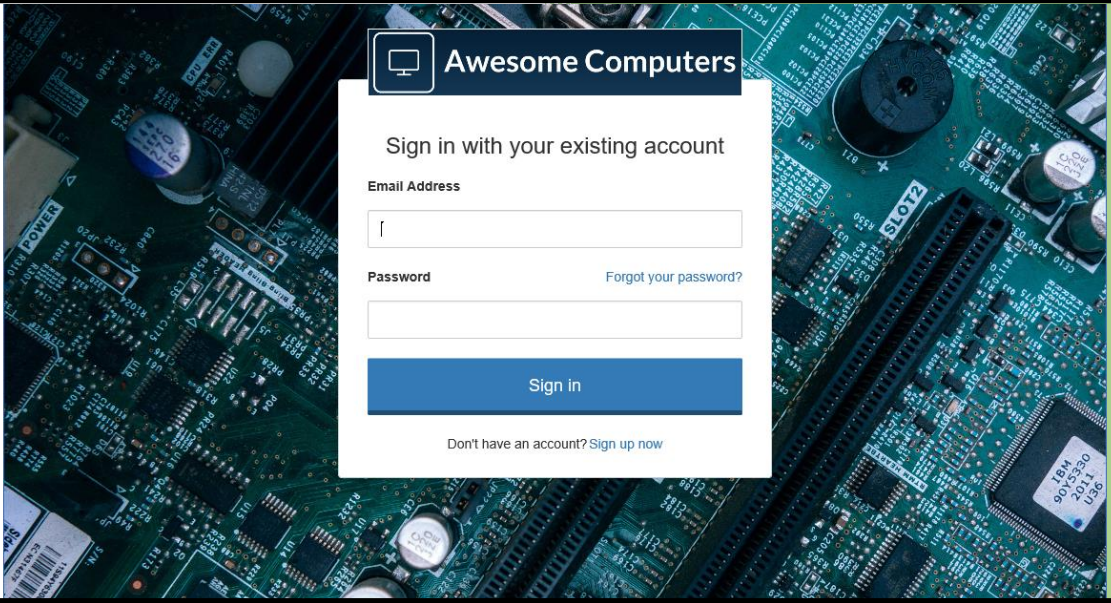
Microsoft Identity

Gaining Expertise with Azure AD B2C


Awesome Computers
Sign in with your existing account
Email Address
Password
Forgot your password?
Sign in
Don't have an account? Sign up now

A course for developers

Version 2.3, December 2020

For the latest version see

https://aka.ms/LearnAADB2C

About this course

Welcome to the ***Gaining expertise with Azure AD B2C*** course for developers.

This self-paced course is designed to take you from initial awareness of Azure AD B2C to being able to create complex user journeys for your customers and leveraging the insights of their behavior with those journeys to determine how to deepen your relationship with those customers.

Each module within the course builds on the previous, and at the end you will have a working .NET web application that:

- Enables customers to log in with local, social, and enterprise accounts.
- Enables customers to edit their profiles and reset their own passwords with an MFA experience.
- Has a fully customized UI
- Requests users to accept terms and conditions

At the end of the course, you will be able to:

- Create an Azure AD B2C tenant.
- Use built-in (UI-based) user flows to
    - Create sign-up, sign-in, profile edit, multi-factor authentication, and self-service password reset journeys.
    - Add many popular identity providers.
    - Enable sign in with social accounts.
    - Customize the user interface.
- Use XML-based custom policies to
    - Create customized sign-up, sign-in, profile edit, multi-factor authentication, and self-service password reset journeys.
    - Enable sign in with social accounts
    - Enable sign in with any OIDC or OAuth-compliant identity provider.
    - Enable REST API calls to external systems to validate user input.
    - Migrate users from other Identity Providers to Azure AD B2C.
    - CRUD users using Microsoft Graph.

Course elements

In most course modules you will see the following elements:

- **See it in action.** Before you complete most modules, you'll go to the Awesome Computers site to see the customer experience you'll build in the module.
- **Step-by-step directions.** Click-through guides or links to online documentation for completing each procedure.
- **Important concepts.** An explanation of some of the concepts important to the procedures in the module, and what happens behind the scenes.
- **Sample application, user flows, and files.** A downloadable version of the Awesome Computers application that you will use throughout course, and other files you will need. Please go to https://aka.ms/B2CCourse-SampleApplication to download all necessary assets.

Course Prerequisites

To be successful in this course, you should be familiar with the following concepts:

- What is Azure AD?
- What are authentication and authorization?
- What is federation?
- What is single sign-on?
- What are open authentication standards?
    - OpenID Connect
    - OAuth 2.0
- Understanding website .CSS

Module 0: Azure AD B2C environment setup

Introduction

At the end of this module, you will be able to:

- Set up an Azure Tenant that supports Azure AD B2C
- Add a co-admin and a test user to the Azure tenant
- Enable the co-admin to manage the Azure AD B2C tenant
- Set up an Azure AD B2C tenant
- Create a test user in the Azure AD B2C tenant
- Link the Azure AD B2C tenant to the Azure subscription

And you will have a deeper understanding of the following concepts:

- What is Azure AD B2C
- What are tenants and subscriptions
- When to use Azure AD B2C

This module should take you 30 minutes to complete.

Accounts, tenants, and subscriptions

In this module, you will create an Azure account, which results in an Azure tenant. To create an Azure AD B2C tenant, your Azure tenant must have a subscription. Azure subscriptions require a billing method and are pay-as-you-go.

Nothing you do in this course should result in any billing to your Azure subscription.

1. Create an Azure co-admin user with the following details
   - Name: B2C Admin
   - Username: b2cadmin@{yourtenantname}.onmicrosoft.com
   - Directory Role: Global Administrator
2. Create an Azure test user with the following details
   - Name: Test User
   - Username: testuser@{yourtenantname}.onmicrosoft.com
   - Directory Role: User (the default)
3. Create an Azure AD B2C tenant with the following details
   - Organization Name: Awesome Computers
   - Initial Domain Name: Awesome{YourLastName} (for example: AwesomeJones. If Awesome{YourLastName} isn't available, choose another name, such as Awesome{YourLastName}22)
4. Link the Azure AD B2C tenant to the Azure subscription.

5. From the Azure portal dashboard, select your Azure AD B2C tenant, Awesome{YourLstName}
6. In the Azure AD B2C tenant, invite the B2C Admin user you created earlier (b2cadmin@{yourtenantname}.onmicrosoft.com) as a guest user and set its role as Global administrator. For detailed steps on how to do this, click here: Invite an external user.
7. Sign out of the Azure portal and sign back in with the B2C Admin credentials.
8. Next, while in the same Azure AD B2C tenant, invite the Test User you created earlier (testuser@{yourtenantname}.onmicrosoft.com) and set its role as User. Here are detailed steps on how to create a test user.
9. Select **All Services**, Search for **B2C**, and mark **Azure AD B2C** as a favorite by selecting the ★

For the rest of this course, use the B2CAdmin account.

Important terms and concepts

- What is Azure AD B2C
- When to use Azure AD B2C

Module 1: Using built-in user flows

Introduction

This module introduces you to user flows in Azure AD B2C that can be used with Identity Providers, or IDPs, as part of the user journeys listed below. User flows save you time and effort within your application so that you do not have to completely write code or create new pages for each user journey. However, it is still possible to use the extensibility of Azure AD B2C to create custom pages for each journey if desired.

At the end of this module, you will be able to:

- Create built-in user flows for
  - The sign-up or sign-in journey
  - The password reset journey
  - The profile edit journey

And you will have a deeper understanding of the following concepts:

- What user flows are, and how applications use them.
- Claims and attributes and how they are used.
- How to view the contents of tokens.
- The pros and cons of using multi-factor authentication.
- Using user flows to facilitate single sign-on among applications.

This module should take you 15 minutes to complete.

See it in action

Before you begin the module, you can experience the end user journeys that you are about to build.

1. Go to the demonstration site and sign up.
2. Enter your profile information
   a. **Note: you must use both an email and a phone number to which you have access.**
3. Edit your profile to add "-1" to your last name.
4. Reset your password and experience the Multi-factor Authentication (MFA) experience.

Configure user flows in Azure AD B2C

'User flows' are the definitions of what your end user sees as a user experience as they perform different authentication actions such as signing up for a new account, logging in, or resetting their password. In this section, you will first create a new web application, and then create a sign-up/sign-in, password reset, and a profile edit flow that the app can use.

Complete the two tutorials below:

- Tutorial: Register an application in Azure AD B2C
- Tutorial: Create user flows in Azure AD B2C

Important terms and concepts

- Use of unique App IDs in Azure AD
- User flows
- Attributes and claims

Module 2: Using IDPs with Azure AD B2C built-in user flows

Introduction

This module introduces you to IDPs and provides you with the steps to federate with Facebook using Azure AD B2C. You will learn how to add Facebook as an identity provider and set up Facebook federation with an Azure AD B2C tenant.

At the end of this module, you will be able to:

- Register a Facebook developer account
- Create a Facebook app
- Create an Azure AD B2C IDP

And, you will have a deeper understanding of the following concepts:

- IDPs and federation
- Authentication protocols supported by Azure AD B2C

This module should take approximately 15 minutes to complete.

See it in action

Before you begin the module, you can experience the end user journeys that you are about to build by launching Module 2 from the demonstration site.

Setup social sign-in

You must have an account in the 'Facebook for Developers' website, prior to implementing this task. The following steps show how to setup an app in the Facebook for Developers website.

1. Create a Facebook app
2. Configure a Facebook account as an identity provider
3. Create a sign-up or sign-in user flow as described in the previous module. Remember to select Facebook as the identity provider.
4. Select the user flow and click Run.

References

The following references are provided for detailed steps to integrate additional IDPs with Azure AD B2C.

- Microsoft
- Facebook
- Google+
- Amazon
- LinkedIn
- Twitter

- GitHub
- Weibo
- QQ
- WeChat

Important terms and concepts

- Identity Providers (IDPs)
- Authentication protocols supported by Azure AD B2C

Module 3: Application integration for a .NET web app

Introduction

In this module, you will download a sample web app and configure it to utilize your Azure AD B2C tenant and user flows created in Module 1. You will then publish the app and examine the claims associated with an id_token. The web app itself is very simple; all it does is configure the OWIN middleware that implements the OAuth 2.0 authentication protocol to integrate with your Azure AD B2C user flows, and then display the resulting identity token when the user has connected.

At the end of this module, you will have registered an app, edited the app to incorporate user flows, and edited the authorization token path.

And, you will have a deeper understanding of the following concepts:

- How to register a relying party app
- How to configure an app to utilize Azure AD B2C user flows
- How to install OWIN middleware
- Examine claims using jwt.ms

This module should take approximately 45 minutes to complete.

