that only a partial unmarshal is necessary to determine message format.

Format Indicator could also be used to handle multiple version of a message interface. The indicator would be set to indicate to which version of the messaging interface the message conforms and would then be used by a Content-based Router to send the message to the component that implements the correct version of the interface. This would enable new versions to run alongside old version while still using the same message infrastructure.

10.11 Data Streaming

As mentioned in section 10.8, “Message Sequence”, handling of very large messages in a messaging solution may require memory resources many times greater than the size of the largest message to be handled. Frequently the architect has no choice but to consume or produce a very large message, a file containing a batch or related transactions, for example, or a large and complex XML message generated by, or intended for, an external application. Handling such messages poses special challenges. Java CAPS can assist with Batch eWay support for data streaming when such messages are manifested as files in a file system. If it is possible to break large messages up into components and process components individually, or collect components and assemble them into a large message. eTL, another of the products in the Java CAPS Suite, can assist in processing large volumes of data. Whilst ETL (Extract, Transfer and Load) is typically associated with one off batch extraction and load of data, Java CAPS’ eTL can be used both standalone and in-stream as part of a larger Java CAPS solution. In this in-stream mode it will be discussed as a possible means of streaming data between a flat file and database table, between flat files or between database tables/views.

10.11.1 Batch eWay Streaming

Java Collaborations that use Batch eWays are commonly implemented to read the entire content of the file into the Payload node of the Batch eWay OTD then unmarshal it into some OTD that gives access to individual “records” or fields. If the file contains multiple records that will be processed individually downstream from the Batch eWay it would be far more efficient to read the file a record at a time and release records for further processing as soon as available. Batch Record eWay provides the ability to stream file data into a parser that breaks it into records. The parser works with delimited, fixed length and “whole file is a single record” records through Connectivity Map-based configuration properties.

Batch eWay Streaming Adapter can be used for streaming data between an FTP Server and a Local file, in either direction, without the need to use the Batch Record eWay. This will be useful if a Java CAPS solution needs to obtain a large file from an FTP Site for use locally or needs to send a local file to the remote FTP Site as efficiently as possible. Section 16.4.2, “Polling JMS Destination”, uses an example of local file to FTP Server streaming to illustrate polling JMS Destination as part of exception and retry handler.

The concept of data streaming will be illustrated with several examples, including the use of Batch Record, Java Buffered IO in conjunction with BatchLocalFile eWay’s StreamAdapters and eTL.

To stream a local file in, process its contents a piece at a time, and stream it back out again, one must use a BatchLocalFile and a BatchRecord for the inbound side as well as a BatchRecord and a BatchLocalFile for the outbound side. The example discussed below implements this functionality within a collaboration triggered by a BatchInbound eWay. Because files are streamed, this collaboration can process files of arbitrarily large size without grossly inflating JVM memory use.

Here is the bare-bones file-to-file streaming collaboration, jcdLocal2Local.

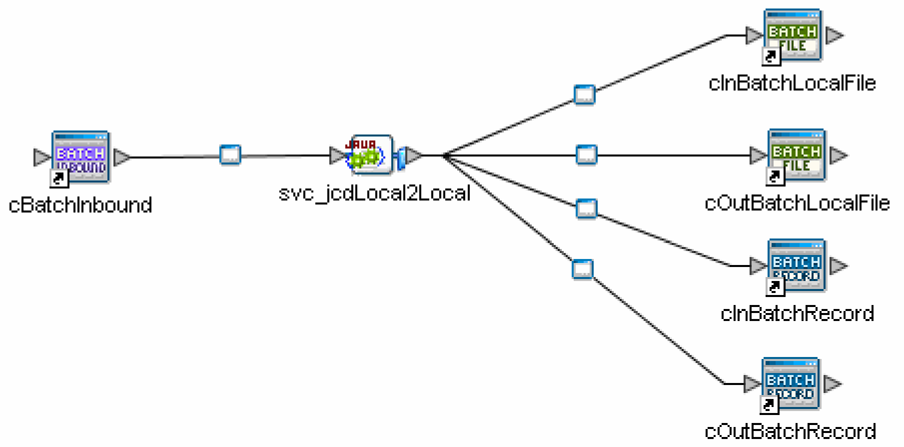
```
public void receive
    (com.stc.connector.batchadapter.appconn.BatchAppconnMessage input
    ,com.stc.eways.batchext.BatchLocal R_BatchLocalFile
    ,com.stc.eways.batchext.BatchLocal W_BatchLocalFile
    ,com.stc.eways.batchext.BatchRecord R_BatchRecord
    ,com.stc.eways.batchext.BatchRecord W_BatchRecord )
        throws Throwable
{
    // populate Batch Local File Client configuration based on the GUID name
    // assigned by the Batch Inbound
    ;
    R_BatchLocalFile.getConfiguration().setTargetFileName
       ( input.getGUIDFileName() );
    R_BatchLocalFile.getConfiguration().setTargetFileNameIsPattern
       ( false );
    R_BatchLocalFile.getConfiguration().setTargetDirectoryNameIsPattern
       ( false );
    R_BatchLocalFile.getConfiguration().setTargetDirectoryName
       ( input.getPathDirName() );
    logger.debug( "\n===>>> GUID File: " + input.getGUIDFileName() );
    ;
    // use streaming from inut through record to output
    ;
    R_BatchRecord.setInputStreamAdapter
            ( R_BatchLocalFile.getClient().getInputStreamAdapter() );
    W_BatchRecord.setOutputStreamAdapter
            ( W_BatchLocalFile.getClient().getOutputStreamAdapter() );
    ;
    byte[] baRecIn = null;
    int i = 0;
    while (R_BatchRecord.get()) {
        baRecIn = R_BatchRecord.getRecord();
        W_BatchRecord.setRecord( baRecIn );
        W_BatchRecord.put();
        i++;
        logger.debug( "\n===>>> Record [" + i + "]");
```

```
      }
      ;
}
```

__Book/MessageExchangePatterns/DataStreaming/Local2Local/jcdLocal2Local

The name and path of the file,, the BatchLocalFile eWay is to process, is provided by the BatchInbound eWay and is set in the collaboration. A StreamingAdapter is obtained for inbound and outbound files. The inbound file is broken into records in a loop and each record is written out to the outbound stream.

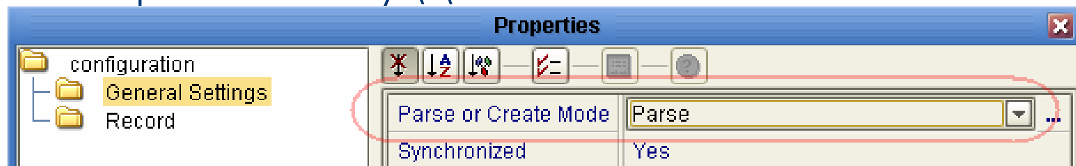
The Connectivity Map for this solution looks like this:



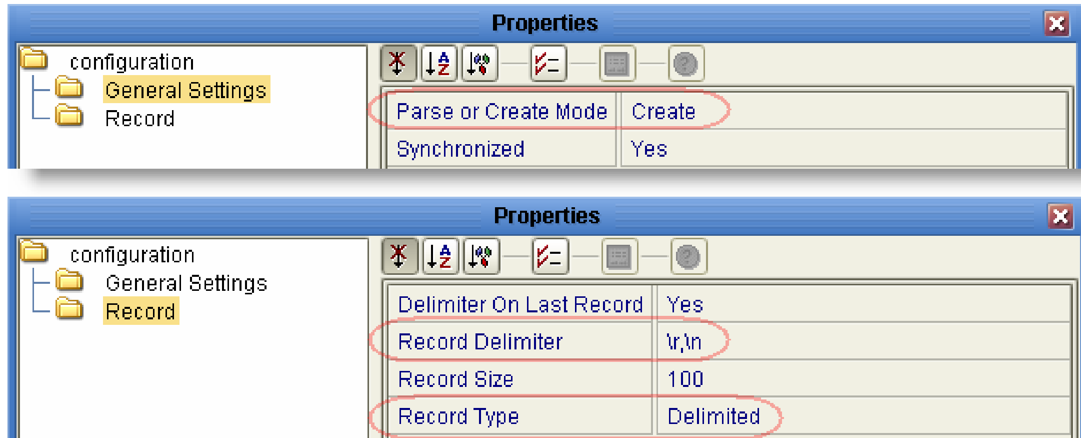
__Book/MessageExchangePatterns/DataStreaming/Local2Local/jcdLocal2Local

To assist parsing files into records and assembling files from records BatchRecord eWays are configured to use \r\n delimiter set. In this example both the input file and the output file are CarriageReturn/NewLine delimited files.

Parse input delimited by \r\n.



Create output delimited by \r\n



Create a Deployment Profile, build and deploy. Any Windows \r\n delimited text file will do as input. The Java Collaboration, as shown above, will write a tracer message with the record number to server.log for each record it processes.

> Note that for fixed record files all records, including the last record, must be the same size. This means that one cannot use the batch record with an arbitrary size record to emulate buffering unless it so happens that the last record is the same size as all other records. When it is not, the last read will fail and the bytes remining to be read will not be accessible and will be lost. Exception will be thrown when that happens.

The same technique can be used to process files containing fixed length records. Rather than specifying Record Type as Delimited we specify it as Fixed and furnish the appropriate value for the Record Size.

> Note that specifying Record Type of Single Record is no different from not using the BatchRecord at all but using Batch eWay, in one of its variants, to load the entire file into memory as a single record.

If it is absolutely necessary to break a file into "buffer-sized" chunks without regard for delimiters or the size of the file, where the file is not guaranteed to have an even length that is a multiple of desired buffer size, it is still possible to stream the file. One would have to use a BatchRecord eWay with a fixed length record of 1 byte on the input side and do one's own buffering.

