

Developing Identity Services (IDS)

Introduction: Identity Services (IDS)	2
Authentication Services	2
Profile Services	3
Other Methods for User Control	4
Authentication Service Internals	5
Plumtree.Remote.Auth	5
Synchronization.....	5
Authentication.....	6
Implementing an Authentication Service.....	6
Implementing the ISyncProvider Interface	7
Implementing the IGroup Interface	11
Implementing the IAuthProvider Interface.....	13
Deploying an Authentication Service	15
Deploying a Java Authentication Service	15
Deploying a .NET Authentication Service	16
Configuring Portal Objects	17
Authentication Web Service Editor.....	17
Authentication Source Editor	18
Profile Service Internals	19
Plumtree.Remote.Profile	19
Profile Synchronization	19
Implementing a Profile Service	20
Implementing the IProfileProvider Interface.....	20
Implementing the IUser Interface	23
Deploying a Profile Service	25
Deploying a Java Profile Service.....	25
Deploying a .NET Profile Service	26
Configuring Portal Objects	27
Profile Web Service Editor.....	27
Global Object Property Map.....	28
User Profile Manager	28

Introduction: Identity Services (IDS)

Identity Services allow you to integrate established repositories of user information into your portal. Users, groups, and group membership configuration can be imported into the portal. Users logging into the portal can be authenticated against the existing system of record. Information about users can be imported from any number of external sources and mapped to portal properties, which can then be made available to the portal or other services.

- **Authentication services** are used to import users into the portal and authenticate them against a back-end system.
- **Profile services** are used to import information about existing portal users from external systems and map that information to portal properties.
- **Other methods** of controlling users in your portal implementation, such as adding functionality to the user creation process, are also available.

Authentication Services

Authentication services are comprised of two parts: **synchronization** and **authentication**. Together, these components import new users and allow them to authenticate against the external system of record.

Synchronization

The synchronization component of an authentication service imports users from an external system into the portal so that the users can be categorized in the portal's group hierarchy. The synchronization process is handled by the portal Automation Server, as scheduled in the Job associated with the Authentication Source object in the portal.

Synchronization does *not* store users' passwords in the portal database. Authentication is handled by the authentication component and the system of record.

Note: Creating an Authentication Source object with a synchronization component creates an associated option in the Authentication Source drop-down list on the portal login page. The name that appears in the drop-down list is the **Description** of the Authentication Source object. Enter a description that all users will recognize.

Authentication

The authentication component of an authentication service handles real-time authentication of portal users against an external system. Since the portal cannot change an externally managed password, a user's login must be compared against the system of record. The remote authentication service must maintain state and handle the communication between the portal and the

back-end system. The user name and password can be captured in the session at login to be used later for basic authentication.

Development

The following pages provide detailed instructions on developing custom authentication services:

- **Authentication Service Internals:** A description of the interfaces that must be implemented when creating an authentication service, and how the interfaces will be called by the portal.
- **Implementing an Authentication Service:** Step by step instructions on implementing the required interfaces, with example code.
- **Deploying an Authentication Service:** How to deploy the authentication service to a Java or .NET application server.
- **Configuring an Authentication Service:** How to configure the authentication service in the portal.

Profile Services

Profile services are used to import information about existing portal users from external systems. This information is mapped to portal properties and made available to other services.

Synchronization

The purpose of a profile service is to import information about portal users from an external system into the portal so that the information can be used by the portal and other services. The first step is to synchronize the user information in the external system with existing users in the portal; this is the process that must be handled by the remote service. As with authentication services, the synchronization process is handled by the portal Automation Server, as scheduled in the Job associated with the Profile Source object in the portal.

Property Mapping: User Information

The profile information imported by the profile service must be associated with portal properties so that it can be accessed by portal objects and other remote services. (For details, see [Configuring a Profile Service](#).)

Development

The following pages provide detailed instructions on developing custom profile services:

- **Profile Service Internals:** A description of the interfaces that must be implemented when creating a profile service, and how the interfaces will be called by the portal.

- **Implementing a Profile Service:** Step by step instructions on implementing the required interfaces, with example code.
- **Deploying a Profile Service:** How to deploy the profile service to a Java or .NET application server.
- **Configuring a Profile Service:** How to configure the profile service in the portal.

Other Methods for User Control

In addition to the authentication and profile services, the following functionality is available to control portal users in your implementation:

- **Experience definitions** let you tailor portal experiences for different groups of users.
- The **ICreateAccountActions Programmable Event Interface (PEI)** allows you to add functionality to the account creation process.
- **Remote User Operations** allow you to access and manage portal users from remote applications.

Authentication Service Internals

The AquaLogic Interaction Development Kit (IDK), formerly called the EDK, allows you to create remote authentication services and related configuration pages without parsing SOAP or accessing the portal API. The IDK Authentication API provides an abstraction from the necessary SOAP calls; you simply implement an object interface. For a complete listing of interfaces, classes, and methods, see the IDK API documentation for [.NET](#) or [Java](#).

