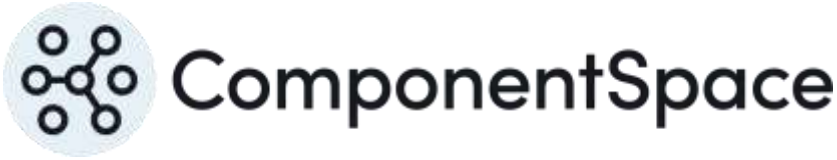


# ComponentSpace

# OpenID Connect for ASP.NET Core

# Examples Guide


www.componentspace.com

# Contents

Introduction ..... 1
Visual Studio Solution Files ..... 1
Setting the Startup Projects ..... 1
Example OpenID Provider ..... 2
Building and Running ..... 2
Example Blazor WASM OpenID Client ..... 2
Building and Running ..... 2
Authentication ..... 3
Authorization ..... 4
Logout ..... 5
Example ASP.NET Core OpenID Client ..... 6
Building and Running ..... 6
Authentication ..... 6
Authorization ..... 8
Logout ..... 8
SAML SSO and OpenID Connect ..... 9
Building and Running ..... 9
Authentication ..... 9
Logout ..... 13
OpenID Provider Conformance Testing ..... 14
Building and Running ..... 14
Code Walkthrough ..... 14
Example OpenID Provider ..... 14
Configuration ..... 14
Startup ..... 14
OpenIDController.GetMetadataAsync ..... 15
OpenIDController.GetKeysAsync ..... 15
OpenIDController.AuthorizeAsync ..... 15
OpenIDController.LogoutAsync ..... 18
OpenIDController.TokenAsync ..... 19
OpenIDController.UserInfoAsync ..... 20
SamlController ..... 20
Error Handling ..... 20

# Introduction

This document describes the example projects shipped with the product.

Refer to the OpenID Connect for ASP.NET Core Installation Guide for instructions on installing the product.

The example projects include OpenID configurations. Refer to the OpenID Connect for ASP.NET Core Configuration Guide for information on OpenID configuration.

The OpenID Connect for ASP.NET Core Developer Guide describes the OpenID APIs called by the example projects.

# Visual Studio Solution Files

Solution files for various .NET Core releases and versions of Visual Studio are included.

Select the appropriate solution file to open the example projects in Visual Studio.

No changes are required for the example projects to build cleanly and run without error in Visual Studio.

## Setting the Startup Projects

Open the Visual Studio solution properties to edit the start-up project to ensure the required projects are run.

For example, start the BlazorWASM, ExampleOpenIDClient and ExampleOpenIDProvider projects for OpenID Connect authentication between these applications.

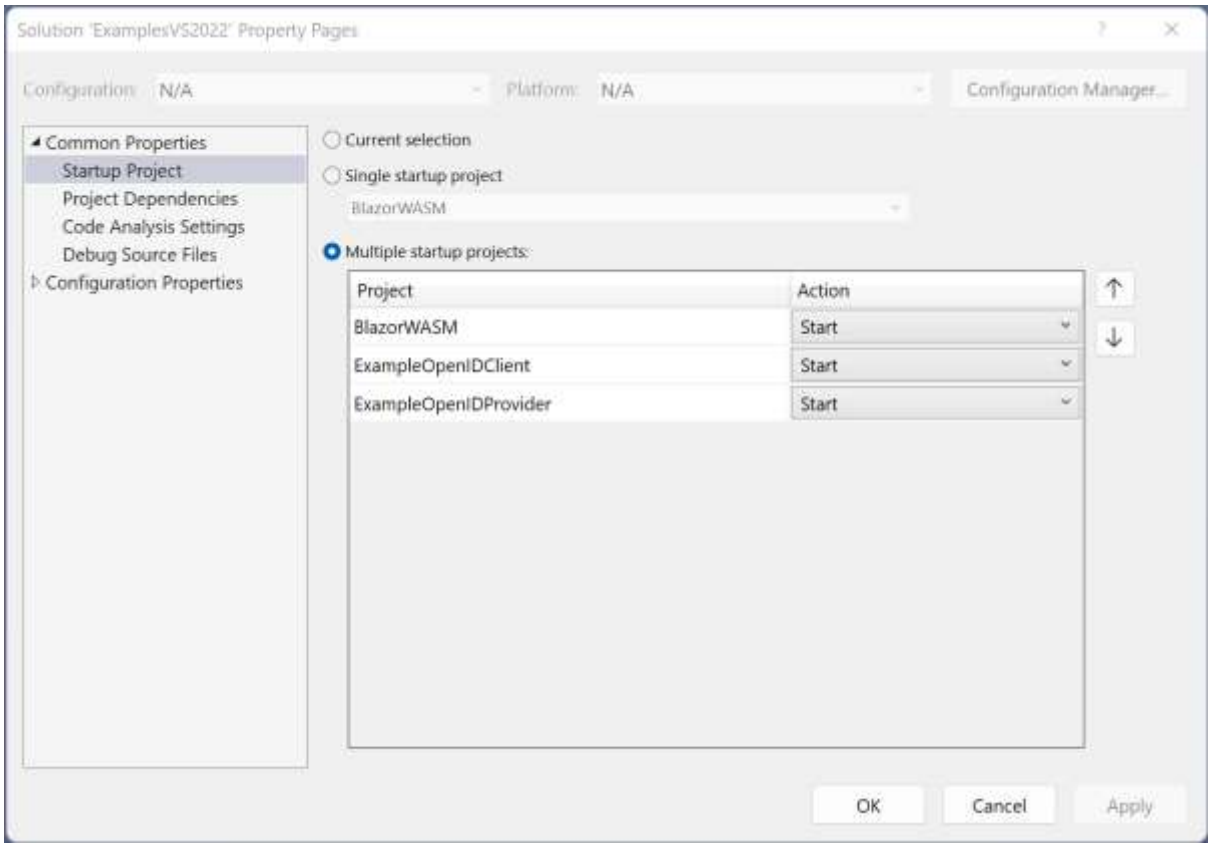



*Figure 1 Startup Projects*

# Example OpenID Provider

The ExampleOpenIDProvider project is an ASP.NET Core web application based off the Visual Studio template.

It demonstrates acting as an OpenID provider and supports:

- Retrieval of the provider's metadata
- Retrieval of the provider's keyset
- Authentication using various OpenID Connect flows
- Logout
- Refresh token support
- Client credentials support
- Authorization through JWT tokens

## Building and Running

The ExampleOpenIDProvider project should build without any errors or warnings.

As it is configured to use the default LocalDB connection string, the simplest approach is to run the application on IIS Express through the Visual Studio debugger.

Note that this database is not used by the OpenID API but is the application's user registry.

To run on IIS, the application must be configured in and published to IIS. It should use a database provider other than LocalDB.

The application is configured to run at https://localhost:44311/.

If this is changed, the configuration for the client applications must be updated to match the new URL.

To demonstrate OpenID Connect, the ExampleOpenIDProvider and either ExampleOpenIDClient or BlazorWASM must be run.

# Example Blazor WASM OpenID Client

The BlazorWASM project is an Blazor application based off the Visual Studio template.

It uses the standard Microsoft OpenID Connect authentication handler.

For more information, refer to the Microsoft documentation.

It demonstrates acting as an OpenID client and supports:

- Retrieval of the provider's configuration
- Retrieval of the provider's keyset
- Authentication
- Logout
- Refresh token support
- Authorization through JWT tokens