Steps to enable a web app to authenticate with Azure AD B2C accounts

1. Complete the following tutorial: Enable authentication in a web application using Azure AD B2C
2. **Examine the OWIN middleware configuration:**
   a) In Solution Explorer, in the **AwesomeComputers** project, expand the **App_Start** folder, and then select the **Startup_Auth.cs** file. This file contains the code that configures the OWIN middleware when the web app starts running.
   b) Examine the **ConfigAuth** method. This is the code that initializes the OWIN middleware.
   c) The remaining methods in the file, **OnRedirectToIdentityProvider**, **OnAuthenticationFailed**, and **OnAuthorizationCodeReceived** are handlers that respond to authentication and authorization events raised by the OWIN middleware. Note that the **OnAuthenticationFailed** method also handles password resets.
3. **Publish the Sample Application:** Following the documentation, publish the sample app
   - App Name: **AwesomeYourLastName**
   - Resource Group: **AwesomeComputersApp**
   - Hosting Plan: Create a new app service.

When the app has deployed, it will launch in the browser and display the sign-in page. ***Do not sign in yet***, just close the browser window.

4. **Test the deployed application**
    a. In a web browser, navigate to your app web address, **https://AwesomeYourLastname.azurewebsites.net**.
    b. In the sign-in page, log in as **TestUser@AwesomeYourLastname.onmicrosoft.com**.
    c. Verify that you are signed in successfully.
    d. Select **View Token Contents**. You will be presented with a page generated by using the web site at https://jwt.ms which you used in Module 1 to display the identity token for the signed-in user.
    e. Select **Claims**, and view the claims passed to the web app by the sign-up/sign-in user flow.

Note: If you select **Back To B2C Training**, you will be taken to the aadb2ctraining web site.

Important terms and concepts

- Azure AD B2C: Types of apps
- Azure AD B2C: Token reference
- **Incorporating Azure AD B2 into existing apps**

To incorporate Azure AD B2C into existing apps you need to perform the following steps:

1. Ensure the app is registered in the Azure AD B2C tenant so that the app can receive tokens and ensure the reply URL is set to the app authorization path.

2. Integrate Azure AD authentication, MSAL libraries, and OpenID Connect middleware into the app project.

3. Change the configuration files to include the following information:

    a. Azure AD B2C tenant ID

    b. Application ID and secret

    c. Azure AD B2C user flow names which you intend to use in the app

    d. Handle the Azure AD B2C notifications

    e. Ensure that the app has the correct permissions to any resources i.e. Azure AD B2C permissions to read directory data

- Azure AD V1 vs V2 endpoints

Module 4: UX customization using built-in functionality

Introduction

In this module, you will customize the user experience of the user flows implemented by your Azure AD B2C app to match the corporate look and feel of the Awesome Computers organization. You will also add localization to the user flows to support Spanish-speaking users, and you will increase the password complexity requirements for users to make the app more robust to attack.

See the end user experience you are about to build by launching Module 4 from the demonstration site.

In the demonstration site, you will perform the following tasks:

- Login on the customized screen.

At the end of this module, you will be able to:

- Customize the HTML pages displayed by Azure AD B2C user flows. To achieve this, you will:
    a. Create a storage account for holding customized content.
    b. Create HTML content for the profile edit user flow and the sign-up/sign-in user flow.
    c. Upload the HTML content to blob storage
    d. Update the sign-up/sign-in and profile edit user flows.
- Customize email messages sent by the Azure AD B2C app.
- Add localization to support Spanish-speaking users.
- Modify the password complexity of user flows.

This module should take approximately 40 minutes to complete.

Customize the HTML pages displayed by Azure AD B2C user flows

You can customize the Azure AD B2C user interface to match the look and feel of your existing apps and provide a seamless experience to users. Specifically, you can customize the pages that appear in response to the combined sign-up/sign-in, sign-up, sign-in, profile editing, and password reset user flows.

To customize the interface, you provide the HTML5 content in the form of HTML fragments and CSS styling. The content includes placeholders where Azure AD B2C-specific content is generated and inserted when the user flows run. You save your HTML5 content to a convenient and accessible location and provide the URL of this content to your user flows; different user flows can reference different HTML fragments. At run-time, the user flow retrieves your content and merges it with content that it generates for the placeholders. Note that the site that hosts your content should expose an HTTPS endpoint with Cross-Origin Resource Sharing (CORS) enabled. The GET and OPTIONS request methods must both allow CORS.

Follow these steps to create a custom HTML page, upload it to the cloud, and associate it with a user flow in the Azure AD B2C tenant.

1. Create HTML Content for the profile edit user flow and the sign-up/sign-in user flow
2. On your computer, create a new folder named **PolicyUX** under the **B2Course-SampleApp** folder. Copy the subfolders named **css, images**, and **fonts** and their content from the **ui-customizations** folder under the sample app to the **PolicyUX** folder.
3. Using Visual Studio, create a custom HTML5 page for the profile edit user flow, as follows.
    a. Replace the URLs highlighted in bold with the address of the blob storage container you created earlier.
    b. Name the file **updateprofile.html** and save it in the **PolicyUX** folder you created in the previous step with the following HTML.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Update Profile</title>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport"
          content="width=device-width, initial-scale=1">
    <!-- TODO: favicon -->
    <link
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.
css"
        rel="stylesheet" type="text/css" />
    <link
href="https://awesomeyoursurnameb2cux.blob.core.windows.net/b2c/css/glob
al.css"
        rel="stylesheet" type="text/css" />
  </head>
  <body>
    <div class="container self_asserted_container">
      <div class="row">
        <div
          class="col-md-6 col-md-offset-3 col-sm-8 col-sm-offset-2">
          <div class="panel panel-default">
            <div class="panel-body">
              <div class="image-center">
                <img alt="Awesome Computers" class="login-logo"
```

```
src="https://AwesomeYourLastnameUX.blob.core.windows.net/b2c/images/logo
.jpg" />
              </div>
              <h3 class="text-center">Update your current profile</h3>
              <div id="api" data-name="SelfAsserted">
              </div>
            </div>
          </div>
        </div>
      </div>
    </div>
  </body>
</html>
```

The page references a stylesheet and image files that provide a corporate look and feel that matches the styling used by other apps published by Awesome Computers. You will upload the CSS and image files to blob storage. Note that the URLs for these items reference the container you created in the previous step.

The **<div>** highlighted in bold is the placeholder that will be replaced with content generated by the user flow at runtime. It is important to set the **id** attribute of this section to **api**. The content generated by the user flow uses a set of specific HTML class and id attributes. These attributes are documented online. The CSS file contains styles that modify the way in which elements in this content are displayed.

4. Create another HTML page for customizing the sign-up/sign-in user experience. Again, specify the URLs for your blob storage account where highlighted. Save this page in the same folder, and name it **unified.html**. Again, notice that this page includes a **<div>** named **api**.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sign in</title>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <!-- TODO: favicon -->
    <link
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css"
        rel="stylesheet" type="text/css" />
    <link
href="https://AwesomeYourLastnameUX.blob.core.windows.net/b2c/css/global.css"
        rel="stylesheet" type="text/css" />
  </head>
  <body>
    <div class="container unified_container">
      <div class="row">
        <div class="col-md-6 col-md-offset-3 col-sm-8 col-sm-offset-2">
          <div class="panel panel-default">
            <div class="panel-body">
              <div class="image-center">
                <img alt="Awesome Computers" class="login-logo"

src="https://AwesomeYourLastnameUX.blob.core.windows.net/b2c/images/logo.jpg"
/>
              </div>
              <h3 class="text-center">Sign in with your existing account</h3>
              <div id="api" data-name="Unified">
              </div>
            </div>
          </div>
        </div>
      </div>
    </div>
  </body>
</html>
```

5. Upload the HTML content to blob storage
6. Update the sign-up/sign-in and profile edit flows to refer to the new content

Customize email messages sent by the Azure AD B2C app

When a user signs up or needs to reset their password, Azure AD B2C responds by sending emails containing confirmation codes to the user's registered email address. These emails have a default style, but you can use the Azure AD company branding feature to modify features such as the logo displayed at the bottom of the email, the signature, and the background color of the banner displayed at the top.

Refer to these guides to implement email branding in your tenant:

1. How do I customize verification emails
2. Add branding to your organization's Azure AD

Bring your own email provider

Instead of customizing the email messages sent by Azure AD B2C, you can also choose to entirely customize the email sender domain, as well as the look and feel of your verification email by bringing your own email provider. While the configuration of a custom email provider is outside the scope of this training course, you can refer to our online documentation on onboarding SendGrid or MailJet as custom email provider into your Azure AD B2C implementation.

Add localization to support Spanish-speaking users

Localization enables Azure AD B2C to present the HTML pages displayed by your app user flows in the language most appropriate to the user and their location. The language to use is passed as a parameter in the HTTP requests sent to Azure AD B2C.

Azure AD B2C provides automatic translation of the HTML content that it generates, depending on the user's login locale, or the locale specified by the browser. You can also customize the way in which translations are performed and provide your own translation strings.

Follow these instructions to add localization to each of your user flows: Select which languages in your user flow are enabled.

Modify the password complexity of user flows

Azure AD B2C can detect brute-force password attacks and dictionary password attacks. It can differentiate between users attempting to recall or mistyping a password, and hackers or botnets. In the event of a suspected attack, Azure AD B2C will lock the affected account. Additionally, to help prevent successful attacks, users should create passwords that are sufficiently complex. You can use the following steps to configure password complexity and enforce custom complexity rules when users sign up or reset their passwords. Note that password strength applies to sign-up/sign-in, password reset, and sign-in user flows, and that you must configure each of these user flows separately.

Follow these instructions to modify the default password complexity rules in your tenant: Password customization in Azure AD B2C

Important terms and concepts

Built-in localization strings for Azure AD B2C user flows, and customization

The language customization feature of Azure AD B2C performs the automatic translation of text generated as part of the HTML content for the various user flows that you define. As noted on the Language customization page in Azure AD B2C:

"*Microsoft is committed to providing the most up-to-date translations for your use. Microsoft continuously improves translations and keeps them in compliance for you. Microsoft will identify bugs and changes in global terminology and make updates that will work seamlessly in your user journey.*"

The translations are based on a predefined set of built-in strings displayed by the user flows. You can override the translations performed and provide your own localized text if the default translations do not meet your requirements, or if you need to display different or additional information.

Each phrase on each page displayed by the user flows is associated with an identifier. You can download the list of identifiers and the corresponding phrases from the **Language customization** page of a user flow to a local file. You can then edit this file to provide your own translations for any phrases and upload this file to the user flow.

