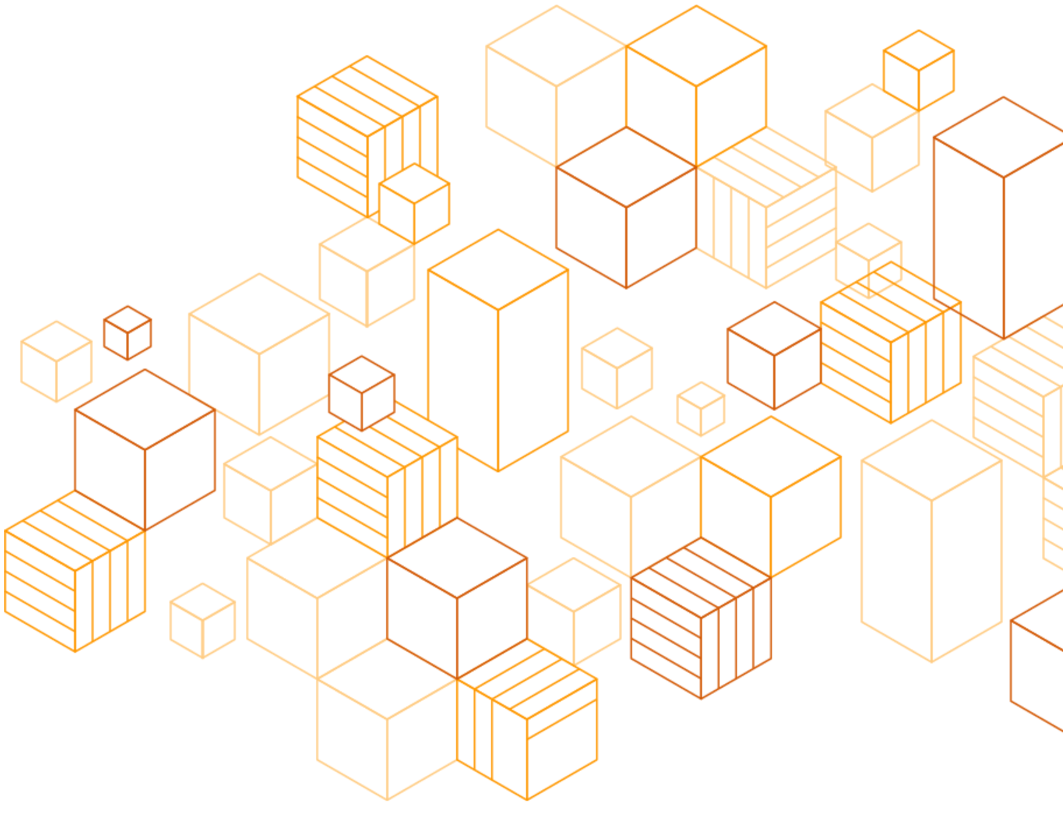
Accessing a private Amazon MWAA environment using federated identities

Technical Guide

February 8, 2022



Notices

Contents

About this Guide

Many organizations use 3rd party identity providers such as Azure Active Directory (Azure AD) and Okta to grant their users access to internal resources. Also, unless you have a specific reason, it is a best practice to avoid exposing application endpoints directly on the internet. Amazon Managed Workflows for Apache Airflow (MWAA) is no exception, and both organizations and users want to continue using the same authentication and authorization mechanism to access their private Amazon MWAA environments. This step-by-step guide explains how to build a solution that allows using federated identities to seamlessly access private Amazon MWAA environments securely.

Overview

Amazon MWAA offers two network access modes for accessing the Airflow Web User Interface (UI) in your environments: public and private.
In both cases, accessing the Airflow Web UI of an Amazon MWAA environment requires authentication via the AWS Management Console. Additionally, if you use the private network access mode, you have to route your traffic over private subnets in your VPC, which means you need a way to reach your VPC from the client you use to access the web application, such as a site-to-site VPN, AWS Direct Connect, or AWS Client VPN.
Depending on your security and connectivity requirements, those options might not be viable. For example, you might want to use AWS Web Application Firewall (WAF) to inspect traffic addressed to the web UI for anomalous patterns or apply geofencing, or you might not want to provide AWS Management Console access to all the Airflow users in your organization. Furthermore, many customers would like to use their existing Identity Providers (IdP) to access their Amazon MWAA environments.
In this guide, you find detailed instructions to set up a solution to provide access to an environment deployed in private network access mode and authenticate users using a federated identity without the need to have permissions to access the AWS Management Console.

Before You Begin / Considerations

This technical guide uses the AWS Command Line Interface (CLI), the AWS Management Console, and an IAM role with appropriate permissions. You can learn how to install and configure the AWS CLI by reviewing Getting started with the AWS CLI and Configuring the AWS CLI.
To make it simpler to reproduce, the steps and CLI commands in the guide use given names for resources such as an Application Load Balancer (ALB) or an Amazon Cognito user pool. Feel free to change them at your convenience.

Cost

The main cost factors for the solution described in this guide fall on the Amazon MWAA environment and the ALB. This cost analysis focuses only on the fixed hourly usage of the deployed components and considers 744 hours (31 days x 24 hours / day) in a month. There are other costs that depend on usage, such as Amazon MWAA additional worker and scheduled instances, and ALB LCUs (Load Balancer Units).
You can find more details on the Amazon MWAA pricing page and Application Load Balancer pricing page.

Amazon MWAA Environment

This technical guide uses a **Small** environment instance, which in the Europe (Ireland) AWS Region is priced at $0.49 per hour. That gives a monthly cost of $364.56.

ALB

Consider one ALB running for the entire month and that there is some traffic every hour that falls within the scope of a single LCU. The price for ALB in the Europe (Ireland) AWS Region is $0.0252 per ALB-hour (or partial hour), and $0.008 per LCU-hour (or partial hour). This amounts to $24.7008 per month.

Architecture Overview

Before jumping into the solution details, it’s important to understand what goes under the hood when you create an Amazon MWAA environment with private access mode. In a nutshell, Amazon MWAA uses its own VPC and resources to host the Airflow Web Server, and creates a VPC interface endpoint. This endpoint is reachable within your VPC from the selected subnets by deploying an Elastic Network Interface (ENI) in each of them. Each of those ENIs have a binding to an IP address from each of your subnets.
You also need to understand that, when you authenticate to the Airflow UI, Amazon MWAA generates a web login token for the environment. This token authorizes access to the environment with an Airflow role that is based on the permissions granted to the IAM principal you use to log in to the AWS Management Console.
The proposed solution is based in six components:

- A VPC with four subnets (two public and two private).
- An Amazon MWAA environment with private access to the Airflow Web Server.
- A public ALB that exposes the UI and authenticates users via Amazon Cognito.
- An Amazon Cognito user pool that uses a federated login via Azure AD and provides the federated user claims to an authorization Lambda function.
- A Lambda function that authorizes access to the Amazon MWAA environment. For that it assumes an IAM role and generates the Amazon MWAA web login token on behalf of the user and handles the logout process.
- A set of IAM roles that grant access to the resources.

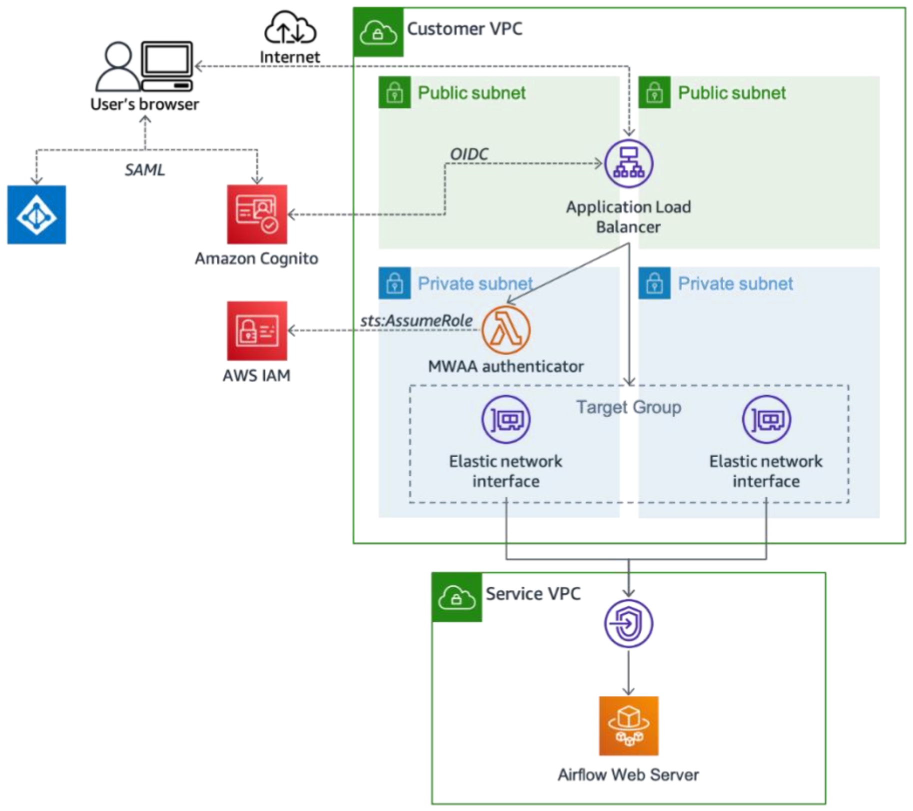


Figure 1 – Architecture overview

Network

VPC for the Amazon MWAA environment

Amazon MWAA requires customers to provide a VPC with at least two subnets to deploy an environment.
Follow the steps in the Create the VPC network Amazon MWAA documentation to create a VPC *with* internet access. The documentation guides you on the process to deploy the resources using an AWS CloudFormation template. This template deploys:

- A VPC with a pair of public and private subnets spread across two Availability Zones.
- An internet gateway, with a default route on the public subnets.
- A pair of NAT gateways (one in each Availability Zone), and default routes for them in the private subnets.
- A self-referencing security group that allows all traffic. This will be used by Amazon MWAA to communicate between internal resources.

If you require the traffic between your Amazon MWAA environment and other resources to go over private networks, you should use Option three: Creating an Amazon VPC network without

Internet access. In that case, you will also need to deploy a pair of public subnets in the VPC to deploy the ALB.
After the deployment has completed, you should have the following portion of the architecture (note that, for clarity, security groups are not depicted):

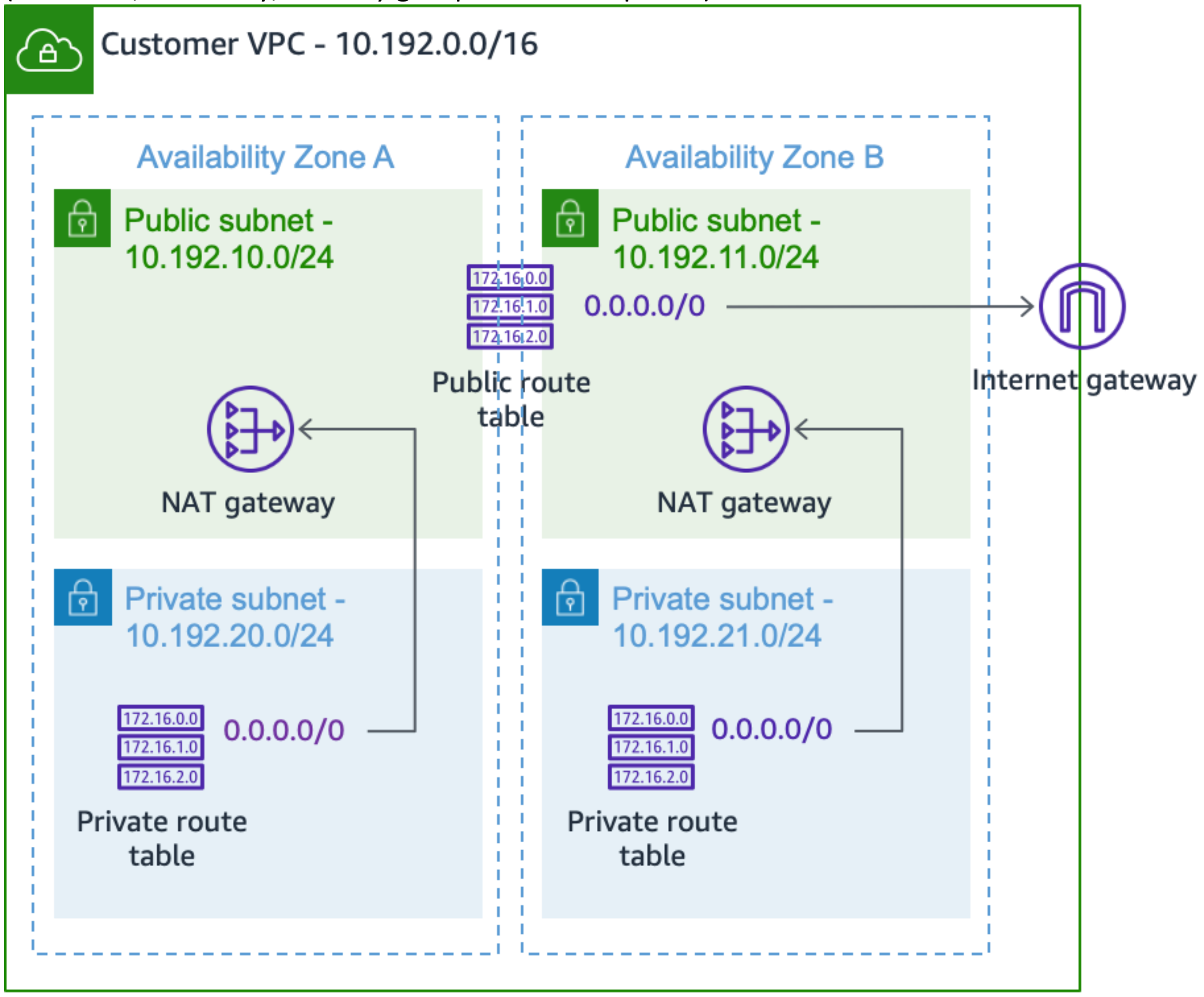
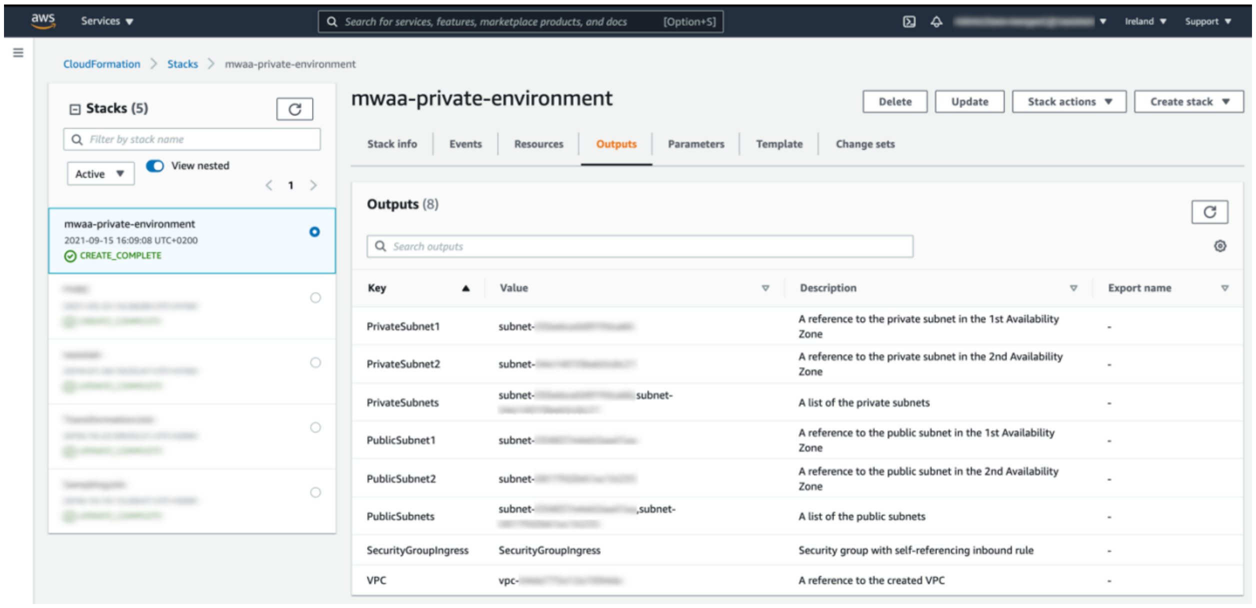


Figure 2 – Network architecture

For the next step, you will need the identifiers of some of the resources you have created so far. You can find those identifiers in the Outputs tab of the CloudFormation template you deployed in the previous step (see Figure 3).



Outputs (8)

Key	Value	Description	Export name
PrivateSubnet1	subnet-[redacted]	A reference to the private subnet in the 1st Availability Zone	-
PrivateSubnet2	subnet-[redacted]	A reference to the private subnet in the 2nd Availability Zone	-
PrivateSubnets	subnet-[redacted] subnet-[redacted]	A list of the private subnets	-
PublicSubnet1	subnet-[redacted]	A reference to the public subnet in the 1st Availability Zone	-
PublicSubnet2	subnet-[redacted]	A reference to the public subnet in the 2nd Availability Zone	-
PublicSubnets	subnet-[redacted],subnet-[redacted]	A list of the public subnets	-
SecurityGroupIngress	SecurityGroupIngress	Security group with self-referencing inbound rule	-
VPC	vpc-[redacted]	A reference to the created VPC	-

Figure 3 – AWS CloudFormation Outputs

Amazon MWAA Environment

Prerequisites

Before deploying the environment, you'll have to create another resource that Amazon MWAA requires: an Amazon S3 bucket. Amazon MWAA will use this S3 bucket to store its Direct Acyclic Graphs (DAGs) code and supporting files, such as plugins.

