Laboratory 6: 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 may find it difficult to complete this lab in an hour so should expect to continue to read, learn about, work on and experiment with this content in your own time.

Exercise 1: API for Hashing and Asymmetric Cryptography

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.

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 it 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 string Message and assign a message 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 `SHA1CryptoServiceProvider` to compute the SHA-1 hash of our message.

```
byte[] sha1ByteMessage;
SHA1 sha1Provider = new SHA1CryptoServiceProvider();
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.

- 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. For example the SHA-1 hash of the string “hello world” is: `2aae6c35c94fcfb415dbe95f408b9ce91ee846ed`.
- Experiment with changing your message slightly (replacing one character, adding / removing punctuation). What can you observe?
- Add a few random bytes to the input as a cryptographic *salt*. Encrypt the same message several times using different salt values. What can you observe?
- Try to use `SHA256CryptoServiceProvider` to compute the SHA-256 hash instead. Observe the lengths of the obtained hexadecimal strings.

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 ciphers*.

In the following, we will demonstrate the use of .NET API to perform *encryption* and *decryption* using *asymmetric cipher RSA*. This cipher is one of the most well-known methods used to perform key exchange in the beginning of Transport Layer Security (TLS) handshake, which is one of the most popular methods of confidentiality implementation in distributed applications.

Use the following code demonstrating the use of `RSACryptoServiceProvider` in your console application (e.g. in the *Main* method):

```
byte[] encryptedByteMessage;
byte[] decryptedByteMessage;
RSACryptoServiceProvider RSA = new RSACryptoServiceProvider();
{
 encryptedByteMessage = RSAEncrypt(asciiByteMessage, RSA.ExportParameters(false));
 Console.Write("Encrypted message: ");
 Console.WriteLine(ByteArrayToHexString(encryptedByteMessage));
 decryptedByteMessage = RSADecrypt(encryptedByteMessage, RSA.ExportParameters(true));
 Console.Write("Decrypted message: ");
 Console.WriteLine(System.Text.Encoding.ASCII.GetString(decryptedByteMessage));
}
```

This code initialises the `RSACryptoServiceProvider` for use in encryption and decryption of the binary representation of our Message. The displayed decrypted message should be equal to the original Message.

Note that the cryptographic API may fail, i.e. extend this code suitably by *try* and *catch* to avoid an exception being unhandled. Also, observe that `RSA.ExportParameters(false)` is used for encryption and `RSA.ExportParameters(true)` is used for decryption of our Message. In fact, the first of these two calls exports the *public key* generated for us by the .NET API, while the second call includes both the *public and the private key*. Only public key is needed to encrypt our Message. However, we also 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.

Encryption and decryption can be performed using the following methods[1].

```
static public byte[] RSAEncrypt(byte[] DataToEncrypt, RSAParameters RSAKeyInfo)
{
    try
    {
        byte[] encryptedData;
        using (RSACryptoServiceProvider RSA = new RSACryptoServiceProvider())
        {

            RSA.ImportParameters(RSAKeyInfo);
            encryptedData = RSA.Encrypt(DataToEncrypt, false);
        }
        return encryptedData;
    }
    catch (CryptographicException e)
    {
        Console.WriteLine(e.Message);
        return null;
    }
}
```