Module 5: Introduction to custom policies

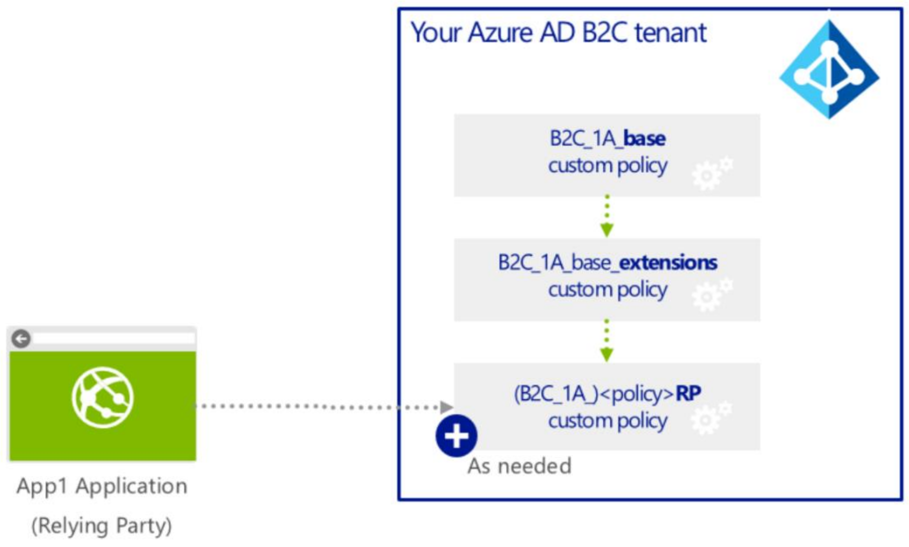
Introduction

Built-in user flows provide a set of standard behaviors that are useful in the most common situations in Azure AD B2C and are not customizable. If you need to implement some form of non-standard or extended behavior, you can implement custom policies. The documentation on Custom policies in Azure AD B2C covers the difference between built-in user flows and custom policies in more detail.

You describe the features implemented by a custom policy by using a set of XML files. These files can define the claims schema, claims transformations, content definitions, claims providers/technical profiles, and user journey orchestration steps, among other elements. You can define a hierarchy of XML files to ease maintenance and provide consistency across policies. The Microsoft recommended approach is to use three types of policy files:

1. A BASE file. This file contains the definitions that are shared all policies in your tenant. Any changes to this file can affect all the policies that depend on it. We recommend not changing this file unless absolutely necessary.
2. An EXTENSIONS file. This file specifies elements that override the base configuration for your tenant.
3. A Relying Party (RP) file. This is a single task-focused file that is invoked directly by the app or service. There could be many RP files depending on RP requirement.

When an app calls the RP policy file, the Identity Experience Framework (IEF) in Azure AD B2C will assemble the policy file by adding all the elements from BASE, then from EXTENSIONS, and lastly from the RP policy file.


Your Azure AD B2C tenant
B2C_1A_base custom policy
B2C_1A_base_extensions custom policy
(B2C_1A_)<policy>RP custom policy
As needed
App1 Application (Relying Party)

At the end of this module, you will be able to:

- Create a custom policy.
- Test and debug a custom policy.
- Modify an existing custom policy.

And, you will have a deeper understanding of the following concepts:

- When to use custom policies rather than built-in user flows
- The Identity Experience Framework (IEF)

This module should take approximately 60 minutes to complete.

See it in action

See the end user experience you are about to build.

In the demonstration site, access the Module 5 area and perform the following tasks:

- Sign up with your email and a password.
- Edit your profile to include a "1" at the end of your last name.
- Change your password and experience the multi-factor authentication process.

Create a custom policy

To create a custom policy, you require the following items:

- Signing and encryption keys
- Azure AD apps that enable users to sign up and sign in.

Complete the following steps:

1. Add signing and encryption keys
2. Register IEF applications
3. Download the custom policy starter set and configure tenant name
    a. TrustFrameworkBase.xml – 2 changes for tenant name
    b. All other policies – 3 changes for tenant name
4. Add application Ids to the policy
    a. TrustFrameworkExtensions.xml – 4 changes total
5. Edit the **TrustFrameworkExtensions.xml**, and replace the <BuildingBlocks> element with the content below. Replace **awesome*yoursurname*b2cux** with the name of the Azure storage account you created in Module 4 to hold the custom **unified.html** fragment for the sign-up/sign-in page. Replace **b2c** with the name of the blob storage container you created.:

```
<BuildingBlocks>
  <ContentDefinitions>
```

```
    <ContentDefinition Id="api.signuporsignin">
        <LoadUri>https://AwesomeYourLastnameb2cux.blob.core.windows.net/b2c/unified.htm
l</LoadUri>
    </ContentDefinition>

    <ContentDefinition Id="api.localaccountsignup">
      <LoadUri>https://AwesomeYourLastnameb2cux.blob.core.windows.net/b2c/unified.html<
/LoadUri>
    </ContentDefinition>

    <ContentDefinition Id="api.selfasserted.profileupdate">
      <LoadUri>https://AwesomeYourLastnameb2cux.blob.core.windows.net/b2c/updateprofile
.html</LoadUri>
    </ContentDefinition>

    <ContentDefinition Id="api.localaccountpasswordreset">
      <LoadUri>https://AwesomeYourLastnameb2cux.blob.core.windows.net/b2c/unified.html<
/LoadUri>
    </ContentDefinition>

  </ContentDefinitions>
</BuildingBlocks>
```

6. Upload the custom policies to your tenant

Update the web application to reference the new policies

1. Using Visual Studio, return to the **B2C-WebAPI-DotNet.sln** solution for the web app you configured and published in Module 3.
2. In Solution Explorer, in the **AwesomeComputers** project, select the **Web.config** file.
3. In the **<appSettings>** section near the start of the file, change the values for the policy references to specify the new custom policy files, as follows:

```
<appSettings>
 …
 <add key="ida:SignUpSignInPolicyId" value="B2C_1A_signup_signin" />
 <add key="ida:EditProfilePolicyId" value="B2C_1A_ProfileEdit" />
 <add key="ida:ResetPasswordPolicyId" value="B2C_1A_PasswordReset" />
</appSettings>
```

4. Save the file.
5. In Solution Explorer, right-click the **AwesomeComputers** project, then select **Publish**.

6. On the **Publish** page, select **Publish**, and wait for the web app to be deployed. When the sign-in page appears in the web browser, close the page (do not sign in).

7. Return to the Azure AD B2C blade in the Azure portal.

8. Under the **Policies** section, select **All policies**, and then select the **B2C_1A_signup_signin** policy.

9. Select **Run now**. When the sign-in page appears, it should be correctly styled, and include the option to sign in using Facebook.

10. Sign in as **TestUser@*AwesomeYourLastname*.onmicrosoft.com**. When the Awesome Computers page appears after signing in, select **Edit Profile**. The edit profile should also be correctly styled.

11. Select **Continue**. When the Awesome Computers page reappears, select **Reset Password**. This page should have the same styling.

12. Close the web app.

Modify the starter pack sign-up/sign-in policy

You can use a custom policy to include additional sign-up/sign-in functionality. In this procedure, you will see how to incorporate a journey step into the sign-up/sign-in process that prompts the user to accept the organization's terms and conditions, add a claim to the sign-up page, and validate input using an external REST API.

1. Using Visual Studio, open the **TrustFrameworkBase.xml** file.

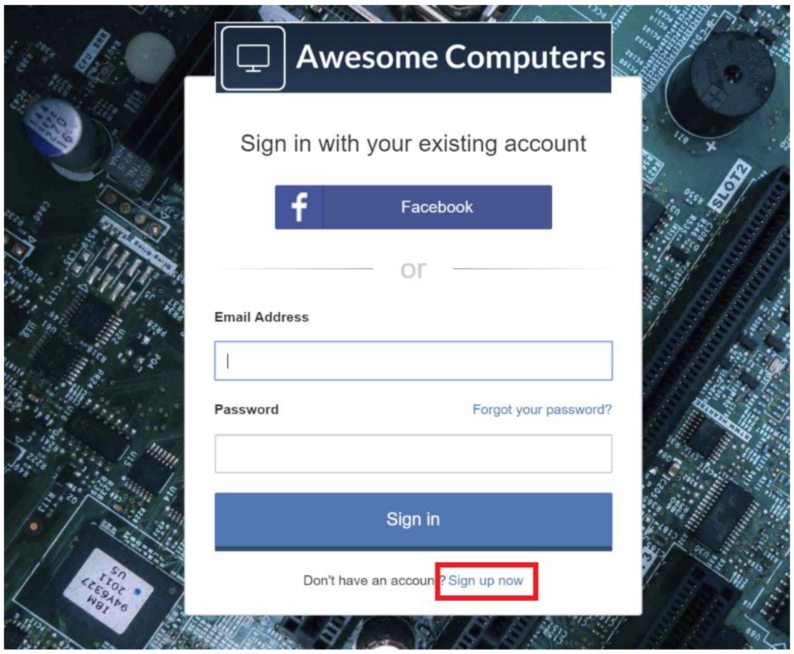
2. Inside the <ClaimsSchema> node, insert the following claim type.

```
<ClaimType Id="TnCs">
  <DisplayName>Terms of Service Consent</DisplayName>
  <DataType>string</DataType>
  <UserHelpText>I agree to the Awesome Computers terms of
service.</UserHelpText>
  <UserInputType>CheckboxMultiSelect</UserInputType>
  <Restriction>
    <Enumeration Text="I agree to the Awesome Computers terms of
service."
     Value="4/19/2018" SelectByDefault="false" />
  </Restriction>
</ClaimType>
```

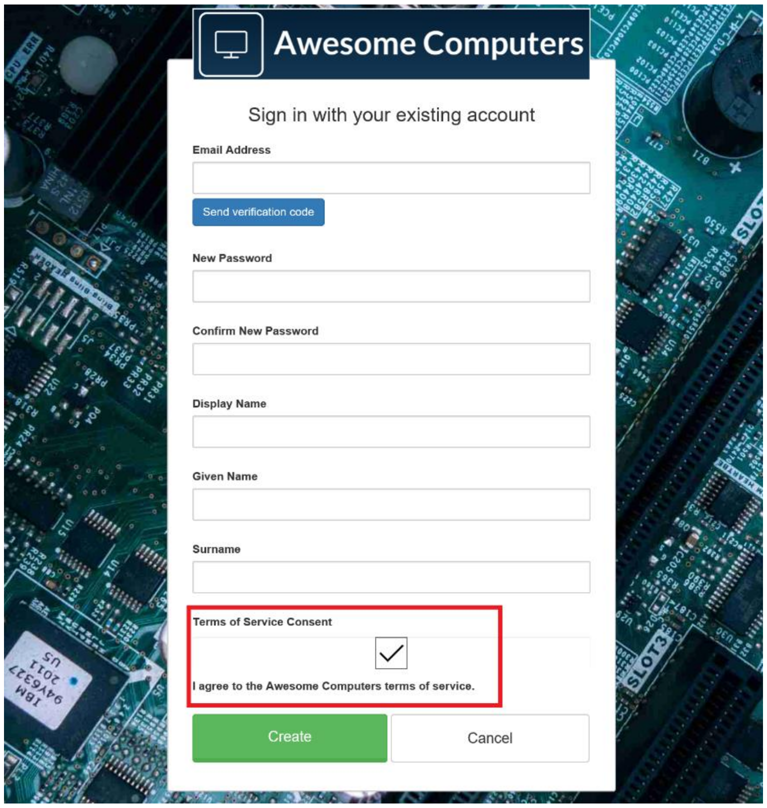
3. Find the element `<TechnicalProfile Id="LocalAccountSignUpWithLogonEmail">`, and in the <OutputClaims> node, add the following element.

```
<OutputClaim ClaimTypeReferenceId="TnCs" Required="true" />
```