Note: The differences between the Java and .NET versions of the IDK are platform-specific. In this guide, method names are listed using the .NET standard of initial capitalization. Java methods are identical, except begin with a lowercase letter. The `ISyncProvider.Initialize` method in the .NET IDK provides the same functionality as the `ISyncProvider.initialize` method in the Java IDK.

Plumtree.Remote.Auth

The `Plumtree.Remote.Auth` namespace (the `com.plumtree.remote.auth` package in Java), provides interfaces for creating authentication and synchronization services for users and groups in the portal. There are three interfaces provided:

- **ISyncProvider**
- **IGroup**
- **IAuthProvider**

To provide synchronization with an external source, implement `ISyncProvider` and `IGroup`. To provide authentication against an external source, implement `IAuthProvider`. In most cases, all three interfaces should be implemented.

Synchronization

User and group synchronization takes place when the associated synchronization Job is run by the portal Automation Service. The synchronization service must maintain state between the portal, the remote server, and the back-end system until synchronization is complete.

Users are imported on each run via `ISyncProvider`. Imported users are put into groups based on information from `IGroup` object(s). The portal typically calls the methods of the authentication service interfaces in the following order:

1. **ISyncProvider.Initialize**
2. **ISyncProvider.GetGroups**
3. **ISyncProvider.Initialize**
4. **ISyncProvider.GetUsers**
5. **ISyncProvider.Initialize**
6. **ISyncProvider.AttachToGroup**
for each group returned in **ISyncProvider.GetGroups**

1. IGroup.GetChildGroups

2. IGroup.GetChildUsers

Note that the portal may take a long time between calls to `GetGroups()`, `GetUsers()`, and `AttachToGroup()`. Because of this, the Java or .NET session on the remote server may time out, so `Initialize()` is called more than once.

Authentication

When a user logs into the portal, the authentication service is called to authenticate against the back-end system. This is done through a single call to **`IAuthProvider.Authenticate`**.

Once logged in, each user is associated with a portal `User` object; authentication services do not need to maintain state.

Implementing an Authentication Service

This section describes the details of how to create an authentication service. This authentication service is very simple, and is intended only as an example. It should not be used in a production environment.

The functional requirements for this authentication service are as follows:

- A single group will be synchronized into the portal: BASEGROUP.
- Ten users will be synchronized into the portal: TESTUSER0 - TESTUSER9.
- The ten users will be members of BASEGROUP.
- The ten users will all authenticate with the same password: TESTUSER.

The following interfaces will be implemented to create this authentication source:

- [ISyncProvider](#)
- [IGroup](#)
- [IAuthProvider](#)

These interfaces are in the **`Plumtree.Remote.Auth`** namespace (C#) or the **`plumtree.remote.auth`** package (Java). Any exceptions thrown in this code can be found in **`Plumtree.Remote`** (C#) or **`plumtree.remote`** (Java).

For the purposes of this authentication service, a simple class, `Constants`, was created in the example namespace/package. `Constants` has two public members, the String constants `GROUPNAME` and `USERNAME`. They are set at follows:

- `Constants.GROUPNAME = "BASEGROUP"`
- `Constants.USERNAME = "TESTUSER"`

These constants are used to provide the group name and a base for the user names to the authentication source code.

For the complete code of this example, see the sample code on the Developer Center.

For details of the classes described in this section, see the IDK (EDK) API documentation for [.NET](#) or [Java](#).

Except for in the example Java code, this section uses C# method names. In the IDK, methods are named the same in C# or Java, except for leading letter capitalization. For example, `ISyncProvider.GetUsers()` in the C# API is `ISyncProvider.getUsers()` in the Java API.

Implementing the ISyncProvider Interface

The `ISyncProvider` interface provides methods used by the portal to synchronize users and groups with a back-end repository. The methods of `ISyncProvider` are typically called in this order:

1. `ISyncProvider.Initialize()`
2. `ISyncProvider.GetGroups()`
3. `ISyncProvider.Initialize()`
4. `ISyncProvider.GetUsers()`
5. `ISyncProvider.Initialize()`
6. `ISyncProvider.AttachToGroup()`

Because the portal may take a long time between calls to `GetGroups()`, `GetUsers()` and `AttachToGroup()`, `Initialize()` is called more than once. This ensures that any configuration information passed to the synchronization service is available, even if the session has timed out.

`ISyncProvider.Initialize`

The `Initialize()` method is passed a `SyncInfo` object from the portal. The `SyncInfo` object is a set of name-value pairs, populated with information entered in the portal Service Configuration Interface (SCI) editor by a portal administrator. Typically, this is information such as credentials for connecting to the back-end system.

Java:

```
public boolean initialize(SyncInfo info)
    throws ServiceException
{
    return true;
}
```

C#:

```
public bool Initialize(SyncInfo syncInfo)
{
```

```
        return true;
    }
```

For this example authentication service, there is no need to perform any initialization, so the method simply returns true.

In a production implementation, a false should be returned if initialization fails. For example, if the authentication service cannot make a connection with the back-end repository. If false is returned, the synchronization job will stop.

