<https://learn.microsoft.com/en-us/azure/active-directory-b2c/custom-policy-overview>

<https://www.ravenswoodtechnology.com/introduction-to-azure-ad-b2c-custom-policies/>

Samples: https://github.com/azure-ad-b2c/samples

# INTRODUCTION TO AZURE AD B2C CUSTOM POLICIES

* [**MARCH 9, 2022**](https://www.ravenswoodtechnology.com/2022/03/09/)
* [**ACTIVE DIRECTORY**](https://www.ravenswoodtechnology.com/category/active-directory/)**,**[**IDENTITY MANAGEMENT**](https://www.ravenswoodtechnology.com/category/identity-management/)
* [**ANDY SCHNEIDER**](https://www.ravenswoodtechnology.com/team/andy-schneider/)

Azure Active Directory (AAD) Business-to-Customer (B2C) provides identity as a service for custom applications. It’s built on the same underlying technology as AAD, with additional critical features required for customer-facing applications, including standard identity management operations such as signing up for an application, signing in, resetting a password, and updating basic user profile information.

B2C enables application developers to manage local logons and use popular identity providers such as Microsoft, Facebook, and Google. B2C allows the developer to have as much control as necessary over the end user’s experience when they sign up or sign in to their application. These experiences are defined by user flows or custom policies.

User flows are templates provided by Microsoft that you can configure in the Azure portal to implement the identity operations mentioned above. Custom policies provide much deeper customization of these operations.

## User Flows or Custom Policies

In general, if you need basic B2C functionality along with the need to customize the UI, you can safely go with standard user flows. If you know your application’s integration with B2C will be very complex and will require a lot of customization, you’ll want to use custom policies. Microsoft is continually working to provide the most requested features in user flows that were previously only available in custom polices.

User flows have become quite powerful and offer enough flexibility for many applications. I would caution developers to not immediately decide to use custom policies simply because they give you more control and influence over the user experience. You can fully customize the UI of user flows with your company’s branding. User flows also allow you to collect and store custom attributes for users. You can call REST APIs to validate information that a user provided when signing up. User flows also enable you to let users log on with other identity providers such as Facebook and Google.

Custom policies are XML configuration files built on the [Identity Experience Framework](https://docs.microsoft.com/en-us/azure/active-directory-b2c/trustframeworkpolicy) (IEF). Whereas user flow templates are predefined in the Azure AD B2C portal for the most common identity management operations, custom policies can be created and updated by an identity developer to complete a wide variety of tasks. Custom policies allow you to federate with any identity provider that supports OpenID Connect (OIDC) or Security Assertion Markup Language (SAML). They can support multi-factor authentication (MFA) with both Microsoft and third-party providers. Custom policies also allow you to call REST APIs anywhere in the user experience and they support the ability to leverage OAuth bearer tokens to authenticate a REST API, whereas user flows only support basic and certificate authentication for REST APIs.

## Organizing Custom Policies

After choosing to use custom policies, it’s important to understand the different types of policies and the overall structure of how they relate to one another.

### Custom Policies Hierarchy

Custom policies are composed in a hierarchical system. Child policies inherit from parent policies. B2C elements in child policies override existing elements in the base or are added as new elements. This is like the idea of inheritance in object-oriented programming. You inherit from a base class to leverage all its features and then override the methods that are unique to your requirements.

It’s highly recommended to use Microsoft’s [B2C Starter Packs](https://github.com/Azure-Samples/active-directory-b2c-custom-policy-starterpack) on GitHub as a starting point for your solution. The default hierarchy in the local accounts starter pack can be seen in the diagram below:

A diagram of a company

Description automatically generated

The base of the hierarchy is the B2C\_1A\_TrustFrameworkBase. The B2C\_1A\_TrustFrameworkExtensions policy inherits from the base. The SignUpSignIn, ProfileEdit, and PasswordReset relying party policies inherit from the extensions policy.

The hierarchy is specified in a parent policy XML tag. The following is an example of the BasePolicy element in my TrustFrameworkExtensions policy:

<**BasePolicy**>

<**TenantId**>andylandexternal.onmicrosoft.com</**TenantId**>

<**PolicyId**>B2C\_1A\_TrustFrameworkBase</**PolicyId**>

</**BasePolicy**>Code language: HTML, XML (xml)

Using the BasePolicy element in policies, it’s possible to chain together as many policies as you like.

### Base Policy

The base policy provided by Microsoft in the starter packs contain the basic elements required to get up and running with B2C, including:

* **Claim Type Definitions:**Claims provide temporary storage of data when a B2C policy is executed. They are used as variables in custom policies.
* **Claim Transformations:**A claim transformation can make a change to a claim value, such as changing a string to all uppercase.
* **Claims Providers:**A claims provider is a collection of technical profiles.
* **Technical Profiles:**A technical profile provides a framework with built-in mechanisms to interface with different systems. A technical profile can be used to read and write to the Azure AD B2C directory. Another technical profile can be used to make calls to a REST API.
* **Content Definitions:**A content definition is used to specify the URLs of HTML templates that are used in a user journey. You can use content definitions to customize the UI of your user journeys.
* **User Journeys:**User journeys specify the steps that a user goes through when a custom policy is called.

The base policy should rarely, if ever, be modified. If this policy is modified, comments should be added so it is clear why the changes were made. Microsoft releases new base policies on a regular basis; fewer customizations made to the base will make it easier to swap out the base and be able to take advantage of new features.

### Extensions Policy

The extensions policy is where an identity developer should make customizations and override the base policy. The extensions policy inherits from the base policy. The extensions policy in the starter pack has no B2C elements. It only has placeholders where you can add your customizations and overrides to the elements in the base policy. Depending on the complexity and number of customizations being made, multiple extensions policies can be used.

### Relying Party Policies

An application interfaces with B2C by calling a relying party policy’s endpoint. Examples of relying party policies from the starter packs are SignUpSignIn, ProfileEdit, and PasswordReset. An application would redirect to the SignUpSignIn policy when a user chooses to sign up for an account. An example endpoint for sign up and sign in is:

<https://andylandexternal.b2clogin.com/andylandexternal.onmicrosoft.com/oauth2/v2.0/authorize?p=B2C_1A_SIGNUP_SIGNIN>

Relying party policies are usually quite small. They simply call a user journey and specify what claims should be provided in the JSON Web Token (JWT) that gets sent back to the calling application. These relying party policies are configured to inherit from the extensions policy mentioned above.

Here is an example of a SignUpSignIn relying party policy.

<**RelyingParty**>

<**DefaultUserJourney** ReferenceId="SignUpOrSignIn" />

<**TechnicalProfile** Id="PolicyProfile">

<**DisplayName**>PolicyProfile</**DisplayName**>

<**Protocol** Name="OpenIdConnect" />

<**OutputClaims**>

<**OutputClaim** ClaimTypeReferenceId="displayName" />

<**OutputClaim** ClaimTypeReferenceId="givenName" />

<**OutputClaim** ClaimTypeReferenceId="surname" />

<**OutputClaim** ClaimTypeReferenceId="email" />

<**OutputClaim** ClaimTypeReferenceId="objectId" PartnerClaimType="sub"/>

<**OutputClaim** ClaimTypeReferenceId="tenantId" AlwaysUseDefaultValue="true" DefaultValue="{Policy:TenantObjectId}" />

</**OutputClaims**>

<**SubjectNamingInfo** ClaimType="sub" />

</**TechnicalProfile**>

</**RelyingParty**>

Code language: HTML, XML (xml)

## Expanding the Extensions Policy

The extensions policy is where most customizations will be made. These customizations can be leveraged by all relying party policies to provide consistency. Customizations enable developers to adhere to the Don’t Repeat Yourself (DRY) programming principle. Customizations can also be made at the relying party policy level, but they will only affect that policy.

If a solution includes a lot of customizations and the extensions policy is growing in unmanageable complexity, you’ll want to expand the extensions policy into two or more policies. Multiple types of elements can be added to the extensions policy. As you build out your custom extensions, it’s important to note which elements depend on which. For example, if you have a technical profile that uses a custom claim, that custom claim needs to be defined in a parent policy.

Below is an example of a profile hierarchy with multiple extensions policies. The extensions policies are named according to what types of elements they contain.

A diagram of a company

Description automatically generated

## Adding and Overriding Elements

The following section assumes the reader is familiar with the basic elements and structure of custom policies. Microsoft has a [good](https://docs.microsoft.com/en-us/azure/active-directory-b2c/custom-policy-overview)[overview](https://docs.microsoft.com/en-us/azure/active-directory-b2c/custom-policy-overview)of the different elements that can be used in custom policies.

### Custom Claims

Claims are the most fundamental element in a B2C policy. Using the example hierarchy above, custom claims should be defined in the ClaimsProviderExtension policy. A basic claim can be created using the ClaimType element.

<**ClaimType** Id="shoeSize">

<**DisplayName**>Shoe Size</**DisplayName**>

<**DataType**>Int</**DataType**>

</**ClaimType**>Code language: HTML, XML (xml)

Along with the claims, there may be predicates and input validations for something like a password claim. It’s reasonable to put these elements in the same file as the claims they are being used by. The point of using multiple extension files is so you’ll know where to add or modify different types of elements.

### Claims Providers and Technical Profiles

A very common task when using custom policies is overriding existing technical profiles. One such example is to be able to read more user attributes from B2C than what you get by default. This can be accomplished by adding more output claims to the AAD-UserReadUsingObjectID technical profile. It’s very tempting to simply edit this technical profile in the base policy; however, it’s best to make these types of modifications in the extensions policy, specifically in the ClaimsProviderExtensionPolicy in the hierarchy above. It isn’t necessary to copy the entire technical profile to your extensions policy. You only need to specify the TechnicalProfile ID, and then you can add your customizations to it. To add a new output claim, such as city, to this technical profile, you would add a snippet like the following to the ClaimsProviderExtensionPolicy.

<**ClaimsProvider**>

<**DisplayName**>Azure Active Directory</**DisplayName**>

<**TechnicalProfiles**>

<**TechnicalProfile** Id="AAD-UserReadUsingObjectId">

<**OutputClaims**>

<**OutputClaim** ClaimTypeReferenceId="city" />

</**OutputClaims**>

</**TechnicalProfile**>

</**TechnicalProfiles**>

</**ClaimsProvider**>Code language: HTML, XML (xml)

### User Journeys

When using custom policies, you’ll most likely be overriding existing user journeys or writing new ones from scratch. Using the custom hierarchy above, these customizations should be added to the User Journeys extension. This can be especially useful if some of the sub-journeys are used by multiple relying party policies.

<https://tsmatz.wordpress.com/2020/05/12/azure-ad-b2c-ief-custom-policy-walkthrough/>

# A Walkthrough For Azure AD B2C Custom Policy (Identity Experience Framework)

BY [TSUYOSHI MATSUZAKI](https://tsmatz.wordpress.com/author/tsmatz/) ON [2020-05-12](https://tsmatz.wordpress.com/2020/05/12/azure-ad-b2c-ief-custom-policy-walkthrough/) • ( [33 COMMENTS](https://tsmatz.wordpress.com/2020/05/12/azure-ad-b2c-ief-custom-policy-walkthrough/#comments) )

Azure AD B2C has much flexibility for a variety of customizations with standard user flows, but sometimes, custom policy (handcrafting policy editing for XML definitions) is required, when you need more advanced and detailed configurations.  
As you can see in GitHub sample policies (see “[Custom CIAM User Journeys](https://github.com/azure-ad-b2c/samples)“), you can achieve a variety of the following scenarios by custom policies. (These are just a part of sample scenarios that you can create.)

