

Building well-architected serverless applications: Controlling serverless API access – part 3

by Julian Wood | on 31 JUL 2020 | in [Amazon API Gateway](#), [Amazon Cognito](#), [AWS Amplify](#), [AWS AppSync](#), [AWS CloudFormation](#), [AWS Identity and Access Management \(IAM\)](#), [AWS Lambda](#), [AWS Well-Architected Tool](#), [Best Practices](#), [Serverless](#) | [Permalink](#) | [Share](#)

This series of blog posts uses the [AWS Well-Architected Tool](#) with the [Serverless Lens](#) to help customers build and operate applications using best practices. In each post, I address the nine serverless-specific questions identified by the Serverless Lens along with the recommended best practices. See the [Introduction post](#) for a table of contents and explanation of the example application.

Security question SEC1: How do you control access to your serverless API?

This post continues [part 2](#) of this security question. Previously, I cover Amazon Cognito user and identity pools, JSON web tokens (JWT), API keys and usage plans.

Best practice: Scope access based on identity's metadata

Authenticated users should be separated into logical groups, roles, or tiers. Separation can also be based on custom authentication token attributes included within Security Assertion Markup Language (SAML) or JSON Web Tokens (JWT). Consider using the user's identity metadata to enable fine-grain control access to resources and actions.

Scoping access based on authentication metadata allows you to provide limited and fine-grained capabilities and access to consumers based on their roles and intent.

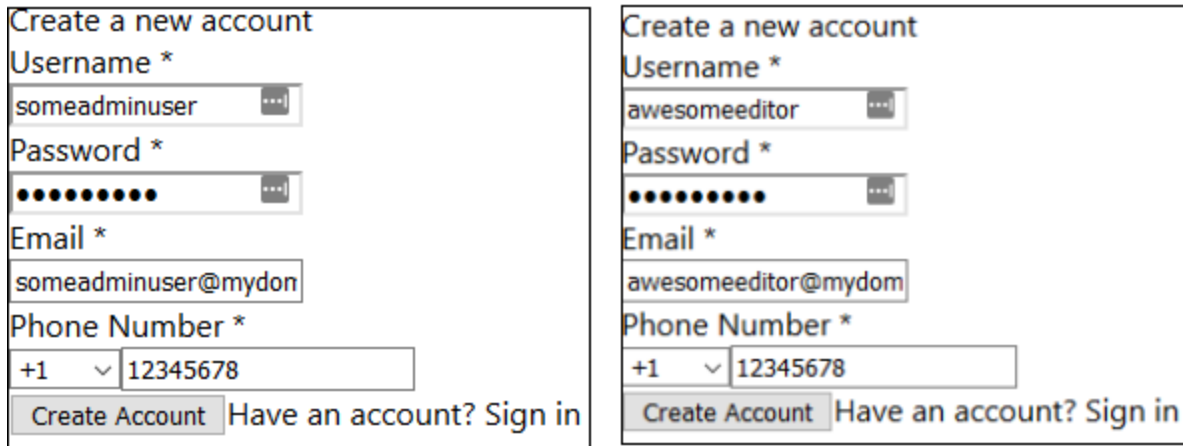
Review levels of access, identity metadata, and separate consumers into logical groups/tiers

With JWT or SAML, ensure you have the right level of information available within the token claims to help you develop authorization logic. Use custom private claims along with a unique namespace for non-public information. Private claims are to share custom information specifically with your application client. Unique namespaces are to avoid name collision for custom claims. For more information, see the [AWS Partner Network](#) blog post "[Understanding JWT Public, Private and Reserved Claims](#)".

With [Amazon Cognito](#), you can use custom attributes or the [Pre Token Generation Lambda Trigger](#) feature. This [AWS Lambda](#) trigger allows you to customize a JWT token claim before the token is generated.

