-configure-aws-auth-cm" \l "gives-access-to-our-iam-roles-to-amazon-eks-cluster)

In order to give access to the IAM Roles we defined previously to our Amazon EKS cluster, we need to add specific **mapRoles** to the aws-auth ConfigMap

The advantage of using Role to access the cluster instead of specifying directly IAM users is that it will be easier to manage so we won't have to update the ConfigMap each time we want to add or remove users, we will just need to add or remove users from the IAM Group and we just configure the ConfigMap to allow the IAM Role associated to the IAM Group.

### [Update the aws-auth configmap to allow our IAM roles](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/6-configure-aws-auth-cm" \l "update-the-aws-auth-configmap-to-allow-our-iam-roles)

The **aws-auth** configmap from the kube-system namespace must be edited in order to allow or delete IAM roles arns.

This file makes the mapping between IAM role and Kubernetes RBAC rights. We can edit it manually:

We can edit it using [eksctl](https://github.com/weaveworks/eksctl) :

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

eksctl create iamidentitymapping \

--cluster eksworkshop-eksctl \

--arn arn:aws:iam::${ACCOUNT\_ID}:role/k8sDev \

--username dev-user

eksctl create iamidentitymapping \

--cluster eksworkshop-eksctl \

--arn arn:aws:iam::${ACCOUNT\_ID}:role/k8sInteg \

--username integ-user

eksctl create iamidentitymapping \

--cluster eksworkshop-eksctl \

--arn arn:aws:iam::${ACCOUNT\_ID}:role/k8sAdmin \

--username admin \

--group system:masters

Check Output

1

2

3

4

5

6

7

8

2023-03-14 09:57:10 [ℹ] checking arn arn:aws:iam::ACCOUNT\_ID:role/k8sDev against entries in the auth ConfigMap

2023-03-14 09:57:10 [ℹ] adding identity "arn:aws:iam::ACCOUNT\_ID:role/k8sDev" to auth ConfigMap

2023-03-14 09:57:10 [ℹ] checking arn arn:aws:iam::ACCOUNT\_ID:role/k8sInteg against entries in the auth ConfigMap

2023-03-14 09:57:10 [ℹ] adding identity "arn:aws:iam::ACCOUNT\_ID:role/k8sInteg" to auth ConfigMap

2023-03-14 09:57:10 [ℹ] checking arn arn:aws:iam::ACCOUNT\_ID:role/k8sAdmin against entries in the auth ConfigMap

2023-03-14 09:57:10 [ℹ] adding identity "arn:aws:iam::ACCOUNT\_ID:role/k8sAdmin" to auth ConfigMap

you should have the config map looking something like:

1

kubectl get cm -n kube-system aws-auth -o yaml

The output looks like below.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

apiVersion: v1

data:

mapRoles: |

- groups:

- system:bootstrappers

- system:nodes

rolearn: arn:aws:iam::ACCOUNT\_ID:role/eksctl-eksworkshop-eksctl-nodegro-NodeInstanceRole-14TKBWBD7KWFH

username: system:node:{{EC2PrivateDNSName}}

- rolearn: arn:aws:iam::ACCOUNT\_ID:role/k8sDev

username: dev-user

- rolearn: arn:aws:iam::ACCOUNT\_ID:role/k8sInteg

username: integ-user

- groups:

- system:masters

rolearn: arn:aws:iam::ACCOUNT\_ID:role/k8sAdmin

username: admin

mapUsers: |

[]

kind: ConfigMap

In the above output, the AWS IAM Role for example arn:aws:iam::ACCOUNT\_ID:role/k8sAdmin is mapped to a Kubernetes RBAC user admin, which is added to the Kubernetes RBAC group system:masters.

We can leverage eksctl to get a list of all identities managed in our cluster.

1

eksctl get iamidentitymapping --cluster eksworkshop-eksctl

The output looks like below.

1

2

3

4

arn:aws:iam::ACCOUNT\_ID:role/eksctl-quick-nodegroup-ng-fe1bbb6-NodeInstanceRole-1KRYARWGGHPTTsystem:node:{{EC2PrivateDNSName}}system:bootstrappers,system:nodes

arn:aws:iam::ACCOUNT\_ID:role/k8sAdmin adminsystem:masters

arn:aws:iam::ACCOUNT\_ID:role/k8sDev dev-user

arn:aws:iam::ACCOUNT\_ID:role/k8sInteg integ-user

Here is what we have created so far:

* a RBAC role for K8sAdmin, that we map to admin user and give access to **system:masters** kubernetes Groups so that it has Full Admin rights on the cluster.

