



JasperETL

powered by Talend

User's Guide

Version 2.1.1

Adapted for **JasperETL** *powered by Talend* version 2.1.1 and later.

Copyright

Find a copy of the GNU Free Documentation License with the source files of this documentation.

User's Guide for JasperETL powered by Talend

Getting started with **JasperETL 9**

Accessing JasperETL	9
Creating a project	12
Describing the GUI	14
Repository	14
Business Models	15
Job Designs	15
Code	15
Routines	15
Snippets	16
Documentation	16
Metadata	16
Recycle bin	16
Graphical workspace	17
Palette	17
Changing the palette position	17
Changing the palette layout and settings	18
Properties, Run and Logs views	18
Properties	18
Logs	18
Run Job	18
Modules and Scheduler	18
Modules view	18
Open Scheduler	19
Outline and Code Summary panel	20
Outline	20
Code viewer	21
Toolbar and Menus	21
Quick access toolbar	21
Menus	22
Configuring JasperETL preferences	22
Perl/Java Interpreter path	23
Status	24
External components	25
Designing a Business Model	27
Objectives	27
Opening or creating a business model	27
Opening a business model	28
Creating a business model	28
Modeling a business model	29

Shapes	30
Connecting shapes	31
Commenting and arranging a model	33
Adding a note or free text	33
Arranging the model view	33
Properties	34
Rulers and Grid	34
Appearance	35
Assignment	36
Assigning repository elements to a Business Model	36
Editing a Business model	37
Renaming a business model	38
Copying and pasting a business model	38
Moving a business model	38
Deleting a business model	38
Saving a business model	38
 Designing a Job Design	 39
Objectives	39
Opening or Creating a job	39
Opening a job	40
Creating a job	40
Getting started with a Job Design	41
Showing, hiding and moving the palette	41
Click and drop elements	42
Adding Notes to a job design	42
Changing panels position	43
Warnings and errors on component	44
Connecting components together	45
Connection types	45
Row connection	45
Main row	45
Lookup row	46
Output row	46
Iterate connection	47
Trigger connections	47
Link connection	48
Multiple Input/Output job	48
Defining job Properties	49
Main	49
View	50
Documentation	50
Properties	51
Setting a built-in schema	51
Setting a repository schema	52

Defining the Start component	53
Defining Metadata items	54
Setting up a DB schema	55
Step 1: general properties	55
Step 2: connection	56
Step 3: table upload	58
Step 4: schema definition	58
Setting up a File Delimited schema	59
Step 1: general properties	59
Step 2: file upload	59
Step 3: schema definition	60
Step 4: final schema	63
Setting up a File Positional schema	64
Step 1: general properties	65
Step 2: connection and file upload	65
Step 3: schema refining	66
Step 4: final schema	66
Setting up a File Regex schema	67
Step 1: general properties	67
Step 2: file upload	67
Step 3: schema definition	68
Step 4: final schema	68
Setting up a FileLDIF schema	68
Step 1: general properties	69
Step 2: file upload	69
Step 3: schema definition	70
Step 4: final schema	71
Setting up a FileXML schema	71
Step 1: general properties	72
Step 2: file upload	72
Step 3: schema definition	72
Step 4: final schema	75
Creating queries using SQLBuilder	75
Database structure comparison	77
Building a query	77
Storing a query in the Repository	79
Mapping data flows in a job	80
tMap operation overview	80
tMap interface	81
Setting the input flow in the Mapper	83
Filling in Input tables with a schema	83
Main and Lookup table content	83
Variables	84
Explicit Join	84
Unique Match (java)	85

First or Last Match (java)	86
All Matches (java)	86
Inner join	86
All rows (java)	87
Filtering an input flow (java)	87
Removing Input entries from table	87
Mapping variables	88
Accessing global or context variables	89
Removing variables	89
Output setting	89
Filters	90
Rejections	91
Inner Join Rejection	91
Removing Output entries	91
Expression editor	91
Schema editor	92
Activating/Disabling a job or sub-job	93
Disabling a Start component	94
Disabling a non-Start component	94
Defining a job design context and related variables	94
Adding or renaming a context	95
Defining the context variables	96
Short creation of context variables	97
StoreSQLQuery	98
Storing contexts in the Repository	98
Defining the parameters in the Context tab	100
Running a job in selected context	101
Running a job	101
Running in normal mode	102
Displaying Statistics	102
Displaying Traces	103
Running in debug mode	103
Saving or exporting your jobs	104
Saving a job	104
Exporting job scripts	104
Generating HTML documentation	106
Automating stats & logs use	106
Shortcuts and aliases	107
Components	115
tAggregateRow	117
tAggregateRow properties	117
Scenario: Aggregating values and sorting data	118
tContextLoad	122
tContextLoad properties	122

