**Azure AD B2C custom policies**

The underlying policy within Identity Experience Framework (IEF).

Azure AD B2C policy is comprised of and how a user experience is generated.

With Azure AD B2C custom policies, you can do more than these sign-ups or sign-in, password reset and edit profile policies/journeys. You can create any policy you would like. Such as:

* Link and unlink a social account to local or vice versa.
* Allow a user to change their sign-in email address, or MFA phone number.
* Invite a user to Azure AD B2C.
* ROPC based flows - A non-interactive flow, commonly used with **mobile applications**.
* Magic link - Where users can redeem an account with a link you send them or launch any custom user journey.
* Sign in and migrate accounts - Allows you to migrate user accounts from any legacy identity provider to Azure AD B2C just in time.

Azure AD B2C is backed by a global, highly available, and secure **cloud identity service** and **directory service**. During policy execution, such as sign-up or sign-in, Azure AD B2C policy exchanges claim with a variety of systems, both having the ability to send and receive claims in either direction

Sign-in with local account flow

1. The user starts their journey by opening their application and clicking on sign-in button. The user is redirected to Azure AD B2C to complete the sign-up or sign-in flow.
2. In this example, the user chooses to sign-in with a local account (an account that is fully managed by Azure AD B2C). The user provides their email and password. Internally, the email address is cross referenced against the **signInNames** user attribute.
3. The user clicks “Sign In”. Azure AD B2C validates the credentials provided by the user. If the user provided the correct credentials, Azure AD B2C reads various user object properties from the directory, such as display name, first name, last name and more.
4. Azure AD B2C finally redirects the user back to the application with a token that was issued.

Azure AD B2C exchanges claims with other systems.

**First step**, the application sends information to execute the authorization request, such as the Azure AD B2C policy Id.

**Second step**, the user provided their credentials, this translates to the “username” and “password” claims being provided by the user. This is a claims exchange, and this claims exchange is **called “self-asserted”.**

**Third step**, Azure AD B2C reads the user object from the directory. Sending the username claim and getting back the **requested claims about the user**.

**Fourth step**, Azure AD B2C builds the **id token and sends** it back to the application.

Sign-in with social account flow

In a more complex scenario, when a user chooses to sign-in with a **Facebook account**. The user is taken to the Facebook sign-in page. After the user completes their login to Facebook a token is returned to Azure AD B2C. Azure AD B2C validates the Facebook token and reads the claims. Then reads the social account profile from the directory.

But, in this case Azure AD B2C also invokes a **REST API, to further integrate with a marketing database, sending and receiving claims from the REST API.**

B2C issues an id token back to the application.

How Azure AD B2C exchanges claims with other systems.

The relying party application, also known as your Application, sends an authentication request to B2C. B2C may send or receive claims from the end-user, social identity provider, multi-factor authentication provider, REST API and with the Directory service (Azure AD).

During the policy execution, Azure AD B2C stores the claims in a temporary memory called a “**claims bag**”. This is stored in real-time to be utilized for any further steps in the policy.

**Claim definition: -**

A claim provides temporary storage of data during an Azure AD B2C policy execution.

It can store information about the user, such as first name, last name, or any other claim obtained from the user or other systems (claims exchanges).

When the policy runs, Azure AD B2C sends and receives claims to and from internal and external parties and then sends a sub-set of these claims to your **relying party application** as part of the token. Claims are used in these ways:

* A claim is saved, read, or updated against the directory user object.
* A claim is received from an external identity provider.
* Claims are sent or received using a **custom REST API service.**
* Data is collected as claims from the **user during the sign-up or edit profile flows**.

A claim should have at least following

* **Id** - an identifier that’s used for the claim type. Other elements can use this identifier in the policy.
* **Display name** - The title that’s displayed to users on various screens. The value can be localized.
* **Data type** - The type of the claim. Data types such as Boolean, date, date and time, int, long, string, string collection, and more.

If a claim is used as part of a page a user can interact with (self-asserted claims exchange), the claim may contain more information, such as:

* **User input type** - The type of input control that should be available to the user when manually entering the claim data for the claim type, for example “text box”. See the user input types defined later on this page.
* **Mask** - An optional string of masking characters that can be applied when displaying the claim. For example, the phone number 324-232-4343 can be masked as XXX-XXX-4343.
* **User help text** - a description of the claim type that can be helpful for users to understand its purpose. The value can be localized.
* **Restriction** - The input validations for this claim, such as a regular expression (Regex) or a list of acceptable values. The value can be localized.
* **Predicate validation reference** - A reference to a predicate validations input element, that allows you to perform a validation process to ensure that only properly formed data is entered. For more information, see Predicates.

