Working with Identity in ASP.NET

Pretty much all websites these days have a login function. Even if they work when browsing anonymously, there is usually an option to become a member or something similar. This means that these websites have some concept of identity to tell their visitors apart. In other words – if you are tasked with building a website, it is likely that you will need to deal with identities as well. The thing is, identity can be hard to get right and the consequences of getting it wrong can be less than fun. In this module, we will dive into the basics of identity in ASP.NET 5.

We will cover the following topics in this module:

- Understanding authentication concepts
- Understanding authorization concepts
- The role of middleware in ASP.NET and identity
- OAuth and OpenID Connect basics
- Integrating with Azure Active Directory
- Working with federated identity

Technical requirements

This module includes short code snippets to demonstrate the concepts that are explained.

The following software is required to make it work:

- Visual Studio 2019
- Some of the samples require you to have an Azure Active Directory (AAD) tenant. If you don't have one already, you can either create one by going to the Azure portal (https://portal.azure.com) and sign up for a free account or even better, sign up for a free Office 365 Developer account, which includes the paid version of AAD as well as the Office 365 services: https://docs.microsoft.com/en-us/office/developer-program/microsoft-365-developer-program.
- The section on federated identity uses AAD B2C. This is a special version of AAD that you need to create separately: https://docs.microsoft.com/en-us/azure/active-directoryb2c/tutorial-create-tenant.

For lab purposes, all of the samples in this module are possible to test free of charge, but regional-specific requirements might need the use of a credit card for verification purposes.

Understanding authentication concepts

Most of us have an understanding of what we mean when we say ''identity'' in everyday speech. In .NET, and coding in general, we need to be more specific before letting a user into our apps. Identity in this context encompasses multiple concepts with different actions and mechanisms along the way to establish who the user is and what they are allowed to do in our systems.

The first piece of the identity puzzle is authentication. In documentation and literature, you will often find this shortened to AuthN. Authentication is about answering the question of who you are. Analogous to the real world, this carries different levels of trust, depending on how this question is answered.

If you met someone you didn't know at a party and asked them what their name was, you would probably be happy with whatever they answered without further verification. You would, however, most likely not be happy with implementing a login function on a website where the user could get away with typing only a username to log in.

A real-life example would be asking someone to provide identity papers – this could be a national ID card, driver's license, passport, or something similar. On websites, the most common method is providing a combination of a username and a secret only you know (for instance, a password).

The simplest form of implementing this in a web app is to use basic authentication, which is part of the HTTP specification. This works by the client side appending a header to the HTTP request with credentials encoded as a Base64 value. In a console app, it would look like this:

```
static void Main(string[] args) {
  var username = "andreas";
  var password = "password";
  var byteEncoding = System.Text.UTF8Encoding.UTF8.GetBytes($"{username}:{password}");
  var credentials = Convert.ToBase64String(byteEncoding);
  Console.WriteLine(credentials);
  HttpClient client = new HttpClient();
  client.DefaultRequestHeaders.Authorization =
       new AuthenticationHeaderValue("Basic"', credentials);
  var response = client.GetAsync("https://localhost:5001");
}
```

The credentials will always be YW5kcmVhczpwYXNzd29yZA== with no random element, so the main benefit of transferring it this way is for encoding purposes. Let's have a quick look at what Base64 is before moving on.

Base64 encoding

All of us are familiar with Base10 (usually called decimal) as this is what we use when doing ordinary arithmetic – we use 0–9 for representing numbers. In computing, Base16 is also often used under the name hexadecimal. Since the numbers only go up to 9, we use letters in addition, so A=10, B=11, and so on up to F=15. Base64 takes this even further by using A-Z, a-z, 0-9, and the + and / characters, with = as a special padding character (to ensure a string is always of a predictable length).

We will not dive into the algorithm of how to convert characters, but as demonstrated in the previous snippet, it will turn something that is human-readable into something that, while still technically readable, is hard to interpret just by looking at it. The main benefit of encoding the data this way is that both plain text and binary data can be transferred without corruption even if you use non-printable or non-readable characters. The HTTP protocol does not, by itself, account for all characters, so for a password with special

characters, it might not be correctly interpreted on the server side if you transfer it without encoding.

Base64 is not a form of encryption, so you cannot trust it for secrets as such and it can be considered plain text even though you, as a human, are not able to decode it on the fly. This also means that using basic auth without HTTPS is an insecure authentication mechanism. Using TLS/SSL to secure the transport greatly improves on this, but it still relies on sending the password over the wire.

With this in the back of our minds, it follows that we are able to decode the Base64 string on the other end of the transmission, and the corresponding server part would look like this:

```
public String Get() {
  var authHeader = HttpContext.Request.
  Headers["Authorization"];
  var base64Creds = AuthenticationHeaderValue.Parse(authHeader).Parameter;
  var byteEncoded = System.Convert.FromBase64String(base64Creds);
  var credentials = System.Text.Encoding.UTF8.GetString(byteEncoded);
  if (credentials == "andreas:password"){
    return "Hello Andreas";
  }
  else {
    return "You didn't pass authentication!";
  }
}
```

Run the server first, then the client, and you'll get some output:

```
Base64 encoded: YW5kcmVhczpwYXNzd29yZA==
Response: Hello Andreas
```

It might not surprise you that this implementation is a bad one since we are hardcoding the username and password in the authentication code. The obvious choice at this point would be to move that into a database and do a lookup instead. That leads us to us calling out one of the most egregious identity implementation errors you can commit – storing passwords directly in the database. Never, ever store the password in the database. You should store a hash of the password that is not reversible and calculate whether the password entered matches what is stored in the database. That way, an attacker will not as easily be able to extract the passwords should they get hold of the database.

This begs the question of what a hash is in this context, so let's cover that next.

How hashing works

A hashing function is an algorithm for converting one value into another one, commonly used for the optimization of lookups in data structures or verification of the initial value. For instance, if we were to create a very basic hashing algorithm, we could use number replacements for characters to create a hash for a given string. Let's say A=1, B=2, and so on. The Password string would then be 16 1 19 19 23 15 4 (each number represents a single character; spaces added for readability). Let's then add these digits and divide by the number of characters – (16 + 1 + 19 + 19 + 23 + 15 + 4) / 8 = 12.125. Going with the integer part only, we end up with 12.

Instead of storing your actual password, we would store the value 12. When we type in Password as the password, we are able to compute the hash again and compare it against the stored value. It's also great because it is not reversible – even if the algorithm is known, it is not possible to reverse engineer the

number 12 to end up with Password, so a copy of the database is not going to help with figuring out the passwords.

Even if you're not a mathematical genius, you will probably spot that this algorithm is weak. With the simple substitution scheme we use, it is fairly easy to create a string that will also produce 12 as the value and thus be valid. A good hashing algorithm should produce unique values so that two different passwords are not likely to have the same hash. Luckily, Microsoft has implemented a number of hashing algorithms for .NET already, so you do not have to roll out your own.

If we were to illustrate this with pseudo-code (we will not compile since we have not implemented database lookups), it would look as follows:

```
var credentials = System.Text.Encoding.UTF8.GetString(byteEncoded);
//Split the credentials into separate parts
var username = credentials.Split(":")[0];
var password = credentials.Split(":")[1];
//Bad
if (db.CheckUser == true && db.CheckPassword == true) {
  return $"Hello {username}";
}
//Good
var myHash = System.Security.Cryptography.SHA256.Create();
var hashEncoder = System.Text.UTF8Encoding.UTF8;
var byteHashedPassword = myHash.ComputeHash(hashEncoder.GetBytes(password));
System.Text.StringBuilder sb = new System.Text.StringBuilder();
foreach (Byte b in byteHashedPassword) sb.Append(b.ToString("x2"));
var hashedPassword = sb;
if (db.CheckUser == true && db.CheckHashedPassword == true) {
  return $"Hello {username}";
}
```

By now, you might be thinking that there's a lot that goes on in authentication, and you are spot on. In fact, basic authentication is not really recommended to use, but it should hopefully have given you an idea of what authentication is. We will show some better techniques after explaining a close companion of authentication, called authorization.

Understanding authorization concepts

The second piece of the identity puzzle is authorization, usually shortened to **AuthZ**. Where **AuthN** is about finding out who you are, AuthZ is about what you are allowed to do.

Going back to the real world and how things work there, let's for a moment consider international air travel. Assume for simplicity's sake that all international travel requires you to show a passport. If you don't have a passport with you, this will be the same as not being authenticated (unauthenticated) and you will not be allowed into the destination country.

If you have a passport, the relevant authorities will examine it by asking the following questions:

- Is it issued by an actual country? (Unfortunately, ''.NET-land'' is not recognized by the United Nations.)
- Does it appear genuine, with watermarks, biometric markers, and so on, or does it look like something you printed at home?
- Can the issuing country be trusted to have good procedures in place for issuing passports?