Scenario: Dynamic context use in MySQL DB insert	123
tCreateTable	126
tCreateTable Properties	126
Scenario: Creating new table in a Mysql Database	127
tAddCRCRow	129
tAddCRCRow properties	129
Scenario: Adding a surrogate key to a file	129
tDB2Input	133
tDB2Input properties	133
Related scenarios	133
tDB2Output	134
tDB2Output properties	134
Related scenarios	134
tDB2Row	135
tDB2Row properties	135
Related scenarios	135
tDBInput	136
tDBInput properties	136
Scenario 1: Displaying selected data from DB table	137
Scenario 2: Using StoreSQLQuery variable	138
tDBOutput	140
DBOutput properties	140
Scenario: Displaying DB output	142
tDBSQLRow	144
tDBSQLRow properties	144
Scenario 1: Resetting a DB auto-increment	145
tDenormalize	147
Scenario 1: Denormalizing on one column in Perl	147
Scenario 2: Denormalizing on multiple columns in Java	149
tDie	152
tDie properties	152
Related scenarios	152
tDTDValidator	153
tDTDValidator Properties	153
Scenario: Validating xml files	153
tELTMysqlInput	156
tELTMysqlInput properties	156
Related scenarios	156
tELTMysqlMap	157
tELTMysqlMap properties	158
Connecting ELT components	158
Mapping and joining tables	159
Adding where clauses	159
Generating the SQL statement	159
Scenario1: Aggregating table columns and filtering	160

Scenario 2: ELT using Alias table	163
tELTMysqlOutput	167
tELTMysqlOutput properties	168
Related scenarios	169
tELTOraacleInput	170
tELTOraacleInput properties	170
Related scenarios	170
tELTOraacleMap	171
tELTOraacleMap properties	172
Connecting ELT components	172
Mapping and joining tables	173
Adding where clauses	173
Generating the SQL statement	173
Scenario 1: Updating Oracle DB entries	173
tELTOraacleOutput	176
tELTOraacleOutput properties	177
Related scenarios	178
tFileCompare	179
Scenario: Comparing unzipped files	179
tFileCopy	182
tFileCopy Properties	182
Scenario: Restoring files from bin	182
tFileDelete	184
tFileDelete Properties	184
Scenario: Deleting files	184
tFileFetch	186
tFileFetch properties	186
Scenario: Fetching data through HTTP	186
tFileInputDelimited	188
tFileInputDelimited properties	188
Scenario: Delimited file content display	189
tFileInputMail	191
tFileInputMail properties	191
Scenario: Extracting key fields from email	191
tFileInputPositional	193
tFileInputPositional properties	193
Scenario: From Positional to XML file	194
tFileInputRegex	197
tFileInputRegex properties	197
Scenario: Regex to Positional file	198
tFileInputXML	201
tFileInputXML Properties	201
Scenario: XML street finder	202
tFileList	204
tFileList properties	204

Scenario: Iterating on a file directory	204
tFileOutputExcel	207
Related scenario	207
tFileOutputLDIF	208
tFileOutputLDIF Properties	208
Scenario: Writing DB data into an LDIF-type file	209
tFileOutputXML	211
tFileOutputXML properties	211
Scenario: From Positional to XML file	212
tFileUnarchive	213
Related scenario	213
tFlowMeter	214
tFlowMeter Properties	214
Related scenario	214
tFor	215
tFor Properties	215
Scenario: Job execution in a loop	215
tFTP	218
tFTP properties	218
tFTP put	218
tFTP get	219
tFTP rename	219
tFTP delete	219
Scenario: Putting files on a remote FTP server	219
tFuzzyMatch	221
tFuzzyMatch properties	221
Scenario 1: Levenshtein distance of 0 in first names	222
Scenario 2: Levenshtein distance of 1 or 2 in first names	224
Scenario 3: Metaphonic distance in first name	225
tLogCatcher	226
tLogCatcher properties	226
Scenario1: warning & log on entries	226
Scenario 2: log & kill a job	228
tLogRow	230
tLogRow properties	230
Scenario: Delimited file content display	230
tMap	231
Scenario 1: Mapping with filter and simple explicit join (Perl)	231
Scenario 2: Mapping with Inner join rejection (Perl)	236
Scenario 3: Cascading join mapping	241
Scenario 4: Advanced mapping with filters, explicit joins and Inner join rejection (Java)	
242	
Scenario 5: Advanced mapping with filters and a check of all rows	247
tMSSqlInput	251
tMSSqlInput properties	251

Related scenarios	251
tMSSqlOutput	252
tMSSqlOutput properties	252
Related scenarios	252
tMSSqlRow	253
tMSSqlRow properties	253
Related scenarios	253
tMySqlConnection	254
tMySqlConnection Properties	254
Scenario: Inserting data in mother/daughter tables	254
tMySqlCommit	259
tMySqlCommit Properties	259
Related scenario	259
tMySqlInput	260
tMySqlInput properties	260
Related scenarios	260
tMySqlOutput	261
tMySqlOutput properties	261
Scenario: Adding new column and altering data	261
tMySqlOutputBulk	263
tMySqlOutputBulk properties	264
Scenario: Inserting transformed data in MySQL database	265
tMySqlBulkExec	269
tMySqlBulkExec properties	270
Related scenarios	271
tMySqlOutputBulkExec	272
tMySqlOutputBulkExec properties	272
Scenario: Inserting data in MySQL database	273
tMySqlRollback	274
tMySqlRollback properties	274
Scenario: Rollback from inserting data in mother/daughter tables	274
tMySqlRow	275
tMySqlRow properties	275
Scenario: Removing and regenerating a MySQL table index	275
tMsgBox	277
tMsgBox properties	277
Scenario: ‘Hello world!’ type test	277
tNormalize	279
Scenario: Normalizing data	279
tOracleInput	282
tOracleInput properties	282
Related scenarios	282
tOracleBulkExec	283
tOracleBulkExec properties	283
Scenario: Truncating and inserting file data into Oracle DB	284