Create a bucket in the same region where you deployed your VPC and, as per Amazon MWAA requirements, enable blocking public access and versioning with the following AWS CLI commands:

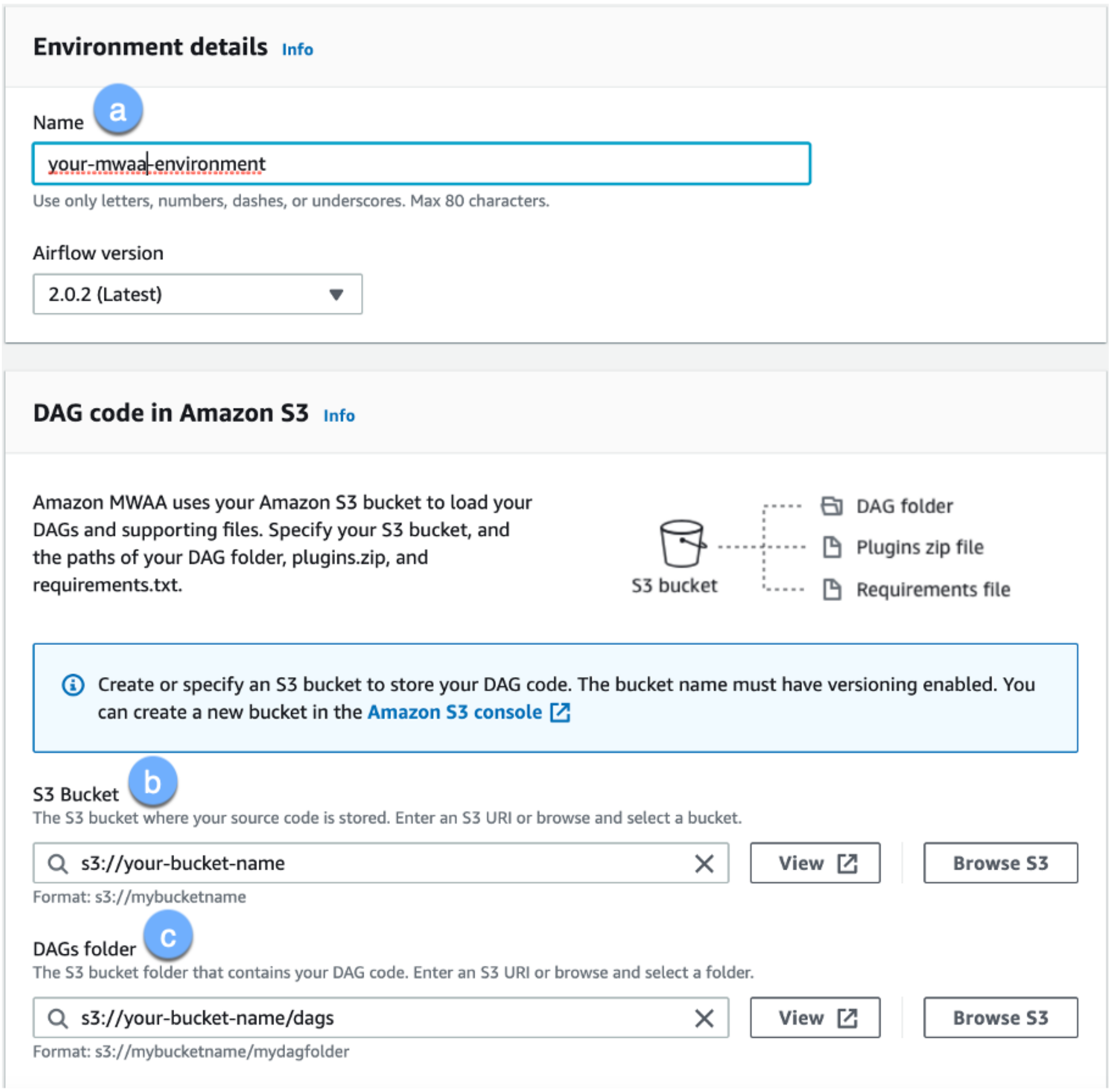
```
aws s3 mb --region your-region s3://your-bucket-name
aws s3api put-bucket-versioning --region your-region \
    --bucket your-bucket-name \
    --versioning-configuration Status=Enabled
aws s3api put-public-access-block --region your-region \
    --bucket your-bucket-name \
    --public-access-block-configuration
"BlockPublicAcls=true,IgnorePublicAcls=true,BlockPublicPolicy=true,Restrict
PublicBuckets=true"
```

Note: You won't need to provide explicit access to this bucket to your end users, Amazon MWAA manages that on your behalf. Additionally, you may want to check your requirements for other security settings, such as encryption.

The next section guides you through the creation of the Amazon MWAA environment using the AWS Management Console.

Creating the Amazon MWAA environment

1. In the Amazon MWAA console, choose **Create environment.**
2. In **Environment details**, fill in:
   a. The name for your environment,
   b. An S3 URI pointing to the bucket you created earlier (e.g.: `s3://`*`your-bucket-name`*), and
   c. Another S3 URI that points to a path in that bucket (e.g.: `s3://`*`your-bucket-name`*`/dags`).


Environment details Info

Name
your-mwaa-environment
Use only letters, numbers, dashes, or underscores. Max 80 characters.

Airflow version
2.0.2 (Latest)

DAG code in Amazon S3 Info

Amazon MWAA uses your Amazon S3 bucket to load your DAGs and supporting files. Specify your S3 bucket, and the paths of your DAG folder, plugins.zip, and requirements.txt.

Create or specify an S3 bucket to store your DAG code. The bucket name must have versioning enabled. You can create a new bucket in the Amazon S3 console

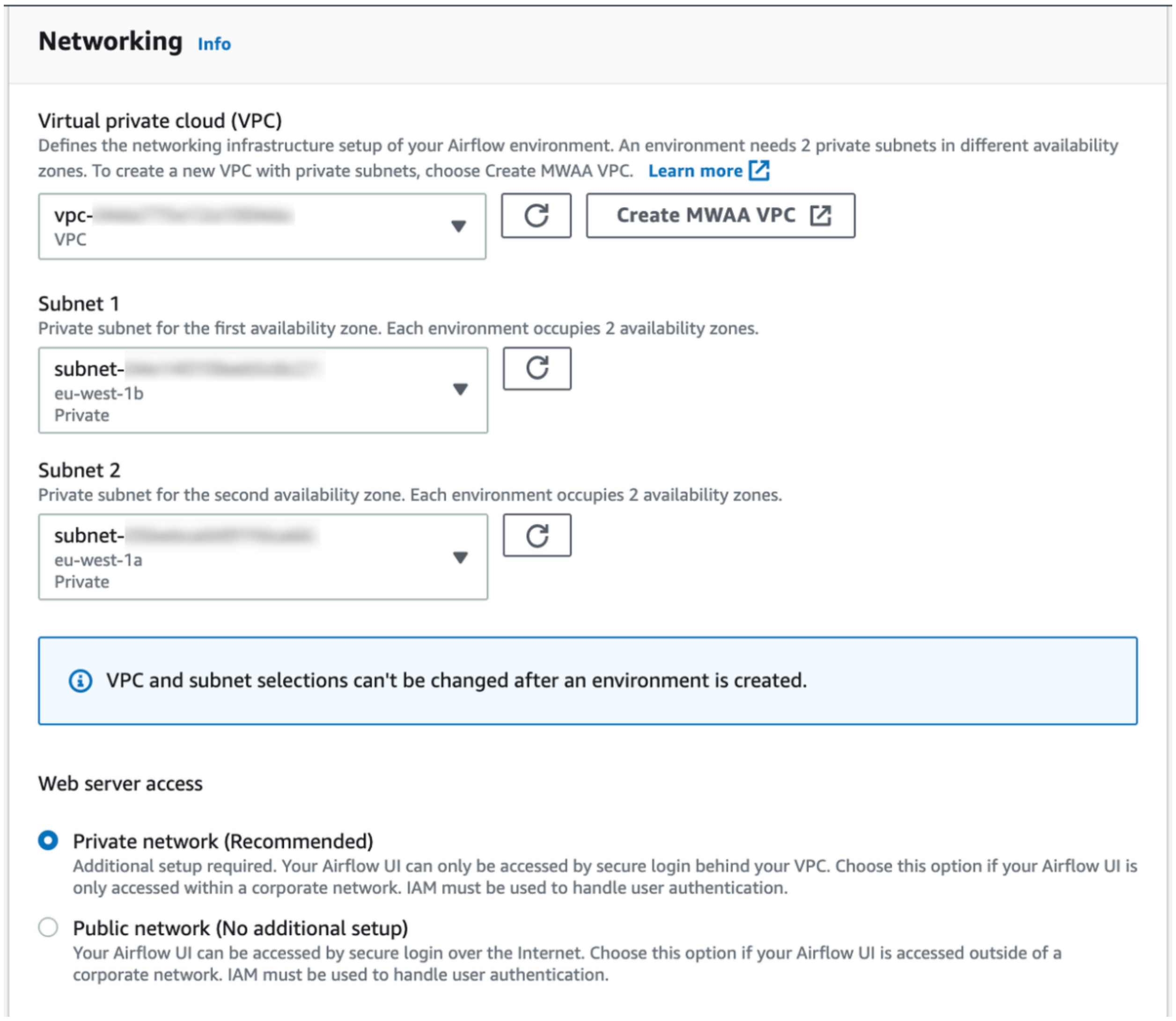
S3 Bucket
The S3 bucket where your source code is stored. Enter an S3 URI or browse and select a bucket.
s3://your-bucket-name
Format: s3://mybucketname

DAGs folder
The S3 bucket folder that contains your DAG code. Enter an S3 URI or browse and select a folder.
s3://your-bucket-name/dags
Format: s3://mybucketname/mydagfolder

Figure 4 – Amazon MWAA environment details (substitute the names for your own).

3. Leave the optional fields for the plugins and requirements files empty. You can update them at a later stage if you need to. Choose **Next**.

4. Select the VPC that was deployed with the CloudFormation template from the drop-down list, and make sure the two private subnets are selected. In the **Web server access** section, make sure **Private network** is toggled.



Networking Info

Virtual private cloud (VPC)
Defines the networking infrastructure setup of your Airflow environment. An environment needs 2 private subnets in different availability zones. To create a new VPC with private subnets, choose Create MWAA VPC. Learn more

vpc-
VPC

Subnet 1
Private subnet for the first availability zone. Each environment occupies 2 availability zones.

subnet-
eu-west-1b
Private

Subnet 2
Private subnet for the second availability zone. Each environment occupies 2 availability zones.

subnet-
eu-west-1a
Private

VPC and subnet selections can't be changed after an environment is created.

Web server access

● Private network (Recommended)
Additional setup required. Your Airflow UI can only be accessed by secure login behind your VPC. Choose this option if your Airflow UI is only accessed within a corporate network. IAM must be used to handle user authentication.

○ Public network (No additional setup)
Your Airflow UI can be accessed by secure login over the Internet. Choose this option if your Airflow UI is accessed outside of a corporate network. IAM must be used to handle user authentication.

Figure 5 – Amazon MWAA networking configuration form

5. In the **Security group(s)** section, deselect the **Create new security group** checkbox, and select the security group that was deployed during the VPC creation earlier.

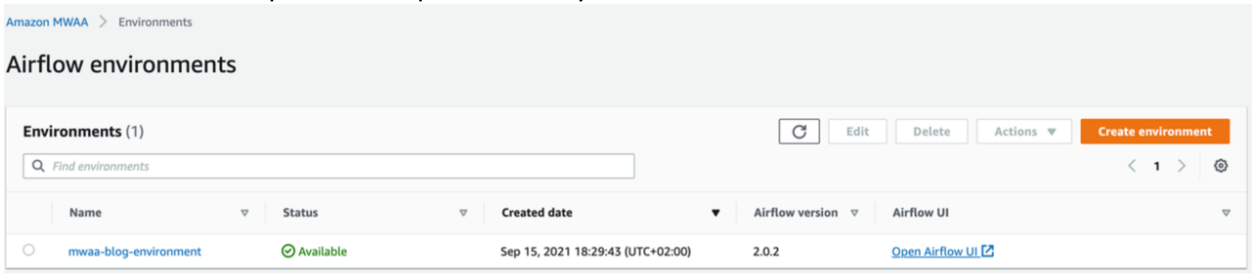
6. Select the environment class that better suits your needs. Since this guide is for demonstration purposes, it uses the **mw1.small class**.

7. Leave the remaining options as default unless you:
   - Need to encrypt data with a different key than the default one.
   - Want to disable Airflow task logs or select a different logging level than INFO.
   - Want to set some Apache Airflow configuration options.
   - Use an existing IAM role for your Amazon MWAA environment.

> Pay attention to the note about IAM: “Amazon MWAA will create and assume the execution role in IAM named `MWAA-your-environment-name-XXXXXX` on your behalf. This role is configured with permission to retrieve code from your Amazon S3 bucket, use your KMS key, and send data to Amazon CloudWatch. You must add permissions to your execution role if your Airflow DAGs require access to any other AWS services.”

8. Select **Next**, make sure the configuration is correct, and choose **Create environment**.

After the creation process finishes, you will see your environment in the console and, as expected, if you choose the Open Airflow UI, you will get a time out error, as you cannot access the Amazon MWAA private endpoints from your browser.


Amazon MWAA > Environments

Airflow environments

Environments (1)

	Name	Status	Created date	Airflow version	Airflow UI
○	mwaa-blog-environment	Available	Sep 15, 2021 18:29:43 (UTC+02:00)	2.0.2	Open Airflow UI

Figure 6 – Airflow environment available in the Amazon MWAA console

ALB

The Amazon MWAA environment you just deployed uses private endpoints, which are not accessible from the internet. In this section, you are going to provide access to these endpoints using an ALB. This ALB will provide a public endpoint that users can access over the Internet. You can protect this endpoint using several non-mutually-exclusive measures, such as using VPC Security Groups, leveraging WebACL rules with AWS WAF, or configuring the ALB to use Amazon Cognito to only allow authorized users through. This guide focuses on the latter, because it is also the mechanism to authenticate the users of your environment using federated identities.

ALB prerequisites

Security group

You will create the ALB in the same VPC you created in the step VPC for the Amazon MWAA environment. More specifically, the ALB will use the two public subnets you deployed in that VPC. You also need a security group that allows access on the port number 443 from the internet.

1. Create the security group using the AWS CLI:

```
aws ec2 create-security-group --region your-region \
    --description 'Security Group for the Amazon MWAA ALB' \
    --group-name mwaa-alb-sg \
    --vpc-id your-vpc-id \
    --tag-specifications 'ResourceType=security-
group,Tags=[{Key=Name,Value=mwaa-alb-sg}]'
```

The output should be similar to the following one (take note of the *GroupId)*.

```
{
    "GroupId": "sg-abcdef01234567890",
    "Tags": [
        {
            "Key": "Name",
            "Value": "mwaa-alb-sg"
        }
    ]
}
```

Note: You might want to enable Deletion protection on your load balancer to prevent it from being deleted accidentally. You might also want to enable Access logs for your Application Load Balancer to capture detailed information about requests sent to your load balancer.

2. This security group needs to allow ingress TCP traffic from everywhere on ports 80 and 443. Add the necessary ingress rules (substitute the group id with the one you got as a response in the previous command):

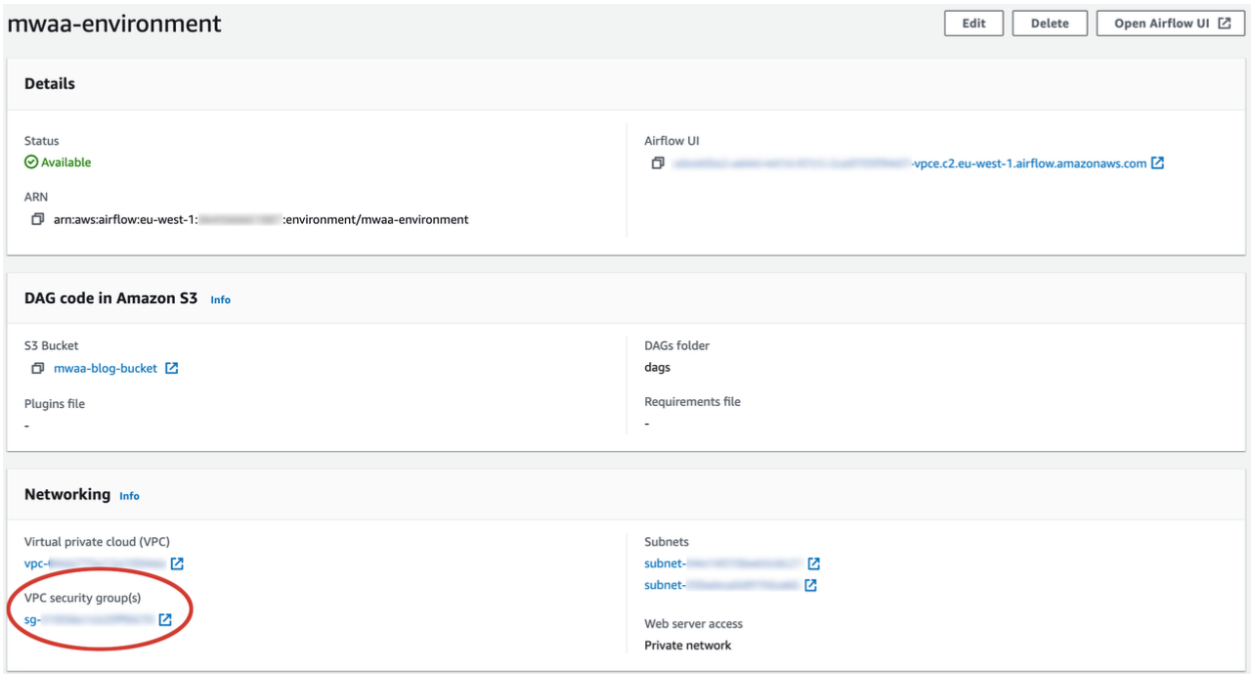
```
aws ec2 authorize-security-group-ingress --region your-region
\
    --group-id your-alb-security-group-id \
    --ip-permissions
'[{"IpProtocol":"tcp","FromPort":80,"ToPort":80,"IpRanges":[{
"CidrIp":"0.0.0.0/0","Description":"Access from Internet on
port 80"}]},

{"IpProtocol":"tcp","FromPort":443,"ToPort":443,"IpRanges":[{
"CidrIp":"0.0.0.0/0","Description":"Access from Internet on
port 443"}]}]'
```

3. You also need to allow access to the Amazon MWAA environment so that the ALB can direct traffic to it. To do this, add a rule to the Amazon MWAA security group that allows TCP traffic on port 443 to the ALB security group:

```
aws ec2 authorize-security-group-ingress --region your-region
\
    --group-id your-mwaa-security-group-id \
    --protocol tcp \
    --port 443 \
    --source-group your-alb-security-group-id
```

Note: You can find the Amazon MWAA security group id in the detail view of your environment in the Amazon MWAA console.



mwaa-environment

Details

Status
Available

ARN
arn:aws:airflow:eu-west-1:[redacted]:environment/mwaa-environment

Airflow UI
[redacted]-vpce.c2.eu-west-1.airflow.amazonaws.com

DAG code in Amazon S3

S3 Bucket
mwaa-blog-bucket

DAGs folder
dags

Plugins file
-

Requirements file
-

Networking

Virtual private cloud (VPC)
vpc-[redacted]

VPC security group(s)
sg-[redacted]

Subnets
subnet-[redacted]
subnet-[redacted]

Web server access
Private network

Figure 7 – Detailed view of the Amazon MWAA environment with its VPC security group highlighted.

Target groups

The ALB needs two target groups:

1. A target group to drive traffic to the Amazon MWAA web server. This target group must contain the two private IP addresses bound to the web server. I will refer to it as the Amazon MWAA target group.
2. The Amazon MWAA authentication Lambda function.

Creating the Amazon MWAA target group

The Amazon MWAA target group must use the HTTPS protocol, an IP target type, and be deployed in the same VPC as the Amazon MWAA environment.

