API (GRAPHQL)

Setup authorization rules

@auth

Authorization is required for applications to interact with your GraphQL API. **API Keys** are best used for public APIs (or parts of your schema which you wish to be public) or prototyping, and you must specify the expiration time before deploying. **IAM** authorization uses Signature Version 4 to make request with policies attached to Roles. OIDC tokens provided by **Amazon Cognito User Pools** or 3rd party OpenID Connect providers can also be used for authorization, and simply enabling this provides a simple access control requiring users to authenticate to be granted top level access to API actions. You can set finer grained access controls using `@auth` on your schema which leverages authorization metadata provided as part of these tokens or set on the database items themselves.

`@auth` object types that are annotated with `@auth` are protected by a set of authorization rules giving you additional controls than the top level authorization on an API. You may use the `@auth` directive on object type definitions and field definitions in your project's schema.

When using the `@auth` directive on object type definitions that are also annotated with `@model`, all resolvers that return objects of that type will be protected. When using the `@auth` directive on a field definition, a resolver will be added to the field that authorize access based on attributes found in the parent type.

Definition

copy

```
# When applied to a type, augments the application with
# owner and group-based authorization rules.
directive @auth(rules: [AuthRule!]!) on OBJECT | FIELD_DEFINITION
input AuthRule {
  allow: AuthStrategy!
```

```
  provider: AuthProvider
  ownerField: String # defaults to "owner" when using owner auth
  identityClaim: String # defaults to "username" when using owner auth
  groupClaim: String # defaults to "cognito:groups" when using Group auth
  groups: [String]  # Required when using Static Group auth
  groupsField: String # defaults to "groups" when using Dynamic Group auth
  operations: [ModelOperation] # Required for finer control

  # The following arguments are deprecated. It is encouraged to use the 'operations' argument
  queries: [ModelQuery]
  mutations: [ModelMutation]
}
enum AuthStrategy { owner groups private public }
enum AuthProvider { apiKey iam oidc userPools }
enum ModelOperation { create update delete read }

# The following objects are deprecated. It is encouraged to use ModelOperations.
enum ModelQuery { get list }
enum ModelMutation { create update delete }
```

> Note: The operations argument was added to replace the 'queries' and 'mutations' arguments. The 'queries' and 'mutations' arguments will continue to work but it is encouraged to move to 'operations'. If both are provided, the 'operations' argument takes precedence over 'queries'.

Owner authorization

By default, enabling `owner` authorization allows any signed in user to create records.

copy

```
# The simplest case
type Post @model @auth(rules: [{ allow: owner }]) {
  id: ID!
  title: String!
}

# The long form way
type Post
  @model
  @auth(
```

```
    rules: [
      { allow: owner, ownerField: "owner", operations: [create, update, delete, read] },
    ])
{
  id: ID!
  title: String!
  owner: String
}
```

> Owner authorization requires an authentication type of **Amazon Cognito User Pools** to be enabled in your app.

Owner authorization specifies whether a user can access or operate against an object. To do so, each object will get an `ownerField` field (by default `owner` will be added to the object if not specified) that stores ownership information and is verified in various ways during resolver execution.

You can use the `operations` argument to specify which operations are enabled as follows:

- **read**: Allow the user to perform queries (`get` and `list` operations) against the API.
- **create**: Inject the logged in user's identity as the `ownerField` automatically.
- **update**: Add conditional update that checks the stored `ownerField` is the same as the signed in user.
- **delete**: Add conditional update that checks the stored `ownerField` is the same as the signed in user.

You must ensure that the `create` operations rule is specified explicitly or inferred from defaults to ensure that the owner's identity is stored with the record so it can be verified on subsequent requests.

copy

```
# owner identity inferred from defaults on every object
type Post @model @auth(rules: [{ allow: owner }]) {
  id: ID!
  title: String!
}

# owner identity specified explicitly on every object
```

```
type Post @model @auth(rules: [{ allow: owner, operations: [create] }]) {
  id: ID!
  title: String!
}

# owner identity not stored on objects
type Post @model @auth(rules: [{ allow: owner, operations: [read] }]) {
  id: ID!
  title: String!
}
```

> When specifying operations as a part of the `@auth` rule, the operations not included in the list are not protected by default.

