Authentication and Authorization

Authentication (AuthN) and Authorization (AuthZ) are two essential components of security in any application or system. They work together to ensure that users have the appropriate permissions to access specific resources and perform certain actions. Here's a brief explanation of AuthN and AuthZ:

Authentication

Is the process of verifying the identity of a user or entity attempting to access a system or application.
Once a user is successfully authenticated, they receive an authentication token (e.g., a JSON Web Token - JWT) that proves their identity for a certain period of time.

Authorization

Is the process of determining whether an authenticated user has the necessary permissions to access specific resources or perform certain actions within a system.
It involves evaluating the user's roles, groups, or claims, which are typically stored in the authentication token obtained during authentication.

How both work together

Authentication comes first in the security process. Before any authorization checks can be performed, the user's identity must be verified through authentication.

After successful authentication, the application can proceed with authorization checks to determine the user's access rights and privileges.

Considerations

AuthN

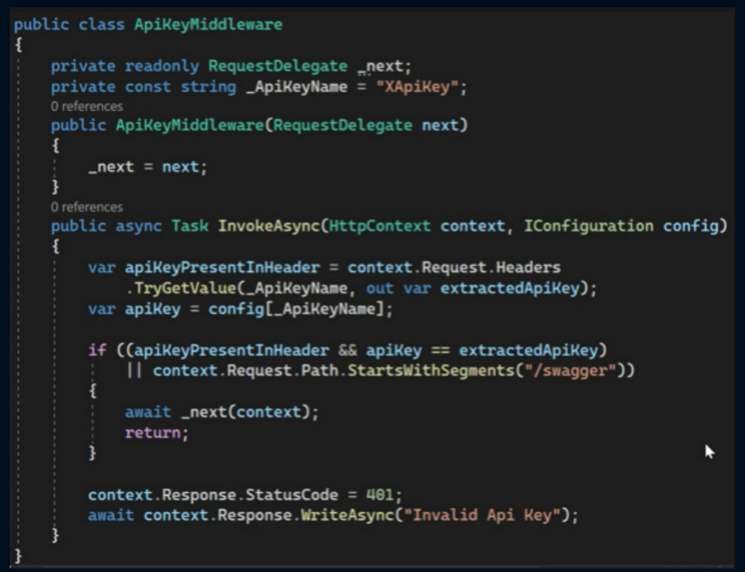
- Identity Cookies
- ASP.NET Core Identity
- Identity Provider

AuthZ

- ASP.Net Core Authorization

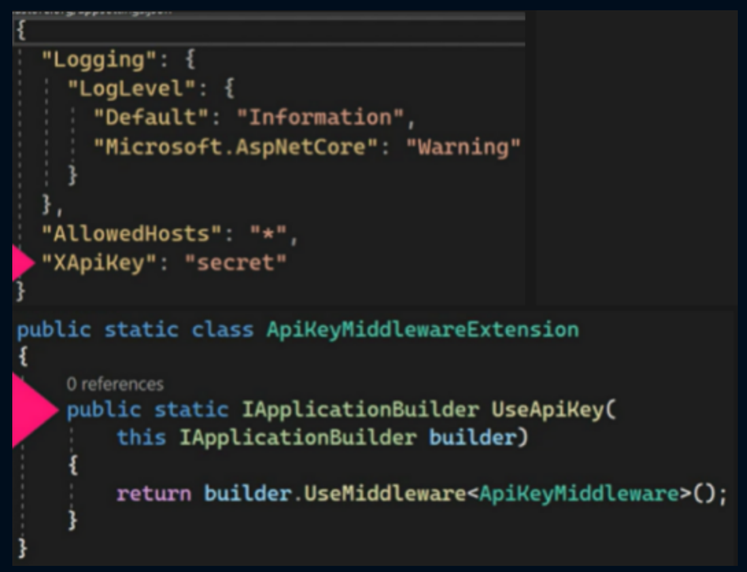
API Key Protection Middleware

Is a security middleware that provides a way to authenticate and authorize API requests using API keys. API keys are short, unique, and secret tokens that clients must include in their requests to access protected endpoints. This middleware is useful when you want to restrict access to certain parts of your API and control who can make requests.



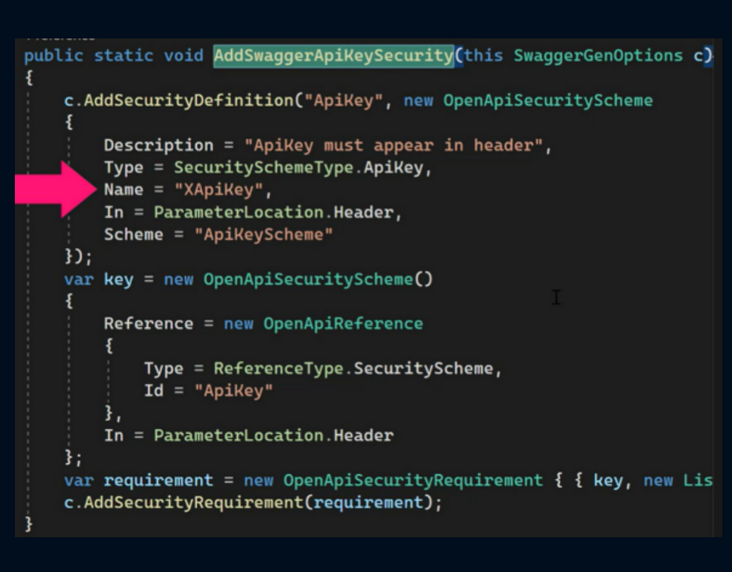
```
public class ApiKeyMiddleware
{
    private readonly RequestDelegate _next;
    private const string _ApiKeyName = "XApiKey";
    0 references
    public ApiKeyMiddleware(RequestDelegate next)
    {
        _next = next;
    }
    0 references
    public async Task InvokeAsync(HttpContext context, IConfiguration config)
    {
        var apiKeyPresentInHeader = context.Request.Headers
            .TryGetValue(_ApiKeyName, out var extractedApiKey);
        var apiKey = config[_ApiKeyName];

        if ((apiKeyPresentInHeader && apiKey == extractedApiKey)
            || context.Request.Path.StartsWithSegments("/swagger"))
        {
            await _next(context);
            return;
        }

        context.Response.StatusCode = 401;
        await context.Response.WriteAsync("Invalid Api Key");
    }
}
```