1. Create the target group with the following AWS CLI command. This command uses the default health check settings, but making sure HTTP redirects (302) are considered healthy. Take note of the *TargetGroupArn* in the response.

```
aws elbv2 create-target-group --region your-region \
    --name mwaa-web-server \
    --port 443 \
    --protocol HTTPS \
    --vpc-id your-vpc-id \
    --health-check-protocol HTTPS \
    --matcher 'HttpCode="200,302"' \
    --target-type ip
```

Registering the Amazon MWAA private IP addresses

Register the IP addresses bound to the VPC endpoint deployed as part of the Amazon MWAA environment, so the ALB can send traffic to them.

1. Retrieve the IP addresses of the Amazon MWAA UI private endpoints following the steps found in the Identifying the private IP addresses of your Apache Airflow Web server and its VPC endpoint guide.

2. After you have the IP addresses, run the following AWS CLI command:

```
aws elbv2 register-targets --region your-region \
    --target-group-arn your-target-group-arn \
    --targets '[{"Id":"your-ip-address-
1","Port":443},{"Id":"your-ip-address-2","Port":443}]'
```

Creating the ALB

Now that all the prerequisites are met, you can create the ALB.

1. Run the following AWS CLI command to deploy an ALB that uses the target group you created earlier.

```
aws elbv2 create-load-balancer --region your-region \
    --name mwaa-alb \
    --subnets your-public-subnet-1-id your-public-subnet-2-id
\
    --security-groups your-alb-security-group-id
```

Note down the *LoadBalancerArn* and *DNSName* returned in the response, as you will need them in the next steps.

Listeners

Your ALB needs at least one listener to start receiving traffic. In this case, you are going to deploy two listeners, for HTTP and HTTPS traffic respectively. You will also configure the HTTP listener to redirect to the HTTPS one.

HTTPS Listener

This type of listener requires an X.509 server certificate so clients can establish a Transport Layer Security (TLS) connection with the ALB.

> Note: This guide uses a self-signed certificate. By deploying a self-signed certificate on an endpoint, modern browsers will warn you about reaching an insecure web site. The best practice is to use a custom domain name for your ALB and use a certificate issued by a Certificate Authority (CA). This guide uses AWS Certificate Manager (ACM), which relies on the Amazon Trust Services LLC Certificate Authority. You can use Route 53 for the domain name and an alias record to point to the ALB, as explained in the Routing traffic to an ELB load balancer guide.

1. Run the following commands to generate a self-signed certificate. Fill in the requested information and make sure to introduce the full DNS name of the ALB as the Common Name (CN) when prompted.

```
openssl genrsa 2048 > privatekey.pem
openssl req -new -key privatekey.pem -out csr.pem
openssl x509 -req -days 1200 -in csr.pem -signkey
privatekey.pem -out public.crt
openssl x509 -in public.crt -out cert.pem
```

2. Import the certificate to AWS Certificate Manager (ACM) with the following AWS CLI command:

```
aws acm import-certificate --certificate fileb://cert.pem  --
private-key fileb://privatekey.pem
```

Take note of the Amazon Resource Name (ARN) of the certificate, as you will need it in the next step.

3. After you have a self-signed certificate imported into ACM, or one issued by ACM itself, you can create the HTTPS listener with this AWS CLI command.

```
aws elbv2 create-listener --region your-region \
    --load-balancer-arn your-alb-arn \
    --protocol HTTPS \
    --port 443 \
    --certificates CertificateArn=your-certificate-arn \
    --default-actions 'Type=forward,TargetGroupArn=your-mwaa-
target-group-arn'
```

4. Now, create an HTTP listener that redirects to the HTTPS endpoint with the same host, path, and query. Write down the listener ARN.

```
aws elbv2 create-listener --region your-region \
    --load-balancer-arn your-alb-arn
    --protocol HTTPS \
    --port 443 \
    --certificates CertificateArn=your-certificate-arn \
    --default-actions
'Type=redirect,RedirectConfig={Protocol=HTTPS,Port=443,Host="
#{host}",Path="/#{path}",Query="#{query}",StatusCode=HTTP_302
}'
```

At this point, the Airflow UI should be accessible by using any of the methods described in the [Creating an Apache Airflow web login token](#) guide. However, not all of your users will have the AWS CLI or Python installed and configured with the right permissions.

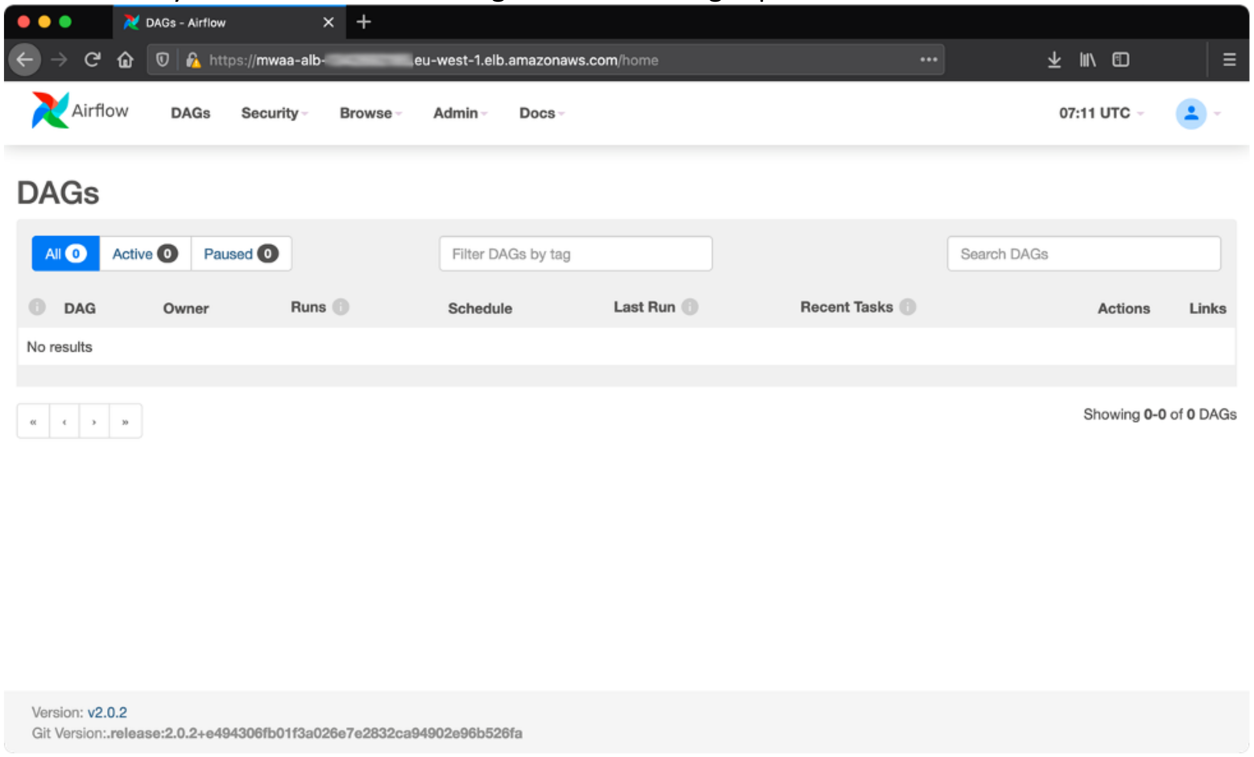


Figure 8 – View of the Apache Airflow web user interface on a browser

So far, you have deployed the following architecture:

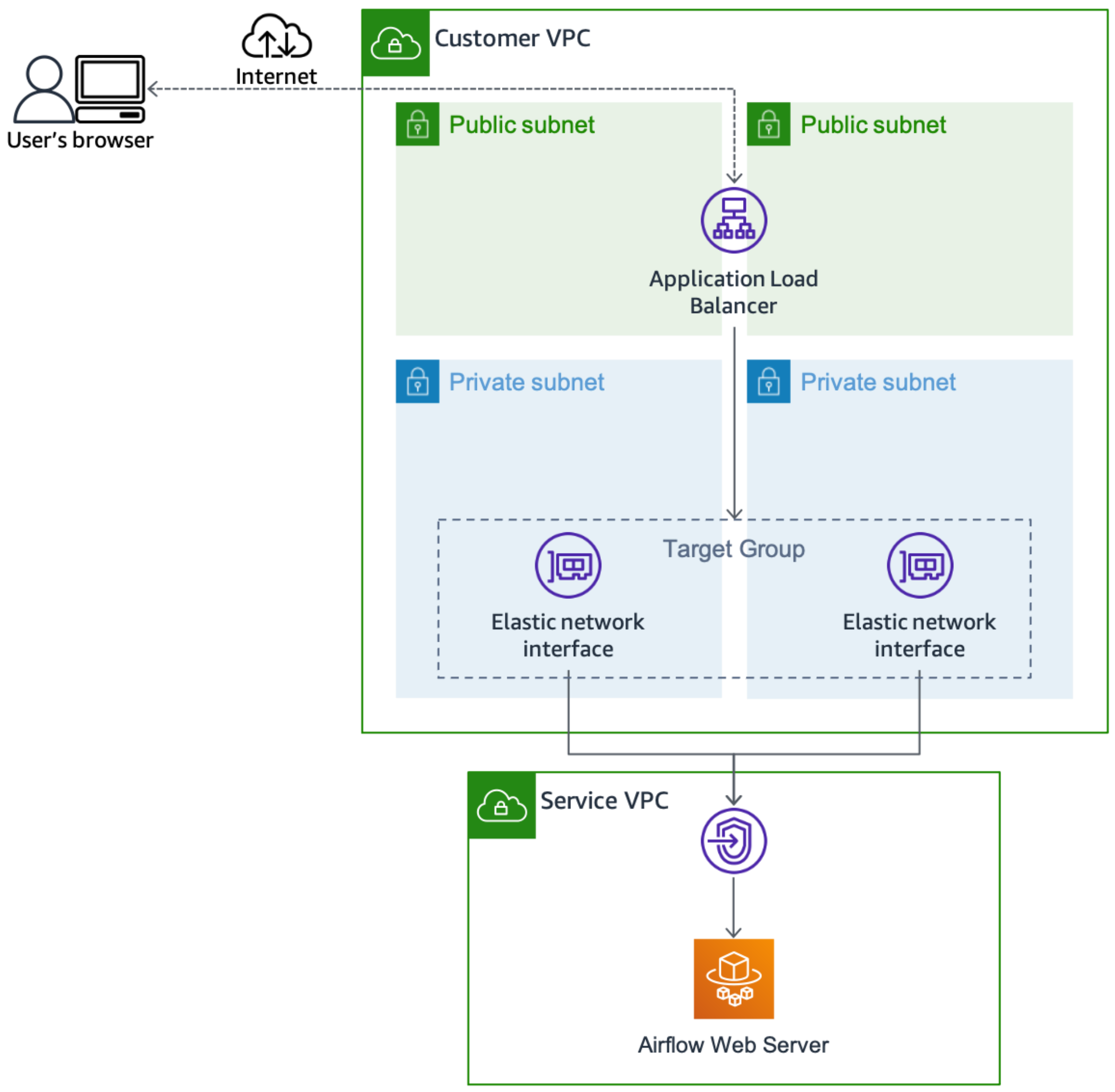


Figure 9 – View of the architecture deployed so far

The next steps of this guide focus on how to incorporate a seamless login to Airflow by using the ALB you just deployed, Amazon Cognito, an external IdP, and AWS Lambda.

Using federated identities to authenticate Amazon MWAA users

Identity federation is a system of trust between two parties for the purpose of authenticating users and conveying information needed to authorize their access to resources. In this system, an identity provider (IdP) is responsible for user authentication, and a service provider (SP), such as a service or an application, controls access to resources.

AWS Elastic Load Balancing (ELB) allows you to delegate authentication from your application to an ALB using the OpenID Connect (OIDC) authentication protocol.

Amazon Cognito

Many of the existing Identity Providers (IdPs) (such as Okta, Auth0, and Azure AD) support OIDC, so you could integrate them directly with an ALB. However, setting up this integration, configuring claims, and verifying tokens usually entails extra steps and added complexity. Amazon Cognito simplifies and harmonizes the configuration for any supported IdP. Additionally, ALB integrates directly with Amazon Cognito user pools, reducing the overall number of steps you need to complete to get the solution running. Using the ALB – Cognito integration, you can directly reference a Cognito user pool identifier and an App Client from your ALB listener rules and users will be redirected to the IdP login page for your application. Amazon Cognito user pools also allow logins via federated IdPs, and offer support for Security Assertion Markup Language (SAML) and OIDC alongside some popular social identity providers. This guide uses an Amazon Cognito user pool with a federated IdP as the authentication layer. This way, the proposed solution allows changing or adding federated IdPs without needing to change the rest of the AWS components used.

Creating and configuring an Amazon Cognito user pool

Follow these steps to create and configure an Amazon Cognito user pool.

1. Create a user pool by running the following AWS CLI command. Note the user pool id in the response, as you will need it in the next step. This user pool uses the default configuration for password policies and other features. You can change them if you intend to manage users in your user pool instead of or in addition to using federated login.

```
aws cognito-idp create-user-pool --region your-region \
    --pool-name mwaa-users
```

2. Create a domain for this user pool with a custom prefix, so the IdP and the ALB can communicate with it.

Although instructions are not included in this guide, you can use your own full domain name with an associated certificate stored in AWS Certificate Manager. For this approach, you also need the ability to add an alias record to the domain's hosted zone after it's associated with this user pool.

```
aws cognito-idp create-user-pool-domain --region your-region
\
    --domain mwaa-env \
    --user-pool-id your-user-pool-id
```

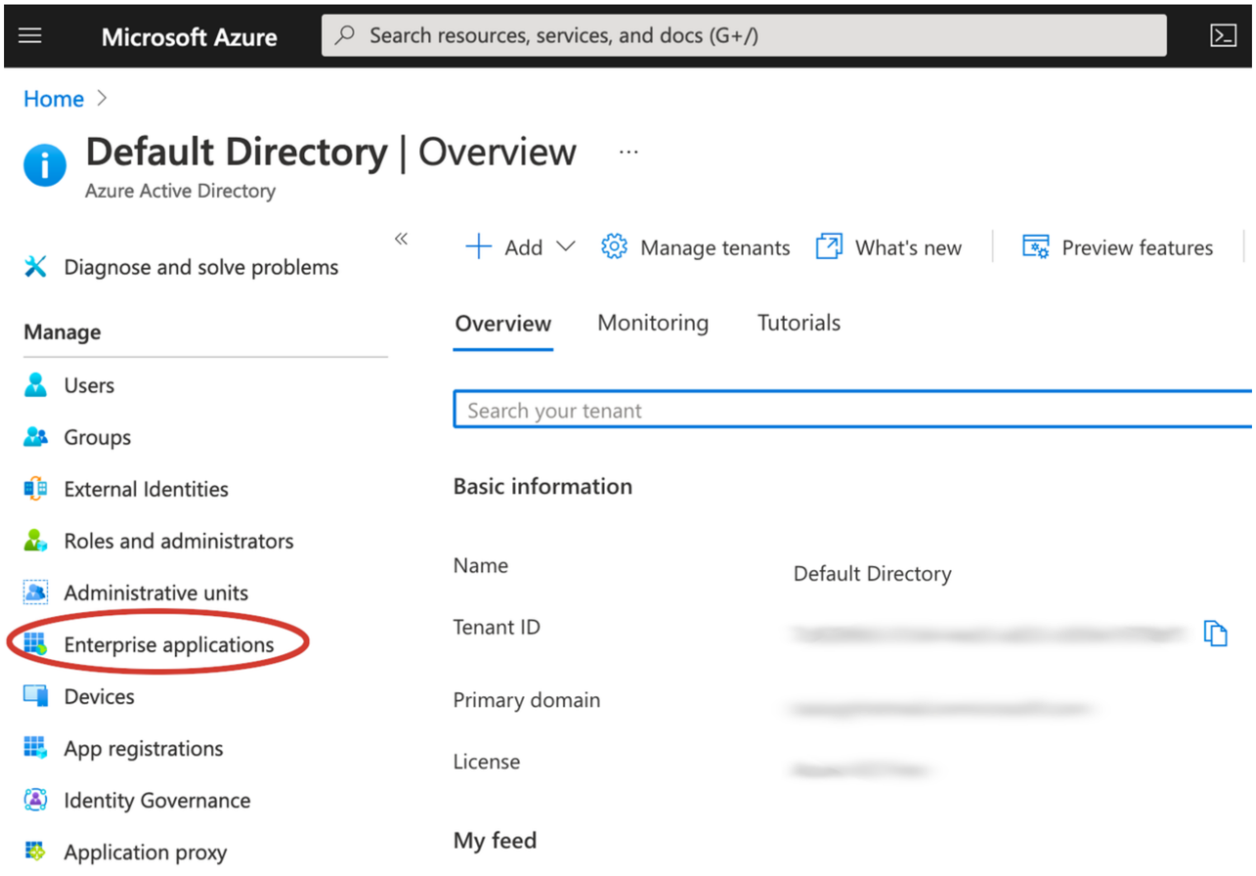
Next, configure the external IdP to integrate with Amazon Cognito and provide claims that will be used to determine the user's permissions to access the Amazon MWAA environment.

Configuring Azure AD as federated Identity Provider

This guide uses Azure AD with SAML integration to illustrate how to integrate with Amazon Cognito and to issue SAML tokens. These SAML tokens will contain claims that will be used to determine users' permissions to access the Amazon MWAA environment.

Creating an enterprise application

1. In the Azure AD console, go to the directory where you want to create your application, choose **Enterprise applications**, and choose **New application**.



Microsoft Azure | Search resources, services, and docs (G+/)

Home >

Default Directory | Overview ...