Let's take a look at a few examples:

copy

```
type Todo @model
  @auth(rules: [{ allow: owner }]) {
  id: ID!
  updatedAt: AWSDateTime!
  content: String!
}
```

In this schema, only the owner of the object has the authorization to perform read (`getTodo` and `listTodos`), update (`updateTodo`), and delete (`deleteTodo`) operations on the owner created object. This prevents the object from being updated or deleted by users other than the creator of the object.

Here's a table outlining which users are permitted to execute which operations. **owner** refers to the user who created the object, **other** refers to all other authenticated users.

	getTodo	listTodos	createTodo	updateTodo	deleteTodo
owner	✓	✓	✓	✓	✓
other	✗	✗	✓	✗	✗

Next, let's say that you wanted to modify the schema to allow only the owner of the object to be able to update or delete, but allow any authenticated user to read the objects.

copy

```
type Todo @model
  @auth(rules: [{ allow: owner, operations: [create, delete, update] }]) {
  id: ID!
  updatedAt: AWSDateTime!
  content: String!
}
```

In this schema, only the owner of the object has the authorization to perform update (`updateTodo`) and delete (`deleteTodo`) operations on the owner created object, but anyone can read them (`getTodo` , `listTodos`). This prevents the object from being updated or deleted by users other than the creator of the object while allowing all authenticated users of the app to read them.

Here's a table outlining which users are permitted to execute which operations. **owner** refers to the user who created the object, **other** refers to all other authenticated users.

	getTodo	listTodos	createTodo	updateTodo	deleteTodo
owner	✓	✓	✓	✓	✓
other	✓	✓	✓	✗	✗

Next, let's say that you wanted to modify the schema to allow only the owner of the object to be able to delete, but allow anyone to create, read, and update.

copy

```
type Todo @model
  @auth(rules: [{ allow: owner, operations: [create, delete] }]) {
  id: ID!
  updatedAt: AWSDateTime!
  content: String!
}
```

In this schema, only the owner of the object has the authorization to perform delete operations on the owner created object, but anyone can read or update them. This is because `read` and `update` aren't specified as owner-only actions, so all users are able to perform them. Since `delete` is specified as an owner only action, only the object's creator can delete the object.

Here's a table outlining which users are permitted to execute which operations. **owner** refers to the user who created the object, **other** refers to all other authenticated users.

	getTodo	listTodos	createTodo	updateTodo	deleteTodo
owner	✓	✓	✓	✓	✓
other	✓	✓	✓	✓	✗

Multiple authorization rules

You may also apply multiple ownership rules on a single `@model` type.

For example, imagine you have a type **Draft** that stores unfinished posts for a blog. You might want to allow the **Draft's** owner to `create`, `update`, `delete`, and `read` **Draft** objects. However, you might also want the **Draft's editors** to be able to update and read **Draft** objects.

To allow for this use case you could use the following type definition:

copy

```
type Draft @model
  @auth(rules: [
    # Defaults to use the "owner" field.
    { allow: owner },

    # Authorize the update mutation and both queries.
    { allow: owner, ownerField: "editors", operations: [update, read] }
  ]) {
  id: ID!
  title: String!
  content: String
  owner: String
  editors: [String]
}
```

Ownership with create mutations

The ownership authorization rule will automatically fill ownership fields unless told explicitly not to do so. To show how this works, lets look at how the create mutation would work for the **Draft** type above:

```
mutation CreateDraft {
  createDraft(input: { title: "A new draft" }) {
    id
    title
    owner
    editors
  }
}
```

Let's assume that when I call this mutation I am logged in as `someuser@my-domain.com`. The result would be:

copy

```
{
    "data": {
        "createDraft": {
            "id": "...",
            "title": "A new draft",
            "owner": "someuser@my-domain.com",
            "editors": ["someuser@my-domain.com"]
        }
    }
}
```

The `Mutation.createDraft` resolver is smart enough to match your auth rules to attributes and will fill them in be default. If you do not want the value to be automatically set all you need to do is include a value for it in your input.