* Additional verification steps, such as, one-time passcode, captcha, or other fine-grained verifications (for instance, “only users belonging some specific groups can log-in”)
* Support an external identity provider, which is not included in built-in providers (such as, Salesforce, ADFS, LINE, …)
* Password-less sign-in with e-mail, phone, FIDO, or others
* Advanced multi-factor authentication enhancements

For the folks who try to edit policies by oneself, here we walkthrough the policy structure in Azure AD B2C custom policy on Identity Experience Framework (IEF).  
By learning this walkthrough, I hope you’ll have a intuitive sense of “where is” or “what is” for each settings in B2C custom policy.

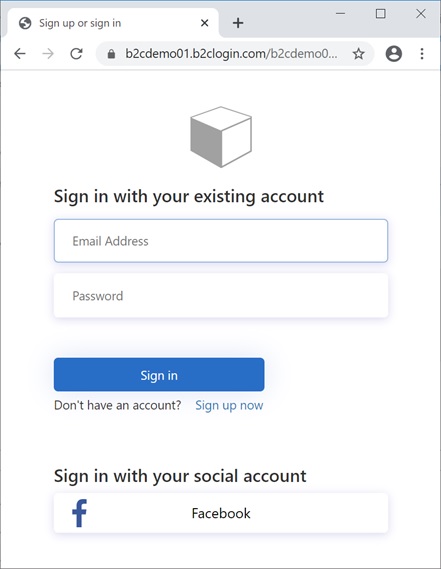
I note that this post assumes that you have basic knowledge for Azure AD B2C (such as, user flows with standard UI, simple UX customization for branding, …) and general identity technologies, such as, OpenID or OAuth.

# Starter Pack Samples and Policy Outline

If you’re new to creating custom policies, please download and apply samples in **custom policy starter pack** (go to [here](https://github.com/Azure-Samples/active-directory-b2c-custom-policy-starterpack)) at first.

Note : Before you apply sample policies in starter pack, please configure required settings, such as, registering key/secret, registering applications, and so on. See official document “[Get started with custom policies in Azure Active Directory B2C](https://docs.microsoft.com/en-us/azure/active-directory-b2c/custom-policy-get-started)“.

For instance, in {starter pack dir}/SocialAndLocalAccounts folder, there exists basic custom policy, with which there’re just Facebook and local account’s providers without any other special configurations. (See below.)  
When you need your own custom policy in your business, you can reuse these start-pack samples and it’s not needed to manually setup from scratch.

[](https://tsmatz.files.wordpress.com/2020/05/20200513_b2c_login.jpg)

Now let’s see this basic configurations in starter pack.

First, you should know that each policy can inherit (and overwrite) an existing (installed) policy with the following <BasePolicy/> element.  
In the example below, this policy B2C\_1A\_TrustFrameworkExtensions inherits the policy B2C\_1A\_TrustFrameworkBase.

Policy Inheritance

<?xml version="1.0" encoding="utf-8" ?>

<TrustFrameworkPolicy PolicyId="B2C\_1A\_TrustFrameworkExtensions" ...>

<BasePolicy> <TenantId>yourtenant.onmicrosoft.com</TenantId>

<PolicyId>B2C\_1A\_TrustFrameworkBase</PolicyId>

</BasePolicy>

/TrustFrameworkPolicy>

In SocialAndLocalAccounts folder, you will find 5 files, such as, TrustFrameworkBase.xml, TrustFrameworkExtensions.xml, and other 3 files. In these policy structure, the core configuration is built in TrustFrameworkBase.xml file, and the additional application-specific settings (such as, client id, scope, required claims) are written in TrustFrameworkExtensions.xml by inheriting TrustFrameworkBase.xml. Other 3 files (SignUpOrSignin.xml, ProfileEdit.xml, PasswordReset.xml) describes respective relying party setting by inheriting TrustFrameworkExtensions.xml.  
Thus, when you apply these policies with wrong orders, you will encounter the errors.

With a bird’s-eye view, the policy has the following outline.

Policy Outline

<TrustFrameworkPolicy ...>

<BuildingBlocks>

<ClaimsSchema ... />

<ClaimsTransformations ... />

<ClientDefinitions ... />

<ContentDefinitions ... />

</BuildingBlocks>

<ClaimsProviders ... />

<UserJourneys ... />

<RelyingParty ... />

</TrustFrameworkPolicy>

Note : For simplicity, I’ll skip descriptions about DisplayControls, Predicates, PredicateValidations, and Localization elements in BuildingBlocks in this post. See official document for these optional elements.

Each element means :

* The scenario (such as, a sign-in scenario, a password-reset scenario, …) is defined as a single UserJourney (user workflow) in UserJourneys collection.  
  The user journey is essentially the experience the user goes through.
* Each scenario (user journey) consists of multiple behaviors, called technical profile. Then these technical profiles are defined in each ClaimsProvider. For instance, local account’s ClaimsProvider might have a variety of technical profiles, such as, “requesting sign-in to user”, “providing self-assertion”, so on and so forth.
* A single technical profile uses customized UI, claims, and other components in it. These components are all defined in BuildingBlocks collection.
* The application can invoke UserJourney through a RelyingParty. For instance, when you make the user sign-in, you should define a RelyingParty which invokes the sign-in UserJourney. Each RelyingParty has a corresponding URL, such as https://{tenant}.b2clogin.com/{tenant}.onmicrosoft.com/oauth2/v2.0/authorize?p={policy name}/..., then the application uses this URL to start a signing-in flow.

From here, I’ll describe what’s doing in each elements, and finally I’ll show you a trivial example of custom policy.

# BuildingBlocks element

As I mentioned above, this element includes various building components for technical profiles.  
Each components are divided by child elements corresponding to the types of components as follows.

<BuildingBlocks>

<ClaimsSchema ... />

<ClaimsTransformations ... />

<ClientDefinitions ... />

<ContentDefinitions ... />

</BuildingBlocks>

## 1. ClaimsSchema

(See [here](https://docs.microsoft.com/en-us/azure/active-directory-b2c/claimsschema) for official reference.)

ClaimsSchema includes all claims used in policy, such as, tenant id, user id, user name, e-mail, so on and so forth. Each claim is defined as a ClaimType child’s element as follows.

<TrustFrameworkPolicy ...>

<BuildingBlocks>

<ClaimsSchema>

<ClaimType Id="client\_id">

<DisplayName>client\_id</DisplayName>

<DataType>string</DataType>

<AdminHelpText>Special parameter passed to EvoSTS.</AdminHelpText>

<UserHelpText>Special parameter passed to EvoSTS.</UserHelpText>

</ClaimType>

<ClaimType Id="scope">

<DisplayName>scope</DisplayName>

<DataType>string</DataType>

<UserHelpText>Special parameter passed for ...</UserHelpText>

</ClaimType>

<ClaimType Id="grant\_type">

<DisplayName>grant\_type</DisplayName>

<DataType>string</DataType>

<UserHelpText>...</UserHelpText>

</ClaimType>

... many many claims

</ClaimsSchema>

<ClaimsTransformations ... />

<ClientDefinitions ... />

<ContentDefinitions ... />

<DisplayControls ... />

</BuildingBlocks>

<ClaimsProviders ... />

<UserJourneys ... />

<RelyingParty ... />

</TrustFrameworkPolicy>

Some claim’s value will be extracted from external providers, however, the name of claim will differ from each protocols. For instance, object Id will be often referred as oid in OIDC/OAuth, but http://schemas.microsoft.com/identity/claims/objectidentifier in SAML.  
In such a case, you can define ClaimType with corresponding claim names as follows.

<ClaimType Id="objectId">

<DisplayName>User's Object ID</DisplayName>

<DataType>string</DataType>

<DefaultPartnerClaimTypes>

<Protocol Name="OAuth2" PartnerClaimType="oid" />

<Protocol Name="OpenIdConnect" PartnerClaimType="oid" />

<Protocol Name="SAML2" PartnerClaimType="http://schemas.microsoft.com/identity/claims/objectidentifier" />

</DefaultPartnerClaimTypes>

<UserHelpText>Object identifier (ID) of the user object in Azure AD.</UserHelpText>

</ClaimType>

<ClaimType Id="tenantId">

<DisplayName>User's Object's Tenant ID</DisplayName>

<DataType>string</DataType>

<DefaultPartnerClaimTypes>

<Protocol Name="OAuth2" PartnerClaimType="tid" />

<Protocol Name="OpenIdConnect" PartnerClaimType="tid" />

<Protocol Name="SAML2" PartnerClaimType="http://schemas.microsoft.com/identity/claims/tenantid" />

</DefaultPartnerClaimTypes>

<UserHelpText>Tenant identifier (ID) of the user object in Azure AD.</UserHelpText>

</ClaimType>

When the claim name differs from each identity providers in the same protocol, you can specify the origin name with PartnerClaimType in each technical profiles.  
For instance, displayName claim is referred as name in Facebook, but referred as screen\_name in Twitter. In this case, you can define as follows in technical profile settings.  
(Later I’ll explain the configuration about technical profiles.)

<ClaimsProvider>

<Domain>facebook.com</Domain>

<DisplayName>Facebook</DisplayName>

<TechnicalProfiles>

<TechnicalProfile Id="Facebook-OAUTH">

...

<OutputClaims>

<OutputClaim ClaimTypeReferenceId="displayName" PartnerClaimType="name" />

...

</OutputClaims>

...

</TechnicalProfile>

</TechnicalProfiles>

</ClaimsProvider>

<ClaimsProvider>

<Domain>twitter.com</Domain>

<TechnicalProfiles>

<TechnicalProfile Id="Twitter-OAUTH">

...

<OutputClaims>

<OutputClaim ClaimTypeReferenceId="displayName" PartnerClaimType="screen\_name" />

...

</OutputClaims>

...

</TechnicalProfile>

</TechnicalProfiles>

</ClaimsProvider>

Not only provider’s claims, but also you can use the application-specific claims.  
For instance, if you want to distinguish whether it’s logged-in by Twitter or not in custom policy, you can define the following flag (authenticationProvider) as a claim, and set the appropriate value to this claim in extracting.  
In UserJourney, you can use this flag for building a logic.

<TrustFrameworkPolicy ...>

<BuildingBlocks>

<ClaimsSchema>

<!-- Define "AuthenticationProvider" claim -->

<ClaimType Id="authenticationProvider">

<DisplayName>AuthenticationProvider</DisplayName>

<DataType>string</DataType>

<UserHelpText>Specifies authenticated IDP name</UserHelpText>

</ClaimType>

...

</ClaimsSchema>

...

</BuildingBlocks>

<ClaimsProviders>

<ClaimsProvider>

<Domain>twitter.com</Domain>

<TechnicalProfiles>

<TechnicalProfile Id="Twitter-OAUTH">

...

<OutputClaims>

<!-- Set "twitter" in this claim, when logged-in with Twitter -->

<OutputClaim ClaimTypeReferenceId="authenticationProvider" DefaultValue="twitter" AlwaysUseDefaultValue="true" />

...

</OutputClaims>

...

</TechnicalProfile>

</TechnicalProfiles>

</ClaimsProvider>

...

</ClaimsProviders>

<UserJourneys ... />

<RelyingParty ... />

</TrustFrameworkPolicy>

## 2. ClaimsTransformations

(See [here](https://docs.microsoft.com/en-us/azure/active-directory-b2c/claimstransformations) for official reference.)