## Building and Running

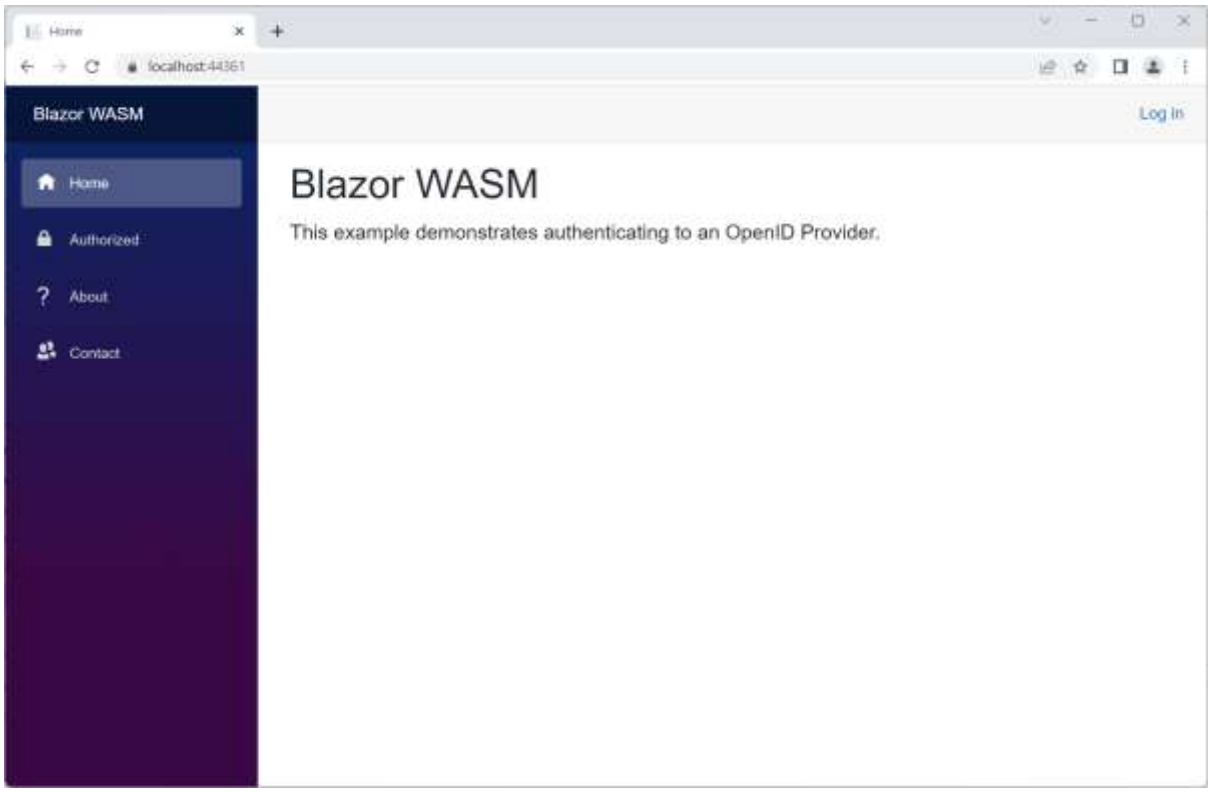
The BlazorWASM project should build without any errors or warnings.

The application is configured to run at https://localhost:44361/.

To demonstrate OpenID Connect, both the ExampleOpenIDProvider and BlazorWASM must be run.

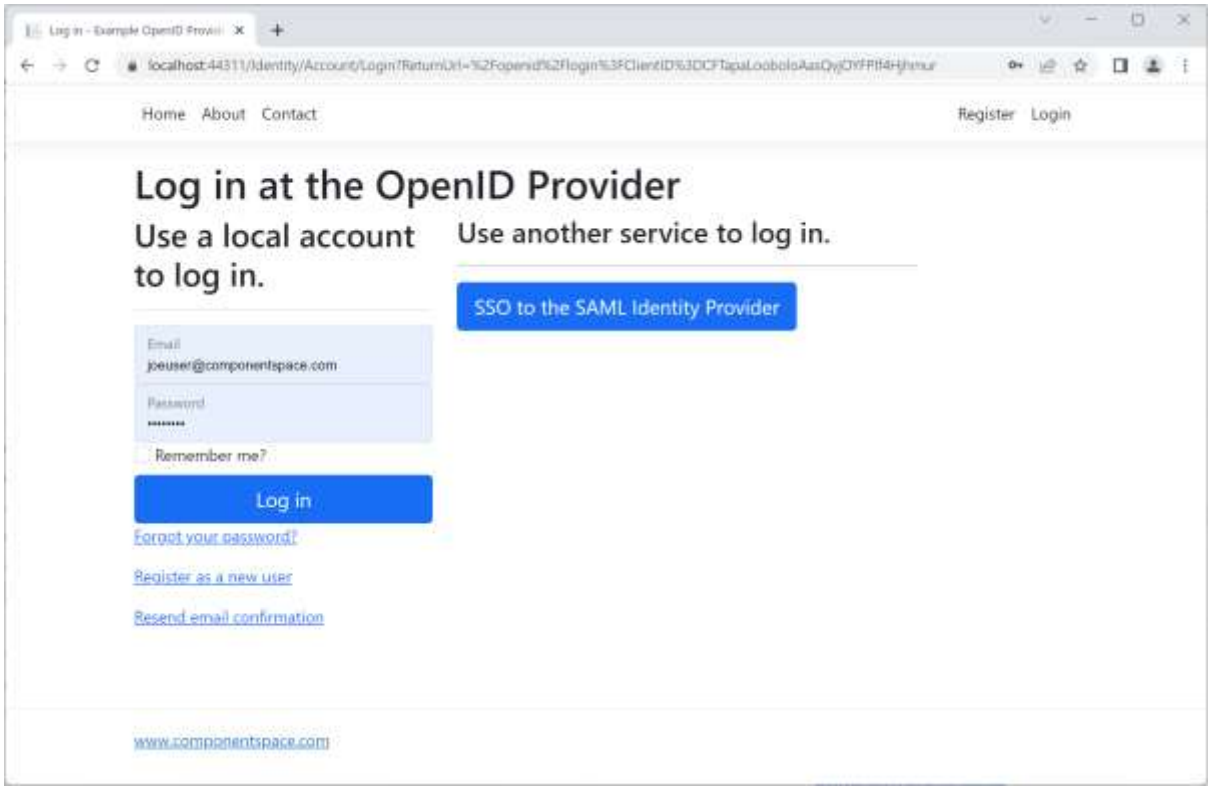
## Authentication

Browse to the BlazorWASM's home page at https://localhost:44361/.

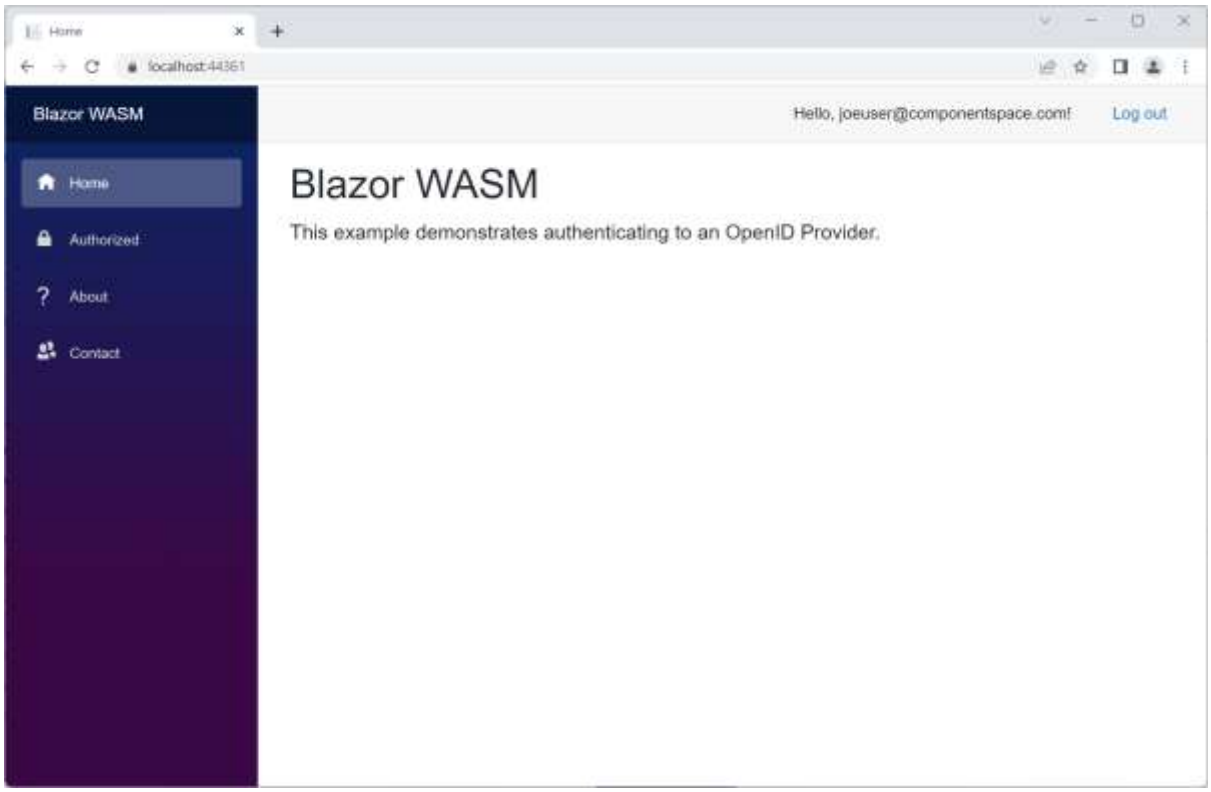



Click the Log in link.

You are prompted to login at the OpenID provider.