If you pass these, you will be authenticated but you might not be able to move on to baggage claims yet. There is a new round of questions:

- Are you a citizen of a country the destination accepts travelers from?
- Are you from a country requiring a visa and if so, do you have one with you?
- Are you a convicted criminal?
- Are you a known terrorist? (The airline should probably check this before letting you on the plane in the first place, but they might have missed it.)

The details will vary depending on which country you would like to get into, but the point is the same. While your identity checks out, there are still other mechanisms in place for giving an approval stamp.

You might have recognized a similar pattern in web apps. For example, if you log in with John as the username, you have the permissions of a regular user and can do database lookups, edits, and so on. Whereas if you login with JohnAdmin as the username, you are given administrative permissions and can access system-wide server settings and whatnot. Revisiting the authentication code from the previous section, we would extend the pseudo-code to something like this:

```
public String Get() {
  var authHeader = HttpContext.Request.
  Headers[''Authorization''];
  var base64Creds = AuthenticationHeaderValue.Parse(authHeader).Parameter;
  var byteEncoded = System.Convert.  FromBase64String(base64Creds);
  var credentials = System.Text.Encoding.UTF8.  GetString(byteEncoded);
  //Split the credentials into separate parts
  var username = credentials.Split(":")[0];
  var password = credentials.Split(":")[1];
  //Password hashing magic omitted
  ...
  //Authentication code omitted
  ...
  var userrole;
  if (db.CheckRole == "Admin"){
    userrole = "Admin";
  }
  if (db.CheckRole == "User"){
    userrole = "User";
  }
  else {
    return "You didn't pass authentication!";
  }
  return $"Hello { userrole}";
}
```

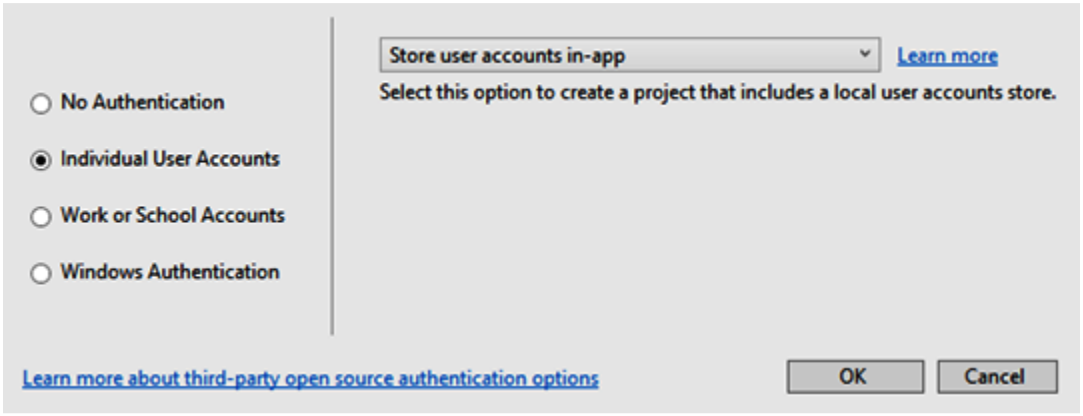
Even though this is also pseudo-code where we're missing the role lookup, we can see how it adds an additional layer when we introduce authorization. It could be that your web app might not need to distinguish between roles, but the point we are making here is one we have been building up to over a couple of pages now.

Do not implement your own identity solution from scratch (or based on this sample code).

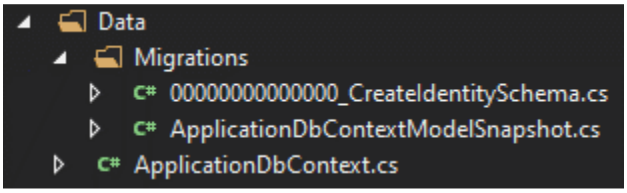
This is not to discredit the knowledge and competency of the students; it is a general best practice that this should be done by those who have it as a full-time job who have access to a team reviewing and testing everything with a critical eye.

Microsoft has included a template in Visual Studio for a SQL-backed web app that implements a similar identity setup:

1. Start Visual Studio and select **Create a new project**.
2. Select the **ASP.NET Core Web Application** template and hit **Next**.
3. Name the solution DB_Auth and select a suitable location for this training's exercises and click on **Create**.
4. Select the **Web Application (Model-View-Controller)** option and click Change under Authentication. Make sure you select **Individual User Accounts** and Store user accounts in-app before clicking **OK**, followed by **Create**:



5. If you take a look at the Data folder, you will see the code that generates a database where the user accounts are stored as shown in figure below:



6. Open up 00000000000000_CreateIdentitySchema.cs. It should be 200+ lines of code, and the user object looks like this:

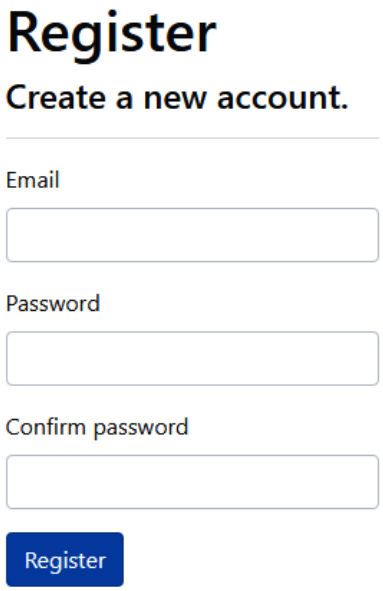
```
migrationBuilder.CreateTable(
  name: "AspNetUsers", columns: table => new {
    Id = table.Column<string>(nullable: false),
    UserName = table.Column<string>(maxLength: 256, nullable: true),
    NormalizedUserName = table.Column<string>(maxLength: 256, nullable: true),
    Email = table.Column<string>(maxLength: 256, nullable: true),
    NormalizedEmail = table.Column<string>(maxLength: 256, nullable: true),
    EmailConfirmed = table.Column<bool>(nullable: false),
    PasswordHash = table.Column<string>(nullable: true),
    SecurityStamp = table.Column<string>(nullable: true),
    ConcurrencyStamp = table.Column<string>(nullable: true),
    PhoneNumber = table.Column<string>(nullable: true),
    PhoneNumberConfirmed = table.Column<bool>(nullable: false),
    TwoFactorEnabled = table.Column<bool>(nullable: false),
    LockoutEnd = table.Column<DateTimeOffset>(nullable: true),
    LockoutEnabled = table.Column<bool>(nullable: false),
    AccessFailedCount = table.Column<int>(nullable: false)
  },
  constraints: table => {
    table.PrimaryKey("PK_AspNetUsers", x => x.Id);
  }
);
```

The names should be fairly self-explanatory, but as you can see, there is a little bit more to it than a username and a hashed password.

7. Taking a quick look at the configuration in Startup.cs, we can see where the database is initialized and requires authentication to happen:

```
public void ConfigureServices(IServiceCollection services) {
  services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"))
  );
  services.AddDefaultIdentity<IdentityUser>(
    options =>options.SignIn.RequireConfirmedAccount = true
  ).AddEntityFrameworkStores<ApplicationDbContext>();
  services.AddControllersWithViews();
}
```

8. Following this up by attempting to run the app, there should be a form for registering an email address and defining a password. Figure below is an example of signing up:



9. If you peek into the rest of the files that were scaffolded, you will notice that there is actually a bit of code to make it all run, and then there's everything in the libraries you don't see, solidifying why you would prefer not to do all of this yourself.

Templates like these used to be very popular years ago as they took away a lot of the hard work and users were accustomed to register on every site they visited. While there's nothing inherently wrong with using this – it's secure and maintained by Microsoft – it has become less common now that there are other options.

We will resume our regular programming soon, but the previous code snippet provides an entry point for us to segue into a topic that technically is not related to identity, but is useful for understanding how different identity pieces play into .NET apps.

The role of middleware in ASP.NET and identity

A lot of technologies and products start with a code name, and when Microsoft came up with Project Katana, it certainly had a zing to the name. This project came about in 2013 to address a couple of shortcomings in .NET at the time.

We're not going to drag up old .NET code and point to flaws in the design here, but even without going into the details, you can probably relate to the challenge of replacing components in your code. Let's say, for instance, that you start out creating a utility for controlling some smart light bulbs you have in your home. During troubleshooting one day, you realize that it would be easier if you captured some information and logged it. The quick-and-dirty method is to append lines to a file called log.txt. This works nicely until you realize that you could use some insight into non-error conditions as well, such as logging when the lights were turned on and off to create some stats for yourself.