Azure Active Directory

+ Add | Manage tenants | What's new | Preview features

Diagnose and solve problems

Manage

- Users
- Groups
- External Identities
- Roles and administrators
- Administrative units
- Enterprise applications
- Devices
- App registrations
- Identity Governance
- Application proxy

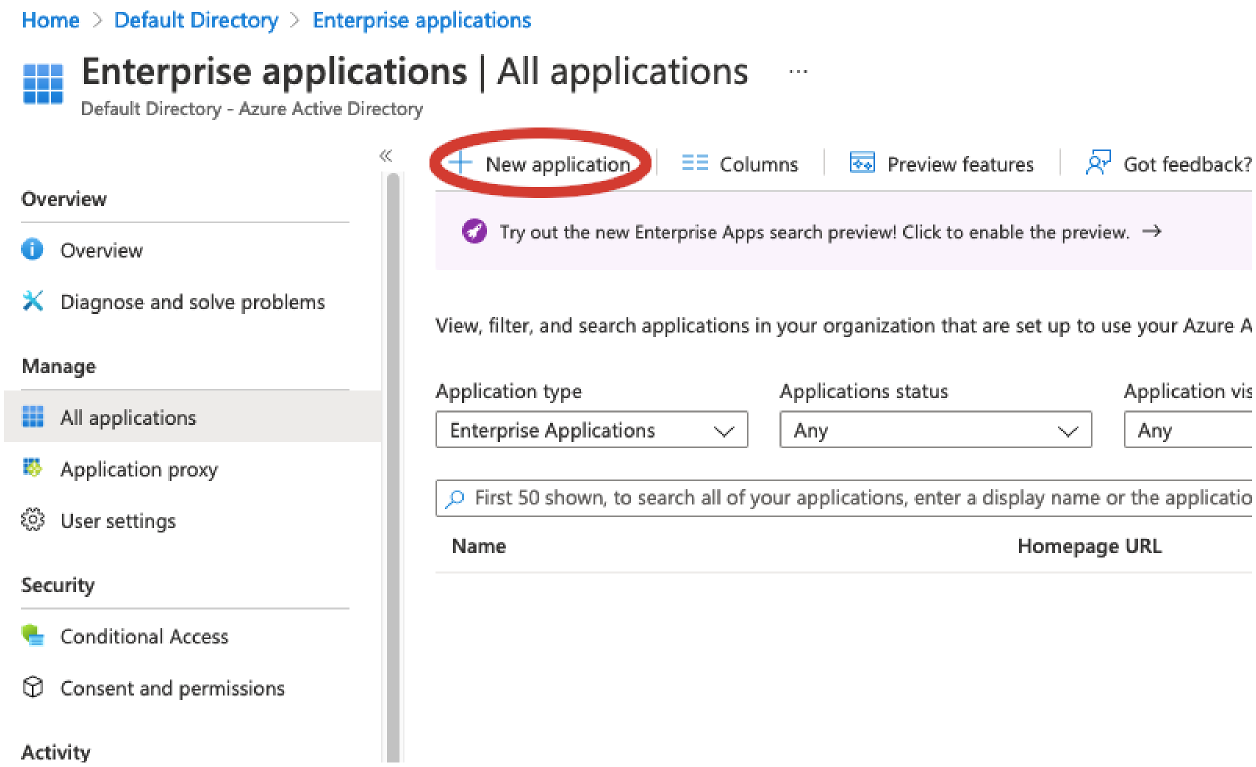
Overview | Monitoring | Tutorials

Search your tenant

Basic information

Name	Default Directory
Tenant ID	[redacted]
Primary domain	[redacted]
License	[redacted]

My feed



Home > Default Directory > Enterprise applications

Enterprise applications | All applications ...

Default Directory - Azure Active Directory

+ New application | Columns | Preview features | Got feedback?

Try out the new Enterprise Apps search preview! Click to enable the preview. →

Overview

- Overview
- Diagnose and solve problems

Manage

- All applications
- Application proxy
- User settings

Security

- Conditional Access
- Consent and permissions

Activity

View, filter, and search applications in your organization that are set up to use your Azure A

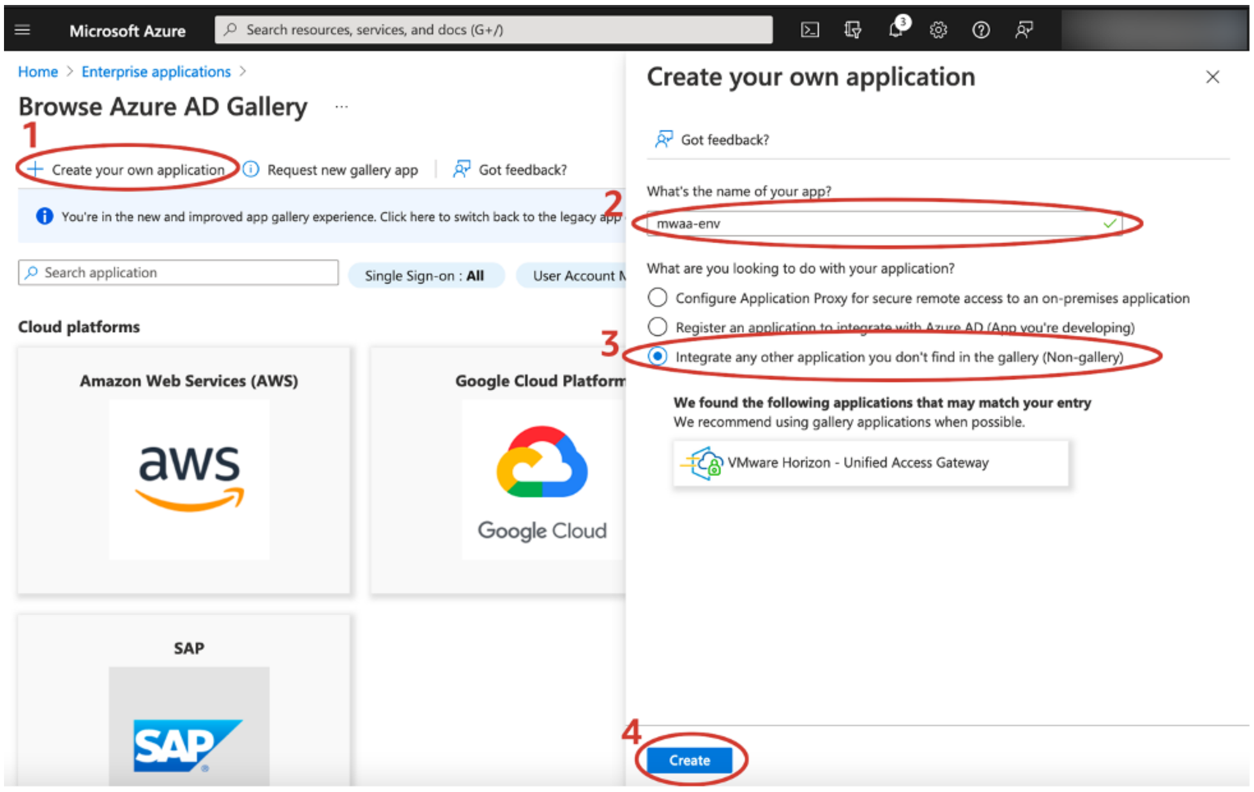
Application type	Applications status	Application vis
Enterprise Applications	Any	Any

First 50 shown, to search all of your applications, enter a display name or the applicatio

Name	Homepage URL

Figure 10 – Creating a new enterprise application in Azure AD

2. Pick (1) **Create your own application** and (2) introduce a name for it. Toggle the option (3) **Integrate any other application you don't find in the gallery (Non-gallery)**. Select (4) **Create**.


Microsoft Azure
Search resources, services, and docs (G+/)
Home > Enterprise applications >
Browse Azure AD Gallery
1
Create your own application
Request new gallery app
Got feedback?
You're in the new and improved app gallery experience. Click here to switch back to the legacy app
Search application
Single Sign-on : All
User Account M
Cloud platforms
Amazon Web Services (AWS)
Google Cloud Platform
SAP
Create your own application
Got feedback?
2
3
4
Create

What's the name of your app?

mwaa-env

What are you looking to do with your application?

- ○ Configure Application Proxy for secure remote access to an on-premises application
- ○ Register an application to integrate with Azure AD (App you're developing)
- ● Integrate any other application you don't find in the gallery (Non-gallery)

We found the following applications that may match your entry
We recommend using gallery applications when possible.

VMware Horizon - Unified Access Gateway

Figure 11 – Selecting the type of enterprise application in Azure AD

Creating users and groups

You need to add the users or groups in your directory to your application so they can access it. Additionally, you need a way to authorize users to access the Amazon MWAA environment with specific roles. There are different ways to do this, such as by using user attributes, roles, or security groups. In this guide, you will use security groups.

1. Go to the directory where you created the enterprise application and choose **Add**, and then **Group**.

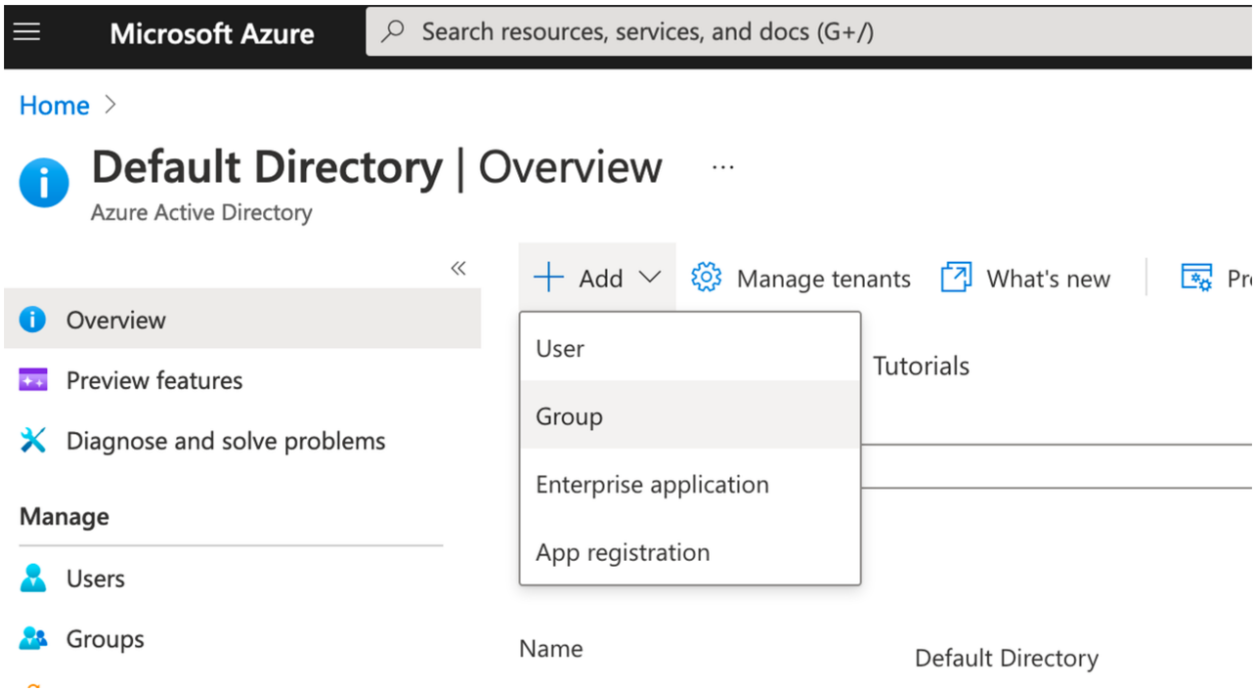
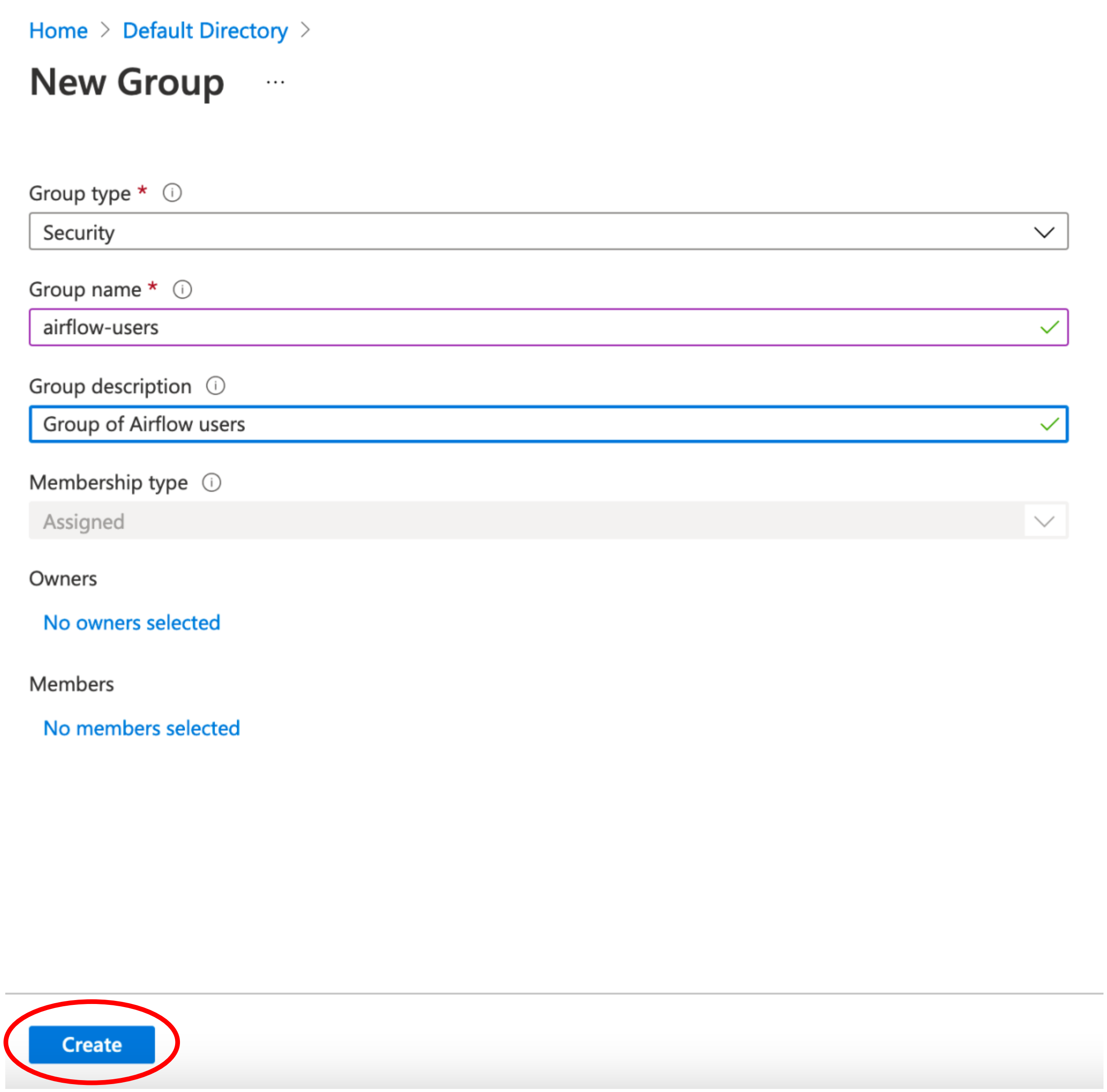


Figure 12 – Creating a group in Azure AD

2. Enter a name, such as *airflow-users*, and a description. Select **Create**.



Home > Default Directory >

New Group ...

Group type * Security

Group name * airflow-users

Group description Group of Airflow users

Membership type Assigned

Owners

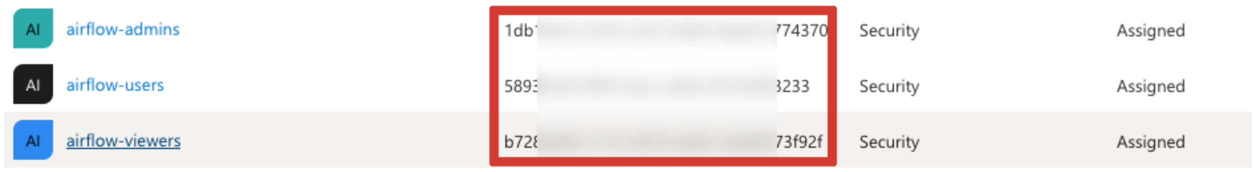
No owners selected

Members

No members selected

Figure 13 – New group window in Azure AD

3. Repeat the process and create two more groups: `airflow-admins` and `airflow-viewers`. When you navigate to the Groups overview in your directory, you should see something like in Figure 14. Take note of the Object Id of each group, as you will use them later on to map them to a corresponding IAM role.



AI	airflow-admins	1db[redacted]74370	Security	Assigned
AI	airflow-users	589[redacted]233	Security	Assigned
AI	airflow-viewers	b72[redacted]73f92f	Security	Assigned

Figure 14 – Groups list view in Azure AD

For testing purposes, create three users in the directory: airflow-user, airflow-admin, and airflow-viewer, adding each of them to the respective group you just created.

4. Choose **Add**, and then **User**. Introduce `airflow-xxxx` as the user's name, and `Airflow User` as name. Choose **0 groups selected** to add the user to the `airflow-xxxxs` group. On the Groups window, select the corresponding group and choose **Select**. Select **Create** and repeat for the remaining users.

Now, add these users or groups to the enterprise application.

5. Navigate to the Amazon MWAA enterprise application and choose **Users** *and* **groups**. There, choose **Add user/group** and then select **None Selected** under **Users**. Select each of the users you just created, choose **Select,** and then choose **Assign**.

Configuring SAML Single Sign-On in Azure AD

You need to configure SAML single-sign-on so the directory users can get access to the enterprise application.

1. Navigate to the Amazon MWAA enterprise application you created earlier and choose **Set up single sign on**, and then choose **SAML**.