tOracleOutput	287
tOracleOutput properties	287
Related scenarios	287
tOracleRow	288
tOracleRow properties	288
Related scenarios	288
tPerl	289
tPerl properties	289
Scenario: Displaying number of processed lines	289
tPostgresqlInput	292
tPostgresqlInput properties	292
Related scenarios	292
tPostgresqlOutput	293
tPostgresqlOutput properties	293
Related scenarios	293
tPostgresqlRow	294
tPostgresqlRow properties	294
Related scenarios	294
tRowGenerator	295
tRowGenerator properties	295
Defining the schema	296
Defining the function	297
Scenario: Generating random java data	297
tRunJob	300
tRunJob Properties	300
Scenario: Executing a remote job	300
tSalesforceInput	304
Related scenario	304
tSalesforceOutput	305
tSalesforceOutput Properties	305
Related scenario	305
tSendMail	306
tSendMail properties	306
Scenario: Email on error	306
tSleep	309
tSleep Properties	309
Related scenarios	309
tSortRow	310
tSortRow properties	310
Scenario: Sorting entries	311
tSQLiteInput	313
tSQLiteInput Properties	313
Scenario: Filtering SQLite data	313
tSQLiteOutput	316
tSQLiteOutput Properties	316

Related Scenario	317
tSQLiteRow	318
tSQLiteRow Properties	318
Scenario: Updating SQLite rows	319
tStatCatcher	321
tStatCatcher properties	321
Scenario: Displaying job stats log	321
tSugarCRMInput	324
tSugarCRMInput Properties	324
Scenario: Extracting account data from SugarCRM	324
tSugarCRMOutput	326
Related Scenario	326
tSybaseInput	327
tSybaseInput properties	327
Related scenarios	327
tSybaseOutput	328
tSybaseOutput properties	328
Related scenarios	328
tSybaseBulkExec	329
tSybaseBulkExec properties	329
Related scenarios	329
tSybaseRow	331
tSybaseRow properties	331
Related scenarios	331
tSystem	332
tSystem properties	332
Scenario: Echo 'Hello World!'	332
tUniqRow	334
tUniqRow properties	334
Scenario: Unduplicating entries	334
tWarn	336
tWarn properties	336
Related scenarios	336
tWebServiceInput	337
Scenario: Extracting images through a Webservice	338
tXSDValidator	340
tDTDValidator Properties	340
Related scenario	340
tXSLT	341
tXSLT	341
Scenario: Transforming XML to html using an XSL stylesheet	341
Managing projects	343
Importing projects	343
Importing Job samples (Demos)	344