This doesn't lend itself as easily to be logged in a text file when you want to use it outside the app. So, you realize it could be nice to have in a database. Then you have to rewrite all those calls to a file to log to a database instead. You get the picture.

It would be nice to have a more generic log.Info("Lights out") method that did not care about the etails. Since logging is a common concern in many apps, there are a number of logging frameworks out there, but there's still a setup ceremony to it per app.This module is about identity, so what's the connection, you say? Well, authentication and authorization are also common use cases for apps. And so is URL routing in web apps, caching, and a couple of other things as well.

Another facet of these components is that you most likely want to run them as early as possible during the initialization of the app – loading the logging component when something fails might be too late.

That was an elaborate setup for saying that Microsoft has built an abstraction called middleware. Project Katana actually covered four components, and this carries over for the current implementation - host, server, middleware, and application.

The host part can be found in Program.cs and for a web app, it looks like this:

```
public class Program {
  public static void Main(string[] args) {
    CreateHostBuilder(args).Build().Run();
  }
  public static IHostBuilder CreateHostBuilder(string[] args)  =>
    Host.CreateDefaultBuilder(args).ConfigureWebHostDefaults(
      webBuilder => {webBuilder.UseStartup<Startup>();}
  );
}
```

If you compare this to the worker service we created in module **Cross-Platform Setup**, you will notice similarities:

```
public class Program {
  public static void Main(string[] args) {
    CreateHostBuilder(args).Build().Run();
  }
  public static IHostBuilder CreateHostBuilder(string[] args)  =>
    Host.CreateDefaultBuilder(args)
      .UseWindowsService()
      .UseSystemd()
      .ConfigureServices((hostContext, services) => {
        services.AddHostedService<Worker>();
```

```
  });
}
```

You're not able to turn any web app into a service by changing these lines, but notice how the pattern is the same.

We've already mentioned, and peeked into, the Startup.cs file, which is where the server and middleware components can be found.

The server and services are invoked by the runtime with this code:

```
public void ConfigureServices(IServiceCollection services) {
  …
  services.AddControllersWithViews();
  …
}
```

The actual runtime might vary, as we've already seen, depending on whether you host in IIS or Kestrel (which does not matter in this context).

The middleware is found in the next section of the file:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
  if (env.IsDevelopment()) {
    app.UseDeveloperExceptionPage();
  } else {
    app.UseExceptionHandler("/Error");
    app.UseHsts();
  }
  …
}
```

This is called a pipeline, and it builds as a sequence – authentication goes before authorization, for instance, but not all middleware is sensitive to which step it is loaded at.

Some of the middleware has a binary behavior – UseHttpsRedirection enables exactly that, and if you don't want it, you simply remove it.

UseEndpoints lets you add specific endpoints you want to listen to:

```
app.UseEndpoints(endpoints => {
  endpoints.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

The beauty of middleware and identity is that you can add custom middleware to the mix, and since the usage is standardized, it is fairly pain-free to change afterward. We did not implement basic auth as middleware, but the boilerplate added by the wizard in Visual Studio for using a local database did.

This will become handy if we were to upgrade our identity implementation to be based on OAuth, which will be covered next.

OAuth and OpenID Connect basics

Basic authentication is simple to implement, and if you need to work with legacy systems, there's a good chance you will run into it. It's not recommended to start new projects using basic authentication though.

There is no shortage of acronyms for protocols in the identity space, and .NET Framework has relied upon different authentication and authorization protocols over the years. We are not able to delve into all of them, nor to do a comparison of the strengths and weaknesses of them.

The most popular set of protocols used for AuthN and AuthZ purposes these days is **OAuth** and **OpenID Connect (OIDC)**, so we will look at parts of both the theory and practical implementations. OAuth is the base protocol and OIDC builds on top of this, so there are some overlapping details we will get back to.

Looking back at basic authentication, we already mentioned that a drawback is the fact that the passwords are transferred over the wire. Both the client and server side have access to the actual password, which is, in many cases, more than they need. For instance, a web app will certainly care about establishing whether you have an administrator role before allowing you access to the administrative settings, but as long as the identity is established, the password doesn't provide any value in doing this authorization step. That's just extra data you need to protect.

OAuth decouples these parts so that the server side does not need to know the password. For the client, it is more a case of ''it depends'' for how this is handled – if a password is required, you can't avoid typing it somewhere. It all starts with what are called **JSON Web Tokens (JWTs)**, so let's cover that first.

JSON web tokens

With OAuth and OIDC, we don't rely on passing around username:password as the key to the kingdom, but instead, we rely on passing around tokens. These tokens are called JWTs, and pronounced jot/jots A JWT is formatted as JSON and contains three parts – a header, payload, and signature. A sample JWT could look like this:

```
{
  "alg": "RS256",
  "kid": "4B92FBAE5D98B4D2AB43ACE4198026073012E17F",
  "x5t": "S5L7rl2YtNKrQ6zkGYAmBzAS4X8",
  "typ": "JWT"
}.{
  "sub": "john.doe @contoso.com",
  "nbf": 1596035128,
  "exp": 1596038728,
  "iss": "contoso",
  "aud": "MyWebApp"
}.[Signature]
```

If you have not seen anything like this before, you probably have (at least) two questions:

- What do all these things mean?
- How does this actually help?

The information in this token is called claims – so, for instance, the "sub" claim is short for subject and has the value john.doe@contoso.com. This claim is usually the user/username (it does not have to be in email format, but this is common).

The rest of the claims are as follows.

The header is as follows:

- "alg ": The algorithm used for generating the signature
- "kid": Key identifier
- "x5t": Key identifier
- "typ": The type of the token

The payload is as follows:

- "nbf ": Not before.The time from which the token is valid; usually the same time as it was issued.
- "exp": Expiration time.The time the token is valid until. Usually an hour from when it was issued (but this is up to the token issuer).
- "iss ": Issuer.The issuer of the token.
- "aud": Audience.Who the token is intended for; usually the app the token is intended for.

This is just a minimal sample token – you can have more claims if you want to, and you choose the format of these. If you want a "foo" claim with a value of "bar" that only makes sense for your application, that is OK. Just be aware that the token does not have an unlimited size – in enterprise environments, some developers try to include all the groups the user is a member of. When the user is a member of 200+ groups, you experience what is known as token bloat, which causes the token to be fragmented when transferring over a network. In most cases, these packets are not reassembled correctly, and things fall apart.

Passing the token to the server is similar to basic authentication in that we add an authorization header where the token is Base64-encoded (token shortened for brevity):

```
Authorization: Bearer eyJhbGciOi...PDh4ck7Q
```

This is nifty as you can send more information than when passing the username and password while still keeping the credentials out of the data transmission. It is called a bearer token because anyone who possesses it can use it. This brings us back to question number two – how is this better? The first impression you get is that any client can craft their own token and that doesn't sound like a good mechanism.

There are two important actions in OAuth/OIDC transactions:

- Issuing a token: This is about controlling who gets a token and this will be protected by one or more mechanisms.
- Validating a token: This is about checking that the token is trustworthy and what the contents are.

Both of them are primarily based on using certificates – signing when issuing and verifying when validating. (Note that this is not the same as certificate-based auth; we're only focusing on the token itself here.)

Let's take a look at how this works in code.

How to generate/issue a token

In module **Cross-Platform Setup**, we showed how to generate a certificate, install it on Windows and Linux, as well as reading it afterward. Building on this, we can use the same certificate for signing a token.

To create an app that will generate a token, do the following:

1. Open up the command line and create a new directory (BearerAuthClient).
2. Run the dotnet new console command.
3. Run the dotnet add package System.IdentityModel.Tokens.Jwt command.
4. We then need to add some code to Program.cs. First, we create the token (based on a generic template):

```
static void Main(string[] args) {
  jwt = new GenericToken {
    Audience = "BearerAuth",
    IssuedAt = DateTime.UtcNow.ToString(),
    iat = DateTimeOffset.UtcNow.ToUnixTimeSeconds().ToString(),
    Expiration = DateTime.UtcNow.AddMinutes(60).ToString(),
    exp = DateTimeOffset.UtcNow.AddMinutes(60).ToUnixTimeSeconds().ToString(),
    Issuer = "Module 08",
    Subject = "john.doe@contoso.com",
  };
```

Then, we set up/retrieve the certificates we use for signing:

```
SigningCredentials = new Lazy<X509SigningCredentials>(()=> {
  X509Store certStore = new X509Store(StoreName.My, StoreLocation.CurrentUser);
  certStore.Open(OpenFlags.ReadOnly);
  X509Certificate2Collection certCollection = certStore.Certificates.Find(
    X509FindType.FindByThumbprint,
    SigningCertThumbprint,
    false
  );

  // Get the first cert with the thumbprint
  if (certCollection.Count > 0) {
   return new X509SigningCredentials(certCollection[0]);
  }
  throw new Exception("Certificate not found");
});
```

