

# Security in Distributed Systems

## Introduction

In this laboratory exercise, we will briefly look at the basic .NET API for *hashing* and *encryption*.

Then we will develop a basic ASP.NET Core MVC website using OAuth2 to remotely authenticate our users.

After completion of this tutorial, you will be able to use `System.Security.Cryptography` for *data encryption* and *integrity checking* and you should have a clearer understanding of the OAuth flow and how to implement it in ASP.NET Core.

In the first exercise, we will use the core .NET API to compute hashes of simple strings and illustrate their user-readable form as hexadecimal strings. Cryptographic hashes are commonly used to *secure passwords* sent over the network, as well as for storing passwords securely in databases. These hashes are often computed in the *salted* form, which means that random information is added as an input to the cryptographic hash function so that it is not easy to guess the password based on the hash, even if the password is common (this is covered in the lectures).

Also, you will learn to use .NET API to encrypt a message using the *RSA cryptosystem*, decrypt it to its original form and understand the difference between the *private key* and the *public key* in *asymmetric cryptography*.

The second exercise is aimed at usage of ASP.NET Core to perform OAuth. We will be authenticating a user without needing to manage their login or store their password/personal profile information/etc. The key learning from this work is to understand the OAuth flow and consider how it can be applied to improve user confidence and maximise the ease-of-use of our services.

**You should expect to continue to read, learn about, work on and experiment with lab/lecture content in your own time.**

## Exercise 1: Hashing

We start with the use of a cryptographic hash function. Such a function computes a short authenticator of a message based on its content. These functions are usually used for integrity checking, i.e. if a message is sent along with its hash, the recipient can *verify* whether the data arrived correctly and was not modified on its way. This is achieved simply by comparing the expected hash obtained from the sender with the hash computed from the data which arrived.



Create a new .NET Core C# console application in Visual Studio. We will now use the cryptographic API of .NET to encrypt and decrypt data.

Create a `string` message and assign a string value to it. Make sure that you are...

```
using System.Security.Cryptography;
```

The cryptographic API operates on sequences of bytes, i.e. we will have to make sure that our data is converted to `byte[]` first. This can be done in the following way (where Message is a string):

```
byte[] asciiByteMessage = System.Text.Encoding.ASCII.GetBytes(message);
```

Note that this transformation requires specification of the encoding of the string (you may experiment with trying Unicode instead of ASCII). After that, we can use the SHA1 Cryptography Service Provider to compute the SHA-1 hash of our message.

```
byte[] sha1ByteMessage;
using (SHA1 sha1Provider = SHA1.Create())
{
    sha1ByteMessage = sha1Provider.ComputeHash(asciiByteMessage);
}
```

Since the output of this cryptographic hash function is of type `byte[]`, this byte sequence is usually expressed as a sequence of hexadecimal digits, in which a pair of digits represents one byte. To compute such a string, we can use a method such as the following one:

```
static string ByteArrayToHexString(byte[] byteArray)
{
    string hexString = "";
    if (null != byteArray)
    {
        foreach (byte b in byteArray)
        {
            hexString += b.ToString("x2");
        }
    }
    return hexString;
}
```

Or we can use the `BitConverter.ToString(Byte[])` Method:

<https://docs.microsoft.com/en-us/dotnet/api/system.bitconverter.tostring>

but note that this adds a '-' char between each hex value.



Display the corresponding hexadecimal representation for your Message.

For example the SHA-1 hash of the string “hello world” is:  
2aae6c35c94fcfb415dbe95f408b9ce91ee846ed

You can check your results by searching for some common SHA-1 hashes on the Web. Since these functions are widely used, there are online tools available for computing of these hashes.



Experiment with changing your message slightly (replacing one character, adding / removing punctuation). What can you observe?

Add a few bytes to the input as a cryptographic *salt*. Apply salts to a few different messages. Encrypt the same message several times using the same salt value.

What do you observe?

You should see that when you add some salt, the hash changes from the hash that was calculated without the salt. This is useful. However, all minor changes or applications of salt also change the hash such that it is totally impossible to identify that the main message is the same.

