Oracle® Fusion Middleware

Developing Plug-Ins for Oracle Unified Directory

19*c* (19.1.0.0.0)
E97672-01
July 2018

Oracle Fusion Middleware Developing Plug-Ins for Oracle Unified Directory, 19*c* (19.1.0.0.0)

E97672-01

Primary Author: Sandhya U.S

Contributing Authors: Devanshi Mohan

Contributors: Lawallambok Wahlang

Contents

Preface

The *Oracle Fusion Middleware Developing Plug-Ins for Oracle Unified Directory* describes how to use the Oracle Unified Directory Plug-In API to programmatically extend OUD functionality.

Audience

This document is intended for software developers who are proficient in using Oracle Unified Directory.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

For more information, see the following documents in the Oracle Unified Directory 12*c* Release 12.2.1.3 documentation set:

- *Release Notes for Oracle Identity Management*
- *Oracle Fusion Middleware Installation Guide for Oracle Unified Directory*
- *Oracle Fusion Middleware Oracle Unified Directory Configuration Reference*
- *Oracle Fusion Middleware Administrator's Guide for Oracle Unified Directory*
- *Oracle Fusion Middleware Java API Reference for Oracle Unified Directory*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.

Convention	Meaning
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

What's New in This Guide

The following topics introduce the new and changed features of *Oracle Fusion Middleware Developing Plug-ins for Unified Directory*, and other significant changes that are described in this guide. This document is the new edition of the formerly titled *Oracle Unified Directory Developer's Guide*.

New Features in Release 12*c* (12.2.1.3.0)

This revision contains no new features. Minor updates were made throughout the guide.

1

Understanding Basic Oracle Unified Directory Plug-in Concepts

You can decide on the benefits and cost of having a Oracle Unified Directory plug-in implementation and OUD workflows. OUD plug-ins are used to write LDAP error codes, on-demand password migration and authentication.
Understanding basic Oracle Unified Directory plug-in is explained in the following sections:

1.1 Determining Whether You Should Implement an OUD Plug-In

The Oracle Unified Directory (OUD) plug-in API provides the means to extend existing Directory Server functionality. You may want to develop a plug-in if you have a very specific directory server requirement that OUD cannot address straight out of the box.

For example, OUD plug-ins have been used successfully to achieve the following:

- LDAP error code and error message writing
- On-demand password migration
- Authentication using multiple password types
- Operation routing based on criteria in user entry

Some of these plug-ins have played a role in helping Oracle customers to seamlessly migrate to OUD. These are just examples of how Directory Server functionality can be enhanced by customizing LDAP operations and programatically manipulating results.

As you analyze the benefits and costs of developing your own OUD plug-in, consider the following:

- To minimize potential points of failure in your directory deployment, you should develop your own OUD plug-in only when no existing OUD functionality, nor any combination of OUD features, can achieve the results you require.
- When upgrading to a later release, you will have to determine whether your custom plug-in is still relevant in light of new OUD functionality that may evolve over time. Moreover, you may have to update your plug-in to ensure backward or forward compatibility with later releases of OUD.

1.2 OUD Plug-Ins and OUD Workflows

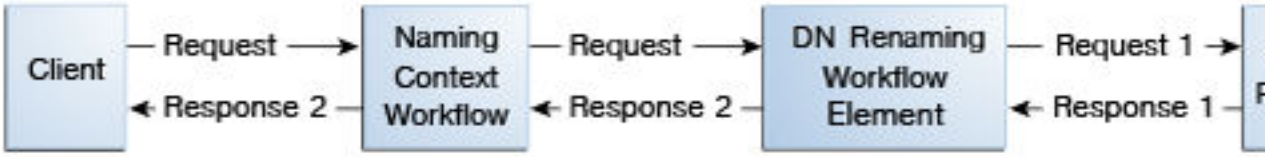
An OUD plug-in can be seen as a new type of OUD workflow element. Workflows and workflow elements are fundamental building blocks within the OUD directory architecture.

See Understanding Oracle Unified Directory Concepts and Architecture in *Oracle Fusion Middleware Administrator's Guide for Oracle Unified Directory*.

An OUD plug-in can be inserted into any OUD workflow element tree. The following are typical tasks that an OUD plug-in can perform within a workflow element tree:

- Intercept LDAP requests from the previous workflow elements in the chain, and keep the option to change or extend them.
- Intercept LDAP entries and results from next workflow element in the chain.
- Stack and leverage other workflow elements delivered out of the box with OUD.
- Invoke a plug-in upon receipt of LDAP requests, and after the routing decision is done by the workflow.

The following figure illustrates a typical OUD workflow containing a Naming Context workflow and a DN Renaming workflow element. An OUD plug-in is inserted downstream from these building blocks, and upstream from a remote backend workflow element.



Once you have developed a number of OUD plug-ins, you can form a plug-in chain within an OUD workflow.

1.3 OUD Plug-In Implementation Points

OUD plug-ins interact with the OUD core server through set of implementation points such as managing and LDAP operation handling.

Following are the sets of implementation points:

- Administrative plug-in management: startup, shutdown, status, and configuration changes
- LDAP operation handling
- The context that is the main interface between the plug-in and the core directory server as it is used to log requests and to create instances of objects manipulated by the plug-in API

The implementation points for managing the plug-in are defined in the `oracle.oud.plugin.ManagedPlugin` interface:

- The `initializePlugin()` method is invoked when the plug-in is initialized at server startup time.
- The `finalizePlugin()` method is invoked when the plug-in is stopped.
- The `handleConfigurationChange()` method is invoked whenever the plug-in configuration is changed. See Making Dynamic OUD Plug-In Configuration Changes.

Implementation points for intercepting LDAP operations are defined in the `oracle.oud.plugin.RequestManager` interface. See Configuring Internal Operations in OUD Plug-in API .

1.4 About Oracle Unified Directory Plug-Ins

An Oracle Unified Directory plug-in is an implementation of `oracle.oud.plugin.ManagedPlugin` which is formed from the following three Java interfaces: `oracle.oud.RequestManager`, `oracle.oud.plugin.Plugin`, and `oracle.oud.plugin.ManagedPlugin`.

Java Interface
Description

oracle.oud.RequestManager
Defines a method for each type of operation defined by the LDAP protocol. The method named `handleAdd` is called each time the plug-in is involved in an LDAP `add` operation. Similar methods exist for `bind`, `compare`, `delete`, `modify`, `modifyDN`, and `search` operations. Exceptions exist for the `abandon` and `unbind` operations; these two types of request cannot be intercepted.

oracle.oud.plugin.Plugin
Associates a name to the plug-in that is unique per instance. Identifying plug-ins is helpful when a plug-in routes the received requests to a particular plug-in among multiple plug-ins.

oracle.oud.plugin.ManagedPlugin
Defines the life cycle of the plug-in. The life cycle begins with the initialization of the plug-in when the server starts or the plug-in is created. Once initialized, a plug-in is able to receive configuration changes. When the server is shut down or the plug-in is removed from the server configuration, the plug-in is finalized.

An OUD plug-in can be followed by one or more plug-ins in a process chain. The most common case is an OUD plug-in that is followed by only one plug-in. This type of plug-in receives requests, may perform extra actions such as logging or modifying the received requests, and then forwards the requests to the next plug-in. When the LDAP operation returns a response, similar actions can be performed.

A plug-in that has no subsequent plug-ins in the process chain is responsible for storing the entries manipulated by the LDAP requests. The storage can be local or remote. In both cases, the plug-in is responsible for assigning the result of the received LDAP requests.

A plug-in that is followed by multiple plug-ins in the process chain is the most difficult case. This type of plug-in is used for only complex architectures that include distribution or load balancing. For example, this type of plug-in might be used for routing bind requests on dedicated plug-ins, and routing other LDAP operations on other plug-ins.