**Note**

This is only for example purpose. It is highly recommended not to add any Kubernetes user to **system:masters** group unless it is necessary

* a RBAC role for k8sDev that we map on dev-user in development Namespace
* a RBAC role for k8sInteg that we map on integ-user in integration Namespace

We will see on next section how we can test it.

# Test Amazon EKS access

## [Automate assumerole with aws cli](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/7-test-eks-access" \l "automate-assumerole-with-aws-cli)

It is possible to automate the retrieval of temporary credentials for the assumed role by configuring the AWS CLI in the files ~/.aws/config and ~/.aws/credentials. As an example, we will define three profiles.

#### [Add in](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/7-test-eks-access" \l "add-in-~.awsconfig:)**[~/.aws/config](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/7-test-eks-access" \l "add-in-~.awsconfig:)**[:](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/7-test-eks-access" \l "add-in-~.awsconfig:)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

mkdir -p ~/.aws

if ! test -f ~/.aws/config; then

cat << EoF >> ~/.aws/config

[profile admin]

role\_arn=arn:aws:iam::${ACCOUNT\_ID}:role/k8sAdmin

source\_profile=eksAdmin

[profile dev]

role\_arn=arn:aws:iam::${ACCOUNT\_ID}:role/k8sDev

source\_profile=eksDev

[profile integ]

role\_arn=arn:aws:iam::${ACCOUNT\_ID}:role/k8sInteg

source\_profile=eksInteg

EoF

else

echo "AWS Config file ~/.aws/config already exists..."

fi

#### [Add in](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/7-test-eks-access" \l "add-in-~.awscredentials:)**[~/.aws/credentials](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/7-test-eks-access" \l "add-in-~.awscredentials:)**[:](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/7-test-eks-access" \l "add-in-~.awscredentials:)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

if ! test -f ~/.aws/credentials; then

cat << EoF >> ~/.aws/credentials

[eksAdmin]

aws\_access\_key\_id=$(jq -r .AccessKey.AccessKeyId /tmp/PaulAdmin.json)

aws\_secret\_access\_key=$(jq -r .AccessKey.SecretAccessKey /tmp/PaulAdmin.json)

[eksDev]

aws\_access\_key\_id=$(jq -r .AccessKey.AccessKeyId /tmp/JeanDev.json)

aws\_secret\_access\_key=$(jq -r .AccessKey.SecretAccessKey /tmp/JeanDev.json)

[eksInteg]

aws\_access\_key\_id=$(jq -r .AccessKey.AccessKeyId /tmp/PierreInteg.json)

aws\_secret\_access\_key=$(jq -r .AccessKey.SecretAccessKey /tmp/PierreInteg.json)

EoF

else

echo "AWS Credentials file ~/.aws/credentials already exists..."

fi

#### [Test this with the dev profile:](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/7-test-eks-access" \l "test-this-with-the-dev-profile:)

1

aws sts get-caller-identity --profile dev

The output looks like below.

1

2

3

4

5

{

"UserId": "AROAUD5VMKW75WJEHFU4X:botocore-session-1581687024",

"Account": "ACCOUNT\_ID",

"Arn": "arn:aws:sts::ACCOUNT\_ID:assumed-role/k8sDev/botocore-session-1581687024"

}

The assumed-role is k8sDev, so we achieved our goal.

When specifying the **--profile dev** parameter we automatically ask for temporary credentials for the role k8sDev. You can test this with **integ** and **admin** also.

1

aws sts get-caller-identity --profile admin

The output looks like below.

1

2

3

4

5

{

"UserId": "AROAUD5VMKW77KXQAL7ZX:botocore-session-1582022121",

"Account": "ACCOUNT\_ID",

"Arn": "arn:aws:sts::ACCOUNT\_ID:assumed-role/k8sAdmin/botocore-session-1582022121"

}

*When specifying the****--profile admin****parameter we automatically ask for temporary credentials for the role k8sAdmin*

## [Using AWS profiles with the Kubectl config file](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/7-test-eks-access" \l "using-aws-profiles-with-the-kubectl-config-file)

It is also possible to specify the AWS\_PROFILE to use with the aws-iam-authenticator in the ~/.kube/config file, so that it will use the appropriate profile.