You should also notice that the same message hashed with the same salt does produce the same hash. The salt is usually unique to the hash it was used to produce. It is often stored alongside the hash it was used to produce in the password database table so that, when the password is provided (e.g. for login), the salt can be added before it is hashed to check that the password matches. This is useful but also could be a security concern... Can you think of any reason(s) why?

Here is one: lots of people STILL use the password “password” for their password. In systems which don’t prevent it, it is still the MOST USED password. Therefore, if you get hold of the database, it doesn’t take much to apply the salt to “password” (and lots of other common passwords) and then hash that to see if it matches.



Try to use [SHA256](#) to compute the SHA-256 hash instead. Observe the lengths of the obtained hexadecimal strings.

Revise the concept of pepper and consider how it differs from salt, and the security implications of this.

Hashes are only one-way transformations. They cannot be ‘decrypted’. To encrypt a message and keep the information secret, or to secure the message before sending it over the network, as well as to decrypt it, we have to use *symmetric* or *asymmetric cryptography*.

## Exercise 2: Asymmetric Cryptography

In the following, we will demonstrate the use of .NET API to perform *encryption* and *decryption* using the *asymmetric algorithm* RSA. This cryptosystem is one of the most well-known methods used to perform key exchange in the beginning of the Transport Layer Security (TLS) handshake, which is one of the most popular methods of confidentiality assurance implemented in distributed applications. (The other algorithm commonly used for this handshake is ECDSA)



Use the following code demonstrating the use of **RSA** in your console application:

```
Console.Write("Plaintext message: ");
Console.WriteLine(message);

byte[] encryptedBytes;
byte[] decryptedBytes;
using (RSA rsaProvider = RSA.Create())
{
    // Encrypt
    encryptedBytes = rsaProvider.Encrypt(asciiByteMessage, RSAEncryptionPadding.OaepSHA1);
    Console.Write("Encrypted message: ");
    Console.WriteLine(ByteArrayToHexString(encryptedBytes));

    // Decrypt
    decryptedBytes = rsaProvider.Decrypt(encryptedBytes, RSAEncryptionPadding.OaepSHA1);
    Console.Write("Decrypted message: ");
    Console.WriteLine(System.Text.Encoding.ASCII.GetString(decryptedBytes));
}
```

This code initialises the **RSA** Crypto Service Provider for use in encryption and decryption of the binary representation of our Message. The displayed decrypted message should be equal to the original plaintext message.



Run the code a few times.

What do you observe about the encrypted message?

You should see that the encrypted message changes each time. This is not great because it would mean that the public and private key pairs are changing each time we run the code. That's fine for our local code but, if we were interacting over a network, we would have to re-issue the public key each time we restarted. Again, that's not awful (it's good to renew the keys periodically) but it's something to keep in mind. See the lectures to explore the use of and challenges with CspParameters.

The critical thing you may have noticed is that we are using the same RSA implementation for encryption and decryption, so we have not had to share the public key ever!

This is not very realistic.



Modify the code into two using blocks:

```
using (RSA rsaEncrypt = RSA.Create())
{
    // Encrypt
    encryptedBytes = rsaEncrypt.Encrypt(asciiByteMessage, RSAEncryptionPadding.OaepSHA1);
    Console.Write("Encrypted message: ");
    Console.WriteLine(ByteArrayToHexString(encryptedBytes));
}
using (RSA rsaDecrypt = RSA.Create())
{
    // Decrypt
    decryptedBytes = rsaDecrypt.Decrypt(encryptedBytes, RSAEncryptionPadding.OaepSHA1);
    Console.Write("Decrypted message: ");
    Console.WriteLine(System.Text.Encoding.ASCII.GetString(decryptedBytes));
}
```

Your code should now be using different instance of the RSA implementation for encryption and decryption. This mirrors doing the work on two separate devices (without all the mess of actually sending data over the network).



Run the code.  
What happens?

Your code should have thrown an exception. This is a useful exception as it identifies that the cryptographic function failed. There are a number of reasons why it might have failed but one of the most common is that *the public key used to encrypt the data is not linked to the private key used to decrypt it*.