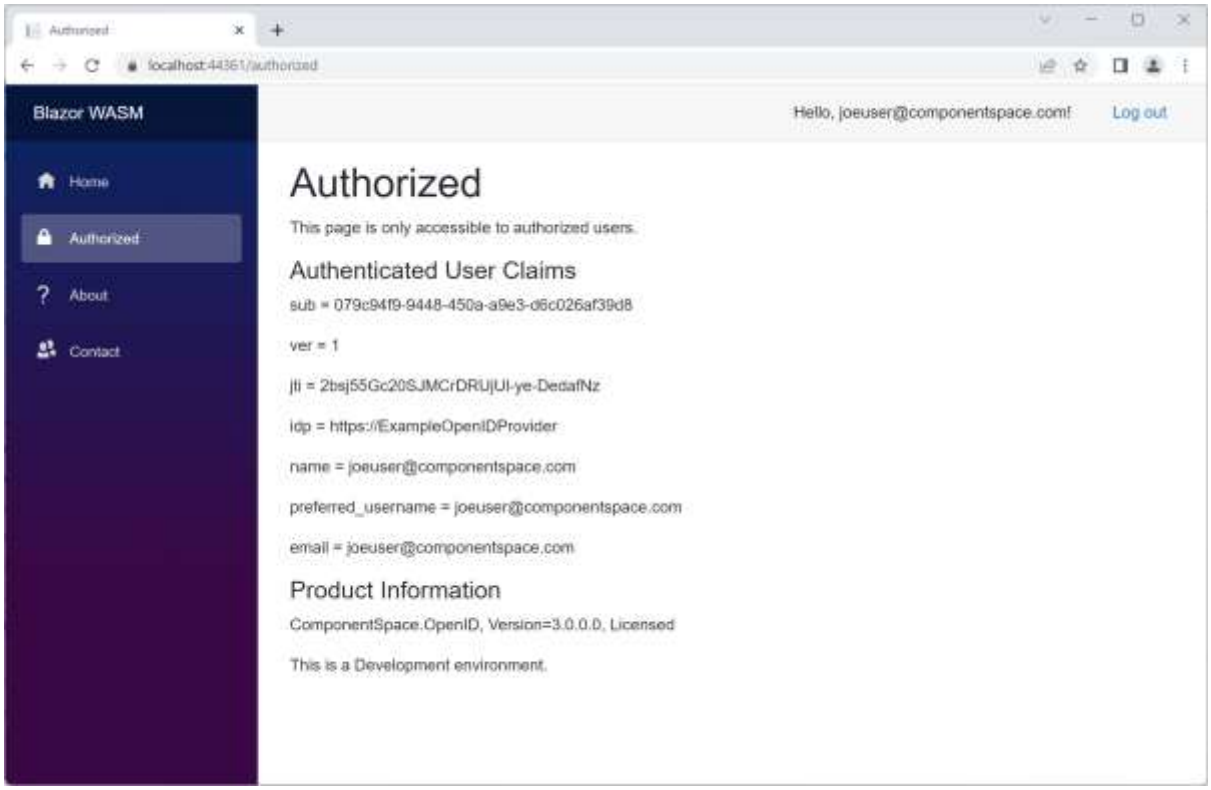

Authentication completes with automatic login at the client. The user identity is that specified by the OpenID provider.




## Authorization

Click the Authorized link.

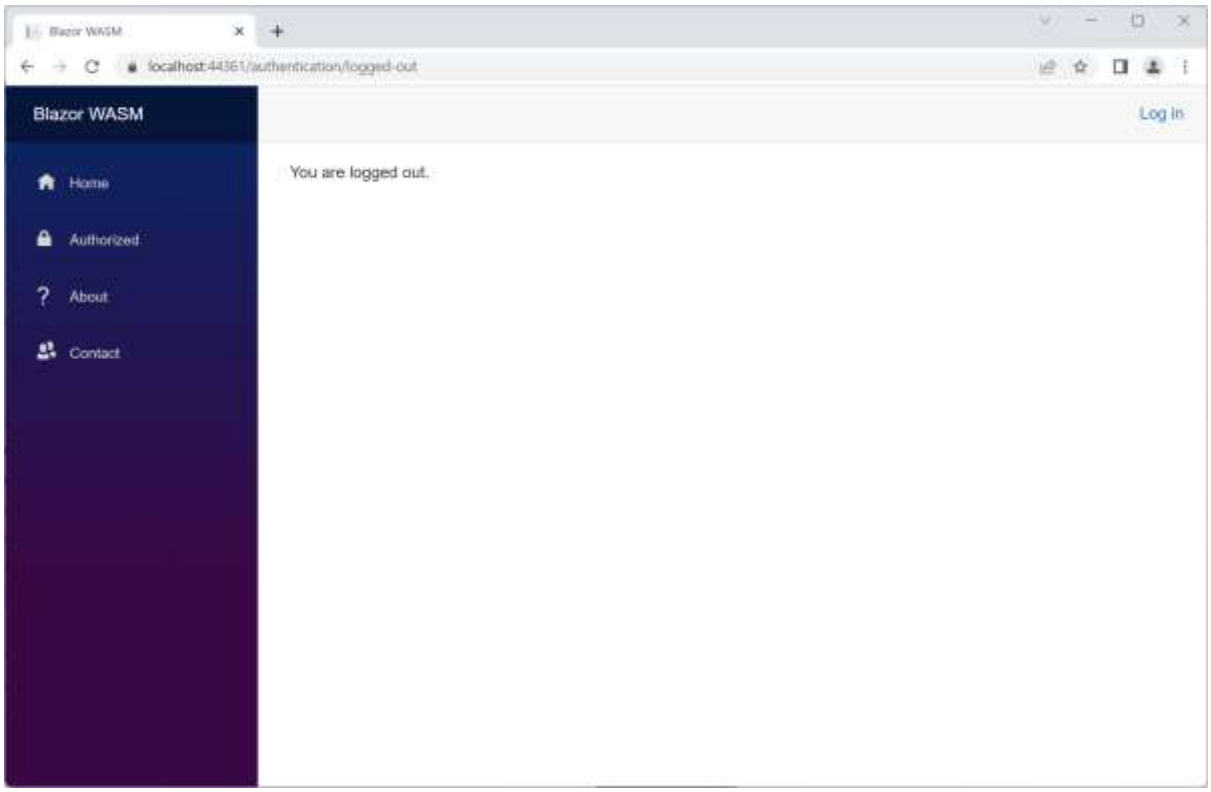
The JWT access token is used to call a secure web API.




## Logout

Click the Log out link.

Logout occurs at both the OpenID provider and client.



# Example ASP.NET Core OpenID Client

The ExampleOpenIDClient project is an ASP.NET Core application based off the Visual Studio template.

It uses the standard Microsoft OpenID Connect authentication handler.

For more information, refer to the Microsoft documentation.

It demonstrates acting as an OpenID client and supports:

- Retrieval of the provider's configuration
- Retrieval of the provider's keyset
- Authentication
- Logout
- Authorization through JWT tokens

## Building and Running

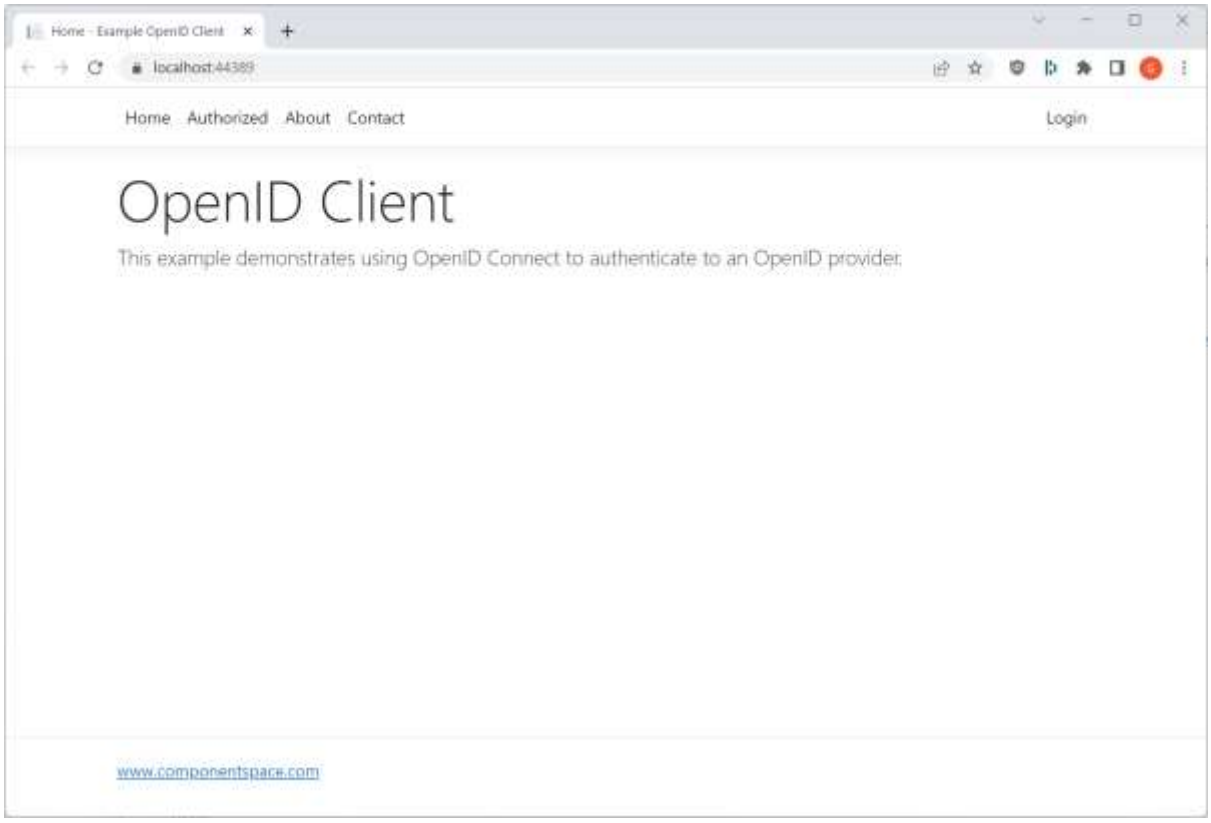
The ExampleOpenIDClient project should build without any errors or warnings.