### [With dev profile](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/7-test-eks-access" \l "with-dev-profile)

Create a new KUBECONFIG file to test this:

1

2

export KUBECONFIG=/tmp/kubeconfig-dev && eksctl utils write-kubeconfig -c eksworkshop-eksctl

cat $KUBECONFIG | yq e '.users.[].user.exec.args += ["--profile", "dev"]' - -- | sed 's/eksworkshop-eksctl./eksworkshop-eksctl-dev./g' | sponge $KUBECONFIG

Check Output

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

2023-03-14 10:16:28 [✔] saved kubeconfig as "/tmp/kubeconfig-dev"

Unable to find image 'mikefarah/yq:latest' locally

latest: Pulling from mikefarah/yq

63b65145d645: Pulling fs layer

865242c25e72: Pulling fs layer

48f2cb577b3c: Pulling fs layer

6b38082b4af1: Pulling fs layer

0a8c5b7f3b42: Pulling fs layer

6b38082b4af1: Waiting

0a8c5b7f3b42: Waiting

48f2cb577b3c: Verifying Checksum

48f2cb577b3c: Download complete

63b65145d645: Verifying Checksum

63b65145d645: Download complete

865242c25e72: Verifying Checksum

865242c25e72: Download complete

63b65145d645: Pull complete

6b38082b4af1: Verifying Checksum

6b38082b4af1: Download complete

0a8c5b7f3b42: Verifying Checksum

0a8c5b7f3b42: Download complete

865242c25e72: Pull complete

48f2cb577b3c: Pull complete

6b38082b4af1: Pull complete

0a8c5b7f3b42: Pull complete

Digest: sha256:29ebb32f7d89a6b8e102a9cf1fb1c073d7154c17e5eda8a584f60f036b11f655

Status: Downloaded newer image for mikefarah/yq:latest

*Note: this assume you uses yq >= version 4. you can reference to*[this page](https://mikefarah.gitbook.io/yq/upgrading-from-v3)*to adapt this command for another version.*

We added the --profile dev parameter to our kubectl config file, so that this will ask kubectl to use our IAM role associated to our dev profile, and we rename the context using suffix **-dev**.

With this configuration we should be able to interact with the **development** namespace, because it has our RBAC role defined.

Let's create a pod:

1

kubectl run nginx-dev --image=nginx -n development

Check Output

1

pod/nginx-dev created

We can list the pods:

1

kubectl get pods -n development

The output looks like below

1

2

NAME READY STATUS RESTARTS AGE

nginx-dev 1/1 Running 0 28s

... but not in other namespaces:

1

kubectl get pods -n integration

The output looks like below

1

Error from server (Forbidden): pods is forbidden: User "dev-user" cannot list resource "pods" in API group "" in the namespace "integration"

#### [Test with integ profile](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/7-test-eks-access" \l "test-with-integ-profile)

1

2

export KUBECONFIG=/tmp/kubeconfig-integ && eksctl utils write-kubeconfig -c eksworkshop-eksctl

cat $KUBECONFIG | yq e '.users.[].user.exec.args += ["--profile", "integ"]' - -- | sed 's/eksworkshop-eksctl./eksworkshop-eksctl-integ./g' | sponge $KUBECONFIG

Check Output

1

2023-03-14 10:24:31 [✔] saved kubeconfig as "/tmp/kubeconfig-integ"

*Note: this assume you uses yq >= version 4. you can reference to*[this page](https://mikefarah.gitbook.io/yq/upgrading-from-v3)*to adapt this command for another version.*

Let's create a pod:

1

kubectl run nginx-integ --image=nginx -n integration

Check Output

1

pod/nginx-integ created

We can list the pods:

1

kubectl get pods -n integration

1

2

NAME READY STATUS RESTARTS AGE

nginx-integ 1/1 Running 0 43s

... but not in other namespaces:

1

kubectl get pods -n development

1

Error from server (Forbidden): pods is forbidden: User "integ-user" cannot list resource "pods" in API group "" in the namespace "development"

#### [Test with admin profile](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/7-test-eks-access" \l "test-with-admin-profile)

