SAP NetWeaver® Identity Management Connector Development Kit

Implementing the Virtual Directory Server Connector

Version 7.2 Rev 1

Preface

The product

The SAP NetWeaver Identity Management Connector Development Kit enables independent software vendors (ISVs) or SAP partners to create an Identity Management connector for their application, and to integrate the application into the Identity Management landscape.

The Connector Development Kit contains information necessary for development of an Identity Management connector – criteria, guidelines, templates, test tool, certification guide, etc.

The reader

This manual is written for people who wish to implement the Virtual Directory Server connector/Identity Management connector for their application.

Prerequisites

To get the most benefit from this manual, you should have the following knowledge:

- Thorough knowledge of the Identity Center.
- Thorough knowledge of the Virtual Directory Server.
- Java programming skills.
- Basic knowledge of LDAP.
- Knowledge and understanding of the target application.

The following software is required:

- SAP NetWeaver Identity Management Virtual Directory Server version 7.1 SP2 or newer, or version 7.2 or newer installed and licensed.
- SAP NetWeaver Identity Management Identity Center version 7.1 SP2 or newer, or version 7.2 or newer installed and licensed (including the Identity Management User Interface installed and configured). At least one dispatcher has been configured and is running.
- SAP Provisioning Framework (the provisioning framework for SAP systems).
- A Java development environment. This can be downloaded from http://java.sun.com (version 1.4/1.5).
- Access to the target application.
- Access to the target application Java library (API).

The manual

This manual gives an overview of the connector methods, parameters and the Virtual Directory Server structures and shows how to configure and implement the application integration code and the application API in the Virtual Directory Server.

Related documents

You can find useful information in the following documents:

- *SAP NetWeaver Identity Management Security Guide*.
- *SAP NetWeaver Identity Management Operations Guide.*
- *Help files for the Virtual Directory Server.*
- *Java Help for the Virtual Directory Server Connector API.*
- *SAP NetWeaver Identity Management Connector Development Kit Overview.*
- *SAP NetWeaver Identity Management Connector Development Kit Virtual Directory Server Connector Testing Tool*.
- *SAP NetWeaver Identity Management Connector Development Kit Certification.*

Virtual Directory Server tutorials:

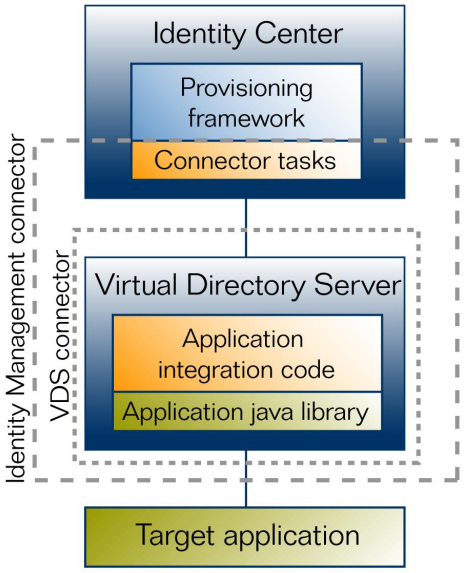
- *SAP NetWeaver Identity Management Virtual Directory Server Tutorial Accessing databases.*
- *SAP NetWeaver Identity Management Virtual Directory Server Tutorial Accessing LDAP servers.*
- *SAP NetWeaver Identity Management Virtual Directory Server Tutorial Joining data sources.*
- *SAP NetWeaver Identity Management Virtual Directory Server Tutorial Performing dynamic add operations.*

Table of contents

Introduction

The SAP NetWeaver Identity Management is a general purpose identity management application which provides the functions and services needed to integrate distributed identity data in the system landscape to efficient, heterogeneous identity lifecycle management.

The purpose of the Connector Development Kit (CDK) is to enable the independent software vendors (ISVs) or SAP partners to create an Identity Management connector for their application, and to integrate the application in the Identity Management landscape.



The processes described in the documentation are valid for both SAP NetWeaver Identity Management 7.1 and 7.2. Most of the screen shots are taken from the 7.2 version. There are separate descriptions in cases where the two versions differ from each other.

Overview

In this section an overview is given over connector methods, parameters and the SAP NetWeaver Identity Management Virtual Directory Server (VDS) structures. Also request processing is explained in details. Parts of the code from the *sample* connector are used to illustrate and explain the desired concept (full sample code is given in section *Sample code: FileBrowserConnector* on page 47).

Connector methods

The application integration code, extending the core VDS code, is implemented as a Java class with predefined interface – which ensures that the VDS core can invoke appropriate connector method when needed (see example of an empty class – without specific target implementation of the methods below).

The *AbstractOperation* class (the Java class that is extended) contains several methods – but only *search, modify, add* and *delete* methods must be implemented. The other methods (*bind, compare, initialize, terminate)* may also be invoked by the core VDS code, but their invocation is optional.

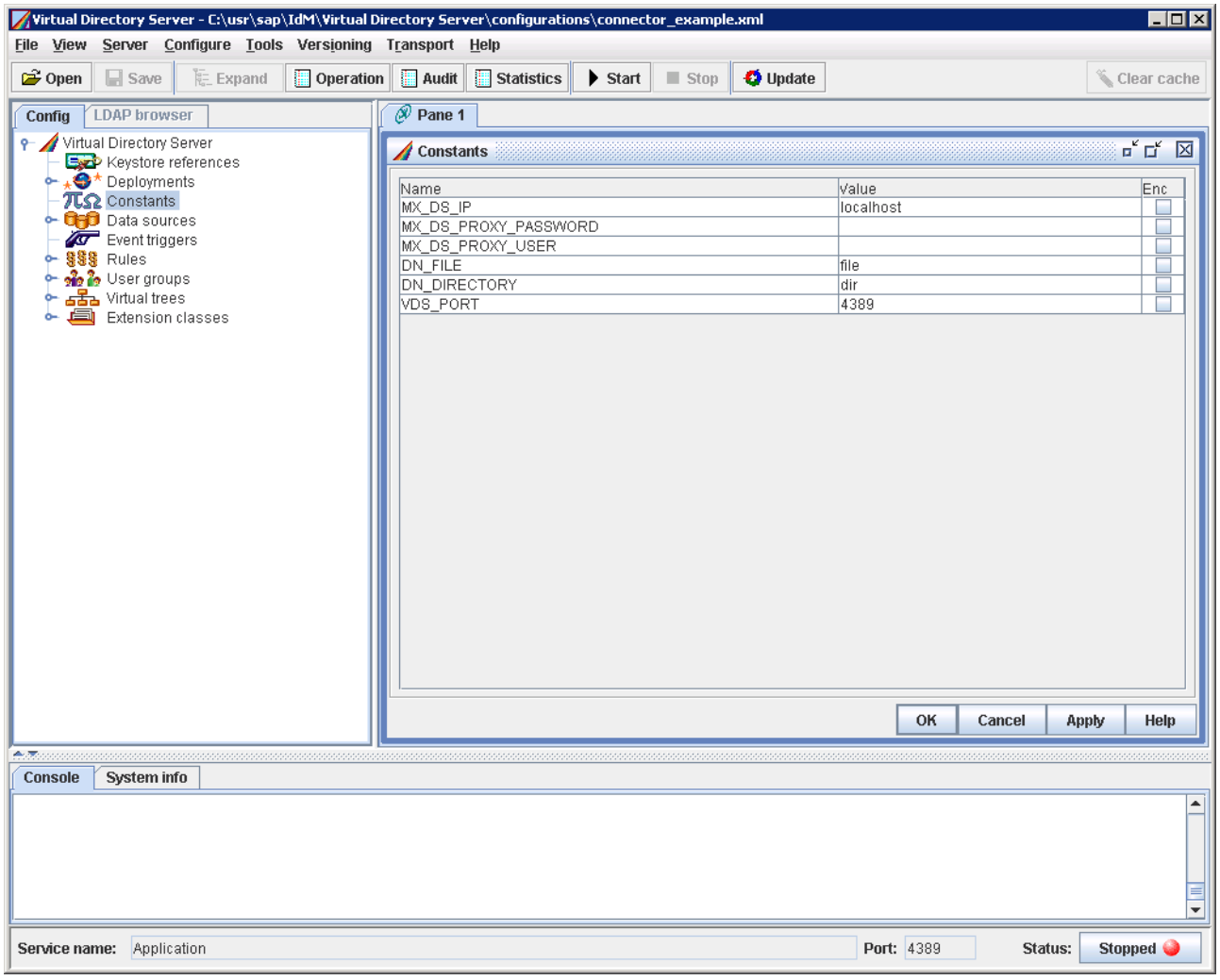
```
// --- This is an example of a Java data class
import com.sap.idm.vds.*;

public class EmptyClass extends com.sap.idm.vds.AbstractOperation {

    public MVDOperationResult bind (MVDHashMap param) {
        MVDOperationResult result = new MVDOperationResult();
        // --- Place your java implementation here
        return result;
    }
    public MVDSearchResults search (MVDHashMap param) {
        MVDSearchResults result = new MVDSearchResults();
        // --- Place your java implementation here
        return result;
    }
    public MVDOperationResult modify (MVDHashMap param) {
        MVDOperationResult result = new MVDOperationResult();
        // --- Place your java implementation here
        return result;
    }
    public MVDOperationResult add (MVDHashMap param) {
        MVDOperationResult result = new MVDOperationResult();
        // --- Place your java implementation here
        return result;
    }
    public MVDOperationResult delete (MVDHashMap param) {
        MVDOperationResult result = new MVDOperationResult();
        // --- Place your java implementation here
        return result;
    }
    public MVDOperationResult compare (MVDHashMap param) {
        MVDOperationResult result = new MVDOperationResult();
        // --- Place your java implementation here
        return result;
    }
    public MVDOperationResult initialize (MVDHashMap param) {
        MVDOperationResult result = new MVDOperationResult();
        // --- Place your java implementation here
        return result;
    }
    public MVDOperationResult terminate (MVDHashMap param) {
        MVDOperationResult result = new MVDOperationResult();
        // --- Place your java implementation here
        return result;
    }
```

Connector parameters

The core VDS code passes the information to the extending integration code using the Java HashMap that is a parameter of all connector methods (see the Java class above).

The HashMap parameter contains multiple key/value pairs. There are two main sources of the information that the core VDS code may pass to the integration code:

1. Properties of the requested operation:
   a) The starting point for the operation.
   b) Filter.
   c) The attributes to be modified, etc.
2. The Virtual Directory Server configuration file. The connector-relevant information may typically be configured in three different places:
   a) Data Source object – contains the connection properties to the target application but also useful values and hints that help the integration code to create a proper target request.
   b) Node object in the Virtual Directory Server virtual tree – the one that is tied to the data source mentioned above (see tutorials for the basic concepts about SAP NW Identity Management Virtual Directory Server).
   c) Constants – i.e. the Virtual Directory Server global values.



Each of the relevant values is added to the HashMap parameter and is passed to the integration code. Each of the values is associated with specific *parameter name*. In order to distinguish between (potentially) overlapping names, the VDS prefixes the parameter names in the following way:

Source	Prefix	Example (passed key)
Operation properties	LDAP_	LDAP_STARTINGPOINT, LDAP_SIZELIMIT
Data Source properties	DS_	DS_ADDITIONALFILTER
Node properties	NODE_	NODE_OBJECTCLASS
Constants	GLOBAL_	GLOBAL_SYSNAME

The full list of all operation properties is given in section *Operational parameters (for each operation)* on page 50.

Virtual Directory Server structures

Since the core VDS code cannot handle the various data structures returned by the target application, the integration code (together with the target application API) is responsible for preparing and converting of the low-level results to well defined VDS structures. This section presents these structures.

MVDOperationResult

This object is used for returning of the operation result. It contains three important properties:

- Operation result code
- Operation additional info message
- The state of the operation (success/failure)

The core VDS code will act based on the operation state found in this structure, and in most cases return the code and message to the caller application.

In the context of the add/modify/delete operations it is the only object that is returned from the connector. In the context of the search operation, this object is part of the *MVDSearchResults* object (described on page 7). It is the responsibility of the integration code (together with the target application API) to set proper values for the object.

The object is initiated with a specific error code and the operation state set to "fail", so returning the object back to the core VDS code without explicitly setting the operation result code and the state will be treated by the core VDS code as an unsuccessful operation execution.

There are several methods for setting (and reading) the properties of this object.

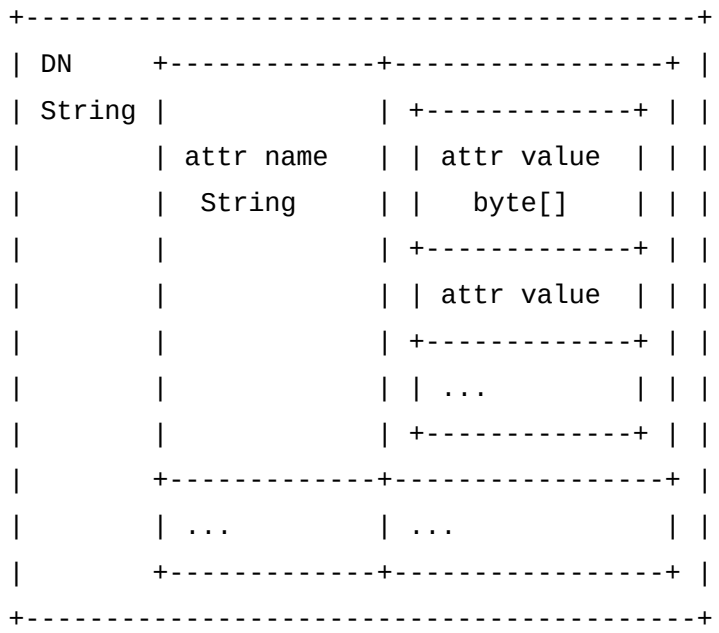
Setting the properties of this object

- `public void setOK()` – sets the state to "success", code to "0" and no additional message.
- `public void setOK(int code, java.lang.String mess)` – sets the state to "success", code to <code>, and sets the additional message (e.g. when returning error code 4 – Size Limit Exceeded).
- `public void setError(int code, java.lang.String mess)` – sets the state to "fail", code to <code>, and sets the additional message.

MVDSearchResultEntry

This object exists only in the context of the search operation. An entry contains:

- A distinguished name – unique identifier of the entry.
- An attribute set – i.e. the entry's attributes and their values. The attribute set is implemented as Java HashMap where keys are attribute names (keys) and their values are implemented using Java Vector (thus giving possibility for returning multiple values for each of the attributes).



There are a number of methods for updating/reading information from this object.

Note:
The byte array containing a value is stored as UTF-8 (important for non-ASCII characters).

Setting and getting the entry identifier (distinguished name)

```
public void setDn(String aDn)

public String getDn()
```

Operation on whole attribute set

Setting and getting the whole attribute set at once:

```
public void setAttrAndValues(HashMap aHash)

public HashMap getAttrAndValues()
```

Merging the whole attribute set with other:

```
public void appendAttrAndValues(HashMap aHash)
```

Operations on "single" attributes in the attribute set

Appending new attribute to the set:		Comments
	`public void setAttrValue(String aAttrName, boolean aAppend, byte[] aAttrValue)`	First parameter is the attribute name to be added to the attribute set.
	`public void setAttrValue(String aAttrName, boolean aAppend, String aAttrValue)`	Second parameter influence the way the attribute is added to the attribute set if it already exists: 1. If true, the new value(s) will be appended, thus creating a multi valued attribute. 2. If false, the old value(s) will be overwritten.
	`public void setAttrValue(String aAttrName, boolean aAppend, Vector aAttrValue)`	It is possible to add: 1. Single attribute value (either String or byte[] (binary)). 2. Multiple values (all elements in Vector). 3. Multiple values (String with multiple values separated by the specified delimiter).
	`public void setAttrValue(String aAttrName, boolean aAppend, String aMultiAttrValue,String aSeparator)`	

Reading the values for a specific attribute:		Comments
	`public Vector getAttrValues(String aAttrName)`	The Vector is always returned (each attribute is potentially a multi-valued attribute).
	`public String getFirstAttrValue(String aAttrName)`	Returns the first value in the Vector (useful when we know that the attribute is actually single-valued so we avoid going through *getAttrValues* first).

Other		Comments
	`public void replaceAttrValue(String aAttrName, String aOldValue, String aNewValue)`	Replaces the old value of the attribute with the new one.
	`public void replaceAttrValue(String aAttrName, HashMap aTheValues)`	Replaces all value pairs in the given hash map for the given attribute.
	`public void constructAttrValue(String aAttrName, Vector aAttrToUse, String aDelimiter)`	Constructs a new attribute using the values of the listed attributes using the specified delimiter as a separator.

Note:
The removal of the attributes from the attribute set is done using Java HashMap's remove method. For example, if "temp" is MVDSearchResultEntry then the following removes the attribute "attr_to_be_removed": `(temp.getAttrValues()).remove("attr_to_be_removed").`

MVDSearchResults

This object exists only in the context of search operation. It is a list of entries that are the result of the search operation. This object extends a Java Vector. It contains:

- None or more *MVDSearchResultEntry* objects (described on page 5).
- Exactly one *MVDOperationResult* object (described on page 4).

Typical operations on this object are:

- Adding an entry to the entry list.
- Carrying out identical attribute operation on all entries in the list (e.g. adding/removing the same attribute).
- Setting the operational state.

Adding an entry to the entry list

As stated above, this object extends Java Vector, so Java Vector's range of *add()* method(s) are sufficient (*add()*, *addAll()*).

Carrying out identical attribute operation on all entries in the list

Adding the same attribute to all entries:

See *MVDSearchResultEntry* methods for adding single attributes – all methods are applicable on *MVDSearchResult* and will be executed on all its members.

Removing the same attribute from all entries:		Comment
	`public void removeAttribute(String aAttrToRemove)`	Removes the attribute from all entries in this *MVDSearchResult* object.

Listing all entries in the list (simple Vector browsing):

Example:

```
for (int ix = 0;ix < result.size(); ix ++) {
MVDSearchResultEntry entry = MVDSearchResultEntry)
result.elementAt(ix);
// do something with entry
}
```

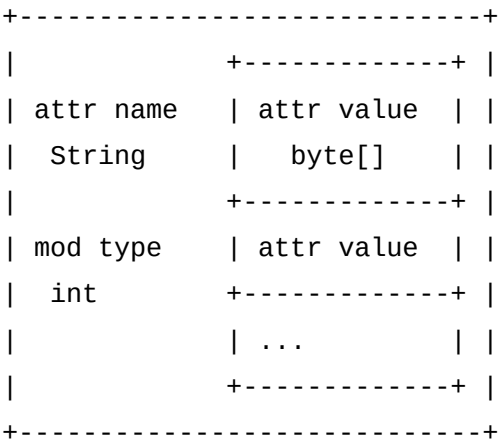
Setting the operational state

The methods for manipulating the result code are inherited from *MVDOperationResult* object described on page 4.

MVDModAttrValue

This object exists only in the context of *modify* operation. It describes how one attribute should be modified. Normally, the core VDS code passes a Vector of such objects to the integration code, describing all changes to be carried out on a single entry. It contains:

- The attribute name of the attribute to be modified.
- The modification mode: 0=add value,1=delete value, 2=replace value.
- The list of the values that are used for modification.