Custom API Key Middleware



```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "XApiKey": "secret"
}

public static class ApiKeyMiddlewareExtension
{
    0 references
    public static IApplicationBuilder UseApiKey(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<ApiKeyMiddleware>();
    }
}
```

Custom API Key Middleware Setup

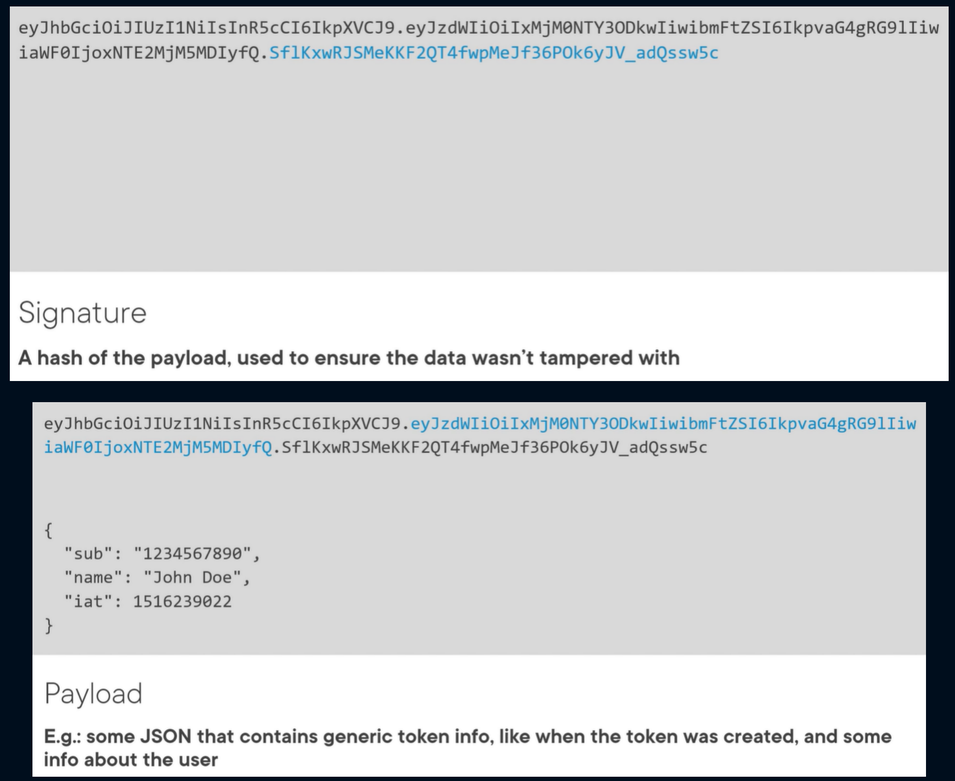


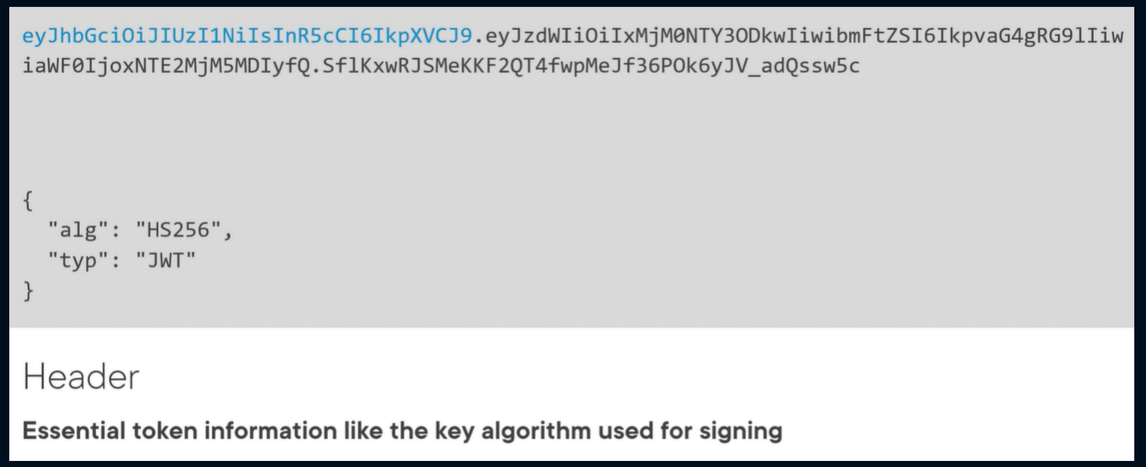
```
public static void AddSwaggerApiKeySecurity(this SwaggerGenOptions c)
{
    c.AddSecurityDefinition("ApiKey", new OpenApiSecurityScheme
    {
        Description = "ApiKey must appear in header",
        Type = SecuritySchemeType.ApiKey,
        Name = "XApiKey",
        In = ParameterLocation.Header,
        Scheme = "ApiKeyScheme"
    });
    var key = new OpenApiSecurityScheme()
    {
        Reference = new OpenApiReference
        {
            Type = ReferenceType.SecurityScheme,
            Id = "ApiKey"
        },
        In = ParameterLocation.Header
    };
    var requirement = new OpenApiSecurityRequirement { { key, new Lis
    c.AddSecurityRequirement(requirement);
}
```