Claims mapping and default value

When Azure AD B2C exchanges claims, the name of the claim used by the partner may differ from the one configured in your policy. For example, Azure AD B2C refers to the first name with **givenName** while Facebook uses **first\_name**. Azure AD B2C supports mapping your partner claim name to the one configured in your Azure AD B2C policy.

In the table below, we can see how various entities give different claim names to the same property.

| **B2C internal name** | **Facebook** | **Google** | **Twitter** |
| --- | --- | --- | --- |
| issuerUserId | id | id | user\_id |
| givenName | first\_name | given\_name | NA |
| surname | last\_name | family\_name | NA |
| displayName | name | name | screen\_name |
| email | - | - | - |

You can use claims mappings to change the name of the claim while sending and receiving data from any claims exchange component. This allows you to be able to interface your Azure AD B2C claim IDs with those of external systems.

You may want to set a default value to a claim, if the partner doesn’t return the claim, or to override the value regardless of the returned value, or to handle null return values.

An example of this is setting the claim called identityProvider to facebook.com and the authenticationSource to socialIdpAuthentication. Subsequent steps in the policy can then use these claims to check whether user sign-in with a local account, or social and take a conditional based action.

**Claims transformations.**

Claims transformations are predefined functions. When exchanging claims with a partner, you may need to convert a given claim **to another one or determine whether one claim is equal to another**. Azure AD B2C has a predefined set of claims transforms which allows manipulating the claims inside the claims bag.

Here are some examples of claims transformations:

* Change a string case to upper or lower case.
* Set a predefined value to a claim.
* Compare two claims and return a Boolean result
* Create a random string.
* Format a string by the way of concatenation.
* Add a claim to a string collection.
* Get current data and time.
* Null a claim!
* Check if a claim has value and return a Boolean result

**Technical profile**

The following will dive into the underlying components of Azure AD B2C, these components are referred to as Technical Profiles which drive the functionality of your Azure AD B2C policy. All interactions with partners to perform claims exchanges are completed via technical profiles.

You can think of **Technical Profiles like functions**. They can:

* Send claims to the partner - “**input claims**”
* Execute a procedure - E.g. Render a page to collect information from the user
* Return claims to Azure AD B2C – “**output claims**”

Technical profiles provide a framework with a built-in mechanism to communicate with known Azure AD B2C components, REST APIs and **Identity Providers via open standard protocols.**

There are various types of technical profile:

* **Identity Provider** - Denoted by the Protocol of the Identity Provider (OAuth1, OAuth2, OpenID Connect, SAML and Ws-Fed)
* **REST API** - Allows interfacing with an external API
* **Self-Asserted** - Allows presenting a page to the user
* **Azure MFA** - Allows presenting the Azure MFA screen to a user.
* **Azure Active Directory** - Allows Read/Write operations against the directory
* **Application Insights** - Allows sending custom events to an App Insights instance.
* **Token Issuer** - Allows issuing a token after authentication completes.

All types of technical profiles share the same concept as per mentioned.

Technical profile execution

1. **Input claims transformation** - Before a claim is provided to this Technical Profile, the claim can be manipulated via a claim transform. For example, **concatenate two claims and provide it as an input claim to this technical profile**.
2. **Input claims** - Claims that are provided as inputs to the type of Technical Profile being executed. Since each type of Technical Profile provides its own built-in functionality, these input claims can act differently.
   * **Self Asserted** - Pre-populate fields that are displayed on the screen.
   * **REST API** - Act as claims to be sent as a JSON key value pair to the API endpoint.
   * **Identity Provider (OpenId)** - Additional query parameters as part of the authentication request.
   * **Azure MFA** - Pre-populate the phone number for an prior enrolled user.
   * **Azure Active Directory** - To lookup the user in the directory.

**Technical profile execution** - As each Technical Profile has a type, the execution is dependent on that.

* + **Self Asserted** - Display a page for the user to interact with. The user can provide information or edit information about their profile.
  + **REST API** - Ability to call a REST API endpoint to exchange claims.
  + **Identity Provider** - Ability to redirect the user to an external Identity Provider.
  + **Azure MFA** - Provides the user a page to perform Azure MFA.
  + **Azure Active Directory** - Allows Azure AD B2C policy to read or write to the directory.

**Validation technical profile** - When a user provides data via a **Self-Asserted technical profile**, this information may need to be validated. A common example is to have the user provide a loyalty number, which is then validated at an external system to determine if it is valid.

A Self-Asserted technical profile can therefore call another Technical Profile, or multiple consecutively, which provide the means to **validate the information**.