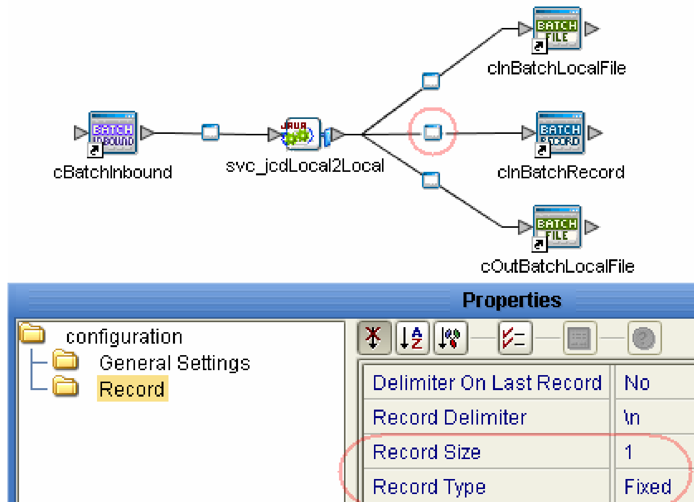
Let's use a collaboration that receives data through a 1-byte BatchRecord, assembles bytes in byte array buffer of some defined size, then writes out each buffer to the output file. The last buffer may be "short" in that there may be fewer than buffer-size bytes left. This must be handled differently from the regular buffer.

The buffering implementation is arbitrary and not necessarily the best. It is here merely as an example.

```
public void receive
    (com.stc.connector.batchadapter.appconn.BatchAppconnMessage input
    ,com.stc.eways.batchext.BatchLocal R_BatchLocalFile
    ,com.stc.eways.batchext.BatchLocal W_BatchLocalFile
    ,com.stc.eways.batchext.BatchRecord R_BatchRecord )
    throws Throwable
{
    // populate Batch Local File Client configuration based on the GUID name
    // assigned by the Batch Inbound
    ;
    R_BatchLocalFile.getConfiguration().setTargetFileName
        ( input.getGUIDFileName() );
    R_BatchLocalFile.getConfiguration().setTargetFileNameIsPattern
        ( false );
    R_BatchLocalFile.getConfiguration().setTargetDirectoryNameIsPattern
        ( false );
    R_BatchLocalFile.getConfiguration().setTargetDirectoryName
        ( input.getPathDirName() );
    logger.debug( "\n===>>> GUID File: " + input.getGUIDFileName() );
    ;
    // use streaming from input through record to output
    ;
    R_BatchRecord.setInputStreamAdapter
        ( R_BatchLocalFile.getClient().getInputStreamAdapter() );
    ;
    long lStart = System.currentTimeMillis();
    byte[] baRecIn = null;
    byte[] baBuffer = new byte[20480000];
    int i = 0;
    int j = 0;
    logger.debug( "\n===>>>After byte array allocation" );
    while (R_BatchRecord.get()) {
        baRecIn = R_BatchRecord.getRecord();
        baBuffer[i] = baRecIn[0];
        if (i == baBuffer.length - 1) {
            logger.debug( "\n===>>> in loop buffer condition with i="
                    + i + " and j=" + j );
            i = 0;
            W_BatchLocalFile.getClient().setPayload( baBuffer );
            W_BatchLocalFile.getClient().put();
            j++;
        } else {
            i++;
        }
    }
    ;
    // write out last short buffer
    ;
    if (i > 0) {
        byte[] baRest = new byte[--i];
        System.arraycopy( baBuffer, 0, baRest, 0, i );
        W_BatchLocalFile.getClient().setPayload( baRest );
        W_BatchLocalFile.getClient().put();
        logger.debug( "\n===>>> Record [" + j + "]" );
        ;
    }
    ;
    logger.debug( "\n===>>>Processed " + j + " buffers in "
    + (System.currentTimeMillis() - lStart) + " milliseconds" );
}
```

__Book/MessageExchangePatterns/DataStreaming/Local2LocalByteStream/jcdLocal2Local

The Connectivity Map for this implementation, and BatchRecord eWay configuration, are shown below.



To test the effectiveness of this approach a number of tests were conducted. The input text file, consisting of 1,050,508 records, delimited by New Line characters, was 387,459,465 bytes in size (387 MB). This file was processed by the collaboration using different buffer sizes. For each run the number of buffers and the execution duration in milliseconds were recorded. The test machine was a Dell Latitude D600 with 2 GB of memory and a 1.6 GHz Intel processor.

The table below summarises run statistics.

Input Buffer Size	Output Buffer Size	Buffers	Duration Milliseconds
1	2048	189189	14830565 (~4 hrs)
1	20480	18918	3248251 (~54 min)
1	204800	1891	1656812 (~28 min)
1	2048000	189	1563579 (~26 min)
1	20480000	18	1549458 (~26 min)

It is worth noting that with 2MB (2048000) and larger buffers the machine was CPU-bound.

In contrast, the solution using two BatchRecord eWays to stream delimited records, described previously, using the same file as input, took 547287 milliseconds (~9 min) to process the file. The system was CPU-bound through the entire process.

A third solution, using Java Buffered IO in conjunction with BatchLocalFile eWay's StreamAdapters and a 2 MB byte array buffer, processed the same file in 68890 milliseconds – that's just over 1 minute! Further more, there was no issue with the file having to be a multiple of the buffer size in length.

The collaboration used to achieve this performance is shown below. Note that BatchLocalFile's StreamAdapter can be asked to provide an InputStream or OutputStream, which can then be used as any other Java IO stream. Please also note that Stream so obtained must subsequently be released.

```
public void receive
    (com.stc.connector.batchadapter.appconn.BatchAppconnMessage input
    ,com.stc.eways.batchext.BatchLocal R_BatchLocalFile
    ,com.stc.eways.batchext.BatchLocal W_BatchLocalFile )
        throws Throwable
{
    // populate Batch Local File Client configuration based on the GUID name
    // assigned by the Batch Inbound
    ;
    R_BatchLocalFile.getConfiguration().setTargetFileName
        ( input.getGUIDFileName() );
    R_BatchLocalFile.getConfiguration().setTargetFileNameIsPattern
        ( false );
    R_BatchLocalFile.getConfiguration().setTargetDirectoryNameIsPattern
        ( false );
    R_BatchLocalFile.getConfiguration().setTargetDirectoryName
        ( input.getPathDirName() );
    logger.debug( "\n===>>> GUID File: " + input.getGUIDFileName() );
    ;
    // use streaming from inut through record to output
    ;
    long lStart = System.currentTimeMillis();
    int iReadCnt = 0;
    byte[] baBuffer = new byte[2048000];
    ;
    com.stc.eways.common.eway.standalone.streaming.InputStreamAdapter isa
        = R_BatchLocalFile.getClient().getInputStreamAdapter();
    java.io.BufferedInputStream bis =
        new java.io.BufferedInputStream( isa.requestInputStream() );
    ;
    com.stc.eways.common.eway.standalone.streaming.OutputStreamAdapter osa
        = W_BatchLocalFile.getClient().getOutputStreamAdapter();
    java.io.BufferedOutputStream bos =
        new java.io.BufferedOutputStream( osa.requestOutputStream() );
    ;
    int i = 0;
    iReadCnt = bis.read( baBuffer );
    while (iReadCnt > 0) {
        bos.write( baBuffer, 0, iReadCnt );
        bos.flush();
        i++;
        if (i % 10000 == 0) {
            logger.debug( "\n===>>> Buffer [" + i
            + "] size " + baBuffer.length );
        }
        iReadCnt = bis.read( baBuffer );
    }
    ;
    isa.releaseInputStream( true );
    osa.releaseOutputStream( true );
    ;
    logger.debug( "\n===>>>Processed " + i + " buffers in "
        + (System.currentTimeMillis() - lStart) + " milliseconds" );
    ;
}
```

__Book/MessageExchangePatterns/DataStreaming/Local2LocalJavaIOBuffered/jcdLocal2Local

With Batch eWay there are good, not so good and downright bad ways of processing large volumes of data as has been demonstrated. Which of the potential solutions for data streaming is

best will very likely depend on the individual requirements. This section presented a number of Batch eWay-based data streaming solutions that can be used and tailored as appropriate.

10.11.2 eTL Streaming

eTL is the component of Java CAPS intended to be used for data Extraction, Transfer and Load. Typically used for bulk standalone extraction, conversion and load of data, Sun SeeBeyond eTL tool can be also used inside an eInsight Business Process as part of a larger Java CAPS Solution.

In this section we will build a simple solution that streams a content of a file to a database table using the eTL tool. First we will construct and test the eTL Collaboration, then we will use this collaboration in an eInsight Business Process to demonstrate how an eTL process can be incorporated into a messaging solution. Finally, we will construct and exercise an ‘equivalent’ non-eTL solution, using the Batch eWay in streaming mode to read the file and the Oracle eWay to populate the table. This will give an opportunity to compare the effort required to develop and implement each solution.

The source file, a Comma Separated Values (CSV) file, csvAUDIT_TRAIL.csv, contains 475,250 rows of data.

