Oracle Endeca Platform Services

Advanced JDBC Column Handler Usage Guide

Version: 11.0
January 2014

The Advanced JDBC Column Handler is an extension to the standard Endeca JDBC record adapter. It provides support for obtaining data from database column types that are not supported by the standard Endeca JDBC record adapter, such as CLOBs and BLOBs.

Table of Contents

Introduction

The Endeca Data Transformation Layer provides a Java-based database adapter for use with any database that has a JDBC driver. This JDBC adapter can be used as a record adapter type to retrieve Endeca records as rows of a SQL query result. Not all database column types are supported by this out-of-the-box Endeca JDBC adapter (see the JDBC Input Format section of Endeca Developer Studio Help for a list of supported database column types). The Advanced JDBC Column Handler extends the set of supported database column types by providing handlers for the following column types:

- LONGVARCHAR
- CLOB
- BLOB
- LONGVARBINARY
- BINARY
- VARBINARY

System Requirements

Endeca Requirements
The Advanced JDBC Column Handler works with Endeca versions 4.8.x or later. There are no special system requirements beyond the installation of Oracle Endeca Guided Search.

JDBC Driver
The JDBC Driver for the specific database type is required for using the JDBC record adapter and should be located somewhere on the file system.

Installation

The Advanced JDBC Column Handler is distributed as a zip file (**advJdbcColumnHandler-[VERSION].zip**) which is a self-contained tree. The file can be unpacked at any location using WinZip, or any other compression utility that supports this format. Unpacking the file creates the subdirectory structure **Endeca\Solutions\advJdbcColumnHandler-[VERSION]\.**

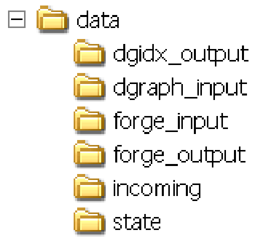
Once the distribution has been unpacked, copy the **AdvJDBCColumnHandler.jar** file somewhere within the local project directory tree. Common locations for this file are within a lib/ or java/ subdirectory.

Configuration Options
Also, if the column handler's default settings should be overridden, copy the **sample/sample_columnHandler.properties** file to somewhere within the local project directory tree. For example, place this file in the lib/ or java/ subdirectory. Be sure to rename the file to **columnHandler.properties**. The follow table lists the configuration options used in the **columnHandler.properties** file:

binaryDataChunkSize	The size (in bytes) of the incremental data chunk read and written while processing binary columns. This is a performance optimization setting; changing this value will not affect the values ultimately returned by the Advanced JDBC Column Handler.

	Default setting: 1024
charDataChunkSize	The size (in bytes) of the incremental data chunk read and written while processing character-data columns. This is a performance optimization setting; changing this value will not affect the values ultimately returned by the Advanced JDBC Column Handler. Default setting: 1024
outputDataDir	The directory to which data files will be written (see File System Output). Relative paths to an output directory will be relative to forge's working directory. Default setting: ../incoming
charDataToDisk	If this setting is "true" character data will be written out to the filesystem (encoded according to the value of the charDataOutputEnc property), and the path to the output file will be returned as the property's value. If false, the character data will be returned as the property's value. Set this option to true for very large char columns. Default setting: false
charDataOutputEnc	Output encoding to use when writing character data to disk. A valid Java charset name must be specified. Some common encoding charsets are listed below: • `US-ASCII` • `ISO-8859-1` • `UTF-8` • `UTF-16BE` • `UTF-16LE` • `UTF-16` This setting is only used if **charDataToDisk** is "true". Default setting: UTF-8

Storing Data on Disk
If binary columns will be used, or if character data columns will be output to the filesystem, you must be sure the output directory exists. Unless overridden by the **outputDataDir** setting described above, the default output directory will be the `incoming` directory parallel to Forge's working directory. For example, the following directory tree shows this `incoming` directory, assuming `forge_input` is configured as forge's working directory:



Note the `incoming` directory is accessible at `../incoming` relative to the pipeline files within `forge_input/`.