```
static public byte[] RSADecrypt(byte[] DataToDecrypt, RSAParameters RSAKeyInfo)
{
    try
    {
        byte[] decryptedData;
        using (RSACryptoServiceProvider RSA = new RSACryptoServiceProvider())
        {
            RSA.ImportParameters(RSAKeyInfo);
            decryptedData = RSA.Decrypt(DataToDecrypt, false);
        }
        return decryptedData;
    }
    catch (CryptographicException e)
    {
        Console.WriteLine(e.ToString());
        return null;
    }
}
```

Include all required methods and code fragments and arrange the code so that the application can be run and that the encryption / decryption functionality is called.

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

[1] https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.rsacryptoserviceprovider

Exercise 2: OAuth2 in ASP.NET Core

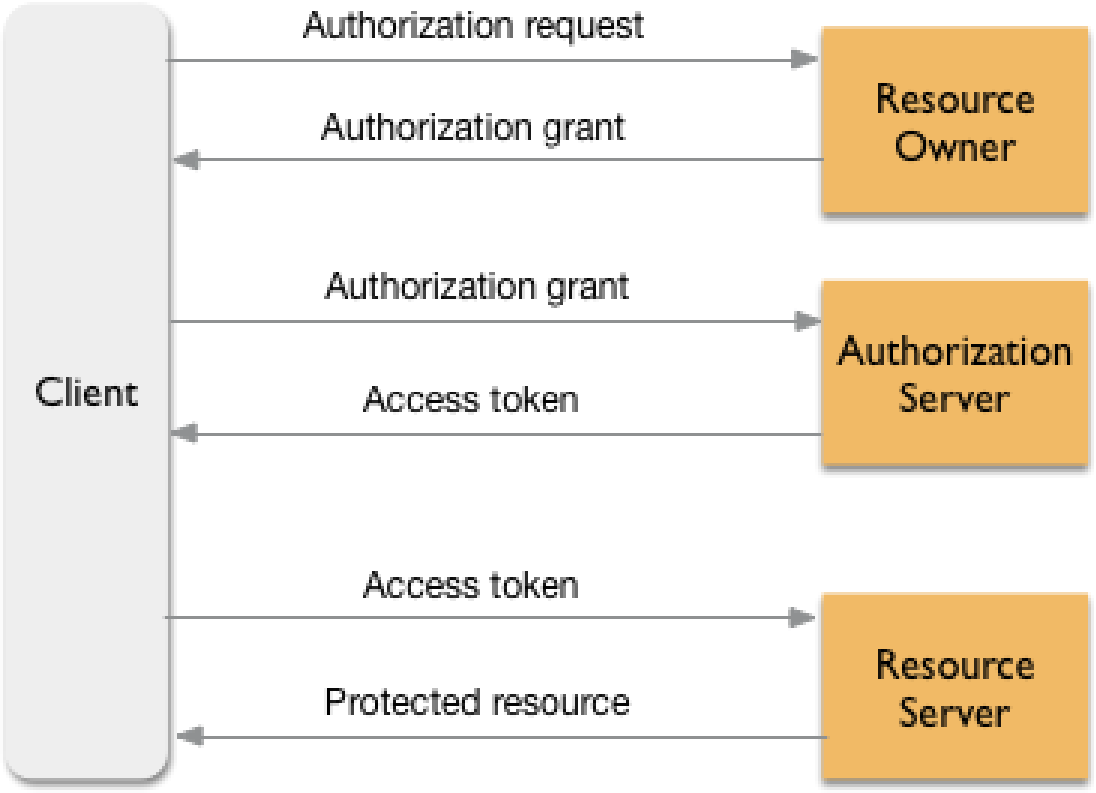
OAuth2 allows us to offer sign-in with credentials from external authentication providers.

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.

Open Visual Studio and create a new ASP.NET Core Web Application called **OAuthLab**

In the next window ensure .NET Core and ASP.NET Core 2.2 are selected

Select **Web Application (Model View Controller)**

Untick 'Configure for HTTPS' and click OK

Once created, open 'Properties' in the Solution Explorer Window and open the launchSettings.json file. Note down the IP address in applicationUrl

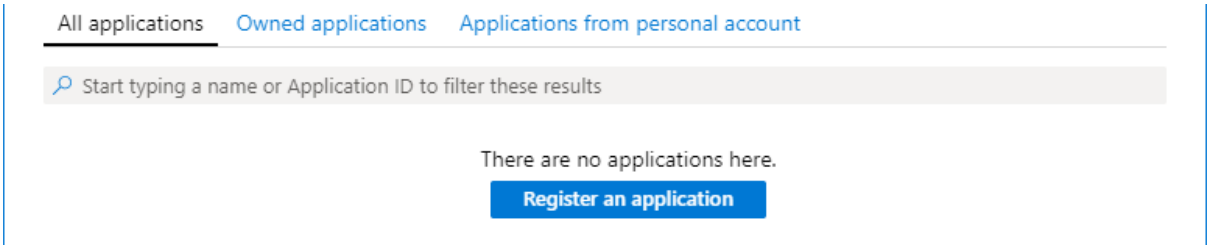
Done. Easy.

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

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)

Once logged in, you should see something like this:



Click 'Register an application'.

In the page that opens…

As Name, insert: **DistSysOAuthLab**

Choose: 'Accounts in any organizational directory'

In Redirect URI insert: http://localhost:<**port**>/signin 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 ‘Certificates & secrets’ in the bar to the left.

Under Client secrets, click ‘New client secret’

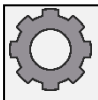
In description insert: **Distributed Systems OAuth Lab Client**

Press Add.

Click Overview on the left. Copy the string after Application (client) ID

We have an app ready to perform OAuth in Azure!

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 2.2.0 and click install.

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

Open the Package Manager Console in Visual Studio

Type:

```
dotnet user-secrets set Authentication:Microsoft:ClientId <Client-Id> --project OAuthLab
```

Where <Client-Id> is the ID you just copied

Return to Azure, Click ‘Certificates & secrets’ in the bar to the left. Copy the client secret value string that was just created.

Return to the Package Manager Console in Visual Studio

Type:

```
dotnet user-secrets set Authentication:Microsoft:ClientSecret <Client-Secret> --project OAuthLab
```

Where <Client-Secret> is the Secret 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?view=aspnetcore-2.2

If you right-click on your project in the Solution Explorer and click ‘Manage User Secrets’ you will see the secrets you just stored.

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 Startup.cs and ensure check you have all of these using statements.
Add any that are missing:

```
using System.Security.Claims;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Authentication.OAuth;
using System.Net.Http;
using System.Net.Http.Headers;
using Newtonsoft.Json.Linq;
using Microsoft.AspNetCore.Authentication.JwtBearer;
```

Now find the Configure method in Startup.cs

Above app.UseMvc() add the code:

```
app.UseAuthentication();
```

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

Next find the ConfigureServices method

Above services.AddMvc() add the line:

```
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:

```
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 oath 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 = Configuration["Authentication:Microsoft:ClientId"];
options.ClientSecret = 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.UserInformationEndpoint = "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");
```

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.

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

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

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

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

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

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

        context.RunClaimActions(user);
    }
};
```

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

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>Signin</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 Home -> Index.cshtml

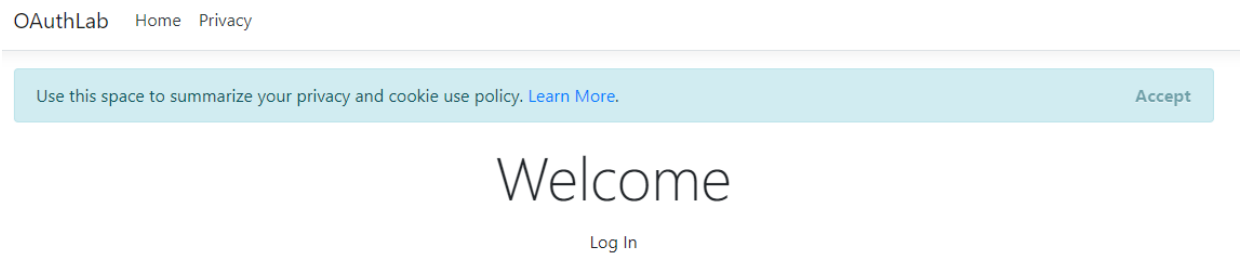
Immediately after the h1 Welcome element add the markup:

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

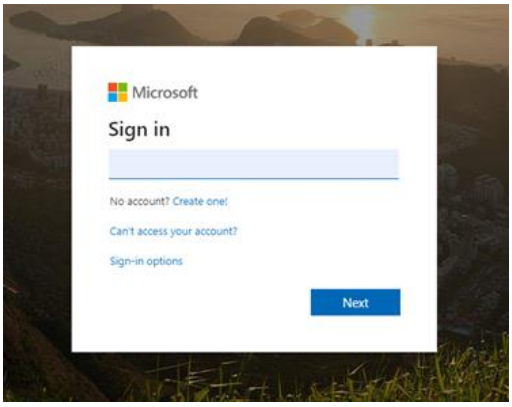
This will create a new button on your website homepage that triggers your Login action inside the Signin controller. That method will then set in motion the OAuth flow culminating in your user being forwarded to the signin/index page where you HOPEFULLY will see a username from your authenticated user.

Run the software by pressing F5.

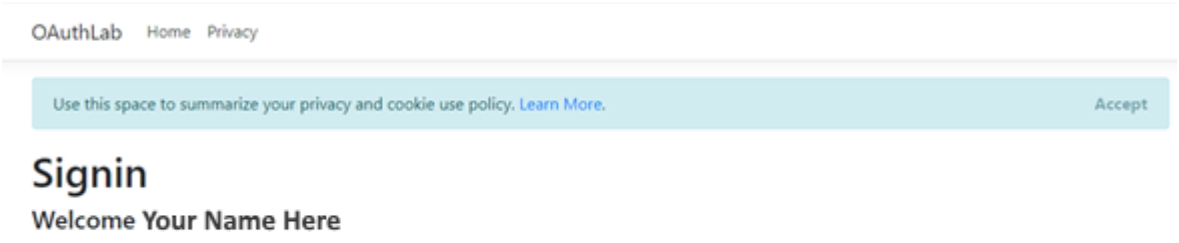
You should (eventually) be looking at something like this:



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



Sign in using your 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.

Next Steps:

- 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.