When you need primitive logic for claims, such as, string converter, formatting, condition checking, so on and so forth, you can use ClaimsTransformation and describe the rules of claim’s transformation that can be used in user journeys.

In ClaimsTransformations, you can use a variety of transformation methods as follows.

|  |  |
| --- | --- |
| **Type** | **Supported Methods** |
| General | CopyClaim, DoesClaimExist, Hash |
| String | AssertStringClaimsAreEqual, ChangeCase, CreateStringClaim, CompareClaims, CompareClaimToValue, CreateRandomString, FormatStringClaim, FormatStringMultipleClaims, GetLocalizedStringsTransformation, GetMappedValueFromLocalizedCollection, LookupValue, NullClaim, ParseDomain, SetClaimsIfRegexMatch, SetClaimsIfStringsAreEqual, SetClaimsIfStringsMatch, StringContains, StringSubstring, StringReplace, StringJoin, StringSplit |
| StringCollection | AddItemToStringCollection, AddParameterToStringCollection, GetSingleItemFromStringCollection, StringCollectionContains, StringCollectionContainsClaim |
| Boolean | AndClaims, AssertBooleanClaimIsEqualToValue, CompareBooleanClaimToValue, OrClaims |
| Integer | ConvertNumberToStringClaim |
| Date | AssertDateTimeIsGreaterThan, ConvertDateToDateTimeClaim, ConvertDateTimeToDateClaim, GetCurrentDateTime, DateTimeComparison |
| JSON | GenerateJson, GetClaimFromJson, GetClaimsFromJsonArray, GetNumericClaimFromJson, GetSingleItemFromJson, GetSingleValueFromJsonArray, XmlStringToJsonString |
| Phone number | ConvertPhoneNumberClaimToString, ConvertStringToPhoneNumberClaim, GetNationalNumberAndCountryCodeFromPhoneNumberString |
| External account | CreateAlternativeSecurityId, AddItemToAlternativeSecurityIdCollection, GetIdentityProvidersFromAlternativeSecurityIdCollectionTransformation, RemoveAlternativeSecurityIdByIdentityProvider |

For instance, when you check if user’s password matches a password on banned password list (such as “password”, “Password”, or “P@ssw0rd”), you can use the following LookupValue transformation method. If it matches one of InputParameters, the claim bannedPassword will be set to “true” (string) in this transformation.  
After this transformation is executed, you can check whether bannedPassword claim is “true” or not using CompareClaims method in transformations.  
(See “[Sign up and Password reset with banned password list](https://github.com/azure-ad-b2c/samples/tree/master/policies/banned-password-list-no-API)” sample in GitHub.)

<ClaimsTransformation Id="passwordToMatchBanList" TransformationMethod="LookupValue">

<InputClaims>

<InputClaim ClaimTypeReferenceId = "newPassword" TransformationClaimType="inputParameterId" />

</InputClaims>

<InputParameters>

<InputParameter Id="password" DataType="string" Value="true" />

<InputParameter Id="Password" DataType="string" Value="true" />

<InputParameter Id="P@ssw0rd" DataType="string" Value="true" />

<InputParameter Id="errorOnFailedLookup" DataType="boolean" Value="false" />

</InputParameters>

<OutputClaims>

<OutputClaim ClaimTypeReferenceId = "bannedPassword" TransformationClaimType="outputClaim" />

</OutputClaims>

</ClaimsTransformation>

In a technical profile, you can call this transformation as follow.  
(Later I’ll explain details about technical profiles.)

<TechnicalProfile Id="PasswordCheckTest">

<DisplayName>Password Check</DisplayName>

<Protocol Name="Proprietary" Handler="Web.TPEngine.Providers.ClaimsTransformationProtocolProvider, Web.TPEngine, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />

...

<OutputClaimsTransformations>

<!-- Check whether it's one of banned passowrds and set "bannedPassword" -->

<OutputClaimsTransformation ReferenceId="passwordToMatchBanList" />

... (Next, check whether "bannedPassword" is "true" or not)

</OutputClaimsTransformations>

</TechnicalProfile>

## 3. ClientDefinitions

This element specifies the client (end-user device). Ordinally, a single ClientDefinition (for web client) is used in ClientDefinitions collection as follows.

<ClientDefinitions>

<ClientDefinition Id="DefaultWeb">

<ClientUIFilterFlags>LineMarkers, MetaRefresh</ClientUIFilterFlags>

</ClientDefinition>

</ClientDefinitions>

This is referred in each user journeys as follows.  
(Later I’ll explain UserJourneys.)

<UserJourney Id="SignUpOrSignIn">

<OrchestrationSteps>

<OrchestrationStep Order="1" ...>

...

</OrchestrationStep>

<OrchestrationStep Order="2" ...>

...

</OrchestrationStep>

<OrchestrationStep Order="3" ...>

...

</OrchestrationStep>

...

</OrchestrationSteps>

<ClientDefinition ReferenceId="DefaultWeb" />

</UserJourney>

## 4. ContentDefinitions

(See [here](https://docs.microsoft.com/en-us/azure/active-directory-b2c/contentdefinitions) for official reference.)

All user-interfaces (i.e, pages) used in technical profiles should be defined as each ContentDefinition.  
The user-interfaces in Azure AD B2C (such as, sign-in/sign-up, identity selector, profile update, …) have corresponding content definition IDs. When you want to define ContentDefinition for this UI, specify this ID in Id attribute.  
When you use default html template (default UI) in B2C without any customization, specify default tempalte URI for LoadUri as follows.

Note : See [here](https://docs.microsoft.com/en-us/azure/active-directory-b2c/contentdefinitions) for all content IDs and default html URIs.

<ContentDefinitions>

<ContentDefinition Id="api.signuporsignin">

<LoadUri>~/tenant/templates/AzureBlue/unified.cshtml</LoadUri>

<RecoveryUri>~/common/default\_page\_error.html</RecoveryUri>

<DataUri>urn:com:microsoft:aad:b2c:elements:unifiedssp:1.0.0</DataUri>

<Metadata>

<Item Key="DisplayName">Signin and Signup</Item>

</Metadata>

</ContentDefinition>

<ContentDefinition Id="api.idpselections">

<LoadUri>~/tenant/templates/AzureBlue/idpSelector.cshtml</LoadUri>

<RecoveryUri>~/common/default\_page\_error.html</RecoveryUri>

<DataUri>urn:com:microsoft:aad:b2c:elements:idpselection:1.0.0</DataUri>

<Metadata>

<Item Key="DisplayName">Idp selection page</Item>

<Item Key="language.intro">Sign in</Item>

</Metadata>

</ContentDefinition>

...

</ContentDefinitions>

When you customize a page for some content ID, specify URL of your customized page for LoadUri entity as follows.

<ContentDefinition Id="api.selfasserted">

<LoadUri>https://xxxxx.azurewebsites.net/b2c-test.html</LoadUri>

<RecoveryUri>~/common/default\_page\_error.html</RecoveryUri>

<DataUri>urn:com:microsoft:aad:b2c:elements:selfasserted:1.1.0</DataUri>

<Metadata>

<Item Key="DisplayName">Collect information from user page</Item>

</Metadata>

</ContentDefinition>

In customized HTML page, you just insert <div id="api"></div> to show the required elements. Then you are free to decorate with custom styles or custom contents in your HTML page. (You can also include JavaScript custom code.)  
For details about B2C UI customization, see “[Customize the user interface in Azure Active Directory B2C](https://docs.microsoft.com/en-us/azure/active-directory-b2c/customize-ui-overview)” in official document.

<!DOCTYPE html>

<html>

<head>

<title>Test</title>

</head>

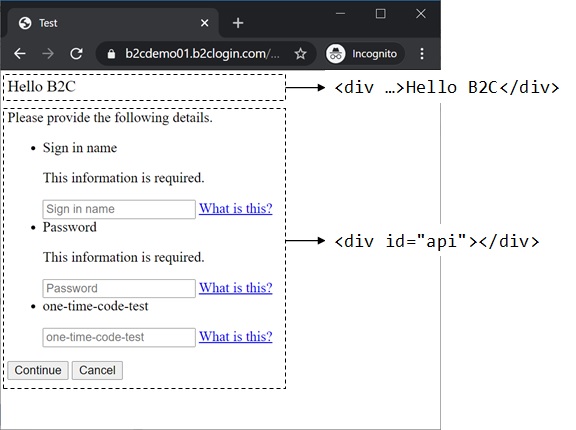
<body>

<div id="hell-test" style="font-size:large;">Hello B2C</div>

<div id="api"></div>

</body>

</html>

[](https://tsmatz.files.wordpress.com/2020/05/20200513_ui_customize.jpg)

# ClaimsProviders element

(See [here](https://docs.microsoft.com/en-us/azure/active-directory-b2c/claimsproviders) for official reference.)

As I mentioned above, ClaimsProvider includes all behaviors for user journeys, called a technical profile (TechnicalProfile element). Every claim providers must have one or more technical profiles.

Not only identity providers (both external account and local account), but also all the required behaviors should be included as TechnicalProfile (and its ClaimsProvider).

In this section, we’ll see many of frequently used ClaimsProvider and TechnicalProfile by categorizing these objects.

## 1. Mandatory Provider (Trust Framework Policy Engine Provider)

Fist of all, I’ll show you a trust framework policy engine provider.  
This is a mandatory provider for all custom policies, then anyway you should include the following ClaimsProvider element in ClaimsProviders.

<ClaimsProvider>

<DisplayName>Trustframework Policy Engine TechnicalProfiles</DisplayName>

<TechnicalProfiles>

<TechnicalProfile Id="TpEngine\_c3bd4fe2-1775-4013-b91d-35f16d377d13">

<DisplayName>Trustframework Policy Engine Default Technical Profile</DisplayName>

<Protocol Name="None" />

<Metadata>

<Item Key="url">{service:te}</Item>

</Metadata>

</TechnicalProfile>

</TechnicalProfiles>

</ClaimsProvider>

## 2. External Provider

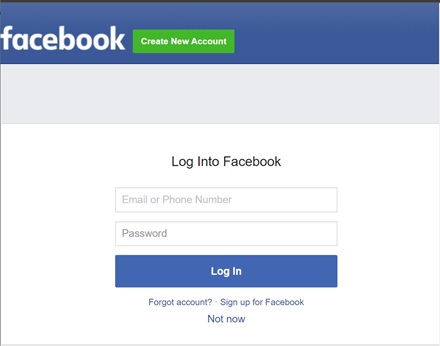
Next, let’s see more comprehensive providers, external providers.

When you use external identity providers (such as, Facebook, Twitter, Goggle, so on and so forth) in custom policy, please describe how to authenticate in this provider with protocol setting (OAuth, OpenID Connect, SAML, …), endpoint url, request’s attributes (such as, client id, scope, grant type, …) and other related properties.

Let’s see the case of Facebook, which uses OAuth 2.0 (RFC 6749).  
In Facebook, the app will authenticate with the following flow. (Here I assume you have basic knowledge about OAuth flow.)

step1 : go to authorization endpoint and show sign-in UI (Facebook)

GET https://www.facebook.com/dialog/oauth?client\_id=2694572064107190&redirect\_uri=https%3A%2F%2Fxxxxx.b2clogin.com%2Fxxxxx.onmicrosoft.com%2Foauth2%2Fauthresp&response\_type=code&scope=email+public\_profile&state=...