Upon initialization, the Advanced JDBC Column Handler will check to see if the **outputDataDir** is writable. If it is not, the Advanced JDBC Column Handler will throw a warning into the Forge log, similar to the following:

```
WARN    09/24/06 14:00:49.958 UTC       FORGE      {forge,baseline}:
(com.endeca.soleng.itl.jdbc.AdvancedJDBCColumnHandler): outputDataDir
../incoming not writable
```

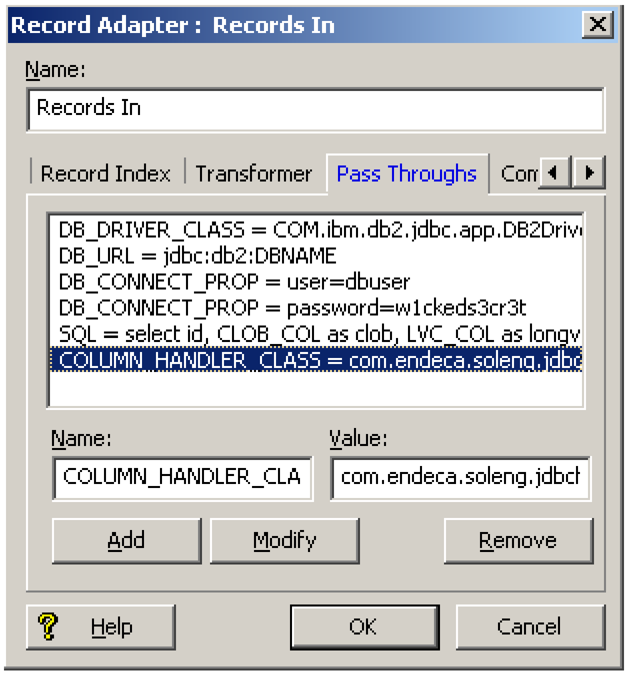
Because the Advanced JDBC Column Handler does not require a writable directory if only character columns are used and character data is not spooled to disk, Forge will continue processing when it encounters a non-writable **outputDataDir**. However, if binary columns are used or if character data is explicitly spooled to disk, Forge will not return any valid data for those columns, and will log additional warnings to the Forge log for each row and column in the database it is not able to process. These warnings will appear similar to the following:

```
WARN    09/24/06 14:02:34.828 UTC       FORGE      {forge,baseline}:
(com.endeca.soleng.itl.jdbc.AdvancedJDBCColumnHandler): IOException while
dumping MyBinaryData column
```

Usage

Create a JDBC record adapter as usual, with PASS_THROUGH values like DB_DRIVER_CLASS, DB_URL, and SQL. See the JDBC section in Endeca Developer Studio Help for more information on how to create a JDBC record adapter.

Add one new PASS_THROUGH with name COLUMN_HANDLER_CLASS and value **com.endeca.soleng.itl.jdbc.AdvancedJDBCColumnHandler**. Below is an example of a complete JDBC record adapter with this additional pass through:



```
DB_DRIVER_CLASS = COM.ibm.db2.jdbc.app.DB2Driv
DB_URL = jdbc:db2:DBNAME
DB_CONNECT_PROP = user=dbuser
DB_CONNECT_PROP = password=w1ckeds3cr3t
SQL = select id, CLOB_COL as clob, LVC_COL as longv
COLUMN_HANDLER_CLASS = com.endeca.soleng.jdbc
```

Name: COLUMN_HANDLER_CLA
Value: com.endeca.soleng.jdbcl

This example generates the following code:

```
<RECORD_ADAPTER COL_DELIMITER="" DIRECTION="INPUT"
FILTER_EMPTY_PROPS="TRUE" FORMAT="JDBC" FRC_PVAL_IDX="FALSE"
MULTI="FALSE" NAME="Records In" PREFIX="" REC_DELIMITER=""
REQUIRE_DATA="TRUE" ROW_DELIMITER="" STATE="FALSE" URL="">

       <COMMENT>My JDBC test adapter</COMMENT>
       <PASS_THROUGH NAME="DB_DRIVER_CLASS">
       COM.ibm.db2.jdbc.app.DB2Driver
       </PASS_THROUGH>

       <PASS_THROUGH NAME="DB_URL">
       jdbc:db2:DBNAME
       </PASS_THROUGH>

       <PASS_THROUGH NAME="DB_CONNECT_PROP">
       user=dbuser
       </PASS_THROUGH>

       <PASS_THROUGH NAME="DB_CONNECT_PROP">
       password=w1ckeds3cr3t
       </PASS_THROUGH>

       <PASS_THROUGH NAME="SQL">
       select id, CLOB_COL as clob, LVC_COL as longvarchar, RESUME_FILE
       as blob from SCHEMA.SUMTABLE_T fetch first 10 rows only
       </PASS_THROUGH>

       <PASS_THROUGH NAME="COLUMN_HANDLER_CLASS">
       com.endeca.soleng.itl.jdbc.AdvancedJDBCColumnHandler
       </PASS_THROUGH>

</RECORD_ADAPTER>
```