For example, to have the resolver automatically set the **owner** but not the **editors**, you would run this:

copy

```
mutation CreateDraft {
  createDraft(
    input: {
      title: "A new draft",
      editors: []
    }
  ) {
    id
```

```
    title
    owner
    editors
  }
}
```

This would return:

copy

```
{
    "data": {
        "createDraft": {
            "id": "...",
            "title": "A new draft",
            "owner": "someuser@my-domain.com",
            "editors": []
        }
    }
}
```

To specify a list of custom **editors**, you could run this:

copy

```
mutation CreateDraft {
  createDraft(
    input: {
      title: "A new draft",
      editors: ["editor1@my-domain.com", "editor2@my-domain.com"]
    }
  ) {
    id
    title
    owner
    editors
  }
}
```

This would return:

```
{
    "data": {
        "createDraft": {
            "id": "...",
            "title": "A new draft",
            "owner": "someuser@my-domain.com",
            "editors": ["editor1@my-domain.com", "editor2@my-domain.com"]
        }
    }
}
```

You can try to perform the same modification to **owner** but this will throw an **Unauthorized** exception because you are no longer the owner of the object you are trying to create.

copy

```
mutation CreateDraft {
  createDraft(
    input: {
      title: "A new draft",
      editors: [],
      owner: null
    }
  ) {
    id
    title
    owner
    editors
  }
}
```

Static group authorization

Static group authorization allows you to protect `@model` types by restricting access to a known set of groups. For example, you can allow all **Admin** users to create, update, delete, get, and list Salary objects.

copy

```
type Salary @model @auth(rules: [{ allow: groups, groups: ["Admin"] }]) {
  id: ID!
  wage: Int

```

```
  currency: String
}
```

When calling the GraphQL API, if the user credential (as specified by the resolver's `$ctx.identity`) is not enrolled in the *Admin* group, the operation will fail.

To enable advanced authorization use cases, you can layer auth rules to provide specialized functionality. To show how you might do that, let's expand the **Draft** example you started in the **Owner Authorization** section above. When you last left off, a **Draft** object could be updated and read by both its owner and any of its editors and could be created and deleted only by its owner. Let's change it so that now any member of the "Admin" group can also create, update, delete, and read a **Draft** object.

copy

```
type Draft @model
  @auth(rules: [
    # Defaults to use the "owner" field.
    { allow: owner },

    # Authorize the update mutation and both queries.
    { allow: owner, ownerField: "editors", operations: [update] },

    # Admin users can access any operation.
    { allow: groups, groups: ["Admin"] }
  ]) {
  id: ID!
  title: String!
  content: String
  owner: String
  editors: [String]!
}
```

Dynamic group authorization

copy

```
# Dynamic group authorization with multiple groups
type Post @model @auth(rules: [{ allow: groups, groupsField: "groups" }]) {
  id: ID!
  title: String
  groups: [String]
```

```
}

# Dynamic group authorization with a single group
type Post @model @auth(rules: [{ allow: groups, groupsField: "group" }]) {
  id: ID!
  title: String
  group: String
}
```

With dynamic group authorization, each record contains an attribute specifying what groups should be able to access it. Use the *groupsField* argument to specify which attribute in the underlying data store holds this group information. To specify that a single group should have access, use a field of type `String`. To specify that multiple groups should have access, use a field of type `[String]`.

Just as with the other auth rules, you can layer dynamic group rules on top of other rules. Let's again expand the **Draft** example from the **Owner Authorization** and **Static Group Authorization** sections above. When you last left off editors could update and read objects, owners had full access, and members of the admin group had full access to **Draft** objects. Now you have a new requirement where each record should be able to specify an optional list of groups that can read the draft. This would allow you to share an individual document with an external team, for example.

copy

```
type Draft @model
  @auth(rules: [
    # Defaults to use the "owner" field.
    { allow: owner },

    # Authorize the update mutation and both queries.
    { allow: owner, ownerField: "editors", operations: [update] },

    # Admin users can access any operation.
    { allow: groups, groups: ["Admin"] }

    # Each record may specify which groups may read them.
    { allow: groups, groupsField: "groupsCanAccess", operations: [read] }
  ]) {
  id: ID!
  title: String!
```

```
  content: String
  owner: String
  editors: [String]!
  groupsCanAccess: [String]!
}
```

With this setup, you could create an object that can be read by the “BizDev” group:

copy

```
mutation CreateDraft {
  createDraft(input: {
    title: "A new draft",
    editors: [],
    groupsCanAccess: ["BizDev"]
  }) {
    id
    groupsCanAccess
  }
}
```

And another draft that can be read by the “Marketing” group:

copy

```
mutation CreateDraft {
  createDraft(input: {
    title: "Another draft",
    editors: [],
    groupsCanAccess: ["Marketing"]
  }) {
    id
    groupsCanAccess
  }
}
```