The destination table, defined by the following DDL, will receive data from the CSV file.

```
CREATE TABLE "UI"."DOC_AUDIT_TRAIL"
   (  "DOC_APPLICATION_CODE" VARCHAR2(22) NOT NULL ENABLE,
      "DOC_ID" VARCHAR2(30) NOT NULL ENABLE,
      "CREATE_DATETIME" DATE NOT NULL ENABLE,
      "U_ID" VARCHAR2(15) NOT NULL ENABLE,
      "USER_ID" VARCHAR2(15) NOT NULL ENABLE,
      "ACTION_TYPE" VARCHAR2(1) NOT NULL ENABLE
   )
```

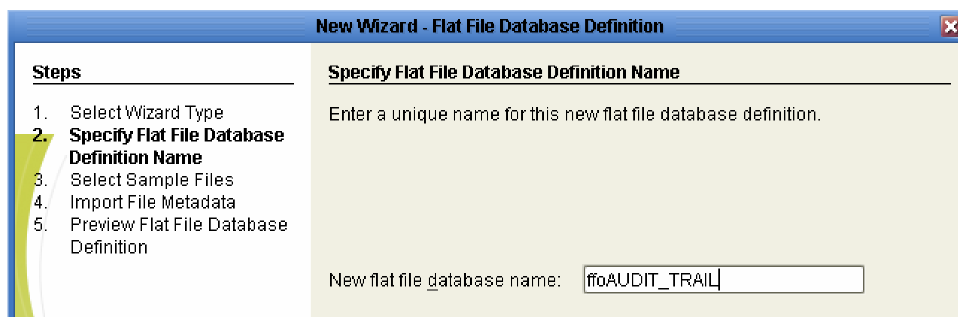
We will create the table using the DDL definition then use the Oracle OTD Wizard to create the new Oracle Database OTD, tblAUDIT_TRAIL.

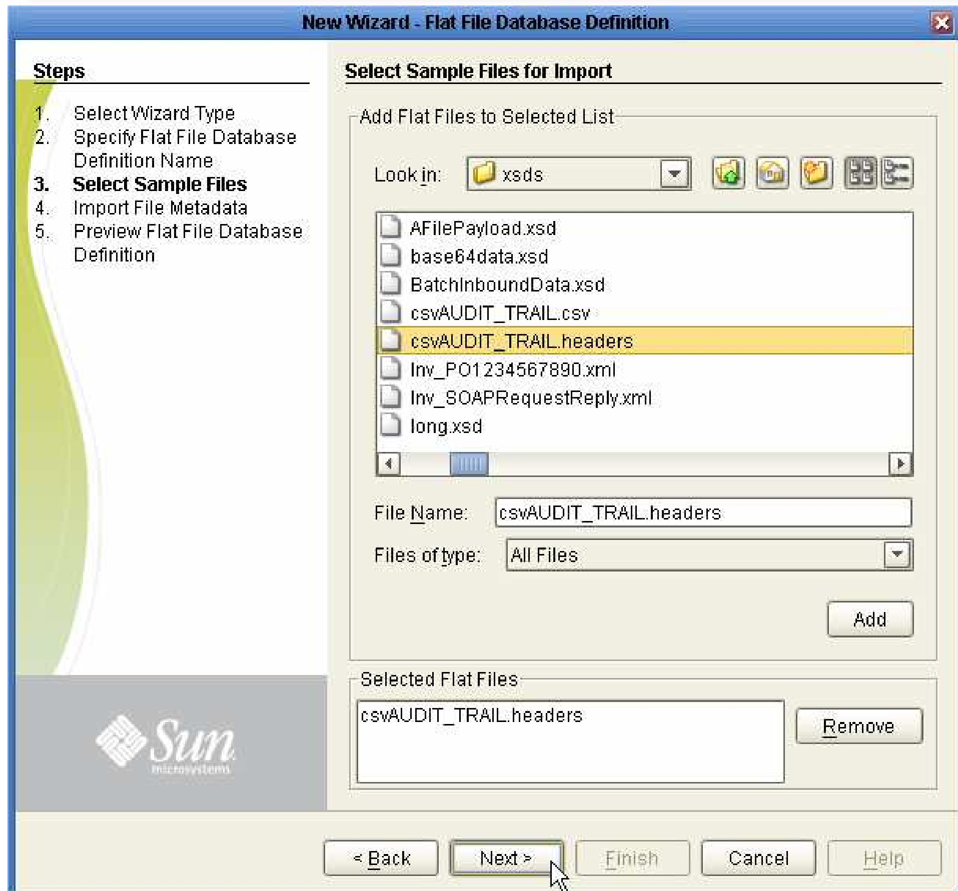
The table definition follows the format of the comma delimited file. Names of the columns correspond to the names of the fields to be read from the file, csvAUDIT_TRAIL.headers. This file contains the list of column names separated by comma characters.

To allow OTD construction and testing before deployment let’s create a file containing the following.

```
DOC_APPLICATION_CODE,DOC_ID,CREATE_DATETIME,U_ID,USER_ID,ACTION_TYPE
FILEBER,107154,2001-05-29T09:17:15,1000251598,GEORGET,V
MEDLAB,6363742200001,2001-05-29T10:31:13,1000047783,GEORGET,V
FILEBER,107188,2001-05-29T10:50:16,1000251598,GEORGET,V
ROMER,31900001,2001-06-15T11:26:46,1000251639,auto,A
FILEBER,106577,2001-06-15T11:32:40,1000192160,GEORGET,V
ROMER,31700001,2001-06-15T11:33:54,1000251639,auto,A
MEDLAB,6753434400002,2001-06-27T08:37:51,1000024167,auto,A
ROMER,33800001,2001-06-27T08:46:42,1000251631,auto,A
ANVISIT,20010627700001,2001-06-27T09:58:14,1000251571,auto,A
ANVISIT,20010627700001,2001-06-27T09:58:45,1000251571,NEGUSJ,V
```

The new OTD's name will be ffoAUDIT_TRAIL. We will follow the steps the wizard suggests. In this example data types and sizes will match these in the database table.





New Wizard - Flat File Database Definition

Steps

1. Select Wizard Type
2. Specify Flat File Database Definition Name
3. Select Sample Files
4. **Import File Metadata (File 1 of 1)**
5. Preview Flat File Database Definition

Import File Metadata for CSVAUDIT_TRAIL_HEADERS (Step 1 of 3)

Define the formatting type and encoding for this file.

Table name: CSVAUDIT_TRAIL

Encoding scheme: ASCII (ISO646-US)

File format: (•) Delimited ( ) FixedWidth

New Wizard - Flat File Database Definition

Steps

1. Select Wizard Type
2. Specify Flat File Database Definition Name
3. Select Sample Files
4. **Import File Metadata (File 1 of 1)**
5. Preview Flat File Database Definition

Import File Metadata for CSVAUDIT_TRAIL (Step 2 of 3)

Supply the following information required to parse this file.

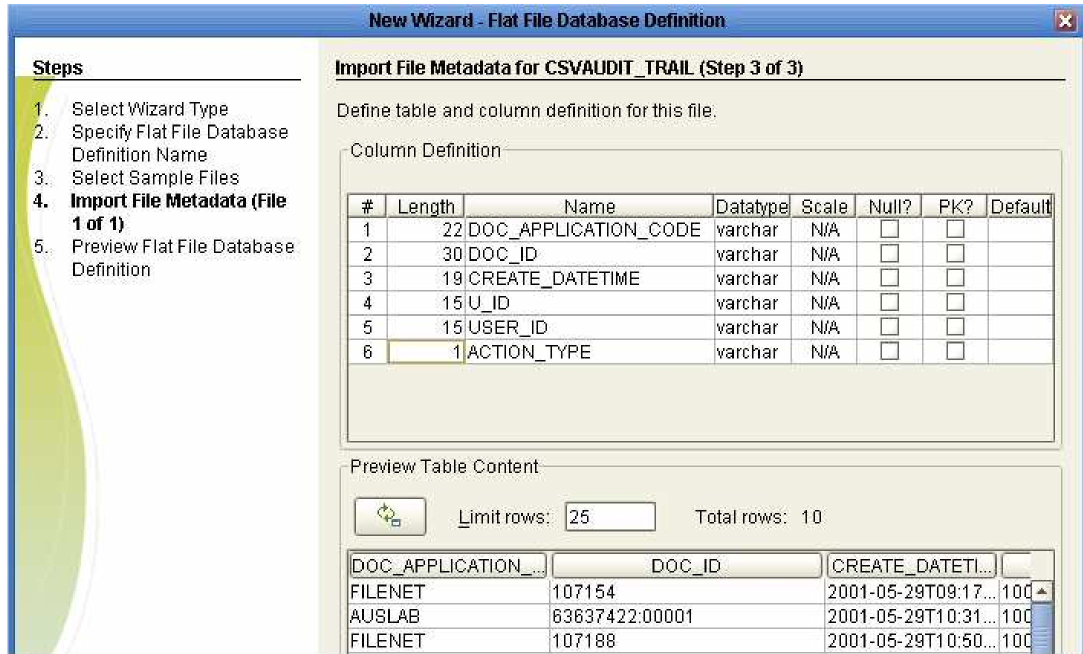
Property	Value
Default data type	varchar
Default precision	20
Record delimiter	{CR}{LF} or {LF}
Field delimiter	{comma}
User defined field delimiter	
Text qualifier	{double quote: "}
First line contains field names?	True
Rows to skip	0
Maximum # of faults to tolerate	0