The application is configured to run at https://localhost:44389/.

To demonstrate OpenID Connect, both the ExampleOpenIDProvider and ExampleOpenIDClient must be run.

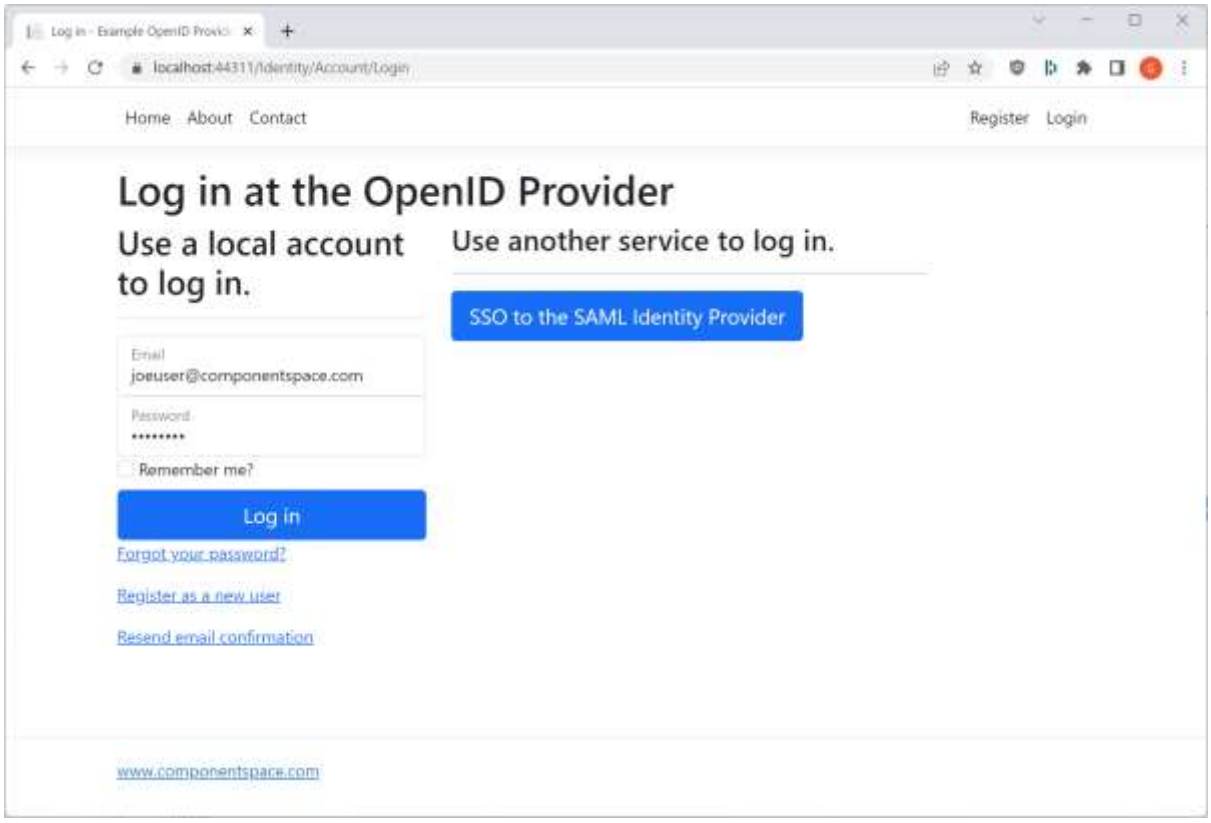
## Authentication

Browse to the ExampleOpenIDClient home page at https://localhost:44389/.

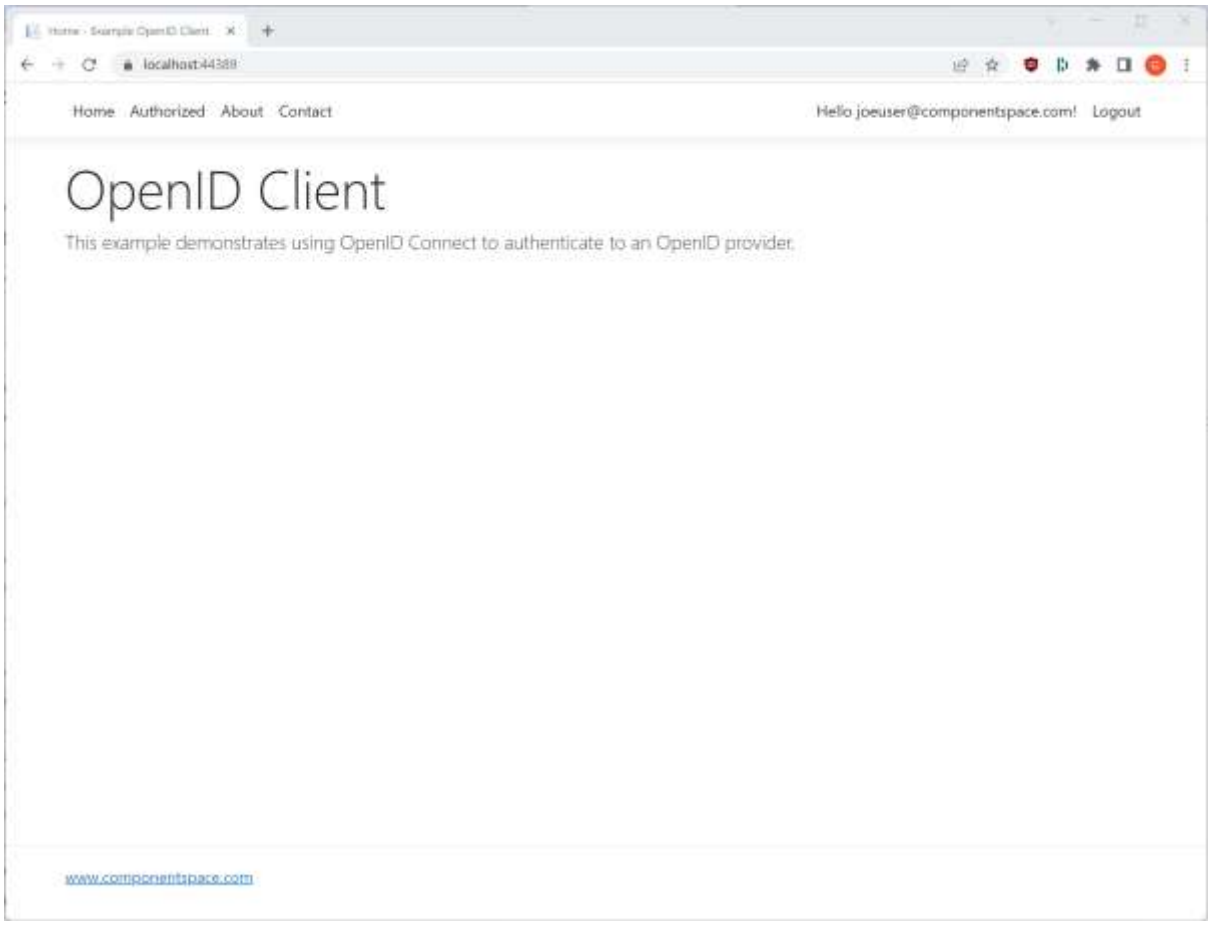



Click the Log in link.

You are prompted to login at the OpenID provider.