Custom API Key Swagger Doc

JWT

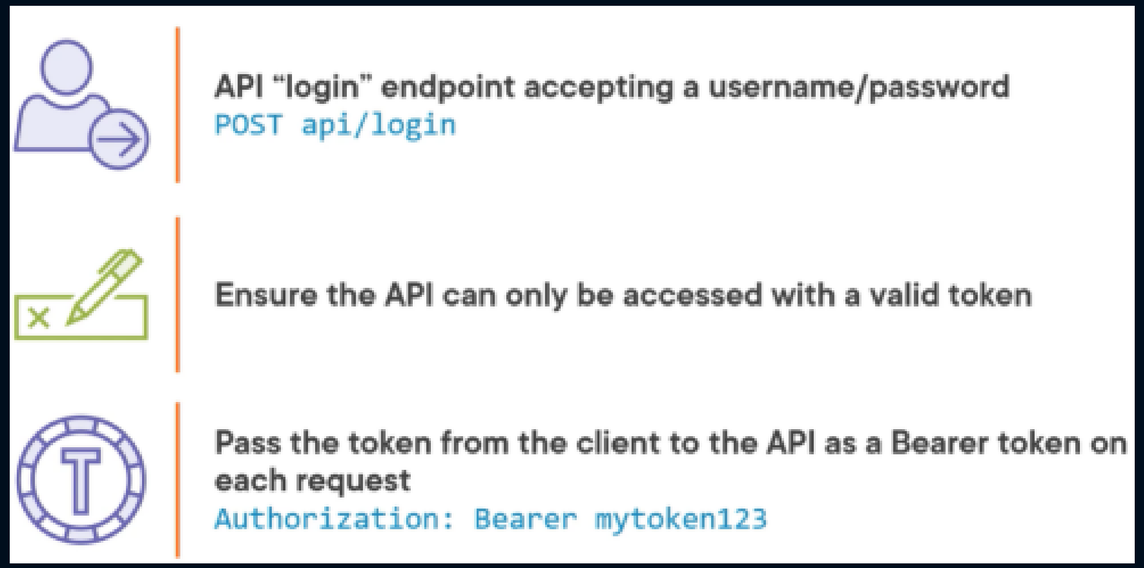
JWT stands for ”JSON Web Token,” and it is a compact, self-contained, and secure way to represent and transmit information between two parties. JWTs are commonly used for authentication and authorization in web applications and APIs: is a credential, can be used as an OAuth Bearer token, should be kept as a secret and must Use TLS (Transport Layer Security)





Token Based security

Send a token on each request, this token represents consent and you should validate the token at level of the API



Token Security implementation

```
[HttpPost("authenticate")]
3 references
public ActionResult<string> Authenticate(
    AuthenticationRequestBody authenticationRequestBody)
{
    // Step 1: validate the username/password
    var user = ValidateUserCredentials(
        authenticationRequestBody.UserName,
        authenticationRequestBody.Password);

    if (user == null)
    {
        return Unauthorized();
    }

    // Step 2: create a token
    var securityKey = new SymmetricSecurityKey(
        Encoding.ASCII.GetBytes(_configuration["Authentication:SecretForKey"]));
    var signingCredentials = new SigningCredentials(
        securityKey, SecurityAlgorithms.HmacSha256);

    var claimsForToken = new List<Claim>();
    claimsForToken.Add(new Claim("sub", user.UserId.ToString()));
    claimsForToken.Add(new Claim("given_name", user.FirstName));
    claimsForToken.Add(new Claim("family_name", user.LastName));
    claimsForToken.Add(new Claim("city", user.City));

    var jwtSecurityToken = new JwtSecurityToken(
        _configuration["Authentication:Issuer"],
        _configuration["Authentication:Audience"],
        claimsForToken,
        DateTime.UtcNow,
        DateTime.UtcNow.AddHours(1),
        signingCredentials);

    var tokenToReturn = new JwtSecurityTokenHandler()
       .WriteToken(jwtSecurityToken);

    return Ok(tokenToReturn);
}
```

Authentication Controller

Token Based security

Send a token on each request, this token represents consent and you should validate the token at level of the API

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "CityInfo.API.Controllers": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.EntityFrameworkCore.Database.Command": "Information"
    }
  },
  "ConnectionStrings": {
    "CityInfoDBConnectionString": "Data Source=CityInfo.db"
  },
  "Authentication": {
    "SecretForKey": "thisisthesecretforgeneratingakey(mustbeatleast32bitlong)",
    "Issuer": "https://localhost:7169",
    "Audience": "cityinfoapi"
  }
}
```

Token App Settings

```
er.Services.AddAuthentication("Bearer")
AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new()
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = builder.Configuration["Authentication:Issuer"],
            ValidAudience = builder.Configuration["Authentication:Audience"],
            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.ASCII.GetBytes(builder.Configuration["Authentication:SecretForKey"]))
        };
    }
);

[Route("api/cities/{cityId}/pointsofinterest")]
[Authorize]
[ApiController]
3 references
public class PointsOfInterestController : ControllerBase
{
```

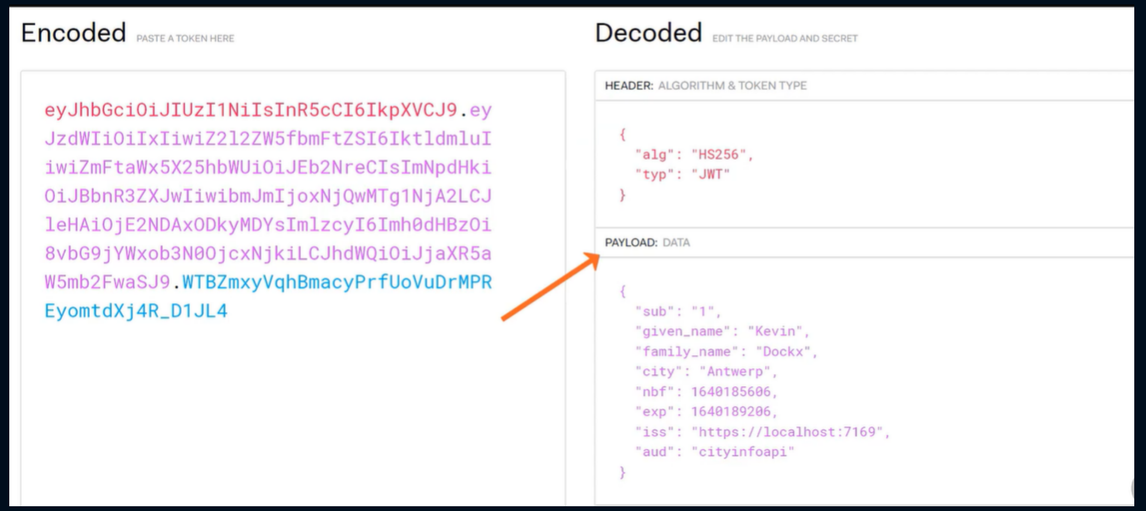
Token Setup and use

Token Based security

Send a token on each request, this token represents consent and you should validate the token at level of the API