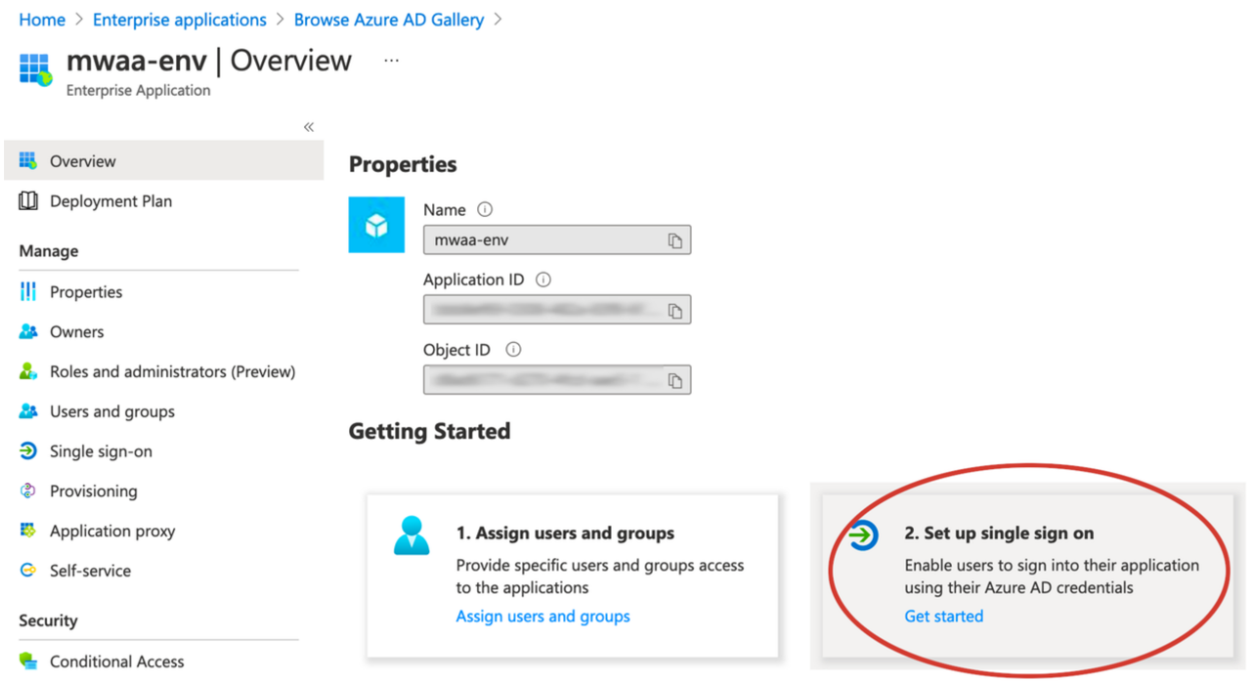
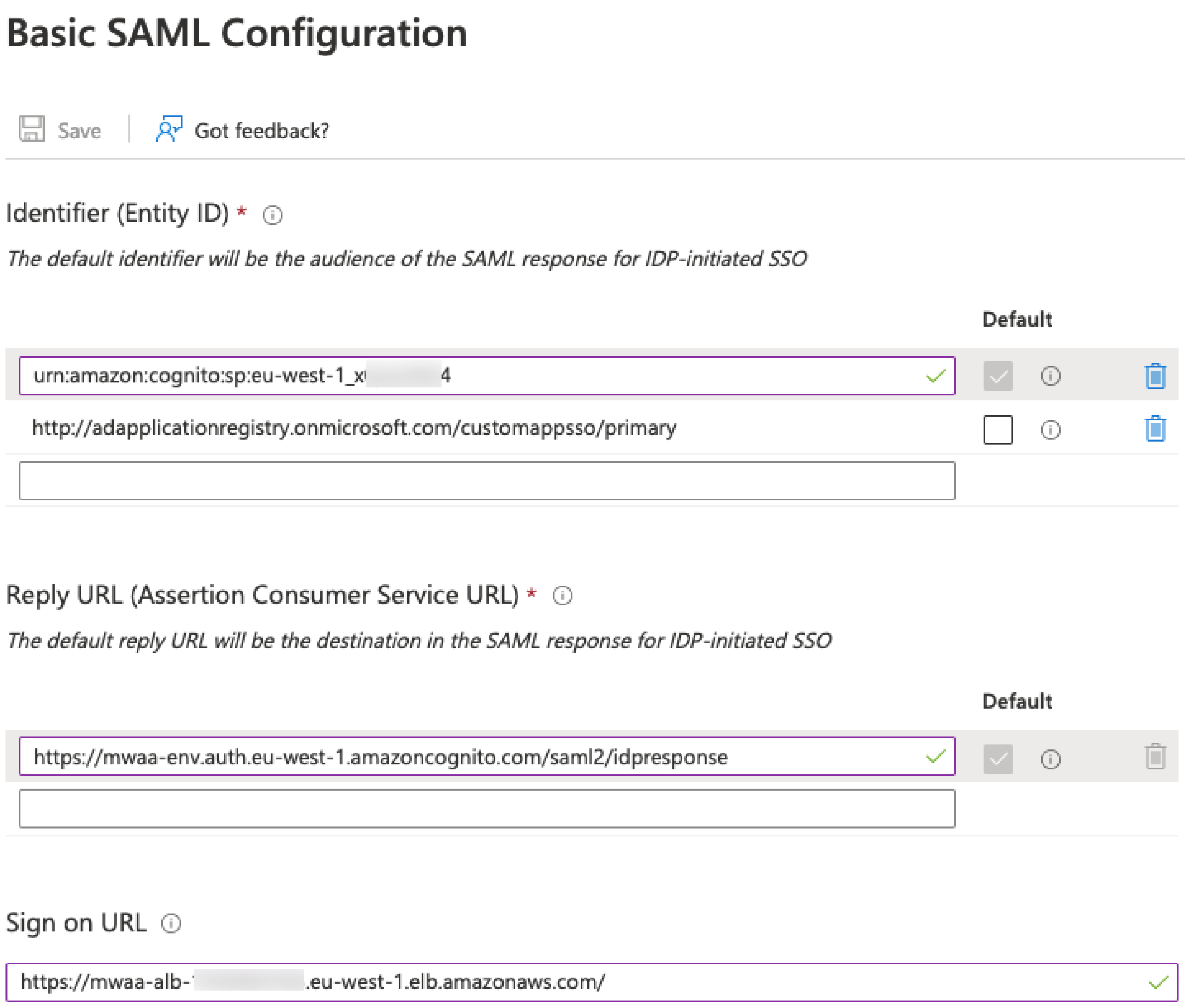


Figure 15 – Setting up single sign on in Azure AD

2. Click the **Edit** button in the **Basic SAML Configuration** section. In the field Identifier (Entity ID), introduce the Cognito user pool Service Provider (SP) urn, which is in the form `urn:amazon:cognito:sp:`*`your-user-pool-id`*. Set it as default.

3. In the Reply URL introduce a URL as this: `https://your-cognito-domain-prefix.auth.your-region.amazoncognito.com/saml2/idpresponse`. This is the URL where the IdP sends the response to a SAML authentication request.
4. In the Sign on URL, introduce your ALB URL so that the IdP allows sign-on requests from it, click **Save**, and close the **Basic SAML Configuration** window.


Basic SAML Configuration

Save | Got feedback?

Identifier (Entity ID) *

The default identifier will be the audience of the SAML response for IDP-initiated SSO

	Default
urn:amazon:cognito:sp:eu-west-1_x[redacted]4	✓
http://adapplicationregistry.onmicrosoft.com/customappsso/primary	☐

Reply URL (Assertion Consumer Service URL) *

The default reply URL will be the destination in the SAML response for IDP-initiated SSO

	Default
https://mwaa-env.auth.eu-west-1.amazoncognito.com/saml2/idpresponse	✓

Sign on URL

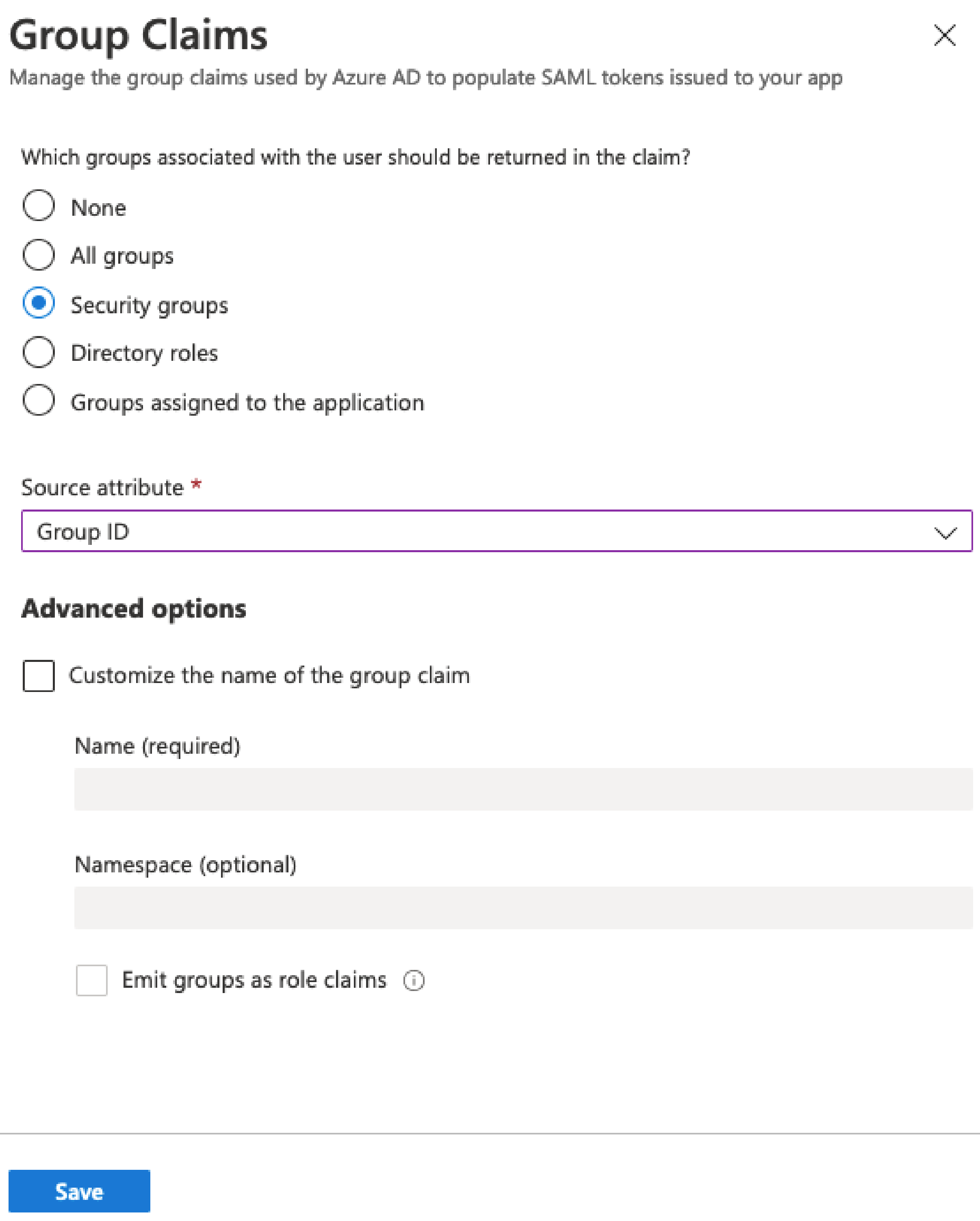
https://mwaa-alb-[redacted].eu-west-1.elb.amazonaws.com/

Figure 16 – Basic SAML configuration window in Azure AD

Configuring user attributes and claims

Following the steps in this section, you configure the claims that will be included in the SAML token issued by the IdP. These claims are crucial for authorizing the users to access the Amazon MWAA environment. In this guide, you will use security groups, so you can use the groups that you created earlier to determine the Airflow role that the users will assume when accessing the web UI.

1. In the Azure AD enterprise application, select the **Single sign-on** button on the left of the screen, and then the **Edit** button in the **User Attributes & Claims** section. In the pop-up window, select **Security groups,** and choose **Group ID** as the **Source attribute**. Select **Save**.



Group Claims

Manage the group claims used by Azure AD to populate SAML tokens issued to your app

Which groups associated with the user should be returned in the claim?

- [ ] None
- [ ] All groups
- [✓ Security groups]
- [ ] Directory roles
- [ ] Groups assigned to the application

Source attribute *

Group ID

Advanced options

- [ ] Customize the name of the group claim

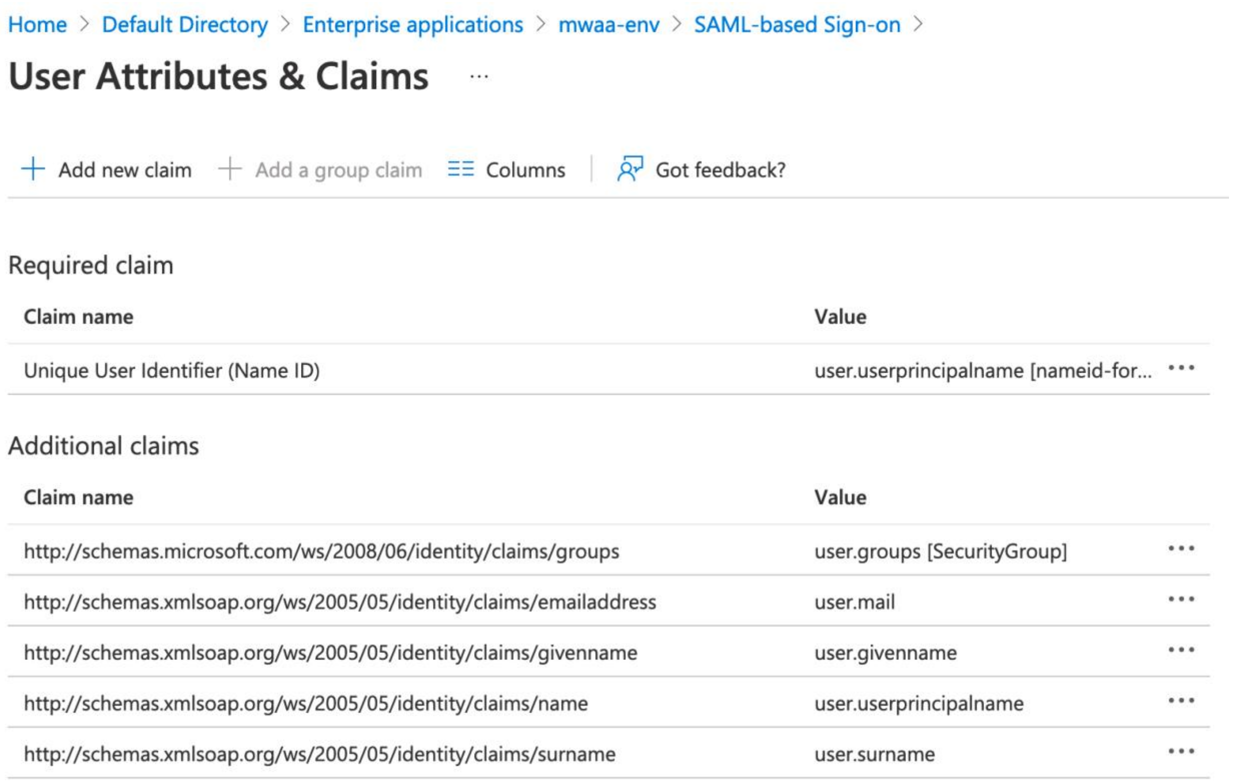
Name (required)

Namespace (optional)

- [ ] Emit groups as role claims

Figure 17 – Configuring group claims for an enterprise application in Azure AD

2. You should see the default claims configured to be issued by Azure AD as in Figure 18.



Home > Default Directory > Enterprise applications > mwaa-env > SAML-based Sign-on >

User Attributes & Claims ...

+ Add new claim + Add a group claim Columns Got feedback?

Required claim

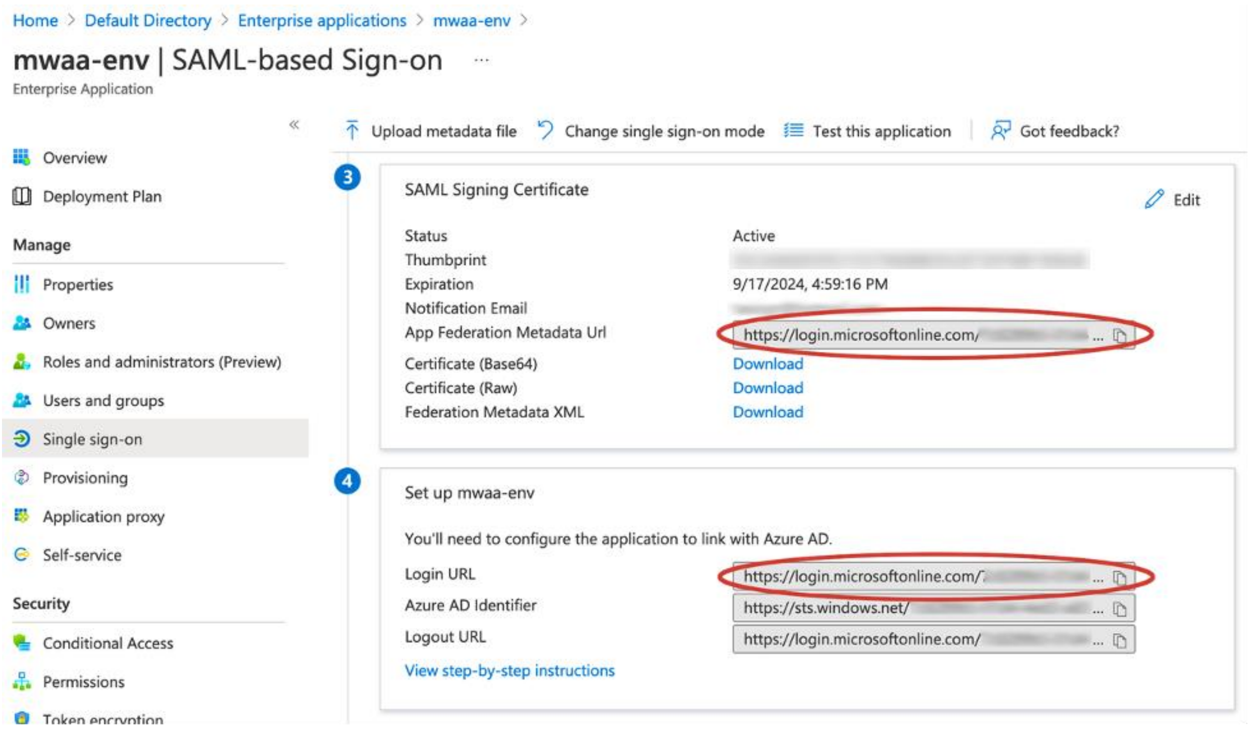
Claim name	Value
Unique User Identifier (Name ID)	user.userprincipalname [nameid-for...

Additional claims

Claim name	Value
http://schemas.microsoft.com/ws/2008/06/identity/claims/groups	user.groups [SecurityGroup]
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress	user.mail
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname	user.givenname
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	user.userprincipalname
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname	user.surname

Figure 18 – Default SAML user attributes and claims in Azure AD

3. Close the window, copy the **App Federation Metadata Url**, and write down the **Login URL**.



Home > Default Directory > Enterprise applications > mwaa-env >

mwaa-env | SAML-based Sign-on ...

Enterprise Application

3 SAML Signing Certificate

Status	Active
Thumbprint	
Expiration	9/17/2024, 4:59:16 PM
Notification Email	
App Federation Metadata Url	https://login.microsoftonline.com/ ...
Certificate (Base64)	Download
Certificate (Raw)	Download
Federation Metadata XML	Download

4 Set up mwaa-env

You'll need to configure the application to link with Azure AD.

Login URL	https://login.microsoftonline.com/ ...
Azure AD Identifier	https://sts.windows.net/ ...
Logout URL	https://login.microsoftonline.com/ ...

View step-by-step instructions

Figure 19 – App federation metadata URL and login URL highlighted in the Azure AD SAML single sign-on detailed view

The IdP is now ready. You can finish configuring the Amazon Cognito user pool.

Configuring Cognito to use the external IdP

Creating Amazon Cognito custom attributes

To receive the claims issued by the IdP, use Cognito custom attributes. In this section, you are going to create custom attributes where you can receive the user name and the groups they belong to.