The OUD plug-in API provides a default implementation of the `oracle.oud.plugin.ManagedPlugin` Java interface that is the abstract class `oracle.oud.plugin.AbstractPlugin`. This class provides a default implementation of a plug-in that performs no action apart from forwarding the received requests to its next plug-in the chain of processing. The default implementation assumes that the plug-in has at least one subsequent plug-in. But you can overwrite appropriate methods to change the default behavior if necessary.You should make your plug-in implementation derive from the `oracle.oud.plugin.AbstractPlugin` class. This will optimize backward compatibility in case the implemented Java interface is changed.

2
Building and Deploying an OUD Plug-In

You can build and deploy an Oracle Unified Directory (OUD) plug-in that does not perform any action.
The following topics explain the building and deploying an OUD Plug-In:

- Before You Begin Deploying OUD Plug-in
- Deploying a Plug-In to an OUD Instance

2.1 Before You Begin Deploying OUD Plug-in

You need to prepare your development environment such as installing OUD and JDK, before you deploy an OUD plug-in.

Complete the following tasks:

- Install Oracle Unified Directory and create a new instance with fifty generated entries.
- Install the Java Development Kit with the exact same version of the Java Runtime Environment running in the Oracle Unified Directory instance.
- Create a new project for the development of your plug-in using your favorite integrated development environment (IDE), and reference the JAR file `oud-sdk.jar` that is located in *install-dir*`/oud/lib/oud-sdk.jar`

2.2 Deploying a Plug-In to an OUD Instance

The Oracle Unified Directory (OUD) plug-in API provides the means to extend existing Directory Server functionality.

Perform the following steps to deploy a plug-in to an OUD instance:

1. Create a new class that extends the class `oracle.oud.plugin.AbstractPlugin`. This class will not perform any action but will be part of the processing. For example:

   ```
   package oracle.oud.example;
   import oracle.oud.plugin.AbstractPlugin;
   /**
   * A plug-in that does not perform any action.
   */
   public class ExamplePlugin
   extends AbstractPlugin
   {

   }
   ```

2. Build your plug-in project.

   The content of the generated plug-in JAR file should contain the following files:

   - `META-INF/MANIFEST.MF`

- `oracle/oud/example/ExamplePlugin.class`

3. To ensure that your plug-in will continue to work with subsequent releases of the OUD plug-in API, you can embed a specific versioning file in the produced JAR file. The name of the file to embed is `plugin.properties`.

   Make the following modifications to the `plugin.properties` file:

   a. To define a target version for all plug-ins contained in the JAR file, add the following:

      ```
      plugin.version=11.1.2.1.0
      ```

   b. To define a target version only for the `ExamplePlugin` plug-in, specify the following:

      ```
      plugin.ExamplePlugin.version=11.1.2.1.0
      ```

   If the `plugin.properties` file is missing, then the behavior of the current implementation of the plug-in API applies.

4. Restart the Oracle Unified Directory instance for the JAR file changes to take effect.

   a. Stop the OUD instance.

      UNIX, Linux

      ```
      $ cd instance-directory/OUD/bin
      $ stop-ds
      ```

      Windows

      ```
      C:\> cd instance-directory\OUD\bat
      C:\> stop-ds
      ```

   b. Copy the plug-in JAR file into the `lib` directory.

      UNIX, Linux

      ```
      # cp plugin.jar lib
      ```

      Windows

      ```
      C:\> copy plugin.jar lib
      ```

   c. Restart OUD instance.

      UNIX, Linux

      ```
      # start-ds
      ```

      Windows

      ```
      C:\> start-ds
      ```

5. Modify the server configuration so that your plug-in is part of the server processing.

   Use the `dsconfig` command to declare a plug-in as a workflow element in an OUD server. You must specify the following information:

   - A plug-in name (`ExamplePlugin` in the example) that uniquely identifies this plug-in instance
   - The name of the Java class that implements the `oracle.oud.plugin.ManagedPlugin` Java interface

- Whether the plug-in is enabled or disabled
- The name of the workflow element that is behind the plug-in to be inserted.

  A plug-in may have 0, 1, or more next workflow elements depending on the use case it implements. For example:

```
# dsconfig create-workflow-element \
    --set enabled:true \
    --set plugin-class:oracle.oud.example.ExamplePlugin \
    --set next-workflow-elements:userRoot \
    --type plugin \
    --element-name ExamplePlugin
```

After creating the plug-in, insert a plug-in workflow element in a workflow. The plug-in workflow element should appear either as the next element of a workflow, or plug-in class as the next element of an existing workflow element. The following command changes the configuration of the workflow `userRoot0` to forward LDAP requests to the previously added example plug-in:

```
# dsconfig set-workflow-prop \
    --workflow-name userRoot0 \
    --set workflow-element:ExamplePlugin
```

It is possible to create several instances of the same plug-in implementation as long each instance has a unique name.

3
Using the OUD Plug-In API Reference

OUD Plug-In API is used to handle requests and responses, results and configuring filters in search requests.

See the *Java API Reference for Oracle Unified Directory* for detailed information about Oracle Unified Directory (OUD) Java classes, methods, and related syntax and usage.

This chapter provides general information about using the OUD Plug-In API.

3.1 Overview of OUD Plug-In Configuration

The OUD Plug-In API provides convenient ways to store, retrieve, modify, and validate the plug-in configuration.

The following sections provide conceptual information and examples for working with OUD plug-ins:

3.1.1 About Storing OUD Plug-In Configuration

OUD stores plug-in configuration as part of the plug-in configuration entry. The configuration elements are stored in the OUD `config.ldif` file as key-value pairs. For simplicity, you should use this mechanism. However, the OUD plug-in architecture allows you to use alternative methods, such as an external file, to retrieve the configuration.

The plug-in configuration is represented as a set of key-value pairs in the default configuration model. Key and value are treated as raw strings by the OUD server and the `dsconfig` command line tools. You can set key-value pairs using the `dsconfig` tool and the `plugin-properties` property associated with plug-in workflow elements.

For more information, see the following examples:

- Example for Adding Plug-in Properties
- Example for Configuring a Custom Property

3.1.1.1 Example for Adding Plug-in Properties

The following example for ***Adding Plug-in Properties*** demonstrates how to add plug-in properties.

```
dsconfig set-workflow-element-prop \
          --element-name ExamplePlugin \
          --add plugin-properties:customProperty=localDB1 \
          --hostname host1 \
          --port 4444 \
--trustStorePath install-dir/OUD/config/admin-truststore \
          --bindDN cn=Directory\ Manager \
          --bindPasswordFile ****** \
          --no-prompt
```

3.1.1.2 Example for Configuring a Custom Property

In the following example for ***Configuring a Custom Property*** the plug-in `ExamplePlugin` is configured with a custom property named `customProperty`. This property is specified as a value of the generic `plugin-properties` parameter.

```
$dsconfig get-workflow-element-prop --element-name ExamplePlugin

Property : Value(s)
---------------------:----------------------------------------------------
enabled : true
next-workflow-elements : localDB1
plugin-class : oracle.oud.plugin.example.ExamplePlugin
plugin-properties : customProperty=localDB1
```

3.1.2 Retrieving OUD Plug-In Configuration

The OUD plug-in configuration is available from the `PluginConfiguration` instance provided during plug-in initialization.The OUD plug-in configuration can be accessed by overriding the `initializePlugin` method.