The main exception that is likely to be thrown is a [CryptographicException](#).



Extend the code suitably by *try* and *catch* to avoid exceptions being unhandled.  
Display a useful message to the user.

Ok, great. Now we have a helpful error but we still are not decrypting our message.

Observe that our RSA instances contain `ToXmlString` and `FromXmlString` member methods.  
E.g.

```
rsaDecrypt.ToXmlString(bool)  
rsaEncrypt.FromXmlString(string);
```

The first of these two calls exports the *keys* generated for us by the .NET API into XML and returns this as a string, while the second call consumes the keys from an XML string to use in the algorithm.

Remember – only the public key is needed to encrypt our Message. However, we need the private key to decrypt it. This is used practically in the TLS handshake, in which the certificate sent to the client by the server contains the public key, while the server keeps its private key secret.

To ensure that only the public key is exported we set the `includePrivateParameters` bool to **false**. If you set this to true and send the keys out to an external service you have exposed the private key and undermined the encryption.



Add sharing of the parameters to ensure `rsaEncrypt` has the public key but not the private key and `rsaDecrypt` has both the public and private key.  
  
You will probably need to create an instance of RSA before you do any encryption/decryption and export it's parameters to two different strings (one with and one without the private key).

Start the application. You should be able to see the original Message as the outcome of encryption and consequent decryption.

Put breakpoints into your code to examine the process.

### Exercise 3: OAuth2 in ASP.NET Core

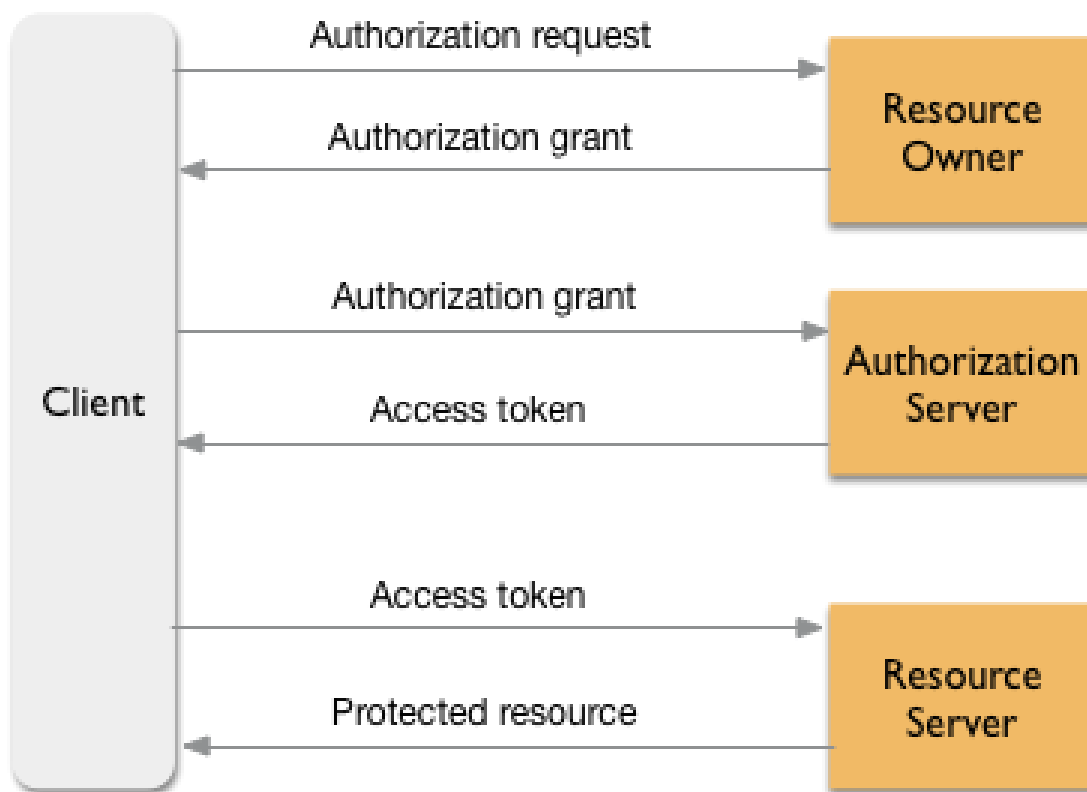
OAuth2 allows us to offer sign-in with credentials from external authentication providers.