Setting/getting the object properties

Simple gets and sets:

- `public String getAttrName()`
- `public void setAttrName(String attrName)`
- `public int getModType()`
- `public void setModType(int m)`

Obtaining modification values

The object extends Java Vector – any Java Vector method can be used.

Request processing

Typically, the integration code's (together with the target application API) processing of a request handed by the core VDS code can be divided in several phases. The phases are:

1. Getting the needed data from input HashMap
2. Pre-processing of data (optional)
3. Executing the target application API
4. Converting the results to the VDS structures
5. Post-processing of data (optional)
6. Returning the result to the core VDS code

Obtaining necessary data

Normally, the amount of information sent (in the HashMap) to the integration code (together with the target application API) is larger than needed. It is the integration code's responsibility to extract the information that is necessary for successful execution of the requested operation on the target application.

The Virtual Directory Server implements several methods for extracting values from input HashMap. Basically, all of them are just variants of Java HashMap's *get* method. Even if it is always possible to use the *get* method to obtain values from the HashMap, it is recommended to use the Virtual Directory Server's methods in the connector context.

Mandatory parameters

In order to carry out the operation on the target application, the connector may require a certain set of parameters, i.e. some of the parameters are mandatory for the requested operation. For instance, all *operation* parameters are mandatory – without them the integration code cannot construct the proper target operation and execute appropriate application API method.

Extracting of such parameters from the HashMap is carried out using one of the following methods:

- `public Object get<Prefix>MandatoryParameter(String parameterName) throws Runtime Exception`, where <Prefix> is one of Ldap/DS/Node/Global and parameterName does not contain prefix (the full HashMap key is constructed by the method, by concatenating prefix, "_" and the parameterName).
- `public Object getMandatoryParameter(String key) throws Runtime Exception`, where key is fully constructed (prefix + "_" + parameterName). If the constructed key is not present in the HashMap, the method throws Java runtime and the processing of the request is abandoned, returning the error to the client application.

Non-mandatory parameters

Non-mandatory parameters are retrieved using the following methods:

- `public Object get<Prefix>Parameter(String parameterName, Object defaultValue) throws Runtime Exception`
- `public Object getParameter(String key, Object defaultValue) throws Runtime Exception`

Same <prefix> values and rule for key construction as for mandatory values. If the constructed key is not present in the HashMap, the method returns the specified *defaultValue*.

Examples

```
Integer scope = (Integer) param.getLdapMandatoryParameter("OPSUBTYPE");
String startingpoint  = (String)
param.getMandatoryParameter("LDAP_STARTINGPOINT");Vector attrNamesAndModValues =
(Vector)param.getLdapMandatoryParameter("DATA");
String parSzLimit = (String) param.getDSParameter("SIZELIMITTYPE", "TOP");
// --- if DS_DZLIMITTYPE not present in the HashMap, “TOP” is returned
```

Data preparation (optional)

This is an optional phase.

Typically, the format and/or type of the values obtained from the input HashMap, does not satisfy the needs of the "low-level" API.

During the data preparation the following will typically happen:

- Pre-processing of the obtained values so they can be used together with the API. The typical operation may be:
  - Type conversions.
  - Pre-processing the incoming filter (in case of the search operation).
- Constructing of new values based on those obtained from HashMap.

Executing the target application API methods

The pre-processed parameters and values are passed to the target API methods and results are achieved.

Result conversion

The core VDS code expects the results from the integration code's (and the target application API's) execution in the specific format. The results obtained from the target API execution cannot be returned directly to the core VDS code. In this phase, the API results need to be converted to the VDS structures expected by the core VDS code. Depending on the requested operation, the following VDS structures are valid responses:

- *MVDSearchResults*
- *MVDSearchResultEntry*
- *MVDOperationResult*

Details about these structures and API for their manipulation are shown in section *Virtual Directory Server structures* on page 4.

Post-processing the data (optional)

This is an optional phase.

Sometimes it may be needed to post-process the results (now stored in the VDS structures) before they are returned to the core VDS code. Typically it may involve the following:

- Removing some entries from the result set.
- Adding artificial attributes to the entries in the result set.

Note:
The post-processing of the target results could also be executed before conversion of the results to VDS structures (while the data is still in the target result format).But since the VDS structures are quite generic (standard Java objects), they are easily manipulated using the standard Java API calls. In addition, the new (complex) methods that are created for their manipulation may be re-used in another connector as well.

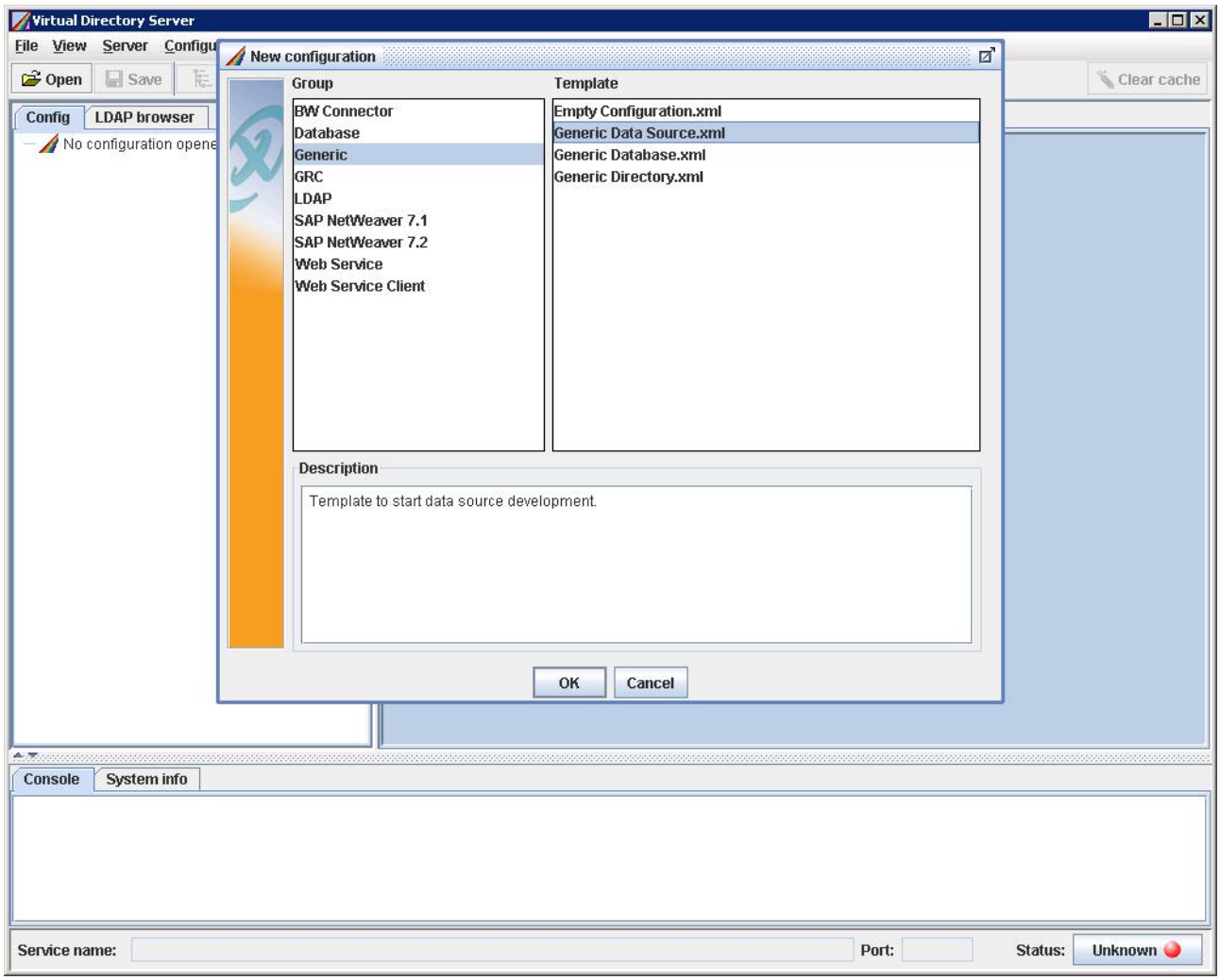
Preparations

Before developing the connector a few preparations must be completed in the Virtual Directory Server.

Preparing the Virtual Directory Server configuration

To prepare the configuration, do the following:

1. Start the Virtual Directory Server console (*Start/All Programs/SAP NetWeaver Identity Management/Virtual Directory Server*).
2. Select "File and then "New…" to open a "New configuration" dialog box.



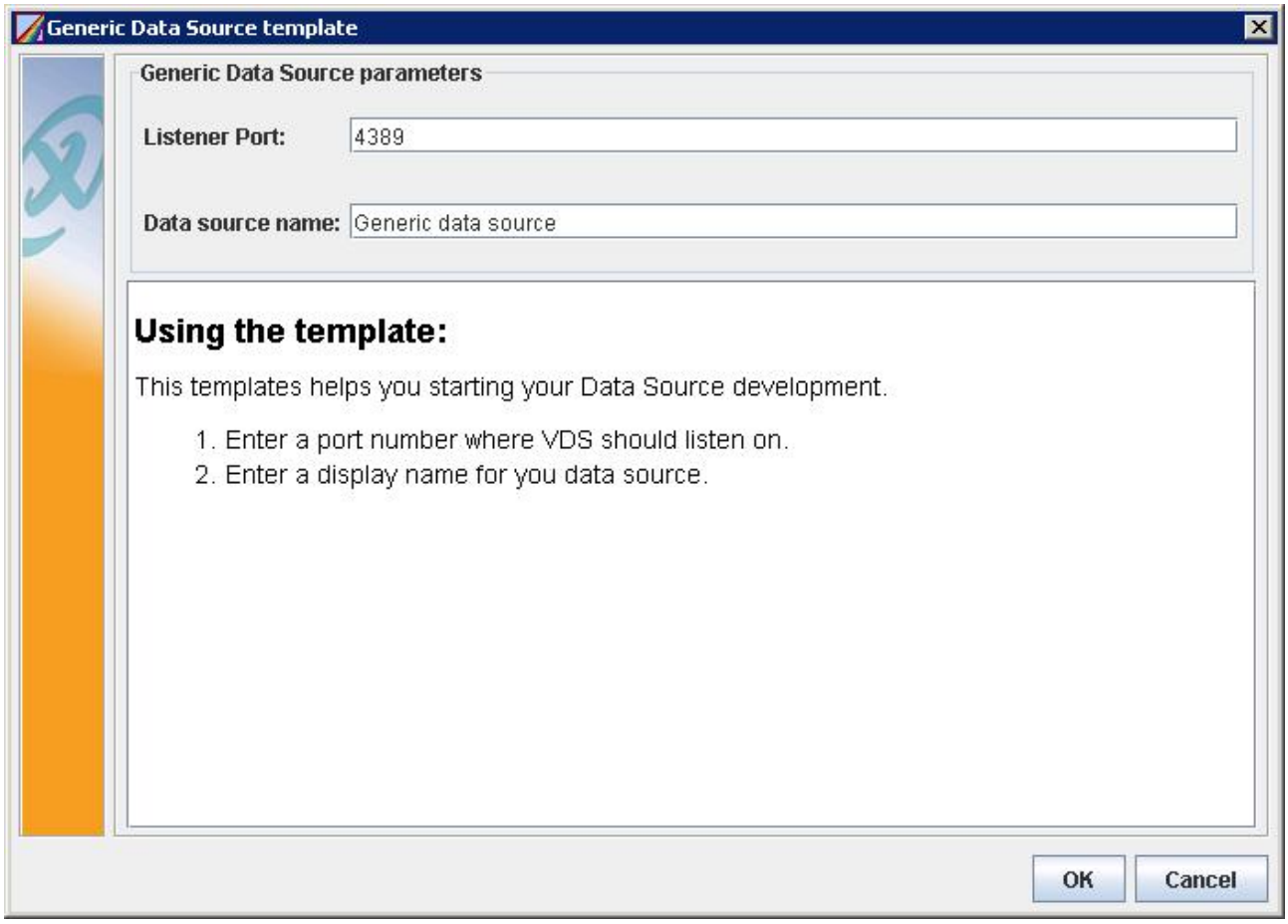
Select "Generic" in the "Group" pane.

Identity Management 7.2
Select the generic template "Generic Data Source.xml" in the "Template" pane to create an empty configuration as container for your connector.

Identity Management 7.1
Select the generic connector template "Generic Connector.xml" in the "Template" pane to create an empty configuration as container for your connector.

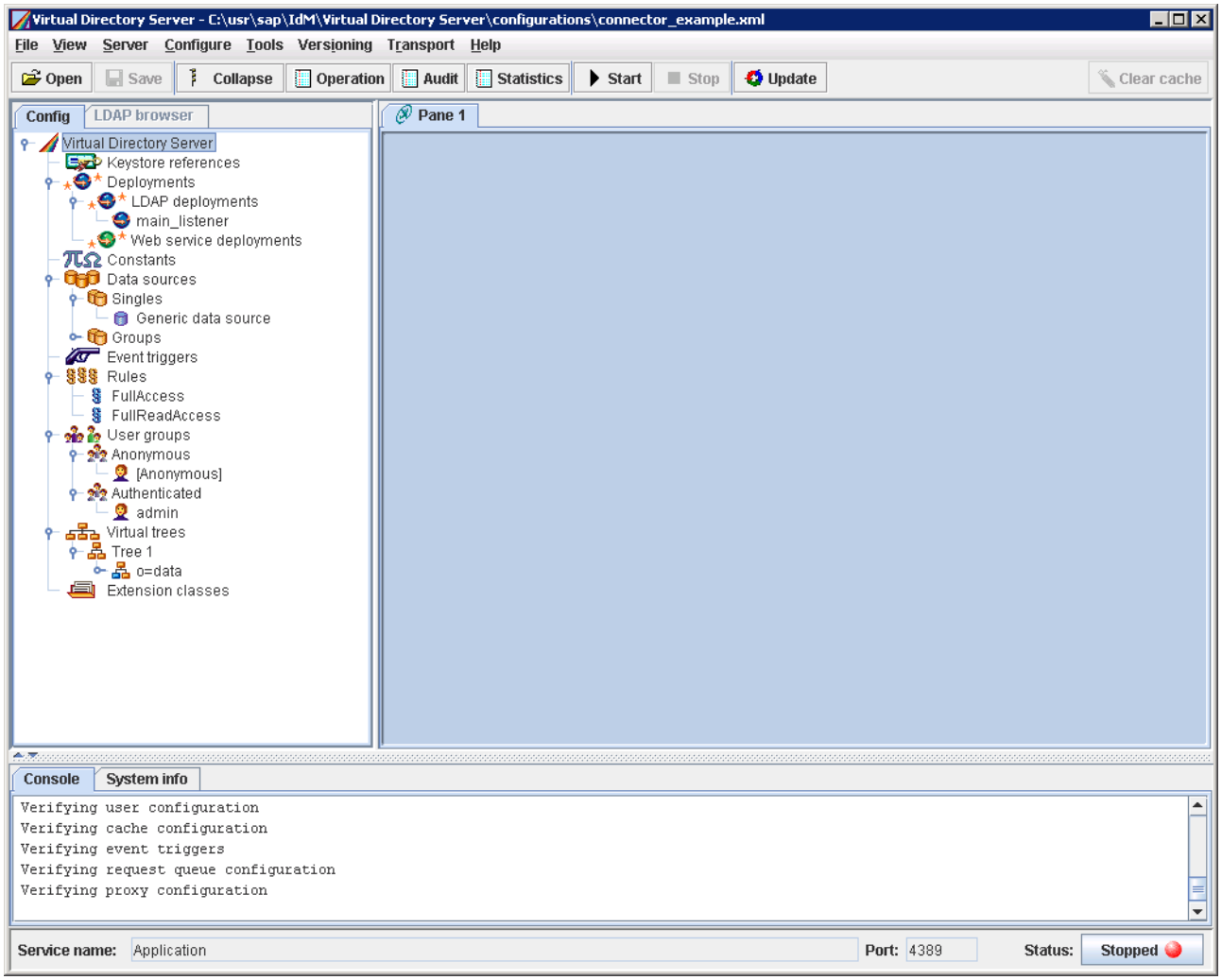
3. Choose "OK".

The dialog box for the template appears – Generic Data Source template for Identity Management 7.2 and Generic Connector template for Identity Management 7.1:



Enter a port number the Virtual Directory Server should listen on and a display name for your data source (here *4389* as the port number, and the default name *Generic data source*).

4. Choose "OK" and save the new configuration in the *configurations* folder (here saved as *connector_example.xml*).



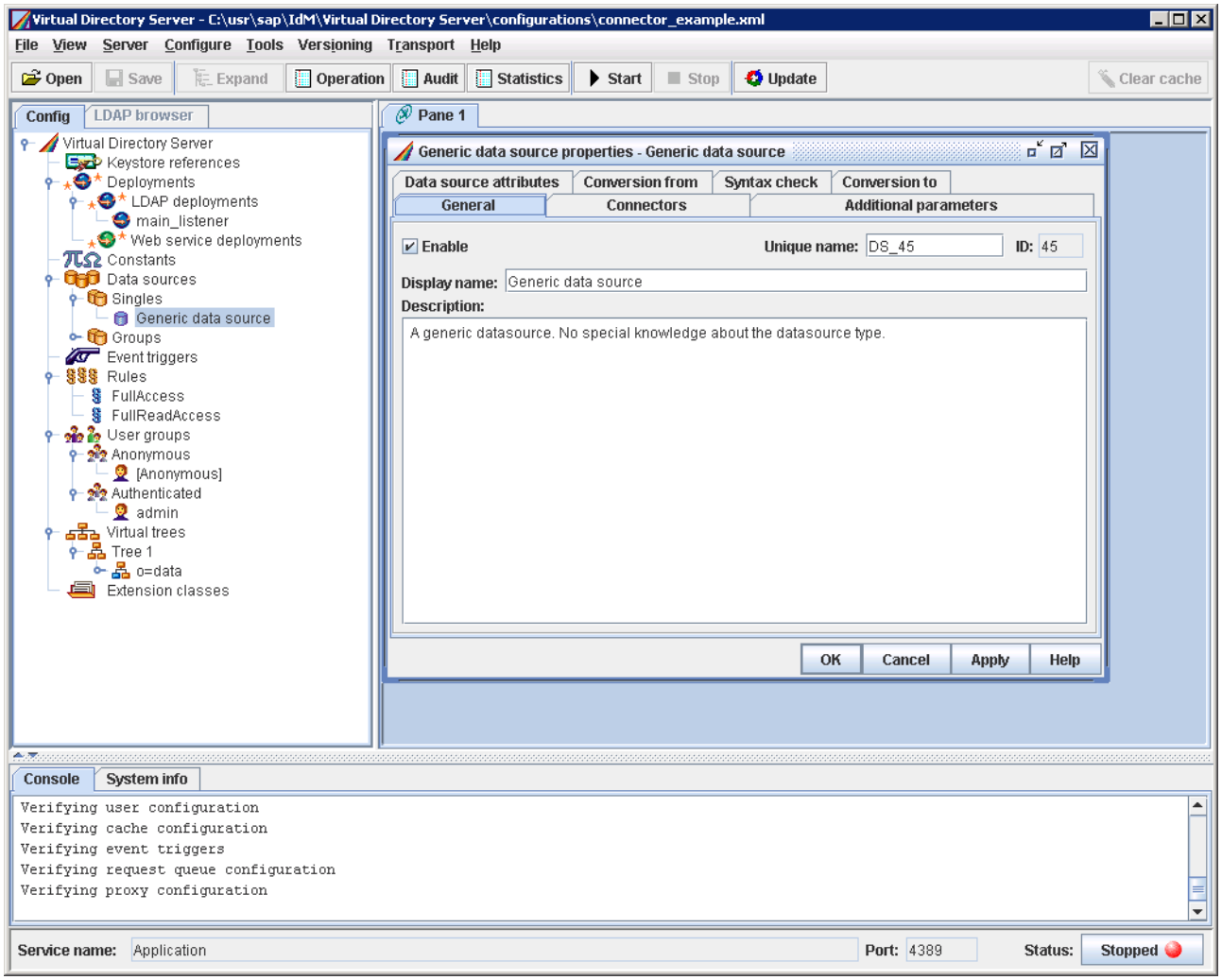
This creates a configuration with the following sample entries:

- LDAP deployments: a listener on the port you previously entered.
- A generic data source with the name you entered previously.
- A set of rules:
  - Full access, which includes ADD/MODIFY/DELETE/COMPARE and SEARCH operations.
  - Full read access, which includes COMPARE and SEARCH operations.
- A set of user groups:
  - Group "Anonymous" with a user named "[Anonymous]" for anonymous LDAP requests.
  - Group "Authenticated" with a user named "admin" and the password "admin" for authenticated LDAP requests.
- A sample virtual tree

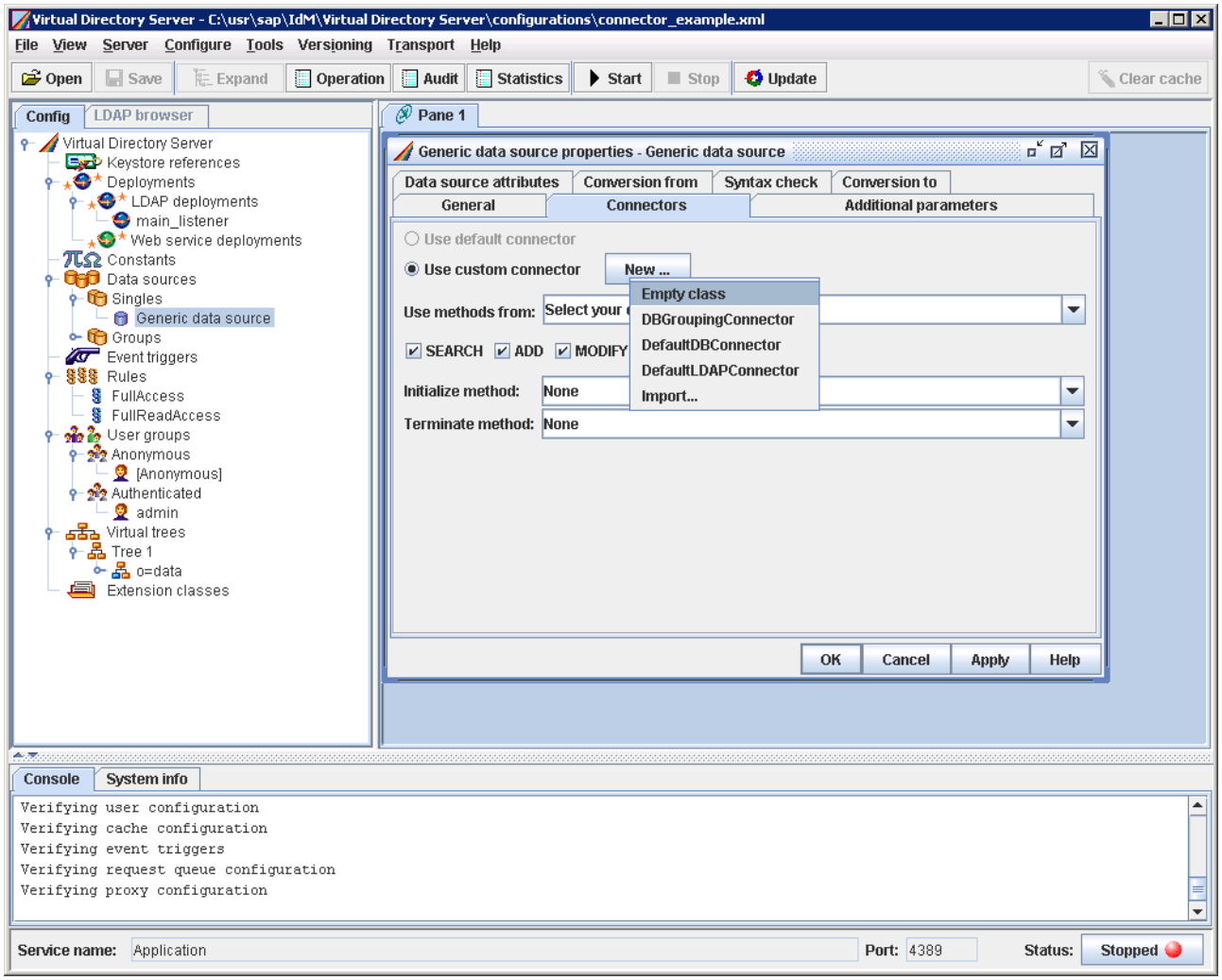
Create sample code

To create a sample code, do the following:

1. View the properties of your data source, either by double-clicking or selecting "Properties…" from the context menu.

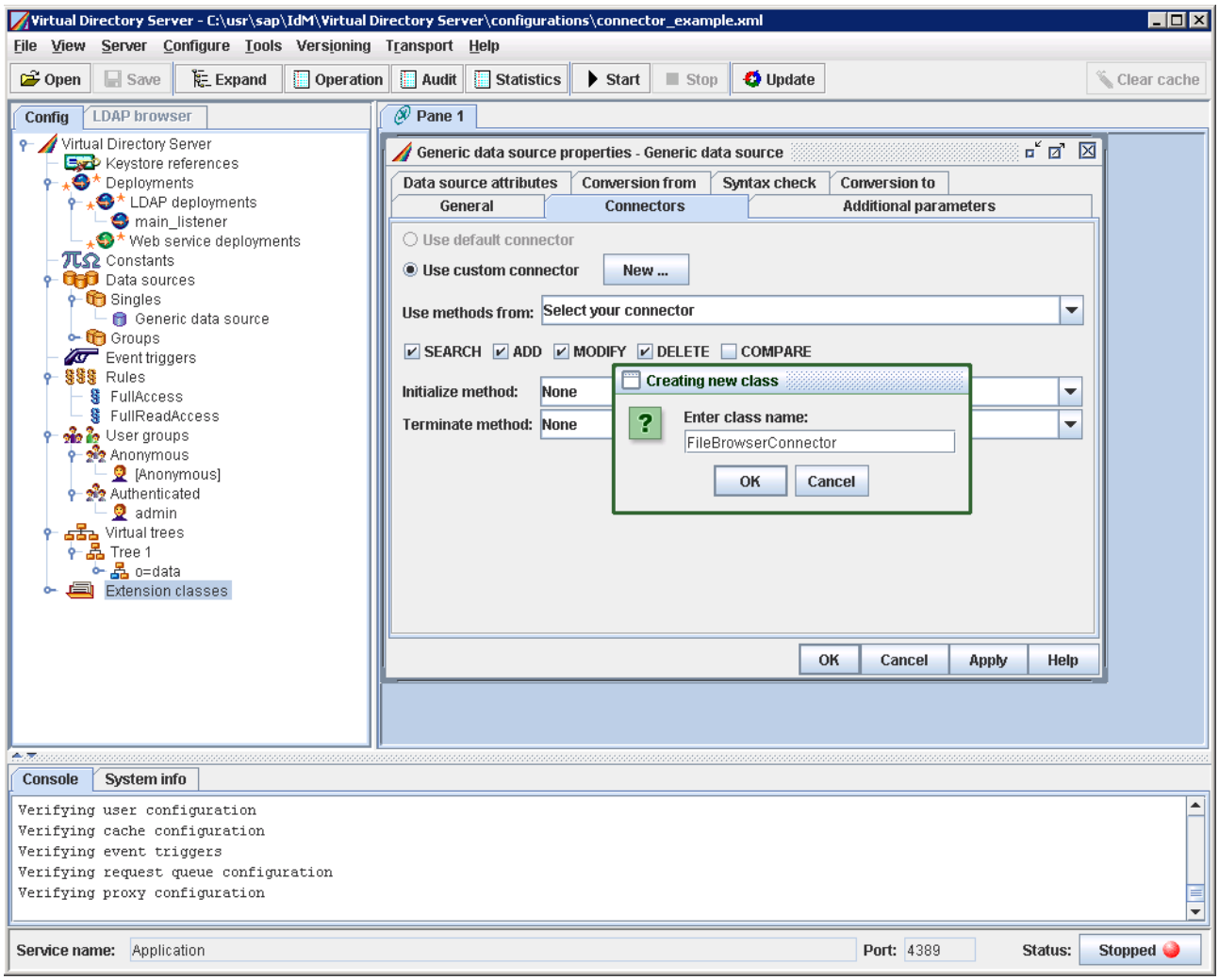


2. Select the "Connectors" tab and choose "New…":



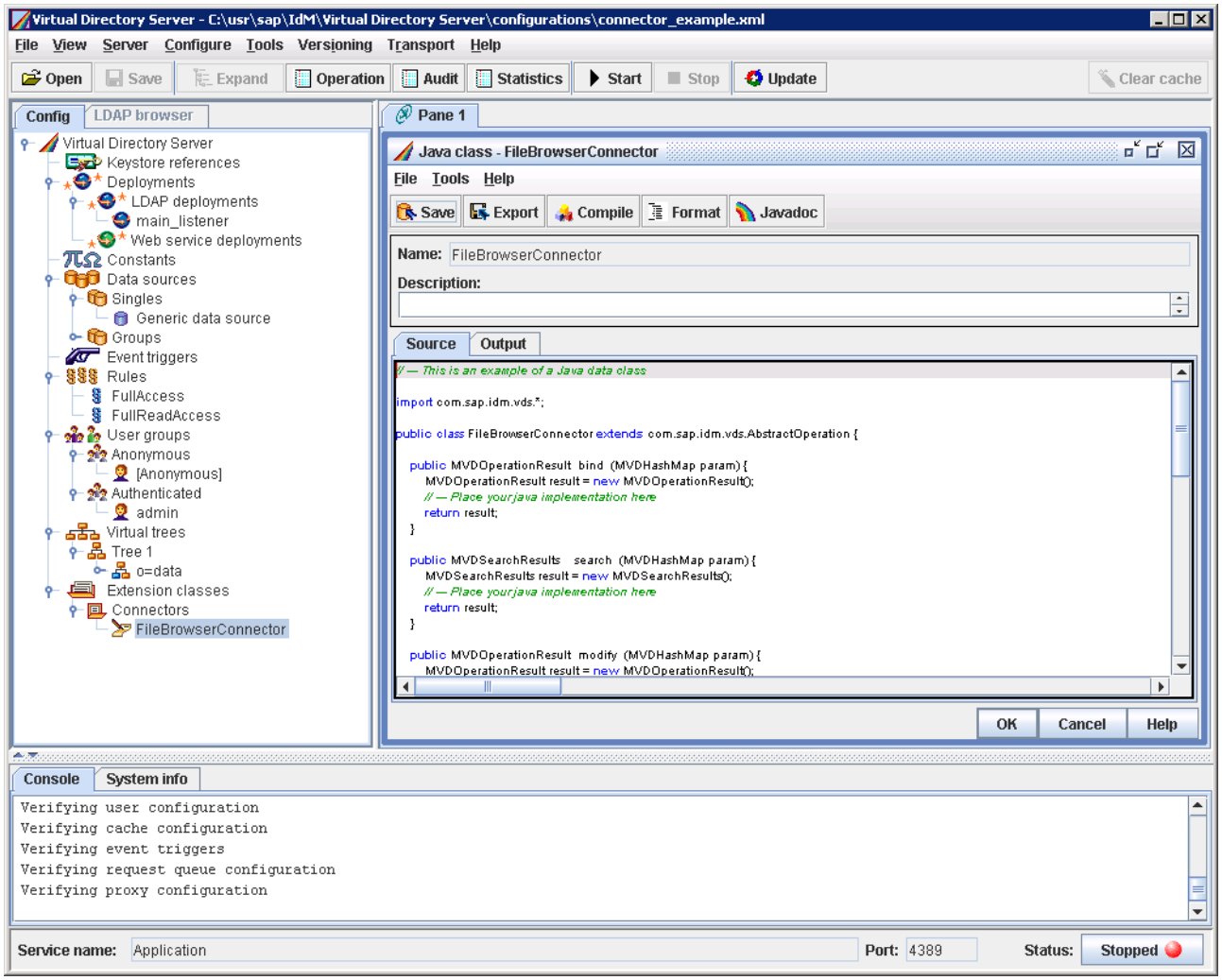
Select "Empty class" template.

3. This will create a new empty class:



Enter the name of the class (e.g. "SampleIntegrationCode" or "MyConnector" etc). The class names cannot contain spaces. Here we give the class name *FileBrowserConnector*, as an example (our sample code).

4. Choose "OK". The class editor opens with the generated implementation stub of the Virtual Directory Server *AbstractOperation* class.



5. Choose the "Save" button (

) to store the class code in the current Virtual Directory Server configuration.

Developing the code

It is possible to create a connector class and add/modify and compile the code in SAP NetWeaver Identity Management Virtual Directory Server. Even if not recommended for development of a complex integration code, it is quite useful when small improvements of an existing code are needed.

Normal way of creating the application integration code is through defining a Java project in one of the Java Development Environments (SAP NetWeaver Developer Studio). This gives the code developer a better flexibility, better code editor and the debugging possibilities.

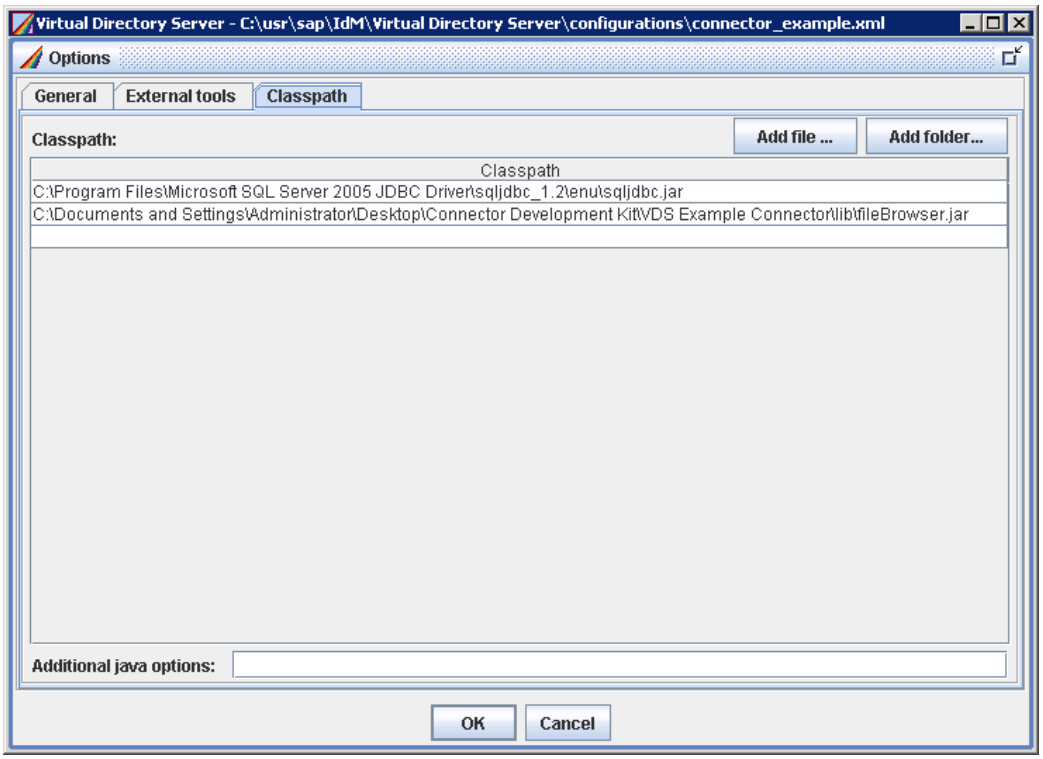
This section describes both the abovementioned methods of creating an application integration code.

Using Virtual Directory Server

Set up run-time environment

For successful compilation of the code, all target jar files have to be in the Virtual Directory Server classpath. To set the VDS Classpath, do the following:

1. Choose **Tools/Options…** from the main menu.
2. Select the "Claspath" tab.

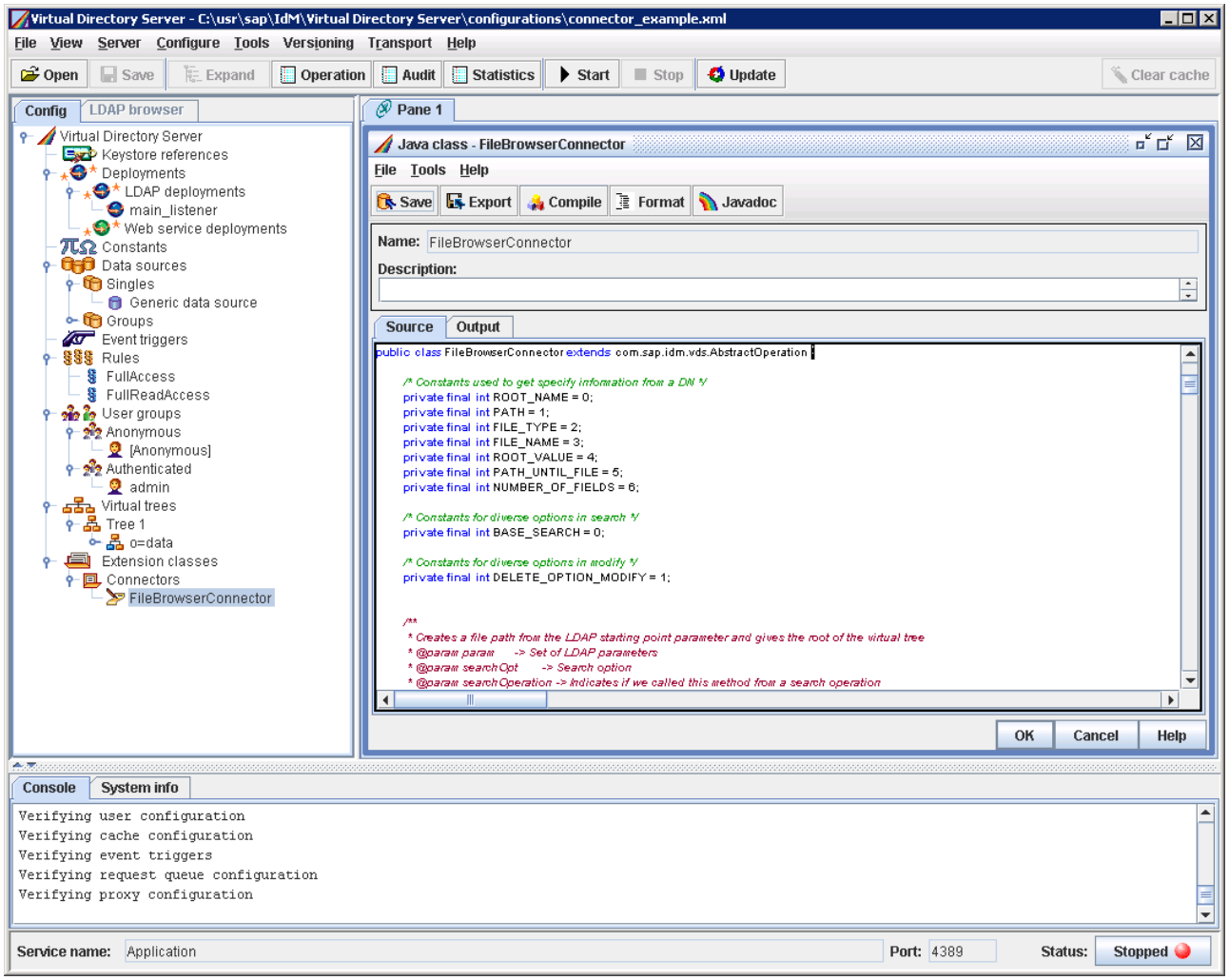


Add necessary jar files. Here we have (in addition to the driver jar file) added *fileBrowser.jar* for our sample code *FileBrowserConnector*.