The final piece is lining up the claims and creating the actual signed token:

```
IList<System.Security.Claims.Claim> claims =
  new List<System.Security.Claims.Claim>();
  claims.Add(new System.Security.Claims.Claim(
    "sub", jwt.Subject,
    System.Security.Claims.ClaimValueTypes.String,
    jwt.Issuer)
  );

  // Create the token
  JwtSecurityToken token = new JwtSecurityToken(
    jwt.Issuer, jwt.Audience, claims,
    DateTime.Parse(jwt.IssuedAt),
    DateTime.Parse(jwt.Expiration),
    SigningCredentials.Value
  );

  // Get the string representation of the signed token and print it
  JwtSecurityTokenHandler jwtHandler = new JwtSecurityTokenHandler();
  output = jwtHandler.WriteToken(token);
  Console.WriteLine($"Token: {output}");
}
```

Note that in order to focus on the important pieces, this is not the complete code

5. Run the dotnet run command. Your output will look similar to figure below:

```
c:\Code\Book\Chapter_07_BearerAuthClient>dotnet run
Token: eyJhbGciOiJSUzI1NiIsImtpZCI6IkRFQUYyQjhBNDg2NEM1
0b3NvLmNvbSIsIm5iZiI6MTU5NjEwNjc2NCwiZXhwIjoxNTk2MTEwMz
RiKogGtX1MH0zd0YddxV12hW0cCTHmLpuX0sL0sn16NJ0CFHGEzMqIX
jA30P6AOhbkVbps5bdjkSeqAyuaZQXQK3LveY8qz1IO3UGzC-sKXrl0
```

This is not intended for you to read, but it is reversible as it is just Base64-encoded. The great part is that your actual secret is not included, so even if someone were able to read it, that's not a problem.

How to validate a token

Generating a token is nice and dandy, but unsurprisingly, we need a counterpart –checking that the token is good and allowing or rejecting access based on this evaluation.

For this, we will also create a server-side code sample:

1. Open up the command line and create a new directory (BearerAuthServer).
2. Run the dotnet new console command.
3. Run the dotnet add package System.IdentityModel.Tokens.Jwt command.
4. The following code goes into EchoController.cs:

```
[HttpGet]
public String Get() {
  var audience = "Module_08_BearerAuth";
  var issuer = "Module 08";
  var authHeader = HttpContext.Request.Headers["Authorization"];
  var base64Token = AuthenticationHeaderValue.Parse(authHeader).Parameter;
  JwtSecurityTokenHandler handler = new JwtSecurityTokenHandler();
  TokenValidationParameters validationParameters = null;
  validationParameters = new TokenValidationParameters {
    ValidIssuer = issuer,
    ValidAudience = audience,
    ValidateLifetime = true,
    ValidateAudience = true,
    ValidateIssuer = true,

    //Needed to force disabling signature validation
    SignatureValidator = delegate (string token,
    TokenValidationParameters parameters) {
      var jwt = new JwtSecurityToken(token);
      return jwt;
    },
    ValidateIssuerSigningKey = false,
  };
  try {
    SecurityToken validatedToken;
    var identity = handler.ValidateToken(base64Token,
    validationParameters, out validatedToken);
    return "Token is valid!";
  } catch (Exception e) {
    return $"Token failed to validate: {e.Message}";
  }
}
```

As in the previous code sample, parts have been left out for readability.

5. Run the dotnet run command.

6. Step back to the client-side code and add the following code:

```
HttpClient client = new HttpClient();
client.DefaultRequestHeaders.Authorization =
  new AuthenticationHeaderValue("Bearer", output);
var response = client.GetAsync("https://localhost:5001/Echo").Result;
Console.WriteLine(response.Content.ReadAsStringAsync().Result.ToString());
```

7. Run the dotnet run command in this folder while the server part is running. You should see an output that says Token is valid.

While there are terms in the server code that intuitively have a meaning, a little bit of explaining of the procedure is probably warranted.

The basics are that we configure values for the issuer (whoever issued the token) and the audience (who the intended recipient of the token is). We then configure the parameters for validating the token; the aforementioned audience and issuer as well as the time stamp of the token.

If the token is valid, we return a message indicating so, and if it fails, we return a different message.

In this code, we disabled checking the signature, and that might seem counterintuitive. You should always validate the signature – if not, anyone can generate a token that will pass as valid as long as they figure out the right values to insert. The reason for disabling this important piece of the puzzle is that the code becomes much more complex if we want to do that. We need to cover some additional topics first before returning to an approach that requires less complexity to get it right.

OAuth flows

It is all nice and dandy to be able to send a token to an API and have it validated, but you might wonder how this would actually work in an app. We can't have a user type in the details we used here, and even if we only did this on a server, there are no credentials involved. That doesn't sound like something you would actually use in real life.

JWTs are a central piece of OAuth, but there is more to the protocols than the token. OAuth consists of what we call ''flows'' that prescribe the steps on the journey to acquiring and using said token. We will not be able to cover all the variants of these flows here, but we will cover a few that are relevant to ASP.NET Core use cases.

There are a couple of terms we need to sort out that applies to all of the flows.

Instead of each application handling the issuing of tokens, we have a central service known as an identity provider. This service usually verifies the credentials (password, certificate, and so on) and takes care of issuing tokens. While this is technically something you can implement on your own, it is highly recommended to go for an established solution in the market (we will be taking a look at using Azure AD for this purpose).

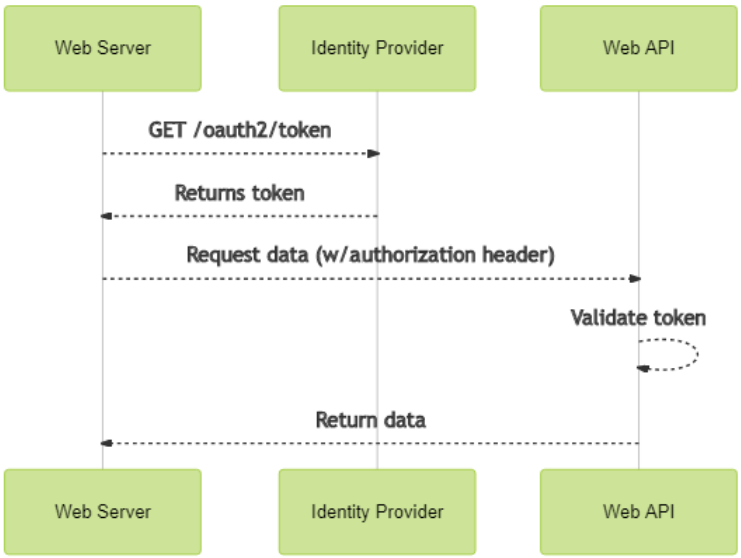
When acquiring tokens, the client requests which permissions it would like. These permissions are known as scopes and are embedded in the token as claims.

The flows described here drive the login for Facebook, Google, and Microsoft, so you have most likely tried them out already even if you didn't give it much thought at the time.

(These providers support multiple flows to support different use cases.)

OAuth Client Credentials grant

The easiest flow to understand is probably the Client Credentials flow as this is the closest to using a username and a password. You would register an application in the UI for the identity provider you're using and get a client ID and a client secret. When you want to acquire a token, you send these to an identity provider and indicate which permissions you would like. The flow goes like figure below:



A very important thing to note is that this flow is only intended for trusted clients. A trusted client typically runs on a server where the code and configuration are not available to the end user. This is typically a service account, or server-side-rendered web apps. The client ID is not sensitive, but paired with the client secret, it potentially enables anyone possessing it to extract information they should not have. If you have a client-side app such as JavaScript that is downloaded to the browser, a mobile app, or something similar,

you should never use the Client Credentials flow. The client secret is usually too long and complex for a user to remember and type in, so for passwords, there are different flows.