The following example for ***Overriding initializePlugin to access the OUD plug-in configuration*** shows overriding the `initializePlugin` method.

```
@Override
  public void initializePlugin(PluginConfiguration configuration,PluginContext
context) throws PluginException
  {

    // Plugin configuration as a Set of properties
    Set<String> properties = configuration.getProperties();

    String aParameter=null;

    for(String value: properties)
    {
      if ( value.startsWith("customProperty=") )
      {
```

```
        aParameter = value.substring(value.indexOf("=")+1);
        break;
      }
    }

    // Expected property not found
    if ( aParameter == null )
    {
      throw new PluginException
                (context.getTypeBuilder().newMessage("customProperty missing in
configuration."));
    }

    // Either use the configuration right now or make it persistent using class
members.
  }
```

In this example, the configuration is retrieved from the raw configuration object as a set of properties. Once the properties are read, they can be used immediately and/or stored for later use in members of the Java class that implements the plug-in.

3.1.3 Creating an Automated Parser for Plug-In Properties

You can create an automated parser for plug-In properties as an alternative method to retrieve plug-in configuration. Follow these steps to create an automated parser for the plug-in properties.

To create an automated parser for the plug-in properties:

1. Create a Java interface that extends the class `oracle.oud.plugin.PluginConfiguration`.
2. For each property that you expect to be retrieved, add a getter in the form `get<property-name>()`. The *property-name* must match the key of the key-value pair defined in the plugin properties. The case of the name is ignored.
3. The returned method type must be of a class that provides a static method `valueOf(String) - java.lang.String.valueOf(String)` matches this assertion.

   The Java interface to parse the plug-in property `customProperty` looks like the following example for ***Parsing a Plug-in Property***.

```
public interface PropertyConfiguration
       extends oracle.oud.plugin.PluginConfiguration
{
  /**
   * Return the value associated to the key 'customProperty'.
   *
   * @return the value associated to the key 'customProperty'.
   */
  String getCustomProperty();
}
```

   Then the initialization of the plugin can be rewritten as in the following example:

```
  @Override
  public void initializePlugin(final PluginConfiguration configuration,
                               final PluginContext context)
    throws PluginException
```

```
  {
    super.initializePlugin(configuration, context);

    PropertyConfiguration propertyConfiguration =
this.getConfiguration(PropertyConfiguration.class);

    String customProperty = propertyConfiguration.getCustomProperty();
    // Perform check...
  }
```

3.1.4 Making Dynamic OUD Plug-In Configuration Changes

Changes to the plug-in configuration can be caught dynamically by overriding the method `handleConfigurationChange()`.

The new configuration can be retrieved as shown in the following example for ***Retrieving Changed Plug-In Configuration***.

```
@Override
  public void handleConfigurationChange(final PluginConfiguration configuration)
    throws PluginException
  {

    // The new configuration is stored in the configuration object
    // parse again the plugin configuration
    String aParameter;
    Set<String> properties = configuration.getProperties();

    for(String value: properties)
    {
      if ( value.startsWith("customProperty=") )
      {
        aParameter = value.substring(value.indexOf("=")+1);
        break;
      }
    }
  }
```

The `handleConfigurationChange()` method is invoked only when the plug-in properties managed by the OUD server are updated. If you decide to store the configuration in an external file, changes to the file content won't be detected dynamically by the mechanism described here.

3.1.5 Validating Plug-In Configuration

The `dsconfig` tool does not make any syntaxtical cases about the custom plug-in configuration properties, so the plug-in must validate the configuration at startup or when the configuration is modified dynamically.

The plug-in code should raise a `PluginException` when it cannot recover from an invalid configuration.

The plug-in is automatically disabled when a `PluginException` is raised during plug-in initialization. Invalid dynamic configuration changes can be rejected by raising a `PluginException` in the `handleConfigurationChange()` method.

3.2 Request Handling with OUD Plug-in API

With OUD plug-in API, you will be able to process OUD server LDAP requests, modifying search requests, forwarding requests, and returning requests.

The topics in this section include:

3.2.1 Overview of LDAP Request Handling with OUD Plug-in API

A plug-in can intercept any LDAP requests processed by the OUD server by implementing the corresponding callbacks defined by the `oracle.oud.RequestManager` interface. Each type of LDAP operation corresponds to a handler method. For example, `add` operations are managed by the `handleAdd()` method and so on.

Received LDAP requests are processed by the server. Thus modifying the properties of the requests can impact the server regarding performance, integrity, and security.

Each property contained in LDAP requests can be retrieved by getters, and modified by setters.

Each handler takes three parameters that are tied together:

- The LDAP request that contains all request properties as provided by the workflow element previous to this plug-in
- The Result handler that is the reference to use to return to the previous workflow element, the result of the LDAP request once processed
- A context that is a toolbox reference that provides access to various elements of the server such as logging subsystem, creation of plug-in API objects, client connection, abandon of request, and so forth

The `bind` request takes a fourth parameter that is the version of the LDAP protocol and that is provided for convenience only.The `abandon` and `unbind` methods cannot be intercepted. The `abandon` of a request can be detected using the request's context. The `unbind` operation means that the client connection will disconnect from the server.

The contract that must respect each plug-in in the process chain is to return the LDAP request in the exact state as it was received. This applies to all implementations of request handler. This is the most important thing that the plug-in does. This is important because although a request already has a result, the request may not be complete.

Consider this example: a plug-in is part of the processing that is performed after a load-balancer. Modifying the requests and giving back the modified request, instead of giving back the request in the state it was received, may make the load-balancer function improperly. Indeed, the request will be modified on the first route, and potentially replayed modified on the second route if the first route fails.

In summary, keep in mind that requests must be submitted in the exact same form as they are received.

3.2.2 Modifying OUD Search Requests with Plug-in API

OUD Search Requests are modified to change the scope of search request with Plug-in API .

The following example modifies the scope of a search request. The search scope is changed to `BASE_OBJECT`, and then restored when the search request has been processed.

1. To intercept the search requests, override the `handleSearch(...)` method in the example plug-in.

```
@Override
 public void handleSearch(final RequestContext requestContext,
                          final SearchRequest request,
                          final SearchResultHandler resultHandler)
   throws UnsupportedOperationException {

   System.out.println("plug-in: search received " + request);

   // Store the received search scope.
   SearchScope scopeReceived = request.getScope();

   // Set a base search scope for all search requests
   request.setScope(SearchScope.BASE_OBJECT);

   System.out.println("plug-in: search modified " + request);

   // Forward the request to the next plug-in.
   this.getConfiguration()
       .getFirstNextPlugin()
       .handleSearch(requestContext,
                     request,
                     resultHandler);

   // Restore the original value to give the request back as received.
   request.setScope(scopeReceived);
 }
```

2. Restart the Oracle Unified Directory instance for the JAR file changes to take effect.

   a. Stop the OUD instance.

   UNIX, Linux

```
$ cd instance-directory/OUD/bin
$ stop-ds
```

   Windows

```
C:\> cd instance-directory\OUD\bat
C:\> stop-ds
```

   b. Copy the plug-in JAR file into the `lib` directory.

UNIX, Linux

```
# cp plugin.jar lib
```

Windows

```
C:\> copy plugin.jar lib
```

c. Restart OUD instance.