3. Choose "OK".

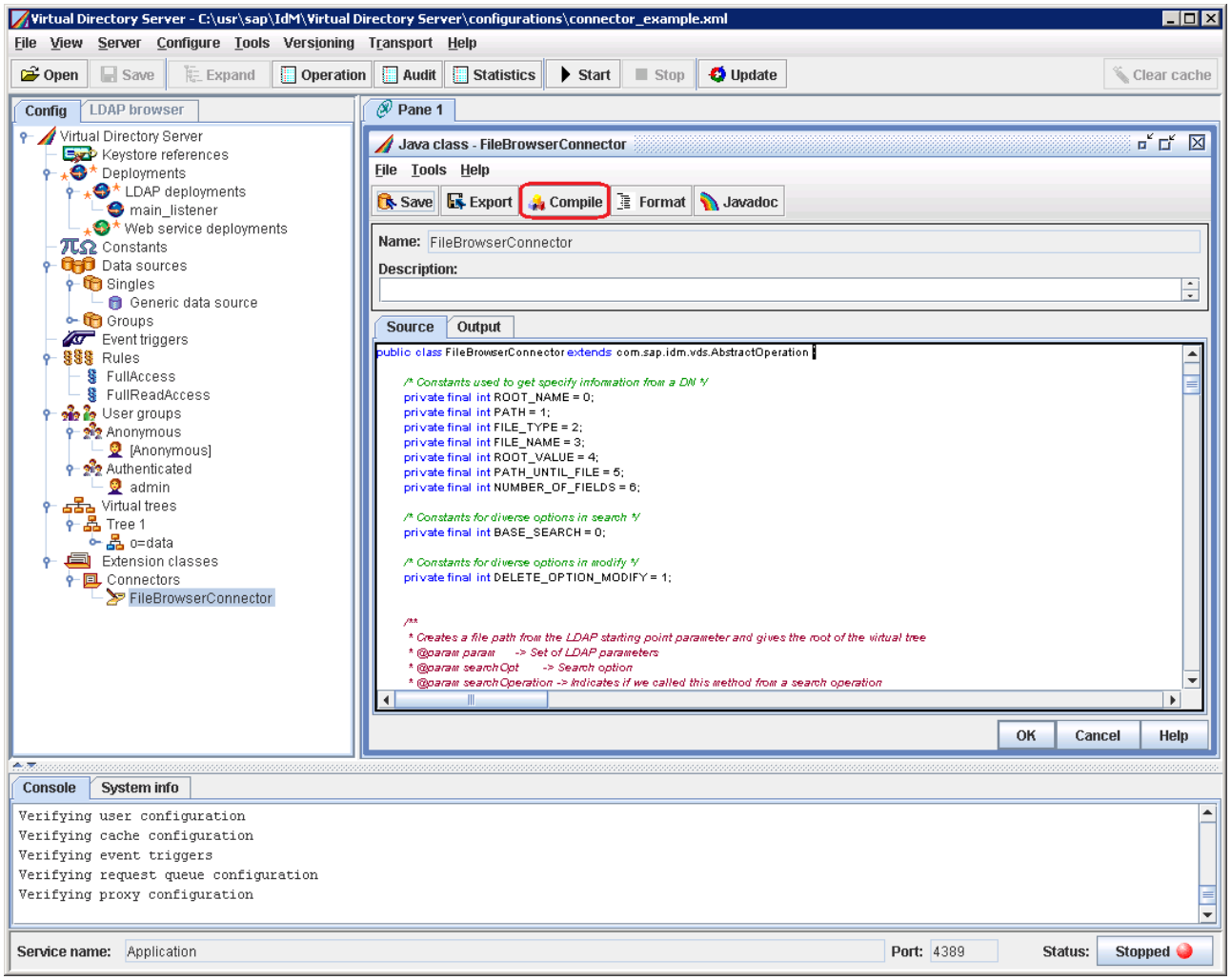
Developing the class code

In the Virtual Directory Server console, edit/extend and save the code in the new Java class:

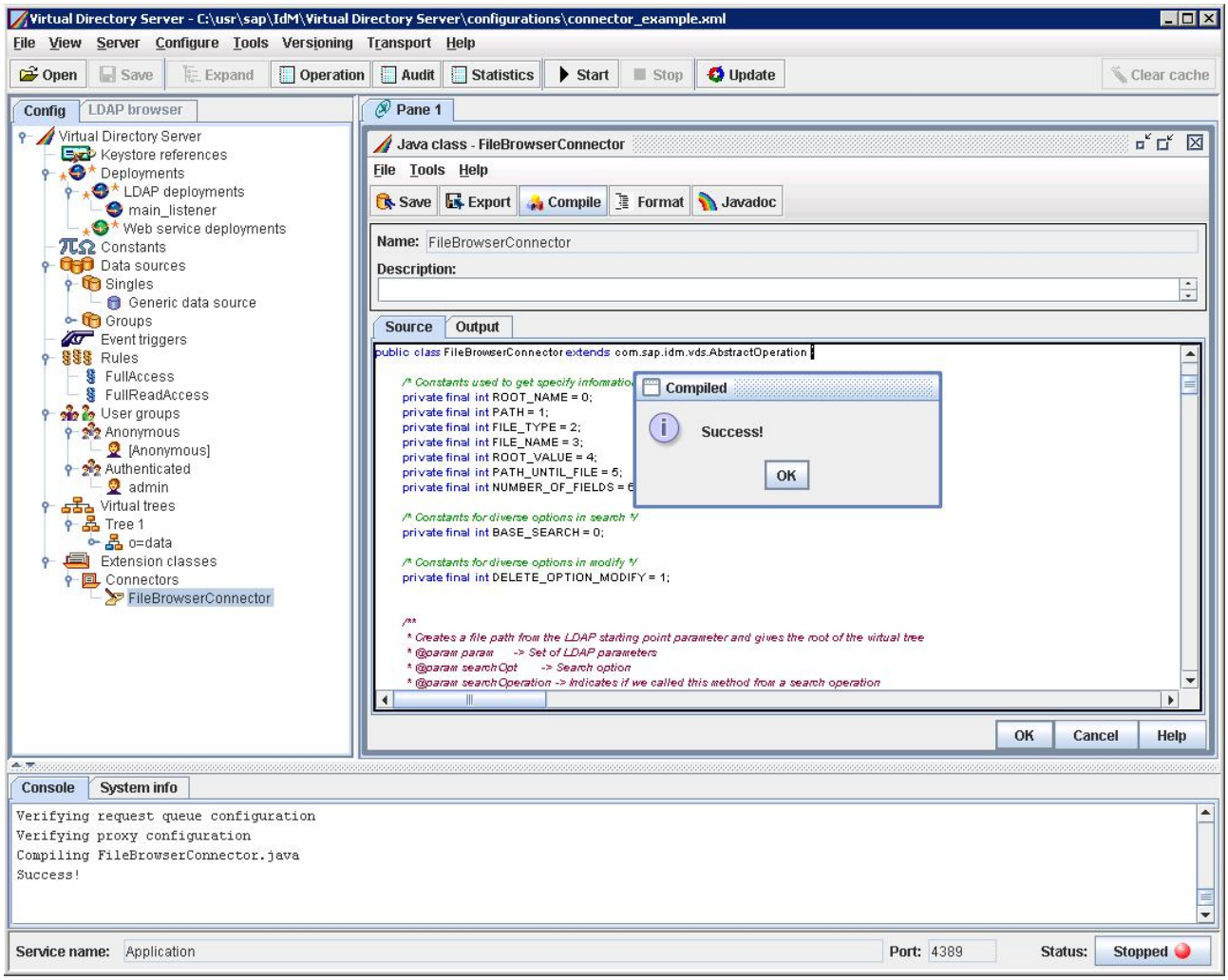


Compiling the class code

You can compile the code from the Virtual Directory Server console by choosing the "Compile" button as shown below:

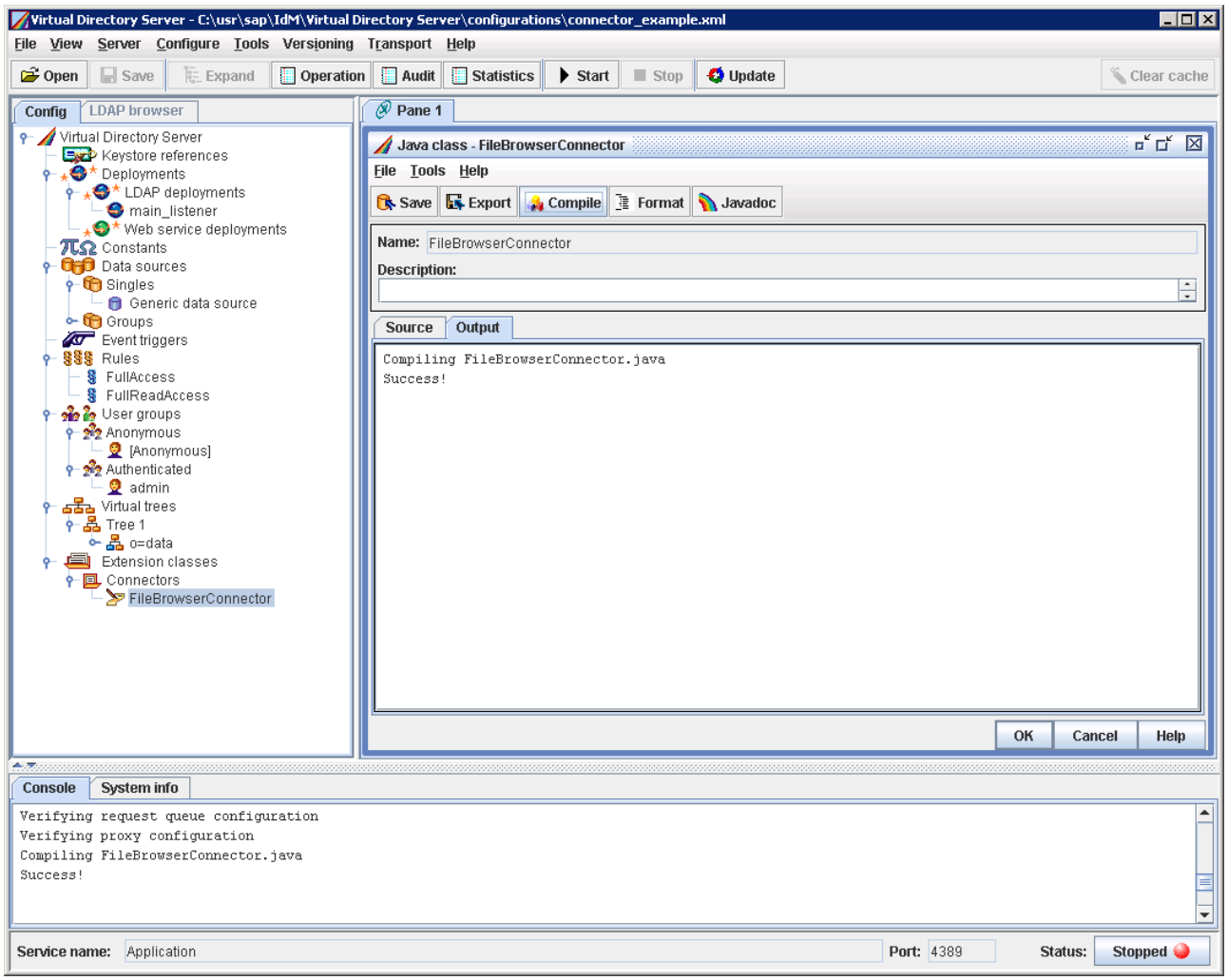


The compilation status is displayed:

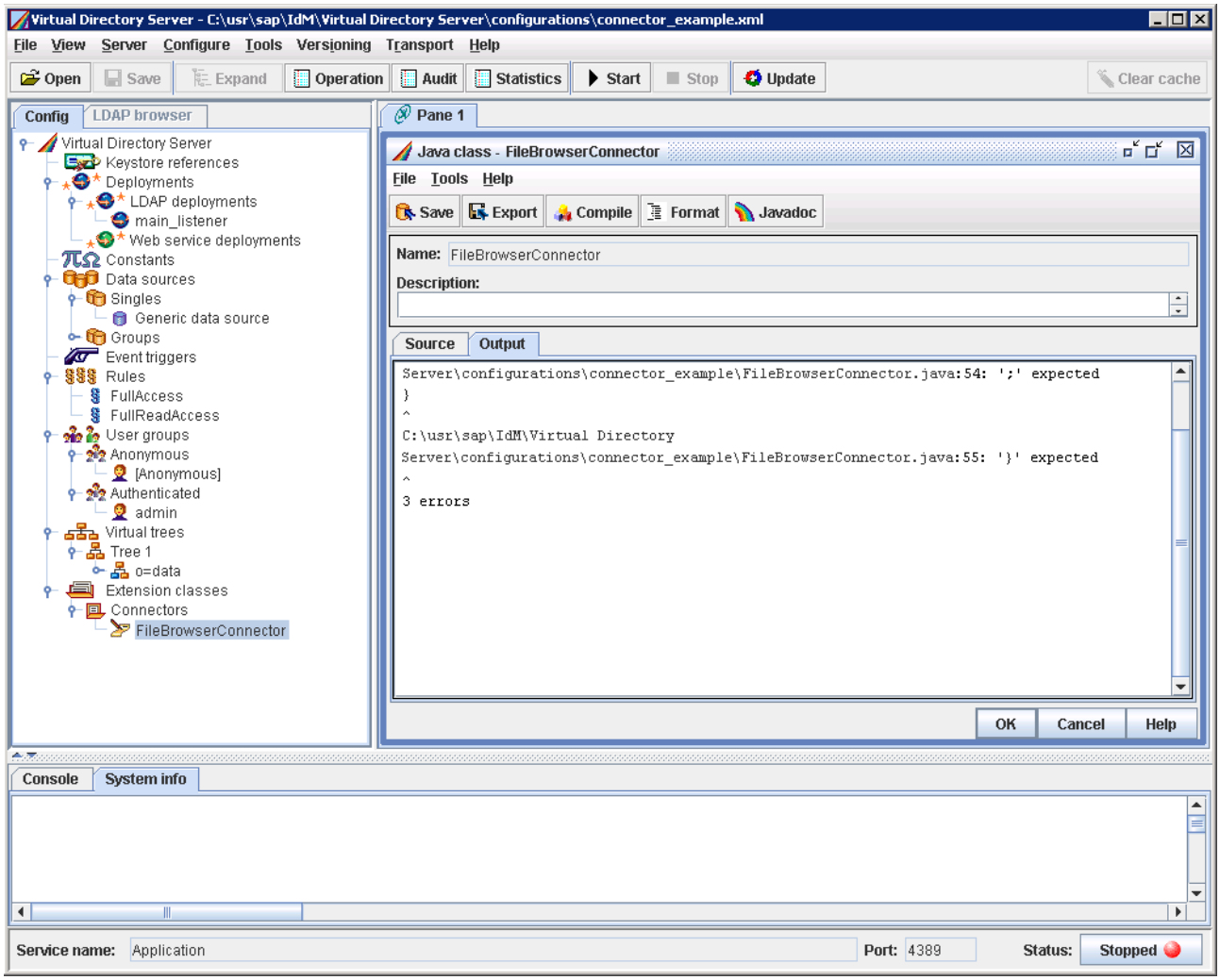


In case of success, a confirmation dialog appears and the information is displayed in the "Console" tab.

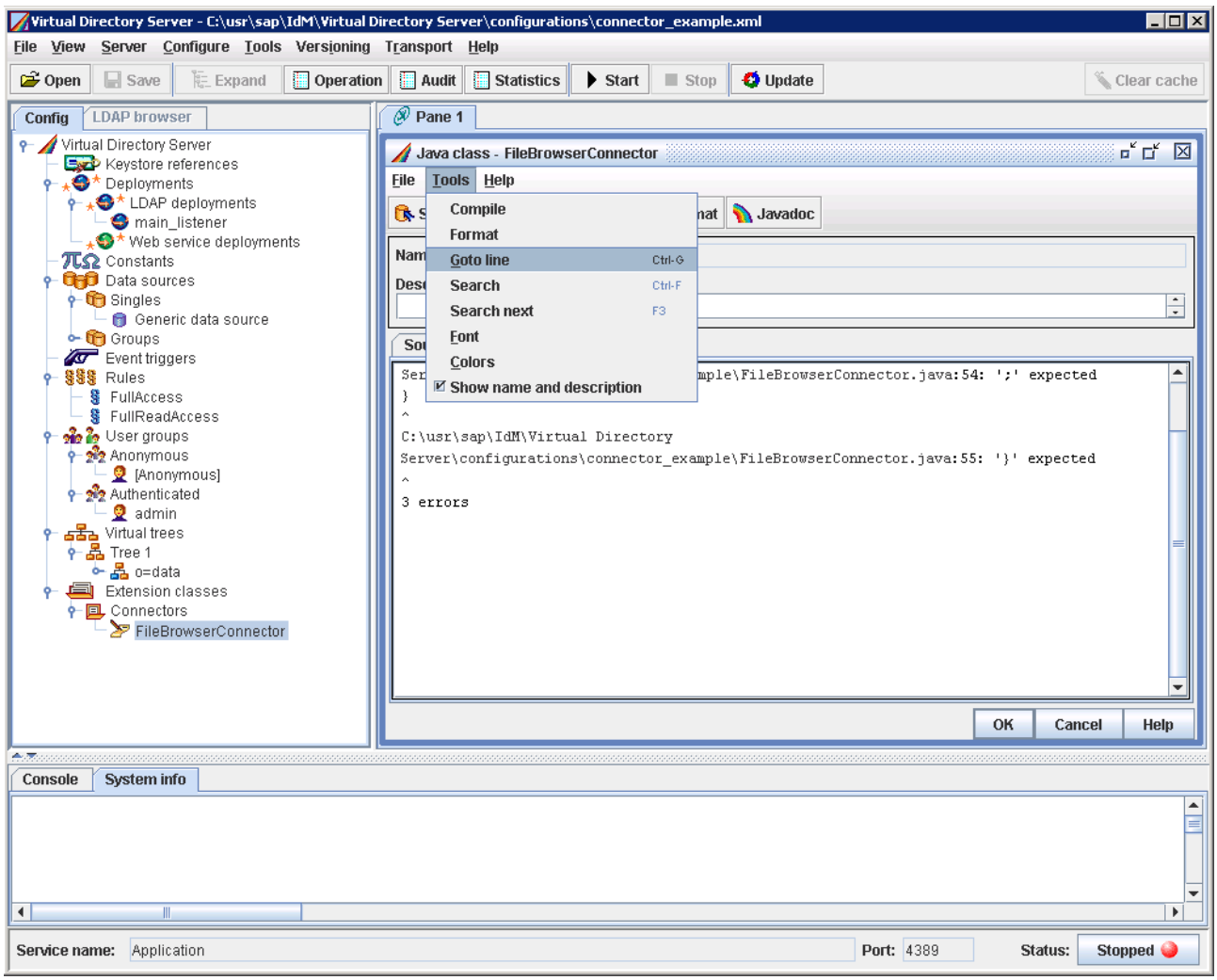
In addition, the information can also be viewed in the "Output" tab.