ISyncProvider.GetGroups

The GetGroups() method is responsible for returning all of the groups from the back-end system. A SyncObject should be created for each group, using the static SyncObject.CreateGroup() method. The return value, a SyncObjectList, is then constructed using an array of SyncObject objects and a boolean flag, *isDone*.

The *isDone* flag determines whether or not the GetGroups() method will be called again. When you have a large number of groups in your back-end system, you can return groups to the portal in smaller batches. The size of each batch should be based on network bandwidth, the SOAP timeout set in the Authentication Web Service, and the speed of the back-end system. As a general rule, return no more than 1000 groups per batch.

If GetUsers() returns SyncObjects in batches, it must maintain state and set *isDone* to false until the last batch. Otherwise, *isDone* should be set to true.

Java:

```
public SyncObjectList getGroups()
    throws ServiceException
{
    SyncObject[] groups = new SyncObject[1];
    groups[0] = SyncObject.createGroup(
        Constants.GROUPNAME,
        Constants.GROUPNAME);
    return new SyncObjectList(groups, true);
}
```

C#:

```
public SyncObjectList GetGroups()
{
    SyncObject[] groups = new SyncObject[1];
    groups[0] = SyncObject.CreateGroup(
```

```

        Constants.GROUPNAME,
        Constants.GROUPNAME);
    return new SyncObjectList(groups, true);
}

```

For this authentication service, `GetGroups()` returns a single group, `BASEGROUP`. As stated above, `Constants.GROUPNAME` is a String set to "BASEGROUP". `SyncObject.CreateGroup()` accepts two String arguments. The first argument is the name the imported group will have in the portal. The second argument is the name of the group in the back-end system. In this case, we set both to "BASEGROUP".

The returned `SyncObjectList` is then constructed with the single item `SyncObject` array and the `isDone` flag set to true, signifying there are no more groups for the portal to synchronize.

ISyncProvider.GetUsers

The `GetUsers()` method is responsible for returning all of the users from the back-end system. Similar to `GetGroups()`, a `SyncObject` should be created for each group. For `GetUsers()`, use the static `SyncObject.CreateUser()` method to create each new `SyncObject`. The return value, a `SyncObjectList`, is then constructed using an array of `SyncObject` objects and a boolean flag, `isDone`.

As with `GetGroups()`, the `isDone` flag tells the portal whether it should call `GetUsers()` again or not, allowing you to break retrieval of users into batches. The size of each batch should be based on network bandwidth, the SOAP timeout set in the Authentication Web Service, and the speed of the back-end system. As a general rule, return no more than 1000 users per batch.

If `GetUsers()` returns `SyncObjects` in batches, it must maintain state and set `isDone` to false until the last batch. Otherwise, `isDone` should be set to true.

Java:

```

public SyncObjectList getUsers()
    throws ServiceException
{
    SyncObject[] users = new SyncObject[10];
    for (int i = 0; i < 10; i++)
    {
        String userName = Constants.USERNAME + i;
        users[i] = SyncObject.createUser(
            userName, userName, userName);
    }
    return new SyncObjectList(users, true);
}

```

```
}
```

C#:

```
public SyncObjectList GetUsers()
{
    SyncObject[] users = new SyncObject[10];
    for (int i = 0; i < 10; i++)
    {
        String userName = Constants.USERNAME + i;
        users[i] = SyncObject.CreateUser(
            userName, userName, userName);
    }
    return new SyncObjectList(users, true);
}
```

For this authentication service, `GetUsers()` returns ten users, TESTUSER0 - TESTUSER9. An array of ten `SyncObject` objects is created, and then populated using a for loop. The `SyncObject.CreateUser()` method takes three `String` arguments: The first is the name of the user in the portal, the second is the user name that will be passed for authentication, and the third is the name of the user in the back-end system. In this case, all are set to the same `String`, `Constants.USERNAME + i`.

The returned `SyncObjectList` is then constructed with the ten item `SyncObject` array and the `isDone` flag set to true, signifying there are no more users for the portal to synchronize.

ISyncProvider.AttachToGroup

`AttachToGroup()` returns an `IGroup` object that allows the portal to query for users and groups contained within a given group. For details on implementing the `IGroup` interface, see [Implementing the IGroup Interface](#), below.

`AttachToGroup()` is passed a `String`, a group identifier on the back-end system. This is the same as the second argument passed to `SyncObject.CreateGroups()` in `ISyncProvider.GetGroups()`. `AttachToGroup()` should return an instance of an implementation of the `IGroup` interface.

Java:

```
public IGroup attachToGroup(String groupID)
    throws ServiceException
{
```

```

    if (groupID.equals(Constants.GROUPNAME))
    {
        return new Group();
    }
    else
    {
        return null;
    }
}

```

C#:

```

public IGroup AttachToGroup(String groupID)
{
    if (groupID.Equals(Constants.GROUPNAME))
        return new Group();
    else
        return null;
}

```

For this authentication service, there is only one group, BASEGROUP. If the group ID passed to AttachToGroup() is BASEGROUP, a Group object is returned. Otherwise, null is returned. This is highly simplified; in a production implementation, AttachToGroup() would query a back-end system and return a group object with specific information for the given group.

