AuthProviderPluginClass Class

Contains methods to create a custom OAuth-based authentication provider plug-in for single sign-on in to Salesforce. Use this class to create a custom authentication provider plug-in if you can't use one of the authentication providers that Salesforce provides.

Namespace

Auth

Usage

To create a custom authentication provider for single sign-on, create a class that extends `Auth.AuthProviderPluginClass`. This class allows you to store the custom configuration for your authentication provider and handle authentication protocols when users log in to Salesforce with their login credentials for an external service provider. In Salesforce, the class that implements this interface appears in the `Provider Type` drop-down list in Auth. Providers in Setup. Make sure that the user you specify to run the class has "Customize Application" and "Manage Auth. Providers" permissions.

As of API version 39.0, use the abstract class `AuthProviderPluginClass` to create a custom external authentication provider. This class replaces the `AuthProviderPlugin` interface. If you've already implemented a custom authentication provider plug-in using the interface, it still works. However, use `AuthProviderPluginClass` to extend your plug-in. If you haven't created an interface, create a custom authentication provider plug-in by extending this abstract class. For more information, see AuthProviderPluginClass Code Example.

- AuthProviderPluginClass Methods
- AuthProviderPluginClass Code Example

AuthProviderPluginClass Methods

The `AuthProviderPluginClass` methods don't support DML options.

- getCustomMetadataType()
  Returns the custom metadata type API name for a custom OAuth-based authentication provider for single sign-on to Salesforce.
- getUserInfo(authProviderConfiguration, response)
  Returns information from the custom authentication provider about the current user. This information is used by the registration handler and in other authentication provider flows.
- handleCallback(authProviderConfiguration, callbackState)
  Uses the authentication provider's supported authentication protocol to return an OAuth access token, OAuth secret or refresh token, and the state passed in when the request for the current user was initiated.
- initiate(authProviderConfiguration, stateToPropagate)
  Returns the URL where the user is redirected for authentication.
- refresh(authProviderConfiguration, refreshToken)
  Returns a new access token, which is used to update an expired access token.

getCustomMetadataType()

```
public String getCustomMetadataType()
```

Return Value

Type: String

The custom metadata type API name for the authentication provider.

Usage

The `getCustomMetadataType()` method returns only custom metadata type names. It does not return custom metadata record names. As of API version 39.0, use this method when extending `Auth.AuthProviderPluginClass` to create a custom external authentication provider.

getUserInfo(authProviderConfiguration, response)

Returns information from the custom authentication provider about the current user. This information is used by the registration handler and in other authentication provider flows.

Signature

```
public Auth.UserData getUserInfo(Map<String,String> authProviderConfiguration,
Auth.AuthProviderTokenResponse response)
```

Parameters

authProviderConfiguration

Type: Map<String,String>

The configuration for the custom authentication provider. When you create a custom metadata type in Salesforce, the configuration populates it with the custom metadata type default values. Or you can set the configuration with values that you enter when you create the custom provider in Auth. Providers in Setup.

response

Type: Auth.AuthProviderTokenResponse

The OAuth access token, OAuth secret or refresh token, and state provided by the authentication provider to authenticate the current user.

Return Value

Type: Auth.UserData

Creates a new instance of the `Auth.UserData` class.

Usage

As of API version 39.0, use this method when extending `Auth.AuthProviderPluginClass` to create a custom authentication provider.

> **Note**
>
> You might choose to get user information in the response from the `handleCallback` method or by another method. However, you must still call `getUserInfo` in the custom authentication handler to avoid getting errors about mixing objects. For example, if you don't call `getUserInfo`, and then try to insert a contact in the `Auth.RegistrationHandler.createUser` method, you get the error, "You cannot mix EntityObjects with different UddInfos within one transaction."
>
> To avoid this error, call `getUserInfo` with dummy user information as follows.

```
    Http http = new Http();
    HTTPResponse res = http.send(req);
```

handleCallback(authProviderConfiguration, callbackState)

Uses the authentication provider's supported authentication protocol to return an OAuth access token, OAuth secret or refresh token, and the state passed in when the request for the current user was initiated.

Signature

```
public Auth.AuthProviderTokenResponse handleCallback(Map<String,String>
authProviderConfiguration, Auth.AuthProviderCallbackState callbackState)
```

Parameters

authProviderConfiguration

Type: Map<StringString>

The configuration for the custom authentication provider. When you create a custom metadata type in Salesforce, the configuration populates with the custom metadata type default values. Or you can set the configuration with values you enter when you create the custom provider in Auth. Providers in Setup.

callbackState

Type: Auth.AuthProviderCallbackState

The class that contains the HTTP headers, body, and queryParams of the authentication request.