Migration tasks	345
Importing Repository items	346
Exporting projects	346

—Getting started with JasperETL—

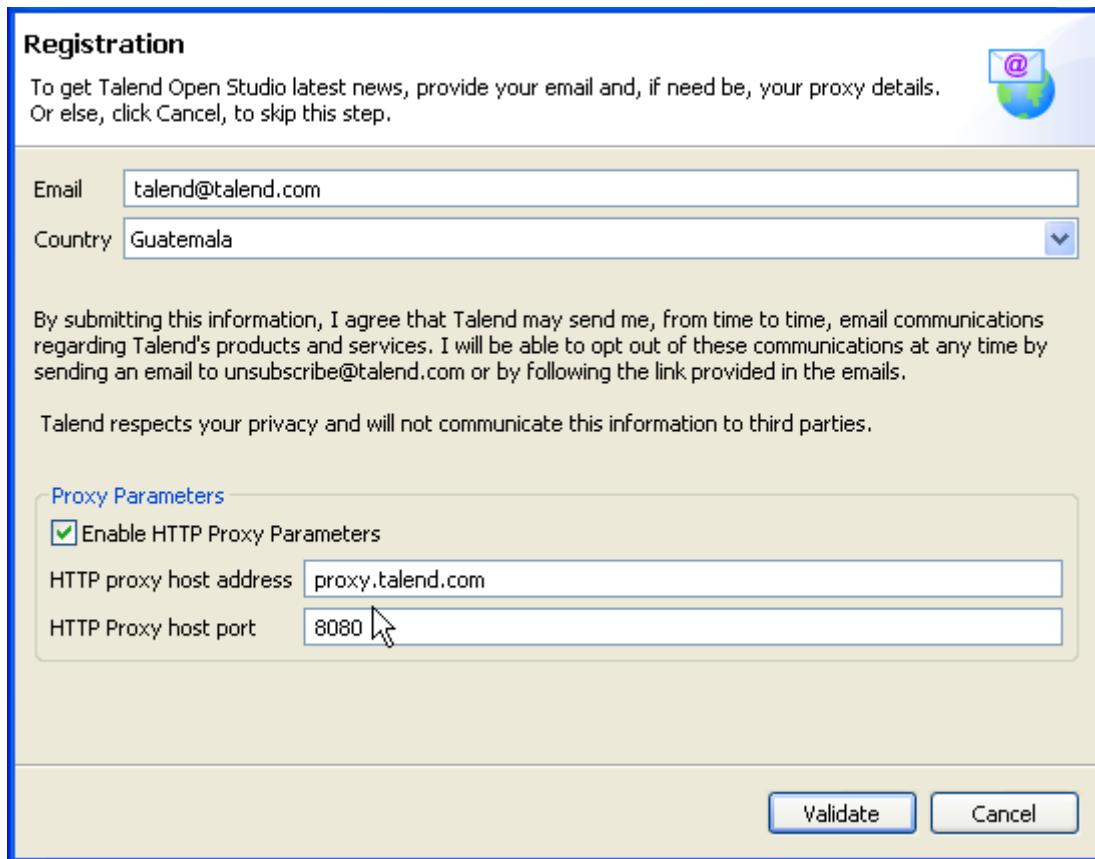
Getting started with JasperETL

Accessing JasperETL

The Setup wizard helps you to install **JasperETL** application. If you unzip it manually, then follow the installation instructions provided on the wiki website.

Read and accept the terms of the license agreement to continue.

A **JasperETL** Registration window prompts you for your email address and location. This information is optional. Click **Cancel**, if you do not wish to be informed for future enhancements of **JasperETL**.

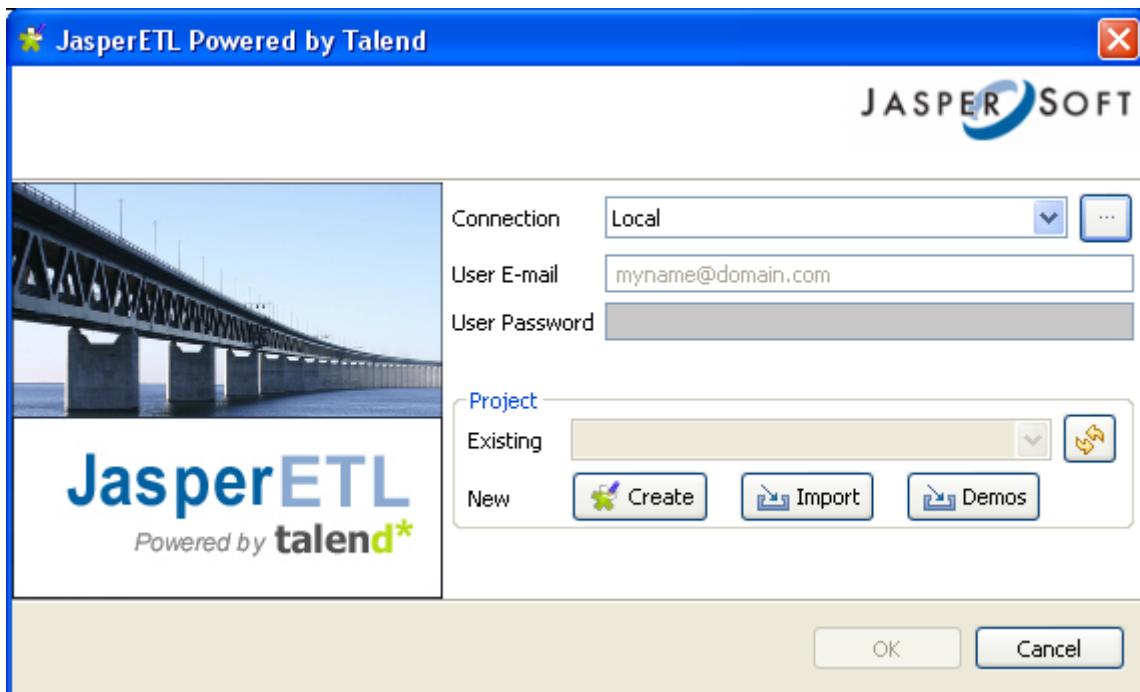


If needed, check the box to **enable HTTP Proxy parameters** and fill in the fields with your proxy details. Make sure you filled in the email address field if you provide proxy details.