OAuth Resource Owner Password Credentials

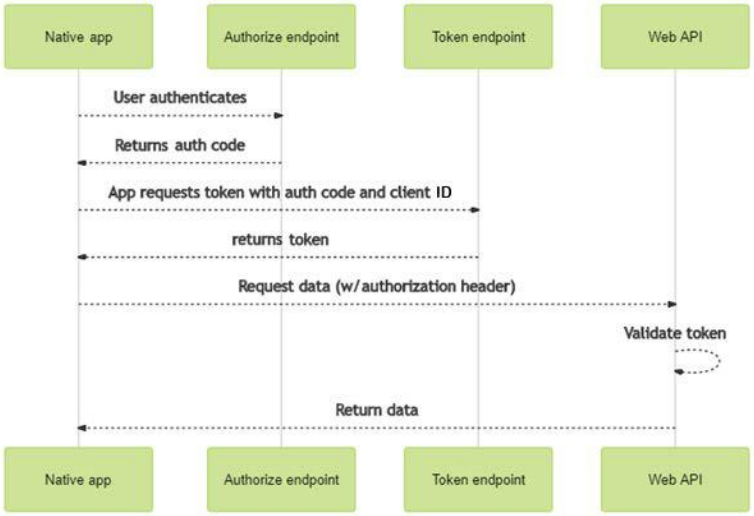
A flow that is similar to Client Credentials but intended for user credentials is the Resource Owner Password Credentials (ROPC) flow. When using an external identity provider, there will often be a predefined look and feel of the login experience and usually, it's rendered as HTML in a browser. Even if there is an option to style it to your own liking, it is not unusual that the people working with user experience will say that they need to tweak some element a certain way for them to be happy.

At this point, you might be thinking it would be great if you could create all the visual aspects yourself and deal with the authentication just like when you're implementing a server-side authentication experience. Such an option exists with this flow, but you should never admit to the designers that it exists. It is highly discouraged to use this flow, by Microsoft and the identity community, because it is inherently less secure than handling the credentials exchange directly at a specialized product for handling identity use cases. The app takes on much more responsibility since it will have knowledge of the user's password.

We only mention it here because it is useful to be aware of it even if it does not come recommended.

OAuth Authorization Code grant

The recommended way to do authentication in a native app is a flow called the Authorization Code flow. It might come off as slightly complicated the first time you run into it, but there is a logic behind it. We need the user to enter their credentials manually, but the app should not be aware of them. At the same time, we want the application to be an entity as well when calling into APIs. A diagram would look like figure below:

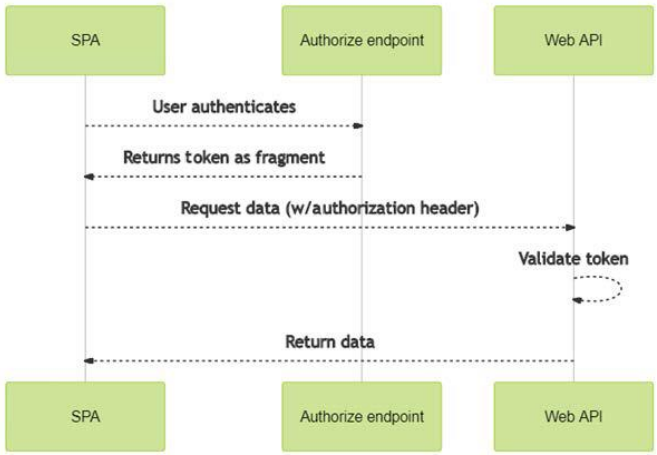


Both the authorize and token endpoints are located on the identity provider.

This diagram does not cover the low-level details, but a possible attack vector in this scenario is that, for instance, on a mobile device, a malicious app might be able to intercept the auth code and use it for its own non-approved purposes. You are recommended to implement an extension to the flow called Proof Key for Code Exchange (PKCE – pronounced pixie), which ensures only the right app can use a specific auth code.

OAuth Implicit Grant flow

It is mostly clear what a classic web app is and what a classic native app means, but where does something such as a JavaScript-based **Single-Page Application (SPA)** fit in? It is sort of a hybrid in the sense that you have code supplied by the browser that is executed locally. This means that you cannot consider it a trusted client. You will see many guides referring to using the Implicit Grant flow for these purposes. It looks like figure below:



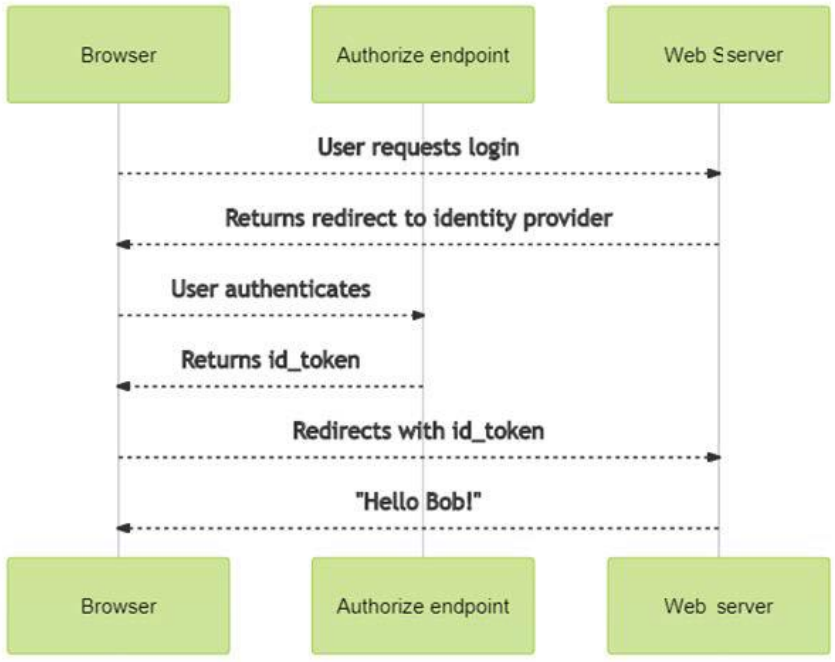
The meaning of fragment here is that the token will be part of a URL when redirecting back to the SPA instead of returning it in the body of the HTTP response. This is due to how most SPAs don't ''jump between pages'' like non-SPA web apps and need to consume data through the URL.

While there are use cases where an implicit grant is suitable, and it is being used in a lot of places, the current recommendation is that auth code with PKCE is more suited for most SPAs. Implicit Grant is less secure, so while it is functionally acceptable, it has other drawbacks.

Note that if you are using libraries to provide this functionality, you should try to find out which of the two flows it uses behind the scenes.

OpenID Connect

All of the previous flows focused on acquiring tokens that said ''you're allowed to access this API.'' This is, of course, a very important scenario to solve, but if you try logging in to a web app without touching an API, you often just want to know ''who signed in.'' For this, we have the OIDC flow, or more correctly, a separate protocol building on top of OAuth as seen in figure below:



The OIDC protocol has some other things included as well that make signing in easier as a developer, which we will get back to in our code samples.

There are other OAuth flows as well, and it can be more elaborate than what we have shown here, but it is out of the scope of this training to cover all the nuances of AuthN and AuthZ.

These flows are no good without an identity provider, so in the next section, we will put everything into context by using a popular provider.

Integrating with Azure Active Directory

Chances are that if you have logged in to a corporate computer the past 20 years, you have used Active Directory, whether you are aware of it or not. AD was introduced with Windows Server 2000 and extended the domain concept introduced in Windows NT 4.0 to provide a complete implementation of centralized identities. When you logged in to your Windows desktop, it provided fairly pain-free access to file shares and servers in an organization as long as you were seated in the office.

With AD, you need at least a couple of servers on-premises and accompanying infrastructure. This isn't feasible in the cloud world of today, but Microsoft built upon what they had to provide Azure Active Directory (AAD) as a cloud identity provider, breaking free from the constraints of physical locations at the same time.

AD is based on older identity protocols, so the OAuth flows and OIDC are not natively supported, but require the use of Active Directory Federation Services (ADFS) as an additional service to support what we just described. This does not carry an extra cost over a Windows Server license, but it is recommended to have dedicated servers for this service.

Conversely, AAD was built with the newer protocols in mind, so it does not support the older protocols without additional components.

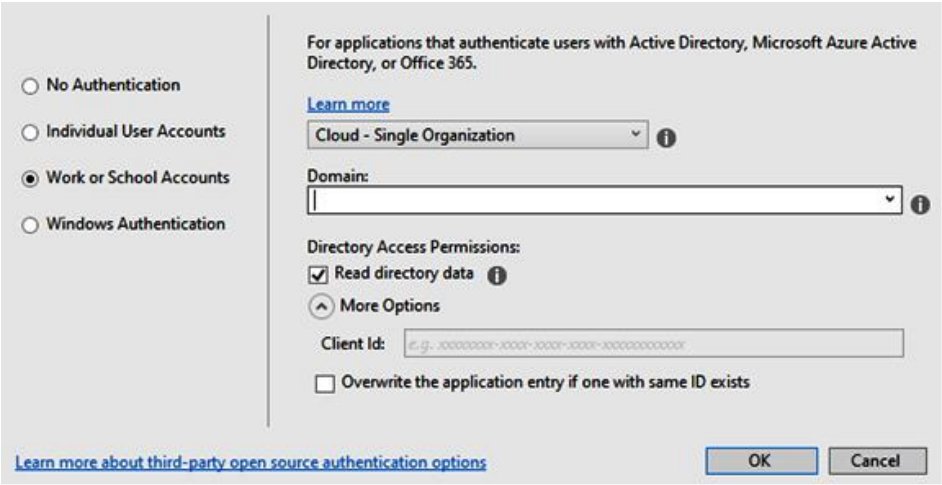
This means that it is likely that if you want to migrate an existing on-premises app with AD support to AAD, you need to do some rewriting of the identity stack. We will not cover this, but rather go straight to the newer protocols. AAD is based on open standards, and you can fairly easily replace it with other identity providers that comply with the standards, so this isn't a Microsoft lock-in either.