To illustrate using Amazon Cognito groups, I use the example from [this blog post](#). The example uses Amplify CLI to create a web application for managing group membership. API Gateway handles authentication using an Amazon Cognito user pool as part of an [administrator API](#). Two Amazon Cognito user pool groups are created using `amplify auth update`, one for *admin*, and one for *editors*.

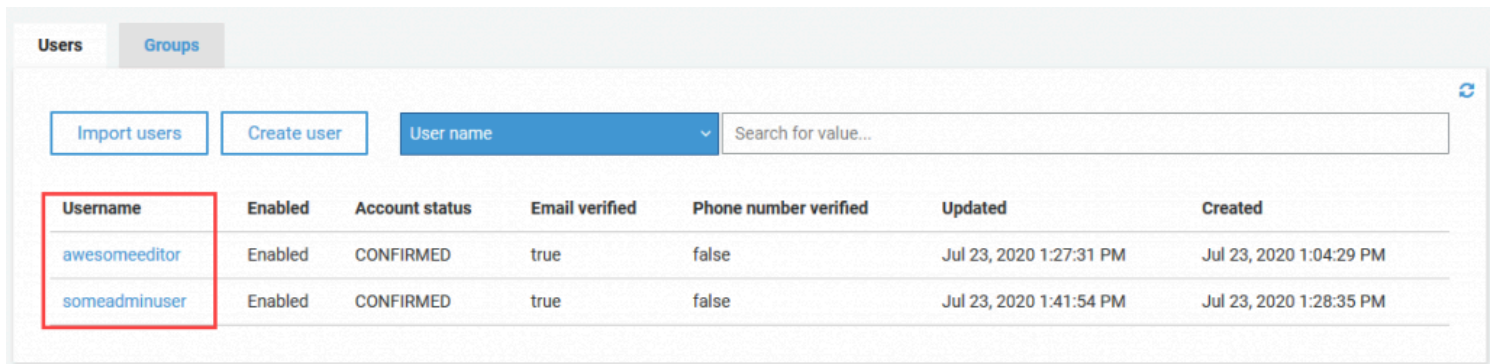
1. I navigate to the deployed web application and create two users, an administrator called *someadminuser* and an editor user called *awesomeeditor*.



Two side-by-side screenshots of the Amazon Cognito 'Create a new account' form. The left form shows the 'someadminuser' account being created with email 'someadminuser@mydon'. The right form shows the 'awesomeeditor' account being created with email 'awesomeeditor@mydom'. Both forms have fields for Username, Password, Email, and Phone Number, and buttons for 'Create Account' and 'Have an account? Sign in'.

Show Amazon Cognito user creation

1. I navigate to the Amazon Cognito user pool console, choose **Users and groups** under *General settings*, and can see that both users are created.



Username	Enabled	Account status	Email verified	Phone number verified	Updated	Created
awesomeeditor	Enabled	CONFIRMED	true	false	Jul 23, 2020 1:27:31 PM	Jul 23, 2020 1:04:29 PM
someadminuser	Enabled	CONFIRMED	true	false	Jul 23, 2020 1:41:54 PM	Jul 23, 2020 1:28:35 PM

View Amazon Cognito users created

1. I choose the **Groups** tab and see that there are two user pool groups set up as part of `amplify auth update`.
2. I add the *someadminuser* to the admin group.

The screenshot shows the AWS IAM console interface for the 'admin' group. On the left, there are tabs for 'Users' and 'Groups', with 'Groups' selected. Below the tabs is a 'Create group' button. Under 'Group Name', 'admin' is selected and highlighted with a red box, and 'editors' is listed below it. In the main content area, the title is 'Groups > admin'. There is a 'Delete group' button. The 'Description' is empty. The 'Role ARN' is 'arn:aws:iam::978558897928:role/eu-west-2_FEAH74vPw-adminGroupRole' and is highlighted with a red box. The 'Precedence' is 1. The 'Updated' timestamp is 'Jul 22, 2020 6:48:17 PM' and the 'Created' timestamp is 'Jul 22, 2020 6:48:17 PM'. At the bottom, there is an 'Add users' button and a table of users.