Return Value

Type: Auth.AuthProviderTokenResponse

Creates an instance of the `AuthProviderTokenResponse` class.

Usage

As of API version 39.0, use this method when extending `Auth.AuthProviderPluginClass` to create a custom authentication provider.

initiate(authProviderConfiguration, stateToPropagate)

Returns the URL where the user is redirected for authentication.

Signature

```
public System.PageReference initiate(Map<String,String> authProviderConfiguration, String
stateToPropagate)
```

Parameters

authProviderConfiguration

Type: Map<StringString>

The configuration for the custom authentication provider. When you create a custom metadata type in Salesforce, the configuration populates with the custom metadata type default values. Or you can set the configuration with values you enter when you create the custom provider in Auth. Providers in Setup.

stateToPropagate

Type: String

The URL of the page where the user is redirected for authentication.

Usage

As of API version 39.0, use this method when extending `Auth.AuthProviderPluginClass` to create a custom authentication provider.

refresh(authProviderConfiguration, refreshToken)

Returns a new access token, which is used to update an expired access token.

Signature

```
public Auth.OAuthRefreshResult refresh(Map<String,String> authProviderConfiguration, String refreshToken)
```

Parameters

authProviderConfiguration

Type: Map<String,String>

The configuration for the custom authentication provider. When you create a custom metadata type in Salesforce, the configuration populates with the custom metadata type default values. Or you can set the configuration with values you enter when you create the custom provider in Auth. Providers in Setup.

refreshToken

Type: String

The refresh token for the user who is logged in.

Return Value

Type: Auth.OAuthRefreshResult

Returns the new access token, or an error message if an error occurs.

Usage

A successful request returns a `Auth.OAuthRefreshResult` with the access token and refresh token in the response. If you receive an error, make sure that you set the error string to the error message. A `NULL` error string indicates no error.

The refresh method works only with named credentials; it doesn't respect the standard OAuth refresh flow. The refresh method with named credentials works only if the earlier request returns a 401.

AuthProviderPluginClass Code Example

The following example demonstrates how to implement a custom Auth. provider plug-in using the abstract class, `Auth.AuthProviderPluginClass`.

```
global class Concur extends Auth.AuthProviderPluginClass {

            // Use this URL for the endpoint that the
            // authentication provider calls back to for configuration.
            public String redirectUrl;
            private String key;
            private String secret;

            // Application redirection to the Concur website for
```

```
// Api name for the custom metadata type created for this auth provider.
private String customMetadataTypeApiName;

// Api URL to access the user in Concur
private String userAPIUrl;

// Version of the user api URL to access data from Concur
private String userAPIVersionUrl;

global String getCustomMetadataType() {
    return customMetadataTypeApiName;
}

global PageReference initiate(Map<string,string>
  authProviderConfiguration, String stateToPropagate)
  {
      authUrl = authProviderConfiguration.get('Auth_Url__c');
      key = authProviderConfiguration.get('Key__c');

      // Here the developer can build up a request of some sort.
      // Ultimately, they return a URL where we will redirect the user.
      String url = authUrl + '?client_id='+ key +'&scope=USER,EXPRPT,LIST&redire
      return new PageReference(url);
  }

  global Auth.AuthProviderTokenResponse handleCallback(Map<string,string>
  authProviderConfiguration, Auth.AuthProviderCallbackState state )
  {
      // Here, the developer will get the callback with actual protocol.
      // Their responsibility is to return a new object called
      // AuthProviderTokenResponse.
      // This will contain an optional accessToken and refreshToken
      key = authProviderConfiguration.get('Key__c');
      secret = authProviderConfiguration.get('Secret__c');
      accessTokenUrl = authProviderConfiguration.get('Access_Token_Url__c');

      Map<String,String> queryParams = state.queryParameters;
      String code = queryParams.get('code');
      String sfdcState = queryParams.get('state');

      HttpRequest req = new HttpRequest();
      String url = accessTokenUrl+'?code=' + code + '&client_id=' + key +
      '&client_secret=' + secret;
      req.setEndpoint(url);
      req.setHeader('Content-Type','application/xml');
      req.setMethod('GET');

      Http http = new Http();
      HTTPResponse res = http.send(req);
      String responseBody = res.getBody();
      String token = getTokenValueFromResponse(responseBody, 'Token', null);

      return new Auth.AuthProviderTokenResponse('Concur', token,
      'refreshToken', sfdcState);
  }

  global Auth.UserData getUserInfo(Map<string,string>
  authProviderConfiguration,
  Auth.AuthProviderTokenResponse response)
  {
      //Here the developer is responsible for constructing an
      //Auth.UserData object
      String token = response.oauthToken;
      HttpRequest req = new HttpRequest();
      userAPIUrl = authProviderConfiguration.get('API_User_Url__c');
      userAPIVersionUrl = authProviderConfiguration.get
      ('API_User_Version_Url__c');
      req.setHeader('Authorization', 'OAuth ' + token);
      req.setEndpoint(userAPIUrl);
      req.setHeader('Content-Type','application/xml');
      req.setMethod('GET');
```

```
                'FirstName', userAPIVersionUrl);
                String lname = getTokenValueFromResponse(responseBody,
                'LastName', userAPIVersionUrl);
                String flname = fname + ' ' + lname;
                String uname = getTokenValueFromResponse(responseBody,
                'EmailAddress', userAPIVersionUrl);
                String locale = getTokenValueFromResponse(responseBody,
                'LocaleName', userAPIVersionUrl);
                Map<String,String> provMap = new Map<String,String>();
                provMap.put('what1', 'noidea1');
                provMap.put('what2', 'noidea2');
                return new Auth.UserData(id, fname, lname, flname,
                uname, 'what', locale, null, 'Concur', null, provMap);
            }

            private String getTokenValueFromResponse(String response,
            String token, String ns)
            {
                Dom.Document docx = new Dom.Document();
                docx.load(response);
                String ret = null;

                dom.XmlNode xroot = docx.getrootelement() ;
                if(xroot != null){ ret = xroot.getChildElement(token, ns).getText();
                }
            return ret;
            }

        }
```