[](https://tsmatz.files.wordpress.com/2020/05/20200513_facebook_signin.jpg)

step2 : redirected to your site with authorization code, after logging-in (Facebook)

GET https://xxxxx.b2clogin.com/xxxxx.onmicrosoft.com/oauth2/authresp?code=AQAXZHF1bK...

step3 : request access token using previous code (Facebook)

POST https://graph.facebook.com/oauth/access\_token

Content-Type: application/x-www-form-urlencoded

grant\_type=authorization\_code

&code=AQAXZHF1bK...

&client\_id=0000000000000000

&client\_secret=14dd677c2d...

&scope=email+public\_profile

&redirect\_uri=https%3A%2F%2Fxxxxx.b2clogin.com%2Fxxxxx.onmicrosoft.com%2Foauth2%2Fauthresp

HTTP/1.1 200 OK

Content-Type: application/json; charset=UTF-8

facebook-api-version: v3.0

{

"access\_token":"EAAmSsvKz6...",

"token\_type":"bearer",

"expires\_in":5183967

}

step4 : get claims using previous access token (Facebook)

GET https://graph.facebook.com/me?fields=id,first\_name,last\_name,name,email&access\_token=EAAmSsvKz6...

HTTP/1.1 200 OK

Content-Type: text/javascript; charset=UTF-8

facebook-api-version: v7.0

{

"id": "3084092101648447",

"first\_name": "Tsuyoshi",

"last\_name": "Matsuzaki",

"name": "Tsuyoshi Matsuzaki",

"email": "tsmatz\u0040outlook.com"

}

Note : Please disable appsecret\_proof and switch your app mode from “Development” to “Live” in Facebook app settings.

[A computer screen shot of a computer

Description automatically generated](https://tsmatz.files.wordpress.com/2020/05/20200513_appsecret_proof.jpg)

With a technical profile in B2C, you will be able to write this OAuth 2.0 flow as follows. (When this technical profile doesn’t work correctly, please trace the previous flow again.)

Facebook claims provider (OAuth2)

<ClaimsProvider>

<Domain>facebook.com</Domain>

<DisplayName>Facebook</DisplayName>

<TechnicalProfiles>

<TechnicalProfile Id="Facebook-OAUTH">

<DisplayName>Facebook</DisplayName>

<Protocol Name="OAuth2" />

<Metadata>

<Item Key="ProviderName">facebook</Item>

<Item Key="authorization\_endpoint">https://www.facebook.com/dialog/oauth</Item>

<Item Key="AccessTokenEndpoint">https://graph.facebook.com/oauth/access\_token</Item>

<Item Key="ClaimsEndpoint">https://graph.facebook.com/me?fields=id,first\_name,last\_name,name,email</Item>

<Item Key="HttpBinding">GET</Item>

<Item Key="AccessTokenResponseFormat">json</Item>

<Item Key="client\_id">0000000000000000</Item>

<Item Key="scope">email public\_profile</Item>

<Item Key="UsePolicyInRedirectUri">0</Item>

</Metadata>

<CryptographicKeys>

<Key Id="client\_secret" StorageReferenceId="B2C\_1A\_FacebookSecret" />

</CryptographicKeys>

<InputClaims />

...

</TechnicalProfile>

</TechnicalProfiles>

</ClaimsProvider>

Note : In order to provide secure information, such as, app key, app password, or app secret, you must register these values as policy keys (manual registration) using portal UI, and specify this key using <CryptographicKeys /> element as above. (Almost providers may need this kind of app key for federations.)

Here I show you several examples corresponding other protocols.

Twitter claims provider (OAuth1)

<ClaimsProvider>

<Domain>twitter.com</Domain>

<DisplayName>Twitter</DisplayName>

<TechnicalProfiles>

<TechnicalProfile Id="Twitter-OAUTH1">

<DisplayName>Twitter</DisplayName>

<Protocol Name="OAuth1" />

<Metadata>

<Item Key="ProviderName">Twitter</Item>

<Item Key="authorization\_endpoint">https://api.twitter.com/oauth/authenticate</Item>

<Item Key="access\_token\_endpoint">https://api.twitter.com/oauth/access\_token</Item>

<Item Key="request\_token\_endpoint">https://api.twitter.com/oauth/request\_token</Item>

<Item Key="ClaimsEndpoint">https://api.twitter.com/1.1/account/verify\_credentials.json?include\_email=true</Item>

<Item Key="ClaimsResponseFormat">json</Item>

<Item Key="client\_id">xxxxxxxxxx</Item>

</Metadata>

<CryptographicKeys>

<Key Id="client\_secret" StorageReferenceId="B2C\_1A\_TwitterSecret" />

</CryptographicKeys>

<InputClaims />

...

</TechnicalProfile>

</TechnicalProfiles>

</ClaimsProvider>

Microsoft Account (OpenID Connect)

<ClaimsProvider>

<Domain>live.com</Domain>

<DisplayName>Microsoft Account</DisplayName>

<TechnicalProfiles>

<TechnicalProfile Id="MSA-OIDC">

<DisplayName>Microsoft Account</DisplayName>

<Protocol Name="OpenIdConnect" />

<Metadata>

<Item Key="ProviderName">https://login.live.com</Item>

<Item Key="METADATA">https://login.live.com/.well-known/openid-configuration</Item>

<Item Key="response\_types">code</Item>

<Item Key="response\_mode">form\_post</Item>

<Item Key="scope">openid profile email</Item>

<Item Key="HttpBinding">POST</Item>

<Item Key="UsePolicyInRedirectUri">0</Item>

<Item Key="client\_id">xxxxxxxxxx</Item>

</Metadata>

<CryptographicKeys>

<Key Id="client\_secret" StorageReferenceId="B2C\_1A\_MSASecret" />

</CryptographicKeys>

...

</TechnicalProfile>

</TechnicalProfiles>

</ClaimsProvider>

Salesforce (SAML2)

<ClaimsProvider>

<Domain>salesforce</Domain>

<DisplayName>Salesforce</DisplayName>

<TechnicalProfiles>

<TechnicalProfile Id="salesforce">

<DisplayName>Salesforce</DisplayName>

<Description>Login with your Salesforce account</Description>

<Protocol Name="SAML2"/>

<Metadata>

<Item Key="WantsEncryptedAssertions">false</Item>

<Item Key="WantsSignedAssertions">false</Item>

<Item Key="PartnerEntity">https://contoso-dev-ed.my.salesforce.com/.well-known/samlidp.xml</Item>

</Metadata>

<CryptographicKeys>

<Key Id="SamlAssertionSigning" StorageReferenceId="B2C\_1A\_SAMLSigningCert"/>

<Key Id="SamlMessageSigning" StorageReferenceId="B2C\_1A\_SAMLSigningCert"/>

</CryptographicKeys>

...

</TechnicalProfile>

</TechnicalProfiles>

</ClaimsProvider>

Note : When you use Azure Active Directory as an external provider, you can use a claim provider with OpenID Connect protocol. (See [here](https://docs.microsoft.com/en-us/azure/active-directory-b2c/identity-provider-azure-ad-multi-tenant-custom) for details.)

The supported protocol (see <Protocol /> elements above) is “OAuth1“, “OAuth2“, “SAML2“, “OpenIdConnect“, “Proprietary“, and “None“.  
The “Proprietary” protocol is not used for external providers, but used for calling .NET assembly in B2C. I’ll show you about this protocol in the next section for local account provider.  
You can nest a technical profile (with IncludeTechnicalProfile) and then “None” protocol will be used in the nested provider. (Or, “None” is also used in built-in providers, such as, an engine provider as above.)

What if other (not-supported) protocols, such as, Basic auth or Form auth, is used in external providers ?  
In this case, you can build your custom logic as REST API, and you can invoke this intermediate service (custom REST API) in your technical profile using the .NET assembly : Web.TPEngine.Providers.RestfulProvider, Web.TPEngine, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null.  
Later in this post, I’ll show you an example of how to use this Restful provider.

<TechnicalProfiles>

<TechnicalProfile Id="REST-EmailVerification">

<DisplayName>Validate user input data and return loyaltyNumber claim</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://xxxxx.azurewebsites.net/test-identity</Item>

<Item Key="AuthenticationType">None</Item>

<Item Key="SendClaimsIn">Body</Item>

</Metadata>

<InputClaims>

<InputClaim ClaimTypeReferenceId="email" />

<InputClaim ClaimTypeReferenceId="verificationCode" PartnerClaimType="code" />

</InputClaims>

<UseTechnicalProfileForSessionManagement ReferenceId="SM-Noop" />

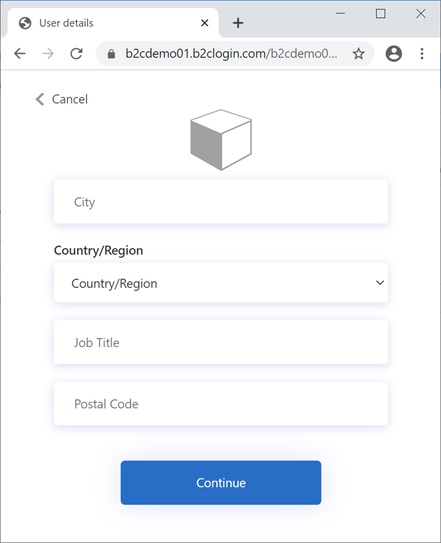
</TechnicalProfile>

</TechnicalProfiles>

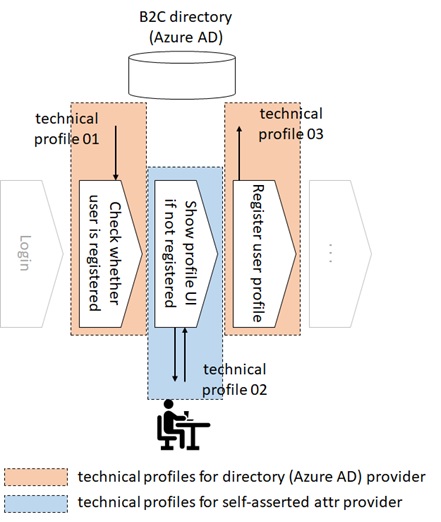
Note (Session Provider) : All technical profiles (except for mandatory profile and JWT issuer) should have one of session providers in a list [here](https://docs.microsoft.com/en-us/azure/active-directory-b2c/custom-policy-reference-sso) to retain session.  
In this post, I don’t describe about session providers, but see custom policy samples in starter pack.

## 3. User Interaction’s Provider

When the user logs-in with external accounts (Facebook, Twitter, Salesforce, …) for the first time, B2C asks user to input the additional required properties (i.e, self-asserted attributes) as follows, and these properties will be stored in B2C directory.

[](https://tsmatz.files.wordpress.com/2020/05/20200513_self_asserted.jpg)

These activities should also be defined as claim’s providers and its technical profiles. (See below.)

[](https://tsmatz.files.wordpress.com/2020/05/20200513_tp_structure.jpg)

First, I’ll show you claim’s providers and technical profiles for self-asserted attributes (the above profiles filled with blue color).

The following is an example provider for self-asserted attributes. (Here I copied the definition from {starter pack}/SocialAndLocalAccounts/TrustFrameworkBase.xml.)  
This technical profile shows a page for self-asserted attributes (content definition ID “api.selfasserted“) to the user and collects user’s attributes. This is done by B2C built-in .NET assembly : Web.TPEngine.Providers.SelfAssertedAttributeProvider, Web.TPEngine, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null .  
In order to invoke this bult-in .NET assembly, you should specify “Proprietary” for Protocol as follows.