Implementing the IGroup Interface

The IGroup interface provides methods that allow the portal to determine relationships between users and groups. The portal takes the IGroup object returned from each call to ISyncProvider.AttachToGroup() and calls two IGroup methods:

1. IGroup.GetChildGroups()
2. IGroup.GetChildUsers()

Similar to the ISyncProvider.GetGroups() and ISyncProvider.GetUsers() methods, the GetChildGroups() and GetChildUsers() methods return objects that contain an array of either groups or users. In both cases, the *isDone* flag can be used to send results back to the portal in batches. The size of each batch should be based on network bandwidth, the SOAP timeout set in the Authentication Web Service, and the speed of the back-end system. As a general rule, return no more than 1000 groups or users per batch.

Group.GetChildGroups

The GetChildGroups() method defines which child groups (subgroups) each group contains. The child groups are returned as ChildGroup objects in a ChildGroupList object. The ChildGroup constructor takes a single String argument, the unique name that identifies the group in the authentication service. The ChildGroupList constructor should be passed the array of ChildGroup objects and the *isDone* flag. If you want to return child groups in batches, your implementation of IGroup must maintain state internally and the *isDone* flag must be set to false until the final batch.

Java:

```
public ChildGroupList getChildGroups()
    throws ServiceException
{
    ChildGroup[] children = new ChildGroup[0];
    return new ChildGroupList(children, true);
}
```

C#:

```
public ChildGroupList GetChildGroups()
{
    ChildGroup[] children = new ChildGroup[0];
    return new ChildGroupList(children, true);
}
```

In this example authentication service, there are no child groups, so an empty array is returned.

IGroup.GetChildUsers

The GetChildUsers() method returns the user membership of the group. The users are returned as ChildUser objects, which are constructed with the same arguments as ISyncProvider.CreateUser(). The ChildUserList constructor should be passed the array of ChildGroup objects and the *isDone* flag. If you want to return child users in batches, your implementation of IGroup must maintain state internally and the *isDone* flag must be set to false until the final batch.

Java:

```
public ChildUserList getChildUsers()
    throws ServiceException
{
    ChildUser[] users = new ChildUser[10];
}
```

```

    for (int i = 0; i < 10; i++)
    {
        String userName = Constants.USERNAME + i;
        users[i] = new ChildUser(userName, userName,
        userName);
    }
    return new ChildUserList(users, true);
}

```

C#:

```

public ChildUserList GetChildUsers()
{
    ChildUser[] users = new ChildUser[10];
    for (int i = 0; i < 10; i++)
    {
        String userName = Constants.USERNAME + i;
        users[i] = new ChildUser(userName, userName,
        userName);
    }
    return new ChildUserList(users, true);
}

```

For this authentication service, GetChildUsers() returns the same ten users as ISyncProvider.GetUsers(), TESTUSER0 - TESTUSER9. An array of ten ChildUser objects is created, and then populated using a for loop. The ChildUser constructor takes three String arguments: the first is the name of the user in the portal, the second is the user name that will be passed for authentication, and the third is the name of the user in the back-end system. In this case, all are set to the same String, Constants.USERNAME + i.

The returned ChildUserList is then constructed with the ten item ChildUser array and the *isDone* flag set to true, signifying there are no more users associated with this group.

Implementing the IAuthProvider Interface

The IAuthProvider interface validates credentials from a portal login against a back-end repository. There is a single method to implement, Authenticate().

IAuthProvider.Authenticate

The Authenticate() method is passed three arguments: two Strings for username and password, and an AuthInfo object. The AuthInfo object, like the SyncInfo object passed to ISyncProvider.Initialize(), is a set of name-value

pairs, populated with information entered in the portal SCI editor by a portal administrator. Typically, this is information such as credentials for connecting to the back-end system.

If the credentials passed are valid, Authenticate() should return normally; however, if the credentials are invalid, an exception of type ServiceException must be thrown. See the IDK API documentation for a complete description of the exceptions derived from ServiceException.

Java:

```
public void authenticate(String username,
                        String password, AuthInfo authinfo)
    throws ServiceException
{
    if (username.startsWith(Constants.USERNAME_BASE) &&
        password.startsWith(Constants.USERNAME_BASE))
    {
        //do nothing- authenticated
    }
    else
    {
        throw new AccessDeniedException();
    }
}
```

C#:

```
public void Authenticate(String username,
                        String password, AuthInfo authInfo)
{
    if (username.StartsWith(Constants.USERNAME) &&
        password.StartsWith(Constants.USERNAME))
    {
        //do nothing- authenticated
    }
    else
    {
        throw new AccessDeniedException();
    }
}
```

```
}
```

For this authentication service, if the username and password passed in start with Constants.USERNAME ("TESTUSER"), the user is authenticated. Otherwise, an AccessDeniedException is thrown.

Note that if a message is provided in the exception, the message will not be displayed to the user in the UI. The message will be caught by the IDK and sent to the [ALI Logging Utilities](#).