`public` authorization

copy

```
# The simplest case
type Post @model @auth(rules: [{ allow: public }]) {
  id: ID!
```

```
  title: String!
}
```

The `public` authorization specifies that everyone will be allowed to access the API, behind the scenes the API will be protected with an API Key. To be able to use `public` the API must have API Key configured. For local execution, this key resides in the file `aws-exports.js` for the JavaScript library and `amplifyconfiguration.json` for Android and iOS under the key `aws_appsync_apiKey` .

copy

```
# public authorization with provider override
type Post @model @auth(rules: [{ allow: public, provider: iam }]) {
  id: ID!
  title: String!
}
```

The @auth directive allows the override of the default provider for a given authorization mode. In the sample above iam is specified as the provider which allows you to use an “UnAuthenticated Role” from Cognito Identity Pools for public access, instead of an API Key. When used in conjunction with amplify add auth the CLI generates scoped down IAM policies for the “UnAuthenticated” role automatically.

`private` authorization

copy

```
# The simplest case
type Post @model @auth(rules: [{ allow: private }]) {
  id: ID!
  title: String!
}
```

The `private` authorization specifies that everyone will be allowed to access the API with a valid JWT token from the configured Cognito User Pool. To be able to use `private` the API must have Cognito User Pool configured.

copy

```
# private authorization with provider override
type Post @model @auth(rules: [{ allow: private, provider: iam }]) {
  id: ID!
  title: String!
}
```

The @auth directive allows the override of the default provider for a given authorization mode. In the sample above iam is specified as the provider which allows you to use an "Authenticated Role" from Cognito Identity Pools for private access. When used in conjunction with amplify add auth the CLI generates scoped down IAM policies for the "Authenticated" role automatically.

Authorization using an `oidc` provider

copy

```
# owner authorization with provider override
type Profile @model @auth(rules: [{ allow: owner, provider: oidc, identityClaim: "sub" }]) {
  id: ID!
  displayNAme: String!
}
```

By using a configured `oidc` provider for the API, it is possible to authenticate the users against it. In the sample above, `oidc` is specified as the provider for the `owner` authorization on the type. The field `identityClaim: "sub"` specifies that the `"sub"` claim from your JWT token is used to provider ownership instead of the default `username` claim, which is used by the Amazon Cognito JWT.

Combining multiple authorization types

Amplify GraphQL APIs have a primary **default** authentication type and, optionally, additional secondary authentication types. The objects and fields in the GraphQL schema can have rules with different authorization providers assigned based on the authentication types configured in your app.

One of the most common scenarios for multiple authorization rules is for combining public and private access. For example, blogs typically allow public access for viewing a post but only allow a post's creator to update or delete that post.

Let's take a look at how you can combine public and private access to achieve this:

copy

```
type Post @model
  @auth (
    rules: [
      # allow all authenticated users ability to create posts
      # allow owners ability to update and delete their posts
      { allow: owner },
```

```
      # allow all authenticated users to read posts
      { allow: private, operations: [read] },

      # allow all guest users (not authenticated) to read posts
      { allow: public, operations: [read] }
    ]
  ) {
  id: ID!
  title: String
  owner: String
}
```

The above schema assumes a combination of **Amazon Cognito User Pools** and **API key** authentication types

Let's take a look at another example. Here the `Post` model is protected by Cognito User Pools by default and the `owner` can perform any operation on the `Post` type. You can also call `getPosts` and `listPosts` from an AWS Lambda function if it is configured with the appropriate IAM policies in its execution role.

copy

```
type Post @model
  @auth (
    rules: [
      { allow: owner },
      { allow: private, provider: iam, operations: [read] }
    ]
  ) {
  id: ID!
  title: String
  owner: String
}
```

The above schema assumes a combination of **Amazon Cognito User Pools** and **IAM** authentication types

Allowed authorization mode vs. provider combinations

The following table shows the allowed combinations of authorization modes and providers.

	owner	groups	public	private
userPools	✓	✓		✓
oidc	✓	✓		
apiKey			✓	
iam			✓	✓

Please note that `groups` is leveraging Cognito User Pools but no provider assignment needed/possible.