Username	Enabled	Account status	Email verified	Phone number verified	Updated
someadminuser	Enabled	CONFIRMED	true	false	Jul 23, 2020

View Amazon Cognito user added to group and IAM role

1. There is an [AWS Identity and Access Management \(IAM\)](#) role associated with the administrator group. This IAM role has an associated identity policy that grants permission to access an S3 bucket for some future application functionality.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3:PutObject",
        "s3:GetObject",
        "s3:ListBucket",
        "s3:DeleteObject"
      ],
      "Resource": [
        "arn:aws:s3:::mystoragebucket194021-dev/*"
      ],
      "Effect": "Allow"
    }
  ]
}
```

```

    ]
  }

```

1. I log on to the web application using both the *someadminuser* and *awesomeeditor* accounts and compare the two JWT `accessToken` Amazon Cognito has generated.

The *someadminuser* has a `cognito:groups` claim within the token showing membership of the user pool group `admin`.

PAYLOAD: "username": "someadminuser"

```

{
  "sub": "5766996d-1f39-443a-93b8-fa551005bd51",
  "cognito:groups": [
    "admin"
  ],
  "event_id": "c1f19c57-8059-49fd-84ae-35e6c907aacf",
  "token_use": "access",
  "scope": "aws.cognito.signin.user.admin",
  "auth_time": 1595514051,
  "iss": "https://cognito-idp.eu-west-2.amazonaws.com/eu-west-2_FEAH74vPw",
  "exp": 1595517651,
  "iat": 1595514051,
  "jti": "016ea567-fd28-4859-9753-afdb34f9374c",
  "client_id": "79imhmi57au44pb09oimic15jf",
  "username": "someadminuser"
}

```

PAYLOAD: "username": "awesomeeditor"

