# How to: Link a local account with multiple external accounts

This article describes how a local account can be linked with multiple external accounts using a custom policy.

Contents

[How to: Link a local account with multiple external accounts 1](#_Toc506046366)

[Contents 1](#_Toc506046367)

[How to: Configure the base file 1](#_Toc506046368)

[Configure the RestApi-AddUserIdentity technical profile for a REST API claims provider 1](#_Toc506046369)

[Configure the Link user journey 2](#_Toc506046370)

[How to: Implement the AddUserIdentity REST API 2](#_Toc506046371)

## How to: Configure the base file

The base file is configured as follows:

1. The **RestApi-AddUserIdentity** technical profile for a **REST API** claims provider.
2. The **Link** user journey.

### Configure the RestApi-AddUserIdentity technical profile for a REST API claims provider

The following snippet shows how the **RestApi-AddUserIdentity** technical profile for a **REST API** claims provider is configured.

<ClaimsProvider>  
 <DisplayName>REST API</DisplayName>  
 <TechnicalProfiles>  
 <TechnicalProfile Id="RestApi-AddUserIdentity">  
 <DisplayName>Add User Identity REST API</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://bankofzamundafxn.azurewebsites.net/api/AddUserIdentityHttpTrigger?code=...</Item>  
 <Item Key="AuthenticationType">Basic</Item>  
 <Item Key="SendClaimsIn">Body</Item>  
 </Metadata>  
 <CryptographicKeys>  
 <Key Id="BasicAuthenticationUsername" StorageReferenceId="B2C\_1A\_GraphClientId" />  
 <Key Id="BasicAuthenticationPassword" StorageReferenceId="B2C\_1A\_GraphClientSecret" />  
 </CryptographicKeys>  
 <InputClaims>  
 <InputClaim ClaimTypeReferenceId="tenantId" PartnerClaimType="tenantId" />  
 <InputClaim ClaimTypeReferenceId="objectId" PartnerClaimType="tenantUserId" />  
 <InputClaim ClaimTypeReferenceId="identityProvider" PartnerClaimType="issuerId" />  
 <InputClaim ClaimTypeReferenceId="socialUserId" PartnerClaimType="issuerUserId" />  
 </InputClaims>  
 <UseTechnicalProfileForSessionManagement ReferenceId="SSOSession-Noop" />  
 </TechnicalProfile>  
 </TechnicalProfiles>  
</ClaimsProvider>

This technical profile:

* References the **tenantId** claim as an input claim that contains the tenant identifier of the Azure AD B2C tenant which is used by the REST API, with the basic authentication credential, to acquire an access token.
* References the **objectId** claim as an input claim that contains the object identifier of the local account to which an external account is to be linked.
* References the **identityProvider** claim as an input claim that contains the identity provider for the external account, such as “contoso.onmicrosoft.com” for a work account or “google.com” for a social account, which is to be linked to the local account.
* References the **socialUserId** claim as an input claim that contains the user identifier of the external account which is to be linked to the local account.

### Configure the Link user journey

The following snippet shows how the **Link** user journey is configured.