AAD in its basic form is free. There are some advanced security features you don't get for free, and you are limited to 50,000 objects, but this should be sufficient even for many production deployments. Per the technical requirements listed at the beginning of the module, we assume you have an AAD tenant for these samples, so you should sign up now if you haven't done so already.

Using AAD unlocks a range of options in the Azure portal. You can, for instance, control whether all the flows we described should be available or whether only a subset is used. In addition, you can specify which users have access, what other data sources the application can access, and more.

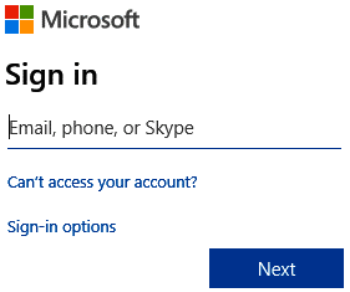
If you have an existing web application, it is possible to add support for AAD to this, but to simplify matters, we will be creating a Blazor app from scratch with the wizard in Visual Studio doing the backend configuration in Azure for us:

1. Start Visual Studio 2019 and select Create a new project.
2. Select Blazor App and click Next.
3. Name the solution AADAuth.
4. Click Change under Authentication.
5. Select Work or School Accounts and select Cloud - Single Organization as shown in figure below:

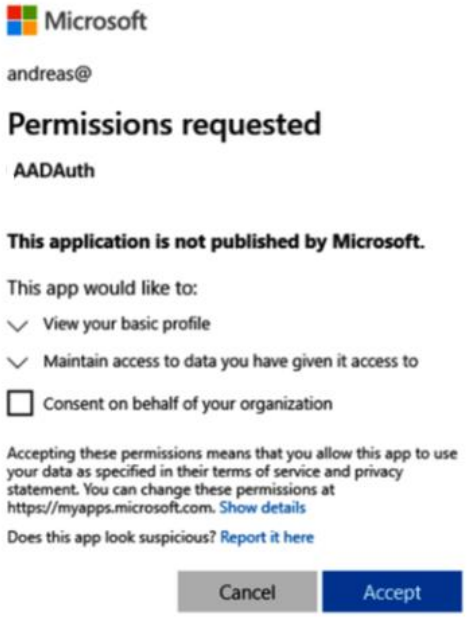


6. Type in the domain name of the AAD tenant you will be using. You will be prompted to sign in if you haven't done so before.
7. Make sure you select **Blazor Server App** and that you have checked **Configure for HTTPS** before clicking **Create**.

If you try running the app, the first thing that will hit you is a sign-in form provided by Microsoft as seen in figure below:



After typing your username followed by the password, the next thing is a request for permissions as shown in figure below:



Provided you click the Accept button, the app will open and in the upper-right corner, you will be greeted with your name. Seems easy enough, but let's take a look at what's going on in the code before adding some more functionality.

If you open Startup.cs, you might notice some code you haven't seen so far:

```
public void ConfigureServices(IServiceCollection services) {
  services.
  AddMicrosoftIdentityWebAppAuthentication(Configuration, "AzureAd");
  services.AddControllersWithViews().AddMicrosoftIdentityUI();
  services.AddAuthorization(options =>{
    // By default, all incoming requests will be authorized
    // according to the default policy
    options.FallbackPolicy = options.DefaultPolicy;
  });
  services.AddRazorPages();
  services.AddServerSideBlazor().AddMicrosoftIdentityConsentHandler();
}
```

In a previous section, we mentioned how easy it is to swap out identity middleware and we can see here how the startup pipeline has seen the addition of middleware both for handling identity and the related UI.

If we take a look at appsettings.json, we can see where our specific configuration is stored:

```
{
  "AzureAd": {
    "Instance": "https://login.microsoftonline.com/",
    "Domain": "contoso.com",
    "TenantId": "tenant - guid",
    "ClientId": "client - guid",
    "CallbackPath": " / signin - oidc"
  },
```

You might find it slightly unfriendly that you are hit with a login prompt before even seeing the web page. There are a lot of pages that offer a default experience when you're not logged in where functionality is unlocked when signing in.

This is controlled by a couple of lines of code in Startup.cs:

```
//Comment out the line below like this services.AddRazorPages();
//And replace with this
services.AddRazorPages(options => {
  options.Conventions.AllowAnonymousToPage("/_Host");
});
```

Be aware that this effectively shuts off authorization for all pages in the Blazor app, so you need to enable it for the pages where you need it. (The details of how you change the default behavior varies between the different view engines – MVC, Razor Pages, and Blazor.)

You can replace the contents of Index.razor with the following code:

```
@page  "/"
<AuthorizeView>
  <Authorized>
    Hello, @context.User.Identity.Name!
    <table class=" table">
      <thead>
```

```
        <tr>
          <th scope=" col">Claim Type</th>
          <th scope=" col">Claim Value</th>
        </tr>
      </thead>
      <tbody>
        @foreach (var claim in context.User.Claims) {
          <tr>
            <td>@claim.Type</td>
            <td>@claim.Value</td>
          </tr>
        }
      </tbody>
    </table>
  </Authorized>
  <NotAuthorized>
    <p>For full functionality please log in</p>
    <a href=" MicrosoftIdentity /Account/SignIn">Log in</a>
  </NotAuthorized>

</AuthorizeView>
```

This will print all the claims in your token, which only makes sense when logged in, and provide a link for logging in when you have not authenticated yet. This approach is suitable for when you need a page to be available both for logged-in and anonymous users.

If you want to block all the content of a page, you can do this (in Counter.razor) by adding the [Authorize] attribute:

```
@page  "/counter"
@attribute [Authorize]

<h1>Counter</h1>
```

Users who are not logged in will simply see a message that they are not authorized.

There are a multitude of ways to configure this. You can create policies that require specific claims to be present, you can create roles that control access to a view, and more. We don't recommend making it more complex than necessary though, especially when starting out.

It can be cumbersome to troubleshoot, so get the basics right first.

Understanding single tenancy versus multi-tenancy

In the wizard, we chose Cloud - Single Organization, but if you checked the dropdown, you probably noticed Cloud - Multiple Organizations as well. We should probably explain those.

An organization here is an AAD tenant. This means that if your company structure has multiple tenants, this is considered to be multiple organizations even though it may be only one legal organization. It is a purely technical definition.

When you create a single organization application, that means that only users of one specific AAD tenant will be able to log in, and the data consumed is primarily data constrained to this tenant. If you build an

app that is only ever to be consumed by you and your co-workers, this is a good choice as there will be a logical boundary and you don't end up spilling data into other organizations.

For multi-org apps, there are a couple of reasons behind you would want to change the configuration. Let's say we have a web shop selling computer supplies to businesses. We make the assumption that most of our customers have AAD already – instead of implementing our own user database, we offer sign in with AAD from customers' tenants. Even though we have a shared database of our sales, we can enforce that, for example, only users signing in from contoso.com can access the orders tagged with Contoso as the company name.

A slightly different setup would be that we are an ISV that sells a piece of software to businesses. If a company is already using AAD, single sign-on would usually be high on their wish list. The app can be architected to create the illusion of being for one organization, but it can reuse a common set of user administration across different companies.

The default setting in a multi-tenant app is that all tenants in AAD are allowed to authenticate. It is possible to restrict this if you want to by editing the token validation parameters, but the most important part of this is that you need to figure out the authorization setup as well.

Understanding consent and permissions

You were asked to grant permissions when running the app, but we didn't really explain this part. The basic concept should be easy to grasp – your AAD account potentially unlocks access to a lot of data if you use other Microsoft services, such as Office 365. We don't want an app to grab whatever it desires, so as a safeguard, the app has to request access and it has to be granted.

There are two types of permissions:

- Delegated permissions are permissions that are valid in a user context. For instance, if an app wants to read your calendar, you as the user has to grant this. Your consent is only applicable to you – it does not enable the app to read other users' calendars.
- Application permissions are permissions that are valid in the broader app context – often in a backend. Say, for instance, the app needs to be able to list all users in the organization – this is not data that is specific to you. This permission needs to be granted by a global admin. This means that if you are not a global admin, and the app cannot function without these permissions, you cannot use the app before someone in the organization with the appropriate role consents.