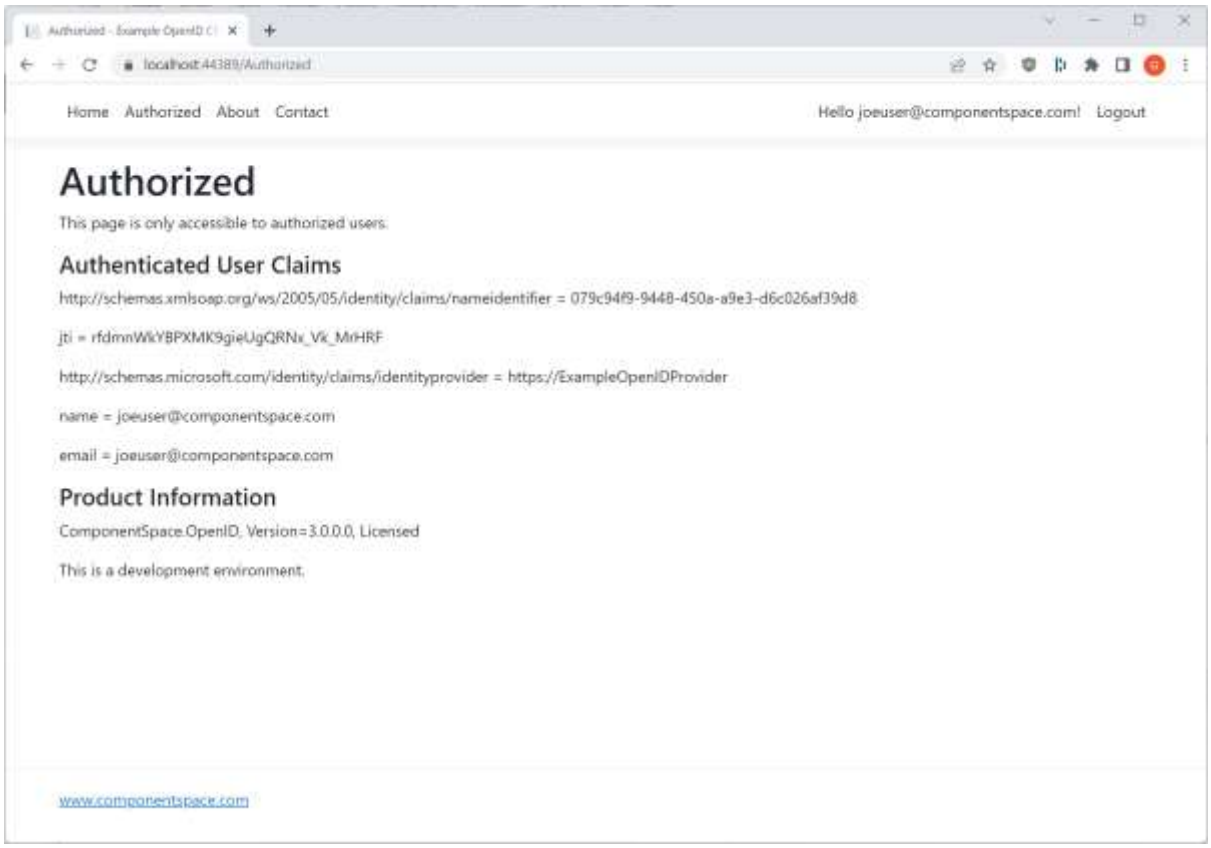

Authentication completes with automatic login at the client. The user identity is that specified by the OpenID provider.



## Authorization

Click the Authorized link.

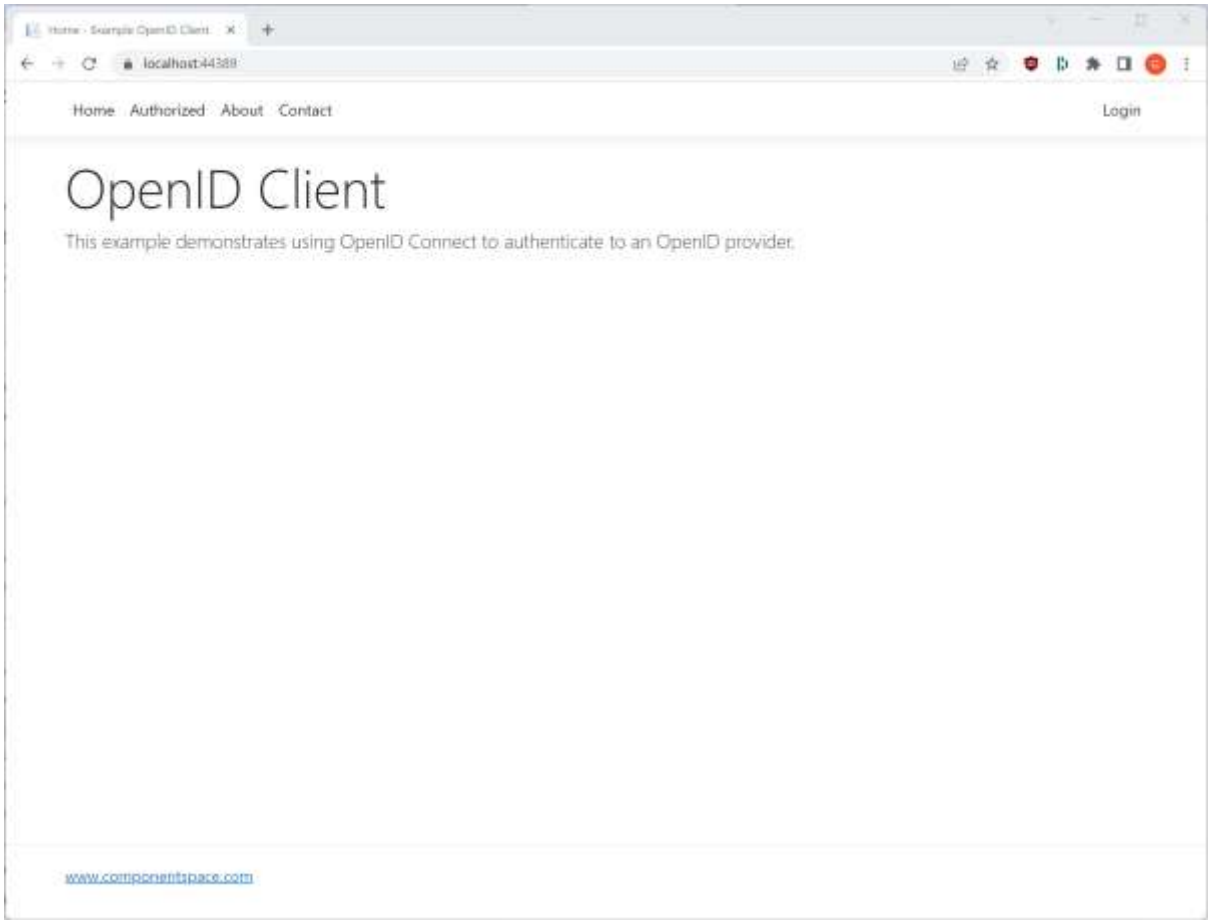
The JWT access token is used to call a secure web API.




## Logout

Click the Log out link.

Logout occurs at both the OpenID provider and client.




# SAML SSO and OpenID Connect

## Building and Running

The OpenID provider supports both local login and SAML SSO.

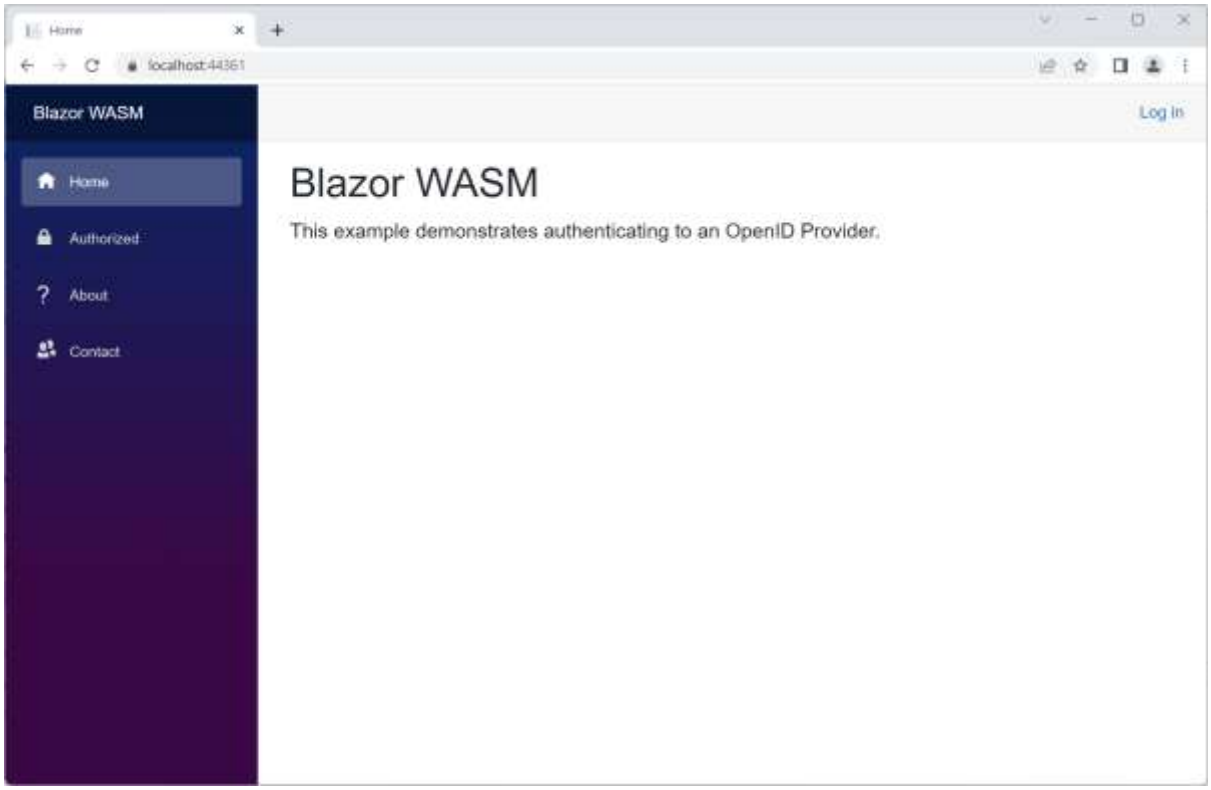
To demonstrate SAML SSO and OpenID Connect, the ExampleIdentityProvider, ExampleOpenIDProvider and BlazorWASM must be run.