<UserJourney Id="Link">  
 <PreserveOriginalAssertion>false</PreserveOriginalAssertion>  
 <OrchestrationSteps>  
 <OrchestrationStep Order="1" Type="ClaimsExchange">  
 <ClaimsExchanges>  
 <ClaimsExchange Id="LocalAccountLoginExchange" TechnicalProfileReferenceId="LocalAccount-Login" />  
 </ClaimsExchanges>  
 </OrchestrationStep>  
 <OrchestrationStep Order="2" Type="CombinedSignInAndSignUp" ContentDefinitionReferenceId="api.signupsignin">  
 <ClaimsProviderSelections>  
 <ClaimsProviderSelection TargetClaimsExchangeId="AzureActiveDirectoryAccountOpenIdConnectExchange" />  
 <ClaimsProviderSelection TargetClaimsExchangeId="GoogleAccountOAuth2Exchange" />  
 </ClaimsProviderSelections>  
 </OrchestrationStep>  
 <OrchestrationStep Order="3" Type="ClaimsExchange">  
 <ClaimsExchanges>  
 <ClaimsExchange Id="AzureActiveDirectoryAccountOpenIdConnectExchange" TechnicalProfileReferenceId="AzureActiveDirectoryAccount-OpenIdConnect" />  
 <ClaimsExchange Id="GoogleAccountOAuth2Exchange" TechnicalProfileReferenceId="GoogleAccount-OAuth2" />  
 </ClaimsExchanges>  
 </OrchestrationStep>  
 <OrchestrationStep Order="4" Type="ClaimsExchange">  
 <ClaimsExchanges>  
 <ClaimsExchange Id="RestApiAddUserIdentityExchange" TechnicalProfileReferenceId="RestApi-AddUserIdentity" />  
 </ClaimsExchanges>  
 </OrchestrationStep>  
 <OrchestrationStep Order="5" Type="ClaimsExchange">  
 <ClaimsExchanges>  
 <ClaimsExchange Id="AzureActiveDirectoryDirectoryReadUserByObjectIdUserExchange" TechnicalProfileReferenceId="AzureActiveDirectoryStore-ReadUserByObjectId-ThrowIfNotExists" />  
 </ClaimsExchanges>  
 </OrchestrationStep>  
 <OrchestrationStep Order="6" Type="SendClaims" CpimIssuerTechnicalProfileReferenceId="JwtIssuer" />  
 </OrchestrationSteps>  
</UserJourney>

This user journey:

1. Prompts the end user to log in with a local account.
2. Prompts the end user to select to link this local account with either a Microsoft work account or a Google social account.
3. Redirects the end user to the identity provider for the external account.
4. Adds the user identity for the external account to the user object for the local account.
5. Reads the user object for the local account.
6. Issues the security token.

## How to: Implement the AddUserIdentity REST API

The following snippet shows how the **AddUserIdentity** REST API is implemented as an Azure function.

const auth = require("basic-auth");  
const request = require("request");  
  
function acquireTokenForApplication(context, tenantId, clientId, clientSecret, resource, callback) {  
 request.post({  
 url: `https://login.microsoftonline.com/${tenantId}/oauth2/token`,  
 form: {  
 "grant\_type": "client\_credentials",  
 "client\_id": clientId,  
 "client\_secret": clientSecret,  
 "resource": resource  
 }  
 }, (err, response, responseBody) => {  
 if (err) {  
 callback(err);  
 return;  
 }  
  
 if (!isSuccessStatusCode(response.statusCode)) {  
 const errorResult = JSON.parse(responseBody);  
 callback(null, errorResult.error);  
 return;  
 }  
  
 const result = JSON.parse(responseBody);  
 callback(null, result.error, result.access\_token);  
 });  
}  
  
function createBadRequestErrorResponse(context, errorMessage) {  
 return createErrorResponse(context, 400, errorMessage || "Bad Request");  
}  
  
function createErrorResponse(context, statusCode, errorMessage) {  
 return {  
 status: statusCode,  
 body: {  
 "version": "1.0.0",  
 "status": statusCode,  
 "userMessage": errorMessage  
 }  
 };  
}  
  
function createNotFoundErrorResponse(context) {  
 return createErrorResponse(context, 404, "Not Found");  
}  
  
function createUnauthorizedErrorResponse(context) {  
 return createErrorResponse(context, 401, "Unauthorized");  
}  
  
function getUser(context, tenantId, accessToken, userId, callback) {  
 request.get({  
 url: `https://graph.windows.net/${tenantId}/users/${userId}?api-version=1.6`,  
 auth: {  
 bearer: accessToken  
 }  
 }, (err, response, responseBody) => {  
 if (err) {  
 callback(err);  
 return;  
 }  
  
 if (!isSuccessStatusCode(response.statusCode)) {  
 const errorResult = JSON.parse(responseBody);  
 callback(null, errorResult["odata.error"].code);  
 return;  
 }  
  