```

{
  "sub": "f2cc70f7-a58b-4d31-bc75-e762408698a2",
  "event_id": "3949ba8e-1381-4511-a1b1-bd6682c54276",
  "token_use": "access",
  "scope": "aws.cognito.signin.user.admin",
  "auth_time": 1595513591,
  "iss": "https://cognito-idp.eu-west-2.amazonaws.com/eu-west-2_FEAH74vPw",
  "exp": 1595517191,
  "iat": 1595513591,
  "jti": "82968aa4-79ab-4d1f-8174-3fef0fddfab9",
  "client_id": "79imhmi57au44pb09oimic15jf",
  "username": "awesomeeditor"
}

```

View JWT with group membership

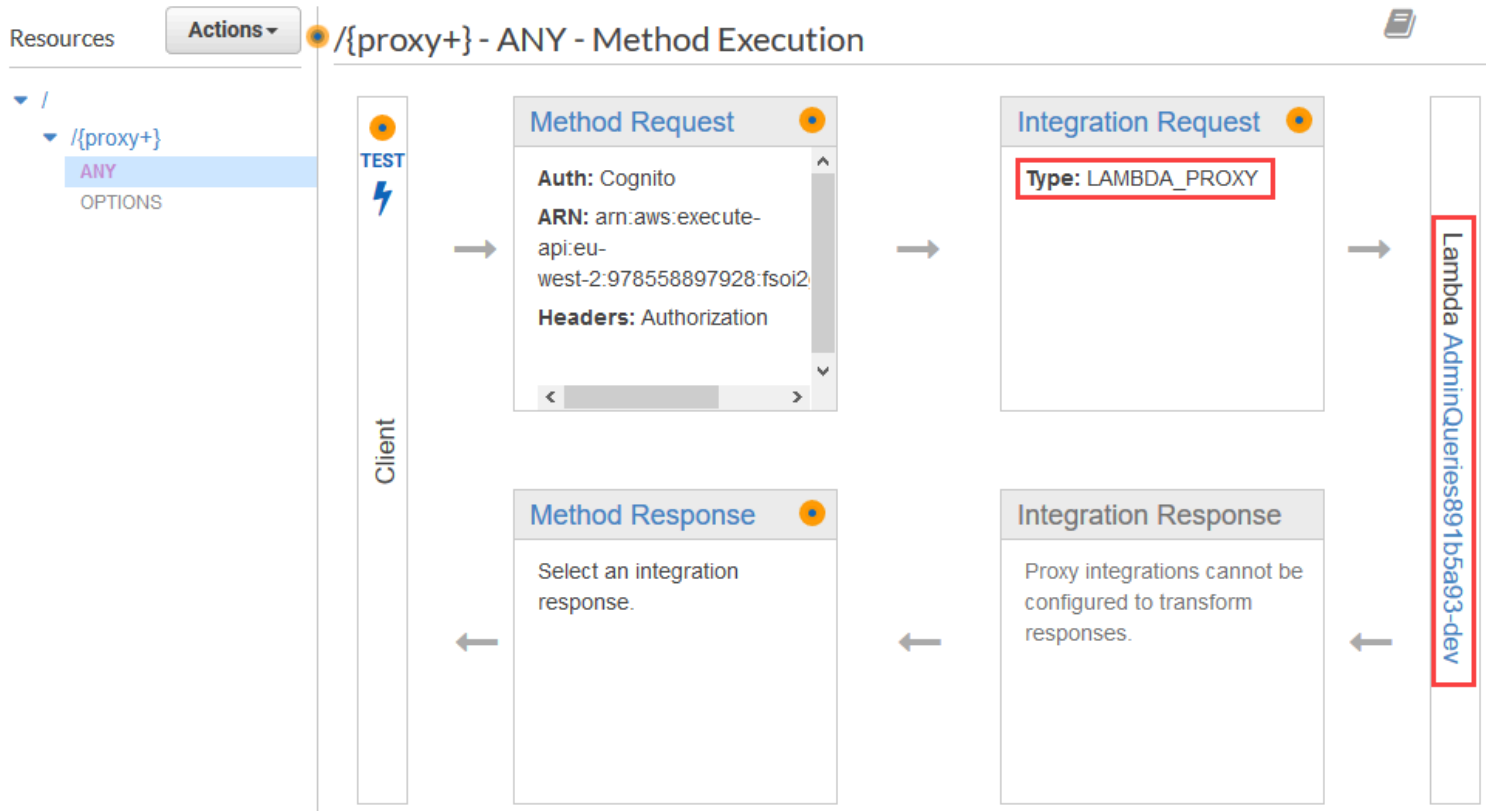
This token with its group claim can be used in a number of ways to authorize access.

Within this example frontend application, the token is used against an API Gateway resource using an Amazon Cognito authorizer.

An Amazon Cognito authorizer is an alternative to using IAM or Lambda authorizers to control access to your API Gateway method. The client first signs in to the user pool, and receives a token. The client then calls the API method with the token which is typically in the request's `Authorization` header. The API call only succeeds if a valid is supplied. Without the correct token, the client isn't authorized to make the call.

In this example, the Amazon Cognito authorizer authorizes access at the API method. Next, the event payload passed to the Lambda function contains the token. The function reads the token information. If the group membership claim includes *admin*, it adds the *awesomeeditor* user to the Amazon Cognito user pool group *editors*.

1. To see how this is configured, I navigate to the API Gateway console and select the *AdminQueries* API.