4. Save the file.
5. Return to the Azure portal and switch to your Azure AD B2C tenant. Open the **Azure AD B2C Blade** and select **Identity Experience Framework.**
6. Select **Upload policy** and upload the **TrustFrameworkBase.xml** policy file. Select **overwrite the policy if it exists**.
7. To add a claim to the sign-up page, open the **SignUpOrSignIn.xml** file using Visual Studio.
8. Find the element `<TechnicalProfile Id="PolicyProfile">`, and in the `<OutputClaims>` node, add `<OutputClaim ClaimTypeReferenceId="TnCs"/>`.
9. Save the file.
10. In the Azure portal, upload the **SignUpOrSignIn.xml** policy file and overwrite the existing policy.
11. Select the **B2C_1A_signup_signin** policy.
12. Select **Run now**.
13. In the sign-in page, select **Sign-up now**.



14. In the sign-up page, notice that the **Terms of Service Consented** checkbox appears.


Awesome Computers

Sign in with your existing account

Email Address

Send verification code

New Password

Confirm New Password

Display Name

Given Name

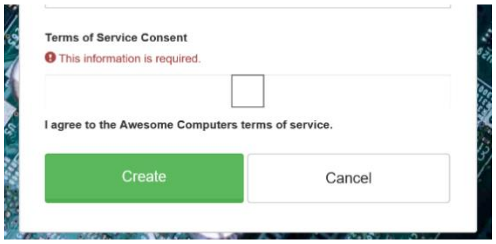
Surname

Terms of Service Consent

✓

I agree to the Awesome Computers terms of service.

15. Attempt to create an account without checking **Terms of Service Consent**. This should fail with the message **This information is required**. You should only be able to proceed once you have checked the box.


Terms of Service Consent

This information is required.

I agree to the Awesome Computers terms of service.

16. Verify that you can create an account if you select the **Terms of Service Consent** box.

Important terms and concepts

When to use built-in user flows versus custom policies

Built-in user flows implement the behavior most commonly required by tasks such as sign-in, sign-up, password reset, and profile editing. They also support scenarios such as authentication through trusted parties, including Facebook, Google, and LinkedIn. If you have no special identity requirements, using the built-in user flows will get you up and running with minimum fuss.

Custom policies are useful if you have special requirements, such as the use of external IDPs. You can use custom policies to:

- Interact with providers that implement the OIDC, OAuth, and SAML protocols.
- Include additional steps in the identity process, such as asking the user to confirm acceptance of terms and conditions when they sign up to a service.
- Perform custom operations, such as transposing requests when RPs and AtPs (attribute providers) do not support the same protocol.

Custom policies provide you with full control over identity behavior and experience. Azure AD B2C custom policies enable you to author your own trust framework. For example, they control how the identity information is exchanged between RPs, IDPs and AtPs. They include all the operational details including claim providers, metadata and technical profiles, claims schema definitions, claims transformation functions, and user journeys necessary to facilitate operational orchestration and automation.

You can define the behavior of custom policies through XML configuration files, and then apply them to your Azure AD B2C container using the IEF.

Module 6: Custom Policies 2 – REST APIs

Introduction

At the end of this module, you will be able to integrate your own REST APIs into custom policies to perform tasks such as validating input claims and sending back output claims. You will be able to:

1. Create a custom policy for REST API integration
2. Modify the profile edit policy to consume data from an external API

See it in action

In the demonstration site, access Module 6 and perform the following tasks:

1. Sign in with a previously created account.
2. When the Store Membership Number field appears, enter a multiple of 5
    a. Only multiple of five will successfully validate against the API

This module will take 60 minutes to complete.

The Azure AD B2C Identity Experience Framework (IEF) provides complete control for configuring policies. It enables you to integrate identity providers with external REST APIs through user journeys. IEF sends and receives data in form of claims. You can configure a policy to exchange claims and perform complex validation of identity information through a REST API by adding orchestration steps inside the user journey.

By the end of this module, you will be able to create a user journey that interacts with a RESTful service.

Prerequisites

Note that this module assumes that you have completed Module 5 and created the XML files that define the custom policies used by the sample web app.

Step-by-step instructions

Create a custom policy for REST API integration

You will create a RESTful service that validates user input and sends an error message if the input data is not valid. The data that the service validates is part of the identity information provided when a user signs up to your app.

Create an API app

1. Start Visual studio
2. On the **File** menu, select **New** and then select **Project**
3. In the **New Project** dialog box, expand **Visual C#**, select **Web**, and then select **ASP.NET Core Web Application**
4. Name the project **StoreMembershipApi**, and then select **OK**
5. In the **New ASP.NET Core Web Application** dialog box, select **Web Application (Model View Controller)**, select **No Authentication** and then select **OK**

Add data models and a controller to the API project

The user will provide a store membership number at the time of registration. The RESTful service will validate this number, and if it is verified, the user is allowed to proceed with registration. If the number is invalid, the service will send back a *validation failed* error.

In this example, the validation is simple; the provided store membership number must be an integer that is divisible by 5.

1. In the Solution Explorer, right-click **Models**, select **Add**, and the select **Class**
2. In the **Add New** Item dialog box, select **Class**. Specify the name **ValidationResponseContent**, and then select **Add**
3. Replace the code in the **ValidationResponseContent.cs** file with the following class:

```
namespace StoreMembershipApi.Models
{
    public class ValidationResponseContent : StoreResponseContent
    {
        public string StoreMembershipNumber { get; set; }
    }
}
```

4. Using the same procedure, add another class to the **Models** folder, named **StoreResponseContent**. Replace the code for the class with these statements:

```
using System.Net;
using System.Reflection;

namespace StoreMembershipApi.Models
{
    public class StoreResponseContent
    {
        public string Version { get; set; }
        public int Status { get; set; }
        public string UserMessage { get; set; }

        public StoreResponseContent()
        { }

        public StoreResponseContent(
          string message,
          HttpStatusCode status)
        {
            this.UserMessage = message;
            this.Status = (int)status;
            this.Version = Assembly
                               .GetExecutingAssembly()
                               .GetName()
                               .Version.ToString();
        }
    }
}
```

5. Add a further class to the **Models** folder, named **MembershipRequest**. Replace the code generated for this class with the following statements:

```
namespace StoreMembershipApi.Models
{
    public class MembershipRequest
    {
        public int StoreMembershipNumber { get; set; }
    }
}
```

6. In the Solution Explorer, right-click the **Controllers** folder, select **Add**, and then select **Controller**

7. In the **Add Scaffold** dialog box, select **API Controller -Empty**, and then select **Add**

8. In the **Add** Controller dialog box, name the controller **MembershipController**, and then select **Add**

9. Replace the code generated for the controller with the following.

```
using System;
using System.Net;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using StoreMembershipApi.Models;

namespace StoreMembershipApi.Controllers
{
    /// <summary>
    /// This controller is responsible for responding to requests from our
    /// custom policies to validate store membership numbers and retrieve
    /// customer membership dates.
    /// </summary>

    [Route("api/[controller]")]
    public class MembershipController : Controller
    {
        /// <summary>
        /// This method receives a storeMembershipnumber from the policy
        /// validation step and returns a 409 Conflict response if the
        /// store membership number is not valid (not a multiple of 5),
        /// and a 200 Ok response on successful validation of the store
        /// membership number.
        /// </summary>
        /// <param name="request">
        ///     Passed from the policy validation step, contains a
        ///     storeMembershipNumber.
        /// </param>
        /// <returns>
        ///     HTTP 200 Ok on success. HTTP 409 Conflict if provided an
        ///     invalid store membership number.
        /// </returns>
        [HttpPost("validate")]
        public IActionResult ValidateMembershipNumber(
            [FromBody] MembershipRequest request
            )
        {
            if (!IsStoreMembershipNumberValid(
              request.StoreMembershipNumber))
            {
                return GenerateErrorMessageWithMessage(
                    "Store membership number is not valid, it " +
                    "must be a multiple of 5!");
            }

            // Return the output claim(s)
            return Ok(new ValidationResponseContent
            {
                StoreMembershipNumber =
                  request.StoreMembershipNumber.ToString()
            });
        }

        /// <summary>
        /// Constructs an HTTP 409 Conflict IActionResult to return back
        /// to the validation or orchestration step. This is used to
        /// communicate with the policy steps to communicate an error
        /// state.
        /// </summary>
        /// <param name="message">
        ///     The message to be passed back to the user to explain why
        ///     the request failed.
        /// </param>
        /// <returns>
        ///     An IActionResult representing an HTTP 409 Conflict with a
        ///     custom payload.
        /// </returns>
        private IActionResult GenerateErrorMessageWithMessage(
          string message)
        {
            return StatusCode(
                (int)HttpStatusCode.Conflict,
                new StoreResponseContent
                {
                    Version = "1.0.0",
                    Status = (int) HttpStatusCode.Conflict,
                    UserMessage = message
                });
        }

        /// <summary>
        ///     Validates a provided store membership number by using the
        ///     modulus operator (see more <see
href="https://docs.microsoft.com/dotnet/csharp/language-
reference/operators/remainder-operator">here</see>)
        ///     to determine if the provided store membership number is a
        ///     multiple of 5. If it is, we return true, if not, we return
        ///     false.
```

```
            /// </summary>
            /// <param name="storeMembershipNumber">
            ///     The store membership number to be validated.
            /// </param>
            /// <returns>
            ///     True on successful validation, false if validation fails.
            /// </returns>
            private bool IsStoreMembershipNumberValid(
              int storeMembershipNumber)
            {
                if (storeMembershipNumber % 5 != 0)
                {
                    return false;
                }

                return true;
            }
        }
    }
```

10. On the **Build** menu, select **Rebuild Solution**, and verify that the solution builds without any errors

Publish the project as an Azure website

1. In the Solution Explorer, right-click the **StoreMembershipApi** project node, and then select **Publish**

2. In the **Pick a publish target** dialog box, select **App Service**, select **Create New**, and then select **Publish**

3. In the **Create App Service** dialog box, login to Azure as **B2CAdmin** in your Azure domain, specify a unique App Name, select a resource group, and then select **Create**

4. Wait while the service is published. The site will launch in the browser when publishing is complete

Record the URL of your service in the browser address bar.