UNIX, Linux

```
# start-ds
```

Windows

```
C:\> start-ds
```

3. Run the following command:

UNIX, Linux

```
$ ldapsearch --hostname localhost --port 1389 --bindDN "cn=directory manager" --
bindPasswordFile /tmp/password --searchScope sub --baseDN "uid=user.
1,ou=people,dc=example,dc=com" "(objectclass=*)"
```

Windows

```
C:\ ldapsearch --hostname localhost --port 1389 --bindDN "cn=directory manager"
--bindPasswordFile C:\tmp\password --searchScope sub --baseDN "uid=user.
1,ou=people,dc=example,dc=com" "(objectclass=*)"
```

4. For each command, the log file (*instance-dir*`/OUD/logs/server.out` on UNIX or Linux, *instance-dir*`\OUD\logs\server.out` on Windows) should contain information similar to this:

```
plug-in: search received SearchRequest(name=uid=user.
1,ou=people,dc=example,dc=com, scope=sub, dereferenceAliasesPolicy=never,
sizeLimit=0, timeLimit=0, typesOnly=false, filter=(objectClass=*),
attributes=[], controls=[])
plug-in: search modified SearchRequest(name=uid=user.
1,ou=people,dc=example,dc=com, scope=base, dereferenceAliasesPolicy=never,
sizeLimit=0, timeLimit=0, typesOnly=false, filter=(objectClass=*),
attributes=[], controls=[])
```

> **Note:**
>
> The request is passed to the next plug-in by calling:
>
> `this.getConfiguration().getFirstNextPlugin().handleSearch(...)`
>
> This is exactly what is done by the default implementation of the `AbstractPlugin`. The same thing can be achieved by calling the following:
>
> `super.handleSearch(...)`

3.2.3 Modifying Search Requests with Wrapper Object

An alternative way of modifying requests is to wrap the original request in a special object named wrapper. A request wrapper is an implementation that offers the same

exact Java interface as the request that it wraps, and then forwards all calls performed on methods to the wrapped request.

To modify the value of the properties, override the appropriate method. The following example demonstrates how to change the scope of search requests.

```
@Override
  public void handleSearch(final RequestContext requestContext,
                           final SearchRequest request,
                           final SearchResultHandler resultHandler)
    throws UnsupportedOperationException {

    SearchRequest newRequest = new SearchRequestWrapper(request)
    {
      @Override
      public SearchScope getScope()
      {
        // Change the scope of this request.
        return SearchScope.BASE_OBJECT;
      }
    };

    // Forward the request to the next plug-in.
    this.getConfiguration()
        .getFirstNextPlugin()
        .handleSearch(requestContext,
                      newRequest,
                      resultHandler);
  }
```

This alternative has the advantage of letting the wrapped request remain untouched. Thus, there is no need to restore the scope property as this one was not changed.

However, if you use this alternative, you may encounter problems. The wrapped request does not know about its outbound wrapper. If processing is performed at the level of the wrapped request, and this processing involves properties that are redefined at the level of the wrapper, then those properties will be ignored. The wrapped request only has access to its own properties.

A wrapper is provided for all types of requests in the `package oracle.oud.requests`.

3.2.4 Forwarding Requests with OUD Plug-in API

In most situations, plug-ins intercept requests, do some processing, then forward the request to the next workflow in the chain. In the vast majority of case, there is exactly one next workflow element. In this case, the request can be passed to the next element by invoking the corresponding method of the super instance.

In the case of a leaf plug-in, all request handlers implemented by `oracle.oud.AbstractPlugin` must be overridden, and a result must be returned as described in the next section.

In some specific cases, a plug-in may be followed by several workflow elements. The plug-in implementation must determine which workflow element the request must be forwarded to. The list of next workflow elements can be retrieved from the `PluginConfiguration` instance through the `getNextPlugins()` method. Then the request is forwarded to the appropriate workflow element by directly invoking the appropriate method as shown in the following example.

```
@Override
  public void handleBind(final RequestContext requestContext,
                         final int version,
                         final BindRequest request,
                         ResultHandler resultHandler)
    throws UnsupportedOperationException
  {

    // Get the original bind DN from the bind request
    DN originalDn = request.getName();

    // Transform the bind DN according to custom algorithm
    DN newDn = transformDN(originalDn);

    BindRequestWrapper wrapper = new BindRequestWrapper(request);

    // Update the wrapper object
    wrapper.setName(newDn);

    // Retrieve the list of next plugins and figure out which one to use
    List<Plugin> nextPlugins = this.getConfiguration().getNextPlugins();


    // Pass the request to the appropriate plugin (assume the first one here)
    nextPlugins.get(0).handleBind(requestContext,
                    version,
                    wrapper,
                    resultHandler);

 }
```

3.2.5 Returning Results with OUD Plug-in API

In some cases, a plug-in may intercept a request and return results by themselves instead of forwarding the request to the next workflow element of the chain.

A result object instance can be created using the `newResult()` method of the `oracle.oud.plugin.PluginContext.TypeBuilder` class. Then this result can be returned to the plug-in caller by invoking the `handleResult()` or `handleErrorResult()` method from the `resultHandler` object passed as an argument of the handler methods. The following example illustrates how to intercept `bind` requests and return an `Invalid Credentials` error.

```
@Override
  public void handleBind(final RequestContext requestContext,
                         final int version,
                         final BindRequest request,
                         ResultHandler resultHandler)
    throws UnsupportedOperationException
  {

    // Get the original bind DN from the bind request
    DN originalDn = request.getName();

    // Apply custom logic to decide whether access is granted or not
    // Assume invalid credentials


    // Create a Result object
    Result error =
```

```
getPluginContext().getTypeBuilder().newResult(ResultCode.INVALID_CREDENTIALS);

    // Return it to the plugin caller
    resultHandler.handleErrorResult(error);

 }
```

Similarly, LDAP entries can be created using the `newSearchResultEntry()` method of the `oracle.oud.plugin.PluginContext.TypeBuilder` class. Then this entry can be returned to the plug-in caller by invoking the `handleSearchResultEntry()` or `handleErrorResult()` method from the `searchResultHandler` object passed as an argument of the `handleSearch()` method.

3.3 Handling Responses in OUD Plug-in

Plug-ins that need to intercept responses must explicitly register their interest by providing their own `ResultHandler` instance before submitting the request to the next workflow element. The `handleResult()` method of the `ResultHandler` is invoked upon successful completion of the operation with the corresponding `Result` instance passed in argument. Conversely, the `handleErrorResult()` of the `ResultHandler` is invoked when an error occurred.

The custom `ResultHandler` implementation can examine the result and modify it, but it is responsible for invoking the appropriate method (`handleResult()` or `handleErrorResult()`) of the original `ErrorHandler` to pass the result to the calling workflow element. For simplicity, you should implement custom `ResultHandler` as a specialization of the `DefaultResultHandler objectclass`. By default, results and errors are passed to the workflow element upstream in the chain, and only the appropriate methods need to be overridden by the plug-in implementation.

Similarly, `SearchResultHandler` must be used for search operations, to intercept both final search result and search entries. The `handleEntry()` method is invoked every time an LDAP entry is returned by the next workflow elements. The custom `SearchResultHandler` implementation must invoke the `handleEntry()` method of the original `SearchResultHandler` to send the entry up the chain.

3.3.1 Example for Intercepting bind failure

OUD plug-in intercepts responses and bind failure is one such type of response.

In the following example, the plug-in intercepts bind failure only.

```
@Override
  public void handleBind(final RequestContext requestContext,
                         final int version,
                         final BindRequest request,
                         ResultHandler resultHandler)
    throws UnsupportedOperationException
  {

    // Create a new ResultHandler to intercept bind result
    CustomResultHandler customBindHandler = new CustomResultHandler(resultHandler);

    // Pass the request to the next plug-in with the custom ResultHandler
    super.handleBind(requestContext,
                     version,
                     request,
```

```
                    customBindHandler);

 }

// implementation of a custom ResultHandler to intercept errors

private class CustomResultHandler
          extends DefaultResultHandler
{

    public CustomResultHandler(ResultHandler resultHandler)
    {
    super(resultHandler);
    }

    @Override
    public void handleErrorResult(Result error)
    {

    // Invoked when Bind fails
    // Examine the result and implement some logic
    // Pass the result up the chain

    super.handleErrorResult(error);
    }

}
```

3.3.2 Example for Intercepting Search Entries and Final Search Results

OUD plug-in intercepts responses and intercepting search entries and the final search result is one such type of response.