2. I view the `/ {proxy+} /ANY` resource.
3. I see that the *Integration Request* is set to `LAMBDA_PROXY` . which calls the *AdminQueries* Lambda function.



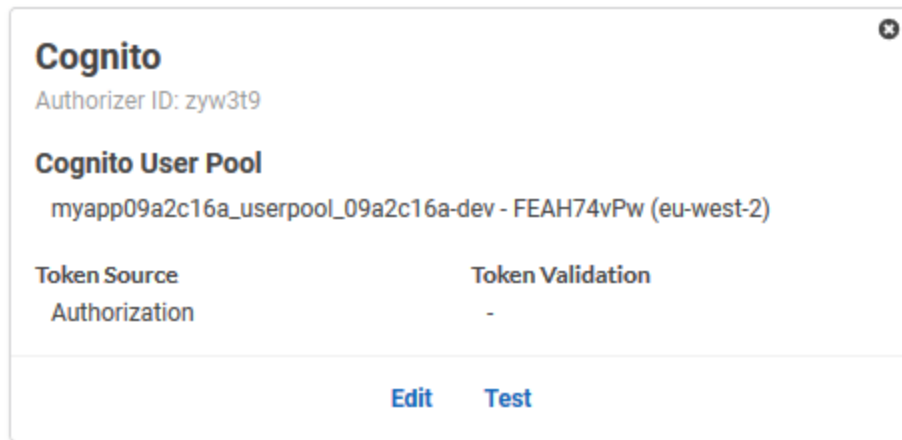
View API Gateway Lambda proxy path

1. I view the *Method Request*.



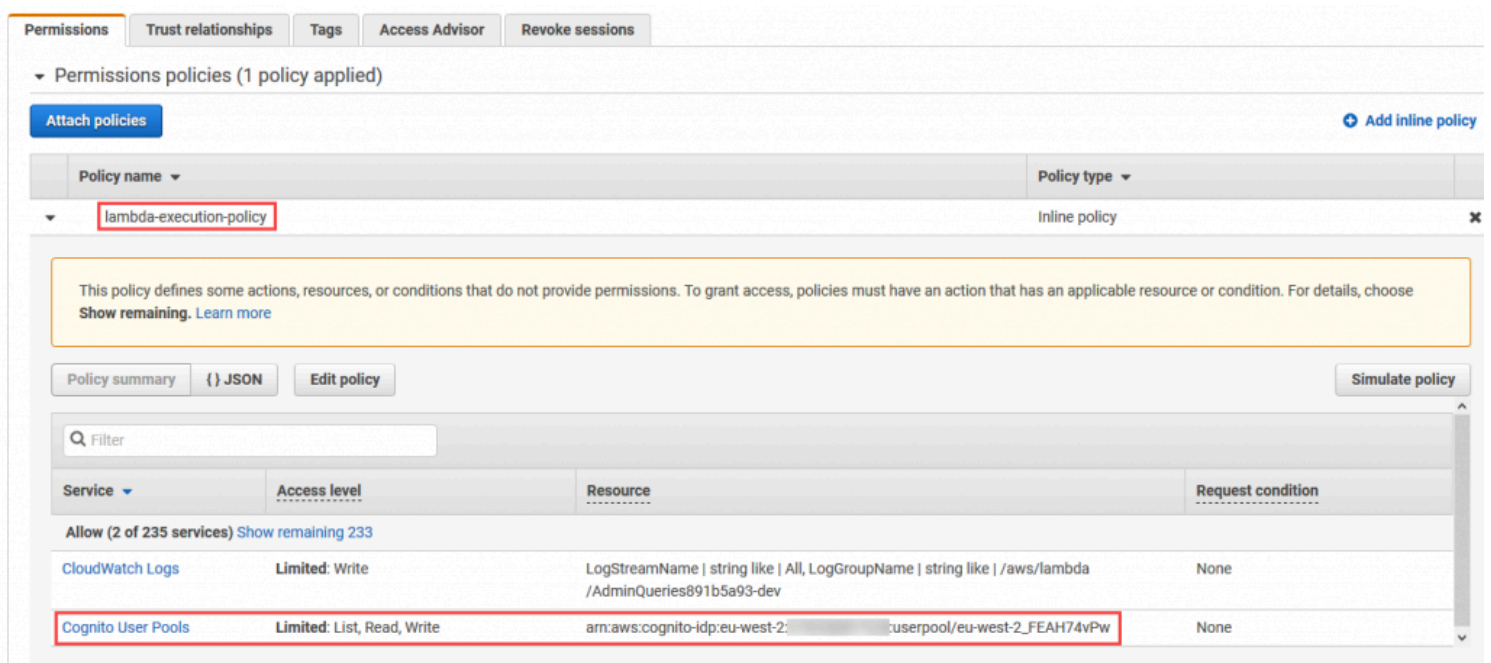
View API Gateway Method Request using Amazon Cognito authorization

1. Authorization is set to an Amazon Cognito user pool authorizer with an OAuth scope of `aws.cognito.signin.user.admin`.
2. I navigate to the *Authorizers* menu item, and can see the configured Amazon Cognito authorizer.
3. In the Amazon Cognito user pool details, the *Token Source* is set to **Authorization**. This is the name of the header sent to the Amazon Cognito user pool for authorization.



View Amazon Cognito authorizer settings

1. I navigate to the [AWS Lambda console](#), select the *AdminQueries* function which `amplify add auth` added, and choose the **Permissions** tab. I select the Execution role and view its *Permissions policies*.
2. I see that the function execution role allows write permission to the Amazon Cognito user pool resource. This allows the function to amend the user pool group membership.



View Lambda execution role permissions including Amazon Cognito write

1. I navigate back to the [AWS Lambda console](#), and view the **configuration** for the *AdminQueries* function. There is an environment variable set for GROUP=admin.

Environment variables (3)