Update custom policies

1. In Visual Studio and open the **TrustFrameworkExtensions.xml** policy file that you created in Module 5

2. In the <BuildingBlocks> node add the following <ClaimsSchema> element

```
<ClaimsSchema>
  <ClaimType Id="extension_storeMembershipNumber">
    <DisplayName>Store Membership Number</DisplayName>
    <DataType>string</DataType>
    <UserHelpText>Your store membership number</UserHelpText>
    <UserInputType>TextBox</UserInputType>
  </ClaimType>
</ClaimsSchema>
```

3. In the <ClaimsProviders> node, add the following <ClaimsProvider> element. Replace the address highlighted in bold with the URL of your REST web service.

```
<ClaimsProvider>
    <DisplayName>REST APIs</DisplayName>
    <TechnicalProfiles>
        <TechnicalProfile Id="ValidateStoreMembershipNumber">
            <DisplayName>Validate Store Membership Number</DisplayName>
            <Protocol
                Name="Proprietary"
                Handler="Web.TPEngine.Providers.RestfulProvider,
Web.TPEngine, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
            <Metadata>
                <Item
Key="ServiceUrl">https://yourwebservice.azurewebsites.net/api/membership/v
alidate</Item>
                <Item Key="AuthenticationType">None</Item>
                <Item Key="SendClaimsIn">Body</Item>
            </Metadata>
            <InputClaims>
                <InputClaim
                    ClaimTypeReferenceId="extension_storeMembershipNumber"
                    PartnerClaimType="storeMembershipNumber" />
            </InputClaims>
            <UseTechnicalProfileForSessionManagement
                   ReferenceId="SM-Noop" />
        </TechnicalProfile>
        <TechnicalProfile Id="LocalAccountSignUpWithLogonEmail">
            <OutputClaims>
                <OutputClaim
                    ClaimTypeReferenceId="extension_storeMembershipNumber"
                    PartnerClaimType="storeMembershipNumber" />
            </OutputClaims>
            <ValidationTechnicalProfiles>
                <ValidationTechnicalProfile
                    ReferenceId="ValidateStoreMembershipNumber" />
            </ValidationTechnicalProfiles>
        </TechnicalProfile>
    </TechnicalProfiles>
</ClaimsProvider>
```

4. Save the file.

5. Using the Identity Experience Framework in the Azure portal, upload the **TrustFrameworkExtensions.xml** file, and overwrite the existing policy.

6. In Visual Studio, open the **SignUpOrSignIn.xml** file in your working directory.

7. Find the element `<TechnicalProfile Id="PolicyProfile">`, and in the `<OutputClaims>` node add the following XML markup:

   ```
   <OutputClaim ClaimTypeReferenceId="extension_storeMembershipNumber" />
   ```

8. Save the file.

9. Using the Identity Experience Framework in the Azure portal, upload the **SignUpOrSignIn.xml** file, and overwrite the existing policy.

Test the custom policy

1. In the Azure portal, select **Azure Active Directory**, and then select **Users**.

2. Select the user you created in module 5, and then select **Delete user**. Select **Yes** to confirm the deletion.

   Note: The purpose of these two steps was to enable you to reuse the same user account in the following test.

3. In the Azure AD B2C tenant, Select the **B2C_1A_signup_signin** and **Run**.

You can visit our online documentation for more information about how to integrate REST API claims exchanges in your Azure AD B2C user journey.

Modify the profile edit policy to return the user store membership date

In this task, you will add an additional step to the RESTful service. This step will read the input user claims and return the Store Membership Date.

1. Return to the **StoreMembershipApi** project in Visual Studio.

2. In Solution Explorer, right-click **Models**, select **Add**, and then select **Class.**

3. In the **Add New** Item dialog box, select **Visual C#**, and then select **Class**. Specify the name **MembershipDateResponseContent** and then select **Add**.

4. Replace the code in the **MembershipDateResponseContent.cs** file with the following code:

```
namespace StoreMembershipApi.Models
{
    public class MembershipDateResponseContent : StoreResponseContent
    {
        public string StoreMembershipDate { get; set; }
    }
}
```

5. In the Solution Explorer, expand the **Controllers** folder, and select the **MembershipController.cs** file**.**

6. Add the following methods to the **MembershipController** class.

```
/// <summary>
///     This method receives a storeMembershipnumber from the policy
///     orchestration step and returns a 409 Conflict response if the
///     store membership number is not valid (not a multiple of 5), and
///     a 200 Ok response containing the member's membership date if the
///     store membership number is valid.
/// </summary>
/// <param name="request">
///     Passed from the policy orchestration step, contains a
///     storeMembershipNumber.
/// </param>
/// <returns>
///     HTTP 200 Ok on success with an obtained membership date. HTTP 409
///     Conflict if provided an invalid store membership number.
/// </returns>
[HttpPost("membershipdate")]
public IActionResult GetMembershipDate(
  [FromBody] MembershipRequest request)
{
    if (!IsStoreMembershipNumberValid(request.StoreMembershipNumber))
    {
        return GenerateErrorMessageWithMessage(
            "Store membership number is not valid, it must be a " +
            "multiple of 5!");
    }

    var membershipDate = ObtainStoreMembershipDate();
    return Ok(new MembershipDateResponseContent
    {
        Version = "1.0.0",
        Status = (int)HttpStatusCode.OK,
        UserMessage = "Membership date located successfully.",
        StoreMembershipDate = membershipDate.ToString("d")
    });
}

/// <summary>
///     Generates a date within the last 90 days to return as the store
///     membership number for a member. In a production app you'd
///     likely contact a database here to get a real membership date for a
///     member.
/// </summary>
/// <returns>A DateTime representing the store membership date for a
member.</returns>
private static DateTime ObtainStoreMembershipDate()
{
    var random = new Random();
    var daysOffOfToday = random.Next(0, 90);
    var randomDate = DateTime.Now.AddDays(-daysOffOfToday);

    return randomDate;
}
```

7. Save the file

8. On the **Build** menu, select **Rebuild Solution**. Verify that the solution compiles without any errors.

9. In the Solution Explorer, right-click the **StoremembershipApi** project, and then select **Publish**.

10. In the **Publish** window, select **Publish**. Wait for the updated service to be deployed.

Update custom policies

1. Using Visual Studio, open the **TrustFrameworkExtensions.xml** policy file.

2. In the <BuildingBlocks> node, insert the following <ClaimType> to the collection in the <ClaimsSchema> node.

```
<ClaimType Id="extension_storeMembershipDate">
    <DisplayName>Store Membership Date</DisplayName>
    <DataType>string</DataType>
    <UserHelpText>Your store membership date</UserHelpText>
    <UserInputType>TextBox</UserInputType>
</ClaimType>
```

3. In the <ClaimsProviders> node, find the <ClaimsProvider> node that contains the <DisplayName>REST APIs</DisplayName> element. Add the following

<TechnicalProfile> node to this ClaimsProvider. Replace the address highlighted in bold with the URL of your REST web service.

```
<!-- Obtain claim technical profile -->
<TechnicalProfile Id="ObtainStoreMembershipDate">
    <DisplayName>Obtain Store Membership Date</DisplayName>
    <Protocol Name="Proprietary"
        Handler="Web.TPEngine.Providers.RestfulProvider, Web.TPEngine,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
    <Metadata>
        <Item
Key="ServiceUrl">https://yourwebservice.azurewebsites.net/api/membership
/membershipdate</Item>
        <Item Key="AuthenticationType">None</Item>
        <Item Key="SendClaimsIn">Body</Item>
    </Metadata>
    <InputClaims>
        <InputClaim
            ClaimTypeReferenceId="extension_storeMembershipNumber"
            PartnerClaimType="storeMembershipNumber" />
    </InputClaims>
    <OutputClaims>
        <OutputClaim
            ClaimTypeReferenceId="extension_storeMembershipDate"
            PartnerClaimType="storeMembershipDate" />
    </OutputClaims>
    <UseTechnicalProfileForSessionManagement
      ReferenceId="SM-Noop" />
</TechnicalProfile>
```

4. Uncomment the <UserJourneys> node at the end of the file, and add the following <UserJourney> element to this node.

```
<UserJourney Id="ProfileEditObtainApiClaim">
    <OrchestrationSteps>
        <OrchestrationStep Order="1"
            Type="ClaimsProviderSelection"
            ContentDefinitionReferenceId="api.idpselections">
            <ClaimsProviderSelections>
                <ClaimsProviderSelection
                    TargetClaimsExchangeId=
                         "LocalAccountSigninEmailExchange" />
            </ClaimsProviderSelections>
        </OrchestrationStep>
        <OrchestrationStep Order="2" Type="ClaimsExchange">
```

```
                <ClaimsExchanges>
                    <ClaimsExchange Id="LocalAccountSigninEmailExchange"
                        TechnicalProfileReferenceId=
                            "SelfAsserted-LocalAccountSignin-Email" />
                </ClaimsExchanges>
            </OrchestrationStep>
            <OrchestrationStep Order="3" Type="ClaimsExchange">
                <Preconditions>
                    <Precondition Type="ClaimEquals"
                       ExecuteActionsIf="true">
                        <Value>authenticationSource</Value>
                        <Value>socialIdpAuthentication</Value>
                        <Action>SkipThisOrchestrationStep</Action>
                    </Precondition>
                </Preconditions>
                <ClaimsExchanges>
                    <ClaimsExchange
                        Id="AADUserReadWithObjectId"
                        TechnicalProfileReferenceId=
                             "AAD-UserReadUsingObjectId" />
                </ClaimsExchanges>
            </OrchestrationStep>
            <OrchestrationStep Order="4" Type="ClaimsExchange">
                <ClaimsExchanges>
                    <ClaimsExchange
                        Id="B2CUserProfileUpdateExchange"
                        TechnicalProfileReferenceId=
                             "SelfAsserted-ProfileUpdate" />
                </ClaimsExchanges>
            </OrchestrationStep>
            <OrchestrationStep Order="5" Type="ClaimsExchange">
                <ClaimsExchanges>
                    <ClaimsExchange
                        Id="GetStoreMembershipDateData"
                        TechnicalProfileReferenceId=
                             "ObtainStoreMembershipDate" />
                </ClaimsExchanges>
            </OrchestrationStep>
            <OrchestrationStep Order="6" Type="SendClaims"
                CpimIssuerTechnicalProfileReferenceId="JwtIssuer" />
        </OrchestrationSteps>
        <ClientDefinition ReferenceId="DefaultWeb" />
    </UserJourney>
```

5. Save the XML file.

6. Using the Identity Experience Framework in the Azure portal, upload the **TrustFrameworkExtensions.xml** file, and overwrite the existing policy.