<ClaimsProvider>

<DisplayName>Self Asserted</DisplayName>

<TechnicalProfiles>

<TechnicalProfile Id="SelfAsserted-Social">

<DisplayName>User ID signup</DisplayName>

<Protocol Name="Proprietary" Handler="Web.TPEngine.Providers.SelfAssertedAttributeProvider, Web.TPEngine, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />

<Metadata>

<Item Key="ContentDefinitionReferenceId">api.selfasserted</Item>

</Metadata>

<CryptographicKeys>

<Key Id="issuer\_secret" StorageReferenceId="B2C\_1A\_TokenSigningKeyContainer" />

</CryptographicKeys>

<InputClaims>

<InputClaim ClaimTypeReferenceId="displayName" />

<InputClaim ClaimTypeReferenceId="givenName" />

<InputClaim ClaimTypeReferenceId="surname" />

</InputClaims>

<OutputClaims>

<!-- These claims are not shown (See descriptions below) -->

<OutputClaim ClaimTypeReferenceId="objectId" />

<OutputClaim ClaimTypeReferenceId="newUser" />

<OutputClaim ClaimTypeReferenceId="executed-SelfAsserted-Input" DefaultValue="true" />

<!-- These claims are shown in UI, if it's not provided by IDP -->

<OutputClaim ClaimTypeReferenceId="displayName" />

<OutputClaim ClaimTypeReferenceId="givenName" />

<OutputClaim ClaimTypeReferenceId="surname" />

</OutputClaims>

<ValidationTechnicalProfiles>

<ValidationTechnicalProfile ReferenceId="AAD-UserWriteUsingAlternativeSecurityId" />

</ValidationTechnicalProfiles>

</TechnicalProfile>

</TechnicalProfiles>

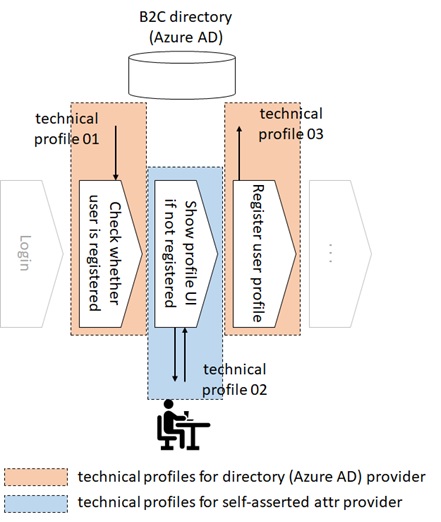
</ClaimsProvider>

If sign-in user does not exist in the directory, a technical profile AAD-UserWriteUsingAlternativeSecurityId (see <ValidationTechnicalProfiles /> above) creates a local account and returns objectId and newUser (which is the flag information for whether it’s new user or not).

Here I showed you a technical profile for only providing self-asserted attributes in external accounts, however, you should define other technical profiles for all user interactions, such as, sign-in, sign-up, password reset, or password change.  
We’ll discuss these UI-centric profiles again in local account’s provider later.

## 4. Directory (Azure AD) Interaction’s Provider

Next, let’s see the directory side of technical profiles (the following profiles filled with orange color).

[](https://tsmatz.files.wordpress.com/2020/05/20200513_tp_structure.jpg)

In Azure AD B2C, proven Azure Active Directory (Azure AD) is used as a backend directory. (All users and applications are stored on Azure AD directory in the tenant.)  
Then all interactions (read / write operations) against a directory should be defined as an Azure AD claim provider.

For directory (Azure AD) interactions, you can use the built-in .NET assembly : Web.TPEngine.Providers.AzureActiveDirectoryProvider, Web.TPEngine, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null .

Using <Metadata /> element, specify the type of operations (“Read” or “Write“) and other behavior’s properties to .NET assembly. For instance, by specifying false in RaiseErrorIfClaimsPrincipalAlreadyExists property in <Metadata /> element, the assembly doesn’t raise errors even when there already exists a user with same objectId in a write operation.

<ClaimsProvider>

<DisplayName>Azure Active Directory</DisplayName>

<TechnicalProfiles>

<TechnicalProfile Id="AAD-Common">

<DisplayName>Azure Active Directory</DisplayName>

<Protocol Name="Proprietary" Handler="Web.TPEngine.Providers.AzureActiveDirectoryProvider, Web.TPEngine, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />

<CryptographicKeys>

<Key Id="issuer\_secret" StorageReferenceId="B2C\_1A\_TokenSigningKeyContainer" />

</CryptographicKeys>

<!-- To suppress SelfAsserted provider from invoking on validation profiles -->

<IncludeInSso>false</IncludeInSso>

<UseTechnicalProfileForSessionManagement ReferenceId="SM-Noop" />

</TechnicalProfile>

<TechnicalProfile Id="AAD-UserWriteUsingAlternativeSecurityId">

<Metadata>

<Item Key="Operation">Write</Item>

<Item Key="RaiseErrorIfClaimsPrincipalAlreadyExists">true</Item>

<Item Key="UserMessageIfClaimsPrincipalAlreadyExists">You are already registered.</Item>

</Metadata>

<IncludeInSso>false</IncludeInSso>

<InputClaimsTransformations>

<InputClaimsTransformation ReferenceId="CreateOtherMailsFromEmail" />

</InputClaimsTransformations>

<InputClaims>

<InputClaim ClaimTypeReferenceId="AlternativeSecurityId" PartnerClaimType="alternativeSecurityId" Required="true" />

</InputClaims>

<PersistedClaims>

<!-- Required claims -->

<PersistedClaim ClaimTypeReferenceId="alternativeSecurityId" />

<PersistedClaim ClaimTypeReferenceId="userPrincipalName" />

<PersistedClaim ClaimTypeReferenceId="mailNickName" DefaultValue="unknown" />

<PersistedClaim ClaimTypeReferenceId="displayName" DefaultValue="unknown" />

<!-- Optional claims -->

<PersistedClaim ClaimTypeReferenceId="otherMails" />

<PersistedClaim ClaimTypeReferenceId="givenName" />

<PersistedClaim ClaimTypeReferenceId="surname" />

</PersistedClaims>

<OutputClaims>

<OutputClaim ClaimTypeReferenceId="objectId" />

<OutputClaim ClaimTypeReferenceId="newUser" PartnerClaimType="newClaimsPrincipalCreated" />

<!--　The following other mails claim is needed for the case when a user is created,

we get otherMails from directory. Self-asserted provider also has an　OutputClaims,

and if this is absent, Self-Asserted provider will prompt the user for otherMails. -->

<OutputClaim ClaimTypeReferenceId="otherMails" />

</OutputClaims>

<IncludeTechnicalProfile ReferenceId="AAD-Common" />

<UseTechnicalProfileForSessionManagement ReferenceId="SM-AAD" />

</TechnicalProfile>

<TechnicalProfile Id="AAD-UserWriteProfileUsingObjectId">

<Metadata>

<Item Key="Operation">Write</Item>

<Item Key="RaiseErrorIfClaimsPrincipalAlreadyExists">false</Item>

<Item Key="RaiseErrorIfClaimsPrincipalDoesNotExist">true</Item>

</Metadata>

<IncludeInSso>false</IncludeInSso>

<InputClaims>

<InputClaim ClaimTypeReferenceId="objectId" Required="true" />

</InputClaims>

<PersistedClaims>

<!-- Required claims -->

<PersistedClaim ClaimTypeReferenceId="objectId" />

<!-- Optional claims -->

<PersistedClaim ClaimTypeReferenceId="givenName" />

<PersistedClaim ClaimTypeReferenceId="surname" />

</PersistedClaims>

<IncludeTechnicalProfile ReferenceId="AAD-Common" />

</TechnicalProfile>

<TechnicalProfile Id="AAD-UserReadUsingAlternativeSecurityId">

<Metadata>

<Item Key="Operation">Read</Item>

<Item Key="RaiseErrorIfClaimsPrincipalDoesNotExist">true</Item>

<Item Key="UserMessageIfClaimsPrincipalDoesNotExist">User does not exist.</Item>

</Metadata>

<InputClaims>

<InputClaim ClaimTypeReferenceId="AlternativeSecurityId" PartnerClaimType="alternativeSecurityId" Required="true" />

</InputClaims>

<OutputClaims>

<!-- Required claims -->

<OutputClaim ClaimTypeReferenceId="objectId" />

<!-- Optional claims -->

<OutputClaim ClaimTypeReferenceId="userPrincipalName" />

<OutputClaim ClaimTypeReferenceId="displayName" />

<OutputClaim ClaimTypeReferenceId="otherMails" />

<OutputClaim ClaimTypeReferenceId="givenName" />

<OutputClaim ClaimTypeReferenceId="surname" />

</OutputClaims>

<IncludeTechnicalProfile ReferenceId="AAD-Common" />

</TechnicalProfile>

</TechnicalProfiles>

</ClaimsProvider>

As you saw in previous “3. User Interaction’s Provider” section, a lot of technical profiles will invoke these Azure AD technical profiles as backend read/write operations into a directory.

## 5. Local Account’s Provider

Now we’re ready to build claim providers (including technical profiles) for local account. The idea is the same as examples in the case of external account which I have explained above.

Unlike an external account, the operations for sign-in and sign-up in local account should be provided with user-interaction’s providers.  
For instance, the following is an example for the sign-in operation. This uses api.selfasserted (which is the same as one for the self-asserted attributes in external account) as a content definition for UI template.

<ClaimsProvider>

<DisplayName>Local Account</DisplayName>

<TechnicalProfiles>

<TechnicalProfile Id="SelfAsserted-LocalAccountSignin-Email">

<DisplayName>Local Account Signin</DisplayName>

<Protocol Name="Proprietary" Handler="Web.TPEngine.Providers.SelfAssertedAttributeProvider, Web.TPEngine, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />

<Metadata>

<Item Key="SignUpTarget">SignUpWithLogonEmailExchange</Item>

<Item Key="setting.operatingMode">Email</Item>

<Item Key="ContentDefinitionReferenceId">api.selfasserted</Item>

</Metadata>

<InputClaims>

<InputClaim ClaimTypeReferenceId="signInName" />

</InputClaims>

<OutputClaims>

<OutputClaim ClaimTypeReferenceId="signInName" Required="true" />

<OutputClaim ClaimTypeReferenceId="password" Required="true" />

<OutputClaim ClaimTypeReferenceId="objectId" />

<OutputClaim ClaimTypeReferenceId="authenticationSource" />

</OutputClaims>

<ValidationTechnicalProfiles>

<ValidationTechnicalProfile ReferenceId="login-NonInteractive" />

</ValidationTechnicalProfiles>

</TechnicalProfile>

</TechnicalProfiles>

</ClaimsProvider>

After the user proceeds to login, a technical profile login-NonInteractive (see above) is invoked for validation.  
As I mentioned above, Azure AD B2C uses Azure Active Directory (Azure AD) as a backend directory. For this reason, this technical profile (login-NonInteractive) will ask whether username (signInName) and password is valid for Azure Active Directory.  
To accomplish this, ROPC grant flow (i.e, the flow with grant\_type=password) is used for validation as follows.