Sample Test Classes

The following example contains test classes for the Concur class.

```
@IsTest
            public class ConcurTestClass {

                private static final String OAUTH_TOKEN = 'testToken';
                private static final String STATE = 'mocktestState';
                private static final String REFRESH_TOKEN = 'refreshToken';
                private static final String LOGIN_ID = 'testLoginId';
                private static final String USERNAME = 'testUsername';
                private static final String FIRST_NAME = 'testFirstName';
                private static final String LAST_NAME = 'testLastName';
                private static final String EMAIL_ADDRESS = 'testEmailAddress';
                private static final String LOCALE_NAME = 'testLocalName';
                private static final String FULL_NAME = FIRST_NAME + ' ' + LAST_NAME;
                private static final String PROVIDER = 'Concur';
                private static final String REDIRECT_URL =
                'http://localhost/services/authcallback/orgId/Concur';
                private static final String KEY = 'testKey';
                private static final String SECRET = 'testSecret';
                private static final String STATE_TO_PROPOGATE = 'testState';
                private static final String ACCESS_TOKEN_URL =
                'http://www.dummyhost.com/accessTokenUri';
                private static final String API_USER_VERSION_URL =
                'http://www.dummyhost.com/user/20/1';
                private static final String AUTH_URL =
                'http://www.dummy.com/authurl';
                private static final String API_USER_URL =
                'www.concursolutions.com/user/api';

            // In the real world scenario, the key and value would be read
            // from the (custom fields in) custom metadata type record.
            private static Map<String,String> setupAuthProviderConfig ()
            {
                Map<String,String> authProviderConfiguration = new Map<String,String>();
```

```
        API_USER_VERSION_URL);
        authProviderConfiguration.put('Redirect_Url__c',REDIRECT_URL);
        return authProviderConfiguration;

    }

    static testMethod void testInitiateMethod()
    {
        String stateToPropogate = 'mocktestState';
        Map<String,String> authProviderConfiguration = setupAuthProviderConfig();
        Concur concurCls = new Concur();
        concurCls.redirectUrl = authProviderConfiguration.get('Redirect_Url__c');
    PageReference expectedUrl = new PageReference(authProviderConfiguration.get('A
    authProviderConfiguration.get('Key__c') +'&scope=USER,EXPRPT,LIST&redirect_uri
    authProviderConfiguration.get('Redirect_Url__c') + '&state=' +
    STATE_TO_PROPOGATE);
        PageReference actualUrl = concurCls.initiate(authProviderConfiguration, ST
        System.assertEquals(expectedUrl.getUrl(), actualUrl.getUrl());
    }

    static testMethod void testHandleCallback()
    {
        Map<String,String> authProviderConfiguration =
        setupAuthProviderConfig();
        Concur concurCls = new Concur();
        concurCls.redirectUrl = authProviderConfiguration.get
        ('Redirect_Url_c');

        Test.setMock(HttpCalloutMock.class, new
        ConcurMockHttpResponseGenerator());

        Map<String,String> queryParams = new Map<String,String>();
        queryParams.put('code','code');
        queryParams.put('state',authProviderConfiguration.get('State_c'));
        Auth.AuthProviderCallbackState cbState =
        new Auth.AuthProviderCallbackState(null,null,queryParams);
        Auth.AuthProviderTokenResponse actualAuthProvResponse =