Deploying an Authentication Service

This section describes how to deploy an authentication service to an application server.

- [Deploying a Java Authentication Service](#)
- [Deploying a .NET Authentication Service](#)

It is assumed you have accomplished the following prior to attempting to deploy an authentication service:

1. IDK (EDK) 5.3 is installed on the server to which you intend to deploy. For details on installing the IDK, see [Setting Up the IDK](#).
2. You have implemented ISyncProvider and IGroup (for synchronization), IAuthProvider (for authentication), or both. Details on implementation can be found on the previous page, [Implementing an Authentication Service](#).

Deploying a Java Authentication Service

To deploy an authentication service to a supported Java application server:

1. Access the IDK deployment servlet (DeployServlet) in a browser:

http://<app server>:<port>/edk/DeployServlet
2. Choose **Auth** and wait for the page to reload.
3. Enter a prefix to identify this authentication service and the fully qualified name of the implementation of IAuthProvider, ISyncProvider, or both.
4. If this service uses SCI, check **Use Service Configuration Interface (SCI)** and enter the fully qualified name of the appropriate implementation of IAdminEditor.
5. Copy and paste the URLs displayed on the results page to a text file; these are the URLs that should be used when you configure the service in the portal.

For detailed instructions on deployment, see [Deploying Web Services](#).

Deploying a .NET Authentication Service

To deploy an authentication service to IIS and .NET:

1. Ensure that you have built your project with the AuthProviderSoapBinding.asmx and/or SyncProviderSoapBinding.asmx SOAP endpoints. If this service uses SCI, also include SCIProviderBinding.asmx. These files can be found in your IDK installation. For example:

C:\Program Files\plumtree\ptedk\5.3\devkit

2. If this service provides authentication, update the Web.config, adding the following nodes to <appSettings>:
 1. <add key="AuthProviderAssembly" value="[ASSEMBLY NAME]" />

Where [ASSEMBLY NAME] is the name of the assembly containing your IAuthProvider implementation.

2. <add key="AuthProviderImpl" value="[FULLY QUALIFIED PATH]" />

Where [FULLY QUALIFIED PATH] is the fully qualified path to the class implementing IAuthProvider.

```
<appSettings>
  <add key="AuthProviderAssembly" value="Helloworld" />
  <add key="AuthProviderImpl" value="Plumtree.Remote.Auth.Helloworld.Auth" />
  ...
```

3. If this service provides synchronization, update the Web.config, adding the following nodes to <appSettings>:

1. <add key="SyncProviderAssembly" value="[ASSEMBLY NAME]" />

Where [ASSEMBLY NAME] is the name of the assembly containing your ISyncProvider implementation.

2. <add key="SyncProviderImpl" value="[FULLY QUALIFIED PATH]" />

Where [FULLY QUALIFIED PATH] is the fully qualified path to the class implementing ISyncProvider.

```
<appSettings>
  ...
  <add key="SyncProviderAssembly" value="Helloworld"
```

```
/>  
  
    <add key="SyncProviderImpl"  
    value="Plumtree.Remote.Auth.Helloworld.Sync" />  
  
    ...
```

4. If this service uses SCI, update the Web.config, adding the following nodes to <appSettings>:
 1. <add key="AdminEditorAssembly" value="[ASSEMBLY NAME]" />

Where [ASSEMBLY NAME] is the name of the assembly containing your IAdminEditor implementation.
 2. <add key="AdminEditorImpl" value="[FULLY QUALIFIED PATH]" />

Where [FULLY QUALIFIED PATH] is the fully qualified path to the class implementing IAdminEditor.

For detailed instructions on deployment, see [Deploying Web Services](#).

Configuring Portal Objects

To register an authentication service in the portal, you must create the following administrative objects:

- **Remote Server** (optional): Multiple services can share a single remote server object. Authentication Web Services can use a Remote Server object or hard-coded URLs. If you will be using a Remote Server object, you must register it with the portal before registering any related Web Service objects.
- **Web Service - Authentication**: Each remote authentication service must have an associated Authentication Web Service object. The Authentication Web Service editor allows you to specify general settings for the back-end system. Multiple Authentication Source objects can use the same Web Service object.
- **Authentication Source - Remote**: Each Authentication Web Service has one or more associate Remote Authentication Source objects that define basic settings.
- **Job**: To run the authentication service, you must schedule a job or add the Authentication Source to an existing job. The Remote Authentication Source editor allows you to set a job.

Authentication Web Service Editor

Configuring the following settings correctly in the Authentication Web Service editor is essential:

- **The encoding style must reflect the service implementation (.NET vs Java)**. The encoding style is set on the Advanced Settings

page. For .NET, you must set the encoding to Document/Literal. Java uses the default, RPC/Encoded.

- **All configuration pages must be entered on the Advanced URLs page.** You can add configuration pages to the Authentication Source editor. These URLs must be entered on the Advanced URLs page.

