RegistrationHandler Interface

Salesforce provides the ability to use an authentication provider, such as Facebook© or Janrain©, for single sign-on into Salesforce.

Namespace

Auth

Usage

To set up single sign-on, you must create a class that implements `Auth.RegistrationHandler`. Classes implementing the `Auth.RegistrationHandler` interface are specified as the `Registration Handler` in authentication provider definitions, and enable single sign-on into Salesforce portals and organizations from third-party services such as Facebook. Using information from the authentication providers, your class must perform the logic of creating and updating user data as appropriate, including any associated account and contact records.

> **Note**
>
> During the user update process, you can use the confirmUser() method to ensure that users are correctly mapped between Salesforce and the third party. For more information, see the ConfirmUserRegistrationHandler Interface.

- **RegistrationHandler Methods**
- **Storing User Information and Getting Access Tokens**
- **Auth.RegistrationHandler Example Implementation**
- **Auth.RegistrationHandler Error Example**
  This example implements the `Auth.RegistrationHandler` interface and shows how to use a custom exception to display an error message in the URL of the page. If you don't use a custom exception, the error code and description appear in the URL and the error description appears on the page.

RegistrationHandler Methods

The following are methods for `RegistrationHandler`.

- **createUser(portalId, userData)**
  Returns a User object using the specified portal ID and user information from the third party, such as the username and email address. The User object corresponds to the third party's user information. It can be a new user that hasn't been inserted in your org's database, or it can represent an existing user record in the database. If it's a new User object, Salesforce inserts a user record for you.
- **updateUser(userId, portalId, userData)**
  Updates the specified user's information. This method is called if the user has logged in before with the authentication provider and then logs in again.

createUser(portalId, userData)

Signature

```
public User createUser(ID portalId, Auth.UserData userData)
```

Parameters

portalId

Type: ID

userData

Type: Auth.UserData

Return Value

Type: User

Usage

The *portalID* value can be null or an empty key if there's no portal configured with this provider.

updateUser(userId, portalId, userData)

Updates the specified user's information. This method is called if the user has logged in before with the authentication provider and then logs in again.

Signature

```
public Void updateUser(ID userId, ID portalId, Auth.UserData userData)
```

Parameters

userId

Type: ID

portalId

Type: ID

userData

Type: Auth.UserData

Return Value

Type: Void

Usage

The *portalID* value can be null or an empty key if there's no portal configured with this provider.

Storing User Information and Getting Access Tokens

The `Auth.UserData` class is used to store user information for `Auth.RegistrationHandler`. The third-party identity provider can send back a large collection of data about the user, including their username, email address, locale, and more. The Salesforce authentication provider framework converts this data into a common format with the `Auth.UserData` class and then sendsit to the registration handler.

Note

If the registration handler wants to use the rest of the data, the `Auth.UserData` class has an `attributeMap` variable. The attribute map is a map of strings (`Map<String, String>`) for the raw values of all the data from the third party. Because the map is `<String, String>`, values that the third party returns that aren't strings (like an array of URLs or a map) are converted into an appropriate string representation. The map includes everything returned by the third-party authentication provider, including the items automatically converted into the common format.

To learn about `Auth.UserData` properties, see Auth.UserData Class.

> **Note**
>
> You can only perform DML operations on additional sObjects in the same transaction with User objects under certain circumstances. For more information, see sObjects That Cannot Be Used Together in DML Operations.

For all authentication providers except Janrain, after a user is authenticated using a provider, the access token associated with that provider for this user can be obtained in Apex using the `Auth.AuthToken` Apex class. `Auth.AuthToken` provides two methods to retrieve access tokens. One is `getAccessToken`, which obtains a single access token. Use this method if the user ID is mapped to a single third-party user. If the user ID is mapped to multiple third-party users, use `getAccessTokenMap`, which returns a map of access tokens for each third-party user. For more information about authentication providers, see Authentication Providers in *Salesforce Help*.

When using Janrain as an authentication provider, you must use the Janrain `accessCredentials` dictionary values to retrieve the access token or its equivalent. Only some providers supported by Janrain provide an access token, while other providers use other fields. The Janrain `accessCredentials` fields are returned in the `attributeMap` variable of the `Auth.UserData` class. See the Janrain `auth_info` documentation for more information on `accessCredentials`.

> **Note**
>
> Not all Janrain account types return `accessCredentials`. Sometimes you must change your account type to receive the information.

To learn about the `Auth.AuthToken` methods, see Auth.AuthToken Class.

User Information in the ID Token and User Info Response

Some identity providers send additional user information in an ID token or in a user info response. To extract user information from these responses, there are some extra steps.

An ID token is formatted as a JWT and includes information about the authenticated user. If the identity provider sends an ID token, Salesforce stores the full encoded JWT in the `idToken` property. Salesforce also stores the decoded JWT payload of the ID token in the `idTokenJSONString` property.

Salesforce doesn't validate the ID token. To validate it, use methods in the `Auth.JWTUtil` class and pass in the encoded JWT stored in the `idToken` property. The methods in the `Auth.JWTUtil` class all return an instance of the `Auth.JWT` object.

