Tenant sign-up and onboarding

07/21/2017 • 6 minutes to read • Contributors all

In this article

Sample code

This article describes how to implement a *sign-up* process in a multitenant application, which allows a customer to sign up their organization for your application.

There are several reasons to implement a sign-up process:

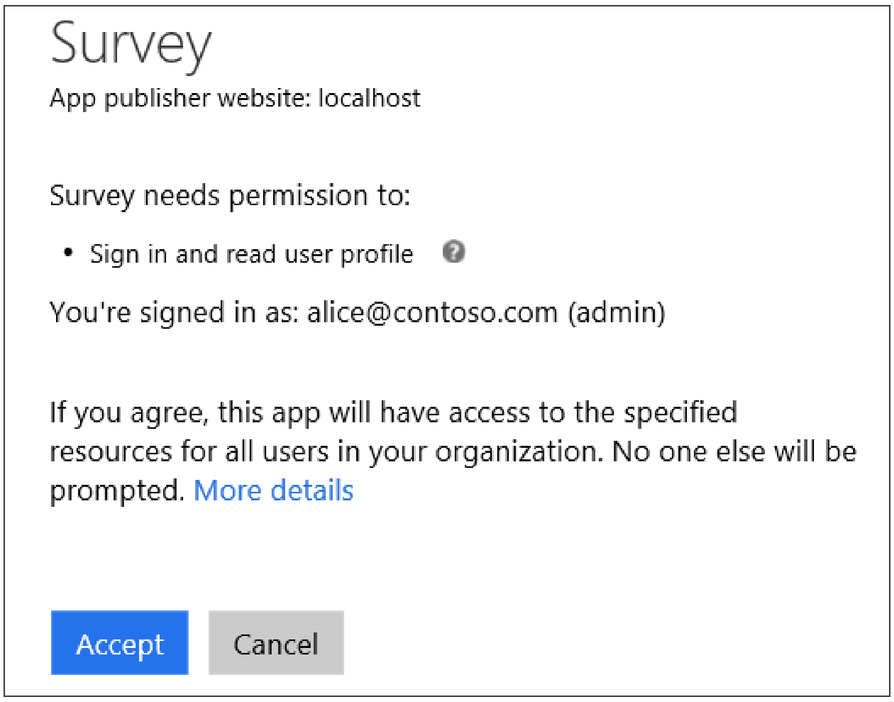
- Allow an AD admin to consent for the customer's entire organization to use the application.
- Collect credit card payment or other customer information.
- Perform any one-time per-tenant setup needed by your application.

Admin consent and Azure AD permissions

In order to authenticate with Azure AD, an application needs access to the user's directory. At a minimum, the application needs permission to read the user's profile. The first time that a user signs in, Azure AD shows a consent page that lists the permissions being requested. By clicking **Accept**, the user grants permission to the application.

By default, consent is granted on a per-user basis. Every user who signs in sees the consent page. However, Azure AD also supports *admin consent*, which allows an AD administrator to consent for an entire organization.

When the admin consent flow is used, the consent page states that the AD admin is granting permission on behalf of the entire tenant:


Survey

App publisher website: localhost

Survey needs permission to:

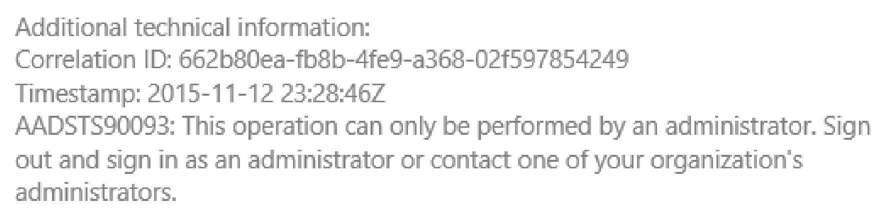
- Sign in and read user profile

You're signed in as: alice@contoso.com (admin)

If you agree, this app will have access to the specified resources for all users in your organization. No one else will be prompted. More details

After the admin clicks **Accept**, other users within the same tenant can sign in, and Azure AD will skip the consent screen.

Only an AD administrator can give admin consent, because it grants permission on behalf of the entire organization. If a non-administrator tries to authenticate with the admin consent flow, Azure AD displays an error:


Additional technical information:
Correlation ID: 662b80ea-fb8b-4fe9-a368-02f597854249
Timestamp: 2015-11-12 23:28:46Z
AADSTS90093: This operation can only be performed by an administrator. Sign out and sign in as an administrator or contact one of your organization's administrators.

If the application requires additional permissions at a later point, the customer will need to sign up again and consent to the updated permissions.