The following example intercepts search entries and the final search results.

```
@Override
    public void handleSearch(final RequestContext requestContext,
                                    final SearchRequest request,
                                    SearchResultHandler resultHandler)
       throws UnsupportedOperationException
           {

           // Create a new SearchResultHandler to intercept search entries
           // and result
           CustomSearchResultHandler customHandler = new
CustomSearchResultHandler(resultHandler);

// Pass the request to the next plug-in with the custom ResultHandler
super.handleSearch(requestContext,
                   request,
                   customHandler);
  }

// implementation of a custom SearchResultHandler to intercept entries and errors
private class CustomSearchResultHandler
extends DefaultSearchResultHandler

{
```

```
public CustomSearchResultHandler(SearchResultHandler resultHandler)
{
  super(resultHandler);
}

@Override
public void handleErrorResult(Result error)
{
// Invoked when Search fails
// Examine the result and implement some logic
// Pass the result up the chain
super.handleErrorResult(error);
}

@Override
public void handleResult(Result result)
{
// Invoked when Search complete
// Examine the result and implement some logic
// Pass the result up the chain
super.handleResult(result);
}

@Override
public boolean handleEntry(SearchResultEntry entry)
{
// Invoked for every search entry to be returned
// Examine the result and implement some logic
// Pass the entry up the chain
return super.handleEntry(entry);
}
}
```

3.4 About Results Handling in OUD Plug-in

Request results are returned using objects called a result handler. All LDAP operations share the same kind of result except the search operation. The search operation has additional results that are entries and references. An LDAP operation is composed of a pair: a request and a result-handler.

The request is used to access the properties of the request. The result handler is used to post the result of the request that has been processed to the previous plug-in.

The topics in this section include:

- Ignoring Search Results in OUD Plug-in
- Intercepting Search Failures in OUD Plug-in
- Counting Entries Returned by Search Requests

3.4.1 Ignoring Search Results in OUD Plug-in

You have to ignore search results in situations, where the plug-in itself is skipped and the results returned by the next plug-in will be passed directly from the next plug-in to the previous plug-in.

In the following example, the result handler provided by the previous plug-in is passed directly to the next plug-in. The consequence is that the results returned by the next

plug-in will be passed directly from the next plug-in to the previous plug-in, skipping the plug-in itself. The only way to detect that the request was processed is by returning from the `handlerSearch(...)` call.

```
@Override
  public void handleSearch(final RequestContext requestContext,
                           final SearchRequest request,
                           final SearchResultHandler resultHandler)
    throws UnsupportedOperationException
  {
    // Pass the resultHandler reference received from the previous plug-in to
    // the next plug-in. This implies that the next plug-in will post the
    // result of the search request directly to the previous plug-in.
    this.getConfiguration()
        .getFirstNextPlugin()
        .handleSearch(requestContext,
                      request,
                      resultHandler);

    // The search request was processed by next plug-in.
  }
```

3.4.2 Intercepting Search Failures in OUD Plug-in

To intercept results returned by a subsequent plug-in, the plug-in must provide its own result handler.

A result handler defines two methods:

- `handleResult(Result)` called by the next plug-in when the request was successful
- `handleErrorResult(Result)` called by the next plug-in when the request was unsuccessful

A search result handler defines two additional methods. These methods must return `True` to specify that the next plug-in can still return other entries or references, or `False` to indicate to the next plug-in that no more entries or references are expected. For example, no more entries or references are expected when the size limit reached.

- `handleEntry(SearchResultEntry)` returned by the next plug-in when an entry is returned
- `handleReference(DN, SearchResultReference)` returned by the next plug-in when a reference is returned

The OUD plug-in API provides a default implementation named `oracle.oud.plugin.DefaultResultHandler` for implementing result handlers. This Java class wraps a result handler (in most cases the result handler provided by the previous plug-in) and by default forwards the received result to the wrapped result handler. To capture a result, a plug-in must override the kind of result it is interested in. A similar default implementation exists for search result handler: `oracle.oud.plugin.DefaultSearchResultHandler`.

The following example shows how to log the result in case the request is unsuccessful.

```
public class EchoErrorResultHandler
        extends DefaultResultHandler
 {
   public EchoErrorResultHandler(ResultHandler resultHandler)
   {
     super(resultHandler);
```

```
    }

    @Override
    public void handleErrorResult(Result error)
    {
      // Echo the result of the request.
      System.out.println("plug-in: error result " + error);

      // Let the default behavior forward the result to the wrapped result
      // handler
      super.handleErrorResult(error);
    }
  }
```

The following example illustrates how to make search operations print out the results in case the request is not successful.

```
@Override
  public void handleSearch(final RequestContext requestContext,
                           final SearchRequest request,
                           final SearchResultHandler resultHandler)
    throws UnsupportedOperationException
  {
    // The result handler passed to the next plug-in will echo the result in
    // case the request was not successful.
    this.getConfiguration()
        .getFirstNextPlugin()
        .handleSearch(requestContext,
                      request,
                      new EchoErrorResultHandler(resultHandler));

    // The search request was processed by next plug-in.
  }
```

Notice the following:

- The result handler is not associated to the request. It is up to the developer to maintain the association by keeping a reference to the request inside the implementation of the result handler.
- A new instance of the custom result handler is required for each instance of received request.

3.4.2.1 Logging the Failures of Search Requests

To log the failures of search requests:

1. Change the example plug-in as shown above.
2. Restart the Oracle Unified Directory instance for the JAR file changes to take effect.
   a. Stop the OUD instance.

      UNIX, Linux

      ```
      $ cd instance-directory/OUD/bin
      $ stop-ds
      ```

Windows

```
C:\> cd instance-directory\OUD\bat
C:\> stop-ds
```

b. Copy the plug-in JAR file into the `lib` directory.

UNIX, Linux

```
# cp plugin.jar lib
```

Windows

```
C:\> copy plugin.jar lib
```

c. Restart OUD instance.

UNIX, Linux

```
# start-ds
```

Windows

```
C:\> start-ds
```

3. Run the following command to search for a user that does not exist.

UNIX, Linux

```
ldapsearch --hostname localhost --port 1389 --bindDN "cn=directory manager" --
bindPasswordFile /tmp/password --searchScope sub --baseDN
"uid=user.unknown,ou=people,dc=example,dc=com" "(objectclass=*)"
the command displays
SEARCH operation failed
Result Code:  32 (No Such Entry)
Additional Information:  The search base entry
'uid=user.unknown,ou=people,dc=example,dc=com' does not exist
Matched DN:  ou=people,dc=example,dc=com
```

Windows

```
ldapsearch --hostname localhost --port 1389 --bindDN "cn=directory manager" --
bindPasswordFile C:\tmp\password  --searchScope sub --baseDN
"uid=user.unknown,ou=people,dc=example,dc=com" "(objectclass=*)"
the command displays
SEARCH operation failed
Result Code:  32 (No Such Entry)
Additional Information:  The search base entry
'uid=user.unknown,ou=people,dc=example,dc=com' does not exist
Matched DN:  ou=people,dc=example,dc=com
```

For each command, the log file (*instance-dir*`/OUD/logs/server.out` on UNIX, Linux or *instance-dir*`\OUD\logs\server.out` on Windows) should contain information similar to the following:

```
plug-in: error result Result(resultCode="No Such Entry",
matchedDN="ou=people,dc=example,dc=com", diagnosticMessage="The search base
entry 'uid=user.unknown,ou=people,dc=example,dc=com' does not exist",
referrals=null, controls=[])
```

3.4.3 Counting Entries Returned by Search Requests

Request results are returned using objects called a result handler. You can count the number of entries by `EntryCounterResultHandler`.