```
1 reference
private CityInfoUser ValidateUserCredentials(string? userName, string? password)
{
    // we don't have a user DB or table.  If you have, check the passed-through
    // username/password against what's stored in the database.
    //
    // For demo purposes, we assume the credentials are valid

    // return a new CityInfoUser (values would normally come from your user DB/table)
    return new CityInfoUser(
        1,
        userName ?? "",
        "Kevin",
        "Dockx",
        "New York City");

}
```

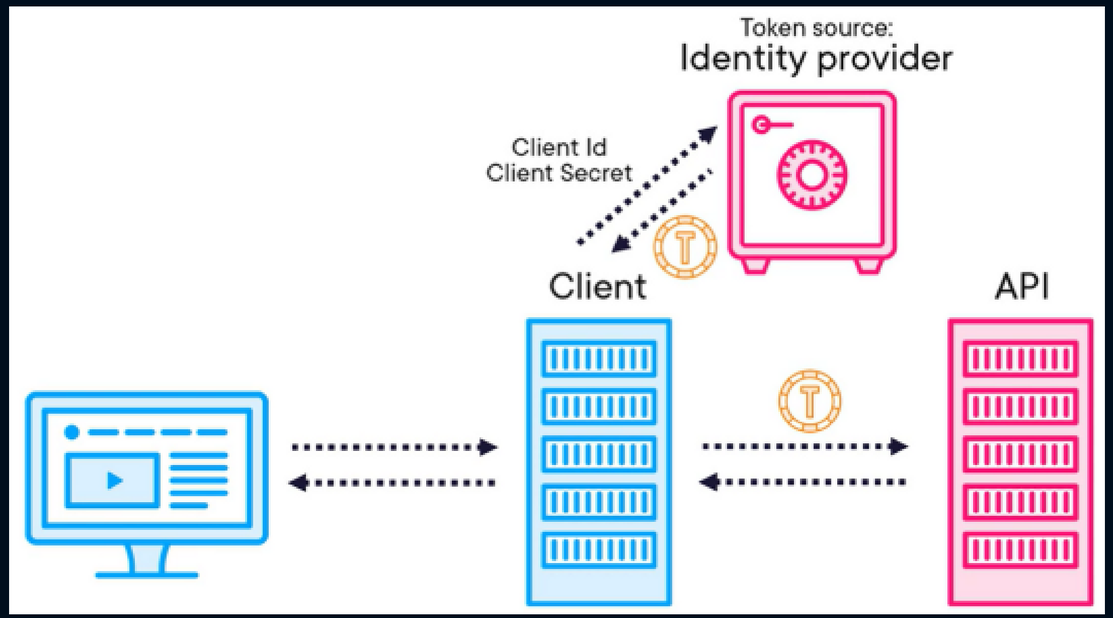
Validate Credentials



Token Decoded

Identity Provider

Is a service that manages and provides user authentication and identity information to other systems and applications. It acts as a centralized authority for verifying the identity of users and issuing authentication tokens, such as JSON Web Tokens (JWT) or Security Assertion Markup Language (SAML) tokens, which can be used by various applications to authenticate users.