That means that:

- Our users don't need to make ANOTHER user account with a password (that they'll forget) just to use our services.
- We can offload the responsibility for keeping passwords safe to the external provider.

It tends to be a win-win!

The OAuth flow looks like this:



We are going to be using your personal Microsoft account for this lab but, if you don't feel comfortable with that, you are under no obligation to do so. If you would prefer to not use your account I would recommend that you look through the steps, read about OAuth and perhaps work with someone else if they are happy to collaborate.

You should already have a personal Microsoft account which you use to get Microsoft Azure Dev Tools For Teaching (where you get the latest version of Visual Studio, for example).



Open Visual Studio choose the template: *ASP.NET Core Web App (Model-View-Controller)* call your new project **OAuthLab**

In the next window ensure .NET 8.0 LTS is selected

Untick 'Configure for HTTPS' and click Create

Once created, open 'Properties' in the Solution Explorer Window and open the launchSettings.json file. Note down the iisExpress port shown in applicationUrl



We are going to use Microsoft as our OAuth provider. Now you'll need a Microsoft Account registered to use Azure.

As suggested above, you probably already have a Microsoft Azure account to download your free student software. If you don't, you can make one now – it's free and you get free software as a student!



Go to <https://go.microsoft.com/fwlink/?linkid=2083908> and log in (or register)

*It is highly likely that it will automatically log you in using your University credentials. If it does, log out of this account and log back in with your personal account. **YOU MUST USE A PERSONAL ACCOUNT. YOUR UNIVERSITY ONE WILL NOT WORK FOR THIS***


Once logged in, you should see something like this:

### App registrations ...

[+ New registration](#) [Endpoints](#) [Troubleshooting](#) [Refresh](#) [Download](#) [Preview features](#) | [Got feedback?](#)

 Starting June 30th, 2020 we will no longer add any new features to Azure Active Directory Authentication Library (ADAL) and Azure AD Graph. We will continue to provide technical support and security updates but we will not add new features. [Learn more](#)

[All applications](#) [Owned applications](#) [Deleted applications](#) [Applications from personal account](#)

 Start typing a display name or application (client) ID to filter these results

 Add filters

This account isn't listed as an owner of any applications in this directory.

[View all applications in the directory](#)

[View all applications from personal account](#)



Click 'New Registration' in the top left.



As Name, insert: **DistSysOAuthLab**

Choose: '**Accounts in any organizational directory and Personal Microsoft Accounts**'

In Redirect URI choose **Web** and insert: [http://localhost:<](http://localhost:<port>) where **port** is the port number you got from the launchSetting.json file above.

Click Register

A new page should open. You have successfully made an App Registration!

Now we need to make some secret codes to identify our app.



Click Manage->Certificates & secrets in the bar to the left.

Under Client secrets, click 'New client secret'

In description insert: **Distributed Systems OAuth Lab Client**

Press Add.

Copy the string under **Value**



Return to your ASP.NET code but keep Azure open.



Right-click the project in the Solution Explorer and click 'Manage NuGet Packages'

In the window that opens, select Browse and type *MicrosoftAccount* into the search bar.

Select **Microsoft.AspNetCore.Authentication.MicrosoftAccount**

On the right, select version 8.0.12 and click install.

In the Nuget window, select Browse and type *JwtBearer* into the search bar.

Select and install the 8.0.12 version of **Microsoft.AspNetCore.Authentication.JwtBearer**

These libraries allow us to use the Microsoft OAuthprovider directly in ASP.NET.