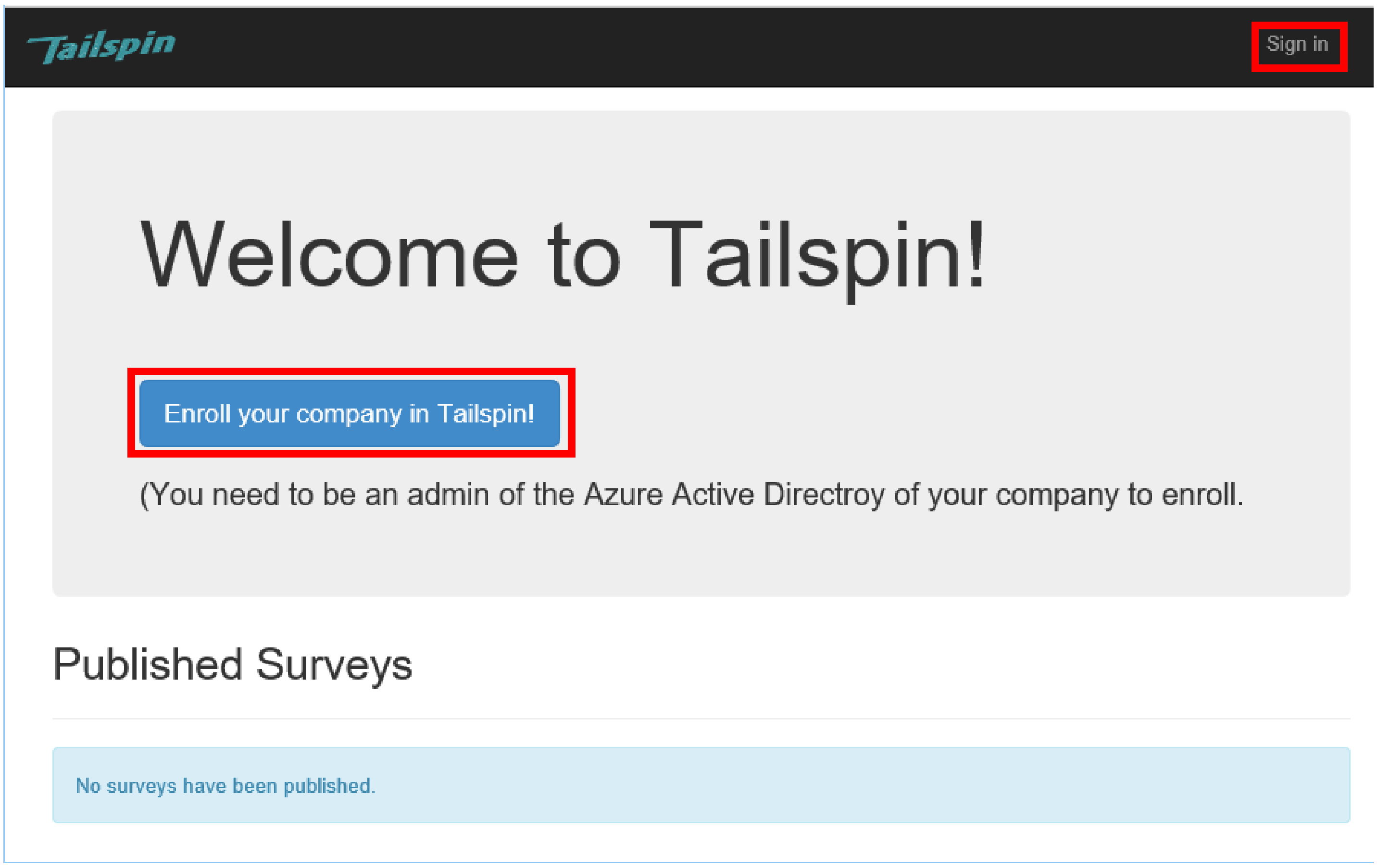
Implementing tenant sign-up

For the Tailspin Surveys application, we defined several requirements for the sign-up process:

- A tenant must sign up before users can sign in.
- Sign-up uses the admin consent flow.
- Sign-up adds the user's tenant to the application database.
- After a tenant signs up, the application shows an onboarding page.

In this section, we'll walk through our implementation of the sign-up process. It's important to understand that "sign up" versus "sign in" is an application concept. During the authentication flow, Azure AD does not inherently know whether the user is in process of signing up. It's up to the application to keep track of the context.

When an anonymous user visits the Surveys application, the user is shown two buttons, one to sign in, and one to "enroll your company" (sign up).


Tailspin
Sign in
Welcome to Tailspin!
Enroll your company in Tailspin!
(You need to be an admin of the Azure Active Directroy of your company to enroll.
Published Surveys
No surveys have been published.