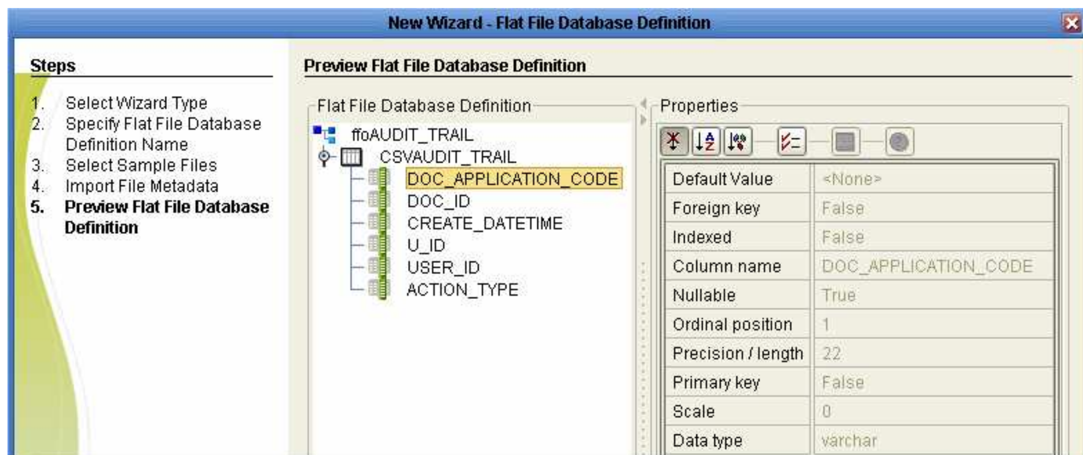
Delimited

Preview of file

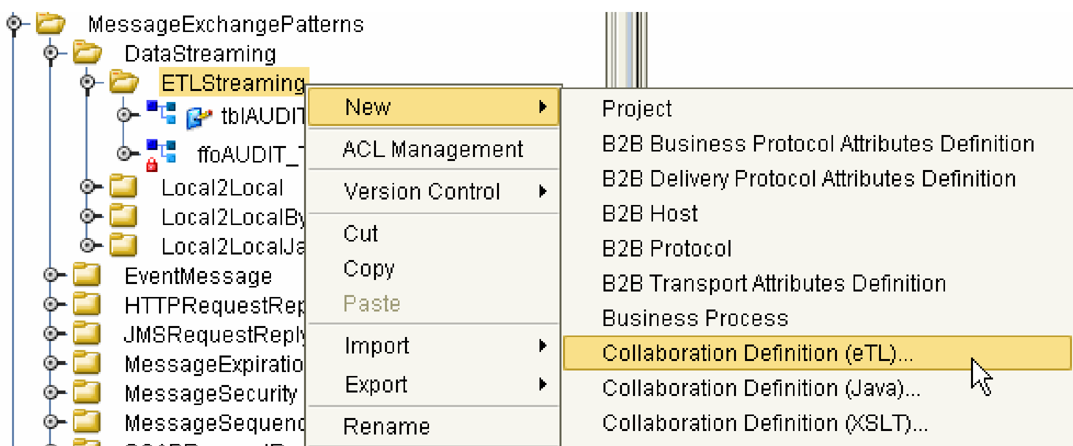
```
DOC_APPLICATION_CODE,DOC_ID,CREATE_DATETIME,U_ID,
FILENET,107154,2001-05-29T09:17:15,1000251598,GE
AUSLAB,63637422:00001,2001-05-29T10:31:13,100004
FILENET,107188,2001-05-29T10:50:16,1000251598,GE
HOMER,319:00001,2001-06-15T11:26:46,1000251639,a
```

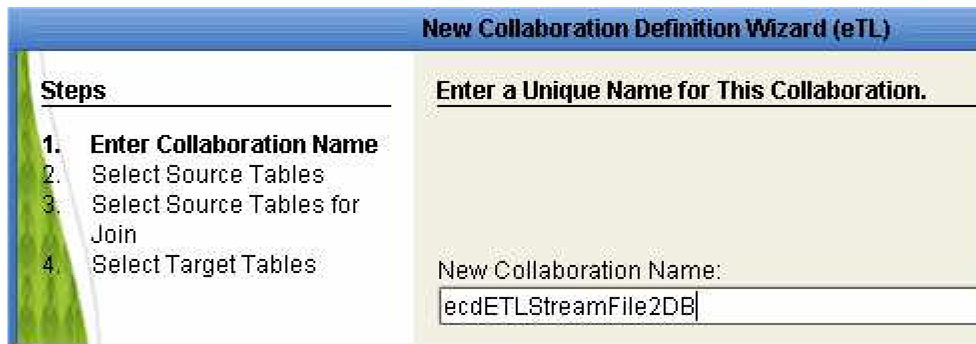
Column CREATE_DATETIME contains date/time string in the format YYYY-MM-DDTHH:MI:SS so we will allow 19 characters to contain it.



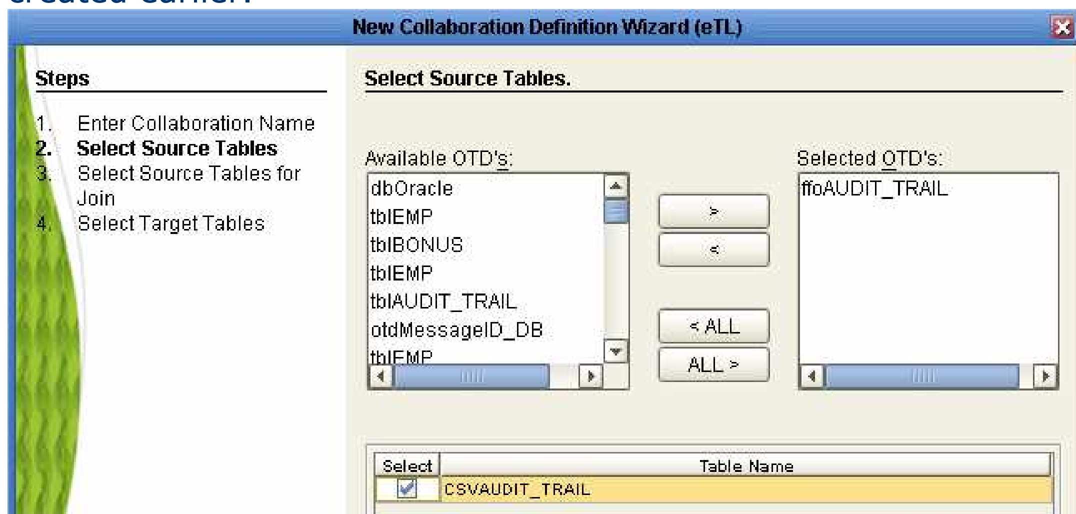


Once the input, ffoAUDIT_TRAIL, and the output, tblAUDIT_TRAIL, OTDs are defined we can proceed to develop the eTL Collaboration Definition, ecdETLStreamFile2DB.

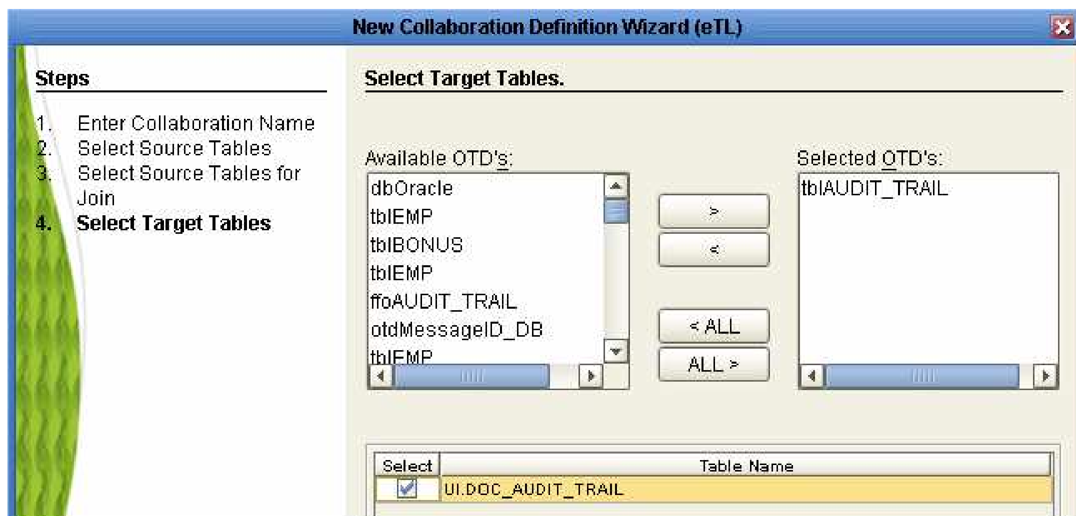




Data will originate in the flat file whose OTD, ffoAUDIT_TRAIL, was created earlier.

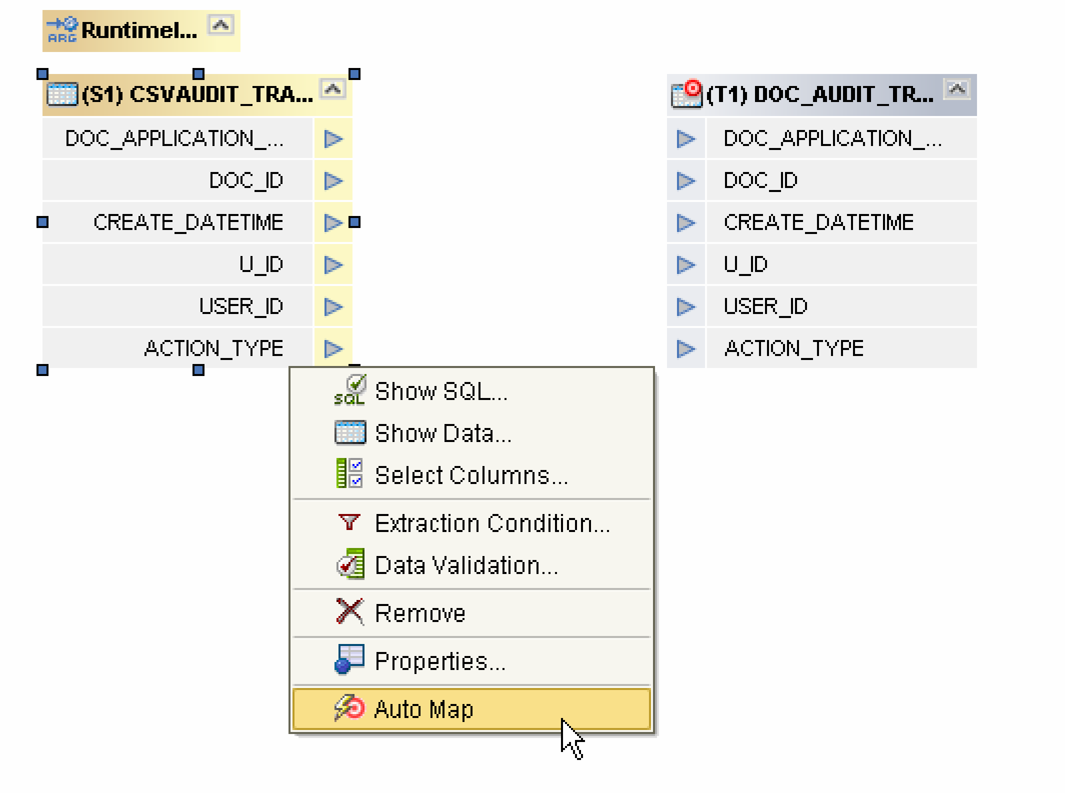


Once transformed, data will be loaded into a database table, tblAUDIT_TRAIL, developed earlier.

