Role-based and resource-based authorization

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

In this article

Role-based authorization
Resource-based authorization

Sample code

Our reference implementation is an ASP.NET Core application. In this article we'll look at two general approaches to authorization, using the authorization APIs provided in ASP.NET Core.

- **Role-based authorization**. Authorizing an action based on the roles assigned to a user. For example, some actions require an administrator role.
- **Resource-based authorization**. Authorizing an action based on a particular resource. For example, every resource has an owner. The owner can delete the resource; other users cannot.

A typical app will employ a mix of both. For example, to delete a resource, the user must be the resource owner *or* an admin.

Role-based authorization

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

Roles apply to *users* of the application. In the Surveys application, a user is either an administrator, creator, or reader.

For a discussion of how to define and manage roles, see Application roles.

Regardless of how you manage the roles, your authorization code will look similar. ASP.NET Core has an abstraction called authorization policies. With this feature, you define authorization policies in code, and then apply those policies to controller actions. The policy is decoupled from the controller.

Create policies

To define a policy, first create a class that implements `IAuthorizationRequirement`. It's easiest to derive from `AuthorizationHandler`. In the `Handle` method, examine the relevant claim(s).

Here is an example from the Tailspin Surveys application:

C# Copy

```csharp
public class SurveyCreatorRequirement : AuthorizationHandler<SurveyCreatorRequirement>, IAutho-
rizationRequirement
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, Survey-
CreatorRequirement requirement)
    {
        if (context.User.HasClaim(ClaimTypes.Role, Roles.SurveyAdmin) ||
            context.User.HasClaim(ClaimTypes.Role, Roles.SurveyCreator))
        {
            context.Succeed(requirement);
        }
        return Task.FromResult(0);
    }
}
```

This class defines the requirement for a user to create a new survey. The user must be in the SurveyAdmin or SurveyCreator role.

In your startup class, define a named policy that includes one or more requirements. If there are multiple requirements, the user must meet *every* requirement to be authorized. The following code defines two policies:

C# Copy

```
services.AddAuthorization(options =>
{
    options.AddPolicy(PolicyNames.RequireSurveyCreator,
        policy =>
        {
            policy.AddRequirements(new SurveyCreatorRequirement());
            policy.RequireAuthenticatedUser(); // Adds DenyAnonymousAuthorizationRequirement
            // By adding the CookieAuthenticationDefaults.AuthenticationScheme, if an authenti-
cated
            // user is not in the appropriate role, they will be redirected to a "forbidden"
page.
            policy.AddAuthenticationSchemes(CookieAuthenticationDefaults.AuthenticationScheme);
        });

    options.AddPolicy(PolicyNames.RequireSurveyAdmin,
        policy =>
        {
            policy.AddRequirements(new SurveyAdminRequirement());
            policy.RequireAuthenticatedUser();
            policy.AddAuthenticationSchemes(CookieAuthenticationDefaults.AuthenticationScheme);
        });
});
```

This code also sets the authentication scheme, which tells ASP.NET which authentication middleware should run if authorization fails. In this case, we specify the cookie authentication middleware, because the cookie authentication middleware can redirect the user to a "Forbidden" page. The location of the Forbidden page is set in the `AccessDeniedPath` option for the cookie middleware; see Configuring the authentication middleware.

Authorize controller actions

Finally, to authorize an action in an MVC controller, set the policy in the `Authorize` attribute:

C# Copy

```
[Authorize(Policy = PolicyNames.RequireSurveyCreator)]
public IActionResult Create()
{
    var survey = new SurveyDTO();
    return View(survey);
}
```

In earlier versions of ASP.NET, you would set the **Roles** property on the attribute:

C# Copy

```
// old way
[Authorize(Roles = "SurveyCreator")]
```

This is still supported in ASP.NET Core, but it has some drawbacks compared with authorization policies:

- It assumes a particular claim type. Policies can check for any claim type. Roles are just a type of claim.
- The role name is hard-coded into the attribute. With policies, the authorization logic is all in one place, making it easier to update or even load from configuration settings.
- Policies enable more complex authorization decisions (for example, age >= 21) that can't be expressed by simple role membership.

Resource-based authorization

Resource based authorization occurs whenever the authorization depends on a specific resource that will be affected by an operation. In the Tailspin Surveys application, every survey has an owner and zero-to-many contributors.

- The owner can read, update, delete, publish, and unpublish the survey.
- The owner can assign contributors to the survey.
- Contributors can read and update the survey.