These buttons invoke actions in the `AccountController` class.

The `SignIn` action returns a **ChallengeResult**, which causes the OpenID Connect middleware to redirect to the authentication endpoint. This is the default way to trigger authentication in ASP.NET Core.

```csharp
[AllowAnonymous]
public IActionResult SignIn()
{
    return new ChallengeResult(
        OpenIdConnectDefaults.AuthenticationScheme,
        new AuthenticationProperties
        {
            IsPersistent = true,
            RedirectUri = Url.Action("SignInCallback", "Account")
        });
}
```

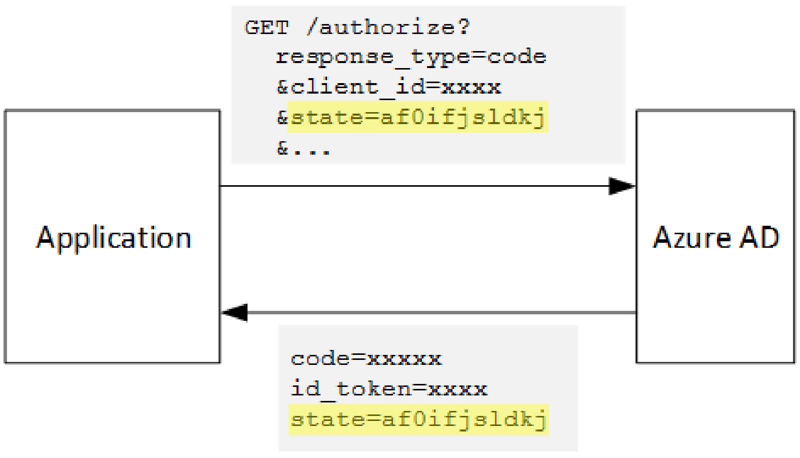
Now compare the `SignUp` action:

```csharp
[AllowAnonymous]
public IActionResult SignUp()
{
    var state = new Dictionary<string, string> { { "signup", "true" }};
    return new ChallengeResult(
        OpenIdConnectDefaults.AuthenticationScheme,
        new AuthenticationProperties(state)
        {
            RedirectUri = Url.Action(nameof(SignUpCallback), "Account")
        });
}
```

Like `SignIn`, the `SignUp` action also returns a `ChallengeResult`. But this time, we add a piece of state information to the `AuthenticationProperties` in the `ChallengeResult`:

- signup: A Boolean flag, indicating that the user has started the sign-up process.

The state information in `AuthenticationProperties` gets added to the OpenID Connect state parameter, which round trips during the authentication flow.



```
GET /authorize?
  response_type=code
  &client_id=xxxx
  &state=af0ifjsldkj
  &...
```

```
code=xxxxx
id_token=xxxx
state=af0ifjsldkj
```

After the user authenticates in Azure AD and gets redirected back to the application, the authentication ticket contains the state. We are using this fact to make sure the "signup" value persists across the entire authentication flow.

Adding the admin consent prompt

In Azure AD, the admin consent flow is triggered by adding a "prompt" parameter to the query string in the authentication request:

```
/authorize?prompt=admin_consent&...
```

The Surveys application adds the prompt during the `RedirectToAuthenticationEndpoint` event. This event is called right before the middleware redirects to the authentication endpoint.

```csharp
public override Task RedirectToAuthenticationEndpoint(RedirectContext context)
{
    if (context.IsSigningUp())
    {
        context.ProtocolMessage.Prompt = "admin_consent";
    }

    _logger.RedirectToIdentityProvider();
    return Task.FromResult(0);
}
```

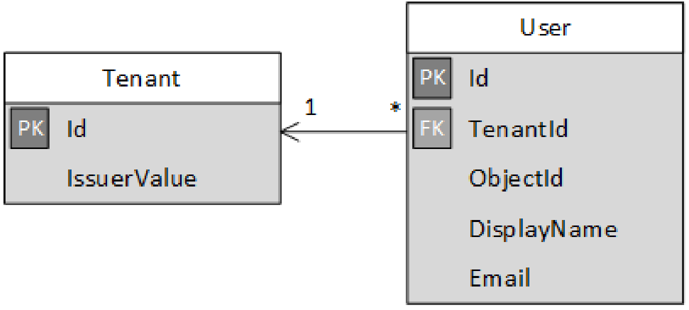
Setting `ProtocolMessage.Prompt` tells the middleware to add the "prompt" parameter to the authentication request.