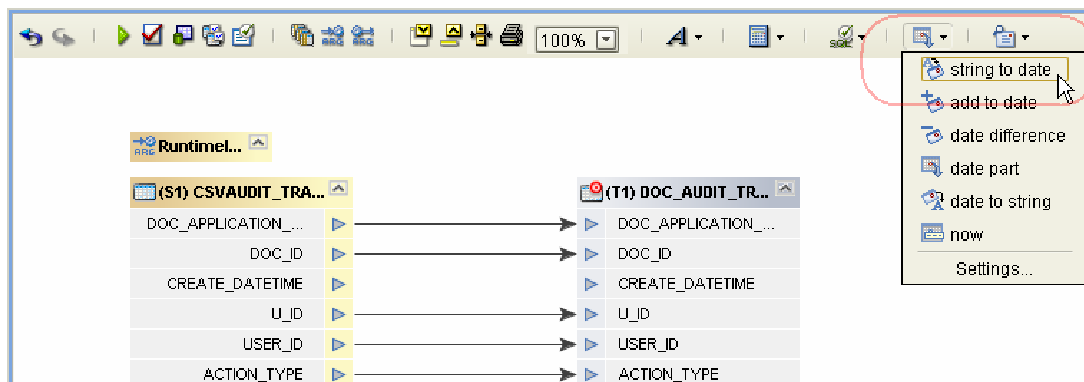


In this example the mapping is perfectly straight forward. All fields in the input structure, with the exception of create_datetime, are mapped directly to the corresponding fields in the output structure.

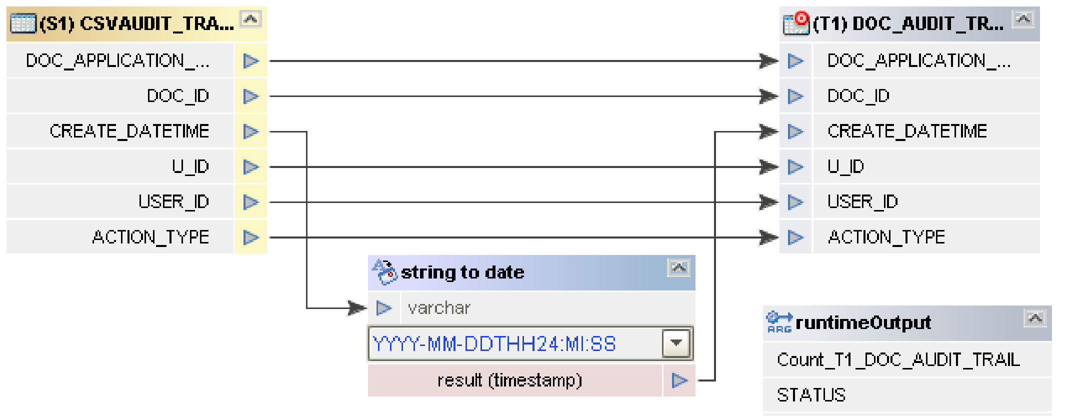
Let's right-click on the source table graphic and choose "automap" operation.



All but the CREATE_DATETIME fields will be mapped. To map CREATE_DATETIME we need to convert the date/time string, in the format YYYY-MM-DDTHH:MI:SS into a timestamp. A convenience function, string to date, available from the date operations drop down, will be used.

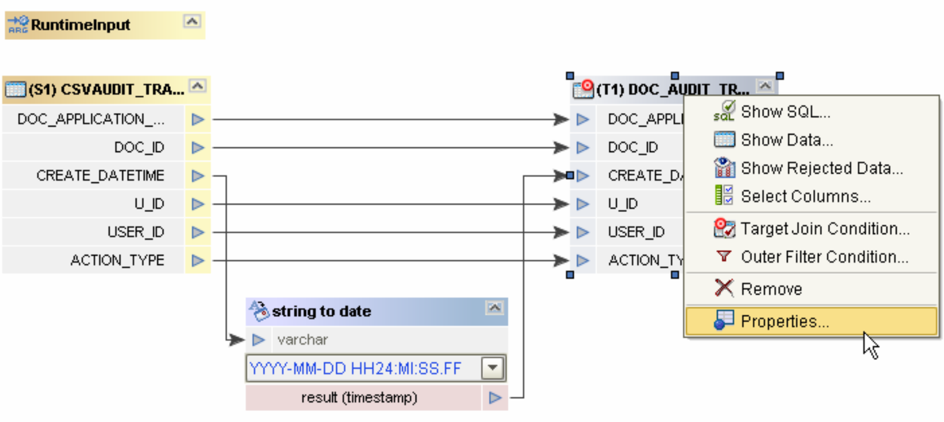


"string to date" function has a number of formats that can be used. We shall choose the appropriate one and complete the mapping.

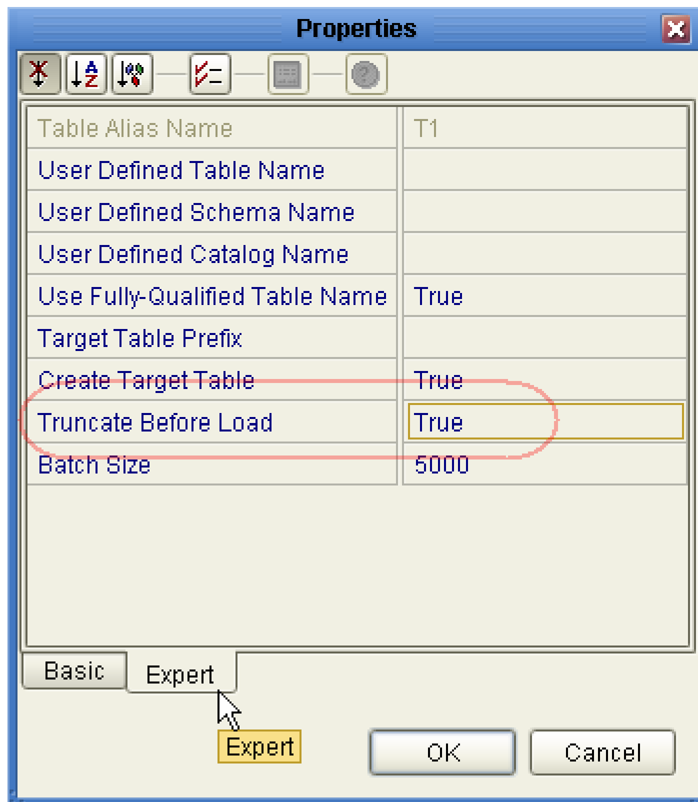


Joins, conversions, functions and other transformations can be applied to the data as it is being transferred, if required.

We wish to clear the target table before load. Right click on the target table and choose Properties.

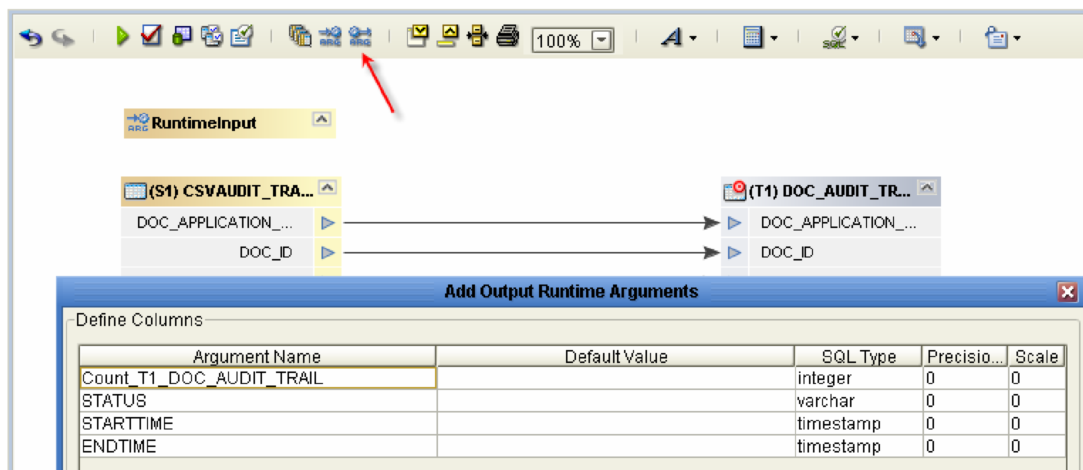


Switch to Expert mode and choose “true” for the value of the property “Truncate Before Load”.