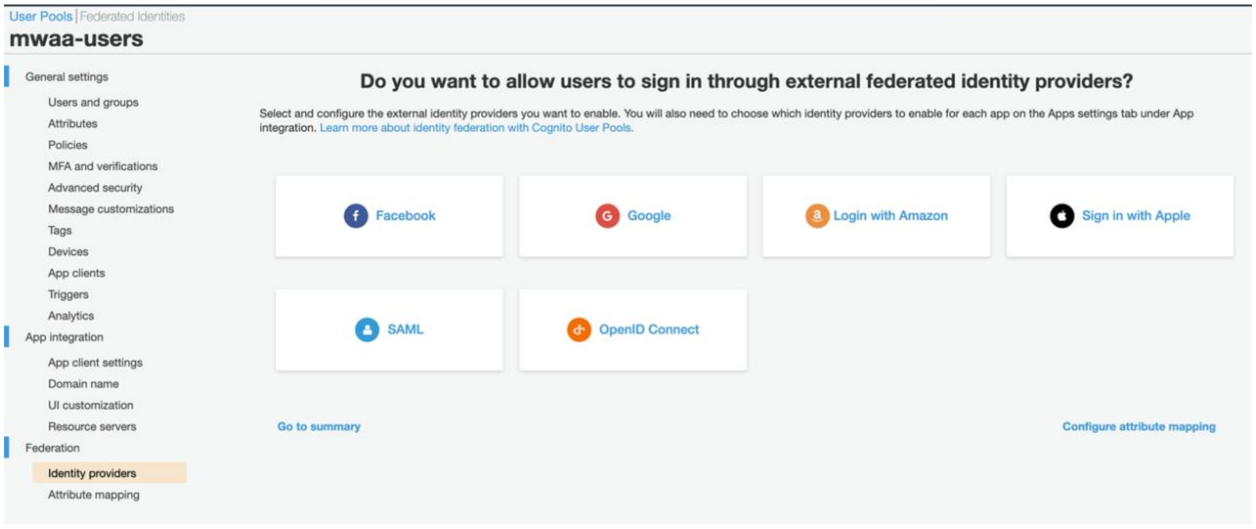
1. Execute the following AWS CLI command.

```
aws cognito-idp add-custom-attributes --region your-region \
    --user-pool-id your-user-pool-id \
    --custom-attributes Name=idp-
groups,AttributeDataType=String,Mutable=true,Required=false
Name=idp-
name,AttributeDataType=String,Mutable=true,Required=false
```

2. Now, you can add the external IdP to the user pool and map the SAML claim to this custom attribute.

Adding the external IdP to the user pool

1. In the AWS Management Console, navigate to Amazon Cognito and choose **Manage user pools**. Select the Amazon MWAA user pool you created earlier and choose **Identity providers** (under **Federation**).

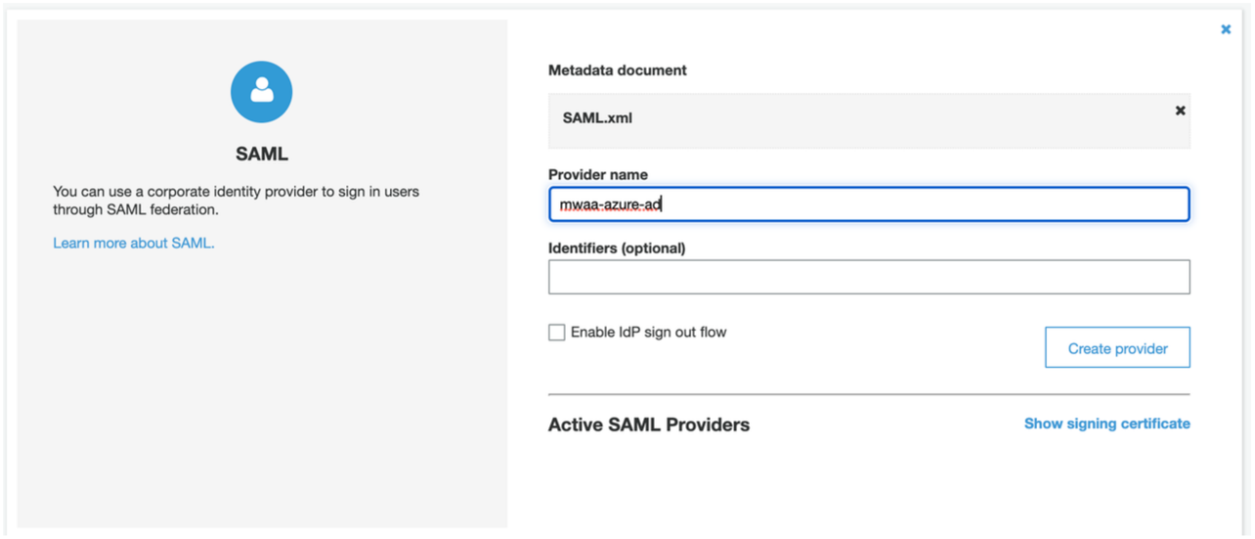


Do you want to allow users to sign in through external federated identity providers?

Select and configure the external identity providers you want to enable. You will also need to choose which identity providers to enable for each app on the Apps settings tab under App integration. Learn more about identity federation with Cognito User Pools.

Figure 20 – Identity providers view for the mwaa-users Cognito user pool

2. Choose SAML and then paste the IdP **App Federation Metadata Url** you copied earlier. Introduce `mwaa-azure-ad` as the name for your identity provider and choose **Create provider**.

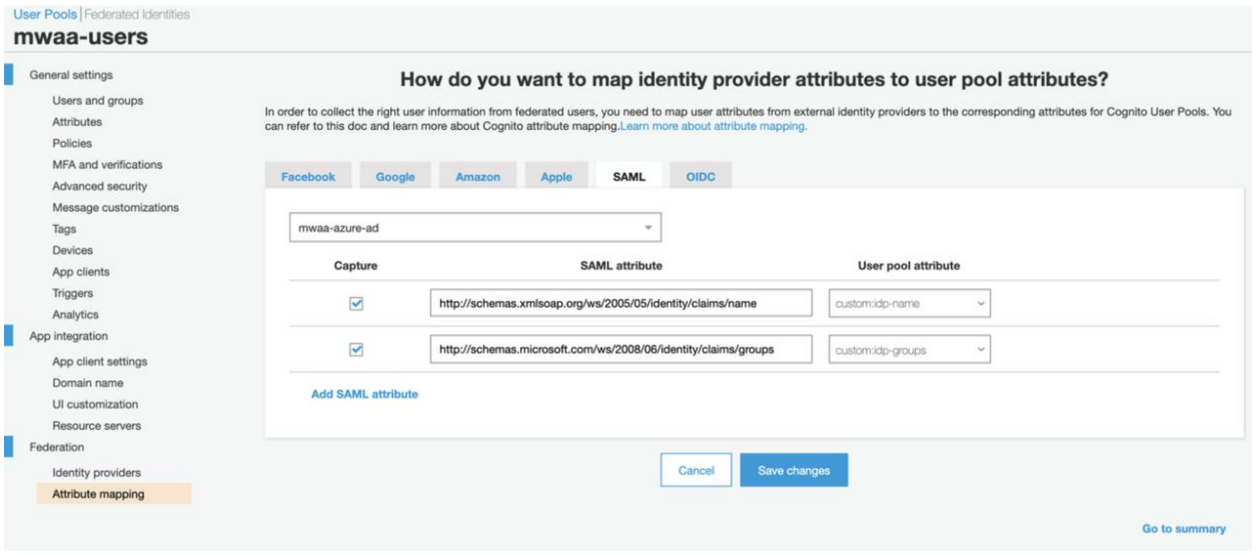


Mapping SAML attributes to Cognito custom attributes

1. In the same window where you configured the SAML IdP, choose **Configure attribute mapping**. Make sure the IdP you registered is selected in the drop-down list and choose **Add SAML attribute**. Paste `http://schemas.microsoft.com/ws/2008/06/identity/claims/groups` in the text box and select `custom:idp-groups` in the drop-down list beside it.

2. Repeat the previous step with `http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name` and `custom:idp-name`. Save the changes.

If you used a different type of claim, paste its claim name instead; you can find it in the SAML claims configuration in your IdP.



How do you want to map identity provider attributes to user pool attributes?

In order to collect the right user information from federated users, you need to map user attributes from external identity providers to the corresponding attributes for Cognito User Pools. You can refer to this doc and learn more about Cognito attribute mapping. Learn more about attribute mapping.

mwaa-azure-ad

Capture	SAML attribute	User pool attribute
✓	http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	custom:idp-name
✓	http://schemas.microsoft.com/ws/2008/06/identity/claims/groups	custom:idp-groups

Creating the Amazon Cognito user pool app client

To integrate an application with Amazon Cognito (Amazon MWAA in this case), you need an app client. This app client will also serve to integrate with the external IdP.

1. Run the following AWS CLI command to create the app client:

```
aws cognito-idp create-user-pool-client --region your-region
\
    --user-pool-id your-user-pool-id \
    --client-name mwaa-app \
    --generate-secret \
    --read-attributes custom:idp-groups custom:idp-name \
    --write-attributes custom:idp-groups custom:idp-name \
    --explicit-auth-flows ALLOW_USER_SRP_AUTH
ALLOW_REFRESH_TOKEN_AUTH \
    --supported-identity-providers your-identity-provider-
name \
    --callback-urls https://your-alb-dns-
name/oauth2/idpresponse \
    --logout-urls https://your-alb-dns-name/logout/close \
    --default-redirect-uri https://your-alb-dns-name/ \
    --allowed-o-auth-flows code \
    --allowed-o-auth-scopes openid \
    --allowed-o-auth-flows-user-pool-client
```

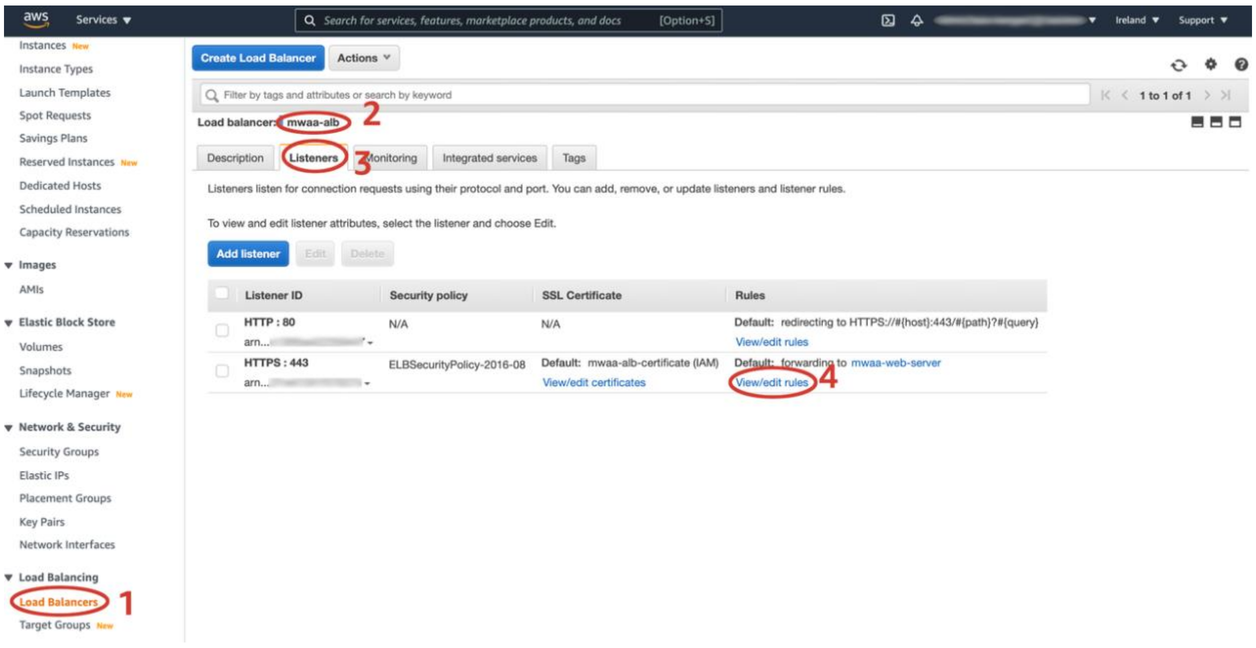
You can find more information about the values for the input parameters in the [Authenticate users using an Application Load Balancer documentation].

Configuring Cognito authentication on the ALB

So far, the ALB is configured to forward the traffic to the Amazon MWAA web servers. By configuring Cognito authentication, the ALB will first try to get a valid token from Cognito before reaching the target group. This way, you make sure that no unauthenticated traffic reaches the Amazon MWAA endpoint.

Adding authentication to the existing ALB rule

1. In the EC2 console, select Load Balancers. Select the Amazon MWAA ALB and choose the **Listeners** tab. In the HTTPS listener, choose **View/edit rules**.



Load balancer: mwaa-alb

Listeners listen for connection requests using their protocol and port. You can add, remove, or update listeners and listener rules.

To view and edit listener attributes, select the listener and choose Edit.

Listener ID	Security policy	SSL Certificate	Rules
HTTP : 80 arn... [redacted]	N/A	N/A	Default: redirecting to HTTPS://#{host}:443/#{path}?#{query} View/edit rules
HTTPS : 443 arn... [redacted]	ELBSecurityPolicy-2016-08	Default: mwaa-alb-certificate (IAM) View/edit certificates	Default: forwarding to mwaa-web-server View/edit rules

Figure 21 – Accessing ALB rules in the AWS Management Console

2. Edit the existing rule by selecting the pencil icon at the top left and then the pencil icon next to the rule. Select the **Add action** drop-down and select **Authenticate…**

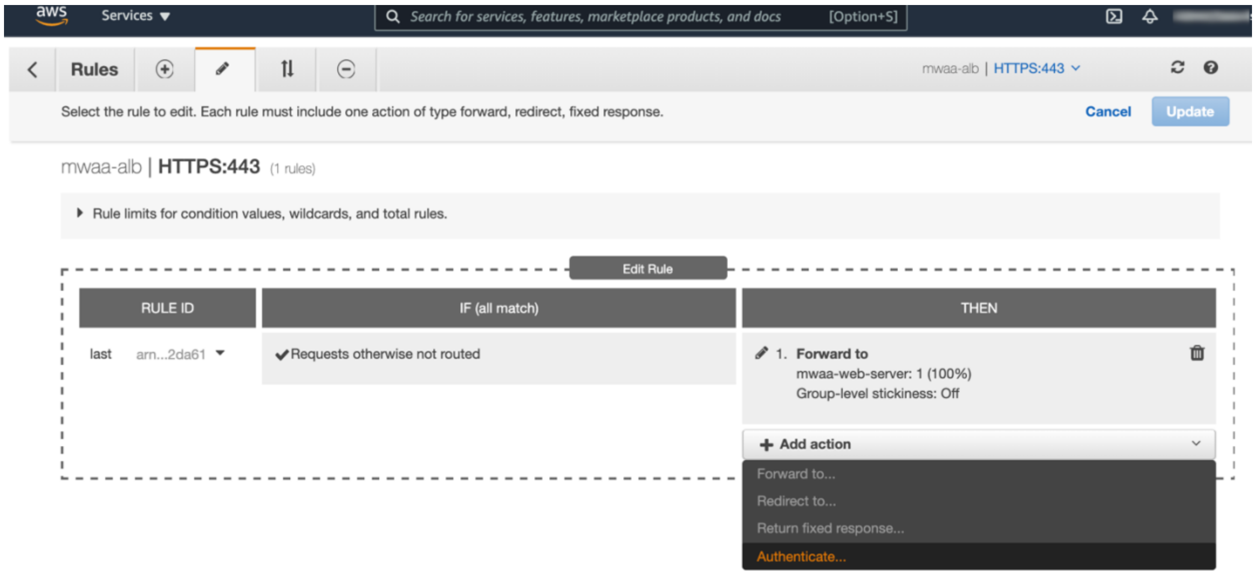

Services
Search for services, features, marketplace products, and docs [Option+S]
Rules
mwaa-alb | HTTPS:443
Select the rule to edit. Each rule must include one action of type forward, redirect, fixed response.
Cancel
Update
mwaa-alb | HTTPS:443 (1 rules)
Rule limits for condition values, wildcards, and total rules.
Edit Rule
RULE ID
IF (all match)
THEN
last arn...2da61
Requests otherwise not routed
1. Forward to
mwaa-web-server: 1 (100%)
Group-level stickiness: Off
Add action
Forward to...
Redirect to...
Return fixed response...
Authenticate...

Figure 22 – Adding an authentication action to an ALB listener rule

3. Select the appropriate user pool and app client from the drop-down lists and save. Choose **Update** and test by accessing the ALB URL on a browser.

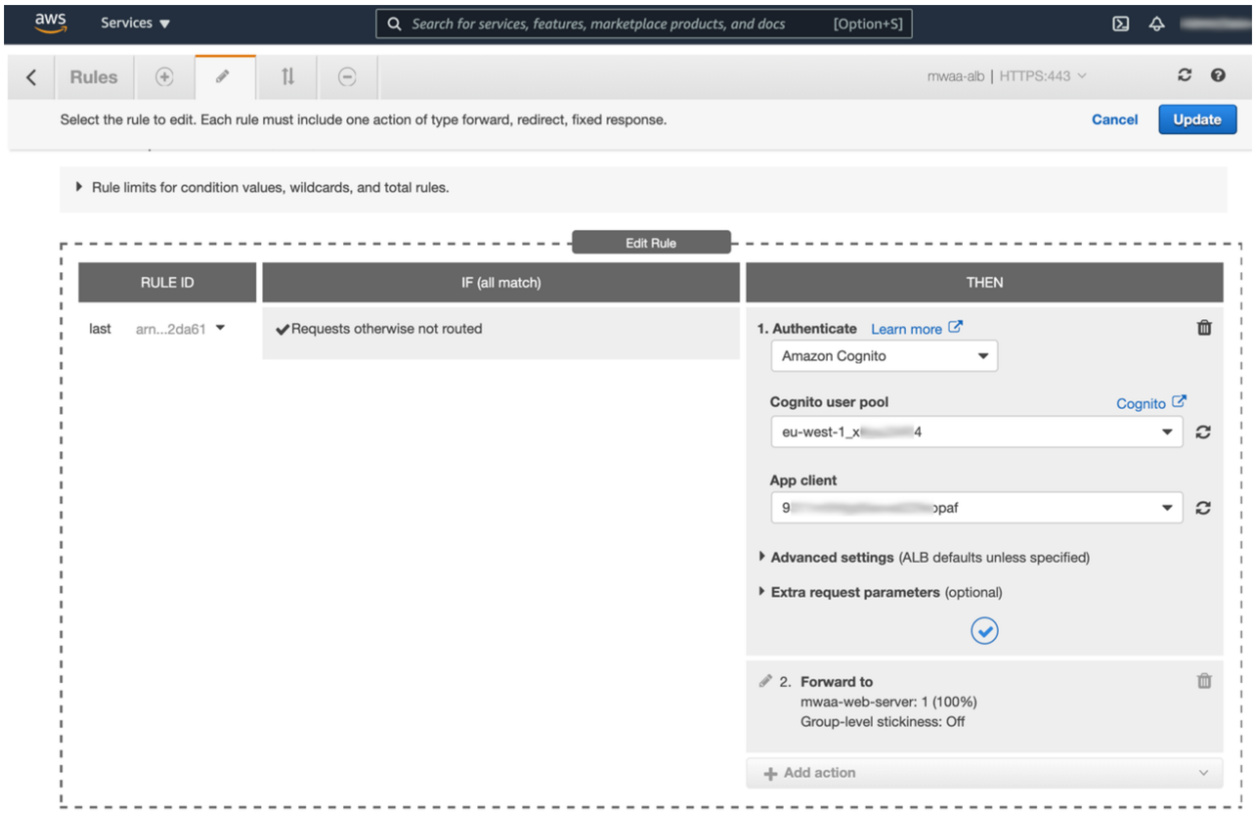

Services
Search for services, features, marketplace products, and docs [Option+S]
Rules
mwaa-alb | HTTPS:443
Select the rule to edit. Each rule must include one action of type forward, redirect, fixed response.
Cancel
Update
Rule limits for condition values, wildcards, and total rules.
Edit Rule
RULE ID
IF (all match)
THEN
last arn...2da61
Requests otherwise not routed
1. Authenticate Learn more
Amazon Cognito
Cognito user pool
Cognito
eu-west-1_x[redacted]4
App client
9[redacted]opaf
Advanced settings (ALB defaults unless specified)
Extra request parameters (optional)
2. Forward to
mwaa-web-server: 1 (100%)
Group-level stickiness: Off
Add action