This will either allow the user to continue or prevent any further execution **if an error is thrown**. In the case where the user can continue, the validation technical profile can output claims and add them to the claims bag for any subsequent validation technical profiles to run as part of this **Self-Asserted technical profile**.

Claims that a Validation technical profile shared (output) with the Self-Asserted technical profile are not automatically available for subsequent steps in the Azure AD B2C policy to use unless those claims are output within the Self-Asserted technical profile itself.

In the example of validating a loyalty number, the Validation technical profile **can reference a REST API technical profile to have the loyalty number validated** at an external system.

1. **Output claims** - Claims to return **to the claims bag**. Depending on the technical profile type, the output claims behave differently, however all types of Technical Profile will return the output claim into the claims bag.
   * **Self Asserted** - Will display the fields to the user, unless satisfied by a Validation technical profile.
   * **REST API** - The expected JSON key value pairs from the APIs response.
   * **Identity Provider** - The expected JSON key value pairs from the JWT received.
   * **Azure MFA** - The ability to confirm the SMS/Phone Number completed MFA.
   * **Azure Active Directory** - Issues claims after reading or writing to a user into the claims bag.

You can use these claims as input claims into subsequent orchestration steps, create conditional logic around the journey execution, or manipulate them via output claims transformations as part of this technical profile.

1. **Output claims transformations** - A technical profile may contain a list of output claims transformations to be executed to manipulate the claims before sending them to the claims bag.

As an example, take the first name and last name and use an output claims transformation to concatenate these values into a new single claim value. Subsequent steps will then be able to have access to the concatenated string.

1. **Session management** - A technical profile can contain a reference to a session management technical profile that is responsible to manage whether the user will skip a step in the policy when experiencing single sign on

**Validation technical profile**

As mentioned above, a Self-Asserted technical profile may define one or more validation technical profiles to be used for validating some or all its output claims.

A validation technical profile is an ordinary technical profile from any type(protocol), such as Azure Active Directory or a REST API. The Validation technical profile returns output claims or returns an HTTP error message.

The most common use of a Validation technical profile is to validate the username and password the user provides during **Sign-In**.

Another example during **Sign-up**, before allowing a user to create their account, Azure AD B2C must check if the account already exists.

Therefore, once the user enters their information through a Self-Asserted technical profile, it must validate the email does not already exist in the directory.

To achieve this, the Self-Asserted technical profile calls a Validation Technical profile which searches the directory for the users email. An Azure Active Directory technical profile type is used, which is configured to throw an error if the user exists, or otherwise continue silently.

You can further augment this process to call another Validation technical profile of type REST API. This could validate any other information the user provided during sign up, for example their ID Number.

**Integration with line of business applications**

You can use REST API technical profile to:

* **Validate user input data** - This action prevents invalid data from being persisted into Azure AD B2C claims bag. If the value from the user is not valid, your **RESTful service should return an error message that instructs the user to provide a valid entry**. For example, you can verify that the email address provided by the user exists in your CRM platform.
* **Overwrite input claims** - For example, if a user enters the first name in all lowercase or all uppercase letters, you can format the name with only the first letter capitalized.
* **Enrich user data by further integrating with corporate line-of-business services** - Your RESTful service can receive the user’s email address, query the CRM platform, and return the user’s loyalty number to Azure AD B2C. The return claims can be stored in the user’s Azure AD account, evaluated in the next Orchestration Steps, included in the id token or a combination.
* **Run custom business logic** - You can send push notifications, update corporate databases, run a user migration process, manage permissions, audit databases, and perform other actions.

**User Journey**

A user journey defines the overall user experience that a Azure AD B2C policy will provide when the **Identity Experience Framework processes the authentication request**. The user is taken through a series of steps to retrieve the claims that are to be presented to the relying party.

Each user journey is a consecutive sequence of **orchestration steps**. Each step has a defined type that specifies the overall type of step to be executed, and a reference to the technical profile which fulfills that execution.

For example, the type **“IdP selection**” - This lets the user select with whom they would like to login with from a list of identity providers (sign-in with local account, Facebook, Microsoft account, Twitter, Google, ect.).

If a user selects to sign-in with a ***local account***, there is an orchestration step in which specifies the technical profile that collects and validates the user credentials and returns the user object Id to the claims bag.

If a user selects to sign-in with a ***social account***, the next orchestration step takes the user to the selected identity provider, such as Facebook.

In the majority of cases, the last orchestration step type issues the token and redirects the user back to the application.

In the following example, we have 7 steps. But a user may not run through all of them. Because a step may contain **preconditions** instructing the policy to skip to the next orchestration step, based on claim comparison or claim existence. This is very similar to an **If** and **Else** statement.