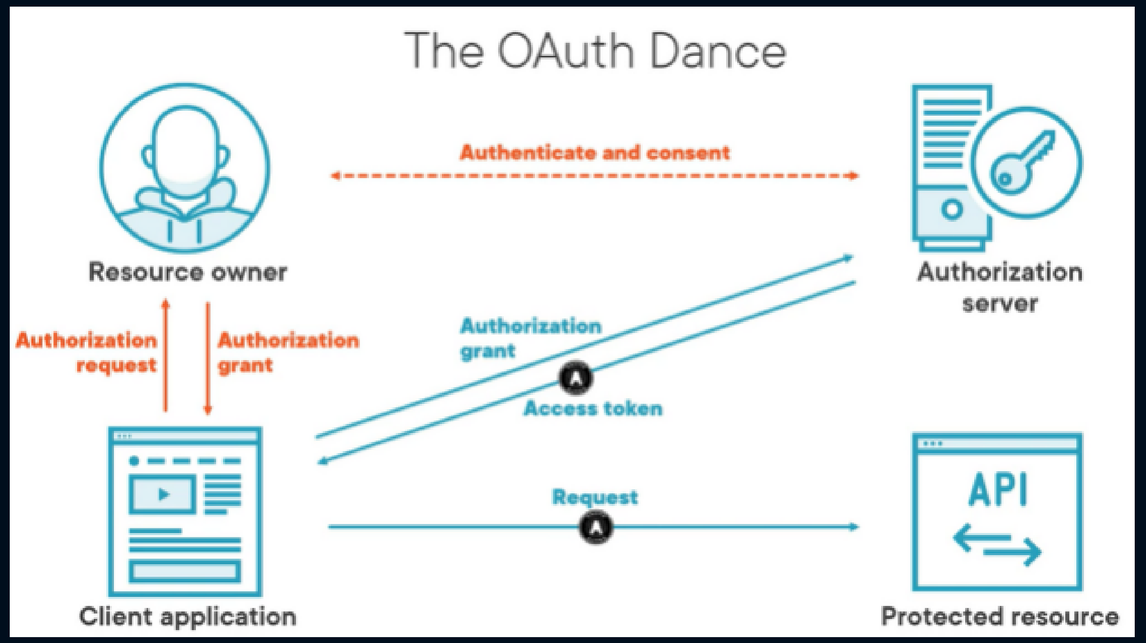
Event declaration and use

You can access the event's delegate and invoke it direclty

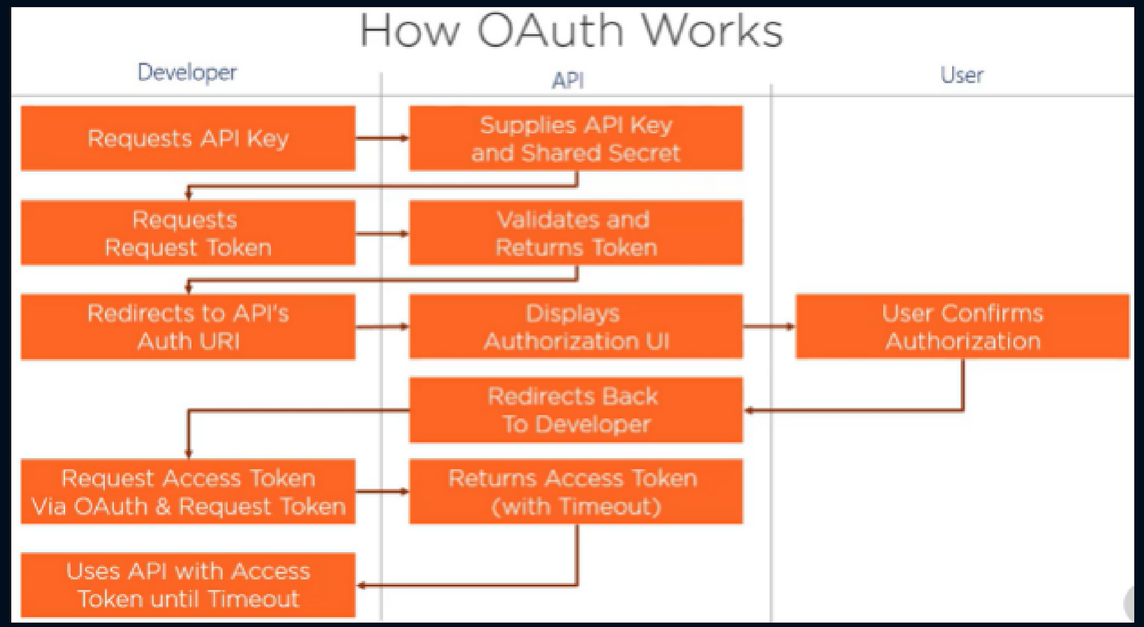
Oauth

OAuth 2.0 is an authorization framework that allows applications to obtain access to resources on behalf of a user without requiring the user's credentials to be shared with the application.

The primary focus of OAuth 2.0 is on securing API access and providing authorization tokens (e.g., access tokens) that clients can use to access protected resources.



Oauth Workflow

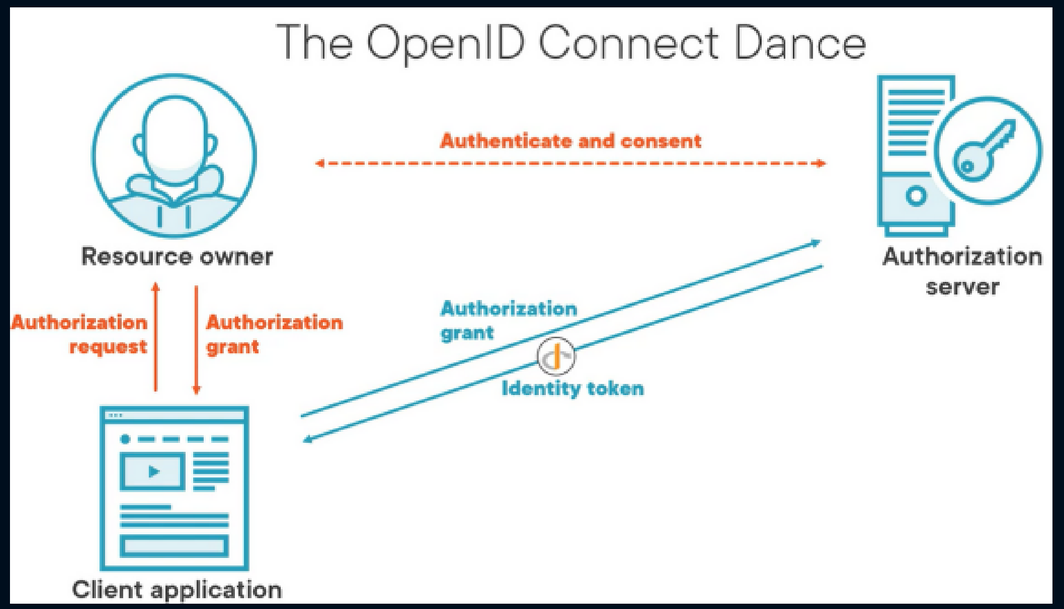


How it works

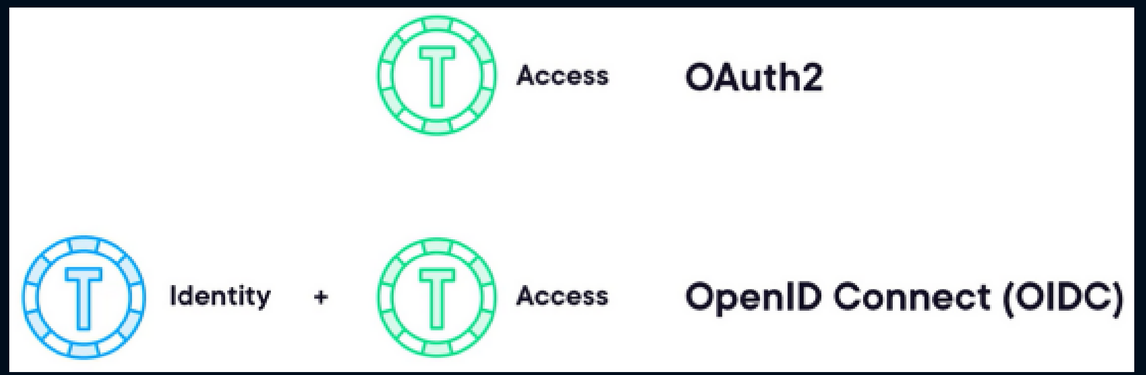
OpenID

OpenID Connect is an authentication layer built on top of OAuth 2.0, specifically designed for user authentication.