Custom claims

`@auth` supports using custom claims if you do not wish to use the default `username` or `cognito:groups` claims from your JWT token which are populated by Amazon Cognito. This can be helpful if you are using tokens from a 3rd party OIDC system or if you wish to populate a claim with a list of groups from an external system, such as when using a Pre Token Generation Lambda Trigger which reads from a database. To use custom claims specify `identityClaim` or `groupClaim` as appropriate like in the example below:

copy

```
type Post
@model
@auth(rules: [
    { allow: owner, identityClaim: "user_id" },
    { allow: groups, groups: ["Moderator"], groupClaim: "user_groups" }
])
{
  id: ID!
  owner: String
  postname: String
  content: String
}
```

In this example the object owner will check against a `user_id` claim. Please note that this is not available by default if the token is generated by Cognito. Use `sub` instead if you are using

Cognito generated token. Similarly if the `user_groups` claim contains a “Moderator” string then access will be granted.

> Note `identityField` is being deprecated for `identityClaim` .

Authorizing subscriptions

> Prior to version 2.0 of the CLI, `@auth` rules did not apply to subscriptions. Instead you were required to either turn them off or use Custom Resolvers to manually add authorization checks. In the latest versions `@auth` protections have been added to subscriptions, however this can introduce different behavior into existing applications: First, `owner` is now a required argument for Owner-based authorization, as shown below. Second, the selection set will set `null` on fields when mutations are invoked if per-field `@auth` is set on that field. Read more here. If you wish to keep the previous behavior set `level: public` on your model as defined below.

When `@auth` is used subscriptions have a few subtle behavior differences than queries and mutations based on their event based nature. When protecting a model using the owner auth strategy, each subscription request will **require** that the user is passed as an argument to the subscription request. If the user field is not passed, the subscription connection will fail. In the case where it is passed, the user will only get notified of updates to records for which they are the owner.

> Subscription filtering uses data passed from mutation to do the filtering. If a mutation does not include `owner` field in the selection set of a owner based auth, Subscription message won't be fired for that mutation.

Alternatively, when the model is protected using the static group auth strategy, the subscription request will only succeed if the user is in an allowed group. Further, the user will only get notifications of updates to records if they are in an allowed group. Note: You don’t need to pass the user as an argument in the subscription request, since the resolver will instead check the contents of your JWT token.

> Dynamic groups have no impact to subscriptions. You will not get notified of any updat
> to them.

For example suppose you have the following schema:

```
type Post @model
@auth(rules: [{allow: owner}])
{
  id: ID!
  owner: String
  postname: String
  content: String
}
```

This means that the subscription must look like the following or it will fail:

```
subscription OnCreatePost {
  onCreatePost(owner: “Bob”){
    postname
    content
  }
}
```

Note that if your type doesn’t already have an `owner` field the Transformer will automatically add this for you. Passing in the current user can be done dynamically in your code by using Auth.currentAuthenticatedUser() in JavaScript, AWSMobileClient.default().username in iOS, or AWSMobileClient.getInstance().getUsername() in Android.

In the case of groups if you define the following:

```
type Post @model
@auth(rules: [{ allow: groups, groups: ["Admin"] }])
{
  id: ID!
  owner: String
  postname: String
  content: String
}
```

Then you don’t need to pass an argument, as the resolver will check the contents of your J token at subscription time and ensure you are in the “Admin” group.

Finally, if you use both owner and group authorization then the username argument becomes optional. This means the following:

- If you don't pass the user in, but are a member of an allowed group, the subscription will notify you of records added.
- If you don't pass the user in, but are NOT a member of an allowed group, the subscription will fail to connect.
- If you pass the user in who IS the owner but is NOT a member of a group, the subscription will notify you of records added of which you are the owner.
- If you pass the user in who is NOT the owner and is NOT a member of a group, the subscription will not notify you of anything as there are no records for which you own

You may disable authorization checks on subscriptions or completely turn off subscriptions as well by specifying either `public` or `off` in `@model`:

copy

```
@model (subscriptions: { level: public })
```

Field level authorization

The `@auth` directive specifies that access to a specific field should be restricted according to its own set of rules. Here are a few situations where this is useful:

Protect access to a field that has different permissions than the parent model

You might want to have a user model where some fields, like *username*, are a part of the public profile and the *ssn* field is visible to owners.

copy

```
type User @model {
  id: ID!
  username: String
  ssn: String @auth(rules: [{ allow: owner, ownerField: "username" }])
}
```