Whether forge will be run on the command line or from a control script, the **AdvJDBCColumnHandler.jar** file will need to be added to the `--javaClasspath` argument. Also, if the **columnHandler.properties** configuration file will be used to override any default options the directory containing the **columnHandler.properties** needs to be added to the `--javaClasspath` argument as well. The following example shows this argument when running forge from the command line:

```
forge --javaClasspath
/path/to/AdvJDBCColumnHandler.jar:/path/to/JDBCDriver.jar:/path/to/propsf
ile_directory /path/to/Pipeline.epx
```

And in a control script:

```
forge : Forge
       working_machine  = $(foundry_machine)
       working_dir      = $(project_root)/data/forge_input
       pipeline         = $(project_root)/data/forge_input/Pipeline.epx
       forge_options    = --javaClasspath
                           $(project_root)/java/AdvJDBCColumnHandler.jar:
                           /path/to/JDBCDriver.jar:/path/to/propsfile_dir
                           ectory
```

If your project requires additional JAR files, such as those required by the JDBC driver, be sure to include those references as well. Also note that Unix systems use a colon to delimit multiple JARs, while Windows uses a semicolon. For more information on setting the classpath, see the section "Overriding Java home and class path settings" within the Endeca Developer Studio Help.

Output

File System Output

The Advanced JDBC Column handler optionally writes character data (CLOB, LONGVARCHAR types) to the filesystem, but by default these character values are returned inline as the Endeca property's value. If configured for output to the file system using the **charDataToDisk** option mentioned above, the files will be created in the **outputDataDir** directory (also configurable) and would have filenames of the form `clob_dataN.tmp`. *N*, in this case is a random number suffix to keep these temporary files distinct.

The binary column type handlers always write their data to the file system, in the **outputDataDir** directory. These file names are of the form `blob_dataN.tmp`. The relative path to each file is returned as the Endeca property's value. For example `../incoming/blob_data22607.tmp`. The pipeline must then read this file in a subsequent record manipulator.

Importing Character Data with IMPORT_PROP

If the **charDataToDisk** option is enabled, character data will be written to the filesystem. One typical way to use acquire the data in these files is to build a record manipulator that uses the IMPORT_PROP expression to read in the character data. An example of such a record manipulator is shown below:

```
<RECORD_MANIPULATOR FRC_PVAL_IDX="TRUE" NAME="BLOB Manip.">
<RECORD_SOURCE>Records In</RECORD_SOURCE>
<EXPRESSION LABEL="" NAME="IF" TYPE="VOID" URL="">

       <COMMENT>if a reference to a CLOB file exists...</COMMENT>
       <EXPRESSION LABEL="" NAME="PROP_EXISTS" TYPE="INTEGER" URL="">
              <EXPRNODE NAME="PROP_NAME" VALUE="CLOB_COL_NAME"/>
       </EXPRESSION>
```

```
      <EXPRESSION LABEL="" NAME="IMPORT_PROP" TYPE="VOID" URL="">
      <COMMENT>pull in the char data and remove the file</COMMENT>
            <EXPRNODE NAME="PROP_NAME" VALUE="CLOB_COL_NAME"/>
            <EXPRNODE NAME="REMOVE_FILES" VALUE="TRUE"/>
            <EXPRNODE NAME="ENCODING" VALUE="UTF-8"/>
      </EXPRESSION>

</EXPRESSION>
</RECORD_MANIPULATOR>
```