The ExampleIdentityProvider project is included with the SAML for ASP.NET Core product.

The same flow can be demonstrated using the ExampleIdentityProvider, ExampleOpenIDProvider and ExampleOpenIDClient.

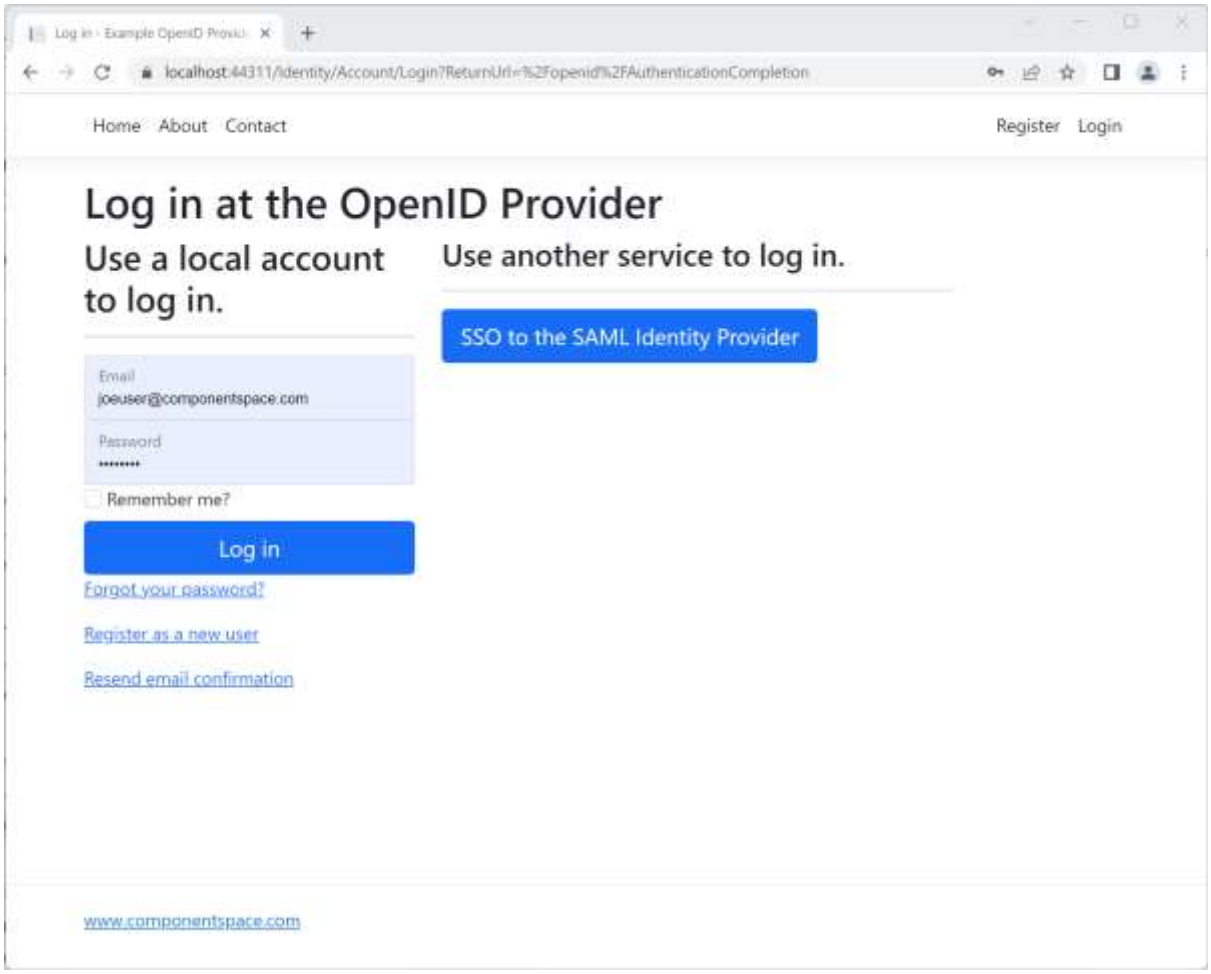
## Authentication

Browse to the BlazorWASM's home page at https://localhost:44361/.

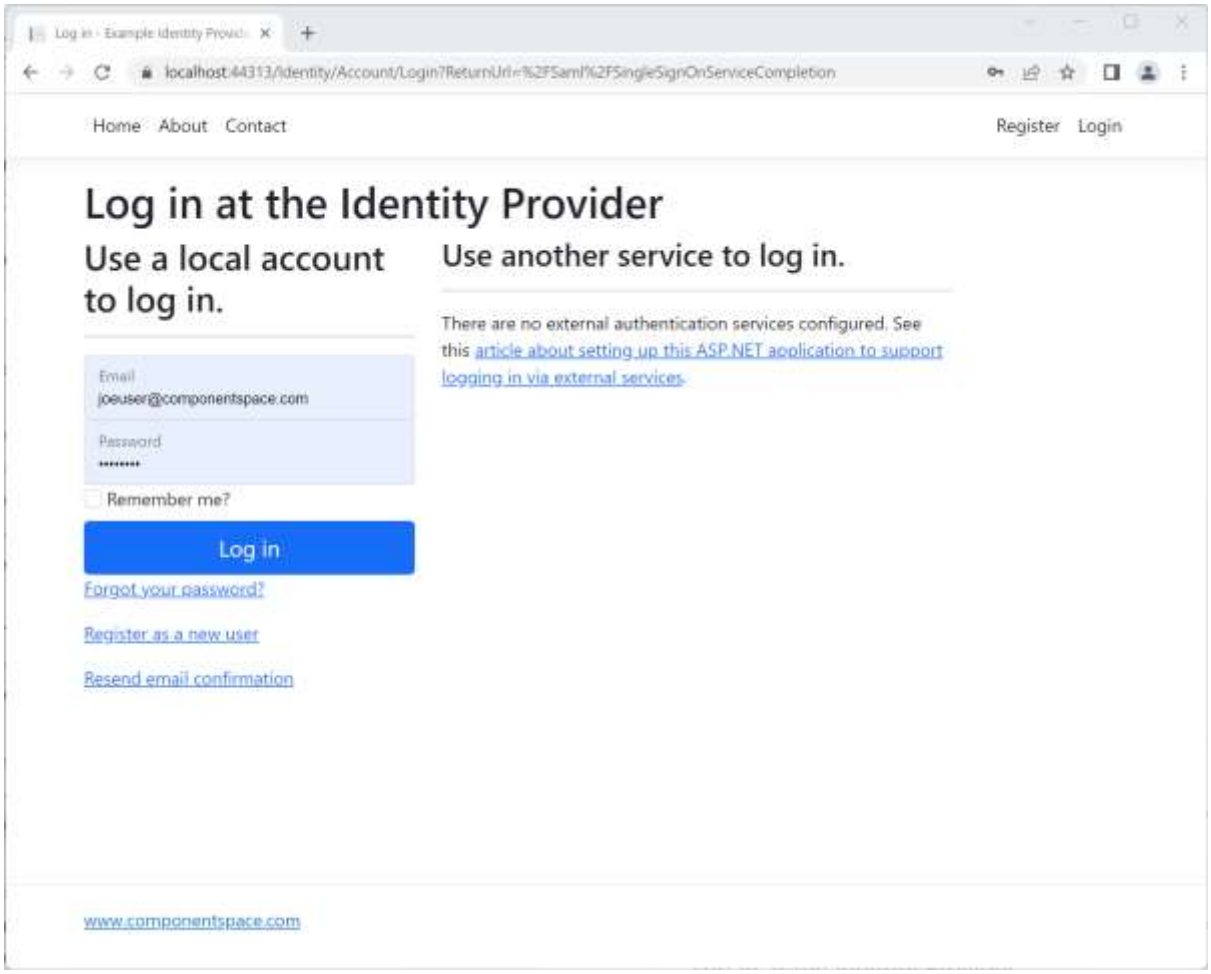



Click the Log in link.

You are prompted to login at the OpenID provider.