Authentication Source Editor

Keep the following in mind when configuring the Authentication Source:

- **Users imported by a synchronization service must be unique by name *and* Authentication Source.** The portal identifies users first by their category, then by username; this combination must be unique per user. It is a best practice to use the source domain for the category name. The category is entered in the Authentication Source editor. You can use the same category for multiple back-end systems, but the systems must not have users or groups with the same name.
- **The *description* of the Authentication Source object is displayed on the portal login page.** Creating an Authentication Source object with a synchronization component creates an option in the authentication source drop-down list on the portal login page. The name that appears in the drop-down list is the *description* of the Authentication Source object. Enter a description that users will recognize.
- **By default, the portal performs partial users synchronization.** Confirm that the synchronization settings are correct for the service. The default of Partial User Synchronization may not perform the synchronization you desire.

For details on editor settings, see the portal online help.

Profile Service Internals

The AquaLogic Interaction Development Kit (IDK), formerly called the EDK, allows you to create remote profile services and related configuration pages without parsing SOAP or accessing the portal API. The IDK Profile API provides an abstraction from the necessary SOAP calls; you simply implement an object interface. For a complete listing of interfaces, classes, and methods, see the IDK API documentation for [.NET](#) or [Java](#).

Note: The differences between the Java and .NET versions of the IDK are platform-specific. In this guide, method names are listed using the .NET standard of initial capitalization. Java methods are identical, except begin with a lowercase letter. The `IProfileProvider.Initialize` method in the .NET IDK provides the same functionality as the `IProfileProvider.initialize` method in the Java IDK.

Plumtree.Remote.Profile

The `Plumtree.Remote.Profile` namespace (`plumtree.remote.profile` package in Java) provides the following interfaces:

- **IProfileProvider**
- **IUser**

To import information from an external source into portal user properties, you must implement both interfaces.

Profile Synchronization

The portal accesses a remote profile service when the associated job is run by the portal Automation Service. The portal calls the methods of the profile service interfaces in the following order:

1. `IProfileProvider.Initialize()`
2. `IProfileProvider.GetGlobalSignature()`
3. `IProfileProvider.AttachToUser()`
 - a. `IUser.GetUserSignature()`
 - b. `IUser.GetUserProperties()`
4. `IProfileProvider.Shutdown()`

Step 3 is called by the portal once for every user in the group or groups to be synchronized. Step 4, the `IProfileProvider.Shutdown()` method is optional. It can be used to clean up resources used by the profile service; however, it may or may not be called by the portal, so it should not be relied upon.

Implementing a Profile Service

This section describes the details of how to create a profile service by taking you step by step through the implementation of a sample profile service. This is a very simple profile service, and is not intended for use in a production environment.

The functional requirement for this profile service is simply:

- For any request, return the user's profile property "REGION" set to "WEST".

The following interfaces will be implemented to create this profile service:

- [IProfileProvider](#)
- [IUser](#)

These interfaces are in the **Plumtree.Remote.Profile** namespace (C#) or the **plumtree.remote.profile** package (Java). Any exceptions thrown in this code can be found in **Plumtree.Remote** (C#) or **plumtree.remote** (Java).

For the complete code of this example, see the sample code on the Developer Center.

For details of the classes described in this section, see the IDK API documentation for [.NET](#) or [Java](#).

Except for in the example Java code, this section uses C# method names. In the IDK, methods are named the same in C# or Java, except for leading letter capitalization. For example, `IProfileProvider.GetGlobalSignature()` in the C# API is `IProfileProvider.getGlobalSignature()` in the Java API.

Implementing the IProfileProvider Interface

The `IProfileProvider` interface is used by the portal to initiate access to, and obtain user profile information from, the back-end repository. The portal calls the methods of `IProfileProvider` in the following order:

1. `IProfileProvider.Initialize()`
2. `IProfileProvider.GetGlobalSignature()`
3. `IProfileProvider.AttachToUser()`
4. `IProfileProvider.Shutdown()`

IProfileProvider.Initialize

`Initialize()` allows the profile service to initialize a session and create a connection to the back-end repository. The method is passed two objects from the portal: `PropertyList` and `ProfileInfo`. `PropertyList` is the list of attributes mapped to properties on the Property Map page of the Profile Source object in the portal. `ProfileInfo` is a set of name-value pairs, populated with information entered in the portal Service Configuration Interface (SCI) editor by a portal administrator. Typically, this is information such as credentials for connecting to the back-end system.

Java:

```
private String[] m_propertyList;
protected String[] getPropertyList()
{
    return m_propertyList;
}
public void initialize(String[] propertyList,
    ProfileInfo profileInfo)
    throws ServiceException
{
    this.m_propertyList = propertyList;
}
```

C#:

```
private string[] m_propertyList;
internal string[] GetPropertyList()
{
    return m_propertyList;
}
public void Initialize(string[] PropertyList,
    ProfileInfo ProfileSourceInfo)
{
    this.m_propertyList = PropertyList;
}
```

In this example profile service, the PropertyList is stored in the m_propertyList member variable, where it can later be accessed by the IUser implementation. The IUser implementation requires the PropertyList to determine which properties to retrieve.