1

2

export KUBECONFIG=/tmp/kubeconfig-admin && eksctl utils write-kubeconfig -c eksworkshop-eksctl

cat $KUBECONFIG | yq e '.users.[].user.exec.args += ["--profile", "admin"]' - -- | sed 's/eksworkshop-eksctl./eksworkshop-eksctl-admin./g' | sponge $KUBECONFIG

Check Output

1

2023-03-14 10:30:52 [✔] saved kubeconfig as "/tmp/kubeconfig-admin"

*Note: this assume you uses yq >= version 4. you can reference to*[this page](https://mikefarah.gitbook.io/yq/upgrading-from-v3)*to adapt this command for another version.*

Let's create a pod in the default namespace:

1

kubectl run nginx-admin --image=nginx

Check Output

1

pod/nginx-admin created

We can list the pods:

1

kubectl get pods

We can list the pods:

1

2

NAME READY STATUS RESTARTS AGE

nginx-admin 1/1 Running 0 2m21s

We can list ALL pods in all namespaces:

1

kubectl get pods -A

The output looks like below.

1

2

3

4

5

6

7

8

9

10

11

12

NAMESPACE NAME READY STATUS RESTARTS AGE

default nginx-admin 1/1 Running 0 15s

development nginx-dev 1/1 Running 0 11m

integration nginx-integ 1/1 Running 0 4m29s

kube-system aws-node-mzbh4 1/1 Running 0 100m

kube-system aws-node-p7nj7 1/1 Running 0 100m

kube-system aws-node-v2kg9 1/1 Running 0 100m

kube-system coredns-85bb8bb6bc-2qbx6 1/1 Running 0 105m

kube-system coredns-85bb8bb6bc-87ndr 1/1 Running 0 105m

kube-system kube-proxy-4n5lc 1/1 Running 0 100m

kube-system kube-proxy-b65xm 1/1 Running 0 100m

kube-system kube-proxy-pr7k7 1/1 Running 0 100m

## [Conclusion](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/7-test-eks-access" \l "conclusion)

In this module, we have seen how to configure Amazon EKS to provide finer access to users combining IAM Groups and Kubernetes RBAC. You can create different groups depending on your needs, configure their associated RBAC access in your cluster, and simply add or remove users from the group to grant or revoke access to your cluster.

Users will only have to configure their AWS CLI in order to automatically retrieve their associated rights in your cluster.

# AWS console access to EKS Cluster

This step is optional, as nearly all of the workshop content is CLI-driven. But, if you'd like full access to your workshop cluster in the EKS console this step is recommended.

The EKS console allows you to see not only the configuration aspects of your cluster, but also to view Kubernetes cluster objects such as Deployments, Pods, and Nodes. For this type of access, the console IAM User or Role needs to be granted permission within the cluster.

By default, the credentials used to create the cluster are automatically granted these permissions. Following along in the workshop, you've created a cluster using temporary IAM credentials from within Cloud9. This means that you'll need to add your AWS Console credentials to the cluster.

#### [Import your EKS Console credentials to your new cluster:](https://catalog.us-east-1.prod.workshops.aws/workshops/165b0729-2791-4452-8920-53b734419050/en-US/2-identity-and-access-management/1-iam-groups-roles-to-manage-eks-access/8-console-credentials" \l "import-your-eks-console-credentials-to-your-new-cluster:)

IAM Users and Roles are bound to an EKS Kubernetes cluster via a ConfigMap named aws-auth. We can use eksctl to do this with one command.

You'll need to determine the correct credential to add for your AWS Console access. If you know this already, you can skip ahead to the eksctl create iamidentitymapping step below.

If you've built your cluster from Cloud9 as part of this tutorial, invoke the following within your environment to determine your IAM Role or User ARN.

1

2

3

4

5

6

7

8

9

c9builder=$(aws cloud9 describe-environment-memberships --environment-id=$C9\_PID | jq -r '.memberships[].userArn')

if echo ${c9builder} | grep -q user; then

rolearn=${c9builder}

echo Role ARN: ${rolearn}

elif echo ${c9builder} | grep -q assumed-role; then

assumedrolename=$(echo ${c9builder} | awk -F/ '{print $(NF-1)}')

rolearn=$(aws iam get-role --role-name ${assumedrolename} --query Role.Arn --output text)

echo Role ARN: ${rolearn}

fi

Check Output

1

Role ARN: arn:aws:iam::XXXXXXXXXXX:role/WSParticipantRole

With your ARN in hand, you can issue the command to create the identity mapping within the cluster.