#### Build the project

Right-click the project in the Solution Explorer and choose 'Manage User Secrets'

In the JSON file that opens, between the brackets type:

`"Authentication:Microsoft:ClientSecret": "<Client-Secret>",`

Where <Client-Secret> is the Client Secret (Value) from Azure you just copied

Below this line (still inside the brackets) add:

`"Authentication:Microsoft:ClientId": "<Application-Client-Id>"`

Return to Azure, select Overview from the left and find **Application (client) ID** under Essentials. Copy this string. Replace `"<Application-Client-Id>"` with the ID you just copied



What you've just done is stored these in the Secret Manager. You can read more about that here:

<https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets>

The next thing to do is add Authentication Options. ASP.NET Core will manage a huge number of different authentication options for us because there are a number of common and standardised techniques. This means that most of the OAuth flow is managed for us under the hood.

We can formulate the HTTP requests ourselves if we need to – so you could do this all manually in another client application by using an HttpClient if you wanted.

The flow and HTTP requests that are being generated on our behalf are detailed here:

<https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-oauth2-auth-code-flow>

We will be using JWT Bearer Tokens stored as a cookie to keep the user logged in with their Microsoft Identity in this lab. You can read all about them here: <https://jwt.io/introduction/>

It's overkill for our purposes but JWT are becoming industry standard, so useful to know about.



Open program.cs

Under `app.UseAuthorization();` in the pipeline configuration section, add:  
`app.UseAuthentication();`

This adds the .NET Core authentication middleware to the pipeline.

In the service configuration area, above `builder.Services.AddControllersWithViews();` add the line:

```
builder.Services.AddAuthentication(options => {  
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;  
    options.DefaultSignInScheme = JwtBearerDefaults.AuthenticationScheme;  
});
```

This sets up defaults for how we are going to Authenticate and Sign-in users (using JWT tokens). You will see some errors for a while because we haven't yet finished this code.

Next we have to chain some specific calls identifying how we are going to store the token, authenticate the user and request specific information about the user.



Following on from the last piece of code, add the chained call:

`.AddCookie("Bearer")`

With the chained call Your code should now look like this:

```
Builder.Services.AddAuthentication(options =>  
{  
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;  
    options.DefaultSignInScheme = JwtBearerDefaults.AuthenticationScheme;  
}).AddCookie("Bearer")
```

This identifies that we are going to add a cookie to the user's browser that stores the bearer token for us. This token will allow us to keep the user logged in whilst remaining stateless.

**Note:** In a production environment it would be essential that we connect via HTTPS to prevent anyone else from accessing this token when it is sent to the client as a cookie. If someone else got hold of it they could gain access to the account (until the session timeout).



Following on from the last piece of code, add the chained call:

```
.AddOAuth("Microsoft", options =>
{
    //TODO: Add options
});
```

This adds a new OAuth Authentication Builder, in which we will define how we are going to use OAuth and what we are going to use it for.

**Note:** There is an inbuilt Microsoft Authentication Provider that does most of this for us... but it doesn't show what's happening so configuring our own helps us to see the steps required for OAuth.



Inside the curly braces, replace the TODO with the lines:

```
options.ClientId = builder.Configuration["Authentication:Microsoft:ClientId"];
options.ClientSecret = builder.Configuration["Authentication:Microsoft:ClientSecret"];
options.CallbackPath = new PathString("/signin");
```

This takes the ClientId and ClientSecret we put in the secret store and will use these to make our OAuth calls.

We are also setting a Callback Path – the path the user will be sent to when they have successfully logged in or when an authentication error occurs.

Underneath the Callback path, add the lines:

```
options.AuthorizationEndpoint = "https://login.microsoftonline.com/common/oauth2/v2.0/authorize";
options.TokenEndpoint = "https://login.microsoftonline.com/common/oauth2/v2.0/token";
options.UserInfoEndpoint = "https://graph.microsoft.com/v1.0/me";
options.Scope.Add("user.read");
```

Remember the OAuth Flow? First we need to perform the authorisation and get a **grant**. That's where the first endpoint address comes in. Then we need to swap this **grant** for a **token**. That's handled by the second endpoint. Finally, we use the **token** to actually get user information from Microsoft – that's the third endpoint.