Protect access to a `@connection` resolver based on some attribute in the source object

This schema will protect access to Post objects connected to a user based on an attribute in the User model. You may turn off top level queries by specifying `queries: null` in the `@model` declaration which restricts access such that queries must go through the `@connection` resol
to reach the model.

copy

```
type User @model {
  id: ID!
  username: String
  posts: [Post]
    @connection(name: "UserPosts")
    @auth(rules: [{ allow: owner, ownerField: "username" }])
}
type Post @model(queries: null) { ... }
```

Protect mutations such that certain fields can have different access rules than the parent model

When used on field definitions, `@auth` directives protect all operations by default. To protect read operations, a resolver is added to the protected field that implements authorization logic. To protect mutation operations, logic is added to existing mutations that will be run if the mutation's input contains the protected field. For example, here is a model where owners and admins can read employee salaries but only admins may create or update them.

copy

```
type Employee @model {
  id: ID!
  email: String
  username: String

  # Owners & members of the "Admin" group may read employee salaries.
  # Only members of the "Admin" group may create an employee with a salary
  # or update a salary.
  salary: String
    @auth(rules: [
      { allow: owner, ownerField: "username", operations: [read] },
      { allow: groups, groups: ["Admin"], operations: [create, update, read] }
    ])
}
```

Note The `delete` operation, when used in @auth directives on field definitions, translates to protecting the update mutation such that the field cannot be set to null unless authorized.

Note: When specifying operations as a part of the @auth rule on a field, the operations not included in the operations list are not protected by default. For example, let's say you have following schema:

copy

```
type Todo
  @model
{
  id: ID!
  owner: String
  updatedAt: AWSDateTime!
  content: String! @auth(rules: [{ allow: owner, operations: [update] }])
}
```

In this schema, only the owner of the object has the authorization to perform update operations on the `content` field. But this does not prevent any other owner (any user other than the creator or owner of the object) to update some other field in the object owned by another user. If you want to prevent update operations on a field, the user would need to explicitly add auth rules to restrict access to that field. One of the ways would be to explicitly specify @auth rules on the fields that you would want to protect like the following:

copy

```
type Todo
  @model
{
  id: ID!
  owner: String
  updatedAt: AWSDateTime! @auth(rules: [{ allow: owner, operations: [update] }]) // or @auth(r
  content: String! @auth(rules: [{ allow: owner, operations: [update] }])
}
```

You can also provide explicit deny rules to your field like the following:

copy

```
type Todo
  @model
{
  id: ID!
  owner: String
  updatedAt: AWSDateTime! @auth(rules: [{ allow: groups, groups: ["ForbiddenGroup"], operatior
  content: String! @auth(rules: [{ allow: owner, operations: [update] }])
}
```

You can also combine top-level @auth rules on the type with field level auth rules. For example, let's consider the following schema:

```
type Todo
  @model @auth(rules: [{ allow: groups, groups: ["Admin"], operations:[update] }])
{
  id: ID!
  owner: String
  updatedAt: AWSDateTime!
  content: String! @auth(rules: [{ allow: owner, operations: [update] }])
}
```

In the above schema users in the `Admin` group have the authorization to create, read, delete and update (except the `content` field in the object of another owner) for the type Todo. An `owner` of an object, has the authorization to create Todo types and read all the objects of type Todo. In addition an `owner` can perform an update operation on the Todo object, only when the `content` field is present as a part of the input. Any other user - who isn't an owner of an object isn't authorized to update that object.

Per-Field with subscriptions

When setting per-field `@auth` the Transformer will alter the response of mutations for those fields by setting them to `null` in order to prevent sensitive data from being sent over subscriptions. For example in the schema below:

```
type Employee @model
  @auth(rules: [
      { allow: owner },
      { allow: groups, groups: ["Admins"] }
  ]) {
    id: ID!
    name: String!
}
```

Getting Started Libraries UI Components **CLI** Console Guides API Reference

Subscribers might be a member of the "Admins" group and should get notified of the new i
however they should not get the `ssn` field. If you run the following mutation:

copy

```
mutation {
  createEmployee(input: {
    name: "Nadia"
    address: "123 First Ave"
    ssn: "392-95-2716"
  }){
    name
    address
    ssn
  }
}
```

The mutation will run successfully, however `ssn` will return null in the GraphQL response. This prevents anyone in the “Admins” group who is subscribed to updates from receiving the private information. Subscribers would still receive the `name` and `address`. The data is still written and this can be verified by running a query.