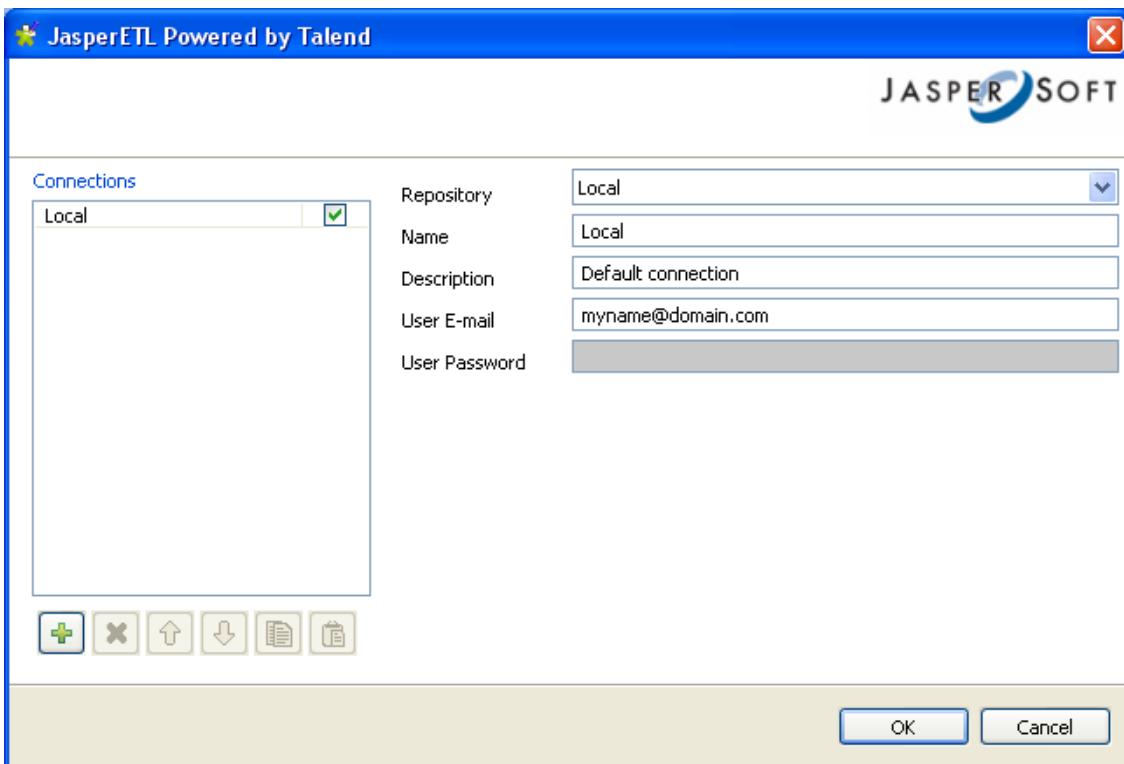
You can fill in or edit the registration information at any time, through *Window > Preference > Talend > Install/Update*.

WARNING—Be ensured that any personal information you may provide will never be transmitted to third parties nor used for another purpose than to inform you about Talend and Talend's products.

JasperETL opens up with the Login window.



- Select the Connection information if you set your username and connection details already.
- When logging in for the first time, click the three-dot button to configure the Connection information, either Local or Remote. Only Local is available for now.



- To add a new Repository information, click the plus (+) button on the left panel
- Type in the email address that will be used as **user login**. This field is compulsory to be able to use **JasperETL**. Be aware that the email entered is never used for other purpose than login use.
- Fill in the **Password** field, if needed. This field is greyed out when the connection is local.
- Click **OK** to validate.

Click **Refresh** to update the list of projects if needed. Then choose the relevant project name and click **OK** to open it.

If you already created projects with previous releases of **JasperETL**, you can import them into your current **JasperETL** workspace using the **Import** function.

Related topic: *Importing projects on page 343*

When creating a project for the first time, there are no default project listed. Click **Create** to launch the Creation wizard.

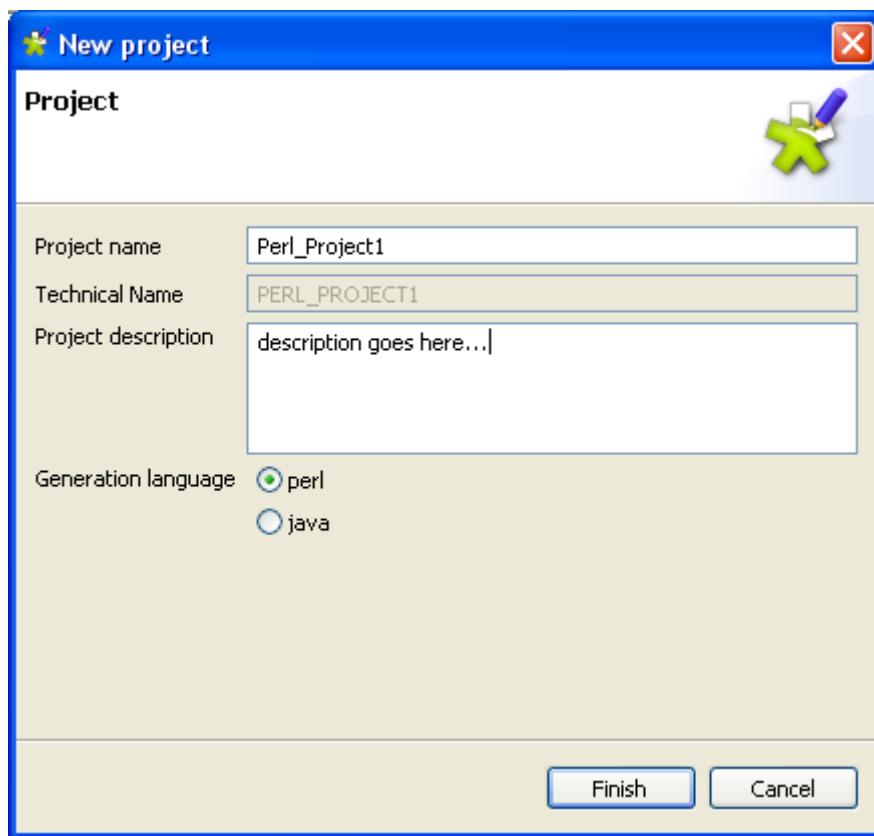
Related topic: *Creating a project on page 12*