We are also specifying what data we want to access (the scope). See:

<https://docs.microsoft.com/en-us/graph/permissions-reference#user-permissions>



Continuing on inside the curly braces of the options parameter, add the lines:

```
options.ClaimActions.MapJsonKey(ClaimTypes.Name, "displayName");
options.CorrelationCookie.SameSite = SameSiteMode.Lax;
```

All clients have an Identity. This Identity is made up of Claims. In this case we are instructing the framework to add the value that is referred to as 'displayName' as a Claim in our current Identity. We are also clarifying that this Claim is a Name. Then we are setting the SameSite policy to tell the browser when to send this cookie to the server.



Finally, following on from the line above, add the lines:

```
options.Events.OnCreatingTicket = async ctx =>
{
    var request = new HttpRequestMessage
        (HttpMethod.Get, ctx.Options.UserInformationEndpoint);

    request.Headers.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/json"));

    request.Headers.Authorization = new AuthenticationHeaderValue(
        "Bearer", ctx.AccessToken);

    var response = await ctx.Backchannel.SendAsync(
        request,
        HttpCompletionOption.ResponseHeadersRead,
        ctx.HttpContext.RequestAborted
    );

    var user = JsonDocument.Parse( await
        response.Content.ReadAsStringAsync());

    ctx.RunClaimActions(user.RootElement);
};
```

This code manages the last part of our OAuth flow. There are a number of events that are called by the framework during OAuth (see: <https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.authentication.oauth.oauthevents>)

We run this code the framework is creating the 'ticket' – this ticket is how the framework can continue to authorise us using just the cookie.

The code creates an HTTP Request message and sends it with the token in the header to the `UserInformationEndpoint` to get the actual protected resource (user information). When the information is received it is formatted as JSON and, finally, the JSON is sent for processing by our Claim action where it will extract the `displayName` for our use.

Phew. Our OAuth flow is now finished. But we still have no actual log-in functionality!



First let's add a Controller to manage the endpoints. One will forward us to Microsoft to log in, the other will be the location that we are redirected to by Microsoft once the user is logged in.



Right-click on the Controllers folder and select Add -> Controller.

Choose MVC Controller - Empty

Name your Controller `SignInController`

It will come pre-configured with an `Index()` method. We can leave that as it is.



Ensure your Controller has these using statements:

```
using Microsoft.AspNetCore.Authentication;  
using Microsoft.AspNetCore.Mvc;
```

Insert a new method into the Controller:

```
public IActionResult Login()  
{  
}
```

This Login action will perform the Challenge to request a user signs in with their Microsoft identity.

Inside the Login method add:

```
return Challenge(new AuthenticationProperties() {  
    RedirectUri = "/signin/index"  
}, "Microsoft");
```

The challenge uses the Microsoft OAuth provider we just set up and directs all returning traffic to the `"/signin/index"` page.

That is the back-end done. It's as easy as that!

Now let's create the actual signin/index view.



Right-click on the Views folder and Add -> New Folder

Call the new folder **Signin**

Right-click on your new Signin folder and Add -> View. Choose **Razor View**

View name: Index

Ensure Template is Empty (we aren't going to use a view model for this today)

Select Use a layout page and choose `~/Views/Shared/_Layout.cshtml`

Press Add

Once it is created, below `<h1>Index</h1>` add the Razor and HTML:

```
<div>  
    @if (User.Identity.IsAuthenticated)  
    {  
        <h4>Welcome @User.Identity.Name</h4>  
    }  
    else  
    {  
        <h4>You are not logged in!</h4>  
    }  
</div>
```

Finally, let's add a log in button to our home page that starts it all off



Open Views->Home -> Index.cshtml

Immediately after the h1 Welcome element add the markup:

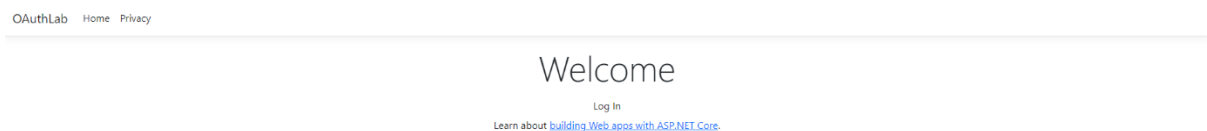
```
<div class="row">
  <div class="col-md-12">
    <a asp-action="Login" asp-controller="Signin" class="btn btn-default">Log In</a>
  </div>
</div>
```

This will create a new button on your website homepage that triggers your Login action inside the Signin controller. That method will then set in motion the OAuth flow culminating in your user being forwarded to the signin/index page where you HOPEFULLY will see a username from your authenticated user.