The environment variables below are encrypted at rest with the default Lambda service key.

Key	Value
ENV	dev
GROUP	admin
USERPOOL	eu-west-2_FEAH74vPw

Lambda function environment variables

The Lambda function code checks if the `authorizer.claims` token includes the `GROUP` environment variable value of `admin`. If not, the function returns `err.statusCode = 403` and an error message. Here is the relevant section of code within the function.

JavaScript

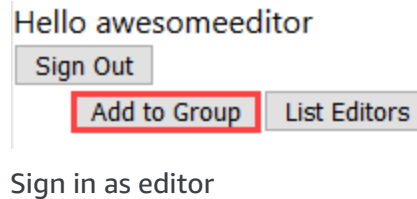
```
// Only perform tasks if the user is in a specific group
const allowedGroup = process.env.GROUP;
...
// Fail if group enforcement is being used
if (req.apiGateway.event.requestContext.authorizer.claims['cognito:groups']) {
  const groups = req.apiGateway.event.requestContext.authorizer.claims['cognito:groups'];
  if (!(allowedGroup && groups.indexOf(allowedGroup) > -1)) {
    const err = new Error('User does not have permissions to perform administrative tasks');
    err.statusCode = 403;
    next(err);
  }
} else {
  const err = new Error('User does not have permissions to perform administrative tasks');
  err.statusCode = 403;
  next(err);
}
```

This example shows using a JWT to perform authorization within a Lambda function.

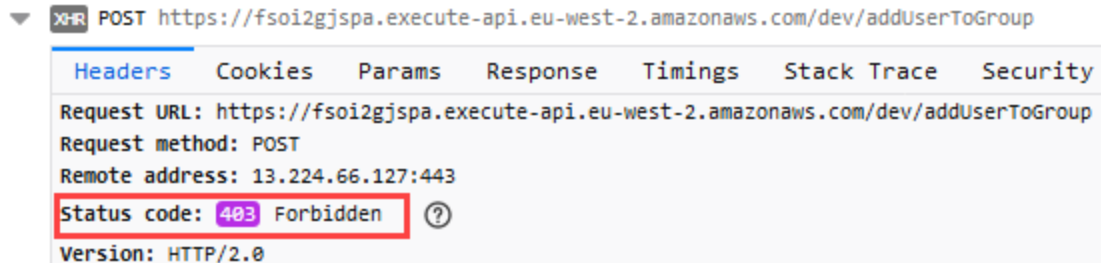
If the authorization is successful, the function continues and adds the *awesomeeditor* user to the *editors* group.

To show this flow in action:

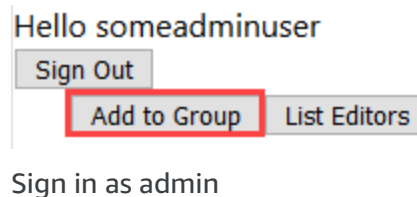
1. I log on to the web application using the *awesomeeditor* account, which is not a member of the *admin* group.