It extends OAuth 2.0 by adding an ID Token, which contains identity information about the authenticated user, such as user ID, name, email, and other user attributes. It enables Single Sign-On (SSO) scenarios, allowing users to authenticate once and access multiple applications without re-entering credentials. You can say it focuses on both AuthN and AuthZ



OpenId

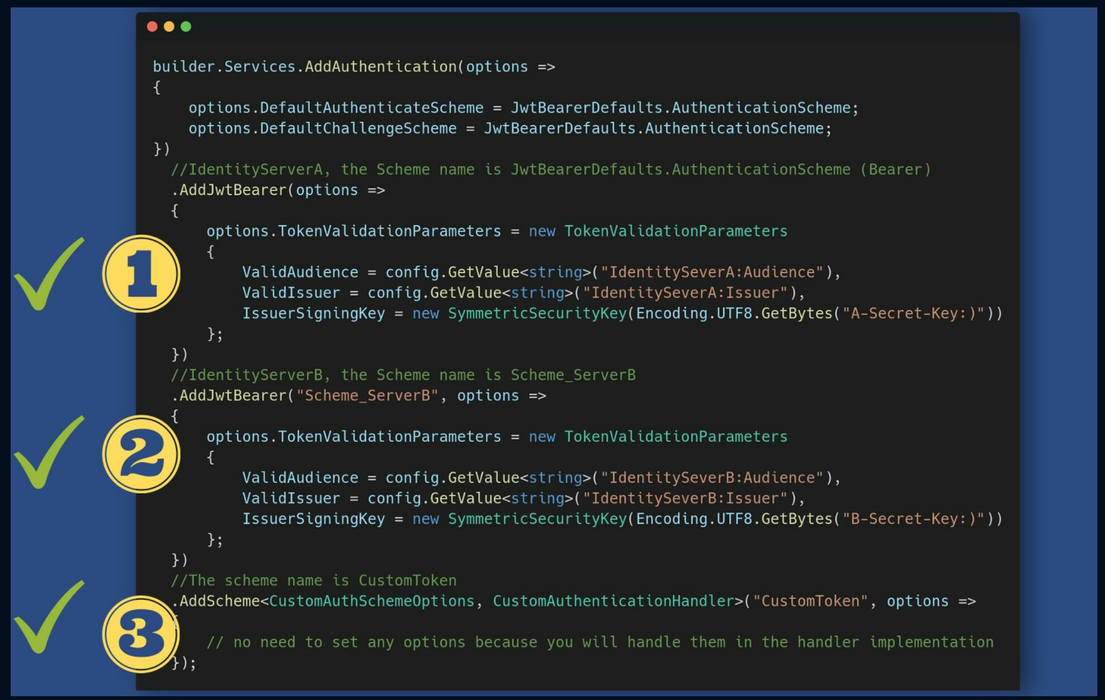


Oauth vs OpenId

Multiple Authentication schemas

The authentication scheme can select which authentication handler is responsible for generating the correct set of claims.

An authentication scheme is a name that corresponds to: an authentication handler, options for configuring that specific instance of the handler and the registered authentication handlers and their configuration options which are called ”schemes”. To implement it make sure to take a look at the link of the image

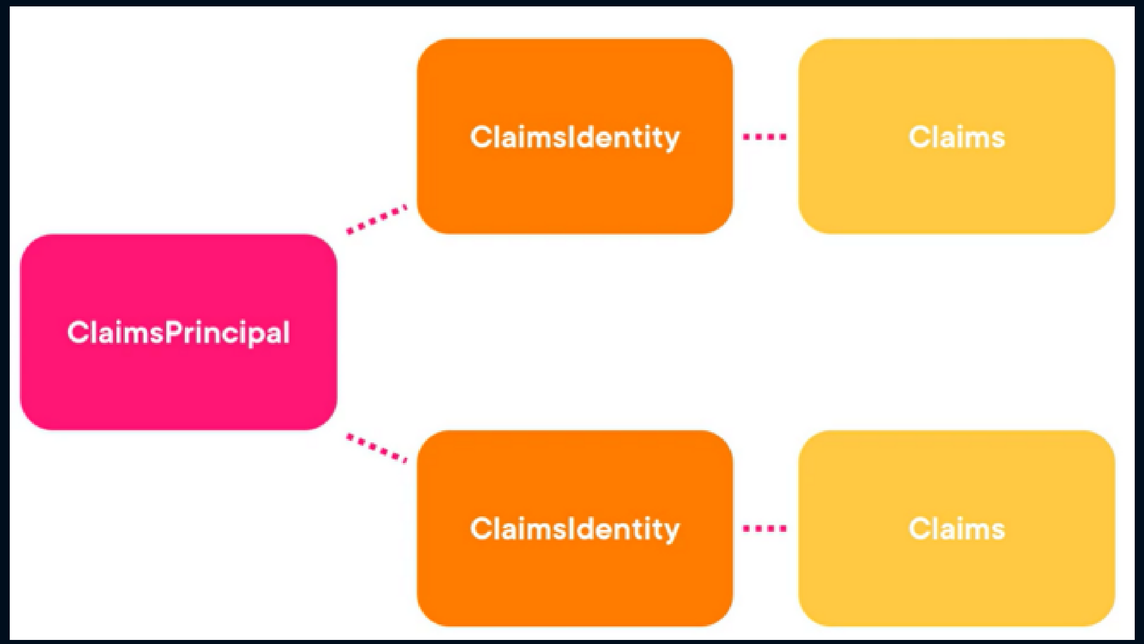