7. Using Visual Studio, open the **ProfileEdit.xml** file.

8. Under the <RelyingParty>, node replace the <DefaultUserJourney> node with the following XML element.

   ```
   <DefaultUserJourney ReferenceId="ProfileEditObtainApiClaim" />
   ```

9. Under the <TechnicalProfile Id="PolicyProfile">node, add the following XML markup below the <OutputClaims> node.

   ```
   <OutputClaim ClaimTypeReferenceId="extension_storeMembershipDate" />
   ```

10. Save the XML file.

11. Using the Identity Experience Framework in the Azure portal, upload the **B2C_1A_ ProfileEdit.xml** file, and overwrite the existing policy.

Test the custom policy

1. In the Azure portal, use the **Azure Active Directory** blade to delete the user you created in the previous test.

2. In the Azure AD B2C tenant, Select the **B2C_1A_signup_signin** and Run.

Module 7: External IDPs, OIDC and SAML

Introduction

At the end of this module, you will be able to:

- Add an OIDC IDP (Google) to Azure AD B2C using custom policies
- Integrate a SAML IDP and map claims (Salesforce) using custom policies

This module should take approximately 75 minutes to complete.

User management and configuring authentication can be a time-consuming process. It can also be error-prone if the authentication and identity requirements are complex, leading to possible issues with security. To reduce costs, it is sometimes worthwhile to consider using third party identity providers to manage and authenticate users, although integration multiple IDPs into a solution can also be a challenge. Using Azure AD B2C simplifies many of these tasks. This module shows two examples for configuring popular third-party external IDPs.

Note that this module assumes that you have completed Module 6 and created the XML files that define the custom policies used by the sample web app.

Integrate an OIDC IDP and map claims

As an example of using an OIDC IDP, the following procedure describes how to integrate Google as an IDP in the Identity Experience Framework.

- Create a Google+ app
- Setup Google+ app key and secret in Azure AD B2C
- Add a claims provider section to represent the Google authentication service
- Register the claims provider in the user journey
- Test the user journey

Integrate a SAML IDP and map claims

This example shows how to integrate the Salesforce IDP as an example of a SAML identity provider. If you don't have a Salesforce developer account you can sign up for a free developer account, and then setup My Domain.

Follow this tutorial to configure Salesforce as a SAML IDP in Azure AD B2C: Set up sign-in with a Salesforce SAML provider by using custom policies in Azure AD B2C

Module 8: Using the Azure AD Graph API and user migration

At the end of this module, you will be able to:

- Use the Azure AD Graph API to query and modify objects in an Azure AD tenant
- Migrate users to Azure AD by using the Azure AD Graph API
- Require users to change their passwords when they log in for the first time after being migrated

And, you will have a deeper understanding of the following concepts:

- The purposes of the Microsoft Graph API and the Azure AD Graph API, and when to use each
- Using Graph Explorer
- Planning user migrations

This module should take approximately 60 minutes to complete.

Introduction

In an on-premises environment, users are frequently authenticated using a local identity provider, possibly also located on-premises. When you migrate an app to Azure and move users to a directory such as Azure AD, you will most likely change authentication to use an identity provider provided by Azure. If your local directory contains many thousands of users, migrating these users and changing their identity configuration is a non-trivial operation. Typically, you would automate the tasks involved in this process. The Azure AD Graph API is designed to help with these tasks. Using the Azure AD Graph API, you can query, create, update, and delete identity and authentication information in Azure AD. The Azure AD Graph API is a REST service, and you can perform individual operations by submitting the appropriate HTTP requests to the API. Alternatively, you can write a custom app that uses the Azure AD Graph API to implement tasks against an Azure AD domain.

Users might be identified by using a social networking identity provider rather than a local IDP, and you can migrate these users also by using the Azure AD Graph API. Additionally, you can amend the password user flow of users once they have been migrated to enforce stronger passwords, and you can configure Azure AD B2C to require users to change their password at first sign-in after migration.

Use the Azure AD Graph API to query, add, update, and delete users in Azure AD

The Azure AD Graph API provides access to a web service that enables you to manage tenants in Azure AD. You can interact with the Azure Graph API from any environment that enables you to compose and submit HTTP requests.

Use the Azure AD Graph API for one-time or standalone queries
You can perform operations using the Azure AD Graph API by submitting HTTP requests through a tool such as PowerShell. Alternatively, you can also use Microsoft's Graph Explorer to execute queries. This tutorial covers several different potential approaches: How to use the Azure AD Graph API.

Use the Azure AD Graph API from a custom app
If you are performing many tasks manually, such as migrating a large number of users, the chances of making an error are high. A more considered approach is to create your own .NET app that automates the tasks required. The following steps configure and run a sample app, to illustrate this process (you can refer to the code online on the github page):

1. Register an app in your tenant
2. Configure permissions for your app
3. Download, configure, and build the sample graph app

Migrate users to Azure AD
Migrating users from your existing identity store
With Azure AD B2C, you can migrate users through Azure AD Graph API. The user migration process falls into two flows:

Pre-migration: This flow applies when you either have clear access to a user's credentials (username and password) or the credentials are encrypted, but you can decrypt them.

Pre-migration and password reset: This flow applies when a user's password is not accessible.

For a detailed tutorial on how to set up the two flows mentioned above, refer to the online documentation on Azure AD B2C: User migration.

Note: In cases where you do not have access to users' passwords in clear text, you may be able to use custom policies to implement a setup in which you concurrently maintain your legacy system as well as Azure AD B2C, and write the users' passwords into Azure AD B2C when the user next logs into the legacy system. This is done using an approach called 'Just In Time' or JIT Migration in which a user's password is concurrently written to Azure AD B2C. JIT migration is somewhat involved and a discussion on it is out of scope for this whitepaper. You can find a useful JIT Migration sample that illustrates this concept on GitHub.

Migrating users with social identities
When you plan to migrate your identity provider to Azure AD B2C, you may also need to migrate users with social identities. You can follow the online tutorial titled Azure AD B2C: Migrate users with social identities to enable this scenario in your tenant.

Important terms and concepts

The Azure AD Graph API and the Microsoft Graph API

Microsoft provides two APIs for managing information stored in Active Directory; the Azure AD Graph API which was designed to operate over tenants hosted by using Azure AD, and the more recent Microsoft Graph API. The Microsoft Graph API unifies many of the features available through the Azure Graph API with other services and Active Directory running on-premises. This unification has resulted in significant overlap between the two APIs. The purpose of this section is to summarize these APIs and describe when you should use them.

The Azure AD Graph API

An Azure AD tenant can contain a large number of objects (users, groups, profiles, permissions, and other items). While it might be feasible to perform some tasks manually, such as adding a new user, implementing an operation that spans a large set of objects in the directory (such as setting a permission for all users) is better performed programmatically. This approach can save time and reduce inconsistencies and errors that might otherwise occur. This is the role of the Azure AD Graph API.

The Azure AD Graph API is a service that provides a programmatic interface to Azure AD. With this API, you can perform tasks such as:

- Retrieve, add, and delete users, and modify the properties of users and their permissions and app licenses. You can maintain a hierarchy of users (users that have a direct report to another user, for example) and assign managers.
- Manage groups. You can organize users into groups and query group membership. A single user can belong to multiple groups, and you can assign permissions to a group (all users in that group inherit these permissions). Groups can contain sub-groups.
- Implement token issuance and token lifetime policies over apps, service principals, specific groups, or the entire organization.
- Manage directory roles. You can use directory roles to control the users and groups that can perform sensitive operations in a directory.
- Create, delete, update, and query domains within a tenant.

Another key feature of the Azure AD Graph API is that it enables you to query the relationships between groups of objects. Some of these relationships might involve a large set of objects, and the relationships themselves could be complex.

You perform operations on a tenant through the Azure AD Graph API by submitting HTTP requests to the API at https://graph.windows.net. All requests are authenticated, and you must provide a valid access token in the request header. The article Authorize access to web apps using OAuth 2.0 and Azure AD describes how to do this.

The general form of a request is:

https://graph.windows.net/{tenant_id}/{resource_path}?{api_version}[odata_query_parameters]

Replace *{tenant_id}* with the name of your Azure AD tenant, and *{resource_path}* with the resource that you are querying, modifying, adding, or deleting. Typically, you should specify *api-version=1.6* for the version number. If a request requires additional parameters, you specify them using OData operators, such as *$filter*, *$orderby*, *$expand*, *$top*, and *$format* For example, to retrieve a list of users whose names start with "A" in your organization, you can send this HTTP GET request:

https://graph.windows.net/myorganization/users?api-version=1.6&$filter=startswith(displayName,'A')

The result is an HTTP response message. The body contains a JSON array containing the matching details. Note that you can use the aliases *myorganization* to refer to the tenant that you are currently signed in to, and *me* to refer to the currently signed-in user.

To create a new user, you would submit a POST request like this:

https://graph.windows.net/myorganization/users?api-version=1.6

The request body must contain the details of the new user, for example:

```
{
  "accountEnabled": true,
  "displayName": "AAA BBB",
  "mailNickname": "AaaBbb",
  "passwordProfile": {
    "password": "Test1234",
    "forceChangePasswordNextLogin": false
  },
  "userPrincipalName": "AAA@mydomain.onmicrosoft.com"
}
```

You can find a full list of the HTTP requests that you can end to the Azure AD Graph API in the Azure AD Graph API Reference.

You can invoke Azure AD Graph API requests directly from the PowerShell command line, and you can use the Microsoft.IdentityModel.Clients.ActiveDirectory NuGet package in your own apps to construct and send requests programmatically; the sample B2CGraphClient app shows how to use this package in a C# app.

The Microsoft Graph API

The Microsoft Graph API enables you to build apps that connect users to their data. Like the Azure Graph API, it uses the relationships and objects defined in Active Directory and Azure AD, but can also operate over data held in Microsoft Office apps such as Excel. You can use this API to build custom tools and utilities to help improve productivity. For example, you can use the

Microsoft Graph API to view and maintain the associations between users and items such as their email, calendars, contacts, documents, devices, and other personal data sources. The Microsoft Graph API facilitates building services that can automate tasks such as scheduling meetings, notify interested users if a document changes, analyze usage patterns over documents and data files, construct custom dashboards, and establish a user's organizational context (which department they work in, who is their manager, and so on). See What can you do with Microsoft Graph? for detailed examples.