In case of compilation errors the list of errors is displayed in the "Output" tab:



The reported compilation errors contain the decisive line which can be reached in the "Source" tab by choosing Tools/Goto line (Ctrl-G):



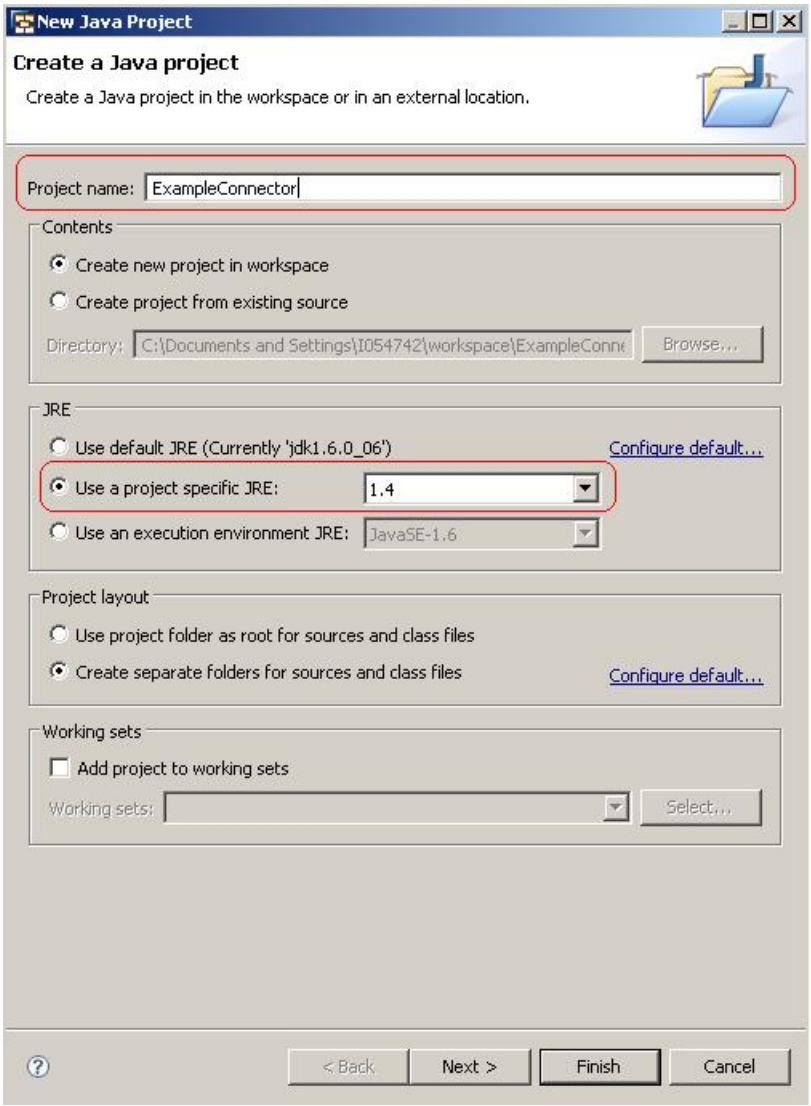
Upon a successful compilation, the class file is created in the work area of the Virtual Directory Server configuration. For details about the work area, how to start the Virtual Directory Server and utilize the code you are creating, see the Virtual Directory Server documentation and tutorials.

Using SAP NetWeaver Developer Studio

Using the SAP NetWeaver Developer Studio for code development is a preferred way of developing your integration code. It is assumed that you have a basic knowledge about the SAP NetWeaver Developer Studio.

Creating new project

In the SAP NetWeaver Developer Studio select **File/New/Java Project** to create new project:



Select the project name (here *ExampleConnector*), select "Use a project specific JRE" and "JRE 1.4" to keep the compatibility with the Virtual Directory Server.

Setting up run-time environment

In order to get access to the Virtual Directory Server structures, the following Virtual Directory Server JAR files have to be in the classpath (see **Project properties/Libraries**):

- *mvd.jar*
- *vdstools.jar*
- *vdsverifier.jar*

These files are available in directory:

```
<VDS installation directory>\lib
```

In addition, all necessary target application JAR files have to be in the classpath too.

Creating the sample class

To create the sample class in the SAP NetWeaver Developer Studio, do the following:

1. Open the Virtual Directory Server configuration you created in the previous section (section *Preparations* on page 12).
2. Expand the "Extension classes" node and the "Connectors" node.
3. Open the created Java class (in our example, the class *FileBrowserConnector*) and copy its content.
4. Create a new class in the project (**New/Class**) and paste the copied content above.
5. Save the class.

Iterative debugging and testing

In order to test the code you are developing, you have to execute the proper operation towards the Virtual Directory Server. For simple connectivity tests and simple searches you can use the built-in LDAP browser (see section *Testing using the internal LDAP browser* on page 31).

But since the code has to be tested with the ADD/MODIFY and the DELETE operation as well, it is recommended to use some standard LDAP client (see section *Testing using an external tool (LDP)* on page 32).

As example we use the sample code *FileBrowserConnector* (see section *Sample code: FileBrowserConnector* on page 47). Load the file *FileBrowserConnector.xml* into the Virtual Directory Server, add the file *lib/fileBrowserAPI.jar* and the directory "src" to the Virtual Directory Server classpath. Start the server (note that you may have to compile the extension class before you can start the server).

See the Virtual Directory Server tutorials for details about authentication, authorization, operation executions, LDAP etc.

Virtual Directory Server

Virtual Directory Server does not provide the code debugging possibility. But it is possible to trace the connector behavior by inspecting the log files generated by the application integration code.

The following method can be used to generate log lines:

```
MVDLogger.Debug(String aMessage)
```

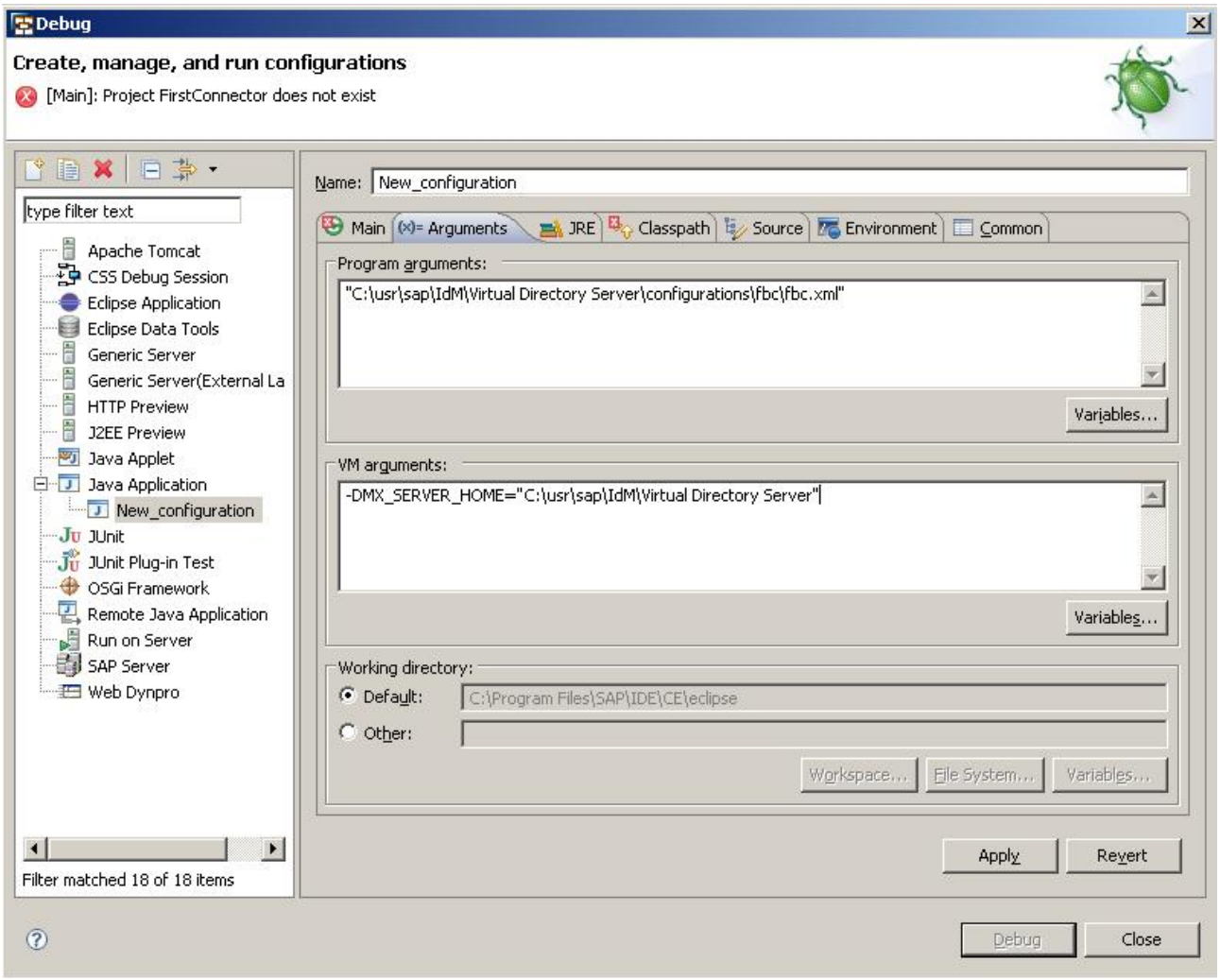
In order to enable logging create a file *standalonelog.prop* in the work area of the Virtual Directory Server configuration and enter the following:

```
LEVEL=DEBUG
EXTENSIONLEVEL=DEBUG
```

You will need the SAP Standalone Log Viewer. You can read more about logging in the document *SAP NetWeaver Identity Management Operations Guide*.

SAP NetWeaver Developer Studio

Create a debugging configuration for your project in the SAP NetWeaver Developer Studio.



In the "Main" tab enter the following:

- Main Class: com.sap.idm.vds.MVDServer

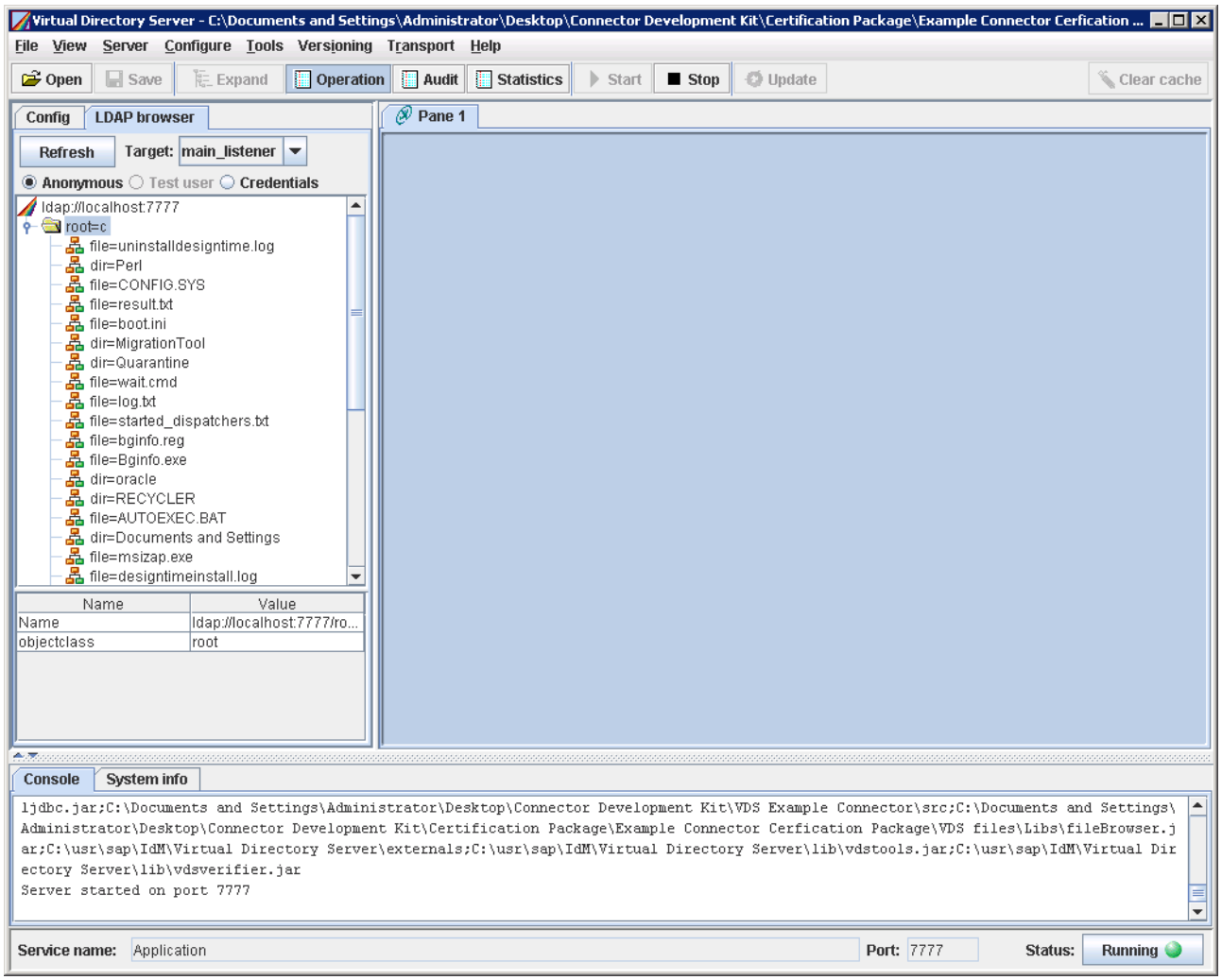
In the "Arguments" tab enter the following:

- Program Arguments: Full path to the Virtual Directory Server configuration you created.
- VM Arguments: -DMX_SERVER_HOME=<path to the VDS installation directory>. Installation directory by default is *C:\Program Files\SAP\IdM\Virtual Directory Server* for SAP NetWeaver Identity Management 7.1 and *C:\usr\sap\IdM\Virtual Directory Server* for SAP NetWeaver Identity Management 7.2.

Now set your break points in the source code and start debugging the Virtual Directory Server. The console output will show the messages of the Virtual Directory Server (e.g. "Server started on port 7777").

Testing using the internal LDAP browser

The Virtual Directory Server's internal LDAP browser provides the possibility to search through your connected backend. Select the "LDAP browser" tab:



To expand the node, double-click the node or select it and select "Do one-level search on this node" from the context menu. This will show all nodes below the selected one.

For displaying properties of the node, select "Do base search on this node" from the context menu.

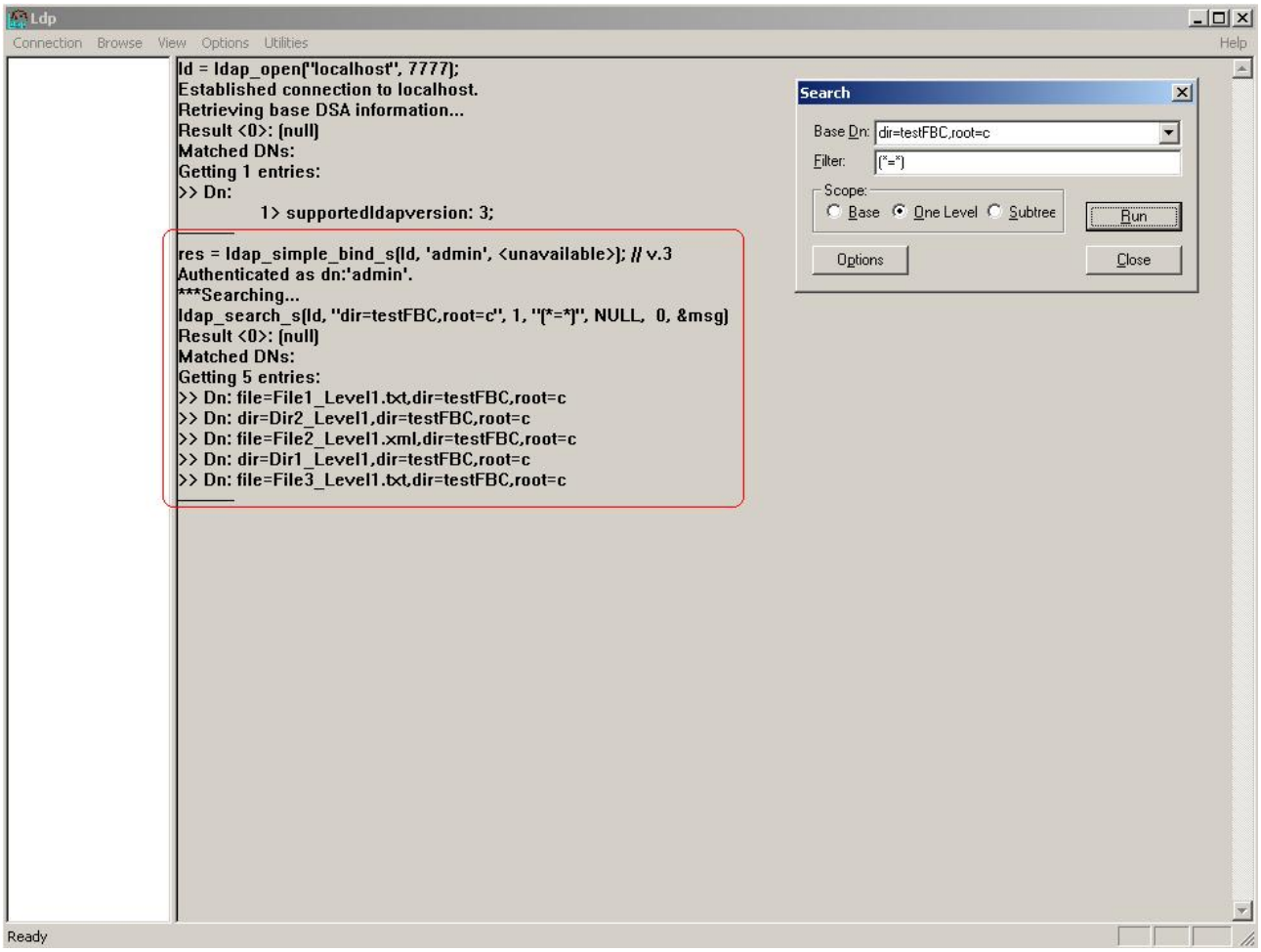
Testing using an external tool (LDP)

Using an external tool is recommended for testing add, modify and delete tasks.