        concurCls.handleCallback(authProviderConfiguration, cbState);
        Auth.AuthProviderTokenResponse expectedAuthProvResponse =
        new Auth.AuthProviderTokenResponse(
        'Concur', OAUTH_TOKEN, REFRESH_TOKEN, null);

        System.assertEquals(expectedAuthProvResponse.provider,
        actualAuthProvResponse.provider);
        System.assertEquals(expectedAuthProvResponse.oauthToken,
        actualAuthProvResponse.oauthToken);
        System.assertEquals(expectedAuthProvResponse.oauthSecretOrRefreshToken,
        actualAuthProvResponse.oauthSecretOrRefreshToken);
        System.assertEquals(expectedAuthProvResponse.state,
        actualAuthProvResponse.state);

    }

    static testMethod void testGetUserInfo()
    {
        Map<String,String> authProviderConfiguration =
        setupAuthProviderConfig();
        Concur concurCls = new Concur();

        Test.setMock(HttpCalloutMock.class, new
        ConcurMockHttpResponseGenerator());

        Auth.AuthProviderTokenResponse response =
        new Auth.AuthProviderTokenResponse(
        PROVIDER, OAUTH_TOKEN ,'sampleOauthSecret', STATE);
        Auth.UserData actualUserData = concurCls.getUserInfo(
        authProviderConfiguration, response) ;

        Map<String,String> provMap = new Map<String,String>();
        provMap.put('key1', 'value1');
        provMap.put('key2', 'value2');
```

```
            actualUserData.firstName);
            System.assertEquals(expectedUserData.lastName,
            actualUserData.lastName);
            System.assertEquals(expectedUserData.fullName,
            actualUserData.fullName);
            System.assertEquals(expectedUserData.email,
            actualUserData.email);
            System.assertEquals(expectedUserData.username,
            actualUserData.username);
            System.assertEquals(expectedUserData.locale,
            actualUserData.locale);
            System.assertEquals(expectedUserData.provider,
            actualUserData.provider);
            System.assertEquals(expectedUserData.siteLoginUrl,
            actualUserData.siteLoginUrl);
        }


        // Implement a mock http response generator for Concur.
        public class ConcurMockHttpResponseGenerator implements HttpCalloutMock
        {
            public HTTPResponse respond(HTTPRequest req)
            {
                String namespace = API_USER_VERSION_URL;
                String prefix = 'mockPrefix';

                Dom.Document doc = new Dom.Document();
                Dom.XmlNode xmlNode = doc.createRootElement(
                'mockRootNodeName', namespace, prefix);
                xmlNode.addChildElement('LoginId', namespace, prefix)
                .addTextNode(LOGIN_ID);
                xmlNode.addChildElement('FirstName', namespace, prefix)
                .addTextNode(FIRST_NAME);
                xmlNode.addChildElement('LastName', namespace, prefix)
                .addTextNode(LAST_NAME);
                xmlNode.addChildElement('EmailAddress', namespace, prefix)
                .addTextNode(EMAIL_ADDRESS);
                xmlNode.addChildElement('LocaleName', namespace, prefix)
                .addTextNode(LOCALE_NAME);
                xmlNode.addChildElement('Token', null, null)
                .addTextNode(OAUTH_TOKEN);
                System.debug(doc.toXmlString());
                // Create a fake response
                HttpResponse res = new HttpResponse();
                res.setHeader('Content-Type', 'application/xml');
                res.setBody(doc.toXmlString());
                res.setStatusCode(200);
                return res;
            }

        }
    }
```

DID THIS ARTICLE SOLVE YOUR ISSUE?
Let us know so we can improve!

Share your feedback