<ClaimsProvider>

<DisplayName>Local Account SignIn</DisplayName>

<TechnicalProfiles>

<TechnicalProfile Id="login-NonInteractive">

<DisplayName>Local Account SignIn</DisplayName>

<Protocol Name="OpenIdConnect" />

<Metadata>

<Item Key="UserMessageIfClaimsPrincipalDoesNotExist">We can't seem to find your account</Item>

<Item Key="UserMessageIfInvalidPassword">Your password is incorrect</Item>

<Item Key="UserMessageIfOldPasswordUsed">Looks like you used an old password</Item>

<Item Key="ProviderName">https://sts.windows.net/</Item>

<Item Key="METADATA">https://login.microsoftonline.com/{tenant}/.well-known/openid-configuration</Item>

<Item Key="authorization\_endpoint">https://login.microsoftonline.com/{tenant}/oauth2/token</Item>

<Item Key="response\_types">id\_token</Item>

<Item Key="response\_mode">query</Item>

<Item Key="scope">email openid</Item>

<!-- Policy Engine Clients -->

<Item Key="UsePolicyInRedirectUri">false</Item>

<Item Key="HttpBinding">POST</Item>

</Metadata>

<InputClaims>

<InputClaim ClaimTypeReferenceId="signInName" PartnerClaimType="username" Required="true" />

<InputClaim ClaimTypeReferenceId="password" Required="true" />

<InputClaim ClaimTypeReferenceId="grant\_type" DefaultValue="password" />

<InputClaim ClaimTypeReferenceId="scope" DefaultValue="openid" />

<InputClaim ClaimTypeReferenceId="nca" PartnerClaimType="nca" DefaultValue="1" />

</InputClaims>

<OutputClaims>

<OutputClaim ClaimTypeReferenceId="objectId" PartnerClaimType="oid" />

<OutputClaim ClaimTypeReferenceId="tenantId" PartnerClaimType="tid" />

<OutputClaim ClaimTypeReferenceId="givenName" PartnerClaimType="given\_name" />

<OutputClaim ClaimTypeReferenceId="surName" PartnerClaimType="family\_name" />

<OutputClaim ClaimTypeReferenceId="displayName" PartnerClaimType="name" />

<OutputClaim ClaimTypeReferenceId="userPrincipalName" PartnerClaimType="upn" />

</OutputClaims>

</TechnicalProfile>

</TechnicalProfiles>

</ClaimsProvider>

Note : As I mentioned in my [old post](https://tsmatz.wordpress.com/2013/07/11/native-application-mobile-app-azure-active-directory-login-authentication/) (sorry, but it’s written in Japanese, since it was written a long time ago), ROPC grant flow is not recommended to be used in production for Azure AD authentication. However, in Azure AD B2C, this flow is used for only local account sign-in, in which the usage is so limited (there will not be MFA or AD federation in this scenario) and used only in the backend.  
This flow (ROPC grant flow) is not supported in Azure AD v2 endpoint, then you should use v1 endpoint as above.

In local account, the other scenarios, such as, sign-up, password-change, and password-reset are also implemented in B2C claim’s providers with user-interactions and directory-interactions. As a result, you might have to define a lot of technical profiles for both UI and directory interactions for local account.  
For simplicity, here I skip explanations for all these other technical profiles, but you can refer TrustFrameworkBase.xml in starter pack samples.

<ClaimsProvider>

<DisplayName>Local Account</DisplayName>

<TechnicalProfiles>

<!-- For sign-in user-interaction -->

<TechnicalProfile Id="SignIn-Sample">

...

</TechnicalProfile>

<!-- For sign-up user-interaction -->

<TechnicalProfile Id="SignUp-Sample">

...

<ValidationTechnicalProfiles>

<ValidationTechnicalProfile ReferenceId="AAD-WriteUser" />

</ValidationTechnicalProfiles>

</TechnicalProfile>

<!-- For password-change user-interaction -->

<TechnicalProfile Id="ChangePassword-Sample">

...

<ValidationTechnicalProfiles>

<ValidationTechnicalProfile ReferenceId="AAD-WritePassword" />

</ValidationTechnicalProfiles>

</TechnicalProfile>

...

</TechnicalProfiles>

</ClaimsProvider>

...

<ClaimsProvider>

<DisplayName>Azure Active Directory</DisplayName>

<TechnicalProfiles>

<!-- For writing a user into a directory -->

<TechnicalProfile Id="AAD-WriteUser">

...

</TechnicalProfile>

<!-- For changing password into a directory -->

<TechnicalProfile Id="AAD-WritePassword">

...

</TechnicalProfile>

...

</TechnicalProfiles>

</ClaimsProvider>

## 6. JWT Issuer Provider

After the user has successfully logged-in, B2C should provide JWT (token) to the applications.

As I mentioned in my old post (see [here](https://tsmatz.wordpress.com/2016/03/08/azure-ad-msa-v2-endpoint-validate-id_token/) for details), a given JWT (both id token and access token) provided by Azure AD consists of the following parts :

* JWT string has 3 tokens delimited by the dot (.) character.
* Each token (3 tokens) respectively means as follows :
  1. Information about certificate (e.g, the type of key, key id (X.509 Thumprint), and so on) :  
     These values doesn’t change for each authentication. (Fixed values)
  2. Extracted claims (e.g, user principal, user name, scope, tenant id, token expiration, and so on) :
  3. A digital signature :  
     This is a byte code which is generated from previous token1 + token2 string and certificate (private key). Thus, if token string (either token1 or token2) is changed, this signature should also be modified. (This signature prevents from tampering.)
* Each delimited tokens are base64 URL encoded string (encoded by RFC 4686) as follows.

[](https://tsmatz.files.wordpress.com/2019/12/20191224_format_jwt_zpsfn1qi8wp.jpg)

A claim provider (and technical profile) for JWT issuer generates a token of this format.  
This JWT issuer should be defined by the following claim provider (and technical profile).

<ClaimsProvider>

<DisplayName>Token Issuer</DisplayName>

<TechnicalProfiles>

<TechnicalProfile Id="JwtIssuer">

<DisplayName>JWT Issuer</DisplayName>

<Protocol Name="None" />

<OutputTokenFormat>JWT</OutputTokenFormat>

<Metadata>

<Item Key="client\_id">{service:te}</Item>

<Item Key="issuer\_refresh\_token\_user\_identity\_claim\_type">objectId</Item>

<Item Key="SendTokenResponseBodyWithJsonNumbers">true</Item>

</Metadata>

<CryptographicKeys>

<Key Id="issuer\_secret" StorageReferenceId="B2C\_1A\_TokenSigningKeyContainer" />

<Key Id="issuer\_refresh\_token\_key" StorageReferenceId="B2C\_1A\_TokenEncryptionKeyContainer" />

</CryptographicKeys>

<InputClaims />

<OutputClaims />

</TechnicalProfile>

</TechnicalProfiles>

</ClaimsProvider>

As I mentioned above, the id token and access token (JWT) in Azure AD B2C has a digital signature that are signed by user-provisioned certificate. This certificate is registered as a policy key using portal UI.  
Then you should specify this policy key in <CryptographicKeys /> element with Id="issuer\_secret" as above.

Note : As I mentioned [here](https://tsmatz.wordpress.com/2016/03/08/azure-ad-msa-v2-endpoint-validate-id_token/), the common certificate (in which the public key can be publically shared by everyone) is used in the usual common endpoint in Azure Active Directory (https://login.microsoftonline.com/common/). However, in Azure AD B2C, the common endpoint is not used. In B2C, only tenant-specific endpoint (https://{url}/{tenant}.onmicrosoft.com/) is used, then the user-provisioned certificate is required for token signature.

Furthermore, the returned refresh token for OAuth is encrypted in B2C. (The id token and access token are not encrypted though.) For this encryption, you should also create encryption key (certificate) as a policy key in portal UI and set a reference for this key in <CryptographicKeys /> element as above.

# UserJourneys element

(See [here](https://docs.microsoft.com/en-us/azure/active-directory-b2c/userjourneys) for official reference.)

All building components in previous sections (such as, content definitions, technical profiles, …) are composed as user workflows in this UserJourneys element.  
There might exist multiple flows, such as, sign-in, sign-up, password-change, and others. Each flow is defined as a single UserJourney.

Here I show you a simple user journey example for sign-in. (This is written in SocialAndLocalAccounts/TrustFrameworkBase.xml in startup pack samples.)  
Let’s see this flow step-by-step here.

<UserJourney Id="SignUpOrSignIn">

<OrchestrationSteps>

<OrchestrationStep Order="1" Type="CombinedSignInAndSignUp" ContentDefinitionReferenceId="api.signuporsignin">

<ClaimsProviderSelections>

<ClaimsProviderSelection TargetClaimsExchangeId="FacebookExchange" />

<ClaimsProviderSelection ValidationClaimsExchangeId="LocalAccountSigninEmailExchange" />

</ClaimsProviderSelections>

<ClaimsExchanges>

<ClaimsExchange Id="LocalAccountSigninEmailExchange" TechnicalProfileReferenceId="SelfAsserted-LocalAccountSignin-Email" />

</ClaimsExchanges>

</OrchestrationStep>

<OrchestrationStep Order="2" Type="ClaimsExchange">

<Preconditions>

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

<Value>objectId</Value>

<Action>SkipThisOrchestrationStep</Action>

</Precondition>

</Preconditions>

<ClaimsExchanges>

<ClaimsExchange Id="FacebookExchange" TechnicalProfileReferenceId="Facebook-OAUTH" />

<ClaimsExchange Id="SignUpWithLogonEmailExchange" TechnicalProfileReferenceId="LocalAccountSignUpWithLogonEmail" />

</ClaimsExchanges>

</OrchestrationStep>

<OrchestrationStep Order="3" Type="ClaimsExchange">

<Preconditions>

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

<Value>authenticationSource</Value>

<Value>localAccountAuthentication</Value>

<Action>SkipThisOrchestrationStep</Action>

</Precondition>

</Preconditions>

<ClaimsExchanges>

<ClaimsExchange Id="AADUserReadUsingAlternativeSecurityId" TechnicalProfileReferenceId="AAD-UserReadUsingAlternativeSecurityId-NoError" />

</ClaimsExchanges>

</OrchestrationStep>

<OrchestrationStep Order="4" Type="ClaimsExchange">

<Preconditions>

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

<Value>objectId</Value>

<Action>SkipThisOrchestrationStep</Action>

</Precondition>

</Preconditions>

<ClaimsExchanges>

<ClaimsExchange Id="SelfAsserted-Social" TechnicalProfileReferenceId="SelfAsserted-Social" />

</ClaimsExchanges>

</OrchestrationStep>

<OrchestrationStep Order="5" 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="6" Type="ClaimsExchange">

<Preconditions>

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

<Value>objectId</Value>

<Action>SkipThisOrchestrationStep</Action>

</Precondition>

</Preconditions>

<ClaimsExchanges>

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

</ClaimsExchanges>

</OrchestrationStep>

<OrchestrationStep Order="7" Type="SendClaims" CpimIssuerTechnicalProfileReferenceId="JwtIssuer" />

</OrchestrationSteps>

<ClientDefinition ReferenceId="DefaultWeb" />

</UserJourney>

First, by running step 1, the following sign-in UI will be shown for the user.  
As you can see, this is a combined user-interface with local account’s sign-in and external provider’s selector.