Searching using LDP

To perform a search, do the following:

1. Select **Browse/Search**.
2. As base DN use *dir=testFBC,root=c*, where "testFBC" is a directory on your C: drive.
3. Choose "Run".

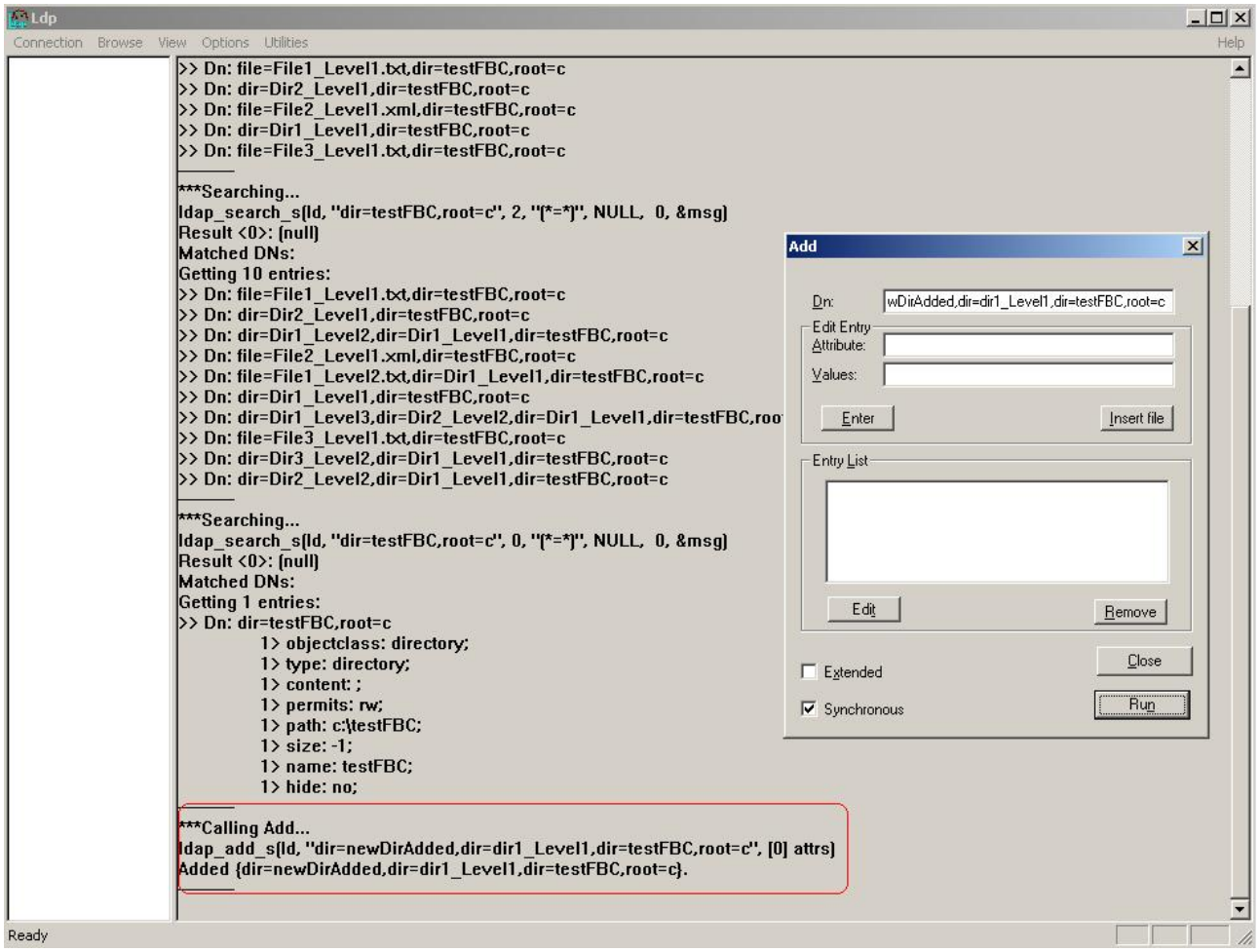


This will display the search result in the window.

Adding using LDP

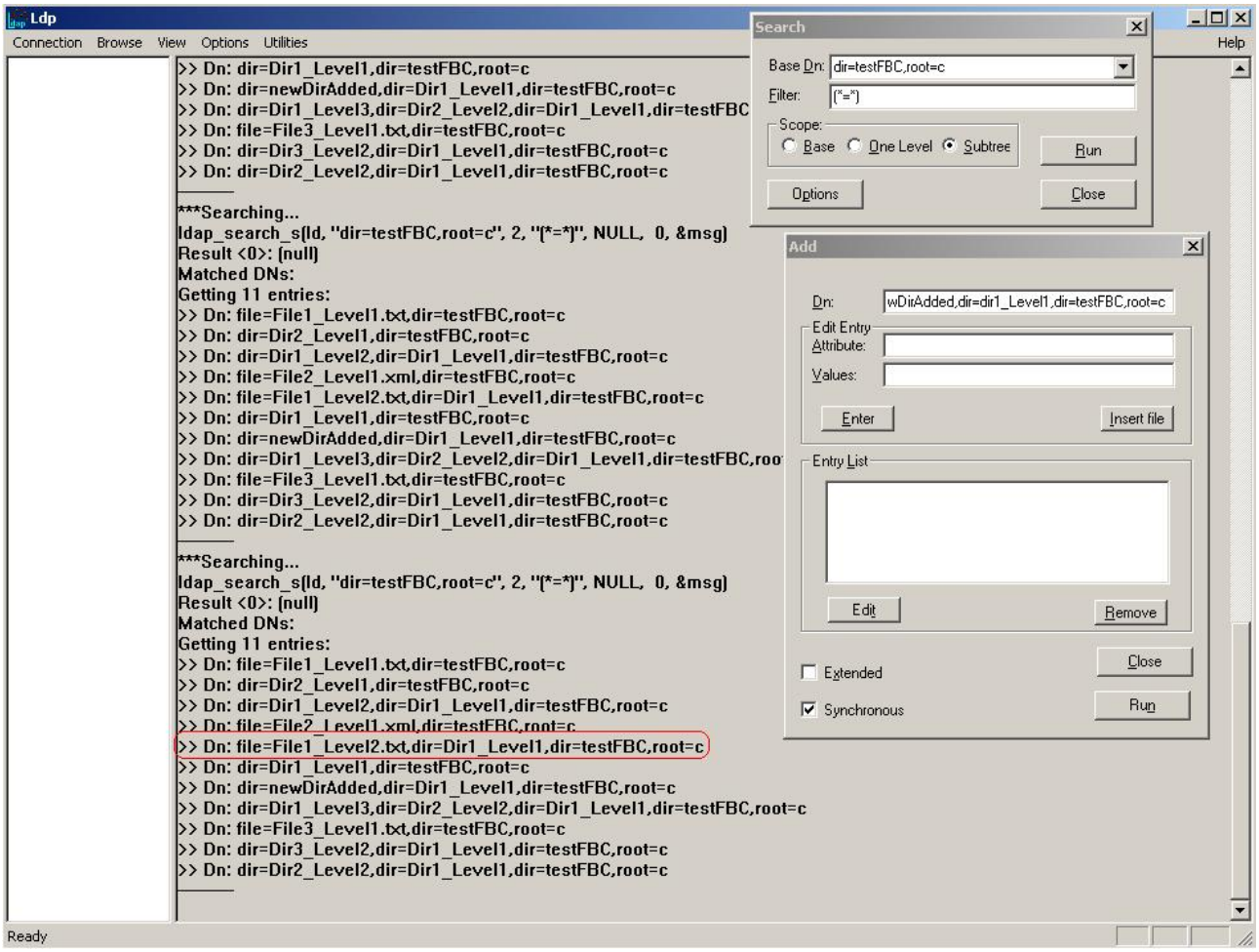
To perform add, do the following:

1. Select "Add child" in the "Browse" menu to create files and/or directories.
2. Create a directory with e.g. DN *dir=newDirAdded,dir=dir1_Level1,dir=testFBC,root=c*.
3. Choose "Run".



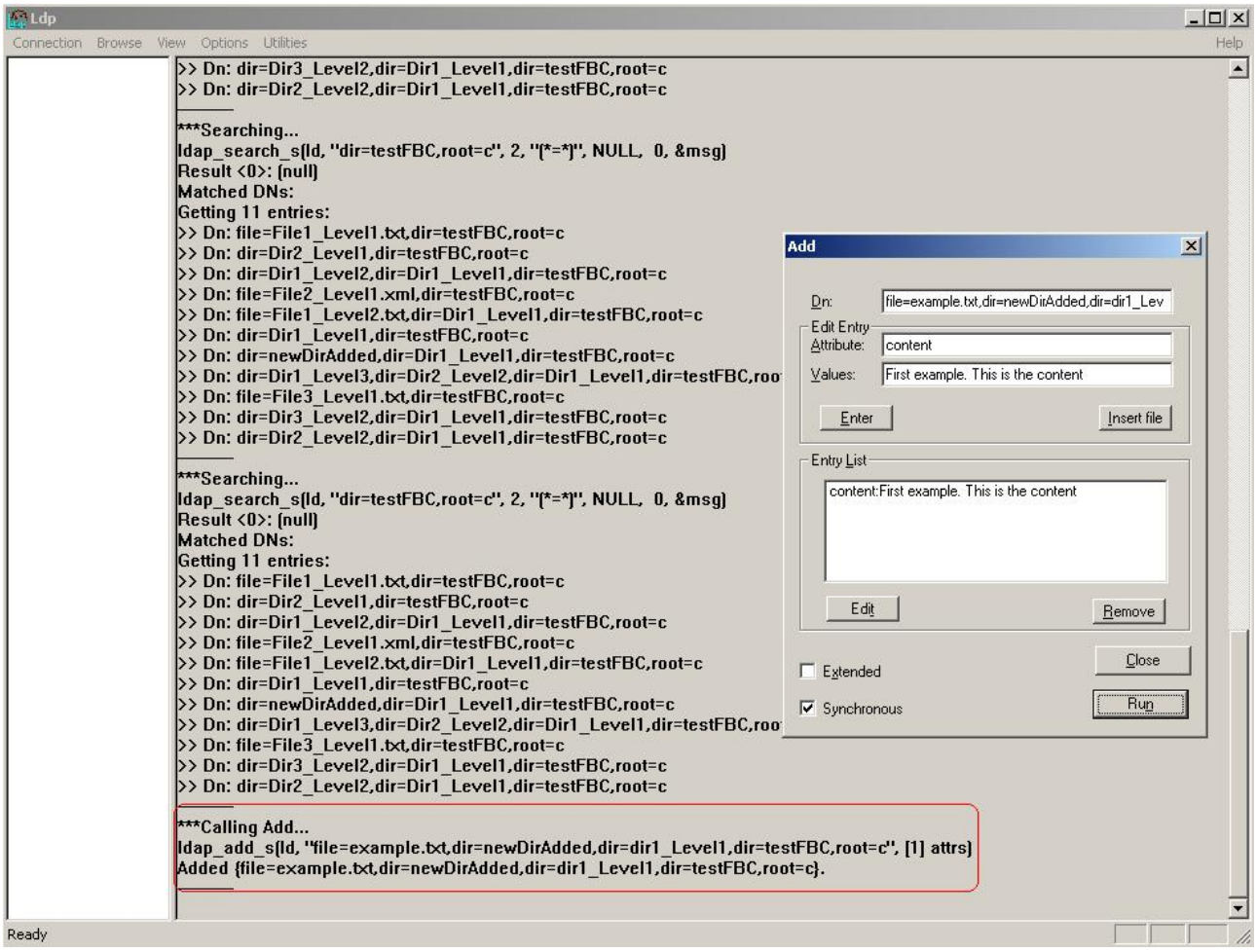
It is displayed that the add operation is called, and the new directory added.

To check if the file was created, make a search with *dir=testFBC,root=c* as starting point.



Example – Creating a new file with content:

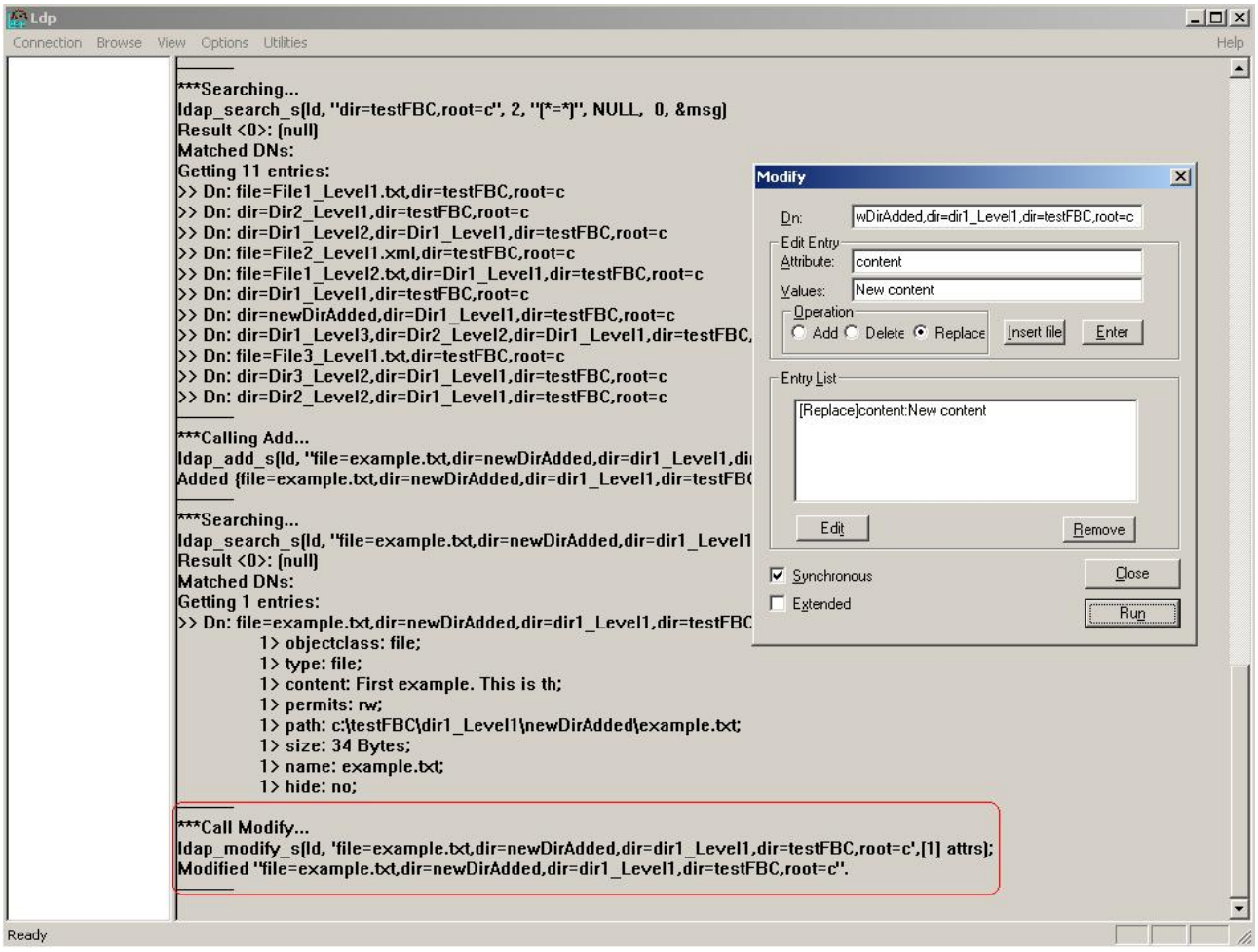
Use DN *file=example.txt,dir=newDirAdded,dir=dir1_Level1,dir=testFBC,root=c* and add an attribute named "content" with the value "First example. This is the content".



Modifying using LDP

To modify using LDP, do the following:

1. Select **Browse/Modify** to modify the existing files and/or directories, e.g. overwrite the content of the previously created file.
2. Enter the following values:
   - As the DN use *file=example.txt,dir=newDirAdded,dir=dir1_Level1,dir=testFBC,root=c*.
   - Add an attribute named "content" with the value "New Content".
   - Select the operation "Replace" to overwrite the existing content.
3. Choose "Run".



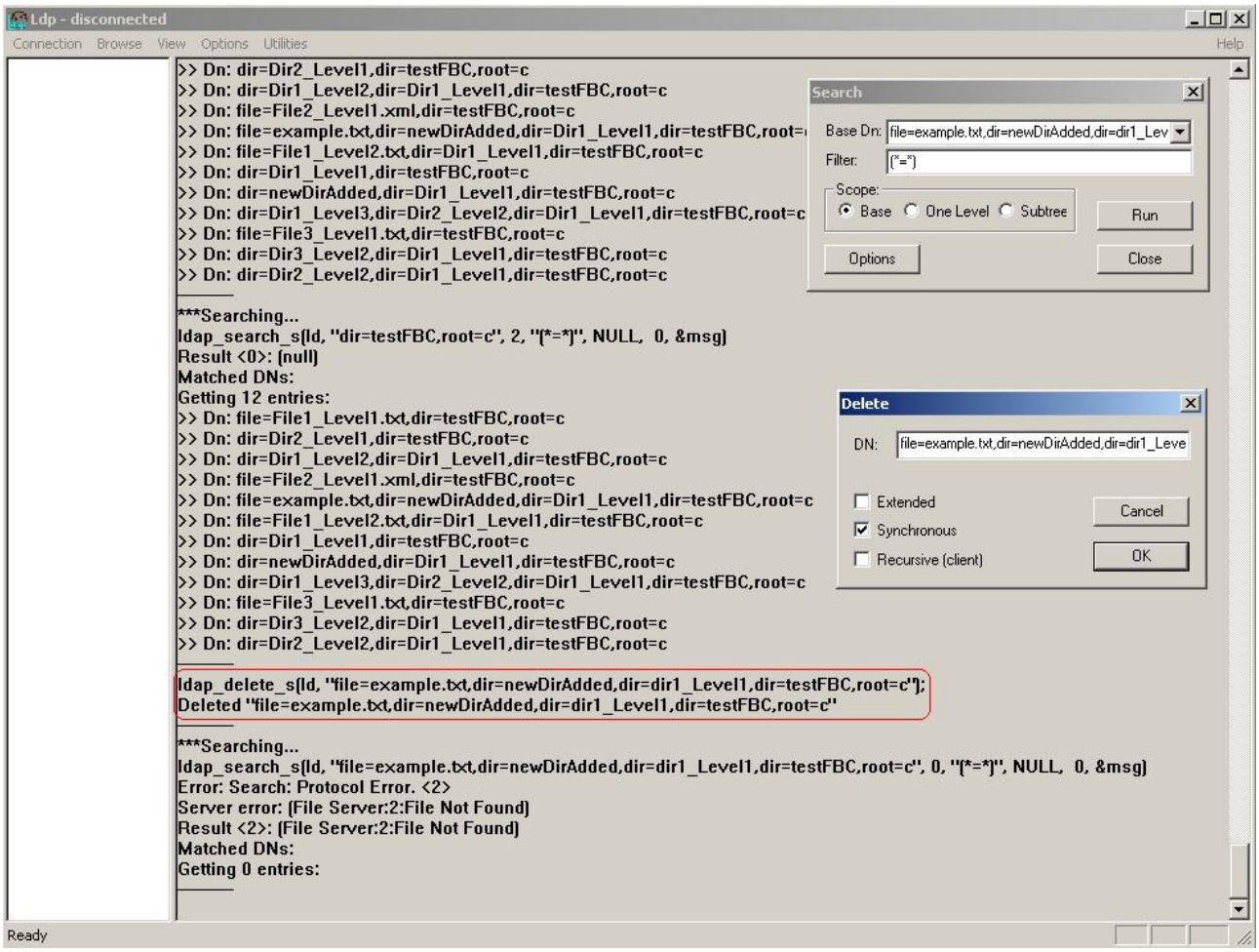
The modify operation is called and the file modified.