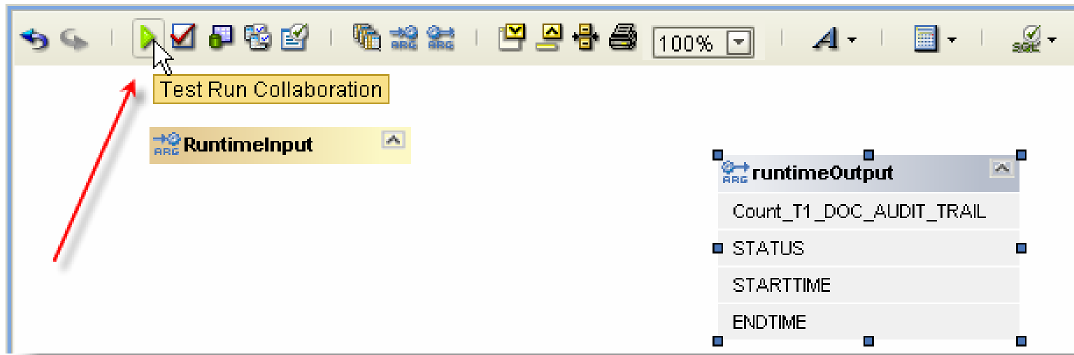


Other properties can be configured as required.

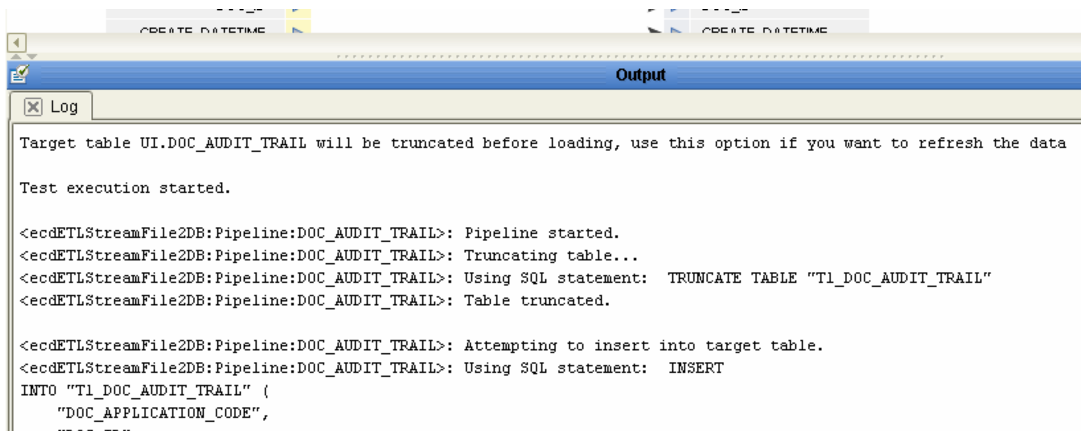
We are interested in determining how many rows were processed, whether processing succeeded and how long it took. We shall add Output Runtime Arguments to the eTL Collaboration so that the eInsight Business Process, which will host this collaboration, can obtain access to these values.



The collaboration, unsophisticated as it is, is now ready. Let's test it with a subset of data.



The collaboration editor window will be divided into two panes with the lower pane containing log of the test.



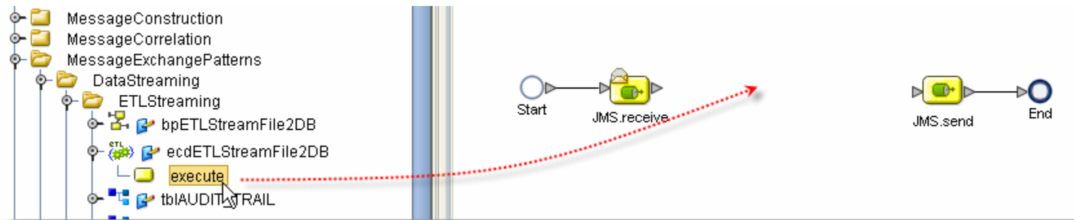
Right clicking on the target table graphic and choosing “Show Data” will display the target table test data load results.

Clearly, a transformation much more complex than the one shown in this example can be constructed. It could use multiple source and target tables, variety of data cleansing and manipulation operations, insert, upsert and delete database operations and more. In this section we are dealing with the basic data streaming use of the eTL. eTL will stream data from a file, or a database, to a database much more efficiently than just about any other method available within Java CAPS.

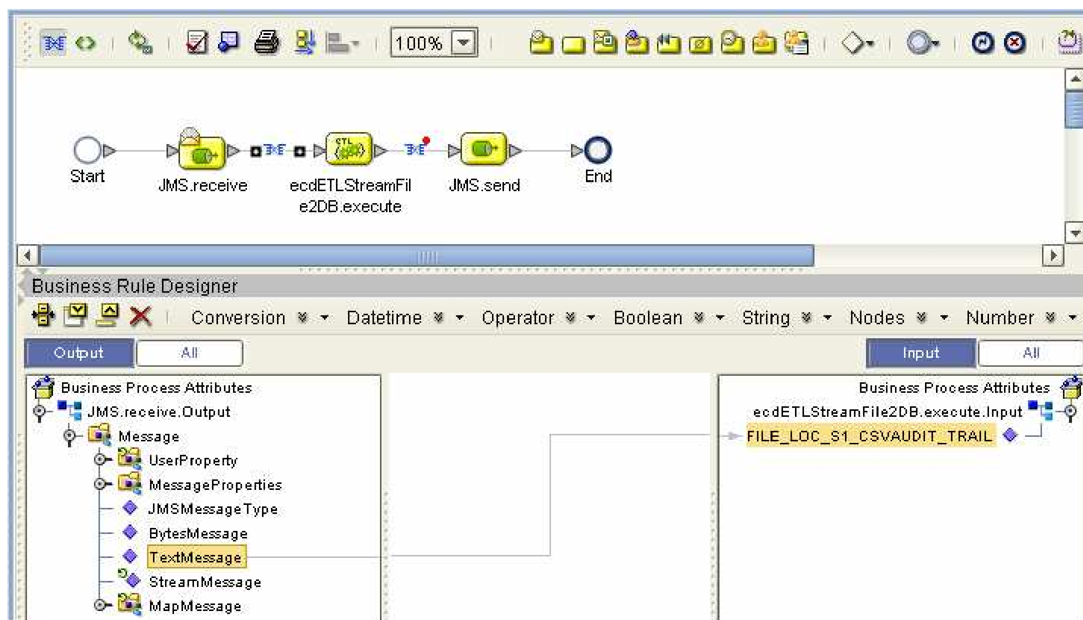
Let’s construct the Business Process that will be triggered by a JMS message, will invoke the eTL Collaboration constructed before, and will set a JMS message containing the results of the operation to a downstream component. Whilst we will not show this here the downstream component could use this information to trigger processing records loaded by the eTL Collaboration.
The Business Process will be called bpETLStreamiFile2DB. The JMS message that will trigger the process will contain the path to the file the eTL Collaboration will process. The results will be concatenated

into a pipe-delimited ('|') message and sent out to the outbound JMS Queue.

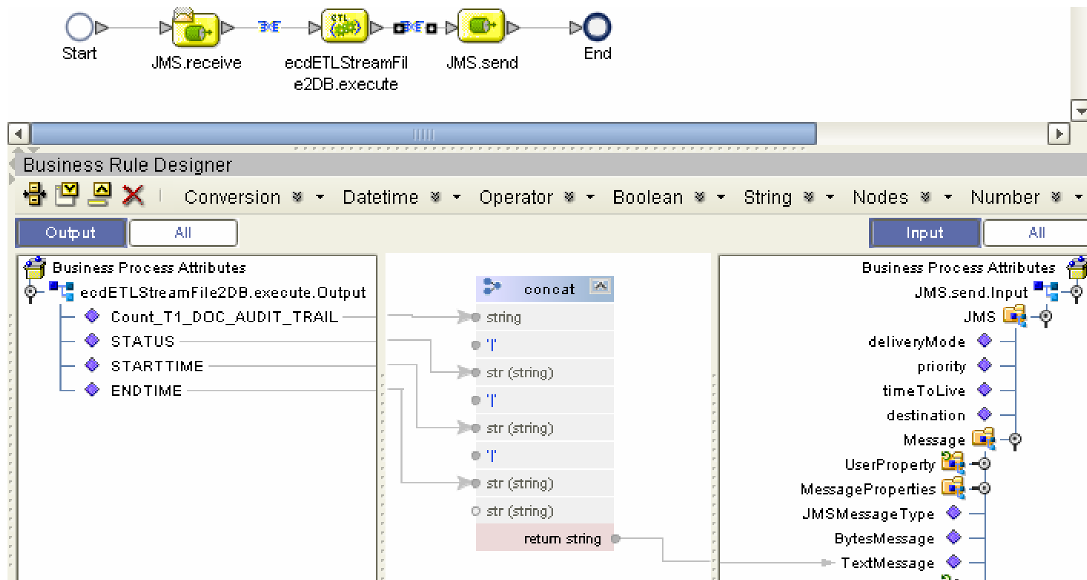
Let's create the process, drag JMS receive and send activities, and drag the 'execute' service of the eTL Collaboration onto the canvas.