IProfileProvider.GetGlobalSignature

GetGlobalSignature() allows the portal to determine whether profile information for any of the users has changed. The portal compares the string returned from GetGlobalSignature() with the string returned from GetGlobalSignature() during the previous run of the profile service job. If the two strings match, the job stops. The IDK does not enforce any restrictions on the string used for the global signature; it can be a last-modified date, a random number, or another identifier.

Java:

```
public String getGlobalSignature()  
    throws ServiceException  
{  
    return new Date().toString();  
}
```

C#:

```
public string GetGlobalSignature()  
{  
    return System.DateTime.Now.Ticks.ToString();  
}
```

This profile service returns a string representation of the current date and time. This ensures the job will continue to `AttachToUser()`.

IProfileProvider.AttachToUser

`AttachToUser()` is called for each user within the group or groups configured in the Profile Source editor in the portal. The first three parameters passed to `AttachToUser()` identify which user the profile service should retrieve from the back-end system:

- **UserID:** The portal user ID. Can be used via the PRC to look up other user attributes.
- **LoginName:** The portal login name. If this user was added using an authentication service, this value corresponds to `ChildUser.UserName`.
- **UniqueName:** Usually the name used to look up the user in the back-end system. If the user was added using an authentication service, this value corresponds to `ChildUser.UserUniqueName`.

If these parameters do not identify a valid user, a `NoSuchUserException` should be thrown.

The final parameter, `LastSignature`, is the signature returned by `IUser.GetUserSignature()` during the previous job.

Java:

```
public IUser attachToUser(int userId, String loginName,  
                          String uniqueName, String lastSignature)  
    throws ServiceException  
{  
    return new User(this);  
}
```

C#:

```
public IUser AttachToUser(int UserID, string LoginName,
                          string UniqueName, string LastSignature)
{
    return new User(this);
}
```

For this profile service, AttachToUser() simply returns an instance of User, the IUser implementation. This is appropriate for this implementation because the profile service updates the "Region" property to "WEST" regardless of the portal user being queried.

IProfileProvider.Shutdown

As a performance optimization, the portal might call the Shutdown() method. No parameters are received or returned. This method is optional on both ends; the profile service might not receive the Shutdown message, and, if received, the profile service can ignore the call to Shutdown().

Shutdown() can be used to clean up resources used by the profile service; however, you should not rely on it being called.

Implementing the IUser Interface

Returned by IProfileProvider.AttachToUser(), the IUser interface is used by the portal to synchronize profile property information for a specific user. The portal calls the methods of IUser in the following order:

1. IUser.GetUserSignature()
2. IUser.GetUserProperties()

Note, for this profile service, the IUser interface implementation, User, has a constructor that accepts a parameter of type Profile (the IProfileProvider implementation), which is stored in the private member m_profile. This is variable is used in GetUserProperties() to access the PropertyList.

IUser.GetUserSignature

GetUserSignature() is similar to IProfileProvider.GetGlobalSignature(), except it allows the portal to determine if a specific user's profile information has changed. If the returned string matches the string returned from GetUserSignature() during the previous job, GetUserProperties() is not called. The IDK does not enforce any restrictions on the string used for the global signature; it can be a last-modified date, a random number, or another identifier.

Java:

```
public String getUserSignature()
```

```
        throws ServiceException
    {
        return new Date().toString();
    }
```

C#:

```
public string GetUserSignature()
{
    return System.DateTime.Now.Ticks.ToString();
}
```

This profile service returns a string representation of the current date and time. This ensures GetUserProperties() is called each time the job runs.

IUser.GetUserProperties

Typically, GetUserProperties() will access the PropertyList object from the IProfileProvider implementation, retrieving values for each property in the PropertyList from the back-end system. GetUserProperties then builds a UserPropertyInfo object to return to the portal. The portal maps the back-end property names with portal properties and updates the portal property values.

Java:

```
public UserPropertyInfo getUserProperties()
    throws ServiceException
{
    String prop;
    UserPropertyInfo info = new UserPropertyInfo();
    for (int i=0; i < m_profile.getPropertyList().length;
i++)
    {
        prop = m_profile.getPropertyList()[i];
        if (prop.equalsIgnoreCase("REGION"))
        {
            info.put(prop, "WEST");
        }
    }
    return info;
}
```

```
}
```

C#:

```
public UserPropertyInfo GetUserProperties()
{
    String prop;
    UserPropertyInfo info = new UserPropertyInfo();
    for (int i=0; i < m_profile.GetPropertyList().Length;
i++)
    {
        prop = m_profile.GetPropertyList()[i];
        if (prop.ToUpper().Equals("REGION"))
        {
            info.Put(prop, "WEST");
        }
    }
    return info;
}
```

In this profile service, a for loop checks each property in the PropertyList against the string "REGION". If "REGION" is found, a UserPropertyInfo object is updated with name "REGION" and value "WEST" and returned to the portal.

Deploying a Profile Service

This section describes how to deploy a profile service to an application server.