1

eksctl create iamidentitymapping --cluster eksworkshop-eksctl --arn ${rolearn} --group system:masters --username admin

Check Output

1

2

2023-05-21 14:10:10 [ℹ] checking arn arn:aws:iam::XXXXXXXXXXX:role/WSParticipantRole against entries in the auth ConfigMap

2023-05-21 14:10:10 [ℹ] adding identity "arn:aws:iam::XXXXXXXXXXX:role/WSParticipantRole" to auth ConfigMap

Note that permissions can be restricted and granular but as this is a workshop cluster, you're adding your console credentials as administrator.

Now you can verify your entry in the AWS auth map within the console.

1

kubectl describe configmap -n kube-system aws-auth

Check Output

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

Name: aws-auth

Namespace: kube-system

Labels: <none>

Annotations: <none>

Data

====

mapRoles:

----

- groups:

- system:bootstrappers

- system:nodes

rolearn: arn:aws:iam::XXXXXXXXXXX:role/eks-bootstrap-template-ws-EKSNodegroupRole-1AL88NMU1M9N4

username: system:node:{{EC2PrivateDNSName}}

- groups:

- system:masters

rolearn: arn:aws:iam::XXXXXXXXXXX:role/WSParticipantRole

username: admin

username: admin

mapUsers:

----

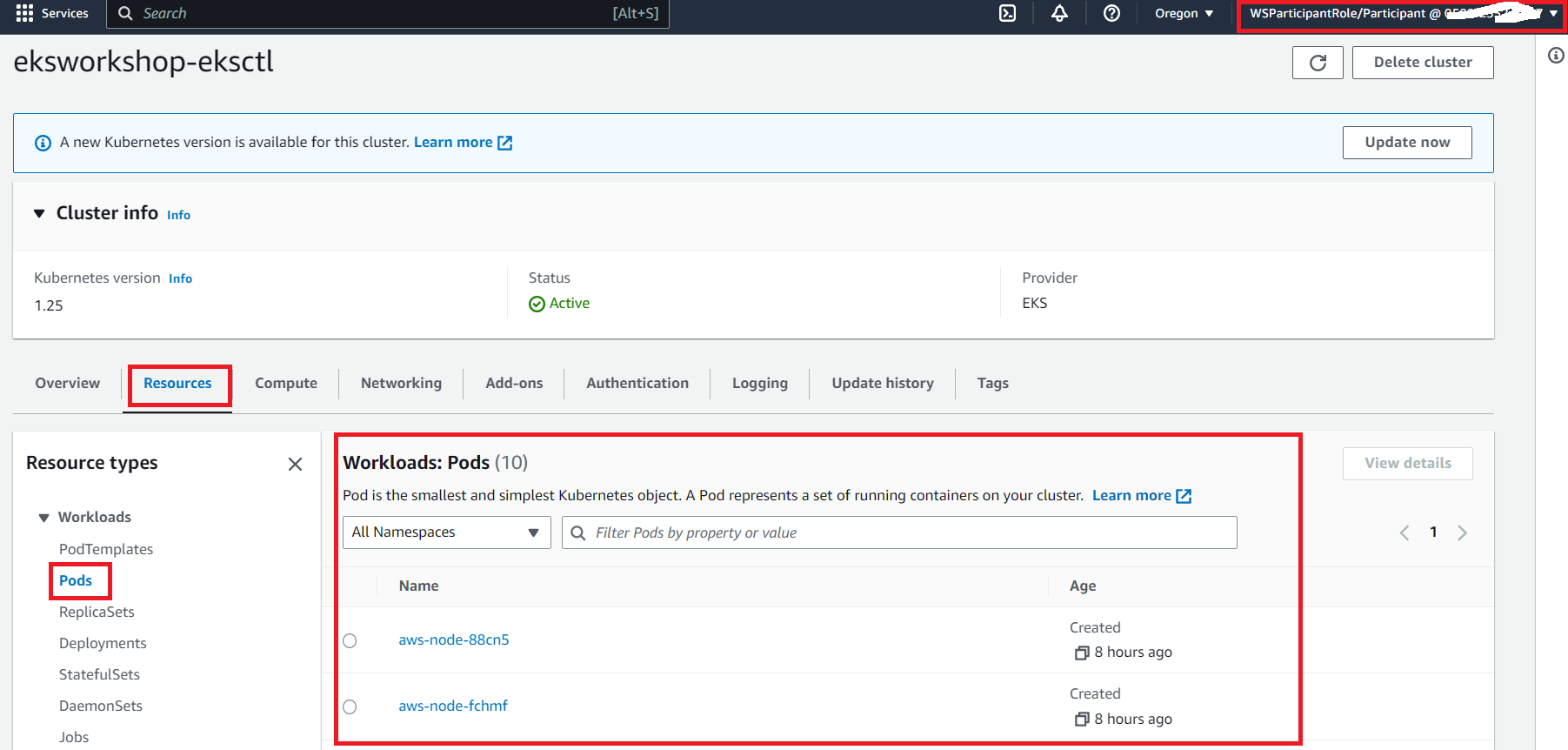
[]