The following example counts the number of entries returned by search requests, and then logs it. The `EntryCounterResultHandler` increments a counter each time the `handleEntry(...)` method is called.

```
public class EntryCounterResultHandler
       extends DefaultSearchResultHandler
{
  // The number of search result entries returned by this search result
  // handler.
  private int entriesCount;

  public EntryCounterResultHandler(SearchResultHandler resultHandler)
  {
    super(resultHandler);
  }

  @Override
  public boolean handleEntry(SearchResultEntry entry)
  {
    this.entriesCount++;

    return super.handleEntry(entry);
  }

  public int getEntriesCount()
  {
    return this.entriesCount;
  }
}
```

The search request handler is modified to pass a result handler that counts returned entries for each search request processed. Once the request processed by the next plug-in, the number of returned entries is logged. See the following example.

```
@Override
  public void handleSearch(final RequestContext requestContext,
                           final SearchRequest request,
                           final SearchResultHandler resultHandler)
    throws UnsupportedOperationException
  {
    EntryCounterResultHandler counter =
                                  new EntryCounterResultHandler(resultHandler);

    // The result handler passed to the next plug-in will count the number of
    // entries returned by the next plug-in.
    this.getConfiguration()
        .getFirstNextPlugin()
        .handleSearch(requestContext,
                      request,
                      counter);

    // The search request was processed by next plug-in.
    System.out.println(String.format("plug-in: request %s returned %d entries",
                                     request,
```

```
                                                counter.getEntriesCount()));
    }
```

3.4.3.1 Logging the Number of Returned Entries of Search Requests

To log the number of returned entries of search requests:

1. Change the example plug-in as shown in the example shown above.
2. Restart the Oracle Unified Directory instance for the JAR file changes to take effect.
   a. Stop the OUD instance.

      UNIX, Linux

      ```
      $ cd instance-directory/OUD/bin
      $ stop-ds
      ```

      Windows

      ```
      C:\> cd instance-directory\OUD\bat
      C:\> stop-ds
      ```

   b. Copy the plug-in JAR file into the `lib` directory.

      UNIX, Linux

      ```
      # cp plugin.jar lib
      ```

      Windows

      ```
      C:\> copy plugin.jar lib
      ```

   c. Restart OUD instance.

      UNIX, Linux

      ```
      # start-ds
      ```

      Windows

      ```
      C:\> start-ds
      ```

3. Run the following command to display all users registered:

   UNIX, Linux

   ```
   ldapsearch --hostname localhost --port 1389 --bindDN "cn=directory manager" --
   bindPasswordFile /tmp/password --searchScope sub --baseDN
   "ou=people,dc=example,dc=com" "(objectclass=*)"
   ```

   Windows

   ```
   ldapsearch --hostname localhost --port 1389 --bindDN "cn=directory manager" --
   bindPasswordFile C:\tmp\password --searchScope sub --baseDN
   "ou=people,dc=example,dc=com" "(objectclass=*)"
   ```

For each command, the log file (*instance-dir*`/OUD/logs/server.out` on UNIX or Linux, *instance-dir*`\OUD\logs\server.out` on Windows) should contain information similar to this:

```
plug-in: request SearchRequest(name=ou=people,dc=example,dc=com, scope=sub,
dereferenceAliasesPolicy=never, sizeLimit=0, timeLimit=0, typesOnly=false,
filter=(objectClass=*), attributes=[], controls=[]) returned 51 entries
```

```
the number of returned entries corresponds to the 50 users plus the entry
ou=people,dc=example,dc=com
```

3.5 Configuring Filters in Search Requests

Sometimes applications need to interact with filters contained in search requests. This interaction is specified by a mechanism based on the visitor design pattern and defined by the Java interface `oracle.oud.types.FilterVisitor<R,P>`.

The topics in this section include:

- About Filter Processing in Search Requests
- Example of Implementation of the `FilterVisitor`
- Example for Verifying and Logging the Presence of `objectclass=*` in a Search Request
- Verifying and Logging Presence of `objectclass=*` in a Search Request

3.5.1 About Filter Processing in Search Requests

The LDAP protocol specifies ten types of filters: `and`, `or`, `not`, `equalityMatch`, `substrings`, `greaterOrEqual`, `lessOrEqual`, `present`, `approxMatch` and `extensibleMatch`.

The filter visitor defines a handler for each type of filter: `visitAndFilter(...)`, `visitOrFilter(...)` and so on.

When a filter is parsed, the types of filters that compose the filter to parse are identified. The visitor methods associated to the identified types are called in sequence.

The `FilterVisitor<R,P>` takes two parameters:

- `<R>` is the returned type of each visitor handler.
- `<P>` is a parameter that can be provided to each visitor handler.

3.5.2 Example of Implementation of the `FilterVisitor`

`FilterVisitor` checks the presence of an attribute in the filter to parse.

The following example provides an implementation of the `FilterVisitor`. An attribute is present in a filter if it is associated to the value `*` such as `objectclass=*`. In that case, `<R>` corresponds to the result of the evaluation. `<R>` is defined as a Boolean that has the value `TRUE` if the attribute is present in the filter to parse, and `FALSE` if the attribute is absent. `<P>` is the parameter and corresponds to a String that defines which attribute must to be checked.If the filter to parse is `objectclass=*`, then calling the visitor with the parameter `objectclass` will return `TRUE`. Other values will return `FALSE`.

Visitors that are composed of sub-filters (`and`, `or` and `not`) forward the check by visiting all the sub-filters they are composed of.

In the following example, for illustration purposes, the relevant visitors also log some information.

```
private class PresenceOfFilterVisitor
          implements FilterVisitor<Boolean,
                                   String>
```

```
  {
    @Override
    public Boolean visitAndFilter(final String presenceName,
                                  final List<Filter> subFilters)
    {
      System.out.println("plug-in: visit AND with " + subFilters);

      boolean result = false;

      // Iterate through all sub filters with this filter visitor.
      for(Filter subFilter: subFilters)
      {
        result = subFilter.accept(this, presenceName);

        if ( result )
        {
          break;
        }
      }

      return result ? Boolean.TRUE : Boolean.FALSE;
    }

    @Override
    public Boolean visitApproxMatchFilter(final String presenceName,
                                          final String attributeDescription,
                                          final ByteString assertionValue)
    {
      return Boolean.FALSE;
    }

    @Override
    public Boolean visitEqualityMatchFilter(final String presenceName,
                                            final String attributeDescription,
                                            final ByteString assertionValue)
    {
      System.out.println("plug-in: visit EQUAL with " + attributeDescription + "=" +
assertionValue);

      return Boolean.FALSE;
    }

    @Override
    public Boolean visitExtensibleMatchFilter(final String presenceName,
                                              final String matchingRule,
                                              final String attributeDescription,
                                              final ByteString assertionValue,
                                              final boolean dnAttributes)
    {
      return Boolean.FALSE;
    }

    @Override
    public Boolean visitGreaterOrEqualFilter(final String presenceName,
                                             final String attributeDescription,
                                             final ByteString assertionValue)
    {
      return Boolean.FALSE;
    }

    @Override
```

```
    public Boolean visitLessOrEqualFilter(final String presenceName,
                                          final String attributeDescription,
                                          final ByteString assertionValue)
    {
      return Boolean.FALSE;
    }

    @Override
    public Boolean visitNotFilter(final String presenceName,
                                  final Filter subFilter)
    {
      System.out.println("plug-in: visit NOT with " + subFilter);

      // Visit the associated filter with this filter visitor.
      return subFilter.accept(this, presenceName);
    }

    @Override
    public Boolean visitOrFilter(final String presenceName,
                                 final List<Filter> subFilters)
    {
      System.out.println("plug-in: visit OR with " + subFilters);

      boolean result = false;

      // Iterate through all sub filters with this filter visitor.
      for(Filter subFilter: subFilters)
      {
        result = subFilter.accept(this, presenceName);

        if ( result )
        {
          break;
        }
      }

return result ? Boolean.TRUE : Boolean.FALSE;
    }

    @Override
    public Boolean visitPresentFilter(final String presenceName,
                                      final String attributeDescription)
    {
      System.out.println("plug-in: visit Presence with '" + attributeDescription +
"'");

      return presenceName.equalsIgnoreCase(attributeDescription) ? Boolean.TRUE
                                                                 : Boolean.FALSE;
    }

    @Override
    public Boolean visitSubstringsFilter(final String presenceName,
                                         final String attributeDescription,
                                         final ByteString initialSubstring,
                                         final List<ByteString> anySubstrings,
                                         final ByteString finalSubstring)
    {
      return Boolean.FALSE;
    }

    @Override
```

```
  public Boolean visitUnrecognizedFilter(final String presenceName,
                                         final byte filterTag,
                                         final ByteString filterBytes)
  {
    return Boolean.FALSE;
  }
}
```