Note that the prompt is only needed during sign-up. Regular sign-in should not include it. To distinguish between them, we check for the `signup` value in the authentication state. The following extension method checks for this condition:

```csharp
internal static bool IsSigningUp(this BaseControlContext context)
{
    Guard.ArgumentNotNull(context, nameof(context));

    string signupValue;
    // Check the HTTP context and convert to string
    if ((context.Ticket == null) ||
        (!context.Ticket.Properties.Items.TryGetValue("signup", out signupValue)))
    {
        return false;
    }

    // We have found the value, so see if it's valid
    bool isSigningUp;
    if (!bool.TryParse(signupValue, out isSigningUp))
    {
        // The value for signup is not a valid boolean, throw

        throw new InvalidOperationException($"'{signupValue}' is an invalid boolean value");
    }

    return isSigningUp;
}
```

Registering a tenant

The Surveys application stores some information about each tenant and user in the application database.



In the Tenant table, IssuerValue is the value of the issuer claim for the tenant. For Azure AD, this is `https://sts.windows.net/<tentantID>` and gives a unique value per tenant.

When a new tenant signs up, the Surveys application writes a tenant record to the database. This happens inside the `AuthenticationValidated` event. (Don't do it before this event, because the ID token won't be validated yet, so you can't trust the claim values. See Authentication.

Here is the relevant code from the Surveys application:

C# Copy

```csharp
public override async Task TokenValidated(TokenValidatedContext context)
{
    var principal = context.AuthenticationTicket.Principal;
    var userId = principal.GetObjectIdentifierValue();
    var tenantManager = context.HttpContext.RequestServices.GetService<TenantManager>();
    var userManager = context.HttpContext.RequestServices.GetService<UserManager>();
    var issuerValue = principal.GetIssuerValue();
    _logger.AuthenticationValidated(userId, issuerValue);

    // Normalize the claims first.
    NormalizeClaims(principal);
    var tenant = await tenantManager.FindByIssuerValueAsync(issuerValue)
        .ConfigureAwait(false);

    if (context.IsSigningUp())
    {
        if (tenant == null)
        {
            tenant = await SignUpTenantAsync(context, tenantManager)
                .ConfigureAwait(false);
        }

        // In this case, we need to go ahead and set up the user signing us up.
        await CreateOrUpdateUserAsync(context.Ticket, userManager, tenant)
            .ConfigureAwait(false);
    }
    else
    {
        if (tenant == null)
        {
            _logger.UnregisteredUserSignInAttempted(userId, issuerValue);
            throw new SecurityTokenValidationException($"Tenant {issuerValue} is not regis-
tered");
        }

        await CreateOrUpdateUserAsync(context.Ticket, userManager, tenant)
            .ConfigureAwait(false);
    }
}
```

This code does the following:

1. Check if the tenant's issuer value is already in the database. If the tenant has not signed up, `FindByIssuerValueAsync` returns null.

2. If the user is signing up:
   a. Add the tenant to the database (`SignUpTenantAsync`).
   b. Add the authenticated user to the database (`CreateOrUpdateUserAsync`).
3. Otherwise complete the normal sign-in flow:
   a. If the tenant's issuer was not found in the database, it means the tenant is not registered, and the customer needs to sign up. In that case, throw an exception to cause the authentication to fail.
   b. Otherwise, create a database record for this user, if there isn't one already (`CreateOrUpdateUserAsync`).

Here is the `SignUpTenantAsync` method that adds the tenant to the database.

C# Copy

```csharp
private async Task<Tenant> SignUpTenantAsync(BaseControlContext context, TenantManager tenant-
Manager)
{
    Guard.ArgumentNotNull(context, nameof(context));
    Guard.ArgumentNotNull(tenantManager, nameof(tenantManager));

    var principal = context.Ticket.Principal;
    var issuerValue = principal.GetIssuerValue();
    var tenant = new Tenant
    {
        IssuerValue = issuerValue,
        Created = DateTimeOffset.UtcNow
    };

    try
    {
        await tenantManager.CreateAsync(tenant)
            .ConfigureAwait(false);
    }
    catch(Exception ex)
    {
        _logger.SignUpTenantFailed(principal.GetObjectIdentifierValue(), issuerValue, ex);
        throw;
    }

    return tenant;
}
```

Here is a summary of the entire sign-up flow in the Surveys application:

1. The user clicks the **Sign Up** button.
2. The `AccountController.SignUp` action returns a challenge result. The authentication state includes "signup" value.
3. In the `RedirectToAuthenticationEndpoint` event, add the `admin_consent` prompt.
4. The OpenID Connect middleware redirects to Azure AD and the user authenticates.
5. In the `AuthenticationValidated` event, look for the "signup" state.
6. Add the tenant to the database.

Next