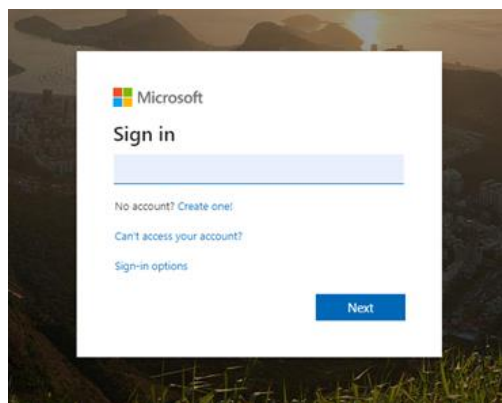


Run the software using IIS Express.

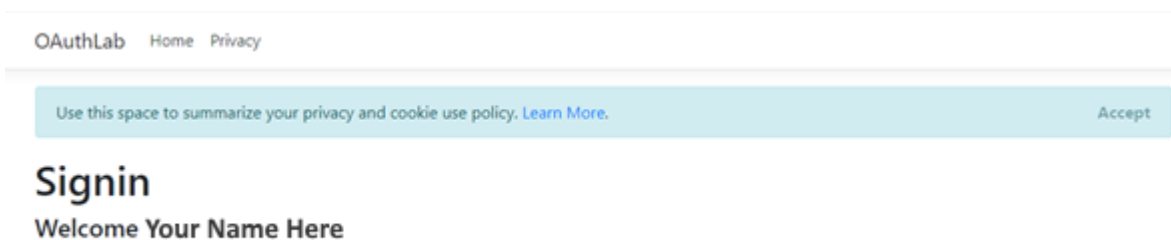
You should (eventually) be looking at something like this:



Click Log In and you should be routed to Microsoft Sign-In.



Sign in using your **Personal** Microsoft Credentials and you should be returned back to ~/signin/index where you should see something like this...



You have successfully authenticated a user using Microsoft's Authorisation endpoint and OAuth2!

Try:

- Click Home then navigate back to ~/signin/index in your browser address bar.  
You are still logged in!
- Now press F12 in your browser to bring up the Developer Window.  
In Chrome, press 'Application'. Open up 'Cookies' and click your web URL.  
See the cookie? That's what is keeping you logged in but because the client is managing it, your server is still stateless.
- Remove the cookie by clicking 'Clear Storage' and 'Clear site data' then navigate back to ~/signin/index in your browser address bar.  
You are no longer logged in because you no longer have a cookie to authenticate yourself.

**OAuth is tricky!** One of the reasons for this is that the flow **must** be followed in order to ensure security of client applications and user details.

If the correct OAuth sequence is not working properly, it's likely your log-in will fail. If it isn't working for you go back and check you followed every step exactly.

This certainly isn't the only way to get this working, but it does introduce some key themes and concepts.

In this lab I have noted that there is [an inbuilt Microsoft Authentication Provider that does most of this for us](#). This is true. It hides most of the OAuth logic and sequence, which makes life much easier!... that is, it makes life easier when we use Microsoft Authentication. If we want to use another OAuth provider, we still need to know how to do it manually and we wouldn't be learning anything about the flow...

So take this time to examine the sequence of events in the code!

### Finishing up the lab:

Next Steps:

- Try mapping some other claims.
  - How about mapping email address?
- Try adding some extra scopes to your authentication and using them when logged in to show more information about the user.  
For this you will need to:
  - Add new scopes to `options.Scope`
  - Add new claim actions with `options.ClaimActions`
  - Change the `UserInformationEndpoint` or add additional requests for protected resources inside your `OnTicketCreating` handler.
- Add a "Logged in as: " message to the homepage that only appears when your user is already logged in.
- Explore some other hashing and encryption methods in .NET.