Note that "owner" and "contributor" are not application roles; they are stored per survey, in the application database. To check whether a user can delete a survey, for example, the app checks whether the user is the owner for that survey.

In ASP.NET Core, implement resource-based authorization by deriving from **AuthorizationHandler** and overriding the **Handle** method.

C# Copy

```csharp
public class SurveyAuthorizationHandler : AuthorizationHandler<OperationAuthorizationRequire-
ment, Survey>
{
    protected override void HandleRequirementAsync(AuthorizationHandlerContext context, Opera-
tionAuthorizationRequirement operation, Survey resource)
    {
    }
}
```

Notice that this class is strongly typed for Survey objects. Register the class for DI on startup:

C# Copy

```csharp
services.AddSingleton<IAuthorizationHandler>(factory =>
{
    return new SurveyAuthorizationHandler();
});
```

To perform authorization checks, use the **IAuthorizationService** interface, which you can inject into your controllers. The following code checks whether a user can read a survey:

C# Copy

```csharp
if (await _authorizationService.AuthorizeAsync(User, survey, Operations.Read) == false)
{
    return StatusCode(403);
}
```

Because we pass in a `Survey` object, this call will invoke the `SurveyAuthorizationHandler`.

In your authorization code, a good approach is to aggregate all of the user's role-based and resource-based permissions, then check the aggregate set against the desired operation. Here is an example from the Surveys app. The application defines several permission types:

- Admin
- Contributor
- Creator
- Owner
- Reader

The application also defines a set of possible operations on surveys:

- Create
- Read
- Update
- Delete
- Publish
- Unpublish

The following code creates a list of permissions for a particular user and survey. Notice that this code looks at both the user's app roles, and the owner/contributor fields in the survey.

C#

Copy

```
public class SurveyAuthorizationHandler : AuthorizationHandler<OperationAuthorizationRequire-
ment, Survey>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, Opera-
tionAuthorizationRequirement requirement, Survey resource)
    {
        var permissions = new List<UserPermissionType>();
        int surveyTenantId = context.User.GetSurveyTenantIdValue();
        int userId = context.User.GetSurveyUserIdValue();
        string user = context.User.GetUserName();

        if (resource.TenantId == surveyTenantId)
        {
            // Admin can do anything, as long as the resource belongs to the admin's tenant.
            if (context.User.HasClaim(ClaimTypes.Role, Roles.SurveyAdmin))
            {
                context.Succeed(requirement);
                return Task.FromResult(0);
            }

            if (context.User.HasClaim(ClaimTypes.Role, Roles.SurveyCreator))
            {
                permissions.Add(UserPermissionType.Creator);
            }
            else
            {
                permissions.Add(UserPermissionType.Reader);
            }

            if (resource.OwnerId == userId)
            {
                permissions.Add(UserPermissionType.Owner);
            }
        }
        if (resource.Contributors != null && resource.Contributors.Any(x => x.UserId ==
userId))
        {
            permissions.Add(UserPermissionType.Contributor);
        }

        if (ValidateUserPermissions[requirement](permissions))
        {
            context.Succeed(requirement);
        }
        return Task.FromResult(0);
        }
    }
```

In a multitenant application, you must ensure that permissions don't "leak" to another tenant's data. In the Surveys app, the Contributor permission is allowed across tenants—you can assign someone from another tenant as a contributor. The other permission types are restricted to resources that belong to that user's tenant. To enforce this requirement, the code checks the tenant ID before granting the permission. (The `TenantId` field as assigned when the survey is created.)

The next step is to check the operation (such as read, update, or delete) against the permissions. The Surveys app implements this step by using a lookup table of functions:

```C#
static readonly Dictionary<OperationAuthorizationRequirement, Func<List<UserPermissionType>, bool>> ValidateUserPermissions
    = new Dictionary<OperationAuthorizationRequirement, Func<List<UserPermissionType>, bool>>

    {
        { Operations.Create, x => x.Contains(UserPermissionType.Creator) },

        { Operations.Read, x => x.Contains(UserPermissionType.Creator) ||
                                x.Contains(UserPermissionType.Reader) ||
                                x.Contains(UserPermissionType.Contributor) ||
                                x.Contains(UserPermissionType.Owner) },

        { Operations.Update, x => x.Contains(UserPermissionType.Contributor) ||
                                x.Contains(UserPermissionType.Owner) },

        { Operations.Delete, x => x.Contains(UserPermissionType.Owner) },

        { Operations.Publish, x => x.Contains(UserPermissionType.Owner) },

        { Operations.UnPublish, x => x.Contains(UserPermissionType.Owner) }
    };
```

Next