BinaryData

====

Events: <none>

You can now view various Kubernetes Objects in the Amazon EKS Cluster in the [AWS Console for Amazon EKS](https://console.aws.amazon.com/eks/home?#/clusters/eksworkshop-eksctl?selectedTab=cluster-resources-tab&selectedResourceId=pods).



For more information, check out the [EKS documentation](https://docs.aws.amazon.com/eks/latest/userguide/add-user-role.html) on this topic.

# Cleanup

Once you have completed this chapter, you can cleanup the files and resources you created by issuing the following commands:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

unset KUBECONFIG

kubectl delete namespace development integration

kubectl delete pod nginx-admin

eksctl delete iamidentitymapping --cluster eksworkshop-eksctl --arn arn:aws:iam::${ACCOUNT\_ID}:role/k8sAdmin

eksctl delete iamidentitymapping --cluster eksworkshop-eksctl --arn arn:aws:iam::${ACCOUNT\_ID}:role/k8sDev

eksctl delete iamidentitymapping --cluster eksworkshop-eksctl --arn arn:aws:iam::${ACCOUNT\_ID}:role/k8sInteg

aws iam remove-user-from-group --group-name k8sAdmin --user-name PaulAdmin

aws iam remove-user-from-group --group-name k8sDev --user-name JeanDev

aws iam remove-user-from-group --group-name k8sInteg --user-name PierreInteg

aws iam delete-group-policy --group-name k8sAdmin --policy-name k8sAdmin-policy

aws iam delete-group-policy --group-name k8sDev --policy-name k8sDev-policy

aws iam delete-group-policy --group-name k8sInteg --policy-name k8sInteg-policy

aws iam delete-group --group-name k8sAdmin

aws iam delete-group --group-name k8sDev

aws iam delete-group --group-name k8sInteg

aws iam delete-access-key --user-name PaulAdmin --access-key-id=$(jq -r .AccessKey.AccessKeyId /tmp/PaulAdmin.json)

aws iam delete-access-key --user-name JeanDev --access-key-id=$(jq -r .AccessKey.AccessKeyId /tmp/JeanDev.json)

aws iam delete-access-key --user-name PierreInteg --access-key-id=$(jq -r .AccessKey.AccessKeyId /tmp/PierreInteg.json)

aws iam delete-user --user-name PaulAdmin

aws iam delete-user --user-name JeanDev

aws iam delete-user --user-name PierreInteg

aws iam delete-role --role-name k8sAdmin

aws iam delete-role --role-name k8sDev

aws iam delete-role --role-name k8sInteg

rm /tmp/\*.json

rm /tmp/kubeconfig\*

# reset aws credentials and config files

rm ~/.aws/{config,credentials}

aws configure set default.region ${AWS\_REGION}

Check Output

1

2

3

4

5

6

namespace "development" deleted

namespace "integration" deleted

pod "nginx-admin" deleted

2023-03-14 10:38:40 [ℹ] removing identity "arn:aws:iam::XXXXXXXXXX:role/k8sAdmin" from auth ConfigMap (username = "admin", groups = ["system:masters"])

2023-03-14 10:38:40 [ℹ] removing identity "arn:aws:iam::XXXXXXXXXX:role/k8sDev" from auth ConfigMap (username = "dev-user", groups = [])

2023-03-14 10:38:42 [ℹ] removing identity "arn:aws:iam::XXXXXXXXXX:role/k8sInteg" from auth ConfigMap (username = "integ-user", groups = [])