Figure 23 – Configuring the authentication action to use the Cognito user pool

Your browser should redirect you to the Azure AD authentication page. There, introduce the credentials for one of the Airflow users you created earlier. You should reach the Airflow website.
At this point, the architecture looks like the following:

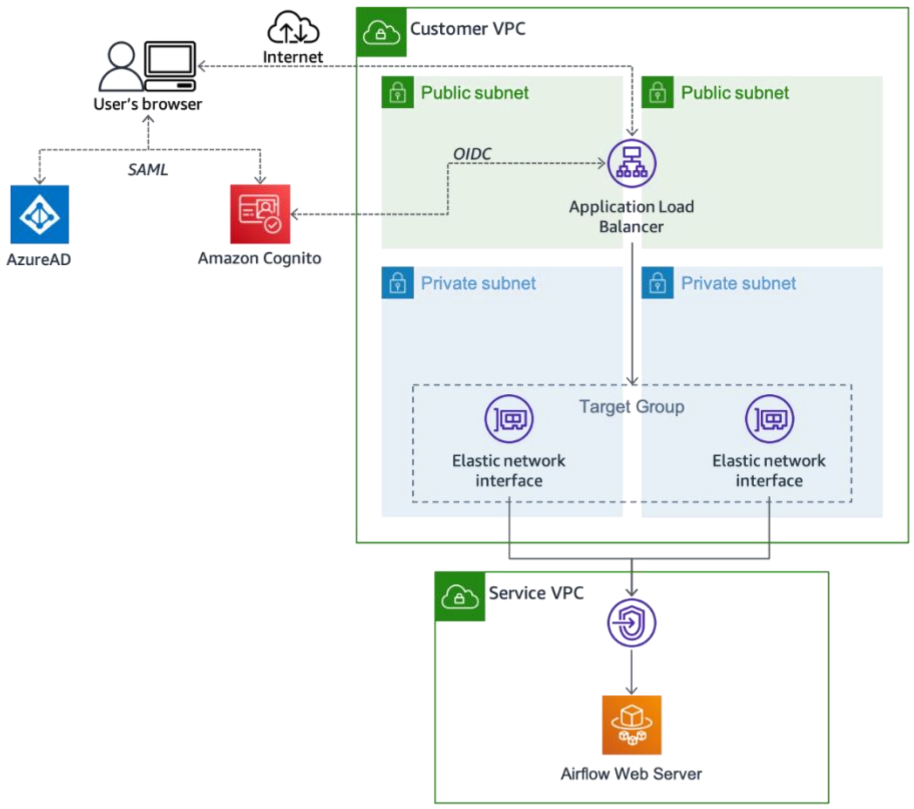


Figure 24 – Architecture diagram after configuring Cognito authentication on the ALB

Authenticating and authorizing Airflow users

Now, as the Creating an Apache Airflow web login token documentation describes, you need to have a way to seamlessly redirect the users to a URL that includes an Airflow *web login token*. This is a JSON Web Token (JWT) that carries claims for Airflow to authenticate a user against an environment and grant them the appropriate **Airflow role**. For this purpose, you will create a **Lambda function** that is triggered by the ALB.

Airflow roles and Amazon MWAA

Amazon MWAA works with the default Airflow Roles: `Admin`, `Op`, `User`, `Viewer`, and `Public`. These roles are described in the Airflow documentation. At the time of writing, Amazon MWAA does not support custom Apache Airflow role-based access control (RBAC) roles.
Amazon MWAA includes environment and Airflow role information in the login token based on the permissions of the principal calling the CreateWebLoginToken API. This means that the Lambda function needs to dynamically assume a role on behalf of the user accessing the

environment, and call the API with its temporary credentials. To keep things simple, you will work with three of the default Airflow Roles: `Admin`, `User`, and `Viewer`, to access the environment you deployed earlier.

Authentication/authorization Lambda function

The authentication/authorization (authX) Lambda function must be triggered when the users try to authenticate into the Airflow UI, or log out from it. This flow will be orchestrated by the ALB using listener rules.

Execution Role

A Lambda function requires an execution role in order to access AWS resources. To generate the web login token on behalf of the user, the Lambda function must dynamically assume one of the roles described in the chapter IAM roles to access the Amazon MWAA environment. To do this, you don't need to grant explicit permission, since it is part of the **Trust Relationship**. However, you need to configure permissions to allow the function to run on a VPC (so it can call the Amazon MWAA endpoint) and to use Amazon CloudWatch Logs (in case you want to do some debugging or get some operational insights).
For that, use the AWSLambdaVPCAccessExecutionRole AWS managed policy and a customer managed policy with the basic Lambda execution permissions with the following content:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "logs:CreateLogGroup",
            "Resource": "arn:aws:logs:your-region:your-
account-id:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": [
                "arn:aws:logs:your-region:your-account-
id:log-group:/aws/lambda/mwaa_authx:*"
            ]
        }
    ]
}
```

1. Copy the previous block of code, paste it to a text editor, substitute the fields highlighted in red text colour, and save it into a file called `lambda-basic-execution-policy.json`
2. Create the IAM customer managed policy with the following command, and note down the policy ARN in the response.

```
aws iam create-policy \
    --path '/service-role/' \
    --policy-name mwaa-authx-lambda-basic-execution-policy \
    --policy-document file://lambda-basic-execution-
policy.json \
    --description "Basic execution policy for the Amazon MWAA
AuthX Lambda function"
```

3. Create the Lambda function execution role and attach the IAM policies with the following commands:

```
aws iam create-role \
    --role-name mwaa-authx-lambda-role \
    --path '/service-role/' \
    --assume-role-policy-document '{"Version": "2012-10-17",
"Statement": [{"Effect": "Allow", "Principal": {"Service":
"lambda.amazonaws.com"}, "Action": "sts:AssumeRole"' \
    --description "Execution role for the Amazon MWAA authX
Lambda function"
```

```
aws iam attach-role-policy \
    --role-name mwaa-authx-lambda-role \
    --policy-arn arn:aws:iam::your-account-id:policy/service-
role/mwaa-authx-lambda-basic-execution-policy
```

```
aws iam attach-role-policy \
    --role-name mwaa-authx-lambda-role \
    --policy-arn arn:aws:iam::aws:policy/service-
role/AWSLambdaVPCAccessExecutionRole
```

IAM roles to access the Amazon MWAA environment

To access the Amazon MWAA environment with the three Airflow roles (`Admin`, `User`, and `Viewer`), you need, respectively, three IAM roles. These roles need an IAM policy that allows access to the CreateWebLoginToken API. Within this policy, you can limit the Amazon MWAA environments and Airflow roles that the user can access.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "airflow:CreateWebLoginToken",
            "Resource": [
                "arn:aws:airflow:your-region:your-account-
id:role/your-environment-name/airflow-role"
            ]
        }
    ]
}
```

1. Create three policies (one for each role) using the following commands and write down the ARNs of each policy.

```
aws iam create-policy \
    --policy-name airflow-admin-web-login-token-policy \
    --description "Policy to allow creating a web login token
for Airflow admins" \
    --policy-document \
    '{
        "Version": "2012-10-17",
        "Statement": [
            {
                "Effect": "Allow",
                "Action": "airflow:CreateWebLoginToken",
                "Resource": [
                    "arn:aws:airflow:your-region:your-
account-id:role/your-environment-name/Admin"
                ]
            }
        ]
    }'
```

```
aws iam create-policy \
    --policy-name airflow-user-web-login-token-policy \
    --description "Policy to allow creating a web login token
for Airflow users" \
    --policy-document \
    '{
        "Version": "2012-10-17",
        "Statement": [
            {
                "Effect": "Allow",
                "Action": "airflow:CreateWebLoginToken",
                "Resource": [
                    "arn:aws:airflow:your-region:your-
account-id:role/your-environment-name/User"
                ]
            }
        ]
    }'
```

```
 aws iam create-policy \
    --policy-name airflow-viewer-web-login-token-policy \
    --description "Policy to allow creating a web login token
for Airflow users" \
    --policy-document \
     '{
         "Version": "2012-10-17",
         "Statement": [
             {
                 "Effect": "Allow",
                 "Action": "airflow:CreateWebLoginToken",
                 "Resource": [
                     "arn:aws:airflow:your-region:your-
account-id:role/your-environment-name/Viewer"
                 ]
             }
         ]
     }'
```

Next, you are going to create three IAM roles, where each will contain one of the previous policies. This way you will have one IAM role per Airflow Role, aptly named as: *airflow-admin-role*, *airflow-user-role*, and *airflow-viewer-role*.
The authX Lambda function that will request the web login token will dynamically assume one of these roles on behalf of the user accessing the environment.
The Trust Relationship of these roles only has to include the IAM role used by the authentication Lambda function. Therefore, the Trust Relationship for all three IAM roles looks like this:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::your-account-id:role/service-
role/mwaa-authx-lambda-role"
      },
      "Action": ["sts:AssumeRole","sts:SetSourceIdentity"]
    }
  ]
}
```

2. Substitute the account id and save the previous code as a file in the same folder from which you are running the AWS CLI, and name it `trust-relationship.json`.

3. Create the roles and attach the policies you created earlier. Use the following AWS CLI commands substituting the necessary fields.

```
aws iam create-role \
    --role-name airflow-admin-role \
    --assume-role-policy-document file://trust-
relationship.json \
    --description "Administrator role for Airflow"
```

```
aws iam attach-role-policy \
    --role-name airflow-admin-role \
    —policy-arn arn:aws:iam::your-account-id:policy/airflow-
admin-web-login-token-policy
```

```
aws iam create-role \
    --role-name airflow-user-role \
    --assume-role-policy-document file://trust-
relationship.json \
    --description "User role for Airflow"
```

```
aws iam attach-role-policy \
    --role-name airflow-user-role \
    —policy-arn arn:aws:iam::your-account-id:policy/airflow-
user-web-login-token-policy
```

```
aws iam create-role \
    --role-name airflow-viewer-role \
    --assume-role-policy-document file://trust-
relationship.json \
    --description "Viewer role for Airflow"
```

```
aws iam attach-role-policy \
    --role-name airflow-viewer-role \
    --policy-arn arn:aws:iam::your-account-id:policy/airflow-
viewer-web-login-token-policy
```

Function source code

Preliminary considerations

- This guide uses a single Lambda function for two different ALB listener rules (logging in and out). The corresponding action is determined within the function by evaluating the URL path. This path is contained in the triggering event that the function receives.
- The Lambda function source code provided in this guide is written in Python 3.9. Make sure you have an appropriate version installed locally.
- Multi value headers: the Lambda function needs to deal with more than one cookie, hence the ALB target group configured for the function needs to have the multi value headers setting enabled. This affects the code used to work with headers in two ways:
    - The field containing the headers included in the event object is named `'multiValueHeaders'`.
    - Headers and query parameters exchanged between the load balancer and the Lambda function use arrays instead of strings.

The source code for the Lambda function is divided in three main Python functions.

Logging users in

Upon an unauthenticated request to the web server, Amazon MWAA redirects the browser to the `/aws_mwaa/aws-console-sso` path. The ALB will use this path in a listener rule to trigger the Lambda function.

For logging users in, the function builds a URL that includes the Airflow web login token (see the Creating an Apache Airflow web login token documentation), and then redirects the user to it. This token is generated by the airflow:CreateWebLoginToken API, which needs to be called using temporary credentials for the IAM role, determined using the output from the function described in the next section.

If the API call succeeds, the function redirects the browser to a URL in the format: `https://HOST/aws_mwaa/aws-console-sso?login=true#<token>`. The `login=true` query parameter in the URL can be used in an ALB listener rule to avoid redirecting again to the Lambda function.
If the API call fails or there are no appropriate claims in the token, the function redirects the user to an error page and terminates the Cognito session.

Mapping federated users to IAM roles

The log-in function needs to have a way of determining if and what IAM role to assume upon an incoming request. In order to do this, this function uses the claims issued by the Identity Provider (IdP). These claims are typically user attributes or groups that a user belongs to.
The function uses the two custom attributes you already defined in the Cognito user pool:

- `custom:idp-groups` – which is mapped to the security groups the user belongs to in Azure AD.
- `custom:idp-name` – which is mapped to the `userprincipalname` attribute.

The function extracts those attributes from the JWT provided by Cognito. After decoding the token payload, the function uses the `custom:idp-groups` attribute in it to determine if the user can access the Amazon MWAA environment and on which Airflow role. For instance, if `custom:idp-groups` contain the group `Amazon MWAA-Test-Admins`, the user should be able to access the Amazon MWAA environment as an Airflow Admin. The `custom:idp-name` attribute is used for logging purposes.
You need to encode such a map using a JSON object like the following one. You will pass this object as an environment variable to the Lambda function, substituting the group ids for the ones in your IdP.

```
[{"idp-group": "1db1943c-xxxx-xxxx-xxxx-4b8d1c774370", "iam-
role": "airflow-admin-role"},
{"idp-group": "58931a95-xxxx-xxxx-xxxx-6f12c8f53233", "iam-
role": "airflow-user-role"},
{"idp-group": "b7282a94-xxxx-xxxx-xxxx-3a0a0673f92f", "iam-
role": "airflow-viewer-role"}]
```

Logging users out

When a user clicks the **Logout** button in the Airflow UI, the browser issues a request on the `/logout/` path, which can be used by an ALB listener rule to trigger the logout process in the Lambda function.
For logging users out, the function expires the ALB authentication and Airflow session cookies. Then it calls the logout URL in the Amazon MWAA web application using its private endpoint, and redirects to the Cognito logout URL, which in turn redirects the user to a closing page.

Bringing it all together

1. Create a directory for the project called, for example, mwaa_authx, and navigate to it.

```
mkdir mwaa_authx && cd mwaa_authx
```

2. Below you can find the source code for the Lambda function. Copy it into an editor and save it as `mwaa_authx_lambda_function.py` in said directory.

```
import os
import json
import base64
import logging
import requests
import jwt
import botocore
import boto3
from urllib.parse import quote

PRIVATE_ENDPOINT = os.environ.get('PRIVATE_ENDPOINT',
'').strip()
Amazon MWAA_ENV_NAME = os.environ.get('Amazon MWAA_ENV_NAME',
'').strip()
AWS_ACCOUNT_ID = os.environ.get('AWS_ACCOUNT_ID', '').strip()
COGNITO_CLIENT_ID = os.environ.get('COGNITO_CLIENT_ID',
'').strip()
COGNITO_DOMAIN = os.environ.get('COGNITO_DOMAIN').strip()
AWS_REGION = os.environ.get('AWS_REGION')
IDP_LOGIN_URI = os.environ.get('IDP_LOGIN_URI').strip()
GROUP_TO_ROLE_MAP =
json.loads(os.environ.get('GROUP_TO_ROLE_MAP', '{}'))
ALB_COOKIE_NAME = os.environ.get('ALB_COOKIE_NAME',
'AWSELBAuthSessionCookie').strip()
LOGOUT_REDIRECT_DELAY = 10 # seconds

sts = boto3.client('sts')
logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    """
        Lambda handler
    """
    logger.info(json.dumps(event))

    path = event['path']
    headers = event['multiValueHeaders']

    if 'x-amzn-oidc-data' in headers:
        encoded_jwt = headers['x-amzn-oidc-data'][0]
        token_payload = decode_jwt(encoded_jwt)
    else:
        # There is no session, close
        return close(headers)

    if path == '/aws_mwaa/aws-console-sso':
        redirect = login(headers, token_payload)
    elif path == '/logout/':
        redirect = logout(headers, 'Logged out successfully')
    else:
        redirect = logout(headers, '')

    logger.info(json.dumps(redirect))

    return redirect

def multivalue_to_singlevalue(headers):
    """
        Convert multi-value headers to single value
    """
    svheaders = {key: value[0] for (key, value) in
headers.items()}
    return svheaders

def singlevalue_to_multivalue(headers):
    """
        Convert single value headers to multi-value headers
    """
    mvheaders = {key: [value] for (key, value) in
headers.items()}
    return mvheaders

def login(headers, jwt_payload):
    """
        Function that returns a redirection to an appropriate
URL that includes a web login token.
    """
    # Role to be determined using claims in JWT token
    role_arn = get_iam_role_arn(jwt_payload)
    user_name = jwt_payload.get('custom:idp-name', role_arn)
    host = headers['host'][0]

    if role_arn:
        mwaa = get_mwaa_client(role_arn, user_name)
        if mwaa:
            # Obtain web login token for the configured 
environment
            try:
                mwaa_web_token = 
mwaa.create_web_login_token(Name=Amazon MWAA_ENV_NAME)[
                    "WebToken"]

                logger.info('Redirecting with Amazon MWAA WEB 
TOKEN')

                redirect = {
                    'statusCode': 302,
                    'statusDescription': '302 Found',
                    'multiValueHeaders': {
                        'Location': 
[f'https://{host}/aws_mwaa/aws-console-
sso?token=true#{mwaa_web_token}']
                    }
                }
            except botocore.exceptions.ClientError as error:
                if error.response['Error']['Code'] == 
'AccessDeniedException':
                    redirect = logout(headers,
                        f'The role "{role_arn}" assigned to 
{user_name} does not have access to the environment "{Amazon 
MWAA_ENV_NAME}".')
                elif error.response['Error']['Code'] == 
'ResourceNotFoundException':
                    redirect = logout(headers, f'Environment 
{Amazon MWAA_ENV_NAME} was not found.')
                else:
                    redirect = logout(headers, error)
        else:
            redirect = logout(headers, 'There was an error 
while logging in, please contact your administrator.')
    else:
        redirect = logout(headers, 'There is no valid role 
associated with your user.')

    return redirect

def logout(headers, message):
    """
        Logs out from Airflow and expires the ALB cookies.
        If a message is present, it displays it for a few
```

```
seconds and redirects to Cognito logout.
    """
    logger.info('LOGGING OUT')

    host = headers['host'][0]

    # Convert multi-value headers to single value to forward
the contents to Airflow
    svheaders = multivalue_to_singlevalue(headers)
    svheaders['host'] = PRIVATE_ENDPOINT
    logger.info(f'CALLING {PRIVATE_ENDPOINT}')
    # issue a request to the Amazon MWAA logout private
endpoint
    response =
requests.get(f'https://{PRIVATE_ENDPOINT}/logout/',
                                headers=svheaders,
                                allow_redirects=True)