You can discover **JasperETL** based on job samples. Install the demos project, in one click, through the **Import demos** button. The Demos project folder is automatically installed in your workspace. And the project is directly accessible from the login window.

When creating a new project, a folder tree is automatically created in the Workspace directory on your repository server. This will correspond to the **Repository** folder tree displaying on **JasperETL** main window.

Creating a project

When you create a project, you need first to fill in a **name** for this project. This field is mandatory.



A contextual message pops up at the top of the window, according to the location of your cursor. It informs you about the nature of data to be filled in, such as forbidden characters.

Note: Note that numbers are not allowed to be used to start a project name. The name is not case sensitive, therefore, YourProject or YOURPROJECT are the same.

The **Technical name** is used by the application as file name of the actual project file. The read-only name usually corresponds to the project name, upper-cased and concatenated with underscores if needed.

Select the **Generation language** between **Perl** and **Java**. From then, you will be required to use the relevant code, i.e. Perl code in perl projects and java code in Java projects.

If you want to switch from one to another projects go through **File > Switch Projects...**

Note: We advise you though to keep Perl projects and Java projects in separate locations and workspaces to avoid language conflicts.

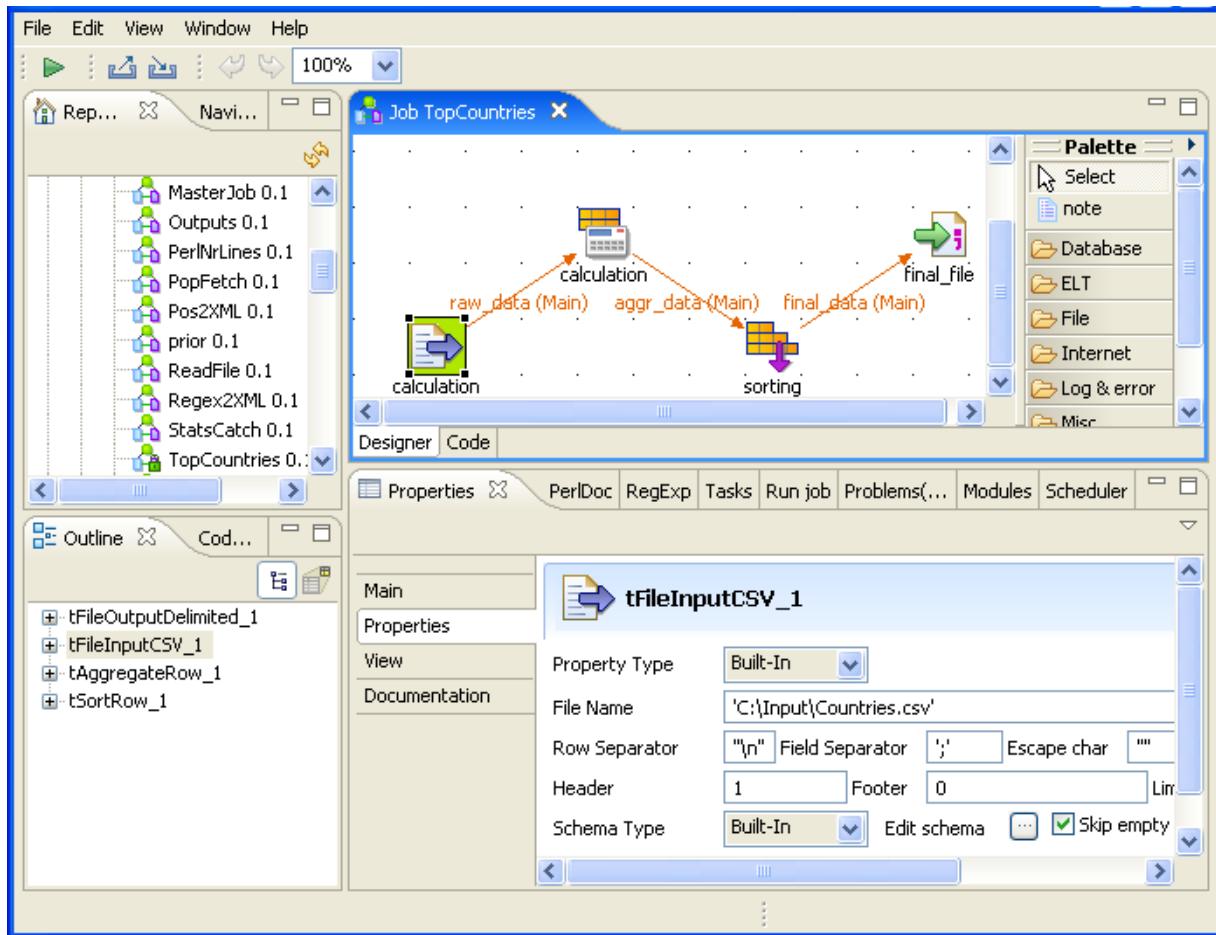
If you already used **JasperETL** and want to import projects from a previous release, see *Importing projects on page 343*.