**Local and social accounts user journey**

The sign-up or sign-in user journey contains the following orchestration steps:

**Initiates an identity provider selection**: Shows the identity provider options to user. If user provides credentials on this screen, then they have chosen to login with a local account. If the user clicks on one of the social identity providers, B2C will move directly to the next step to handle this.

![Sign-up or sign-in]/assets/images/docs/uj-local-and-social-account-susi.png)

If user had signed-in locally (in the previous orchestration step/s), skip this step, since in this step we want to handle external account logins. In this step sign in with the social provider selected earlier by executing its respective technical profile, or register new user with a local account.

If local account had been chosen earlier (user had signed-in or signed-up with a local account), skip this step, otherwise lookup the social account by the social account unique id obtained from the social identity provider and read its attributes from the Azure AD B2C directory.

In this step, B2C attempts to find a representation of the social account in the directory. If the account is not found, B2C will not raise an error message, since this means the user has signed-in with Facebook for the first time.

If a social account representation of a user was found skip this step. Otherwise since this user logging in for the first time with their social account, present the user a registration form. Once the user submits this page create the account in the directory. The Self-Asserted technical profile invokes a validation technical profile that actually writes the account into the directory.

If this is a social account login, skip this step, else read all desired attributes from the user signed in with a local account from the user directory.

Create a token with the claims of the user collected during the user journey to send back to app within the id token.

**Preconditions** can be used similar to an if/else statement. They can be used to conditionally execute an orchestration step (technical profile), or validation technical profile

The following is an example which applies a precondition to an orchestration step within the user journey. This orchestration step will execute a technical profile, therefore the precondition governs whether the technical profile will or will not be run.

<OrchestrationStep Order="5" Type="ClaimsExchange">

<Preconditions>

<Precondition Type="ClaimsExist" ExecuteActionsIf="true">

<Value>objectId</Value>

<Action>SkipThisOrchestrationStep</Action>

</Precondition>

</Preconditions>

<ClaimsExchanges>

<ClaimsExchange Id="AADUserWrite" TechnicalProfileReferenceId="AAD-UserWriteUsingAlternativeSecurityId" />

</ClaimsExchanges>

Breaking this down:

1. This states that this precondition will execute the desired action if the claim exists in the Azure AD B2C claim bag.

<Precondition Type="ClaimsExist" ExecuteActionsIf="true">

1. This states that the claim which is being evaluated to being within the claim bag is called objectId.

<Value>objectId</Value>

1. This is the action which will be executed once the condition is met.

<Action>SkipThisOrchestrationStep</Action>

If the condition is not met, then the orchestration step is not skipped, and the technical profile is executed.

A precondition can also base its result on the value of a claim.

<Precondition Type="ClaimEquals" ExecuteActionsIf="true">

<Value>name</Value>

<Value>John Smith</Value>

<Action>SkipThisOrchestrationStep</Action>

</Precondition>

For preconditions based **Boolean** claims, the Value must be True or False, this is case sensitive.

For more information on configuring a precondition, see this [reference](https://docs.microsoft.com/en-us/azure/active-directory-b2c/userjourneys#preconditions) for preconditions to control the execution of orchestration steps, and this [reference](https://docs.microsoft.com/en-us/azure/active-directory-b2c/validation-technical-profile#validationtechnicalprofiles) to control the execution of validation technical profiles.

**Custom policies** are XML configuration files that define the behavior and user experience of an Azure AD B2C policy, with the use of:

* Claim definitions.
* Claims transformations.
* Technical profiles
* User journey/s
* Replying Party

While built-in policies (also referred to as User Flows) are predefined in the Azure AD B2C portal for the most common authentication journeys, custom policies can be fully edited by an identity developer to complete many different tasks wrapped in an authentication journey.

**Policy execution**

Your **relying party application** calls a **relying party policy**. The relying party policy specifies which **user journey** to execute.

A user journey defines the business logic of what a user will experience. Each user journey is a set of orchestration steps that performs a series of claims exchanges with various partners, in a consecutive order adding to the claims bag as each step completes.

Each **orchestration step** calls a **Technical Profile**. Technical profiles provide a framework with a built-in mechanism to communicate with known Azure AD B2C components, **REST APIs and Identity Providers via open standard protocols.**

An orchestration step may have one or more **preconditions** to determine if the orchestration step should execute during this flow or skipped.

**Policy file structure**

An Azure AD B2C policy follows an **inheritance model**, whereby configurations being made in a **chained file can take dependencies on configuration elements defined earlier in the chain of files.** This allows you to scale your design without impacting other work that you have created and add additional policies to leverage existing work.