<OrchestrationStep Order="1" Type="CombinedSignInAndSignUp" ContentDefinitionReferenceId="api.signuporsignin">

<ClaimsProviderSelections>

<ClaimsProviderSelection TargetClaimsExchangeId="FacebookExchange" />

<ClaimsProviderSelection ValidationClaimsExchangeId="LocalAccountSigninEmailExchange" />

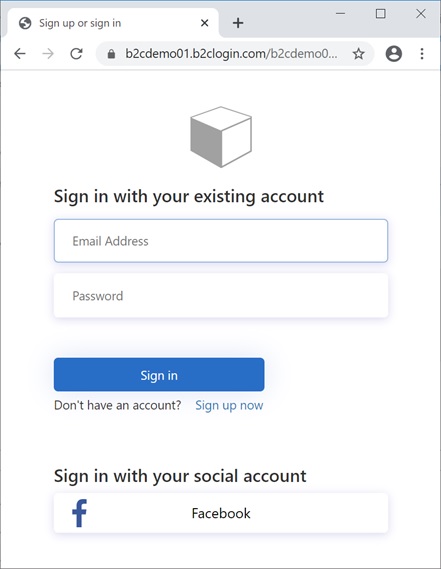
</ClaimsProviderSelections>

<ClaimsExchanges>

<ClaimsExchange Id="LocalAccountSigninEmailExchange" TechnicalProfileReferenceId="SelfAsserted-LocalAccountSignin-Email" />

</ClaimsExchanges>

</OrchestrationStep>

[](https://tsmatz.files.wordpress.com/2020/05/20200513_b2c_login.jpg)

By the following <Precondition /> in step 2, if objectId is already assigned, this step will be skipped.  
What does this mean ?

When the user has successfully logged-in with a local account in step 1, the objectId will be assigned. However, the user has selected “Facebook” provider, objectId is not assigned and this step (step 2) will be evaluated. (That is, when the user has selected “Facebook” provider, this step will be evaluated.)  
When the user has selected “Facebook” provider in step 1, the claim exchange “FacebookExchange” is executed in the next step by TargetClaimsExchangeId="FacebookExchange" in step 1.

As a result, a technical profile named “Facebook-OAUTH” is executed, when the user selected “Facebook” in step 1. (This technical profile will redirect web browser to Facebook login by OAuth manner.)

<OrchestrationStep Order="2" Type="ClaimsExchange">

<Preconditions>

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

<Value>objectId</Value>

<Action>SkipThisOrchestrationStep</Action>

</Precondition>

</Preconditions>

<ClaimsExchanges>

<ClaimsExchange Id="FacebookExchange" TechnicalProfileReferenceId="Facebook-OAUTH" />

<ClaimsExchange Id="SignUpWithLogonEmailExchange" TechnicalProfileReferenceId="LocalAccountSignUpWithLogonEmail" />

</ClaimsExchanges>

</OrchestrationStep>

There exist another case in this step 2.  
What if the user has clicked the link “sign up now” in step 1 ?

By the following definition in a technical profile SelfAsserted-LocalAccountSignin-Email (see TrustFrameworkBase.xml file in starter pack), the claim exchange SignUpWithLogonEmailExchange (in step 2) will be invoked, when the user has clicked sign-up link.  
As a result, a technical profile LocalAccountSignUpWithLogonEmail will be executed, when the user has clicked “sign up now” link in step 1.

<TechnicalProfile Id="SelfAsserted-LocalAccountSignin-Email">

<DisplayName>Local Account Signin</DisplayName>

<Protocol Name="Proprietary" Handler="Web.TPEngine.Providers.SelfAssertedAttributeProvider, Web.TPEngine, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />

<Metadata>

<Item Key="SignUpTarget">SignUpWithLogonEmailExchange</Item>

<Item Key="setting.operatingMode">Email</Item>

<Item Key="setting.forgotPasswordLinkLocation">None</Item>

<Item Key="ContentDefinitionReferenceId">api.selfasserted</Item>

</Metadata>

...

</TechnicalProfile>

Let’s go to step 3.

By the previous technical profiles for both local account authentication (SelfAsserted-LocalAccountSignin-Email) and Facebook authentication (Facebook-OAUTH), authenticationSource will be set to “localAccountAuthentication” (string) and “socialIdpAuthentication” (string) respectively. (See TrustFrameworkBase.xml file in starter pack for details.)

By the following <Precondition /> in step 3, if authenticationSource is equal to “localAccountAuthentication“, this step will be skipped. That is, if Facebook authentication is selected, this step (technical profile AAD-UserReadUsingAlternativeSecurityId-NoError) is executed.

Inside this technical profile AAD-UserReadUsingAlternativeSecurityId-NoError, corresponding objectId is retrieved from B2C directory (i.e, Azure AD) using the user’s sign-in name. Even when the corresponding user object is not extracted in a directory (i.e, there’s no corresponding user in a directory, since the user signs-in for the first time), there will occur no errors in this profile.

<OrchestrationStep Order="3" Type="ClaimsExchange">

<Preconditions>

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

<Value>authenticationSource</Value>

<Value>localAccountAuthentication</Value>

<Action>SkipThisOrchestrationStep</Action>

</Precondition>

</Preconditions>

<ClaimsExchanges>

<ClaimsExchange Id="AADUserReadUsingAlternativeSecurityId" TechnicalProfileReferenceId="AAD-UserReadUsingAlternativeSecurityId-NoError" />

</ClaimsExchanges>

</OrchestrationStep>

In step 4, if there doesn’t exist objectId (i.e, if Facebook authentication is selected for the first time), a technical profile SelfAsserted-Social is executed.

<OrchestrationStep Order="4" Type="ClaimsExchange">

<Preconditions>

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

<Value>objectId</Value>

<Action>SkipThisOrchestrationStep</Action>

</Precondition>

</Preconditions>

<ClaimsExchanges>

<ClaimsExchange Id="SelfAsserted-Social" TechnicalProfileReferenceId="SelfAsserted-Social" />

</ClaimsExchanges>

</OrchestrationStep>

This profile (SelfAsserted-Social) shows UI for self-asserted attributes as follows.  
After the user has set these attributes and continued, this profile will add a user object in a directory with these self-asserted attributes.

[A screenshot of a computer

Description automatically generated](https://tsmatz.files.wordpress.com/2020/05/20200513_test_asserted.jpg)

In step 5, if local account’s authentication is selected, the technical profile AAD-UserReadUsingObjectId is executed.  
This technical profile will retrieve the user information from B2C directory (backend Azure AD).

<OrchestrationStep Order="5" 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>

The step 6 is optional.  
If there’s no attributes to collect in external authentication, the previous SelfAsserted-Social profile will be skipped. As a result, the user object won’t be created in B2C directory.  
In that case, this step 6 will make sure to create a user object in B2C directory.

<OrchestrationStep Order="6" Type="ClaimsExchange">

<Preconditions>

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

<Value>objectId</Value>

<Action>SkipThisOrchestrationStep</Action>

</Precondition>

</Preconditions>

<ClaimsExchanges>

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

</ClaimsExchanges>

</OrchestrationStep>

Finally in step 7, B2C will generate JWT for relying party applications.

<OrchestrationStep Order="7" Type="SendClaims" CpimIssuerTechnicalProfileReferenceId="JwtIssuer" />

In TrustFrameworkBase.xml (starter pack), there exist several other scenarios, such as, local account’s sign-up, password-change, and so on.  
Please refer these user journeys for your learning.

# RelyingParty element

(See [here](https://docs.microsoft.com/en-us/azure/active-directory-b2c/relyingparty) for official reference.)

The RelyingParty element is for the relying party application (such as, web application, mobile application, and others). Note that this RelyingParty element doesn’t register your application itself (which can be registered on “Applications” menu in B2C portal UI), but registers only policies for these applications.

In this element, it specifies which user journey is used for this application. And it provides endpoint url for the relying party application.  
For instance, your application can invoke a sign-in request by using the endpoint : https://{tenant name}.b2clogin.com/tfp/{tenant name}.onmicrosoft.com/{policy name}/oauth2/v2.0/authorize (the policy name is such like “B2C\_1A\_signup\_signin“).

This element also includes a list of claims which are needed for the application.

<RelyingParty>

<DefaultUserJourney ReferenceId="SignUpOrSignIn" />

<TechnicalProfile Id="PolicyProfile">

<DisplayName>PolicyProfile</DisplayName>

<Protocol Name="OpenIdConnect" />

<OutputClaims>

<OutputClaim ClaimTypeReferenceId="displayName" />

<OutputClaim ClaimTypeReferenceId="givenName" />

<OutputClaim ClaimTypeReferenceId="surname" />

<OutputClaim ClaimTypeReferenceId="email" />

<OutputClaim ClaimTypeReferenceId="objectId" PartnerClaimType="sub"/>

<OutputClaim ClaimTypeReferenceId="identityProvider" />

<OutputClaim ClaimTypeReferenceId="tenantId" AlwaysUseDefaultValue="true" DefaultValue="{Policy:TenantObjectId}" />

</OutputClaims>

<SubjectNamingInfo ClaimType="sub" />

</TechnicalProfile>

</RelyingParty>

Note : Above {Policy:TenantObjectId} is called a claim resolver. If you want to set environment-dependent variables (such as, tenant name, tenant object id) in policy, you can use claim resolvers. See “[About claim resolvers in Azure Active Directory B2C custom policies](https://docs.microsoft.com/en-us/azure/active-directory-b2c/claim-resolver-overview)” in official document.

# A Brief Example of Customization

Let’s see a simple customization example.  
In this example, I’ll add a custom one-time passcode for sign-in.

Note (Aug 2022) : Now you can enable one-time passcode (TOTP) as Azure AD B2C MFA option. (See [here](https://techcommunity.microsoft.com/t5/microsoft-entra-azure-ad-blog/totp-based-mfa-for-azure-ad-is-now-generally-available/ba-p/3600448).)  
This example is for the purpose of your understanding.

As you saw above, a lot of settings (such as, all claims, all technical profiles, …) are needed in the real custom policy, then we now start from a starter pack sample in {stater pack}/SocialAndLocalAccounts folder.

First, we add the following new claim type one-time-code-test in <ClaimsSchema />.

<BuildingBlocks>

<ClaimsSchema>

...

<ClaimType Id="one-time-code-test">

<DisplayName>one-time-code-test</DisplayName>

<DataType>string</DataType>

<UserHelpText>This is one time code for test</UserHelpText>

<UserInputType>TextBox</UserInputType>

</ClaimType>

...

In {starter pack}/SocialAndLocalAccounts, a technical profile SelfAsserted-LocalAccountSignin-Email is used for local account’s sign-in.  
Now I’ll add the following 2 custom settings in this existing technical profile.

First, by adding the following one-time-code-test claim for output claims, this claim will be shown in sign-in UI to be collected.

Second, as I mentioned above, login-NonInteractive (ROPC grant flow for Azure AD) is invoked for validating sign-in in this technical profile. In this example, we now add the following custom profile Check-OneTimeCode for sign-in validation. (Later I’ll implement this Check-OneTimeCode.)  
Note that ValidationTechnicalProfiles are order-aware. That means, when I add Check-OneTimeCode after login-NonInteractive as follows, username and password is checked by login-NonInteractive at first. Then if username and password is valid, one-time code is checked by Check-OneTimeCode.