3.5.3 Example for Verifying and Logging the Presence of `objectclass=*` in a Search Request

You can verify and log `objectclass=*` in a search request filter processed by the plug-in.

The following example verifies and logs the presence of `objectclass=*` in a search request filter.

```
@Override
  public void handleSearch(final RequestContext requestContext,
                           final SearchRequest request,
                           final SearchResultHandler resultHandler)
    throws UnsupportedOperationException
  {
    Filter filter = request.getFilter();

    System.out.println("plug-in: visitor returned "
                       + filter.accept(new PresenceOfFilterVisitor(),
                                       "objectclass"));

    // Pass the resultHandler reference received from the previous plug-in to
    // the next plug-in. This implies that the next plug-in will post the
    // result of the search request directly to the previous plug-in.
    super.handleSearch(requestContext,
                       request,
                       resultHandler);

    // The search request was processed by next plug-in.
  }
```

3.5.4 Verifying and Logging Presence of `objectclass=*` in a Search Request

You can verify and log the presence of `objectclass=*` in a search request filter processed by the plug-in.

1. Change the example plug-in as shown in **Example for Verifying and Logging the Presence of `objectclass=*` in a Search Request**.
2. Restart the Oracle Unified Directory instance for the JAR file changes to take effect.
   a. Stop the OUD instance.

      UNIX, Linux

      ```
      $ cd instance-directory/OUD/bin
      $ stop-ds
      ```

Windows

```
C:\> cd instance-directory\OUD\bat
C:\> stop-ds
```

b. Copy the plug-in JAR file into the `lib` directory.

UNIX, Linux

```
# cp plugin.jar lib
```

Windows

```
C:\> copy plugin.jar lib
```

c. Restart OUD instance.

UNIX, Linux

```
# start-ds
```

Windows

```
C:\> start-ds
```

3. Run the following command to display all users registered:

UNIX, Linux

```
ldapsearch --hostname localhost --port 1389 --bindDN "cn=directory manager" --
bindPasswordFile /tmp/password --searchScope sub --baseDN
"ou=people,dc=example,dc=com" "(objectclass=*)"
```

Windows

```
ldapsearch --hostname localhost --port 1389 --bindDN "cn=directory manager" --
bindPasswordFile C:\tmp\password --searchScope sub --baseDN
"ou=people,dc=example,dc=com" "(objectclass=*)"
```

For each command, the log file (*instance-dir*/`OUD/logs/server.out` on UNIX or Linux, *instance-dir*/`OUD/logs/server.out` on Windows) should contain similar to the following:

```
plug-in: visit Presence with 'objectClass' plug-in: visitor returned true
```

Running the command with different filters shows how the visitor mechanism works. Searching with the filter `&(|(!(uid=user.1)))` logs the following:

```
plugin: visit AND with [(|(!(uid=user.1)))]
    plugin: visit OR with [(!(uid=user.1))]
    plugin: visit NOT with (uid=user.1)
    plugin: visit EQUAL with uid=user.1
    plugin: visitor returned false
```

3.6 Configuring Internal Operations in OUD Plug-in API

The API provides methods to make LDAP-like calls into the OUD. In first method, plug-in makes calls within the current plug-in workflow and in second method, the plug-in makes LDAP-like calls into the OUD.

Configuring internal operations in OUD plug-in is described in the following section:

- About Internal LDAP Requests

- Understanding OUD Plug-in API Internal Requests

3.6.1 About Internal LDAP Requests

Internal LDAP requests are internal, that are not initiated directly by external requests from clients, but internally by plug-ins. Use internal request calls when your plug-in needs OUD to perform an operation for which no client request exists. For instance, a plug-in can do a search request to the user entry to retrieve additional credentials upon reception of a bind request from a client

`oracle.oud.plugin.RequestManager` callbacks are invoked for every operation processed by OUD, including internal operation. In many cases, plug-ins apply to operations directly initiated by a client application only. It is possible to make distinction between internal operation and regular operation by calling the `isInternal()` method on the request object.

Internal LDAP requests are created through the `oracle.oud.plugin RequestBuilder` objectclass. A reference to a `requestBuilder` can be retrieved from a `RequestContext` associated with a request received from a client application through the `getRequestBuilder()` method.

The user credentials used to perform an internal operation is specified at creation time. In general, internal operations are performed within the current a security context, with the credentials of the user which triggered the plug-in. In some situations, internal operations require privileged access. For instance, an internal search performed before handing a bind request will be performed as `anonymous` because at that point of time, the current user is not authenticated yet.

3.6.1.1 Creating Internal LDAP Requests

To create a privilege request, use a privilege `RequestBuilder` with the call `requestContext.getRequestBuilder(true)`. Only requests created from this builder can be performed with privileges of `root.otherwise`, and get a default `requestBuilder` through `requestContext.getRequestBuilder(false)`.

3.6.2 Understanding OUD Plug-in API Internal Requests

The OUD plug-in API provides two ways to invoke internal requests. In the first mode, the plug-in makes calls within the current plug-in workflow by invoking the appropriate subsequent workflow element configured in the chain if any. In the second mode, the plug-in makes LDAP-like calls into the OUD as though they were coming from an end client.

Each call offers the ability to let the router select appropriate workflow for the operation.

Results from internal requests can be retrieved using result handlers, as described in About Results Handling in OUD Plug-in.

- About Mode 1 of the OUD Plug-in API
- Implementing Mode 1 of the OUD Plug-in API
- About Mode 2 of the OUD Plug-in API
- Implementing Mode 2 of the OUD Plug-in API

3.6.2.1 About Mode 1 of the OUD Plug-in API

The next workflow elements of a plug-in can be retrieved from the plug-in configuration through the call `configuration.getNextPlugins()`. The name of these plug-ins can be retrieved using the `getName()` method. After retrieving the name, you can select which workflow element the request must be sent to in the situation where there are more than one next workflow element configured. For instance, a plug-in providing a load-balancing service would probably have several subsequent workflow elements configured. Once the internal request is instantiated, and the target workflow element is located, the request can be submitted via the appropriate handler method.

3.6.2.2 Implementing Mode 1 of the OUD Plug-in API

The following example requires the server schema to be modified to accept the attribute `customTimeStamp`. The plug-in uses an internal `modify` operation to store the login time in the user entry attribute `customTimeStamp`.