```
builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
  //IdentityServerA, the Scheme name is JwtBearerDefaults.AuthenticationScheme (Bearer)
  .AddJwtBearer(options =>
  {
      options.TokenValidationParameters = new TokenValidationParameters
      {
          ValidAudience = config.GetValue<string>("IdentitySeverA:Audience"),
          ValidIssuer = config.GetValue<string>("IdentitySeverA:Issuer"),
          IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("A-Secret-Key:)"))
      };
  })
  //IdentityServerB, the Scheme name is Scheme_ServerB
  .AddJwtBearer("Scheme_ServerB", options =>
  {
      options.TokenValidationParameters = new TokenValidationParameters
      {
          ValidAudience = config.GetValue<string>("IdentitySeverB:Audience"),
          ValidIssuer = config.GetValue<string>("IdentitySeverB:Issuer"),
          IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("B-Secret-Key:)"))
      };
  })
  //The scheme name is CustomToken
  .AddScheme<CustomAuthSchemeOptions, CustomAuthenticationHandler>("CustomToken", options =>

      // no need to set any options because you will handle them in the handler implementation
  });
```

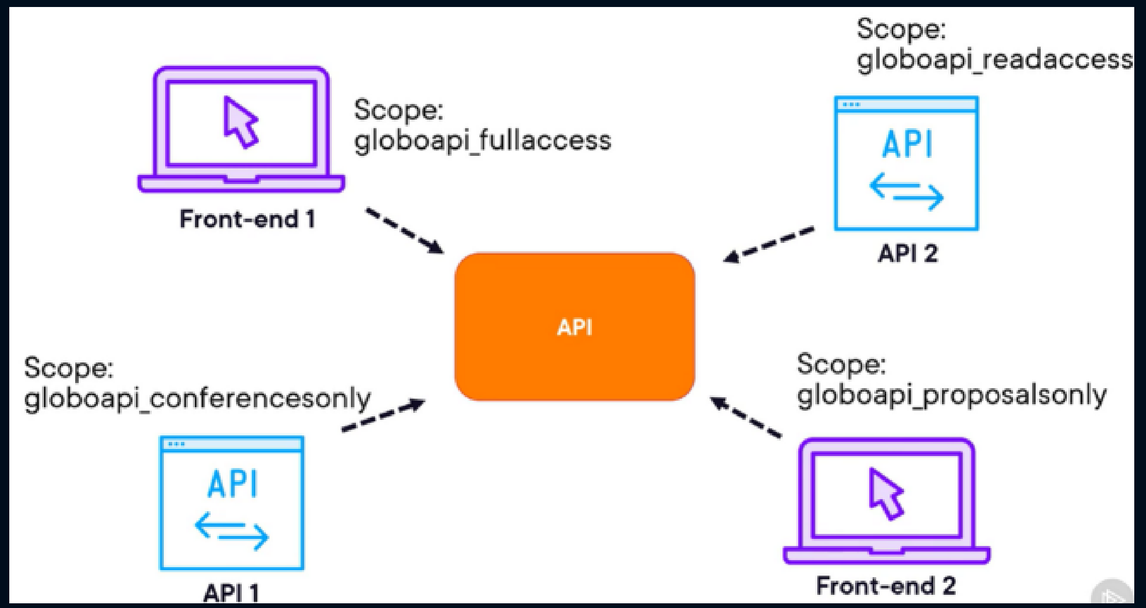
Authentication Schemes

Claims

Claims, in the context of authentication and authorization, are statements about a user or entity that provide specific information or attributes related to their identity. They are typically represented as key-value pairs and are used in security tokens like JSON Web Tokens (JWTs) or Security Assertion Markup Language (SAML) tokens.



Claims



Authorization using Scope

Authorization Policies and IAuthorizationService

Authorization policies are sets of one or more requirements that must be satisfied for a user to be authorized.
Policies help group multiple requirements together, making it easier to manage and apply authorization logic in a modular and reusable manner.
To apply authorization, you can use attributes like [Authorize] or [Authorize(Policy = "PolicyName")] on controllers or actions to require users to meet specific authorization requirements.
Additionally, you can perform programmatic authorization checks in your code by using the IAuthorizationService to check whether a user meets a particular requirement.

```
builder.Services.AddAuthorization(o =>
{
    o.AddPolicy("fullaccess", p =>
        p.RequireClaim(JwtClaimTypes.Scope, "globoapi_full"));
    o.AddPolicy("isadmin", p =>
        p.RequireClaim(JwtClaimTypes.Role, "admin"));
    o.AddPolicy("isemployee", p =>
        p.RequireClaim("employeeno"));

