

# Application roles

07/21/2017 • 5 minutes to read • Contributors      all

## In this article

[Roles using Azure AD App Roles](#)

[Roles using Azure AD security groups](#)

[Roles using an application role manager](#)



[Sample code](#)

Application roles are used to assign permissions to users. For example, the [Tailspin Surveys](#) application defines the following roles:

- Administrator. Can perform all CRUD operations on any survey that belongs to that tenant.
- Creator. Can create new surveys.
- Reader. Can read any surveys that belong to that tenant.

You can see that roles ultimately get translated into permissions, during [authorization](#). But the first question is how to assign and manage roles. We identified three main options:

- [Azure AD App Roles](#)
- [Azure AD security groups](#)
- [Application role manager](#).

## Roles using Azure AD App Roles

This is the approach that we used in the Tailspin Surveys app.

In this approach, The SaaS provider defines the application roles by adding them to the application manifest. After a customer signs up, an admin for the customer's AD directory assigns users to the roles. When a user signs in, the user's assigned roles are sent as claims.

### Note

If the customer has Azure AD Premium, the admin can assign a security group to a role, and members of the group will inherit the app role. This is a convenient way to manage roles, because the group owner doesn't need to be an AD admin.

Advantages of this approach:


- Simple programming model.
- Roles are specific to the application. The role claims for one application are not sent to another application.
- If the customer removes the application from their AD tenant, the roles go away.
- The application doesn't need any extra Active Directory permissions, other than reading the user's profile.

Drawbacks:

- Customers without Azure AD Premium cannot assign security groups to roles. For these customers, all user assignments must be done by an AD administrator.
- If you have a backend web API, which is separate from the web app, then role assignments for the web app don't apply to the web API. For more discussion of this point, see [Securing a backend web API](#).

## Implementation


**Define the roles.** The SaaS provider declares the app roles in the [application manifest](#). For example, here is the manifest entry for the Surveys app:

JSON 

```
"appRoles": [
  {
    "allowedMemberTypes": [
      "User"
    ],
    "description": "Creators can create Surveys",
    "displayName": "SurveyCreator",
    "id": "1b4f816e-5eaf-48b9-8613-7923830595ad",
    "isEnabled": true,
    "value": "SurveyCreator"
  },
  {
    "allowedMemberTypes": [
      "User"
    ],
    "description": "Administrators can manage the Surveys in their tenant",
    "displayName": "SurveyAdmin",
    "id": "c20e145e-5459-4a6c-a074-b942bbd4cfe1",
    "isEnabled": true,
    "value": "SurveyAdmin"
  }
],
```

The `value` property appears in the role claim. The `id` property is the unique identifier for the defined role. Always generate a new GUID value for `id`.

**Assign users.** When a new customer signs up, the application is registered in the customer's AD tenant. At this point, an AD admin for that tenant can assign users to roles.

 **Note**

As noted earlier, customers with Azure AD Premium can also assign security groups to roles.

The following screenshot from the Azure portal shows users and groups for the Survey application. Admin and Creator are groups, assigned to SurveyAdmin and SurveyCreator roles respectively. Alice is a user who was assigned directly to the SurveyAdmin role. Bob and Charles are users that have not been directly assigned to a role.

DISPLAY NAME	OBJECT TYPE	ROLE ASSIGNED
Admin	Group	SurveyAdmin
Alice	User	SurveyAdmin
Bob	User	Default Access
Charles	User	Default Access
Creators	Group	SurveyCreator

As shown in the following screenshot, Charles is part of the Admin group, so he inherits the SurveyAdmin role. In the case of Bob, he has not been assigned a role yet.

Admin - Members					
Group					
<div><div><div><div><div></div><div>Overview</div></div></div><div><div></div><div>MANAGE</div></div><div><div></div><div>Properties</div></div><div><div></div><div>Members</div></div></div></div>	<div><div><div><div></div><div>Add members</div></div></div><table><tr><th>NAME</th><th>TYPE</th></tr><tr><td><div><div>CH</div><div>Charles</div></div></td><td>User</td></tr></table></div>	NAME	TYPE	<div><div>CH</div><div>Charles</div></div>	User
NAME	TYPE				
<div><div>CH</div><div>Charles</div></div>	User				

#### Note

An alternative approach is for the application to assign roles programmatically, using the Azure AD Graph API. However, this requires the application to obtain write permissions for the customer's AD directory. An application with those permissions could do a lot of mischief — the customer is trusting the app not to mess up their directory. Many customers might be unwilling to grant this level of access.

**Get role claims.** When a user signs in, the application receives the user's assigned role(s) in a claim with type `http://schemas.microsoft.com/ws/2008/06/identity/claims/role`.

A user can have multiple roles, or no role. In your authorization code, don't assume the user has exactly one role claim. Instead, write code that checks whether a particular claim value is present:

C#

Copy

```
if (context.User.HasClaim(ClaimTypes.Role, "Admin")) { ... }
```

## Roles using Azure AD security groups

In this approach, roles are represented as AD security groups. The application assigns permissions to users based on their security group memberships.

Advantages:

- For customers who do not have Azure AD Premium, this approach enables the customer to use security groups to manage role assignments.

Disadvantages:

- Complexity. Because every tenant sends different group claims, the app must keep track of which security groups correspond to which application roles, for each tenant.
- If the customer removes the application from their AD tenant, the security groups are left in their AD directory.

## Implementation

In the application manifest, set the `groupMembershipClaims` property to "SecurityGroup". This is needed to get group membership claims from Azure AD.

JSON

Copy

```
{  
  // ...  
  "groupMembershipClaims": "SecurityGroup",  
}
```

When a new customer signs up, the application instructs the customer to create security groups for the roles needed by the application. The customer then needs to enter the group object IDs into the application. The application stores these in a table that maps group IDs to application roles, per tenant.

#### ⓘ Note

Alternatively, the application could create the groups programmatically, using the Azure AD Graph API. This would be less error prone. However, it requires the application to obtain "read and write all groups" permissions for the customer's AD directory. Many customers might be unwilling to grant this level of access.

When a user signs in:

1. The application receives the user's groups as claims. The value of each claim is the object ID of a group.
2. Azure AD limits the number of groups sent in the token. If the number of groups exceeds this limit, Azure AD sends a special "overage" claim. If that claim is present, the application must query the Azure AD Graph API to get all of the groups to which that user belongs. For details, see [Authorization in Cloud Applications using AD Groups], under the section titled "Groups claim overage".
3. The application looks up the object IDs in its own database, to find the corresponding application roles to assign to the user.
4. The application adds a custom claim value to the user principal that expresses the application role. For example:  
survey\_role = "SurveyAdmin".

Authorization policies should use the custom role claim, not the group claim.

## Roles using an application role manager

With this approach, application roles are not stored in Azure AD at all. Instead, the application stores the role assignments for each user in its own DB — for example, using the **RoleManager** class in ASP.NET Identity.

Advantages:

- The app has full control over the roles and user assignments.

Drawbacks:

- More complex, harder to maintain.
- Cannot use AD security groups to manage role assignments.
- Stores user information in the application database, where it can get out of sync with the tenant's AD directory, as users are added or removed.

[Next](#)