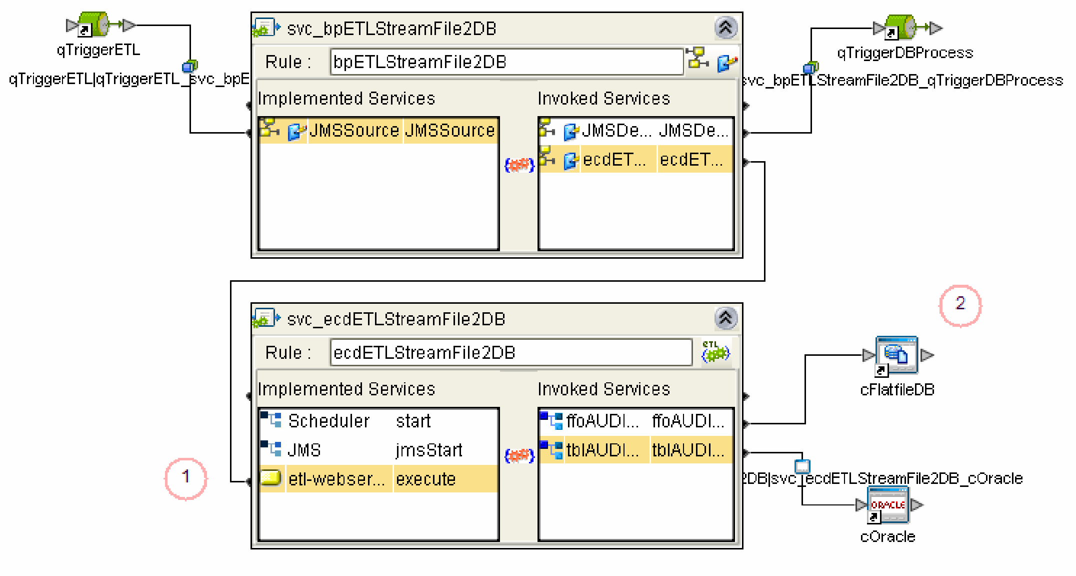
Business Rules mapping between the JMS Receive Activity will provide the eTL Collaboration with the file path.



Business Rules mapping between the eTL Collaboration and the JMS Send Activity will construct the outbound message.



Let's construct the Connectivity Map, cmETLStreamFile2DB.



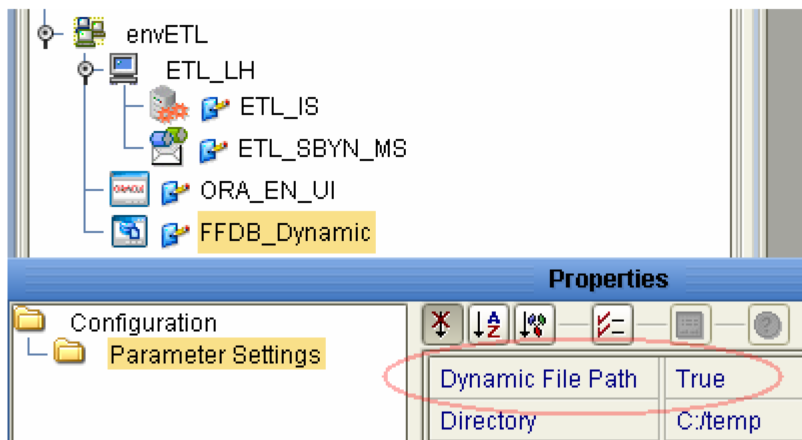
Of the three possible inputs to the eTL Collaboration (1) we choose the "execute" service. This is because we wish to invoke this collaboration as part of a business process. We could have triggered it by a timer or by a JMS message directly.

Since we are loading a file using eTL we must use a special "Flat File DB" connector (2) rather than the regular Batch eWay.

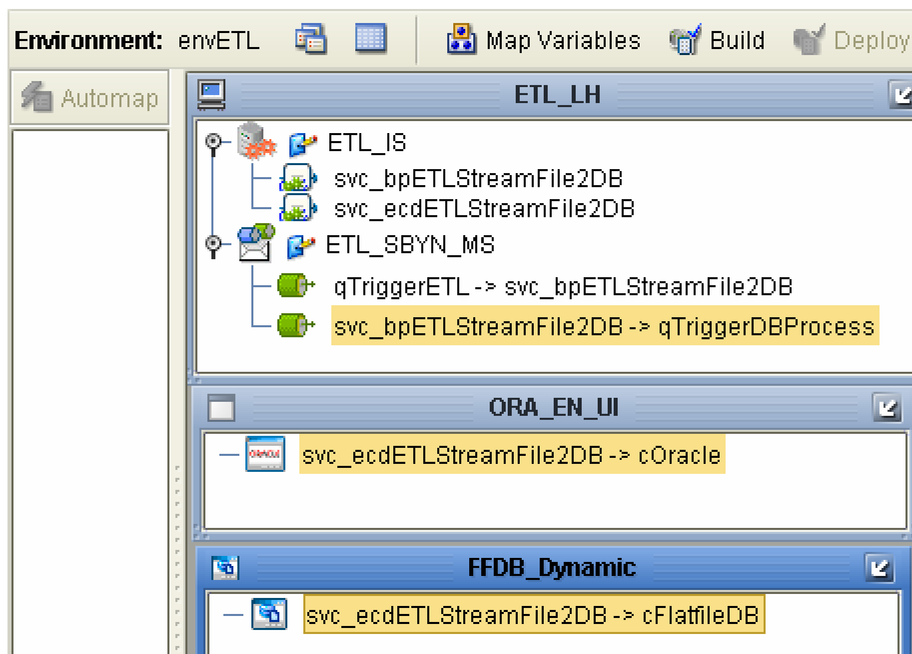
To avoid clattering our existing environments and to show just the essential containers, let's create a brand new environment, envETL. Components mapped to containers in this environment will be deployed to the same Application Server Domain as components

mapped to containers in the bkEnv, which we have been using in most of the examples.

The environment will have the Logical Host with one Integration Server and one JMS Message Server, one Oracle External and one Flat File Database external system. The configuration of all externals, except the Flat File Database, will be much the same as usual. The Flat File Database will be configured to support dynamic file path so that the source file can be set at runtime. Furthermore, because we are using eTL, the Integration Server "Application Workspace Directory" property must be set to point to a directory that eTL can use for its work files.



Lt's now create a new Deployment Profile, ETLStreamFile2DB, which will allow us to map Connectivity Map components to the envETL environment.



Submitting the JMS text message with the name of the flat file containing our 474,803 records initiates the ETL process. On the book development platform the process takes around 9 minutes. The results JMS message shows record count, status, and start and stop times.

```
474803|Success|2007-02-04T07:03:13.71Z|2007-02-04T07:12:06.67Z
```

The Enterprise Manager can be used to inspect runtime information pertaining to the eTL Collaboration and, if the appropriate inbound connector, like the Scheduler, is used, the collaboration can be stopped and started through the Enterprise Manager.

	Totals	Average
Extracted:	0	0
Loaded:	0	0
Rejected:	0	0

EXECUTIONID	TARGETTABLE	STARTDATE	ENDDATE	EXTRACTED	LOADED	REJECTED	EXCEPTION_MSG
1	T1_DOC_AUDIT_TRAIL	2007-02-05 21:15:19.581				NULL	

The figure above shows the eTL Collaboration as it is executing. Once completed, runtime statistics and counters become available for the eTL run.

	Totals	Average
Extracted:	474803	474803
Loaded:	474803	474803
Rejected:	0	0

EXECUTIONID	TARGETTABLE	STARTDATE	ENDDATE	EXTRACTED	LOADED	REJECTED	EXCEPTION_MS
1	T1_DOC_AUDIT_TRAIL	2007-02-05 21:15:19.581	2007-02-05 21:23:16.276	474803	474803	0	

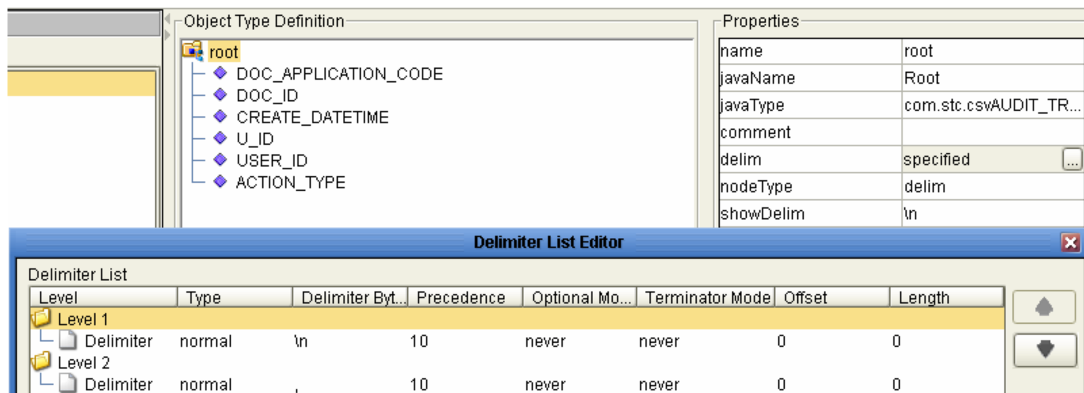
To contrast the eTL-based solution with a "regular" solution let's develop a Java Collaboration that streams the content of the same file to the same database table using the Batch Local File eWay, the Batch Record eWay and the Oracle eWay. As in the eTL example above the Java Collaboration will be triggered by a JMS message containing the path to the file and will submit a JMS message containing the count of records it inserted, the start time and the

end time. Batch Record streaming mode will be used to break the file into records and insert records into the database table.