You send requests to the Microsoft Graph API using queries of the form:

https://graph.microsoft.com/{version}/{resource}?query-parameters

Replace *{version}* with the version of the API you are using (specify *v1.0* for the most recent implementation), and *{resource}* with the object that you are referencing. As an example, this query finds the last user to modify the file data.txt in the specified folder on the specified drive:

https://graph.microsoft.com/v1.0/me/drive/root/children/data.txt/lastModifiedByUser

Note that, as with the Azure AD Graph API, you must provide a valid access token in the security header of the request – see Get access tokens to call Microsoft Graph for details.

When to use which Graph API

New development of the Azure AD Graph API has stopped, and Microsoft is focusing on the Microsoft Graph API. In most cases, this means that while existing code that runs using the Azure AD Graph API will still be supported, you should consider using the Microsoft Graph API for new apps wherever possible. However, there are some situations where the functionality of the Azure AD Graph API is not yet available in the Microsoft Graph API. One example concerns Azure AD B2C. If you are working strictly with Azure AD B2C you should continue to use the Azure AD Graph API. As of this guide’s last update, Azure AD Graph is scheduled to be retired by June 2022.

Graph Explorer

Graph Explorer is a web app that provides a friendly user interface to the graph APIs. There are two versions available; one based on the Azure AD Graph API and another that uses the Microsoft Graph API. Both operate in a similar manner to send requests to your tenant and return results as an HTTP response. You provide your login credentials, and this information is included in the security header of each request.

Using the Azure AD Graph Explorer

To use the Azure AD Graph Explorer, perform the following steps:

1. Using a web browser, navigate to **https://graphexplorer.azurewebsites.net**.
2. In the title bar, select **Login**.
3. Provide the credentials of an account with administrative access to your Azure AD domain.

4. To run a query, select the **GET** verb, provide a URI that references the resources to find, such as https://graph.windows.net/myorganization/users?$filter=startswith(displayName,'A') to find all users whose name starts with the letter "A", select the API version to use, and then select **Go**. The results will appear in the main body of the window:
   Note that you can expand the Response Headers pane to see the HTTP header information included with the response.
5. To perform an operation such as adding a new user, select the POST verb, specify the URI of the resource that you are adding, provide the details of the resource as the message body, and then select **Go**. For example, to create a new user, specify a URI such as https://graph.windows.net/myorganization/users, and enter the information for the new user as the message body, in JSON format. You can find the details of the properties that you can specify for the different types of objects that you can create in Azure AD by using the Azure Graph API in the Azure AD Graph API Reference. If the operation is successful, the response message shows the details of the object that was created:
   Note that you must be logged in as an account that has administrative rights in the domain to create, delete, and modify objects in Azure AD.
6. To modify an object, select the **PATCH** verb, provide the URI of the resource that you wish to modify, and specify the new details for the resource as the message body. The following example disables the account just created)
   https://graph.windows.net/myorganization/users/johns@jscmtenant.onmicrosoft.com) by setting the **accountEnabled** attribute to **false**:
7. To remove an object, select the **DELETE** verb, and provide the URI of the object:

You can also look at our online documentation on how to get started with the Azure AD Graph API.

Using the Microsoft Graph Explorer

The Microsoft Graph Explorer is similar to the Azure AD Graph Explorer. You connect to it by using a browser and navigating to https://developer.microsoft.com/graph/graph-explorer. You should sign in to the Azure AD domain using a valid domain account:

You run queries by selecting the **GET** verb and specifying the URI of the resource you want to retrieve. You perform inserts, and updates by using the **PUT**, **POST**, and **PATCH** verbs and providing the details of the new or amended item (**POST** creates a new resource, **PATCH** modifies a resource, and **PUT** replaces an existing resource with a new one). You remove objects by selecting the **DELETE** verb and specifying the URI of the resource to remove.

Microsoft Graph Explorer can potentially query and amend a lot of your own data (calendars, email, contacts, and so on). By default, much of this data is inaccessible unless your explicitly grant access over it. To do this, select **modify permissions** and then select the objects and permissions that you want to enable:

You can also read more about the Microsoft Graph Explorer in our online documentation.

Planning user migrations

A key part of a successful migration to Azure AD is migrating your users to Azure. This migration process needs to be carefully planned to ensure that you obtain the result that you need, which is a clean directory service which contains all the relevant user information that your apps need to function correctly.

Identity types

When creating accounts in Azure AD, those accounts will be one of two types:

- Local accounts
- Social accounts

In addition, you can have combined accounts, where users' social accounts are linked to their local accounts in one identity. With local accounts, users will sign in, whereas with social accounts, they will sign up to your Azure AD B2C service.

Determining identity use cases

When carrying out this planning process, you must identify the use cases for which you want to employ identity data within Azure AD. You may have apps and resources that are for your internal employees only, along with other apps that are for your customers. Employees should log on to internal-only resources using their corporate credentials (sign-in), whereas customers will use another credential source to connect to your service (sign-up), such as a free cloud-based platform such as Outlook.com, Hotmail or Google, or a social media platform like Facebook or LinkedIn. Your initial landing page would direct the user to the correct option (sign-up or sign-in), depending on whether they are an employee or a customer.

You may have additional complications, such as the requirement to integrate federated identities from, say, a partner organization, where partners may have access to more apps than customers, but not the full range of internal resources that employees can open. For more information on Azure AD federation, see the *Azure AD federation compatibility list*.

If your use cases dictate that you will have multiple identity sources to connect to Azure AD, you will need to implement custom policies alongside Azure AD B2C and the Identity Experience Framework. For more information, see *Azure AD B2C: Custom policies*.

Access to External Apps

If you need to provide access to apps that are not in the Azure AD app gallery, you can implement this functionality without writing code in Azure ADPremium. Azure AD Premium then brings the following additional capabilities:

- Self-service integration of any app that supports SAML 2.0 identity providers (SP-initiated or IDP-initiated)
- Self-service integration of any web app that has an HTML-based sign-in page using password-based SSO
- Self-service connection of apps that use the SCIM protocol for user provisioning
- Ability to add links to any app in the Office 365 app launcher or the Azure AD access panel

You can enable users to connect to LOB SaaS apps that are currently not in the Azure AD app gallery, along with any third-party apps that you control, either published from your on-premises servers or running in the cloud.

For more information on configuring single sign-on in these cases, see *Configuring single sign-on to apps that are not in the Azure AD app gallery*

Identifying source identity providers

From your use cases, you should be able to identify which identity providers will act as the source for authenticating either employees, federated partners, or customers to Azure AD. With on-premises directory services, such as Active Directory, Oracle Directory Server, or 389 Directory Server, you will look to import the necessary user attributes into Azure AD.

Required attributes include the following values:

- **accountEnabled** – must be set to true for the account to be usable.
- **displayName** - The name to display in the address book for the user.
- **passwordProfile** - The password profile for the user (with social accounts, a password must be specified, but its value is ignored)
- **userPrincipalName** - The user principal name, for example, someuser@contoso.com. The user principal name must contain one of the verified domains for the tenant.
- **mailNickname** - The mail alias for the user. This value can be the same as **userPrincipalName**.
- **signInNames** - One or more **SignInName** records that specify the sign-in names for the user. Each sign-in name must be unique across the company/tenant. For social accounts only, this property can be left empty.
- **userIdentities** - One or more **UserIdentity** records that specify the social account type and the unique user identifier from the social identity provider.
- [optional] **otherMails** - For social account only, the user's email addresses

If you are intending to use Azure AD with Office 365 for your internal users, there are additional attributes that need to be synchronized. For more information, see *Azure AD Connect sync: Attributes synchronized to Azure AD*.

Cleaning up the source directory

Prior to any directory migration, we recommended carrying out a clean-up of your existing directory service, removing or merging old or duplicate user accounts and generally ensuring that your current IDP is in the best possible shape prior to running the migration itself. You are also recommended to run clean-up tools such as DCDIAG and NTDSUTIL METADATA CLEANUP in Active Directory to test and verify the directory service.

When you have cleaned up the source directories, you can use the Graph API to perform the migration.

Password migration and policies

When creating new user accounts in Azure AD, there is a difference in how passwords are imported, depending on the level of access that you have to users' passwords in the source IDP.

- If you can access or decrypt the users' passwords, those passwords will migrate across to Azure AD.
- If you can't import users' passwords, because they are perhaps hashed or stored in a location you can't access, Azure AD will create a random password for each new user account and ask the user to change his or her password when they first sign in.

Azure AD password policies will apply to any passwords imported into Azure AD B2C. New passwords and any password resets will require strong passwords that comply with these polices. To migrate accounts that have weaker passwords than those required by Azure AD, you can disable the strong password requirement by setting the **passwordPolicies** property to **DisableStrongPassword**.

Module 9: Auditing and reporting

Introduction

This module introduces you to auditing and reporting in Azure AD B2C that can be used for troubleshooting or reporting on activities within your app. Azure Application Insights will be used to collect audit and diagnostic information from the web app to monitor events.

At the end of this module, you will be able to:

- Create and integrate Application Insights with Azure AD B2C.
- View audit events in the Azure portal.
- Download audit logs using the reporting API.

And you will have a deeper understanding of the following concepts:

- How to monitor user journeys
- Security events
- Azure AD app permissions
- Utilizing the reporting API
- Integrating Application Insights with Azure B2C
- Downloading events from the Azure portal
- Downloading events using the reporting API

This module should take approximately 20 minutes to complete.

Setup Application Insights

First, you will need to create an app insights resource for the data to be sent from Azure AD B2C custom policies.

1. Log into the Azure portal as **B2CAdmin@AwesomeYourLastname.onmicrosoft.com**, where **AwesomeYourLastname** is the name of your Azure AD B2C tenant. Switch to the Azure tenant.

2. Set the following options in the **Application Insights** configuration blade:

   - Name**: AwesomeComputersAppInsights**.
   - Application Type: **ASP.Net web application**.
   - Subscription: Specify your current subscription.
   - Resource Group: Either create new resource group or select an existing resource group.
   - Location: Select your closest location.

3. Record the **Instrumentation Key**. You will need this when configuring the custom policies in the next procedure.

Use audit logs to export app, user flow, and user activity data

Follow the steps below to update the custom policies created in Modules 5 to 8.