 const user = JSON.parse(responseBody);  
 callback(null, null, user);  
 });  
}  
  
function isSuccessStatusCode(statusCode) {  
 return statusCode >= 200 && statusCode < 300;  
}  
  
function updateUser(context, tenantId, accessToken, user, callback) {  
 request.patch({  
 url: `https://graph.windows.net/${tenantId}/users/${user.objectId}?api-version=1.6`,  
 auth: {  
 bearer: accessToken  
 },  
 body: {  
 userIdentities: user.userIdentities  
 },  
 json: true  
 }, (err, response, responseBody) => {  
 if (err) {  
 callback(err);  
 return;  
 }  
  
 if (!isSuccessStatusCode(response.statusCode)) {  
 const errorResult = responseBody;  
 callback(null, errorResult["odata.error"].code);  
 return;  
 }  
  
 callback(null, null);  
 });  
}  
  
module.exports = function (context, req) {  
 var client = auth(req);  
  
 if (typeof client === "undefined" || !client) {  
 context.res = createUnauthorizedErrorResponse(context);  
 context.done();  
 return;  
 }  
  
 if (typeof req.body.tenantId === "undefined" || !req.body.tenantId) {  
 context.res = createBadRequestErrorResponse(context, "Tenant is missing.");  
 context.done();  
 return;  
 }  
  
 if (typeof req.body.tenantUserId === "undefined" || !req.body.tenantUserId) {  
 context.res = createBadRequestErrorResponse(context, "Tenant user ID is missing.");  
 context.done();  
 return;  
 }  
  
 if (typeof req.body.issuerId === "undefined" || !req.body.issuerId) {  
 context.res = createBadRequestErrorResponse(context, "Issuer ID is missing.");  
 context.done();  
 return;  
 }  
  
 if (typeof req.body.issuerUserId === "undefined" || !req.body.issuerUserId) {  
 context.res = createBadRequestErrorResponse(context, "Issuer user ID is missing.");  
 context.done();  
 return;  
 }  
  
 acquireTokenForApplication(context, req.body.tenantId, client.name, client.pass, "https://graph.windows.net", (err, error, accessToken) => {  
 if (err) {  
 context.res = createInternalServerErrorErrorResponse(context);  
 context.done();  
 return;  
 }  
  
 if (error) {  
 context.res = createInternalServerErrorErrorResponse(context);  
 context.done();  
 return;  
 }  
  
 getUser(context, req.body.tenantId, accessToken, req.body.tenantUserId, (err, error, user) => {  
 if (err) {  
 context.res = createInternalServerErrorErrorResponse(context);  
 context.done();  
 return;  
 }  
  
 if (error) {  
 context.res = createInternalServerErrorErrorResponse(context);  
 context.done();  
 return;  
 }  
  
 if (!user) {  
 context.res = createNotFoundErrorResponse(context);  
 context.done();  
 return;  
 }  
  
 if (user.userIdentities === null) {  
 user.userIdentities = [];  
 }  
  
 user.userIdentities.push({  
 issuer: req.body.issuerId,  
 issuerUserId: new Buffer(req.body.issuerUserId).toString("base64")  
 });  
  
 updateUser(context, req.body.tenantId, accessToken, user, (err, error) => {  
 if (err) {  
 context.res = createInternalServerErrorErrorResponse(context);  
 context.done();  
 return;  
 }  
  
 if (error) {  
 context.res = createInternalServerErrorErrorResponse(context);  
 context.done();  
 return;  
 }  
  
 context.res = {};  
 context.done();  
 });  
 });  
 });  
};

This REST API adds the user identity for the external account to the user object for the local account, by appending the issuer and issuerUserId properties for this user identity to the end of the userIdentities property for this user object, where the issuerUserId property for the user identity is written as a Base64-encoded string.

{  
 "issuer": "google.com",  
 "issuerUserId": "MTEyNzI3MjIzNzA4MDE0OTYyOTA0"  
}