I choose the **Add to Group** button.



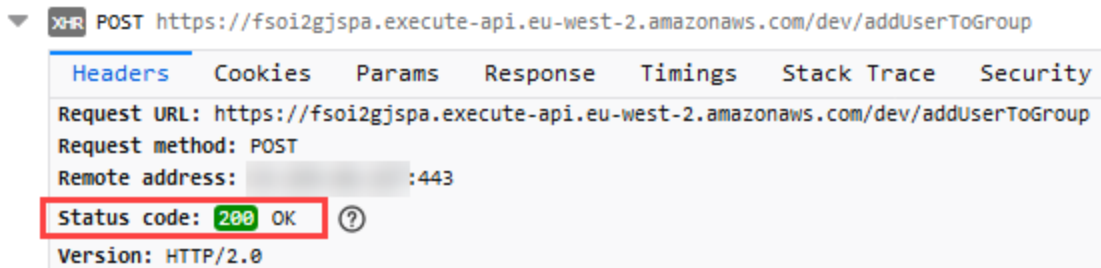
1. Using the browser developer tools I see that the API request has failed, returning the 403 error code from the Lambda function.



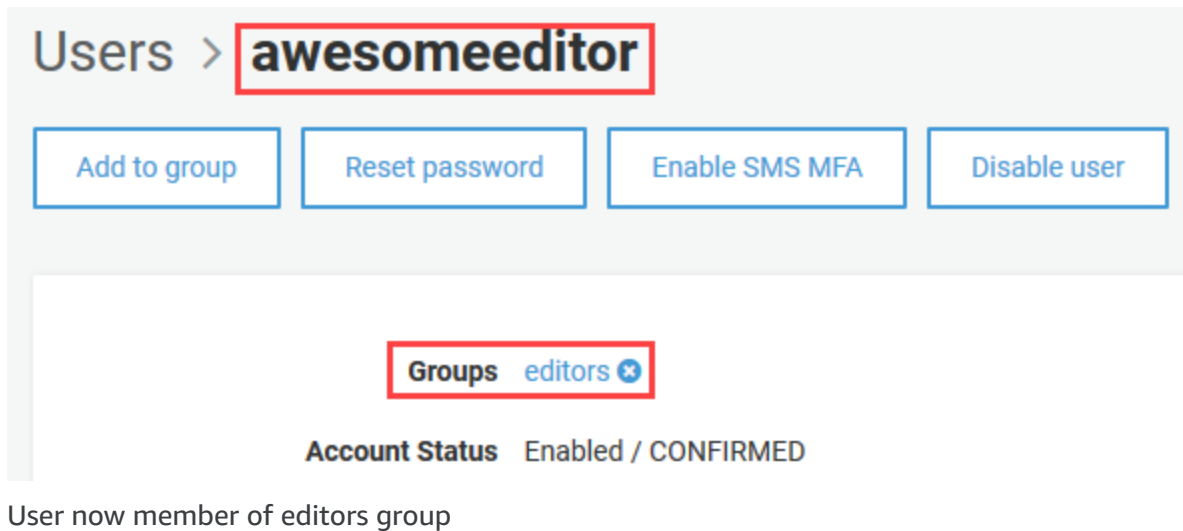
1. I log on to the web application using the *someadminuser* account and choose the **Add to Group** button.



1. Using the browser developer tools I see that the API request is now successful as the user is a member of the *admin* group.



1. I navigate back to the Amazon Cognito user pool console, and view **Users and groups**. The *awesomeeditor* user is now a member of the *editors* group.



The Lambda function has added the *awesomeeditor* account to the *editors* group.

Implement authorization logic based on authentication metadata

Another way to separate users for authorization is using Amazon Cognito to define a resource server with custom scopes.

A resource server is a server for access-protected resources. It handles authenticated requests from an app that has an access token. This API can be hosted in Amazon API Gateway or outside of AWS. A scope is a level of access that an app can request to a resource. For example, if you have a resource server for airline flight details, it might define two scopes. One scope for all customers with read access to view the flight details, and one for airline employees with write access to add new flights. When the app makes an API call to request access and passes an access token, the token has one or more embedded scopes.