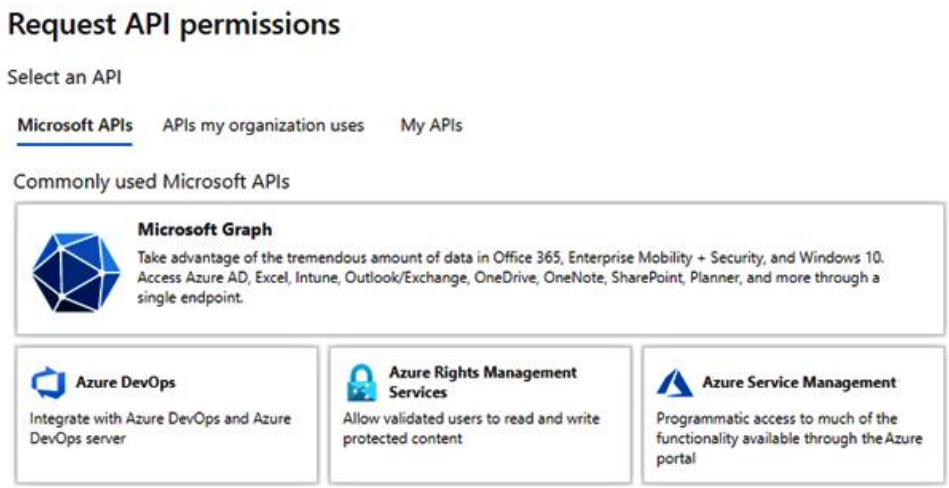
As we mentioned previously, the technical term in code for these permissions is scope. A default OIDC flow requests the offline_access and User.Read scopes, and if you want to read the calendar, you would add Calendars.Read. This is found in Startup.cs:

```
public void ConfigureServices(IServiceCollection services) {
  services.AddAuthentication
  (OpenIdConnectDefaults.AuthenticationScheme).AddMicrosoftIdentityWebApp(options => {
    Configuration.Bind("AzureAD", options);
    options.ResponseType = "code";
    options.SaveTokens = true;
    options.Scope.Add("offline_access");
    options.Scope.Add("User.Read");
    options.Scope.Add("Calendars.Read");

  });
```

Note that while you will not be prompted again to consent to the same set of permissions between separate logins, you will need to re-consent if the app requests more scopes than what you originally consented to.

You might be thinking – how do we figure out what the scopes are named? If you locate the app registration in the Azure portal, you can browse the list dynamically as shown in figure below:



For Microsoft APIs, it is, of course, also listed in the online documentation, so you don't have to take a guess as to what the permission is called.

Having permission to read the calendar is helpful, but this does not mean that calendar entries start pouring in by themselves. That requires more code. We need to elaborate on a couple of concepts first, though.

Every user in an AAD tenant can authenticate and acquire a token. This is done through the AAD endpoints, and in the code we used, this was done with the Microsoft.

Identity.Web library. This is intended for backend usage, such as web apps running server side (we used Blazor Server) and protected web APIs.

To acquire tokens on a client, we use a different library, called Microsoft Authentication Library (MSAL), which can run on native apps in C#, JavaScript-based web apps, and so on. It works with the same endpoints but implements different OAuth flows. When searching the internet, you might also come across a library called ADAL, which is the older and deprecated library; you should not be using it any longer.

Calendar data is dependent on having an Office 365 license. This data is exposed through Microsoft Graph, which is a gateway for a number of Microsoft services providing a coherent API surface. To interact with the Microsoft Graph, you can use the Microsoft Graph NuGet package after using one of the aforementioned libraries to acquire a token.

With that covered, we can circle back to the question of how to read the calendar entries.

The client has already acquired a token, so the first approach would probably be to think that this can be leveraged fairly easily. The token is not directly accessible to the app though, as it is stored in the browser

session, so you would need to retrieve it with some extra steps. Microsoft has fortunately made these steps much easier with the Microsoft.Identity.Web library.

Behind the scenes, the library invokes an OAuth flow called On-Behalf-Of (OBO). We're not painting the full picture of the flow here, but the high-level view is that the app first lets the user authenticate before using the token to perform a second call to the identity provider authenticating as itself as well. This enables the app to build out more complex scenarios when you have a lot of backend APIs.

To make this work, we have to do a couple of things:

1. Go to the Azure portal and locate the app registration in AAD.
2. Go to the **API Permissions** blade and click **Add a permission**.
3. Select **Microsoft Graph**, the **Delegated permissions** permission type, and locate Calendars.Read and Calendars.ReadWrite in the list.
4. Click **Add permission**.
5. Go to the **Certificates and secrets** blade and click **New client secret**. Give it a name such as MySecret and select when it expires, before clicking **Add**.
6. Make a copy of the secret immediately as it will not be retrievable after navigating away from the page.
7. Add new configurations to appsettings.json:

```
"AzureAd": {
  …
  "ClientSecret": "copied from the portal",
  "CallbackPath": "/signin - oidc"
},
"Graph": {
  "BaseUrl": "https://graph.microsoft.com/v1.0",
  "Scopes": "user.read calendars.read calendars.readwrite"
 },
"Logging": {
```

8. Go back to Startup.cs and change the code we added previously to look like this:

```
string[] initialScopes =
       Configuration.GetValue<string>("Graph: Scopes")?.Split(' ');
services
  .AddAuthentication(OpenIdConnectDefaults.AuthenticationScheme)
  .AddMicrosoftIdentityWebApp(Configuration.GetSection("AzureAd"))
  .EnableTokenAcquisitionToCallDownstreamApi(initialScopes)
  .AddInMemoryTokenCaches()
  .AddMicrosoftGraph(Configuration.GetSection("Graph"));
```

9. Since this is a Blazor app, we will add a page called Calendar to show the calendar entries. The first part is adding the following at the top:

```
@page  "/Calendar"
@using Microsoft.Graph
@inject Microsoft.Graph.GraphServiceClient GraphClient
```

   The injected GraphClient takes care of passing along the token you need to call Microsoft Graph.
10. You need a code section to actually call the graph:

```
@code {
  private List<Event> eventList = new List<Event>();
  protected override async Task OnInitializedAsync() {
    try {
```

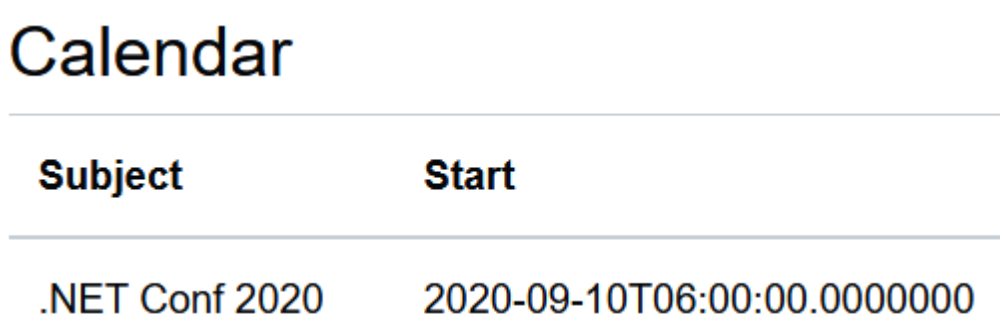
```
      var events = await GraphClient.Me.Events.Request()
        .Select("subject, body, organizer, start, end, location")
        .GetAsync();
      eventList = events.CurrentPage.ToList();
    } catch (Exception ex) {
      var error = ex.Message;
    }
  }
}
```

11. Then, you need to print it all out, as shown in the following code block:

```
<AuthorizeView>
  <Authorized>
    <table class="table">
      <thead>
        <tr>
          <th scope="col">Subject</th>
          <th scope="col">Start</th>
          <th scope="col">Entry</th>
        </tr>
      </thead>
      <tbody>
        @foreach (var entry in eventList) {
          <tr>
            <td>@entry.Subject</td>
            <td>@entry.Start.DateTime.ToString()</td>
            <td>@entry.End.DateTime.ToString()</td>
          </tr>
        }
      </tbody>
    </table>
  </Authorized>
  <NotAuthorized>
    <p>For full functionality please log in</p>
    <a href="MicrosoftIdentity/Account/SignIn">
      Log in
    </a>
  </NotAuthorized>
</AuthorizeView>
```

We wrap it inside AuthorizeView to avoid any errors arising from not being logged in – if you don't log in, you're not getting any data, so it's not risky in that sense to skip it, but we like messages making sense for the user instead of things not working.

12. Running the app and manually appending /Calendar to the URL, you should see a list of entries as shown in figure below:



Note that it is common when running in debug mode that you may have to log out and back in again for things to work properly when working with tokens. This can be caused by the browser storing a session while the token cache is emptied between runs (when using the in-memory cache).