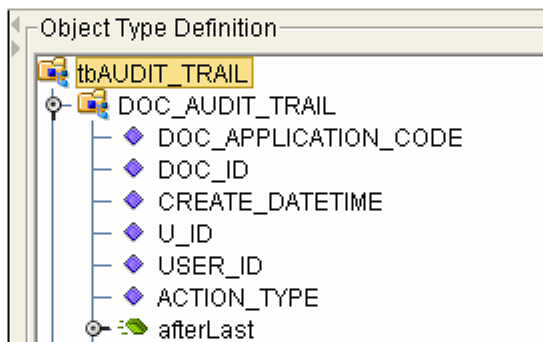
Let's create a file, csvAUDIT_TRAIL.ffd, with the following contents.

```
DOC_APPLICATION_CODE
DOC_ID
CREATE_DATETIME
U_ID
USER_ID
ACTION_TYPE
```

Let's now create a new OTD using the "UD OTD from file" wizard. Once the OTD is created let's add two levels of delimiters, one with the new line delimiter "\n" and one with the comma delimiter.



The Oracle table OTD will be the same as in the previous example.

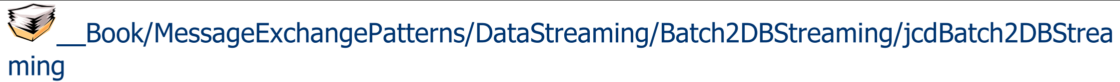


The Java Collaboration, functionally-equivalent to the eTL Collaboration discussed previously, is shown below. This collaboration uses the Batch Local File eWay and the Batch Record eWay's streaming adapter to read the input file a record at a time. It parses each record and inserts it into the database table. The

collaboration needs an Oracle Table OTD for the target table and a User-defined delimited OTD for the input file.

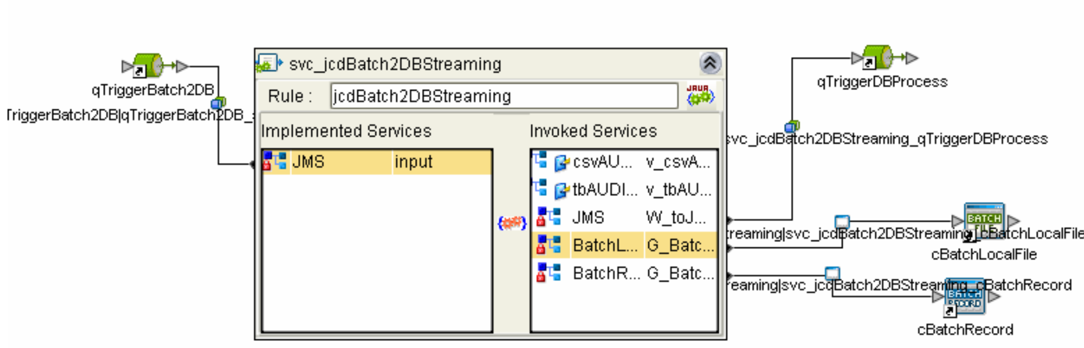
```
public void receive
    (com.stc.connectors.jms.Message input
    ,com.stc.csvAUDIT_TRAIL1967709979.Root v_csvAUDIT_TRAIL
    ,tbAUDIT_TRAIL.TbAUDIT_TRAILOTD v_tbAUDIT_TRAIL
    ,com.stc.connectors.jms.JMS W_toJMS
    ,com.stc.eways.batchext.BatchLocal G_BatchLocalFile
    ,com.stc.eways.batchext.BatchRecord G_BatchRecord )
        throws Throwable
{
    G_BatchLocalFile.getConfiguration().setTargetFileName
        ( input.getTextMessage() );
    G_BatchLocalFile.getConfiguration().setTargetFileNameIsPattern
        ( false );
    ;
    // use streaming from input through record to output
    ;
    G_BatchRecord.setInputStreamAdapter
        ( G_BatchLocalFile.getClient().getInputStreamAdapter() );
    ;
    int i = 0;
    byte[] baRecIn = null;
    long lStart = System.currentTimeMillis();
    ;
    v_tbAUDIT_TRAIL.getDOC_AUDIT_TRAIL().insert();
    logger.debug( "\n===>>> initied for insert" );
    ;
    try {
        ;
        while (G_BatchRecord.get()) {
            baRecIn = G_BatchRecord.getRecord();
            if (i > 0) {
                v_csvAUDIT_TRAIL.unmarshalFromBytes( baRecIn );
                ;
                v_tbAUDIT_TRAIL.getDOC_AUDIT_TRAIL().setDOC_APPLICATION_CODE
                    ( v_csvAUDIT_TRAIL.getDocApplicationCode() );
                v_tbAUDIT_TRAIL.getDOC_AUDIT_TRAIL().setDOC_ID
                    ( v_csvAUDIT_TRAIL.getDocId() );
                v_tbAUDIT_TRAIL.getDOC_AUDIT_TRAIL().setCREATE_DATETIME
                    ( typeConverter.stringToTimestamp
                        ( v_csvAUDIT_TRAIL.getCreateDatetime().replaceAll
                            ( "T", " " )
                           , "yyyy-MM-dd hh:mm:ss", false, "" ) );
                v_tbAUDIT_TRAIL.getDOC_AUDIT_TRAIL().setU_ID
                    ( v_csvAUDIT_TRAIL.getUId() );
                v_tbAUDIT_TRAIL.getDOC_AUDIT_TRAIL().setUSER_ID
                    ( v_csvAUDIT_TRAIL.getUserId() );
                v_tbAUDIT_TRAIL.getDOC_AUDIT_TRAIL().setACTION_TYPE
                    ( v_csvAUDIT_TRAIL.getActionType() );
                ;
                v_tbAUDIT_TRAIL.getDOC_AUDIT_TRAIL().insertRow();
            } else {
                logger.debug( "\n===>>> Skipping first record" );
            }
            if (i % 10000 == 0) {
                logger.debug( "\n" + i );
            }
            i++;
        }
        W_toJMS.sendText( "" + --i + "|"
            + (System.currentTimeMillis() - lStart) + " milliseconds" );
        logger.debug
            ( "\n===>>> done " + --i + " in "
            + (System.currentTimeMillis() - lStart)
            + " milliseconds" );
```

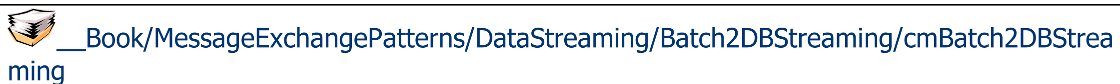
```
    } catch ( Exception e ) {
        logger.error( "Exception in the collab after processing "
              + i + " records", e );
    }
}
```



__Book/MessageExchangePatterns/DataStreaming/Batch2DBStreaming/jcdBatch2DBStreaming

The Connectivity Map, cmBatch2DBStreaming, looks like this.





__Book/MessageExchangePatterns/DataStreaming/Batch2DBStreaming/cmBatch2DBStreaming

Let’s create the deployment profile, dpBatch2DBStreaming, build and deploy. Before exercising the solution we must make sure the database table is cleared of any data that might have been inserted during the previous exercise. Unlike the eTL Collaboration, where “truncate before use” property was set, we must clear the database table explicitly.

```
SQL> truncate table  ui.doc_audit_trail ;
```

Let’s now exercise the solution by submitting a JMS text message containing the name of the file we used in the previous example.

Once the process completes the database table contains all records from the input file.

The eTL Collaboration developed in this section is a very simple collaboration. It uses very little of the built-in eTL facilities for joining tables, transformation, user-defined functions, built-in test facility and others that make eTL much more suitable for bulk data transformation and load than other means. Because of the simplicity of the requirement the functionally-equivalent Java Collaboration was constructed with ease and was quite simple. If the eTL process was more complex, involving multiple source and target tables, joins and complex transformations, the functionally-

equivalent Java Collaboration or eInsight Business Process would have been much more complex to develop and much less efficient to execute.

10.12 Message Security

In some circumstances it may be necessary to ensure confidentiality and integrity of messages that travel through the messaging system. One or both of two methods are typically used to protect messages in transit.

The more common and easier to relate to, because of its ubiquity, is the method that secures the point-to-point channel over which messages travel. Channel Security is typically provided by adding the Secure Sockets Layer (SSL) to the TCP/IP Protocol-based channels. Secure Sockets Layer [SSL] standard specifies how encryption can be applied to all bytes travelling through a point-to-point channel. It also specifies how End Points can authenticate each other, exchange cryptographic material, choose encryption algorithms and negotiate protocol version. In essence, and this is trivialising the matter considerably, to complete establishment of a Secure Session the two End Points perform a Cryptographic Handshake, during which capabilities, cryptographic material and credentials are negotiated and exchanged, before any payload data is sent. If the SSL Handshake fails, no session is established and the End Points do not exchange data. If the SSL Handshake succeeds, the endpoints cooperate, for the reminder of the session, in encrypting on send, and decrypting on receive, the byte stream that represents the payload data. Java CAPS provides the SSL-based Channel Security capability in the HTTP Client and the HTTP Server eWay, and in the JMS Message Server implementation. See 21.4, “Secure Sockets Layer (SSL, TLS)”, for a comprehensive discussion on SSL configuration in Java CAPS for solutions using HTTP eWay and Web Services End Points.

The less common is the method that individually secures each message. Snce encryption and digital signature attributes are applied to the message itself a secured message can traverse multiple channels and multiple components whilst preserving security. The Secure Messaging Extension eWay, available in ICAN 5.0.5, is expected to re-appear mid 2007 in conjunction with the release of the Sun B2B Suite 5.1, a successor to the eXchange Integrator of the 4.x and 5.0. It is expected to support both the Secure Multipurpose Mail Extensions (S/MIME) for encryption and digitial signing, and the XML Digital Signatures and the XML Encryption functionality. The SME eWay will provide easy means of securing individual messages.