```
{
  "origin_jti": "b4dfa2c5-80fa-443f-bc95-
d559a8aa2aff",
  "sub": "f2cc70f7-a58b-4d31-bc75-e762408698a2",
  "event_id": "81975df2-
c631-41f2-8819-8e8c94c0fe78",
  "token_use": "access",
  "scope": "aws.cognito.signin.user.admin",
  "auth_time": 1595603961,
  "iss": "https://cognito-idp.eu-
west-2.amazonaws.com/eu-west-2_FEAH74vPw",
  "exp": 1596130551,
  "iat": 1596126951,
  "jti": "d2cb453e-8b92-4fb3-8d2b-d1c1400914e4",
  "client_id": "79imhmi57au44pb09oimic15jf",
  "username": "awesomeeditor"
}
```

JWT with scope

This allows you to provide different access levels to API resources for different application clients based on the custom scopes. It is another mechanism for separating users during authentication.

For authorizing based on token claims, use an [API Gateway Lambda authorizer](#).

For more information, see [“Using Amazon Cognito User Pool Scopes with Amazon API Gateway”](#).

With [AWS AppSync](#), use GraphQL resolvers. AWS Amplify can also generate fine-grained authorization logic via GraphQL transformers (directives). You can annotate your GraphQL schema to a specific data type, data field, and specific GraphQL operation you want to allow access. These can include JWT groups or custom claims. For more information, see [“GraphQL API Security with AWS AppSync and Amplify”](#), and the AWS AppSync documentation for [Authorization Use Cases](#), and [fine-grained access control](#).

Improvement plan summary:

1. Review levels of access, identity metadata and separate consumers into logical groups/tiers.
2. Implement authorization logic based on authentication metadata

Conclusion

Controlling serverless application API access using authentication and authorization mechanisms can help protect against unauthorized access and prevent unnecessary use of resources. In [part 1](#), I cover the different mechanisms for authorization available for API Gateway and AWS AppSync. I explain the different approaches for public or private endpoints and show how to use IAM to control access to internal or private API consumers.

In [part 2](#), I cover using Amplify CLI to add a GraphQL API with an Amazon Cognito user pool handling authentication. I explain how to view JSON Web Token (JWT) claims, and how to use Amazon Cognito identity pools to grant temporary access to AWS services. I also show how to use API keys and API Gateway usage plans for rate limiting and throttling requests.

In this post, I cover separating authenticated users into logical groups. I first show how to use Amazon Cognito user pool groups to separate users with an Amazon Cognito authorizer to control access to an API Gateway method. I also show how JWTs can be passed to a Lambda function to perform authorization within a function. I then explain how to also separate users using custom scopes by defining an Amazon Cognito resource server.

In the next post in the series, I cover the second security question from the Well-Architected Serverless Lens – [Managing serverless security boundaries](#).

TAGS: [Amazon API Gateway](#), [Amazon Cognito](#), [AppSync](#), [AWS Amplify](#), [AWS CloudFormation](#), [AWS Lambda](#), [IAM](#), [serverless](#), [well-architected](#)