In the Login window, select the project you've created. Click **OK** to launch **JasperETL**.

Note: A generation initialization window comes up when launching the application. Wait until the initialization is complete.

Describing the GUI

JasperETL opens on a multi-panel window.



JasperETL window is composed of the following panels:

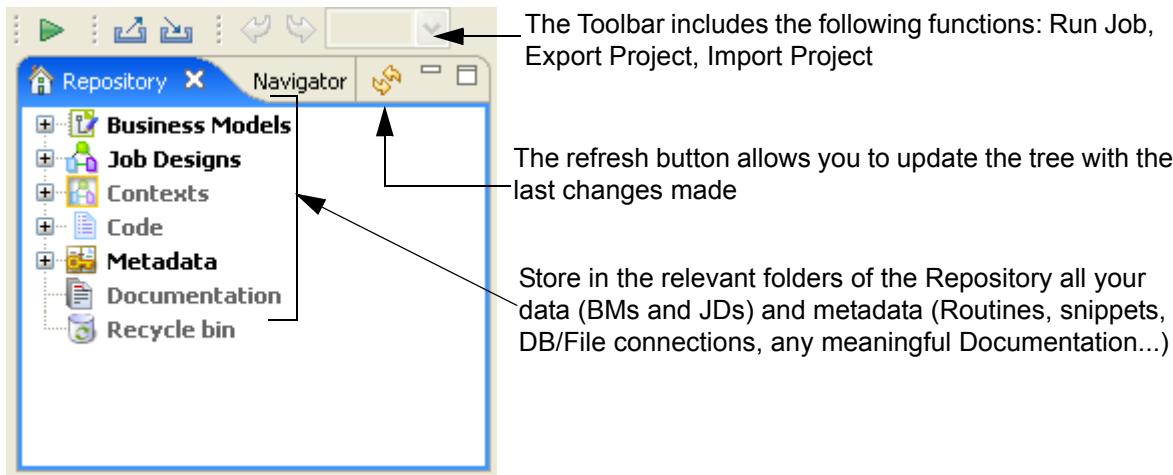
- Repository
- Graphical workspace
- Properties, Run and Logs views
- Outline and Code Viewer

The various panels and their respective features are detailed hereafter.

Repository

The **Repository** is a toolbox gathering all technical items that can be used either to describe business models or to design job designs. It gives access to the Business models, the job designs, as well as reusable routines or documentation.

The Repository centralizes and stores on the file system all necessary elements for any job design and business modeling contained in a project.



The repository gathers together the following components in a folder tree view:

Business Models

Under the **Business Models** node, are grouped all business models of the project. Double-click on the name of the model to open it on the graphical modeling workspace.

Related topic: *Designing a Business Model on page 27*

Job Designs

The **Job designs** folder shows all job flowcharts designed for the current project. Double-click on the name of the flowcharts to open it on the modeling workspace.

Related topic: *Designing a Job Design on page 39*

Code

The **Code** library groups the routines available for this project as well as snippets (to come) and other pieces of code that could be reused in the project.

Click on the relevant tree entry to develop the appropriate code piece.

Related topic: *Designing a Job Design on page 39*

Routines

A **Routine** is a piece of code which can be iterative in a technical job hence is likely to be reused several times within the same project.

Under **Routines**, a **System** folder groups all pre-defined routines. Developing this node again in the repository, various routine files display such as **Dates**, **Misc** and **String** gathering default pieces of codes according to their nature.

Double-click on one of the file. The **Routines editor** opens up as a new tab and can be moved around the modeling workspace by simply holding down the mouse and releasing at the target location.

Use these routines as reference for building your own or copy the one you need into the relevant properties field of your job.

To create a new routine, right-click on the Routines entry of the Repository, and select **Create a routine** in the pop-up menu. The routine editor opens up on a template file containing a default piece of code such as:

```
sub printTalend {  
    print "Talend\n"
```

Replace it with your own and when closing it, the routine is saved as a new node under Routines.

You can also create directories to classify the user's routines.

Note: The System folder, along with its content is read-only.

Snippets

Snippets are small pieces of code that can be duplicated across components or jobs to automate transformation for example. This feature will be available soon.