    # Convert single value headers to multi-value headers so
the ALB processes them correctly
    headers_to_forward =
singlevalue_to_multivalue(response.headers)

    redirect_uri = quote(f'https://{host}/logout/close',
safe="")
    cognito_logout_uri = \

f'https://{COGNITO_DOMAIN}.auth.{AWS_REGION}.amazoncognito.co
m/logout?client_id=' + \

f'{COGNITO_CLIENT_ID}&response_type=code&logout_uri={redirect
_uri}&scope=openid'

    headers = headers_to_forward
    headers['Location'] = [cognito_logout_uri]
    expire_alb_cookies(headers)

    if message:
        body = error_redirection_body(message,
cognito_logout_uri)
        headers['Content-Type'] = ['text/html']
        redirect = {
            'statusCode': 200,
            'multiValueHeaders': headers,
            'body': body,
            'isBase64Encoded': False
        }
    else:
        redirect = {
            'statusCode': 302,
            'statusDescription': '302 Found',
            'multiValueHeaders': headers
        }

    return redirect

def get_mwaa_client(role_arn, user_name):
    """
        Returns an Amazon MWAA client under the given IAM 
role
    """
    mwaa = None

    try:
        logger.info(f'Assuming role "{role_arn}" with source 
identity "{user_name}"...')
        credentials = sts.assume_role(
            RoleArn=role_arn,
            RoleSessionName=user_name,
            DurationSeconds=900, # This is the minimum 
allowed
            SourceIdentity=user_name
        )['Credentials']

        access_key = credentials['AccessKeyId']
        secret_key = credentials['SecretAccessKey']
        session_token = credentials['SessionToken']

        # create service client using the assumed role 
credentials, e.g. S3
        mwaa = boto3.client(
            'mwaa',
            aws_access_key_id=access_key,
            aws_secret_access_key=secret_key,
            aws_session_token=session_token
        )
    except botocore.exceptions.ClientError as error:
        logger.error(f'Error while assuming role {role_arn}. 
{error}')
    except Exception as error:
        logger.error(f'Unknown error while assuming role 
{role_arn}. {error}')
```

```
    return mwaa

def get_iam_role_arn(jwt_payload):
    """
        Returns the name of an IAM role based on the
'custom:idp-groups' contained in the JWT token
    """
    # This list contains the mappings between IdP groups and
their corresponding IAM role.
    # The list is sorted by precedence, so, if a user belongs
to more than one group, it's given
    # mapped to a role that contains more permissions
    role_arn = ''

    logger.info(f'JWT payload: {jwt_payload}')
    if 'custom:idp-groups' in jwt_payload:
        user_groups = parse_groups(jwt_payload['custom:idp-
groups'])
        for mapping in GROUP_TO_ROLE_MAP:
            if mapping['idp-group'] in user_groups:
                role_name = mapping['iam-role']
                role_arn =
f'arn:aws:iam::{AWS_ACCOUNT_ID}:role/{role_name}'
                break
    return role_arn

def parse_groups(groups):
    """
        Converts the groups SAML claim content to a list of
strings
    """
    # The groups SAML claim comes in a string
    # When there is more than one group id, the string starts
and ends with square brackets
    # There might also be spaces between the group ids
    groups = groups.replace('[', '').replace(']',
'').replace(' ', '')
    return groups.split(',')

def decode_jwt(encoded_jwt):
    """
        Decodes a JSON Web Token issued by the ALB after
successful authentication
        against an OIDC IdP (e.g.: Cognito).
```

```
https://docs.aws.amazon.com/elasticloadbalancing/latest/appli
cation/listener-authenticate-users.html
    """
    # Step 1: Get the key id from JWT headers (the kid field)
    jwt_headers = encoded_jwt.split('.')[0]
    decoded_jwt_headers = base64.b64decode(jwt_headers)
    decoded_jwt_headers = decoded_jwt_headers.decode("utf-8")
    decoded_json = json.loads(decoded_jwt_headers)
    kid = decoded_json['kid']

    # Step 2: Get the public key from regional endpoint
    url = f'https://public-
keys.auth.elb.{AWS_REGION}.amazonaws.com/{kid}'
    req = requests.get(url)
    pub_key = req.text

    # Step 3: Get the payload
    payload = jwt.decode(encoded_jwt, pub_key,
                         algorithms=[decoded_json['alg']])

    return payload

def expire_alb_cookies(headers):
    """
        Sets ALB session cookies to expire
    """
    alb_cookies = [f'{ALB_COOKIE_NAME}-1=del;Max-Age=-
1;Path=/;',
                   f'{ALB_COOKIE_NAME}-0=del;Max-Age=-
1;Path=/;']

    if 'Set-Cookie' in headers:
        headers['Set-Cookie'] += alb_cookies
    else:
        headers['Set-Cookie'] = alb_cookies

def error_redirection_body(message, logout_uri):
    """
        Returns an HTML string that displays an error message
and redirects the browser to the logout_uri
    """
    body = f'<html><body><h3>{message}</h3><br><br>Closing
session in ' + \
           f'<span
id="countdown">{LOGOUT_REDIRECT_DELAY}</span> seconds' + \
            '</body></html><script type="text/javascript">' +
\
           f'var seconds = {LOGOUT_REDIRECT_DELAY};' + \
            'function countdown() {' + \
            '    seconds -= 1;' + \
            '    if (seconds < 0) {' + \
           f'         window.location = "{logout_uri}?";' + \
            '    } else {' + \
            '
document.getElementById("countdown").innerHTML = seconds;' +
\
            '        window.setTimeout("countdown()", 1000);'
+ \
            '    }' + \
            '}' + \
            'countdown();' + \
            '</script>'
    return body

def close(headers):
    """
        Requests user to close the current tab
    """
    body = '<html><body><h3>You can now close this
tab.</h3></body></html>'
    headers['Content-Type'] = ['text/html']
    return {
        'statusCode': 200,
        'multiValueHeaders': headers,
        'body': body,
        'isBase64Encoded': False
    }
```

Deploying the Lambda function

Dependencies

The Lambda function depends on a few external Python libraries.

1. Create a new file called `requirements.txt` in the same directory where you stored the function source code, and copy the following contents into it.

```
pyjwt
requests
urllib3
```

2. Install the dependencies into a new package directory.

```
pip install -t ./package -r requirements.txt
```

3. Download and unzip the following wheel files into the package directory using these commands.

```
wget
https://files.pythonhosted.org/packages/be/2a/6d266eea47dbb2d
872bbd1b8954a2d167668481ff34ebb70ffdd1113eeab/cffi-1.14.6-
cp39-cp39-manylinux1_x86_64.whl
unzip cffi-1.14.6-cp39-cp39-manylinux1_x86_64.whl -d
./package
rm cffi-1.14.6-cp39-cp39-manylinux1_x86_64.whl
```

```
wget
https://files.pythonhosted.org/packages/07/fa/f63509370561201
ffa852e4f3fb105c76ced6927f951e4cc6a3973d1a527/cryptography-
35.0.0-cp36-abi3-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl
unzip cryptography-35.0.0-cp36-abi3-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl -d ./package
rm cryptography-35.0.0-cp36-abi3-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl
```

The Lambda function needs these wheel files because the libraries must be compiled for the native operating system. If you are using Linux on x86 as your development machine, you can include `cryptography` and `cffi` in the `requirements.txt` file

Deployment package

1. After the needed libraries are installed in the package directory, create a deployment package with them, making sure they are located at the root of the .zip file. This command generates a file named `mwaa-auth-package.zip` in your project directory.

```
cd package && zip -r ../mwaa-authx-package.zip .
```

2. Add the lambda_function.py file to the root of the zip file.

```
cd .. && zip -g mwaa-authx-package.zip
mwaa_authx_lambda_function.py
```

3. The zip file you just created is the deployment package for the Lambda function. In order to continue, upload it to an S3 bucket (for example, the same one you created at the beginning of this guide).

```
aws s3 cp ./mwaa-authx-package.zip s3://your-bucket-
name/lambda/
```

Environment variables

The code provided in this guide uses environment variables. Environment variables allow you to configure function parameters without changing the function code.

1. Create a JSON file called `env.json` and save it in your project directory; it contains the environment variables that the Lambda function needs, which include the group to role mapping described earlier.

```
{
    "Variables": {
        "ALB_COOKIE_NAME": "AWSELBAuthSessionCookie",
        "AWS_ACCOUNT_ID": "your-account-id",
        "COGNITO_CLIENT_ID":"your-cognito-client-app-id",
        "COGNITO_DOMAIN": "your-cognito-domain-prefix ",
        "GROUP_TO_ROLE_MAP":
            "[{\"idp-group\":\"your-idp-airflow-admins-group-
id\",\"iam-role\":\"airflow-admins-role\"},{\"idp-
group\":\"your-idp-airflow-users-group-id\",\"iam-
role\":\"airflow-users-role\"},{\"idp-group\":\"your-idp-
airflow-viewers-group-id\",\"iam-role\": \"airflow-viewers-
role\"}]",
        "IDP_LOGIN_URI":"your-idp-login-url",
        "MWAA_ENV_NAME":"your-mwaa-environment-name",
        "PRIVATE_ENDPOINT":"your-mwaa-private-endpoint-
domain-name"
    }
}
```

Creating the Lambda function

1. From your project directory, run the following AWS CLI command and write down the function ARN, as you need that to register it as part of an ALB target group.

```
aws lambda create-function --region your-region \
    --function-name mwaa_authx \
    --description "Function to authenticate and authorize
users into an Amazon MWAA environment" \
    --role arn:aws:iam::your-account-id:role/service-
role/mwaa-authx-lambda-role \
    --runtime python3.9 \
    --handler mwaa_authx_lambda_function.lambda_handler \
    --code S3Bucket=your-bucket-name,S3Key=lambda/mwaa-authx-
package.zip \
    --timeout 10 \
    --vpc-config SubnetIds=your-private-subnet-1-id,your-
private-subnet-2-id,SecurityGroupIds=your-alb-security-group-
id \
    --package-type Zip \
    --environment file://env.json
```

Finishing the ALB configuration

Target group for the Lambda function

The ALB needs a new target group that points to the authX Lambda function. This target group needs to have the multiValueHeaders option enabled.

1. Use the following AWS CLI commands to create the target group and configure it accordingly.

```
aws elbv2 create-target-group --region your-region \
    --name authx-lambda-tg \
    --target-type lambda
```

```
aws elbv2 modify-target-group-attributes --region your-region
\
    --target-group-arn your-target-group-arn \
    --attributes lambda.multi_value_headers.enabled
```

2. Add permissions for the ALB to trigger the Lambda function and register the function with the target group running the following AWS CLI commands:

```
aws lambda add-permission --region your-region \
    --function-name mwaa_authx \
    --statement-id load-balancer \
    --principal elasticloadbalancing.amazonaws.com \
    --action lambda:InvokeFunction \
    --source-arn your-target-group-arn
```

```
aws elbv2 register-targets --region your-region \
    --target-group-arn your-target-group-arn \
    --targets your-lambda-function-arn
```

Note: the target registration might fail because it will take a while until the Lambda function is fully deployed.

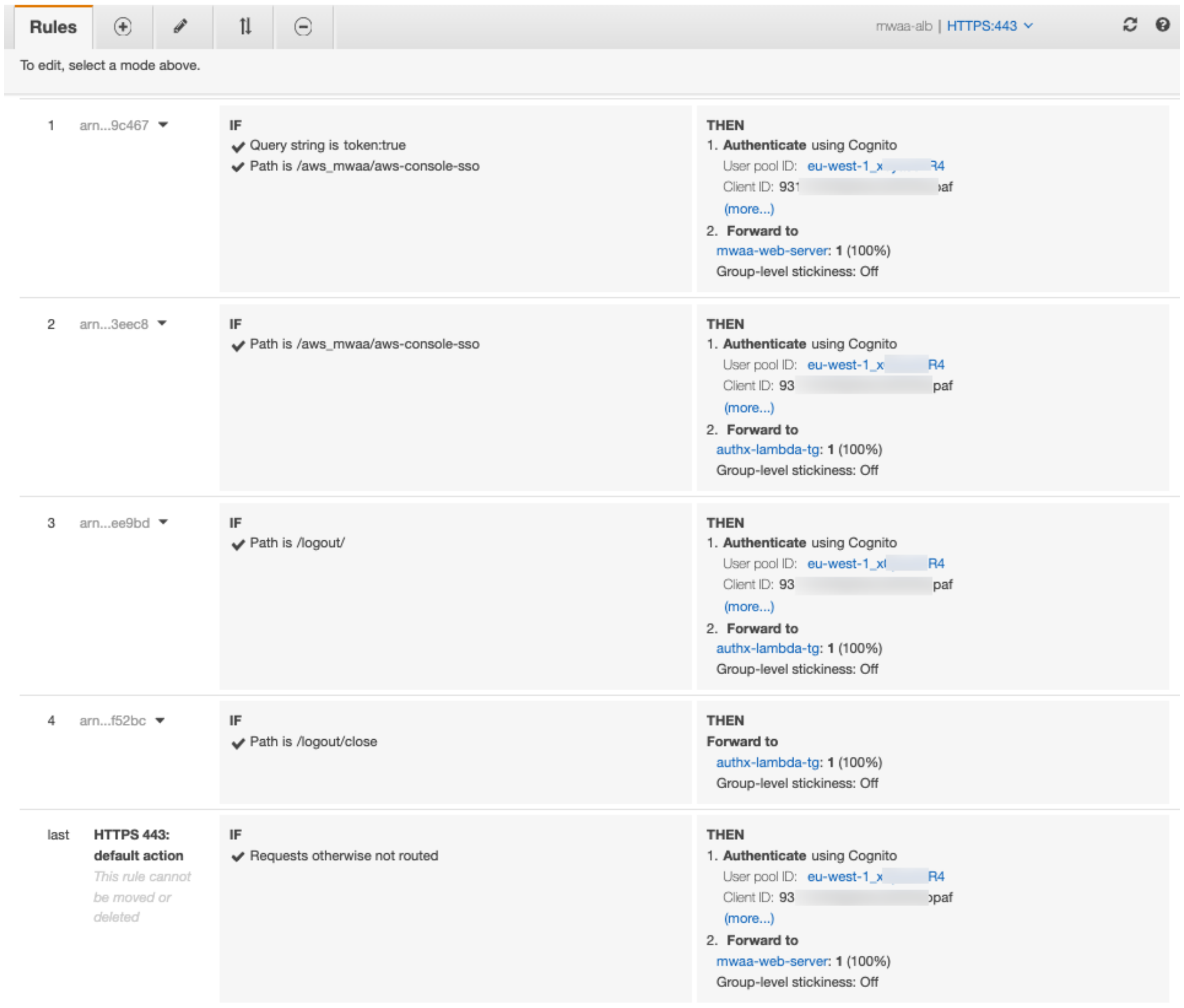
ALB Rules

Aside from the default action rule you created when deploying the ALB, you need four additional ones to handle the authentication and logging out of the Airflow web UI.
The typical flow works as follows:

1. The user introduces the ALB URL in the browser, which matches the last rule in the listener. This rule forces Cognito authentication, so the browser takes the user to the IdP login page. After authenticated, the request is forwarded to Amazon MWAA, which then redirects to the `https://`*`alb-domain`*`/aws_mwaa/aws-console-sso` URL.
2. The request to `https://`*`alb-domain`*`/aws_mwaa/aws-console-sso` reaches the ALB and matches the rule number `2`. The ALB verifies the Cognito token and forwards the request to the authX Lambda function.
3. The Lambda function verifies the token contents and, if the federated identity has the permissions, generates a Amazon MWAA web login token and redirects to `https://`*`alb-domain`* `/aws_mwaa/aws-console-sso?token:true#<web-login-token>`.
4. The request to `https://`*`alb-domain`*`/aws_mwaa/aws-console-sso?token:true` triggers rule number `1`, which also verifies the Cognito token and then forwards to Amazon MWAA. Amazon MWAA starts a session with the web login token included in the request and the user can interact with the Airflow UI.
5. Any further requests to Amazon MWAA (except logging out) will match the last rule and will be forwarded to Amazon MWAA until the Cognito token expires.

6. When the user logs out from Airflow, the browser sends a request to `https://`*`alb-domain`*`/logout/`. This triggers rule number `3`, that verifies the Cognito token and forwards the request to the Lambda function. The function invalidates the ALB session cookies and redirects to the Cognito logout URL including a parameter called `logout_uri` with value `https://`*`alb-domain`*`/logout/close`. This parameter is used by Cognito to redirect the browser to an appropriate URI and must be included in the Logout URL configuration of the app client.

7. Cognito closes the session and redirects to `https://`*`alb-domain`*`/logout/close`. This request triggers rule number `4`. This rule sends a request without an authentication token to the Lambda function. The function returns a simple HTML web page indicating the user to close the browser window.



		IF	THEN
1	arn...9c467	✓ Query string is token:true ✓ Path is /aws_mwaa/aws-console-sso	1. **Authenticate** using Cognito User pool ID: eu-west-1_x R4 Client ID: 931 af (more...) 2. **Forward to** mwaa-web-server: **1** (100%) Group-level stickiness: Off
2	arn...3eec8	✓ Path is /aws_mwaa/aws-console-sso	1. **Authenticate** using Cognito User pool ID: eu-west-1_x R4 Client ID: 93 paf (more...) 2. **Forward to** authx-lambda-tg: **1** (100%) Group-level stickiness: Off
3	arn...ee9bd	✓ Path is /logout/	1. **Authenticate** using Cognito User pool ID: eu-west-1_xl R4 Client ID: 93 paf (more...) 2. **Forward to** authx-lambda-tg: **1** (100%) Group-level stickiness: Off
4	arn...f52bc	✓ Path is /logout/close	**Forward to** authx-lambda-tg: **1** (100%) Group-level stickiness: Off
last	**HTTPS 443: default action** *This rule cannot be moved or deleted*	✓ Requests otherwise not routed	1. **Authenticate** using Cognito User pool ID: eu-west-1_x R4 Client ID: 93 paf (more...) 2. **Forward to** mwaa-web-server: **1** (100%) Group-level stickiness: Off

Figure 25 – All the required rules configured for the ALB listener

Security

This technical guide describes how you can securely grant access to a private Amazon MWAA environment.

The component that authorizes access to that resource is the authentication and authorization Lambda function. Therefore, you must protect that Lambda function to prevent changes that would grant access to unauthorized identities.

You can achieve this by following the Grant least privilege best practice.

Conclusion

By following this technical guide, you have deployed a serverless solution to enable your users to access a private Amazon MWAA environment using the same identity provider they use to access other resources in their organization. Although this guide uses Azure AD as the IdP, you can use this architecture to integrate other IdPs, such as Okta.

Following the architecture described in this guide, you can also consider building similar solutions for other custom or off-the-shelf applications running on AWS.

Contributors

Contributors to this document include:

- Tasio Guevara, Senior Solutions Architect, Amazon Web Services.

Document Revisions

Date	Description
February 2022	First publication