We've come a long way, but there are still a few things to look at, such as expanding beyond your current AAD tenant.

Working with federated identity

Since you integrated with a specific AAD tenant assigned to you, it's easy to perceive it as your identity provider. Microsoft operates on a larger scale though, and on a technical level, you are federating with an external identity provider.

So, what does this actually mean?

Going back to our initial example from the real world, you could say that a passport is an example of federated identity. Even if you are not the entity issuing passports, you trust that there is a good procedure in place by the issuing authority and you accept it as proof of identity. You could choose to not trust this identity and build your own system for verifying that people are who they say they are, but it would most likely be timeconsuming and expensive if you even managed to provide the same level of authenticity. How much of a hassle it is to order a passport in different countries probably varies, but just imagine how unfriendly it would be as a traveler to acquire multiple passports in the different countries you traveled to.

In the past couple of years, you have most likely seen an option for logging in with Facebook or Google on a website you've visited. Instead of creating a new account, you can click these buttons and as long as you accept that the website is able to read some of your identity attributes, you're good to go. Sure, these providers probably have a lower level of trust than a federal entity in your own country, but odds are they have invested a decent amount of effort into making sure their user account database is secure and not too easily hackable. And for you, as the user, they save you from the effort of coming up with yet another password to remember.

Both passports and Google accounts are examples of federated identity. While your application might have a user database for access and licensing purposes, you only have a reference to their identity since that is provided by someone else that you trust to provide authentication services.

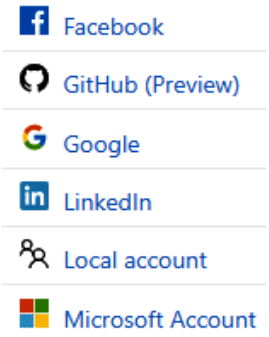
What happens on a high level is that you create an account for the application in a control pane for your chosen identity provider, where you provide a couple of relevant attributes, and correspondingly, you configure metadata as in the previous section, pointing to the identity provider.

.NET 5 and ASP.NET Core 5 provides libraries provides libraries for assisting you with this, and it's not necessarily hard to do by itself. However, what happens during the life cycle of your app is that you start with Google and Facebook and it's working. Then, someone asks you to add Apple to make it easier for iOS users. And then you add a provider that uses ''last name'' instead of ''surname,'' breaking your data model. Even if your response is that you love a challenge, it could be that this is causing friction as your login code gets bloated as you start adding more and more logic to handle it that requires new builds and releases.

As you might be able to guess, this leads to the inevitable There's an Azure service for that. There is a version of AAD called AAD B2C, which is designed to handle such scenarios. The B2C part stands for

business to consumer, but it's really about external identities in general. The way it works is that you set up a nested federation where your app trusts AAD B2C, and AAD B2C in turn trusts other identity providers. If you need to add a new provider or customize claims, you can do so in Azure without recompiling your app.

There are actually two types of user accounts in AAD B2C: local and social. Social is another term for federated in this context as it doesn't have to be an account on a social network per se. The beauty is that there are several providers pre-created that can be easily added by stepping through a wizard as you can see in figure below:



If your provider is not on the list, you can add generic OIDC providers. If you want a non-standard configuration, you can even add a non-B2C AAD tenant as an identity provider.

The local account does not federate to other providers but is instead a specialized version of AAD for adding individual accounts with any email address. A regular AD tenant is usually an organization where it's normal that users can look up the details of other users, be parts of groups, and so on. In a B2C tenant, each user is an island and cannot see other users. If you remember back to the sample where we created local accounts in the form of a database, you could say that this competes with that, but it's both ways more powerful and, in most instances, easier to use than maintaining your own database.

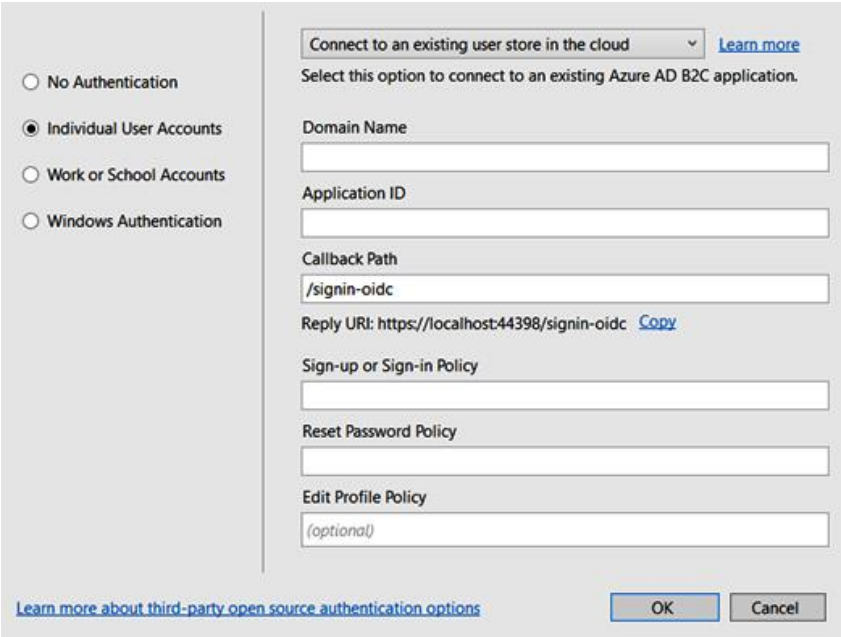
Different types of user journeys (sign up, sign in, password reset) can be configured through wizards, and you can also replace the styling if you so wish.

If you want to go deeper, there's also the option to use custom policies, which entails diving into XML files for a coding-like experience. It offers great flexibility with the option to call into backend APIs during the flows and more. Be warned that this can be quite the opposite of user-friendly, so only use it if the wizard-driven policies don't cover your use case.

While AAD B2C has a different feature set than regular AAD, the endpoints used for acquiring a token are also compliant with standards, so it's a fairly easy job to adapt your code.

In a basic form, you can actually use the same code as we used for authenticating with regular AAD, and change appsettings.json to point to a B2C tenant with attributes created in said tenant. This will actually work nicely if you only have one flow defined that handle signing up and signing in. It will not work if you also want to provide options, such as password reset and profile editing.

The recommended way to get started before you have a full overview of the AAD B2C service is having Visual Studio generate things for you, by opting to use B2C as the provider when choosing the authentication configuration during project creation in Visual Studio. The choices can be found under Individual User Accounts and Connect to an existing user store in the cloud as shown in figure below:



At first glance, it might appear like AAD B2C adds complexity for unclear benefits since these things can be achieved directly in the code. To be clear – like so many other things, there are good use cases and there are less-good use cases. The great thing is that it will require very few changes to the code, should you want to use B2C, and most of the work in AAD B2C can be ''outsourced'' to identity pros.

A note on UIs for identity

Whether you write your own identity implementation from scratch or rely on AAD, you need a UI if the user is to type in a username and password. In general, there are three different approaches to implementing this:

- **Popups:** You can break out a separate smaller window for the user to type in credentials. Once they've been verified, the popups disappear and you're back in the web app. There's nothing wrong with this method from a technical perspective, but a lot of users have popups blocked in their browser and many perceive it as an annoying UI.
- **Redirects:** The method we implemented when integrating with AAD was based on redirects. You start at https://localhost, you get sent to https:// microsoftonline.com, and then back to https://localhost again. This is a very common approach. It is easy to implement and supports the flows we have described in a secure manner.
- **Iframe:** The sleekest method is probably to embed the login form as part of the web app and keep the user in the same context. To make this work, you need to do some tricks on the backend with cookies and sessions. This is not a problem when you control everything, but it becomes a problem if you want to use federated identities. Single-tenant AAD could in theory support Iframe, but doesn't do so at the time of preparing this training. Providers, such as Facebook and Google, do not support it, due to security implications – for instance, creating login experiences intended for harvesting passwords. In addition, the major browsers are implementing more mechanisms for blocking third-party cookies to ensure privacy, so it may be blocked there as well. Make sure you are on top of all the moving parts before attempting to implement this UI.

Summary

This module took us on a journey from basic auth to federated identities. It started with explaining what authentication and authorization are all about. There were details, such as understanding what Base64 encoding and hashing are good for. The sample implementations of AuthN and AuthZ intended to give you a better understanding of what's going on, even though you will probably not implement or use all of these techniques. The walkthrough of OAuth and introducing AAD should put you in a good position to implement production-grade identity in your web apps.

Not every app needs to be super secure, but this should have set you up for web apps that can be more personal than treating all visitors as anonymous users.

With identity covered, the next module will dive into another hot topic these days, as we cover the ins and outs of working with containers.