Processing Binary Data with the Document Converter

One typical usage scenario for binary column data is to read in documents like PDFs or Word files from the database. In this case, the Advanced JDBC Column Handler would write out this binary column data to the temporary files mentioned above. Then the pipeline would invoke the Document Converter to convert these binary-formatted files into plaintext Endeca properties indexed for search. The following example pipeline component could be used to do this conversion:

```
<RECORD_MANIPULATOR FRC_PVAL_IDX="TRUE" NAME="BLOB Manip.">
<RECORD_SOURCE>Records In</RECORD_SOURCE>
<EXPRESSION LABEL="" NAME="IF" TYPE="VOID" URL="">

      <COMMENT>if a reference to a BLOB file exists...</COMMENT>
      <EXPRESSION LABEL="" NAME="PROP_EXISTS" TYPE="INTEGER" URL="">
            <EXPRNODE NAME="PROP_NAME" VALUE="BLOB_COL_NAME"/>
      </EXPRESSION>

      <EXPRESSION LABEL="" NAME="RENAME" TYPE="VOID" URL="">
      <COMMENT>… rename the BLOB property,</COMMENT>
            <EXPRNODE NAME="OLD_NAME" VALUE="BLOB_COL_NAME"/>
            <EXPRNODE NAME="NEW_NAME" VALUE="Endeca.Document.Body"/>
      </EXPRESSION>

      <EXPRESSION LABEL="" NAME="CONVERTTOTEXT" TYPE="VOID" URL="">
      <COMMENT>extract the searchable text from the file,</COMMENT>
            <EXPRNODE NAME="RESPONSE_TIMEOUT" VALUE="300"/>
      </EXPRESSION>

      <EXPRESSION TYPE="VOID" NAME="REMOVE_EXPORTED_PROP">
      <COMMENT>and then remove the file from the filesystem.</COMMENT>
            <EXPRNODE NAME="PROP_NAME" VALUE="Endeca.Document.Body"/>
            <EXPRNODE NAME="REMOVE_PROPS" VALUE="TRUE"/>
      </EXPRESSION>

</EXPRESSION>
</RECORD_MANIPULATOR>
```

Note that the binary column's property name should be renamed to *Endeca.Document.Body*, since this is the property sought by the Document converter module. After this manipulator processes a record, it will create properties like *Endeca.Document.Text*, which contains the converted document text and *Endeca.Document.Encoding*, which reflects the binary file format detected. For more information on the Document converter module, see the VOID CONVERTTOTEXT section of the Data Foundry Expression Reference.

Troubleshooting

Logging Output

Logging output will be directed to the Forge log. Non-fatal warning messages may be seen here. For example, if a column handler encounters some sort of stream-reading error, the following log message would appear in the forge logfile:

```
WARN    09/22/06 14:00:49.958 UTC       FORGE      {forge,baseline}:
(com.endeca.soleng.itl.jdbc.AdvancedJDBCColumnHandler): Could not read
data from LONGVARCHAR column “text”: out of memory
```

The record returned for this row will have a null value for the property in question. Processing will continue with the next row in the SQL query result. The only fatal errors which stop forge from running will occur if an unsupported column type is encountered. Unsupported Java SQL column types include the following:

- ARRAY
- DISTINCT
- JAVA_OBJECT
- OTHER
- REF
- STRUCT

If any of these column types are returned in the SQL result, forge will produce an error message like the following:

```
ERROR   07/31/06 13:29:07.903 UTC FORGE {forge,baseline}:
(AdapterRunner): Unsupported Java SQL column type ARRAY
```

JDBC Driver
The Advanced JDBC Column Handler processes data retrieved by the JDBC database driver. If you encounter any problems, the first step is to ensure that the database driver is performing correctly. Testing the database driver outside of forge will verify this; a good first step is to test the driver using a standalone Java program.

Oracle drivers in particular can be troublesome. If you experience errors using the column handler with Oracle's JDBC drivers, you should check the following guidelines:

1. If you are using Oracle 9i or later, make sure that you are using the JDBC driver implementation that came with your Oracle server installation. If you have a patched revision of the Oracle server (e.g. 9.2.0.6), it is likely that the patch contains an updated JDBC driver. Check to make sure that you are using the patched JDBC driver.

2. If you are using earlier versions of Oracle 9i, try using the OCI driver instead of the thin driver. Later versions of the Oracle 9i thin driver have full support for CLOBs and BLOBs, but earlier Oracle 9i drivers do not.

3. If you are using Oracle 8i or earlier, please contact Oracle Endeca Customer Support.

4. If you still have issues, please contact Oracle Endeca Customer Support. Please provide them with your JDBC configuration settings (without passwords please!) and the version numbers for both the Oracle server and the JDBC driver that you are using.