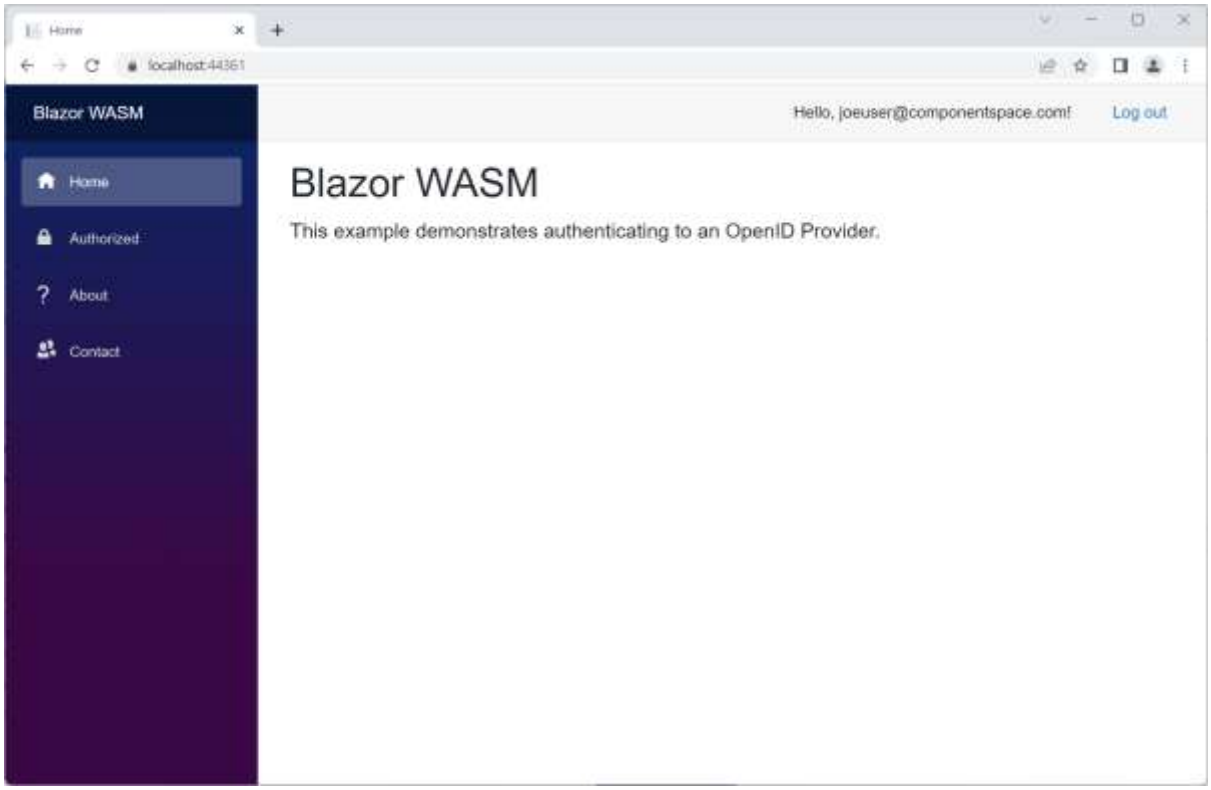



Rather than logging in locally at the OpenID provider, click the “SSO to the SAML Identity Provider” button to initiate SAML SSO to the ExampleIdentityProvider.




Login at the identity provider.

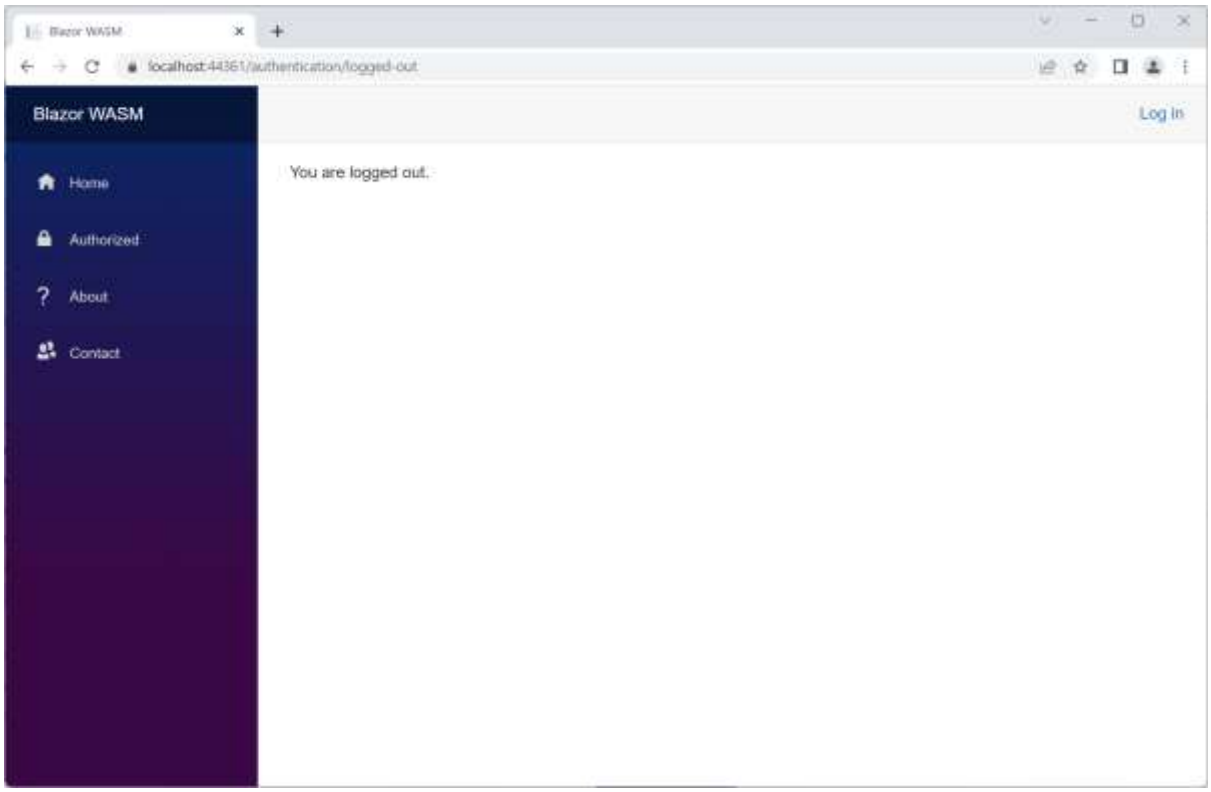
Authentication completes with automatic login at the client. The user identity is that specified by the OpenID provider and originally retrieved from the SAML identity provider.




## Logout

Click the Log out link.

Logout occurs at the SAML identity provider, OpenID provider and client.



# OpenID Provider Conformance Testing

The ConformanceTesting project is an ASP.NET Core web application based off the Visual Studio template.

It demonstrates acting as an OpenID provider and supports:

- Retrieval of the provider's metadata
- Retrieval of the provider's keyset
- Authentication using various OpenID Connect flows
- Logout

## Building and Running

The ConformanceTesting project should build without any errors or warnings.

It's used for conformance testing with the OpenID Foundation's OpenID Connect OPs test plans.

https://openid.net/certification/connect_op_testing/

It's been successfully run against the following test plans:

- oidcc-config-certification-test-plan
- oidcc-basic-certification-test-plan
- oidcc-implicit-certification-test-plan
- oidcc-hybrid-certification-test-plan
- oidcc-formpost-basic-certification-test-plan
- oidcc-formpost-implicit-certification-test-plan
- oidcc-formpost-hybrid-certification-test-plan
- oidcc-rp-initiated-logout-certification-test-plan, response_type=code
- oidcc-rp-initiated-logout-certification-test-plan, response_type=id_token
- oidcc-rp-initiated-logout-certification-test-plan, response_type=id_token token
- oidcc-rp-initiated-logout-certification-test-plan, response_type=code id_token
- oidcc-rp-initiated-logout-certification-test-plan, response_type=code token
- oidcc-rp-initiated-logout-certification-test-plan, response_type=code id_token token

# Code Walkthrough

## Example OpenID Provider

### Configuration

The appsettings.json includes the OpenID configuration.

The configuration consists of a single OpenID configuration specifying a single OpenID provider and multiple clients.

Refer to the OpenID Connect for ASP.NET Core Configuration Guide for more information.

### Startup

The Program class includes the following code.

```
// Add OpenID provider services.
builder.Services.AddOpenIDProvider(builder.Configuration.GetSection("OpenIDProvider"));
```

This code registers the OpenID configuration and adds the OpenID provider services.

### OpenIDController.GetMetadataAsync

The OpenID controller includes the following action to support OpenID provider configuration retrieval.

```
[Route(".well-known/openid-configuration")]
[ResponseCache(Duration = 600, Location = ResponseCacheLocation.Any)]
public async Task<IActionResult> GetMetadataAsync ()
{
   // Return the OpenID provider's metadata.
   return await _openIDProvider.GetMetadataAsync ();
}
```

The route ".well-known/openid-configuration" is the standard specified by the OpenID Connect Discovery specification.

