7 Types of Authorization in ASP.NET Core Web API

Aram Tchekrekjian

@AramT87

Authorization

Authorization is the process of verifying the permissions and access of an authenticated user or client

It ensures your API is secure and that users only access what they're permitted to.

This document will introduce you to the 7 types of authorization in ASP.NET Core Web API

@AramT87

1

Simple Authorization

Simple Authorization

This is the default and the most basic form of authorization

Using only the [Authorize] attribute on any controller would enable the simple authorization on it.

This means that any authenticated user is able to access that endpoint providing their access token

Can be used for generic purposes like (e.g., access to their profile screen)

Simple Authorization

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer();

builder.Services.AddAuthorization(); // basic setup

var app = builder.Build();

app.UseAuthentication();
app.UseAuthorization();

// Simple protected endpoint
app.MapGet("/simple", [Authorize] () => "You are authorized!");

app.Run();
```

@AramT87

2

Role-based Authorization

RBAC

Role-based Authorization

Grants access based on user roles (e.g., Admin, Manager, User).

Roles are usually stored in JWT claims, Identity DB, or external providers.

You can use it Restrict admin dashboards or Limit critical API actions (e.g., deleting records).

Role-based Authorization

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer();

builder.Services.AddAuthorization(); //no need to add roles here.
// They will be included within the JWT,
// or defined by the Identity provider

var app = builder.Build();

app.UseAuthentication();
app.UseAuthorization();

// Role-based protected endpoint
app.MapGet("/role-only", [Authorize(Roles = "Admin")] () =>
    "Hello Admin!");

app.Run();
```

@AramT87

3

Policy-based Authorization

PBAC

Policy-based Authorization

Policies are named sets of requirements.

You can combine multiple rules (roles, claims, assertions).

Use this when access rules go beyond a simple role check (e.g., "Only Admins OR Managers").

@AramT87

Policy-based Authorization

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer();

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("RequireAdmin", policy =>
        policy.RequireRole("Admin"));
});

var app = builder.Build();

app.UseAuthentication();
app.UseAuthorization();

// Policy-based protected endpoint
app.MapGet("/policy-only", [Authorize(Policy = "RequireAdmin")] () =>
    "You passed the Admin policy!");

app.Run();
```

@AramT87

4

Claims-based Authorization

CBAC

Claims-based Authorization

Authorizes users based on claims in their identity (e.g., EmployeeId, Department).

Use this when permissions depend on attributes of the user, like employee records, region, or subscription tier.

@AramT87

Claims-based Authorization

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer();

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("EmployeeIdPolicy", policy =>
        policy.RequireClaim("EmployeeId"));
});

var app = builder.Build();

app.UseAuthentication();
app.UseAuthorization();

// Claim-based protected endpoint
app.MapGet("/claims-only",
    [Authorize(Policy = "EmployeeIdPolicy")] (HttpContext ctx) =>
    $"Welcome employee {ctx.User.FindFirst("EmployeeId")?.Value}");

app.Run();
```

@AramT87

5

Custom Requirement Authorization

CRA

Custom Req. Authorization

Here you create your own requirement and handler logic

The Requirement will implement the IRequirementHandler

And then implement a custom authorization Handler for that requirement

This type can be useful when built-in checks (roles/claims) are not enough, like “User must be 18 years old” or “User must own > 5 projects.”

@AramT87

Custom Req. Authorization

```cs
public class MinimumAgeRequirement : IAuthorizationRequirement
{
    public int Age { get; }
    public MinimumAgeRequirement(int age) => Age = age;
}

public class MinimumAgeHandler : AuthorizationHandler<MinimumAgeRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        MinimumAgeRequirement requirement)
    {
        var claim = context.User.FindFirst("DateOfBirth");
        if (claim != null && DateTime.TryParse(claim.Value, out var dob))
        {
            var age = DateTime.Today.Year - dob.Year;
            if (dob > DateTime.Today.AddYears(-age)) age--;
            if (age >= requirement.Age)
                context.Succeed(requirement);
        }
        return Task.CompletedTask;
    }
}

// Program.cs
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("MinimumAge", policy =>
        policy.Requirements.Add(new MinimumAgeRequirement(18)));
});

builder.Services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();

// requirement-based protected endpoint
app.MapGet("/min-age", [Authorize(Policy = "MinimumAge")] () =>
    "You meet the minimum age requirement!");
```

@AramT87

6

Endpoint-specific Authorization

ESA

Endpoint-specific Authorization

Instead of using attributes, you apply authorization directly on the endpoint in the pipeline.

Use this when dynamically adding routes or applying policies conditionally.

Useful in APIs that build routes at runtime.

@AramT87

Endpoint-specific Authorization

```
public class MinimumAgeRequirement : IAuthorizationRequirement
{
    public int Age { get; }
    public MinimumAgeRequirement(int age) => Age = age;
}

public class MinimumAgeHandler : AuthorizationHandler<MinimumAgeRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        MinimumAgeRequirement requirement)
    {
        var claim = context.User.FindFirst("DateOfBirth");
        if (claim != null && DateTime.TryParse(claim.Value, out var dob))
        {
            var age = DateTime.Today.Year - dob.Year;
            if (dob > DateTime.Today.AddYears(-age)) age--;
            if (age >= requirement.Age)
                context.Succeed(requirement);
        }
        return Task.CompletedTask;
    }
}

// Program.cs
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("MinimumAge", policy =>
        policy.Requirements.Add(new MinimumAgeRequirement(18)));
});

builder.Services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();

// requirement-based protected endpoint
app.MapGet("/min-age", [Authorize(Policy = "MinimumAge")] () =>
    "You meet the minimum age requirement!");
```

@AramT87

7

Resource-specific Authorization

RSA

Resource-specific Authorization

Authorizes access to a specific resource.

You compare the current user with the resource they’re trying to access.

This is useful for fine-grained control: e.g., user can only edit their own document, not someone else’s.

@AramT87

Resource-specific Authorization

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("DocumentAccessPolicy", policy =>
        policy.RequireAssertion(ctx =>
        {
            if (ctx.Resource is Document doc)
            {
                // Allow if user has "Manager" role
                if (ctx.User.IsInRole("Manager"))
                    return true;

                // Allow if user owns the document
                var userId = ctx.User.FindFirst("UserId")?.Value;
                return userId == doc.OwnerId;
            }
            return false;
        }));
});

var documents = new List<Document>
{
    new("doc1", "user1", "Document 1 content"),
    new("doc2", "user2", "Document 2 content")
};

app.MapGet("/documents/{id}",
    async (string id, IAuthorizationService auth, HttpContext ctx) =>
{
    var document = documents.FirstOrDefault(d => d.Id == id);
    if (document is null)
        return Results.NotFound();

    var result = await auth.AuthorizeAsync(
        ctx.User, document, "DocumentAccessPolicy");

    return result.Succeeded ?
            Results.Ok(document.Content) :
            Results.Forbid();
});

public record Document(string Id, string OwnerId, string Content);
```

@AramT87

Summary

Here are the 7 types of Authorization in ASP.NET Core Web API

1. Simple
2. Role-based
3. Policy-based
4. Claims-based
5. Custom-requirement
6. Endpoint-specific
7. Resource-specific

@AramT87

Found this useful?

Consider Reposting

Aram Tchekrekjian

<Coding♪onata />

@AramT87

Thank You

Follow me for more content

Aram Tchekrekjian

Microsoft® MVP Most Valuable Professional

AramT87

Get Free Tips and Tutorials in .NET and C#

Join 1000+ Readers

CodingSonata.com/newsletters

<Coding♪onata />