Once you validate the JWT, you can use methods in the `Auth.JWT` class to access specific claims. For example, the Apex code in this snippet validates the ID token using a public keys endpoint from the identity provider and then retrieves the value of an `email` claim stored in the token.

```
Auth.JWT jwt = Auth.JWTUtil.validateJWTWithKeysEndpoint(userdata.idToken, keysEndpoint, tr

// Retrieve email claim from id token
```

JSON string and then write code to retrieve the claim you want. To deserialize the `idTokenJSONString`, use the `JSON.deserialize (jsonString, apexType)` method in the `System.JSON` class.

The user info response, if returned by the identity provider, is also a JSON object that has been serialized into a string. The user info response is stored in the `userInfoJSONString` property. You can use the `JSON.deserialize (jsonString, apexType)` method to deserialize the user info response so that you can retrieve specific information.

This example snippet creates a custom class to store the user info response. It then deserializes the user info response in the `userInfoJSONString` into this custom class structure.

```
public class UserInfoResponse {
    public String preferred_username;
    public String email;
    public Boolean email_verified;
    public String given_name;
    public String family_name;
    public String locale;
}

UserInfoResponse userInfo = (UserInfoResponse)System.JSON.deserialize(userData.userInfoJSO
System.debug(userInfo.email);
```

Auth.RegistrationHandler Example Implementation

This example implements the `Auth.RegistrationHandler` interface that creates as well as updates a standard user based on data provided by the authentication provider. Error checking has been omitted to keep the example simple.

```
global class StandardUserRegistrationHandler implements Auth.RegistrationHandler{
    global User createUser(Id portalId, Auth.UserData data) {
        User u = new User();
        Profile p = [SELECT Id FROM profile WHERE name='Standard User'];
        u.Username = data.username + '@salesforce.com';
        u.Email = data.email;
        u.LastName = data.lastName;
        u.FirstName = data.firstName;
        String alias = data.username;
        if(alias.length() > 8) {
            alias = alias.substring(0, 8);
        }
        u.Alias = alias;
        u.LanguageLocaleKey = data.attributeMap.get('language');
        u.LocaleSidKey = data.locale;
        u.EmailEncodingKey = 'UTF-8';
        u.TimeZoneSidKey = 'America/Los_Angeles';
        u.ProfileId = p.Id;
        return u;
    }

    global void updateUser(Id userId, Id portalId, Auth.UserData data) {
        User u = new User(id=userId);
        u.Username = data.username + '@salesforce.com';
        u.Email = data.email;
        u.LastName = data.lastName;
        u.FirstName = data.firstName;
        String alias = data.username;
        if(alias.length() > 8) {
            alias = alias.substring(0, 8);
        }
```

The following example tests the above code.

```
@isTest
private class StandardUserRegistrationHandlerTest {
static testMethod void testCreateAndUpdateUser() {
    StandardUserRegistrationHandler handler = new StandardUserRegistrationHandler();
    Auth.UserData sampleData = new Auth.UserData('testId', 'testFirst', 'testLast',
        'testFirst testLast', 'testuser@example.org', null, 'testuserlong', 'en_US', 'face
        null, new Map<String, String>{'language' => 'en_US'});
    User u = handler.createUser(null, sampleData);
    System.assertEquals('testuserlong@salesforce.com', u.username);
    System.assertEquals('testuser@example.org', u.email);
    System.assertEquals('testLast', u.lastName);
    System.assertEquals('testFirst', u.firstName);
    System.assertEquals('testuser', u.alias);
    insert(u);
    String uid = u.id;

    sampleData = new Auth.UserData('testNewId', 'testNewFirst', 'testNewLast',
        'testNewFirst testNewLast', 'testnewuser@example.org', null, 'testnewuserlong', 'e
        null, new Map<String, String>{});
    handler.updateUser(uid, null, sampleData);

    User updatedUser = [SELECT username, email, firstName, lastName, alias FROM user WHERE
    System.assertEquals('testnewuserlong@salesforce.com', updatedUser.username);
    System.assertEquals('testnewuser@example.org', updatedUser.email);
    System.assertEquals('testNewLast', updatedUser.lastName);
    System.assertEquals('testNewFirst', updatedUser.firstName);
    System.assertEquals('testnewu', updatedUser.alias);
}
}
```

Auth.RegistrationHandler Error Example

This example implements the `Auth.RegistrationHandler` interface and shows how to use a custom exception to display an error message in the URL of the page. If you don't use a custom exception, the error code and description appear in the URL and the error description appears on the page.

To limit this example to the custom exception, some code was omitted.

```
global class RegHandler implements Auth.RegistrationHandler {

    class RegHandlerException extends Exception {}

        global User createUser(Id portalId, Auth.UserData data){
            List<Profile> profiles = [SELECT Id, Name, UserType FROM Profile WHERE Name =
            Profile profile = profiles.isEmpty() ? null : profiles[0];
            if(profile==null)
               throw new RegHandlerException('Cannot find the profile. For help, contact y
...
        }

        global void updateUser(Id userId, Id portalId, Auth.UserData data){
            User u = new User(id=userId);
            u.lastName = data.lastName;
            u.firstName = data.firstName;
            update(u);
        }
}
```

DEVELOPER CENTERS

- Heroku
- MuleSoft
- Tableau
- Commerce Cloud
- Lightning Design System
- Einstein
- Quip

POPULAR RESOURCES

- Documentation
- Component Library
- APIs
- Trailhead
- Sample Apps
- Podcasts
- AppExchange

COMMUNITY

- Trailblazer Community
- Events and Calendar
- Partner Community
- Blog
- Salesforce Admins
- Salesforce Architects