Deleting using LDP

To delete using LDP, do the following:

1. Select **Browse/Delete** to delete the existing files and/or empty the directories.
2. Delete e.g. the previously created file with the DN *file=example.txt,dir=newDirAdded,dir=dir1_Level1,dir=testFBC,root=c*.

3. Choose "Run".



The delete operation is called and the file deleted. You may verify that the file is deleted by performing a search for the given file.

Deliverables

The partner responsible for the target application integration is also responsible for providing a set of deliverables:

- Necessary target application Java libraries.
- An initial data set needs to be provided by the partner (the number of entries has to be at least 300). It is a partner's responsibility also to provide the necessary documentation for the correct installation of the initial data set.
- Identity Center, including the Identity Management User Interface, deliverables:
  - Job and the connector tasks in the Identity Center necessary for the certification process, with installation and configuration documentation (a repository definition, a job which reads entries from the target application and a search job, a job for creating of account privilege, tasks for adding, modifying and deleting a user, and a User Interface task for editing of a user) as described in *SAP NetWeaver Identity Management Connector Development Kit Certification.*
  - (Optional) Identity store schema extension: if the connector requires any identity store schema extensions, these need to be provided for import (an exported .mcc file). Store the exported schema file the folder *IC files* in the connector certification package as described in *SAP NetWeaver Identity Management Connector Development Kit Certification*.
- Virtual Directory Server deliverables (Optional, not to be delivered if vendor specific (connector) module is used instead of Virtual Directory Server):
  - Virtual Directory Server (VDS) configuration (as template) explained in the section *Creating the Virtual Directory Server configuration as template* below.
  - (Optional) Data Source configuration (as template) explained in the section *Creating the Data Source configuration as template* on page 45.
  - Connector class (compiled for JDK 1.4 or 1.5) or as source file (stored in the Virtual Directory Server configuration file). The class is stored in *<VDS_dir>\configurations* in a map with the same name as your configuration file.

  Note:
  Whether the class should be compiled for JDK 1.4 or 1.5 will depend on which AS Java version the connector class is to be deployed on. Compiling the class for JDK 1.4, the Virtual Directory Server configuration that utilizes this class will be deployable on AS Java versions 7.0, EHP 1 for SAP NW CE 7.1, SAP NW CE 7.2 and SAP NW 7.3. Compiling for JDK 1.5 will result in connector not being deployable on AS Java 7.0 versions.
- Vendor specific (connector) module (Optional, only delivered if used instead of Virtual Directory Server).
- Documentation deliverables:
  - Functionality description sheet which documents the purpose of the connector, which features are supported, as well as entry types and attributes. Also installation job/description for the connector tasks in the Identity Center and the repository definitions and installation job/description for the VDS connector (configuration with the code and the API) need to be included in the documentation. The documentation must be submitted in English only. See *SAP NetWeaver Identity Management Connector Development Kit Certification* for details.

Creating the Virtual Directory Server configuration as template

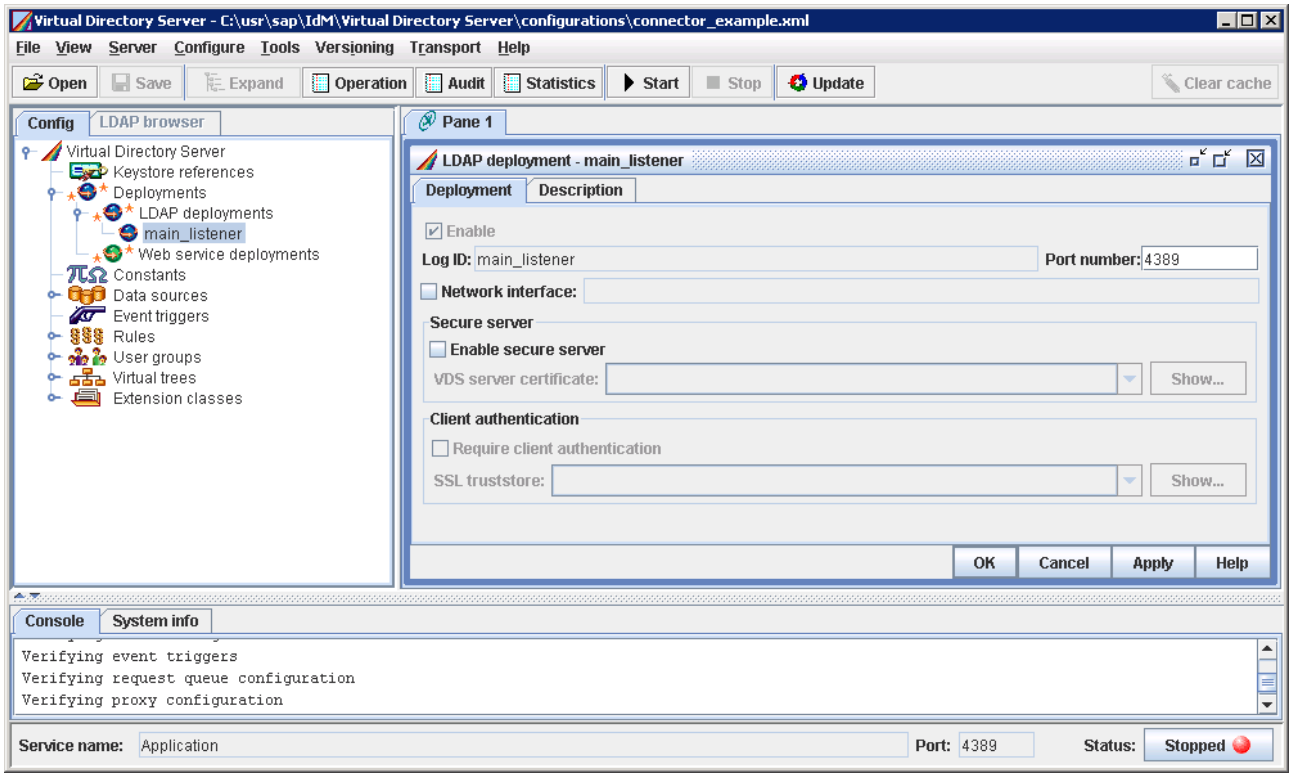
The server configuration is stored in an XML file. When creating a server configuration, you specify a file name that is used to store the configuration. This file is normally stored in *<installation directory for Virtual Directory Server>\configurations*. Before the configuration file can be used as template, some constants should be defined. What constants are useful to define will vary from time to time, but usually the following is of interest:

- VDS server port number
- Data source parameters (the parameters will depend on the data source type, e.g. LDAP, data base etc.):
  - Name used to identify the data source (display name)
  - Server (server's host name or IP address)
  - Port number
  - Directory server's starting point
  - User name to access the server
  - Password to access the server
  - Database (JDBC URL to access the database)

Creating the constants

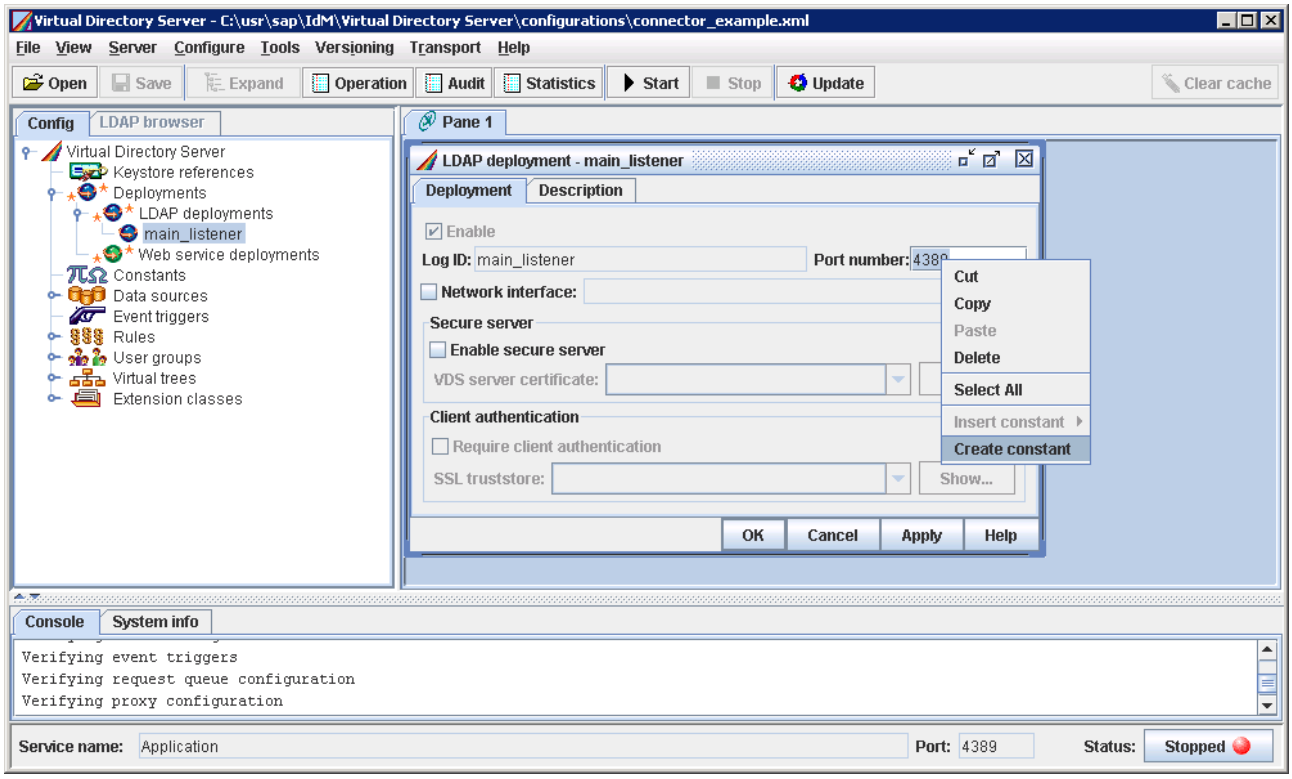
To create a constant for the Virtual Directory Server parameter port number in your Virtual Directory Server configuration, do the following:

1. Navigate to and select the node "main_listener" in the configuration tree, then view the properties either by double-clicking the node or selecting "Properties…" from the context menu.

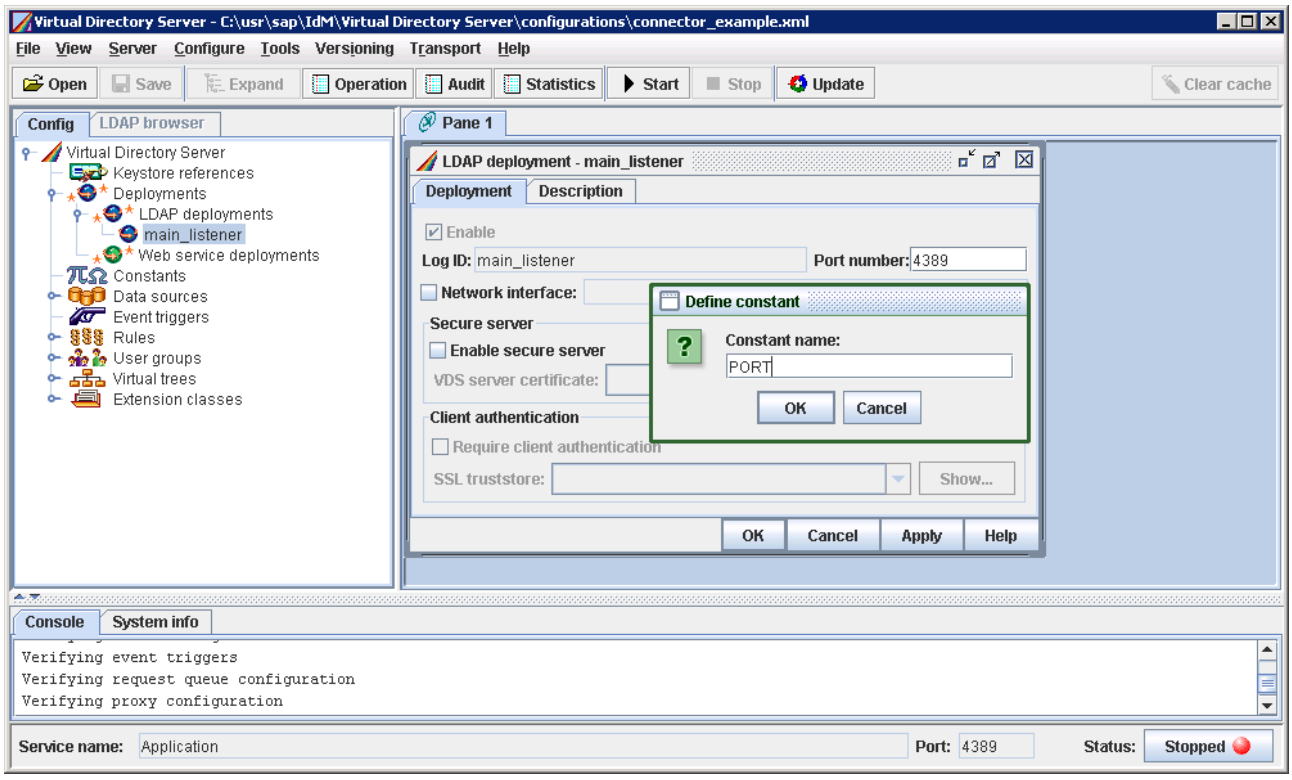


This will open the properties dialog box.

2. Select the parameter value and select "Create constant" from the context menu.



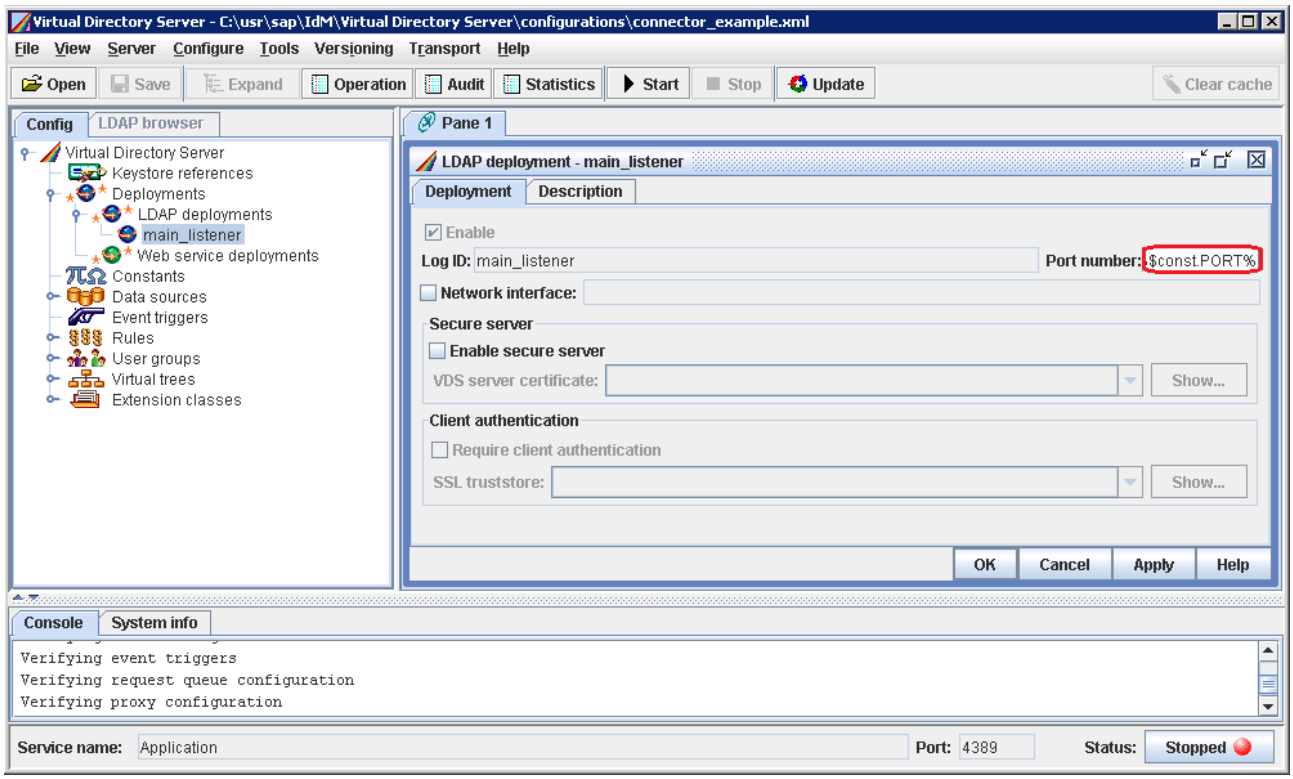
This opens the "Define constant" dialog box:



Enter a name for the constant.

3. Choose "OK" to close the dialog box and add the constant.

The constant is now added:

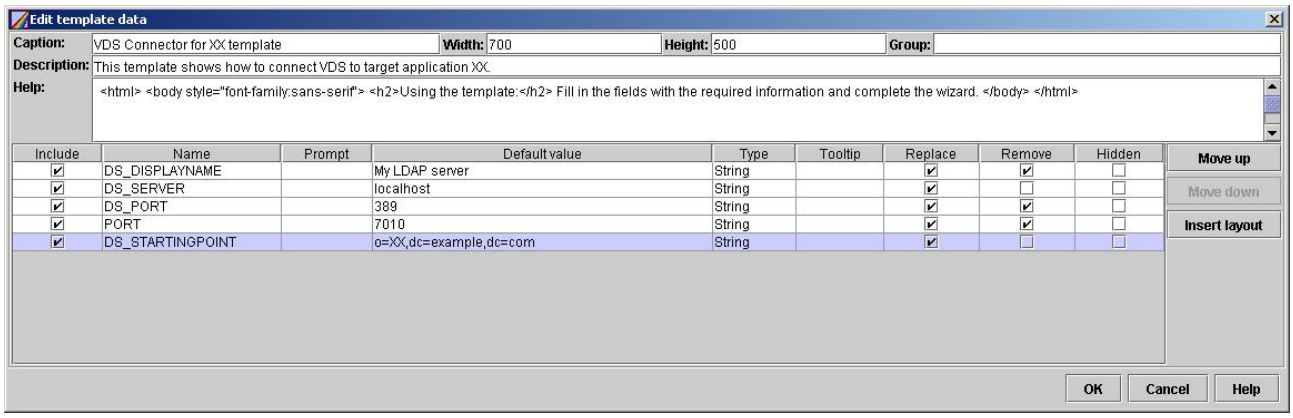


4. Choose "OK" to confirm and close the properties dialog box.
5. Now repeat the previous steps for all the parameters you wish to crate constants for.