- [Deploying a Java Profile Service](#)
- [Deploying a .NET Profile Service](#)

It is assumed you have accomplished the following prior to attempting to deploy a profile service:

1. IDK (EDK) 5.3 is installed on the server to which you intend to deploy. For details on installing the IDK, see [Setting Up the IDK](#).
2. You have implemented IProfileProvider and IUser. Details on implementation can be found on the previous page, [Implementing a Profile Service](#).

Deploying a Java Profile Service

To deploy a profile service to a supported Java application server:

1. Access the IDK deployment servlet (DeployServlet) in a browser:
http://<app server>:<port>/edk/DeployServlet
2. Choose **Profile** and wait for the page to reload.
3. Enter a prefix to identify this profile service and the fully qualified name of the implementation of IProfileProvider.
4. If this service uses SCI, check **Use Service Configuration Interface (SCI)** and enter the fully qualified name of the appropriate implementation of IAdminEditor.
5. Copy and paste the URLs displayed on the results page to a text file; these are the URLs that should be used when you configure the service in the portal.

For detailed instructions on deployment, see [Deploying Web Services](#).

Deploying a .NET Profile Service

To deploy a profile service to IIS and .NET:

1. Ensure that you have built your project with the ProfileProviderSoapBinding.asmx SOAP endpoint. If this service uses SCI, also include SCIProviderBinding.asmx. These files can be found in your IDK installation. For example:

C:\Program Files\plumtree\ptedk\5.3\devkit

2. Update the Web.config, adding the following nodes to <appSettings>:
 1. <add key="ProfileProviderAssembly" value="[ASSEMBLY NAME]" />

Where [ASSEMBLY NAME] is the name of the assembly containing your IProfileProvider implementation.

2. <add key="ProfileProviderImpl" value="[FULLY QUALIFIED PATH]" />

Where [FULLY QUALIFIED PATH] is the fully qualified path to the class implementing IProfileProvider.

```
<appSettings>
  <add key="ProfileProviderAssembly"
value="HelloWorldProf_CS" />
  <add key="ProfileProviderImpl"
value="HelloWorldProf_CS.Profile" />
  ...

```

3. If this service uses SCI, add the following nodes to <appSettings>:
 1. <add key="AdminEditorAssembly" value="[ASSEMBLY NAME]" />

Where [ASSEMBLY NAME] is the name of the assembly containing your IAdminEditor implementation.

2. `<add key="AdminEditorImpl" value="[FULLY QUALIFIED PATH]" />`

Where [FULLY QUALIFIED PATH] is the fully qualified path to the class implementing IAdminEditor.

For detailed instructions on deployment, see [Deploying Web Services](#).

Configuring Portal Objects

To register a profile service in the portal, you must create the following administrative objects:

- **Remote Server** (optional): Multiple services can share a single Remote Server object. Profile Sources can use a Remote Server object or hard-coded URLs. If you will be using a Remote Server object, you must register it with the portal before registering any related Web Service objects.
- **Web Service - Profile**: Each Remote Profile Service must have an associated Profile Web Service object. The Profile Web Service editor allows you to specify general settings for the back-end system. Multiple Profile Source objects can use the same Web Service object.
- **Profile Source - Remote**: Each Profile Web Service has one or more associated Remote Profile Source objects that define basic settings. The Remote Profile Source editor can include service configuration pages created for the Profile Source and entered in the Profile Web Service editor.
- **Property**: To create new user information properties, you must first create a Property object using the Property editor.
- **Global Object Property Map**: To add new user information properties to the portal, use the Global Object Property Map.
- **User Profile Manager**: To associate user information properties with user profiles in the portal, use the User Profile Manager.
- **Job**: To run the profile service, you must schedule a job or add the Profile Source to an existing job. The Remote Profile Source editor allows you to set a job.

Profile Web Service Editor

Configuring the following settings correctly in the Profile Web Service editor is essential:

- **The encoding style must reflect the service implementation (.NET vs Java)**. The encoding style is set on the Advanced Settings page. For .NET, you must set the encoding to Document/Literal. Java uses the default, RPC/Encoded.

- **All configuration pages must be entered on the Advanced URLs page.** You can add configuration pages to the Profile Source editor. These URLs must be entered on the Advanced URLs page.

Global Object Property Map

The Global Object Property Map displays the types of portal objects with which you can associate properties. Values for a portal object's associated properties are specified on the Properties and Names page of the object's editor.

To import new user information properties into the portal, you must add mappings to the Global Object Property Map. Each property must be mapped to the User object.

To create a property in the portal, choose Create Object... | Property in portal Administration. After you have created a property, you can add it to the Global Object Property Map.

For more details on the Global Object Property Map, see the portal online help.

User Profile Manager

When a profile service imports user information into the portal, the attributes imported must be associated with portal properties. To make these properties available to other services, you must associate them with user information using the User Profile Manager.

On the User Information Property Map page of the User Profile Manager, add any properties that should be associated with user information settings. (The properties must already exist in the portal and be associated with the User object in the Global Object Property Map as explained in the above section.)

For details on the User Profile Manager, see the portal online help.