1. Open the **TrustFrameWorkExtensions.xml** file using Visual Studio.

2. Under the **<TrustPolicyFramework>** element at the start of the file, add the following attributes shown in bold text.

```
<TrustFrameworkPolicy
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.microsoft.com/online/cpim/schemas/2013/06"
  PolicySchemaVersion="0.3.0.0"
  TenantId="AwesomeYourLastname.onmicrosoft.com"
  PolicyId="B2C_1A_TrustFrameworkExtensions"
  PublicPolicyUri="http://AwesomeYourLastname.onmicrosoft.com/
B2C_1A_TrustFrameworkExtensions"
  DeploymentMode="Development"
  UserJourneyRecorderEndpoint="urn:journeyrecorder:applicationinsights">
```

3. Add the following **<RelyingParty>** node to the end of the file, before the closing **</TrustFrameworkPolicy>** tag. Replace ***instrumentation-key*** with your AppInsights instrumentation key.

```
<RelyingParty>
  <UserJourneyBehaviors>
```

```
      <JourneyInsights
        TelemetryEngine="ApplicationInsights"
        InstrumentationKey="instrumentation-key"
        DeveloperMode="true"
        ClientEnabled="false"
        ServerEnabled="true"
        TelemetryVersion="1.0.0" />
    </UserJourneyBehaviors>
  </RelyingParty>
```

   Note: You must not set the **DeveloperMode** option to true in a production system.

4. Save the file.
5. Open the **SignUpOrSignIn.xml** file using Visual Studio.
6. In the **<RelyingParty>** node, remove the **<UserJourneyBehaviors>** element. This element is now redundant.
7. Save the file.
8. Return to the Azure portal, and switch to your Azure AD B2C tenant.
9. In the **Azure AD B2C** blade, select **Identity Experience Framework**.
10. Select **Upload Policy**, and upload the **TrustFrameWorkExtensions.xml** file, overwriting the existing policy.
11. Select **Upload Policy** again, and upload the **SignUpOrSignIn.xml** file, overwriting the existing policy.
12. Select the **B2C_1A_signup_signin** policy, and then select **Run now**. This will generate some data which will be recorded by Application Insights. Repeat this step several times, and perform tasks such as changing your password or updating your profile using the sample web app.

Note: There will be a delay of a few minutes while Application Insights gathers the data and you can view reports; they are not real-time.

Download data into an insights/analytics system

You can view the data sent from Azure AD B2C in the Application Insights resource directly. Follow the steps below to view the data.

1. In the Azure portal, switch to the Azure tenant (not the Azure AD B2C tenant) and open the Application Insights resource you have created.
2. In the **Overview** blade, select **Analytics**.

From here you can explore the metrics and trace information collected by Application Insights for the custom policy which was uploaded with the telemetry key in the previous step.

3. Select **+** next to the **Home Page** tab on the main page, this will open a new query page.
4. Examples of queries are listed below. Alternatively, you can select custom time ranges and select **Last 24 Hours**, then select **RUN**.

Query	Description
traces	See all of the logs generated by Azure AD B2C
traces | where timestamp > ago(1d)	See all of the logs generated by Azure AD B2C for the last day

5. When the results are returned, you can drill into them to examine the details. For example, if you expand a record, and then expand **customDimensions**, you can see which user journey triggered the event.
6. If you expand the **message** element, you can see the details of the steps performed by the user journey.
7. You can also download the results using the export option. This enables you to analyze the data using other tools, such as Excel.

Visit our online documentation to learn more about queries in log analytics.

Auditing admin and security events

You can view all auditing and security events either by using the Azure portal or by downloading audit activities using the Azure Reporting API. The Reporting API extends the Graph API with a series of endpoints that generate summary usage data for your Azure AD B2C tenant.

Follow this online tutorial to learn how to use Admin Audit Logs in the Azure portal as well as to configure the Azure Reporting API: Accessing Azure AD B2C audit Logs

Additional logging and monitoring capabilities in Azure AD B2C

While outside the scope of this training guide, you can use Azure Monitor to route Azure Active Directory B2C (Azure AD B2C) sign-in and auditing logs, or export them to different monitoring solutions. You can retain the logs for long-term use or integrate with third-party security information and event management (SIEM) tools to gain insights into your environment. Additionally, the Azure AD B2C Reports and Alerts GitHub repository has several useful resources, queries and Azure Monitor dashboards you can download and customize in your own environment.

Glossary

What is Azure AD B2C

Azure AD B2C is a customer identity management system that enables you to abstract the authentication and authorization functions from your application. Azure AD B2C enables you to programmatically construct user journeys including sign-up, sign-in, and profile management.

Azure AD B2C interacts with identity providers, customers, other systems (such as CRM systems) and a local directory to complete identity tasks.

The underlying platform that establishes multi-party trust and completes these steps is called the Identity Experience Framework (IEF). This framework and a user flow (also called a user journey or a Trust Framework user flow) explicitly defines the actors, the actions, the protocols, and the sequence of steps to complete. You can find out more about Azure AD B2C here.

When to use Azure AD B2C

Azure AD B2C is best used when you want to sign up and sign in users for web apps, iOS apps, Android apps, single page apps, or desktop apps. It can enable the user journeys discussed above and can interact with REST APIs to validate information with other systems, such as validating an address with the postal service, or validating a governmental identity with a government system.

If you are looking for identity services for enterprise accounts, you should investigate Azure AD and its capabilities around securing employee access to corporate resources. However, if you are looking for identity services for partners or suppliers to give them access to enterprise resources, you should also refer to our documentation on Azure AD External Identities that encompasses consumer as well as partner and supplier scenarios. This will help you understand the different capabilities of the Azure AD platform and the pricing tiers associated with the features you are interested in deploying.

Accounts, tenants, and subscriptions

An **Azure account** is created when you sign up for Azure. Creating your Azure account creates an **Azure Tenant**, which is accessed by logging in to https://portal.azure.com with the credentials that you used to create your Azure account. This is the **Azure portal**. Inside the portal, you create and manage different types of user accounts and other resources.

An **Azure subscription** is a billing agreement you create in conjunction with your Azure account. Azure subscriptions are usually pay-as-you-go, and you only incur charges for the services that you use. To create an Azure AD B2C tenant, your Azure account must have a subscription.

An **Azure AD B2C tenant** is a resource that you create within your Azure tenant.

Unique App *IDs in Azure AD*

Each application registered in an Azure AD or Azure AD B2C tenant has a unique app ID which is utilized as part of the trust mechanisms built into the Azure AD authentication libraries. Applications contain unique IDs and secrets, which are used in apps to ensure that they are authorized to request tokens from Azure AD tenants.

User flows

Built-in user flows

Built-in user flows provide the user experience and are associated with IDPs and apps for various user journeys. These include Sign up, Sign In, Password Resets and profile editing user flows. User flows can be applied to multiple IDPs and apps that are registered in Azure AD B2C which makes the development effort faster, reducing the time you need to spend on coding the user journeys.

User flows are important to ensure that the right user attributes are captured as part of the sign up or sign in process and are flexible enough so that you can add custom attributes too. User flows also allow you to control the look and feel of the user journeys and also enforce Multi-Factor Authentication.

How user flows work

User flows you define are associated with one or more IDPsappuser flows will determine the behavior when a user authenticates to the IDP. When the token from the IDP is validated by Azure AD B2C, ensuring the signature is trusted, Azure AD B2C then uses a user flow specified in the Azure AD B2C tenant for the specific IDP and app.

How apps use user flows

Applications receive user id_tokens to make authorization decisions. Users attributes, which may be collected during a sign-up or sign-in user flow, are used as claims that are present in the id_token. Applications can then utilize the claims in the id_token, to understand the user in more detail e.g. a claim can be used to expose different UI elements based on a role associated with a user's claims, which are taken from attributes.

Using user flows for SSO across apps

Multiple apps within a tenant may use the same user flows, which can result in Single Sign-on (SSO) to all the apps to which the user has access. By default, the configuration is set at the tenant level, which means that multiple apps can share the same user session. This means that once the user is authenticated to an app, the user can access multiple apps registered in the same Azure AD B2C tenant, without having to follow a user journey or be re-directed for authentication credentials.

For further information, see Azure AD B2C: Token, session and single sign-on configuration.

Attributes and claims

Attributes are pieces of information which are tied to a user profile. You may want to collect specific attributes as part of a sign-up or sign-in user flow to ensure that you have collected the necessary information for the app you are developing. Users will be presented with a form, for each user journey, to ensure that the attributes are collected as part of the initial sign up or sign in process.

Attributes of a user can be added to the user's claims, which is essentially updating the user's id_token through the user flows, when they are authenticated by Azure AD B2C which be utilized across different apps.

Identity Providers

An Identity Provider's primary purpose is to store information for user or system principals that has its own mechanisms to provide authentication to a known subject. A subject can be referred to as a user identity. A typical example of an identity provider is Azure AD; it encompasses user identities, groups and various pieces of information for each object it stores. For example, Azure AD provides authentication services to Office 365 and each Office 365 app relies on Azure AD to provide a user identity token, using common authentication protocols. Authentication occurs at the IDP and authorization occurs within an app.

Authentication protocols supported by Azure AD B2C

A token that is signed by a trusted IDP is provided to Azure AD B2C, utilizing one of the following authentication protocols which are supported by Azure AD B2C.

- OAuth 2.0
- OpenID Connect (OIDC)
- SAML 2.0

A user ID token or app token is provided by Azure AD B2C to an app, which provides an authentication context for the user or app respectively. Applications can utilize claims, based on the identity, present within the token e.g. to make decisions on what the users role is within the app.

Azure AD V1 vs V2 endpoints

Azure AD provides a v1 and v2 endpoint which utilizes OAuth 2.0 to issue JSON Web Tokens. It is important to understand the differences between both endpoints and the limitations. The v1 endpoints are commonly access using the Active Directory Authentication Library for work or school accounts. The v2 endpoint is utilized using the Microsoft Authentication Library and provides integration with Azure AD B2C and thus provides access to any IDP supported by Azure AD B2C. For further information, see What's different about the v2.0 endpoint?

Identity Experience Framework

The Identity Experience Framework (IEF) is a fully configurable, policy driven, cloud based Azure service that coordinates trust between various entities such as claim providers. It implements standard protocol formats such as OIDC, OAuth, SAML as well as non-standard formats, such as claims based on proprietary REST APIs. The IEF provides a user-friendly step-by-step interface that simplifies many of the tasks involved when implementing identity policies

Using the IEF, you provide the following essential information.

- The legal, security, privacy and data protection policies to which participants must conform.
- The contacts and process required to become an accredited user.
- The trusted identity/claims providers to use.
- The trusted relying parties and their requirements.
- The run time rules to enforce for exchanging identity information.

For built-in user flows, you simply use the various wizards provided by the IEF blade in the Azure portal. For more complex, custom requirements, you specify the details by using custom policy files that you upload to IEF.

Using the IEF also minimizes the complexity of configuring identity federation. Federated identity provides the end user identity assurance. By delegating Identity to third parties, a single user identity can be used with multiple relying parties. Identity assurance requires that the IDPs and AtPs must adhere to specific security, privacy, and operational policies and practices. RPs must build the trust relationships between IDPs and AtPs they choose to work with. The IEF assists in this task, essentially reducing the process to two steps; establishing a trust relationship, and performing a single metadata exchange.