Clients will access this endpoint to retrieve the OpenID provider's metadata.

Caching directives may be included as required.

GetMetadataAsync is called to retrieve the OpenID provider's metadata from its configuration.

### OpenIDController.GetKeysAsync

The controller includes the following action to support OpenID provider JSON Web Key Set document retrieval.

```
[Route("openid/keys")]
[ResponseCache(Duration = 600, Location = ResponseCacheLocation.Any)]
public async Task<IActionResult> GetKeysAsync()
{
   // Return the OpenID provider's keys.
   return await _openIDProvider.GetKeysAsync();
}
```

Any route may be specified but it must match with the JwksUri included in the OpenID provider metadata.

Clients will access this endpoint to retrieve the OpenID provider's JWKS document.

Caching directives may be included as required.

GetKeysAsync is called to retrieve the OpenID provider's JWKS document which is derived from the configured provider certificates.

### OpenIDController.AuthorizeAsync

The controller includes the following action to support receiving and processing authentication requests.

```
[Route("openid/authorize")]
public async Task<IActionResult> AuthorizeAsync()
{
```

```
  try
  {
    // Receive and process the OpenID authentication request.
    var authenticationRequest = await _openIDProvider.ReceiveAuthnRequestAsync();

    // If the user is authenticated but login is required, re-authenticate the user.
    // Otherwise, send an authentication response.
    // If the user isn't authenticated but login isn't permitted, send an error response.
    // Otherwise, authenticate the user.
    if (User.Identity is not null && User.Identity.IsAuthenticated)
    {
      if (authenticationRequest.Prompt == OpenIDConstants.PromptModes.Login)
      {
        return Redirect("/Identity/Account/Logout?handler=initiate&returnUrl=/openid/login");
      }

      return await SendAuthnResponseAsync();
    }
    else
    {
      if (authenticationRequest.Prompt == OpenIDConstants.PromptModes.None)
      {
        throw new LoginRequiredException();
      }

      return RedirectToAction("Login");
    }
  }

  catch (Exception exception)
  {
    // Send an authentication error response to the client.
    return await _openIDProvider.SendAuthnErrorResponseAsync(exception);
  }
}
```

Any route may be specified but it must match with the AuthorizationEndpoint included in the OpenID provider metadata.

Clients will access this endpoint to initiate authentication.

ReceiveAuthnRequestAsync is called to receive and process the authentication request.

An internal redirect to the Login action is performed to ensure the user is logged in.

If any errors occur, SendAuthnErrorResponseAsync is called to create and send an error authentication response to the client.

```
[Authorize]
[Route("openid/login")]
public async Task<IActionResult> LoginAsync()
{
  // Now that the user has been authenticated, send an authentication response to the client.
  return await SendAuthnResponseAsync();
```

```
}
```

The authorize attribute on the method ensures only authenticated users can respond to the authentication request.

```
private async Task<IActionResult> SendAuthnResponseAsync(string clientID)
{
   try
   {
      // Confirm the user has been authenticated.
      if (User.Identity is null)
      {
         throw new Exception("The user isn't logged in.");
      }

      var nameIdentifier = User.FindFirst(ClaimTypes.NameIdentifier);

      if (nameIdentifier is null || nameIdentifier.Value is null)
      {
         throw new Exception("The name identifier is missing.");
      }

      // Create some claims based off the user identity to include in the identity token or user info.
      var claims = new List<Claim>();

      if (User.Identity.Name is not null)
      {
         claims.Add(new Claim(OpenIDConstants.ClaimNames.Name, User.Identity.Name));
         claims.Add(new Claim(OpenIDConstants.ClaimNames.PreferredUsername, User.Identity.Name));
      }

      var userClaim = User.FindFirst(ClaimTypes.GivenName);

      if (userClaim is not null)
      {
         claims.Add(new Claim(OpenIDConstants.ClaimNames.GivenName, userClaim.Value));
      }

      userClaim = User.FindFirst(ClaimTypes.Surname);

      if (userClaim is not null)
      {
         claims.Add(new Claim(OpenIDConstants.ClaimNames.FamilyName, userClaim.Value));
      }

      userClaim = User.FindFirst(ClaimTypes.Email);

      if (userClaim is not null)
      {
         claims.Add(new Claim(OpenIDConstants.ClaimNames.Email, userClaim.Value));
      }

      // The JWT access token can be used by the application for API authorization purposes, if required.
      // The scp claim is used when performing the "required scope" authorization check.
```

```
      var jwtClaims = new List<Claim>()
      {
         new Claim(ClaimConstants.Scp, LicenseController.GetLicenseScope)
      };

      var accessToken = await _openIDProvider.CreateJwtAccessTokenAsync(clientID,
_configuration["JWT:Audience"], nameIdentifier.Value, LicenseController.GetLicenseScope, jwtClaims);

      // Refresh tokens are optional but if supported the application is responsible for their control and
lifetime.
      var refreshToken = _idGenerator.Generate();

      var openIDUser = new OpenIDUser()
      {
         RefreshToken = refreshToken,
         ClientID = clientID,
         Subject = nameIdentifier.Value
      };

      await _openIDUserContext.AddAsync(openIDUser);
      await _openIDUserContext.SaveChangesAsync();

      // Send an authentication response to the client.
      return await _openIDProvider.SendAuthnResponseAsync(nameIdentifier.Value, claims, accessToken,
refreshToken);
   }

   catch (Exception exception)
   {
      // Send an authentication error response to the client.
      return await _openIDProvider.SendAuthnErrorResponseAsync(exception);
   }
}
```

The user's name and some associated claims, to be included in the ID token returned to the client, are retrieved.

SendAuthnResponseAsync is called to construct and send an authentication response including an ID token to the client. Exactly when the ID token is sent is dependent on the flow.

A JWT access token is created for subsequent authorized calls to a web API.

A refresh token is created and stored in a simple Entity Framework database. This is required only if refresh tokens are to be supported and the application is responsible for their control and lifetime.

If any errors occur, SendAuthnErrorResponseAsync is called to create and send an error authentication response to the client.

### OpenIDController.LogoutAsync

The controller includes the following action to support receiving and processing logout requests.

```
[Route("openid/logout")]
public async Task<IActionResult> LogoutAsync()
{
```

```
    // Receive and process the OpenID logout request.
    await _openIDProvider.ReceiveLogoutRequestAsync();

    if (User.Identity is not null && User.Identity.IsAuthenticated)
    {
      // Logout locally.
      return Redirect("/Identity/Account/Logout?handler=initiate&returnUrl=/openid/logout-callback");
    }

    // Send a logout response to the client.
    return await LogoutCallbackAsync();
}
```

Any route may be specified but it must match with the EndSessionEndpoint included in the OpenID provider metadata.

Clients will access this endpoint to initiate logout.

ReceiveLogoutRequestAsync is called to receive and process the logout request.

An internal redirect to a handler within the Microsoft identity scaffolding is performed to logout the user.

```
[Route("openid/logout-callback")]
public async Task<IActionResult> LogoutCallbackAsync()
{
    // Now that the user is logged out, send a logout response to the client.
    return await _openIDProvider.SendLogoutResponseAsync();
}
```

Once logout completes, SendLogoutResponseAsync is called to send a logout response to the client.

### OpenIDController.TokenAsync

The controller includes the following action to support returning the ID token as part of the authorization code flow.

```
[Route("openid/token")]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
public async Task<IActionResult> TokenAsync()
{
    // Return the OpenID tokens.
    return await _openIDProvider.GetTokensAsync(GetRefreshTokenResultAsync,
GetClientCredentialsResultAsync, GetUserCredentialsResultAsync);
}
```

Any route may be specified but it must match with the TokenEndpoint included in the OpenID provider metadata.

Clients will access this endpoint to retrieve ID tokens.

GetTokensAsync is called to retrieve the ID token.

Optional delegate methods support refresh_token, client_credentials and password grant types.

### OpenIDController.UserInfoAsync

The controller includes the following action to support returning user information.

```
[Route("openid/userinfo")]
public async Task<IActionResult> UserInfoAsync()
{
   // Return the user information.
   return await _openIDProvider.GetUserInfoAsync();
}
```

Any route may be specified but it must match with the UserinfoEndpoint included in the OpenID provider metadata.

Clients will access this endpoint for user information.

### SamlController

The SAML controller supports the OpenID provider acting as a SAML service provider.

The OpenID provider acts as an authentication protocol converter. It allows OpenID clients to authenticate to SAML identity providers.

The OpenID controller has no knowledge of the SAML controller and the two work independently.

Refer to the SAML for ASP.NET Core product documentation for information regarding the SAML API and configuration.

## Error Handling

As these are example applications, no error handling is included. Exceptions are not caught and therefore are displayed in the browser.

In a production application, exceptions should be caught and processed.

Refer to the OpenID Connect for ASP.NET Core Developer Guide for more information.