Let’s look at a relying party application. It calls the Relying Party policy file to execute a specific task to initiate **the sign-in flow**.

The Identity Experience Framework in Azure AD B2C stacks all of the elements from the Base file, completing the chain of files to the Relying Party policy file to assemble the current policy in effect. Elements of the same type and name in further in the chain of files override those elements those already configured in file earlier in the chain.

As described, your application will always **reference the relying party policy id as part of the authentication request**.

The child policy at any level can inherit from the parent policy and extend it by adding new elements or overriding some or all of the elements in the parent policies.

**There is no limit on the number of levels in this chained structure**.

* A relying party application, such as a web, mobile, or desktop application, calls the relying party (RP) policy.
* The RP policy configures the list of claims the relying party application receives as part of the token that is issued.
* Multiple applications can use the same policy. All applications will receive the same token and claims in this case.
* A single application can use multiple policies and each application could receive different claims in the returned token.
* A relying party policy, points to the user journey to be executed.
* A tenant can have multiple polices (**built-in and custom**). You can combine a built-in policy with custom policies. For example the sign-in journey can be a custom policy, whilst the profile edit journey can be a built-in policy.

**Relying party policy endpoints**

For each relying party policy created, Azure AD B2C provides you the endpoints that your application may use:

* **Well-known configuration** – which provide the metadata for Azure AD B2C.
  + https://tenant-name.b2clogin.com/tenant-name.onmicrosoft.com/policy-id/oauth2/v2.0/.well-known/openid-configuration
* **Authorize** endpoint to make authentication requests
  + https://tenant-name.b2clogin.com/tenant-name.onmicrosoft.com/policy-id/oauth2/v2.0/authorize
* **Token** endpoint to redeem authorization codes for access tokens
  + https://tenant-name.b2clogin.com/tenant-name.onmicrosoft.com/policy-id/oauth2/v2.0/token
* **Logout** to logout the user from Azure AD B2C.
  + https://tenant-name.b2clogin.com/tenant-name.onmicrosoft.com/policy-id/oauth2/v2.0/logout

**Customize your policy**

A **content definition** allows linking a page type referenced in a Self-Asserted technical profile and the HTML template, localization and page contract (see later).

The self-asserted technical profile points to a content definition identifier. The content definition contains:

* The URL of the HTML template
* Pointer to the localizations
* Defines the [page contract](https://docs.microsoft.com/en-us/azure/active-directory-b2c/page-layout)

*<ContentDefinition Id="api.error">*

*<DataUri>urn:com:microsoft:aad:b2c:elements:contract:globalexception:1.2.0</DataUri>*

*</ContentDefinition>*

*<ContentDefinition Id="api.idpselections">*

*<DataUri>urn:com:microsoft:aad:b2c:elements:contract:providerselection:1.2.0</DataUri>*

*</ContentDefinition>*

*<ContentDefinition Id="api.idpselections.signup">*

*<DataUri>urn:com:microsoft:aad:b2c:elements:contract:providerselection:1.2.0</DataUri>*

*</ContentDefinition>*

*<ContentDefinition Id="api.signuporsignin">*

*<DataUri>urn:com:microsoft:aad:b2c:elements:contract:unifiedssp:2.1.0</DataUri>*

*</ContentDefinition>*

*<ContentDefinition Id="api.selfasserted">*

*<DataUri>urn:com:microsoft:aad:b2c:elements:contract:selfasserted:2.1.0</DataUri>*

*</ContentDefinition>*

*<ContentDefinition Id="api.selfasserted.profileupdate">*

*<DataUri>urn:com:microsoft:aad:b2c:elements:contract:selfasserted:2.1.0</DataUri>*

*</ContentDefinition>*

*<ContentDefinition Id="api.localaccountsignup">*

*<DataUri>urn:com:microsoft:aad:b2c:elements:contract:selfasserted:2.1.0</DataUri>*

*</ContentDefinition>*

*<ContentDefinition Id="api.localaccountpasswordreset">*

*<DataUri>urn:com:microsoft:aad:b2c:elements:contract:selfasserted:2.1.0</DataUri>*

Note: You can add the localization to the content definition OR directly to the self-asserted technical profile

[Concepts - Azure AD B2C (azure-ad-b2c.github.io)](https://azure-ad-b2c.github.io/azureadb2ccommunity.io/docs/custom-policy-concepts/)

[Best practices - Azure AD B2C (azure-ad-b2c.github.io)](https://azure-ad-b2c.github.io/azureadb2ccommunity.io/docs/custom-policy-best-practices/)