Generates

The `@auth` directive will add authorization snippets to any relevant resolver mapping templates at compile time. Different operations use different methods of authorization.

Owner Authorization

copy

```
type Post @model @auth(rules: [{ allow: owner }]) {
  id: ID!
  title: String!
}
```

The generated resolvers would be protected like so:

- `Mutation.createX` : Verify the requesting user has a valid credential and automatically set the **owner** attribute to equal `$ctx.identity.username`.
- `Mutation.updateX` : Update the condition expression so that the DynamoDB `UpdateItem` operation only succeeds if the record’s **owner** attribute equals the caller’s `$ctx.identity.username`.
- `Mutation.deleteX` : Update the condition expression so that the DynamoDB `DeleteItem` operation only succeeds if the record’s **owner** attribute equals the caller’s `$ctx.identity.username`.

- `Query.getX` : In the response mapping template verify that the result's **owner** attribute is the same as the `$ctx.identity.username` . If it is not return null.
- `Query.listX` : In the response mapping template filter the result's **items** such that only items with an **owner** attribute that is the same as the `$ctx.identity.username` are returned.
- `@connection` resolvers: In the response mapping template filter the result's **items** such that only items with an **owner** attribute that is the same as the `$ctx.identity.username` are returned. This is not enabled when using the `queries` argument.

Static group authorization

copy

```
type Post @model @auth(rules: [{ allow: groups, groups: ["Admin"] }]) {
  id: ID!
  title: String!
  groups: String
}
```

Static group auth is simpler than the others. The generated resolvers would be protected like so:

- `Mutation.createX` : Verify the requesting user has a valid credential and that `$ctx.identity.claims.get("cognito:groups")` contains the **Admin** group. If it does not, fail.
- `Mutation.updateX` : Verify the requesting user has a valid credential and that `$ctx.identity.claims.get("cognito:groups")` contains the **Admin** group. If it does not, fail.
- `Mutation.deleteX` : Verify the requesting user has a valid credential and that `$ctx.identity.claims.get("cognito:groups")` contains the **Admin** group. If it does not, fail.
- `Query.getX` : Verify the requesting user has a valid credential and that `$ctx.identity.claims.get("cognito:groups")` contains the **Admin** group. If it does not, fail.
- `Query.listX` : Verify the requesting user has a valid credential and that `$ctx.identity.claims.get("cognito:groups")` contains the **Admin** group. If it does not, fail.
- `@connection` resolvers: Verify the requesting user has a valid credential and that `$ctx.identity.claims.get("cognito:groups")` contains the **Admin** group. If it does not, fail. This is not enabled when using the `queries` argument.

Dynamic group authorization

copy

```
type Post @model @auth(rules: [{ allow: groups, groupsField: "groups" }]) {
  id: ID!
  title: String!
```

```
    groups: String
}
```

The generated resolvers would be protected like so:

- `Mutation.createX` : Verify the requesting user has a valid credential and that it contains a claim to at least one group passed to the query in the `$ctx.args.input.groups` argument.
- `Mutation.updateX` : Update the condition expression so that the DynamoDB `UpdateItem` operation only succeeds if the record's **groups** attribute contains at least one of the caller's claimed groups via `$ctx.identity.claims.get("cognito:groups") .`
- `Mutation.deleteX` : Update the condition expression so that the DynamoDB `DeleteItem` operation only succeeds if the record's **groups** attribute contains at least one of the caller's claimed groups via `$ctx.identity.claims.get("cognito:groups")`
- `Query.getX` : In the response mapping template verify that the result's **groups** attribute contains at least one of the caller's claimed groups via `$ctx.identity.claims.get("cognito:groups") .`
- `Query.listX` : In the response mapping template filter the result's **items** such that only items with a

groups attribute that contains at least one of the caller's claimed groups via `$ctx.identity.claims.get("cognito:groups") .`

- `@connection` resolver: In the response mapping template filter the result's **items** such that only items with a

groups attribute that contains at least one of the caller's claimed groups via `$ctx.identity.claims.get("cognito:groups")` . This is not enabled when using the `queries` argument.

PREVIOUS
Index your data with keys

NEXT
Add relationships between types

Members
Newsletters