Notice that a privilege request builder `getRequestBuilder(true)` must be used because at that point of the processing, the bind is not yet completed. So the user is considered to be anonymous.

```
@Override
public void handleBind(final RequestContext requestContext,
                       final int version,
                       final BindRequest request,
                       ResultHandler resultHandler)
    throws UnsupportedOperationException
  {

    ...

    // Get a privileged request builder
    RequestBuilder myRequestBuilder = requestContext.getRequestBuilder(true);

    // Create a new modify request using that builder
    // Target LDAP entry is the user about to be authenticated
    ModifyRequest addTimestampModifyRequest =
myRequestBuilder.newModifyRequest(request.getName());

    // Populate the modification object
    addTimestampModifyRequest.addModification(ModificationType.REPLACE,
        "customTimeStamp", System.currentTimeMillis()) ;

    // Create a ResultHandler to catch the result of the modify operation
    ResultHandler modifyResultHandler = new CustomModifyResultHandler(resultHandler);

    // submit the request to the next workflow element
    getConfiguration().getFirstNextPlugins().handleModify(requestContext,
        addTimestampModifyRequest, modifyResultHandler);

           ...

  }
```

3.6.2.3 About Mode 2 of the OUD Plug-in API

In this mode, the request is performed through an internal request manager object. This object can be obtained from a `RequestContext` through the method `getInternalRequestManager()`. Then the request can be submitted through the appropriate handler method.

Each request is subject to routing to the appropriate workflow, so an internal request initiated by a plug-in within a given workflow may be routed to the same workflow. There are situations where a plug-in can intercept requests it generated by itself. To prevent unexpected recursive loops in the internal operation processing, it is possible to attach an additional attachment (contextual information) to an internal operation when it is submitted. This attachment can be retrieved and checked by the proxy upon reception of a new request to detect loops and take the appropriate action. Attachments can be managed via the `AttachmentHolder` interface implemented by the Request objects.

3.6.2.4 Implementing Mode 2 of the OUD Plug-in API

The following example searches for the `customTimeStamp` attribute in the entry of a user before a modification. A modify request is created with the current user credentials and submitted through the internal request manager as if it was coming from an end client. For clarity, exception handling was removed from the code example.

```
public void handleModify(final RequestContext requestContext,
                         final ModifyRequest request,
                         ResultHandler resultHandler)
    throws UnsupportedOperationException
  {
    ...

    // Get a standard request builder
    RequestBuilder myRequestBuilder = requestContext.getRequestBuilder(false);

    // Create a new search request using that builder
    // Target LDAP entry is the user about to be modified
    SearchRequest getLastTimestampRequest = myRequestBuilder.newSearchRequest(
        request.getName(), SearchScope.BASE_OBJECT,
        getPluginContext().getTypeBuilder().newFilter("(objectclass=*)"),
        "currentTimeStamp");

    // Create a ResultHandler to catch the result of the search operation
    SearchResultHandler searchResultHandler = new
CustomSearchResultHandler(resultHandler);

    // submit the request via the internal request manager
    requestContext.getInternalRequestManager().handleSearch(requestContext,
        getLastTimestampRequest, searchResultHandler);

          ...
  }
```

The following example shows how to deal with loops. The first time a search request is received by the plug-in, it has no attachment with name `nbLoops`. The plug-in flags the request with an attachment (`name=nbLoops`, `value=1`), then rebalance the request to the internal request manager. The search request will eventually come back to the plug-in. The second time the plug-in gets the attachment, increment the value to `2` and set the

attainment to the request. Then rebalance it to the internal request manager. The third time, since the value (2) is greater or equal to `MAX_LOOPS`, the plug-in will send the request to the next Workflow element (with `method super.handleSearch(...)`

```
// Let search requests loop 2 times within the internal request manager,
// before sending them to next WorkflowElement
public static final int MAX_LOOPS = 2;

  @Override
  public void handleSearch(RequestContext requestContext,
                                          SearchRequest request,
                                          SearchResultHandler resultHandler)
      throws UnsupportedOperationException
  {
    String name = "nbLoops";
    Integer nbLoops = 0;
    Set<String> attachmentNames = request.getAttachmentNames();

    // Get "nbLoops" attachment value, if ound in the request
    if (attachmentNames.contains(name))
    {
      nbLoops = (Integer) request.getAttachment(name);
    }

    // if we reach max number of loops...
    if (nbLoops >= MAX_LOOPS)
    {
      // ...remove attachment
      request.removeAttachment(name);

      // forward request to next WorkflowElement
      super.handleSearch(requestContext,
                         request,
                         resultHandler);
    } else
   {
      // increment nbLoops value
      nbLoops++;

      // set attachment nbLoops new value
      request.setAttachment(name, nbLoops);

      // log request (as internal op) + attachment value
      Logger logger = requestContext.getLogger();
      HashMap<String, String> map = new HashMap<String, String>();
      map.put("nbLoops", Integer.toString(nbLoops));
      logger.logSearchRequestIntermediateMessage(request, map);

      // re-balance tge search request via the internal request manager
      requestContext.getInternalRequestManager().handleSearch(requestContext,
                                                               request,
                                                               resultHandler);
    }
```

3.7 About OUD Plug-in Exceptions

Plug-in implementation can raise the subclass of `PluginException` when unexpected error conditions occur.

The behavior of the server depends on when the exception is raised. When raised during LDAP operation processing, a LDAP error 80 "Internal Error" is returned to the client application. When raised during plug-in initialization, the plug-in is disabled.

3.8 Logging and Debugging Exceptions in the OUD Plug-in API

You can handle logging and debugging exceptions in OUD plug-in API. `oracle.oud.plugin.RequestContext.Logger` interface is used to log a message in the OUD.

The topics in this section include:

- About Logging and Debugging Exceptions in the OUD Plug-in API
- Debugging the Plug-In When Servicing a Client Request
- Debugging Plug-In Initialization

3.8.1 About Logging and Debugging Exceptions in the OUD Plug-in API

Uncaught exceptions generated within the plug-in API are logged in the OUD debug log with the `Warning` level.

The standard output of the plug-in is redirected to the log file (*instance-dir*`/OUD/logs/debug` on UNIX or Linux, *instance-dir*`\OUD\logs\debug` on Windows) present in the OUD directory server instance hosting the plug-in.

During plug-in development you can enable the debug log using the following `dsconfig` command:

```
dsconfig set-log-publisher-prop \ --publisher-name "File-Based Debug Logger" \ --set
default-debug-level:warning \ --set enabled:true
```

The plug-in implementation can log a message in the OUD `access`, `error`, or `debug` log using the `oracle.oud.plugin.RequestContext.Logger` interface.

3.8.2 Debugging the Plug-In When Servicing a Client Request

Unexpected error conditions occur during implementation of the plug-in. You need to debug the plug-in when servicing a client request through an IDE.

Follow these steps to debug the plug-in when servicing a client request:

1. Export `OPENDS_JAVA_ARGS` with the value of `start-ds.java-args` taken from *instance-directory*`/config/java.properties` plus `-Xdebug -Xrunjdwp:transport=dt_socket,address=127.0.0.1:8888,server=y,suspend=n`
2. Restart the OUD instance.

   This will open the debug port 8888.
3. Attach to the OUD process on port 8888, and debug the plug-in through an IDE.

3.8.3 Debugging Plug-In Initialization

Unexpected error conditions occur during implementation of the plug-in. You need to debug plug-in initialization.

Follow these steps to debug plug-in initialization:

1. Export `OPENDS_JAVA_ARGS` with the value of `start-ds.java-args` taken from *instance-directory*`/config/java.properties` plus `-Xdebug -Xrunjdwp:transport=dt_socket,address=127.0.0.1:8888,server=y,suspend=y`
2. Restart the OUD instance.

   This will open the debug port 8888.
3. At this point, you *must* attach three times to the OUD process on port 8888 before you can debug the plug-in initialization code (using the `pluginInitialization()` method).

You should export `OPENDS_JAVA_ARGS` rather than modify the `java.properties` file. Exporting `OPENDS_JAVA_ARGS` does not require you to change the OUD instance configuration files, posing no risk to exporting the debug JVM `args` in production.