<TechnicalProfile Id="SelfAsserted-LocalAccountSignin-Email">

<DisplayName>Local Account Signin</DisplayName>

<Protocol Name="Proprietary" Handler="Web.TPEngine.Providers.SelfAssertedAttributeProvider, Web.TPEngine, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />

<Metadata>

<Item Key="SignUpTarget">SignUpWithLogonEmailExchange</Item>

<Item Key="setting.operatingMode">Email</Item>

<Item Key="setting.forgotPasswordLinkLocation">None</Item>

<Item Key="ContentDefinitionReferenceId">api.selfasserted</Item>

</Metadata>

<IncludeInSso>false</IncludeInSso>

<InputClaims>

<InputClaim ClaimTypeReferenceId="signInName" />

</InputClaims>

<OutputClaims>

<OutputClaim ClaimTypeReferenceId="signInName" Required="true" />

<OutputClaim ClaimTypeReferenceId="password" Required="true" />

<OutputClaim ClaimTypeReferenceId="objectId" />

<OutputClaim ClaimTypeReferenceId="authenticationSource" />

<OutputClaim ClaimTypeReferenceId="one-time-code-test" /> <!-- added here -->

</OutputClaims>

<ValidationTechnicalProfiles>

<ValidationTechnicalProfile ReferenceId="login-NonInteractive" />

<ValidationTechnicalProfile ReferenceId="Check-OneTimeCode" /> <!-- added here -->

</ValidationTechnicalProfiles>

<UseTechnicalProfileForSessionManagement ReferenceId="SM-AAD" />

</TechnicalProfile>

Now I’ll create a new technical profile Check-OneTimeCode, which checks whether one-time code is valid or not.  
As I mentioned above, we can use a built-in assembly for Restful provider (Web.TPEngine.Providers.RestfulProvider, Web.TPEngine, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null) for this kind of custom logic.  
By the following technical profile, HTTP request with an input claim (one-time-code-test) in HTTP body will be invoked into the endpoint https://xxxxx.azurewebsites.net/b2c-test.php.

<TechnicalProfile Id="Check-OneTimeCode">

<DisplayName>Check OneTimeCode with External Custom REST</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://xxxxx.azurewebsites.net/b2c-test.php</Item>

<Item Key="AuthenticationType">None</Item>

<Item Key="AllowInsecureAuthInProduction">true</Item>

<Item Key="SendClaimsIn">Body</Item> <!-- incoming format -->

<Item Key="ClaimsFormat">Body</Item> <!-- returning format -->

</Metadata>

<InputClaims>

<InputClaim ClaimTypeReferenceId="one-time-code-test" DefaultValue="" />

</InputClaims>

</TechnicalProfile>

This endpoint will receive the following HTTP POST request for B2C validation.  
If the validation is passed, return HTTP OK (200). If not, return HTTP error status.

POST /home/index

Accept: application/json

Content-Type: application/json; charset=utf-8

{"one-time-code-test":"abc"}

For instance, by the following trivial example, only “hellob2c” (fixed string) is passed for one-time code validation. (Sorry, but this example is not “one-time”, but “always” same valid code.)

<?php

$parm = json\_decode(file\_get\_contents('php://input'), true);

if($parm['one-time-code-test'] != 'hellob2c') {

http\_response\_code(403);

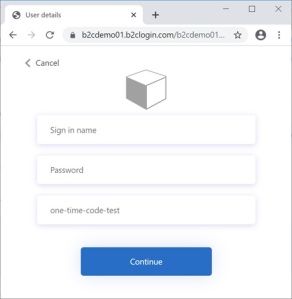
$err = array('message' => 'One time code is invalid');

echo json\_encode($err);

}

Finally, we build a user journey and invoke this from a relying party.  
In our example scenario, we assume that our policy performs the following actions.

1. First, it shows IDP selector as follows.  
   A screenshot of a computer

   Description automatically generated
2. When the user has clicked “Local Account Signin”, the previously customized profile SelfAsserted-LocalAccountSignin-Email is executed.  
   As I mentioned above, this profile will show the following login UI with a custom one-time code textbox. (And it checks whether one-time code is valid or not, when the user pushes “Continue” button.)  
   
3. After the login has succeeded, JWT is returned into the application.

Our completed custom policy (including a user journey and relying party) would be as follows.  
Note that this inherits TrustFrameworkExtensions.xml. (You can add this file into {starter pack}/SocialAndLocalAccounts for a new scenario.)

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<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="xxxxx.onmicrosoft.com"

PolicyId="B2C\_1A\_CustomTest01"

PublicPolicyUri="http://xxxxx.onmicrosoft.com/B2C\_1A\_CustomTest01">

<BasePolicy>

<TenantId>xxxxx.onmicrosoft.com</TenantId>

<PolicyId>B2C\_1A\_TrustFrameworkExtensions</PolicyId>

</BasePolicy>

<BuildingBlocks>

<ClaimsSchema>

<ClaimType Id="one-time-code-test">

<DisplayName>one-time-code-test</DisplayName>

<DataType>string</DataType>

<UserHelpText>This is one time code for test</UserHelpText>

<UserInputType>TextBox</UserInputType>

</ClaimType>

</ClaimsSchema>

</BuildingBlocks>

<ClaimsProviders>

<ClaimsProvider>

<DisplayName>Test01 Provider</DisplayName>

<TechnicalProfiles>

<TechnicalProfile Id="SelfAsserted-LocalAccountSignin-Email">

<DisplayName>Local Account Signin</DisplayName>

<Protocol Name="Proprietary" Handler="Web.TPEngine.Providers.SelfAssertedAttributeProvider, Web.TPEngine, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />

<Metadata>

<Item Key="SignUpTarget">SignUpWithLogonEmailExchange</Item>

<Item Key="setting.operatingMode">Email</Item>

<Item Key="setting.forgotPasswordLinkLocation">None</Item>

<Item Key="ContentDefinitionReferenceId">api.selfasserted</Item>

</Metadata>

<IncludeInSso>false</IncludeInSso>

<InputClaims>

<InputClaim ClaimTypeReferenceId="signInName" />

</InputClaims>

<OutputClaims>

<OutputClaim ClaimTypeReferenceId="signInName" Required="true" />

<OutputClaim ClaimTypeReferenceId="password" Required="true" />

<OutputClaim ClaimTypeReferenceId="objectId" />

<OutputClaim ClaimTypeReferenceId="authenticationSource" />

<OutputClaim ClaimTypeReferenceId="one-time-code-test" />

</OutputClaims>

<ValidationTechnicalProfiles>

<ValidationTechnicalProfile ReferenceId="login-NonInteractive" />

<ValidationTechnicalProfile ReferenceId="Check-OneTimeCode" />

</ValidationTechnicalProfiles>

<UseTechnicalProfileForSessionManagement ReferenceId="SM-AAD" />

</TechnicalProfile>

<TechnicalProfile Id="Check-OneTimeCode">

<DisplayName>Check OneTimeCode with External Custom REST</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://xxxxx.azurewebsites.net/b2c-test.php</Item>

<Item Key="AuthenticationType">None</Item>

<Item Key="AllowInsecureAuthInProduction">true</Item>

<Item Key="SendClaimsIn">Body</Item> <!-- incoming format -->

<Item Key="ClaimsFormat">Body</Item> <!-- returning format -->

</Metadata>

<InputClaims>

<InputClaim ClaimTypeReferenceId="one-time-code-test" DefaultValue="" />

</InputClaims>

</TechnicalProfile>

</TechnicalProfiles>

</ClaimsProvider>

</ClaimsProviders>

<UserJourneys>

<UserJourney Id="TrivialOneTimePassJourney">

<OrchestrationSteps>

<OrchestrationStep Order="1" Type="ClaimsProviderSelection" ContentDefinitionReferenceId="api.idpselections">

<ClaimsProviderSelections>

<ClaimsProviderSelection TargetClaimsExchangeId="FacebookExchange" />

<ClaimsProviderSelection TargetClaimsExchangeId="LocalAccountSigninEmailExchange" />

</ClaimsProviderSelections>

</OrchestrationStep>

<OrchestrationStep Order="2" Type="ClaimsExchange">

<ClaimsExchanges>

<ClaimsExchange Id="FacebookExchange" TechnicalProfileReferenceId="Facebook-OAUTH" />

<ClaimsExchange Id="LocalAccountSigninEmailExchange" TechnicalProfileReferenceId="SelfAsserted-LocalAccountSignin-Email" />

</ClaimsExchanges>

</OrchestrationStep>

<OrchestrationStep Order="3" Type="ClaimsExchange">

<Preconditions>

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

<Value>authenticationSource</Value>

<Value>localAccountAuthentication</Value>

<Action>SkipThisOrchestrationStep</Action>

</Precondition>

</Preconditions>

<ClaimsExchanges>

<ClaimsExchange Id="AADUserReadUsingAlternativeSecurityId" TechnicalProfileReferenceId="AAD-UserReadUsingAlternativeSecurityId-NoError" />

</ClaimsExchanges>

</OrchestrationStep>

<OrchestrationStep Order="4" Type="ClaimsExchange">

<Preconditions>

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

<Value>objectId</Value>

<Action>SkipThisOrchestrationStep</Action>

</Precondition>

</Preconditions>

<ClaimsExchanges>

<ClaimsExchange Id="SelfAsserted-Social" TechnicalProfileReferenceId="SelfAsserted-Social" />

</ClaimsExchanges>

</OrchestrationStep>

<OrchestrationStep Order="5" 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="6" Type="SendClaims" CpimIssuerTechnicalProfileReferenceId="JwtIssuer" />

</OrchestrationSteps>

<ClientDefinition ReferenceId="DefaultWeb" />

</UserJourney>

</UserJourneys>

<RelyingParty>

<DefaultUserJourney ReferenceId="TrivialOneTimePassJourney" />

<TechnicalProfile Id="OneTimePassTest">

<DisplayName>OneTimePassTest</DisplayName>

<Protocol Name="OpenIdConnect" />

<OutputClaims>

<OutputClaim ClaimTypeReferenceId="displayName" />

<OutputClaim ClaimTypeReferenceId="givenName" />

<OutputClaim ClaimTypeReferenceId="surname" />

<OutputClaim ClaimTypeReferenceId="email" />

<OutputClaim ClaimTypeReferenceId="objectId" PartnerClaimType="sub"/>

<OutputClaim ClaimTypeReferenceId="identityProvider" />

<OutputClaim ClaimTypeReferenceId="tenantId" AlwaysUseDefaultValue="true" DefaultValue="{Policy:TenantObjectId}" />

</OutputClaims>

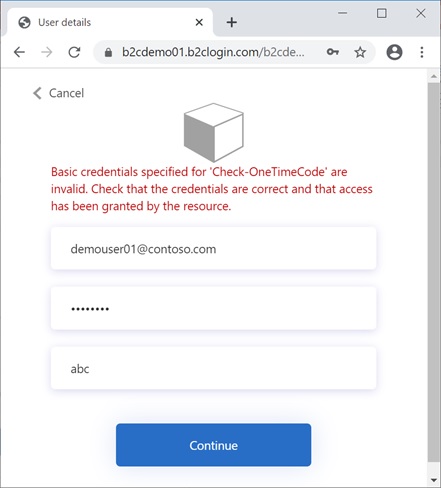
<SubjectNamingInfo ClaimType="sub" />

</TechnicalProfile>

</RelyingParty>

</TrustFrameworkPolicy>

In this trivial example, when the user has filled the wrong one-time code, the login will be rejected as follows.

[](https://tsmatz.files.wordpress.com/2020/05/20200513_test_codeerror.jpg)

Now you have learned an idea of custom policy !  
Let’s go into the following samples and learn more advanced scenarios in this example.

GitHub – Azure Active Directory B2C: Custom User Journeys