    o.DefaultPolicy = new AuthorizationPolicyBuilder()
        .RequireClaim(JwtClaimTypes.Role, "contributor")
        .RequireAuthenticatedUser()
        .Build();
});
```

Setup Authorization

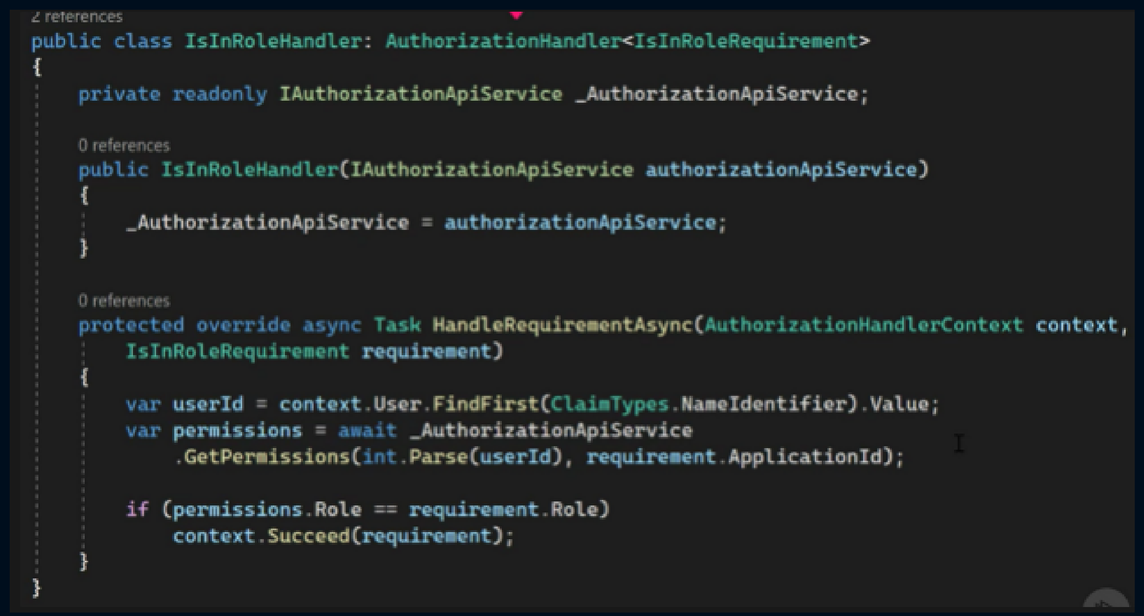
```
public ConferenceController(IConferenceRepository repo,
    IAuthorizationService authService)
{
    _Repo = repo;
    _AuthService = authService;
}

[HttpGet("{id}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
1 reference
public async Task<IActionResult> GetById(int id)
{
    var result = await _AuthService
        .AuthorizeAsync(User, "isadmin");
    if (result.Succeeded)
    {

    }
    var conference = _Repo.GetById(id);
    if (conference == null)
    {
        return NotFound();
    }
    return Ok(conference);
}
```

Inject Authorization

Requirements and Handlers

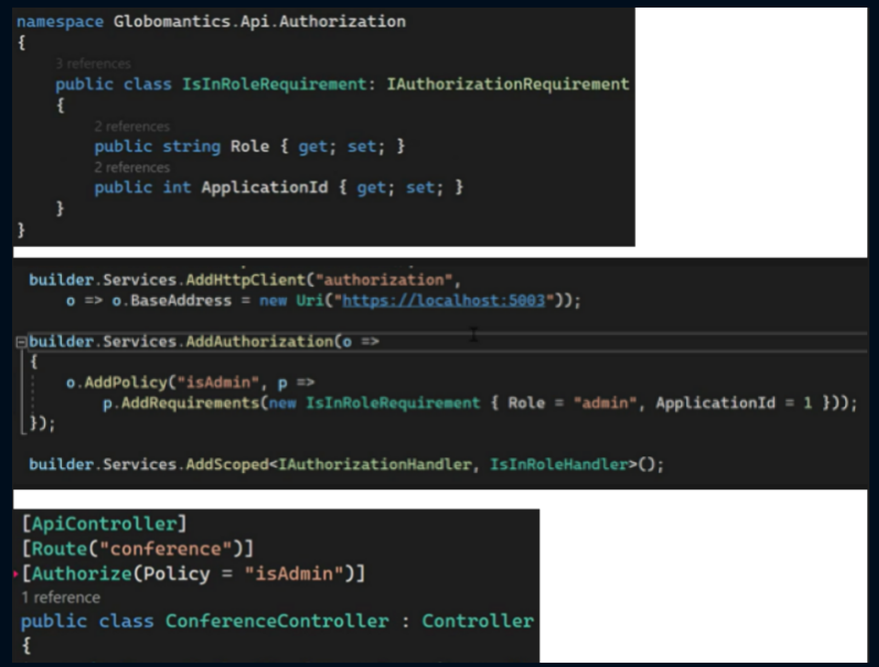
Requirements represent the conditions that must be met for a user to be authorized to perform an action or access a resource. A requirement is a C# class that implements the IAuthorizationRequirement interface, which typically contains some properties or data representing the criteria for authorization.

Handlers are responsible for evaluating the requirements and determining whether a user meets those requirements. A handler is a C# class that implements the AuthorizationHandler<TRequirement> abstract class, where TRequirement is the specific requirement type being handled.



```
2 references
public class IsInRoleHandler: AuthorizationHandler<IsInRoleRequirement>
{
    private readonly IAuthorizationApiService _AuthorizationApiService;

    0 references
    public IsInRoleHandler(IAuthorizationApiService authorizationApiService)
    {
        _AuthorizationApiService = authorizationApiService;
    }

    0 references
    protected override async Task HandleRequirementAsync(AuthorizationHandlerContext context,
        IsInRoleRequirement requirement)
    {
        var userId = context.User.FindFirst(ClaimTypes.NameIdentifier).Value;
        var permissions = await _AuthorizationApiService
            .GetPermissions(int.Parse(userId), requirement.ApplicationId);

        if (permissions.Role == requirement.Role)
            context.Succeed(requirement);
    }
}
```

Authorization Handler



```
namespace Globomantics.Api.Authorization
{
    3 references
    public class IsInRoleRequirement: IAuthorizationRequirement
    {
        2 references
        public string Role { get; set; }
        2 references
        public int ApplicationId { get; set; }
    }
}
```

```
builder.Services.AddHttpClient("authorization",
    o => o.BaseAddress = new Uri("https://localhost:5003"));

builder.Services.AddAuthorization(o =>
{
    o.AddPolicy("isAdmin", p =>
        p.AddRequirements(new IsInRoleRequirement { Role = "admin", ApplicationId = 1 }));
});

builder.Services.AddScoped<IAuthorizationHandler, IsInRoleHandler>();
```

```
[ApiController]
[Route("conference")]
[Authorize(Policy = "isAdmin")]
1 reference
public class ConferenceController : Controller
{
```

Setup

Demo

We will check the following:

Check Authentication and Authorization project

You can check the following Repository(ies) for some examples:

ASP NET Core 6 Web API

Task

Exercise/ Homework

This is a continuation of the previous exercise.

- Add authentication for your api
- Add documentation to your api about the authentication mechanism