Editing the template data

The template data can be edited. To edit, do the following:

1. In the Virtual Directory Server console, select **Tools/Edit template data…** from the main menu. This will open "Edit template data" dialog box:



Fill in the fields:

Caption:
This will be displayed in th e title bar of the wizard dialog box.

Width/Height:
This will define the width and the height of the wizard dialog box.

Group:
The configuration templates are grouped to facilitate locating the correct template. Enter the name of the group that the template should belong to, e.g. *LDAP*, *Database*, *Generic*, etc. If nothing is specified in the field, the template will be placed in the *Generic* group.

Help:
Here you enter a text that describes the necessary steps to complete the wizard. This can be prerequisites and/or information about how to fill in the fields in the wizard. The text can be in HTML syntax.

Controls:
The list contains all constants that were defined. You can change the order of the controls by selecting a control and choosing "Move up" or "Move down". You can also add layout elements (separator, label, filler, group, tab) by choosing "Insert layout". The columns are:

Include:
Select the check box to include the control in the wizard.

Name:
This is the name of the constant (or in the case of layout elements "Layout").

Prompt:
If defined, this will be displayed in the wizard instead of the constant name.

Default:
The default value of the constant can be entered here.

Type:
Select a type for the control (String, Boolean, FileName, JDBCURL, Password).

Tooltip:
Enter here the text that is used as a tooltip for the field.

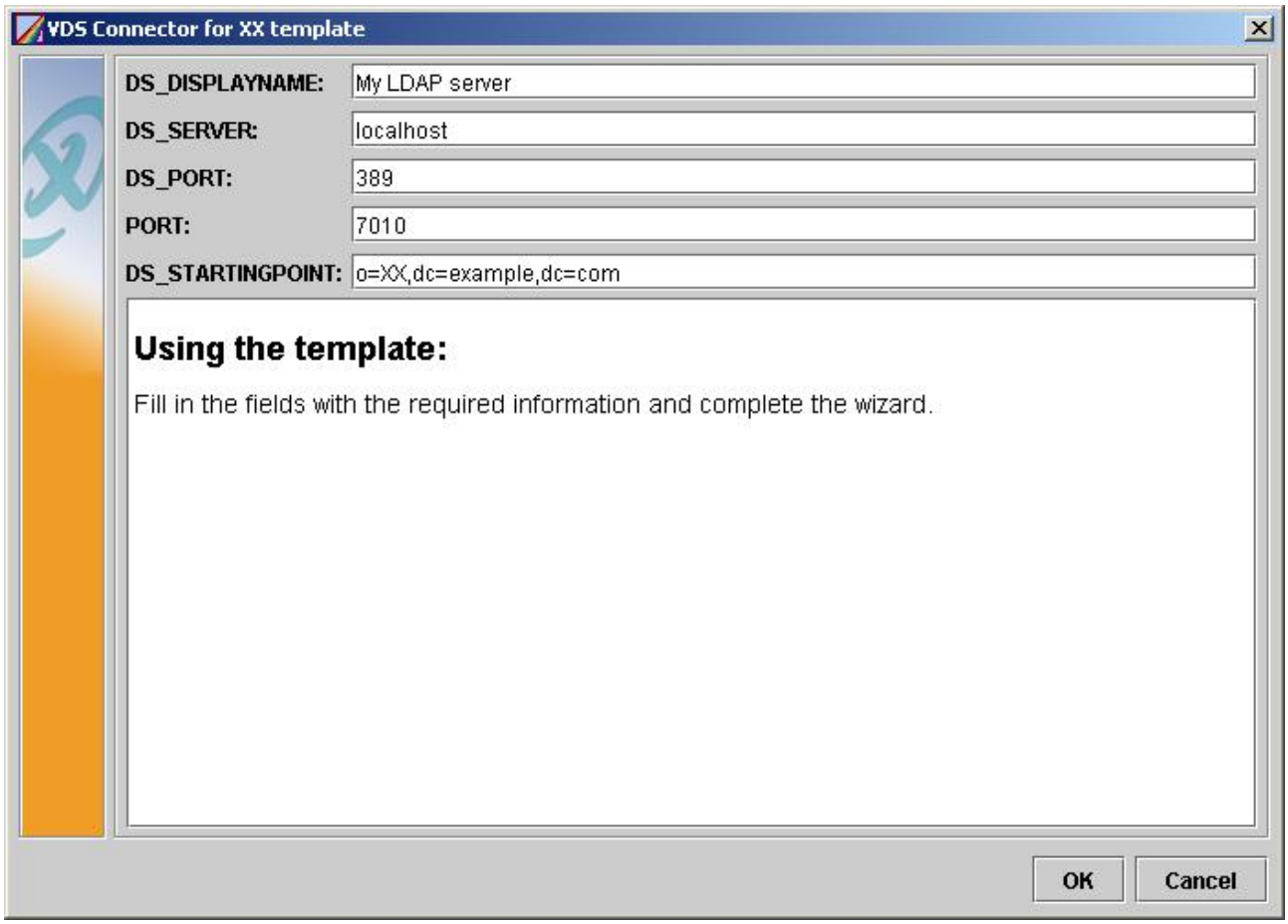
Replace:
Select to specify that the constant should be replaced with the constant value, removing the reference to the constant.

Remove:
Select to specify that this constant also should be replaced from the list of constants. Should be used with the "Replace" option.

Hidden:
Select to specify that the control should not be displayed in the template wizard. A default value should be specified for the controls that are not displayed.

2. Choose "OK" to confirm and to close the dialog box.

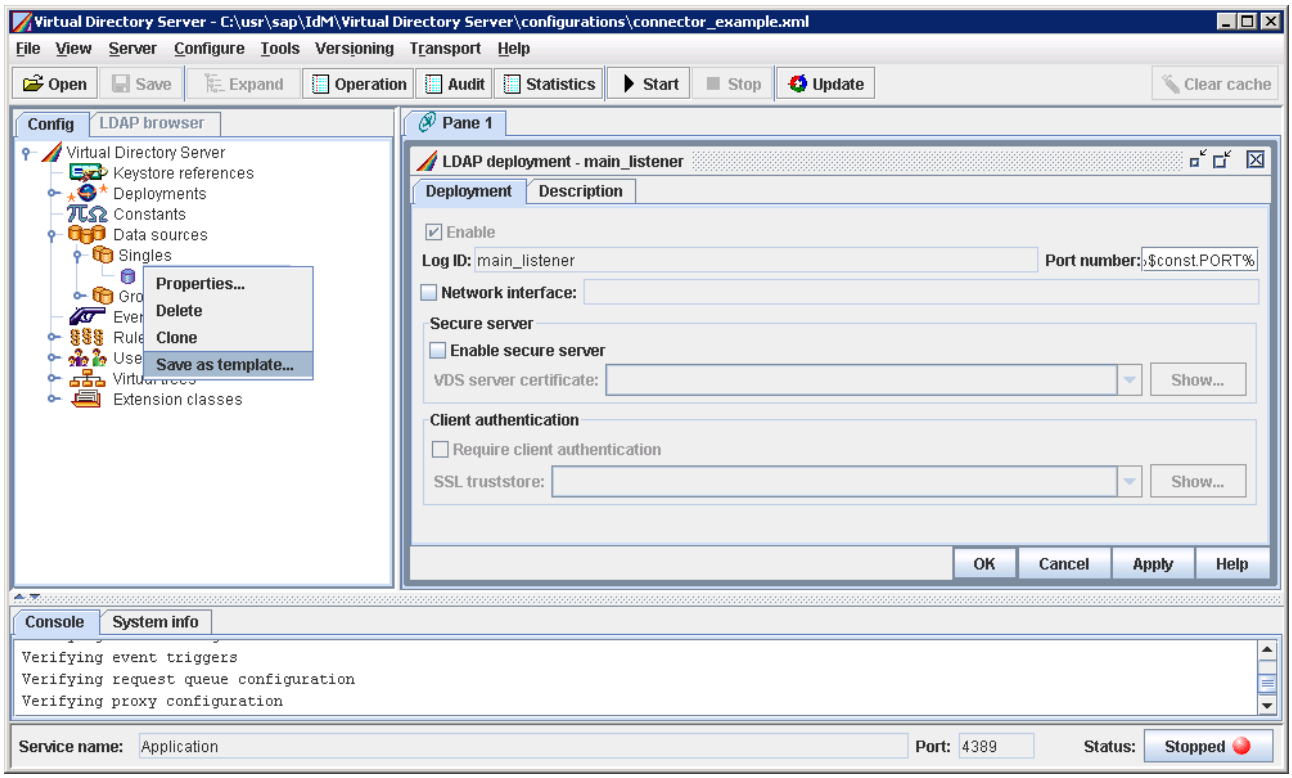
The template wizard might look something like this:



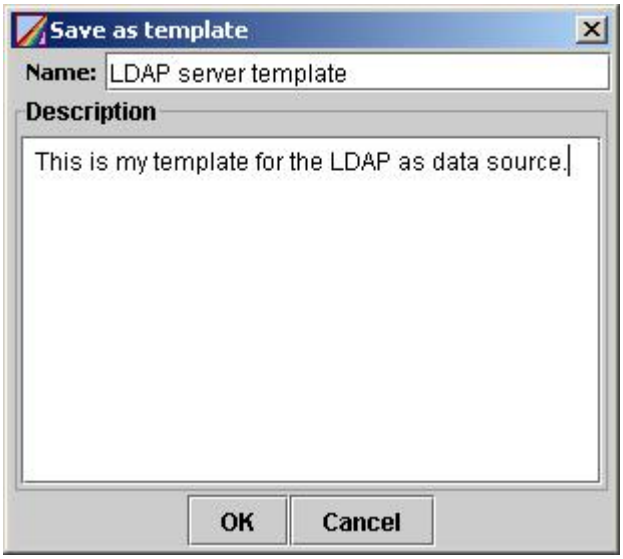
Creating the Data Source configuration as template

To create the Data Source configuration as a template, do the following:

1. Create the constants for the parameters you wish (if not already created).
2. Select the data source in the configuration tree and select "Save as template…" from the context menu.



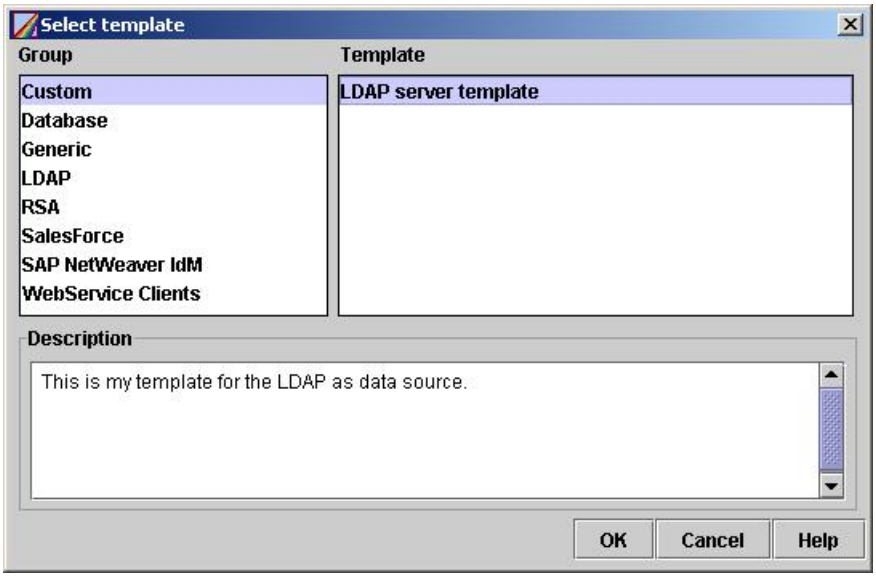
This opens up a "Save as template" dialog box:



Enter the name for the data source template and the short description.

3. Choose "OK" to confirm and to close the dialog box.

The data source template is stored under "Custom" template group in the template wizard for the data source (to open, navigate to and select Data sources/Singles and select "New…" from the context menu):



Sample code: FileBrowserConnector

All operations/methods described in section *Virtual Directory Server structures* on page 4 are implemented in the *FileBrowserConnector* sample, which uses *fileBrowserAPI.jar* as the target library. This jar file exposes the low-level API for listing/adding and (limited) manipulation of the text files.

It acts as an LDAP File Browser – i.e. executing the valid LDAP operations makes it possible to list/add/modify/remove text files on the disk.

Only search and add method are shown here. The additional comments (sample comments) are added here – they are correlating the code with the previously discussed development phases.

Full sample code listing is given as a Java file in the sample package.

Search method

```
     /**
      * Realizes the search operation according to one of the next options:
      * - Base search -> Gives a set of attributes of a file
      * - One level search -> Gives the set of adjoining files and directories
      * under a directory
      * - Subtree search -> Gives the complete subtree of files and directories
      * under a directory
      * @param param -> set of LDAP parameters
      * @return -> The result indicating success, or indicating error if some
      * error occurred
      */

public MVDSearchResults search (MVDHashMap param) {

// --- Sample Comment
// --- Create resulting VDS structure

MVDSearchResults result = new MVDSearchResults();

// --- Sample Comment: Fetching necessary parameters from the HashMap.
// --- Some of the parameters are configured on
// --- Data Source (prefix DS_, see the sample VDS configuration file) while
// --- others are coming from the inbound operation (prefix LDAP_).
// --- This phase is important and quite common for each of the connectors.

   /* Gets the name to refer directories in DN */
   String dn_dir = (String)param.getDSMandatoryParameter("DN_DIRECTORY");

   /* Gets the name to refer normal files in DN */
   String dn_file = (String)param.getDSMandatoryParameter("DN_FILE");

   /* Gets the starting point parameter */
   String aSp = (String)param.getLdapMandatoryParameter("STARTINGPOINT");

   /* Gets the search type */
   int searchOpt = ((Integer)
param.getLdapMandatoryParameter("OPSUBTYPE")).intValue();

   /* Gets the filter */
   String filter = (String)param.getLdapMandatoryParameter("URLFILTER");

// --- Sample Comment:  Pre-processing input data.
// --- It is very connector specific information here.

   /* Checks if the filter syntax is correct */
   String checkFilter = GenericOperations.cleanTrailingAndLeadingSpaces(filter);
   if (checkFilter.endsWith(")")==false || checkFilter.startsWith("(")==false ||
     checkFilter.indexOf("=")<0) {

// --- Sample Comment Sending result back.
// --- In this case – no results are fetched and the result code is set to
specific code
```

```
     result.setError(87, "Invalid filter format");
            return result;
   }

   /* Transforms the starting point in LDAP format to a understandable format for
the operating system */
       Vector v = this.DNtoOS(aSp,true,dn_dir,dn_file);
       if (v==null) {
            result.setError(34, "Error in DN format for this operation");
            return result;
       }
   /* Gets the complete path where to do the search from */
   String path = (String)v.get(PATH);
   /* If path is null means that something was not correct in the DN syntax */
   if (path==null) {
            result.setError(21, "Null Path");
            return result;
   }

   /* Checks if it is expected a directory as starting point */
   boolean isDNDirectory =
(((String)v.get(FILE_TYPE)).equalsIgnoreCase(dn_file))==false;

   /* Memorizes name of the root of the DN */
   String root_name = (String)v.get(ROOT_NAME);

// --- Sample Comment
// --- Finally, the API from the target JAR file is executed


   /* Does the search */
     FileSearch fs = new FileSearch();
     int resCode = fs.search(path, filter, searchOpt, isDNDirectory);
     if (resCode!=FileConstants.SUCCESS) {
            result.setError(FileConstants.LDAP_CODES[resCode],
FileConstants.MESSAGES[resCode]);
            return result;
     }

     if (isDNDirectory==false && searchOpt!=FileConstants.BASE_SEARCH) {
            result.setError(21, "This search operation is only possible for
directories");
            return result;
     }

     HashMap res = fs.getSearchResult();


// --- Sample Comment
// --- The returned structures do not fit into the VDS results - they have to
converted

     /* If the option is base search then ... */
   if (searchOpt==BASE_SEARCH) {
            if (res.size()>0) {
                   /* ... the entry is a file attribute and one of its values */

// --- Sample Comment
// --- Creating a VDS entry object that will be filled in with obtained

        MVDSearchResultEntry e = new  MVDSearchResultEntry();
            e.setDn(aSp);
        /* Adds to the entry the set of attributes and values received from the
call to the search method of the API file browser connector */
                   for (Iterator it=res.keySet().iterator(); it.hasNext(); ) {
                          String key = (String)it.next();
                          String val = (String)res.get(key);
                          e.setAttrValue(key,true,val);
                   }
                   /* Adds the entry to the result */
                   result.add(e);
            }
        }
        else {
                   /* Creates the result to be returned */
                   for (Iterator it = res.keySet().iterator(); it.hasNext(); ) {
```

```
                        String key = (String)it.next();
                    HashMap aType = (HashMap)res.get(key);
                        MVDSearchResultEntry e = new  MVDSearchResultEntry();
                        String fileType = (String)aType.get("objectclass");
                        /* ... the entry is a DN indicating a path file or
directory */

     e.setDn(this.OStoDN(key,fileType,root_name,dn_dir,dn_file));

e.setAttrValue("objectclass",true,(String)aType.get("objectclass"));
                e.setAttrValue("name",true,(String)aType.get("name"));
                result.add(e);
                   }
        }

            result.setOK();
        return result;
    }
```

Operational parameters (for each operation)

Search operation

Key	Value	Type
LDAP_STARTINGPOINT	The effective starting point for this node operation of this search request. (Note! One search to the VDS could result in several back-end searches, each with its own starting point).	*String, DN format*
LDAP_SZLIMIT	The size limit for this request.	*Integer*
LDAP_TMLIMIT	The time limit for this request.	*Integer*
LDAP_OPSUBTYPE	The search operation type for this node: 0 for BASE-Level 1 for ONE-Level 2 for SUB-Level	*Integer*
LDAP_ATTRS	A list of requested attributes (after the cleaning and conversion executed by the core VDS code). First element contains "ALLATTRIBUTESCLEANED", if all attributes were cleaned in the process mentioned above.	*Vector*
LDAP_DBFILTER	The requested filter in the S*QL* format (WHERE ...).	*String*
LDAP_URLFILTER	The requested *LDAP* filter in the *URL* format (after the cleaning and conversion). "(IGNOREDFILTER)" - if all of the filter attributes are in the list of ignored attributes.	*String*
LDAP__ORIGINALSTARTINGPOINT	The original starting point of the request (rarely used – the LDAP_STARTINGPOINT is more important one).	*String*
LDAP__ORIGINALSEARCHTYPE	The original search operation type of the request.	*Integer*

Modify operation

Operational parameters for the Modify operation are:

Key	Value	Type
LDAP_DN	The *Distinguished Name* of the entry on which to perform the modify operation.	*String, DN Fornat*
LDAP_DATA	The list of attributes to be modified, as well as values and desired modification mode.	*Vector of MVDModAttrValues*

Add operation

Key	Value	Type
LDAP_DN	The *Distinguished Name* of the new entry.	*String, DN format*
LDAP_DATA	The attributes of the new entry.	*HashMap*

Delete operation

Key	Value	Type
LDAP_DN	The *Distinguished Name* of the entry to be deleted.	String