Documentation

The **Documentation** directory gathers all types of documents, of any format. This could be, for example, specification documents or a description of technical format of a file. Double-click to open the document in the relevant application.

Related topic: *Designing a Job Design on page 39*

Metadata

The **Metadata** folder bundles files holding redundant information you want to reuse in various jobs, such as schemas and property data.

Related topic: *Defining Metadata items on page 54*

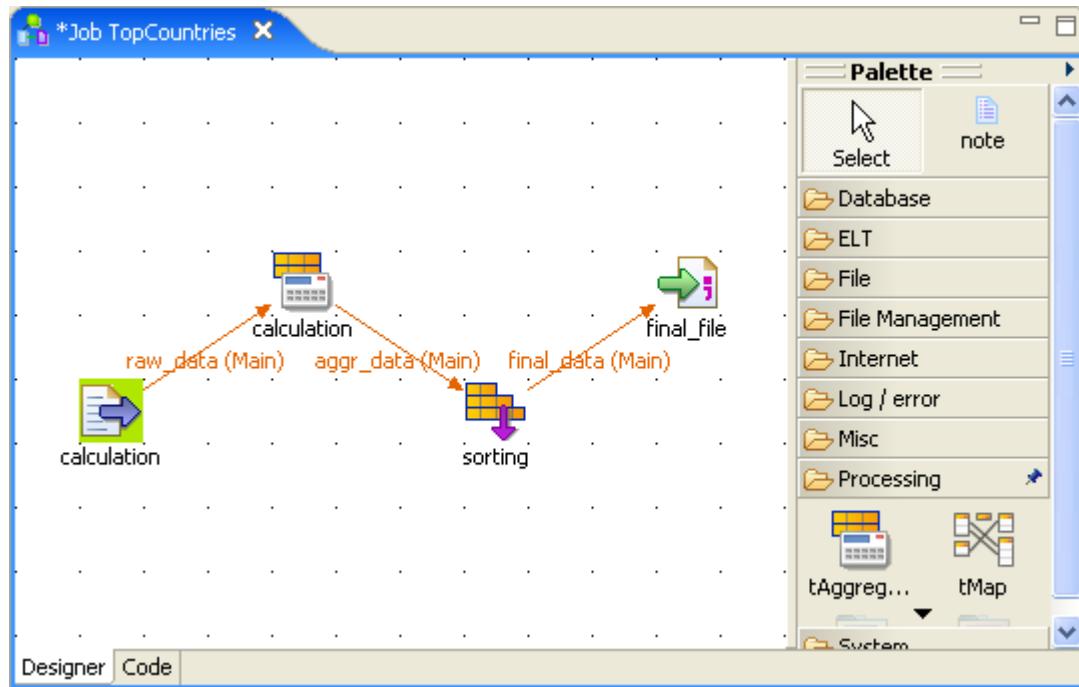
Recycle bin

Drag and drop elements from the **Repository** tree into the recycle bin or press *del* key to get rid of irrelevant or obsolete items

Note that the deleted elements are still present on your filesystem, in the recycle bin, until you right-click on the recycle bin icon and select **Empty Recycle bin**.

Graphical workspace

The **Graphical workspace** is **JasperETL**'s single flowcharting editor, where both business models as well as job designs can be laid out.



You can open and edit both job designs and business models in this single graphical editor. Flowcharts you open display in a handy tab system.

A **Palette** is docked at the top of the **workspace** to help you draw the model corresponding to your workflow needs.

Palette

From the **Palette**, depending on whether you're designing a job or modeling a business model, click and drop shapes, branches, notes or technical components to the workspace, then define and format them using the various tools offered in the **Properties** panel.

Related topics:

- *Designing a Business Model on page 27*
- *Designing a Job Design on page 39*

Changing the palette position

If the **Palette** doesn't show or if you want to set it apart in a panel, go to **Window > Show view...> General > Palette**. The Palette opens in a separate view that you can move around wherever you like within **JasperETL**'s window.

Changing the palette layout and settings

You can change the layout of the component list to display components in column or in list, as icons only or with short description.

You can also enlarge the component icons for better readability of the component list.

To do so, right-click and select the option in the list or click **Settings** to open the configuration window and fine-tune the layout.

Properties, Run and Logs views

The **Properties**, **Run Jobs** and **Logs** tabs gather all information relative to the graphical elements in selection in the modeling workspace or the actual execution of a complete job.

See also: *Modules and Scheduler on page 18*

Properties

The content of the **Properties** tab varies according to the selected item in the workspace.

For instance, when inserting a shape in the modeling workspace, the **Properties** tab offers a range of formatting tools to help you customize your business model and improve the readability of the whole business model.

In the case, you are working on a job design, the **Properties** tab offers you to set the operating parameters of the component and hence set this way each step of the technical job.

Logs

The **Logs** are mainly used for job designs. They show the results or errors of particular job design.

Note: However note that the log tab has also an informative function for Perl component operating progress for example

Run Job

The **Run Job** tab obviously shows the current job execution. This tab becomes a log console at the end of an execution.

For details about the job execution, see *Running a job on page 101*.

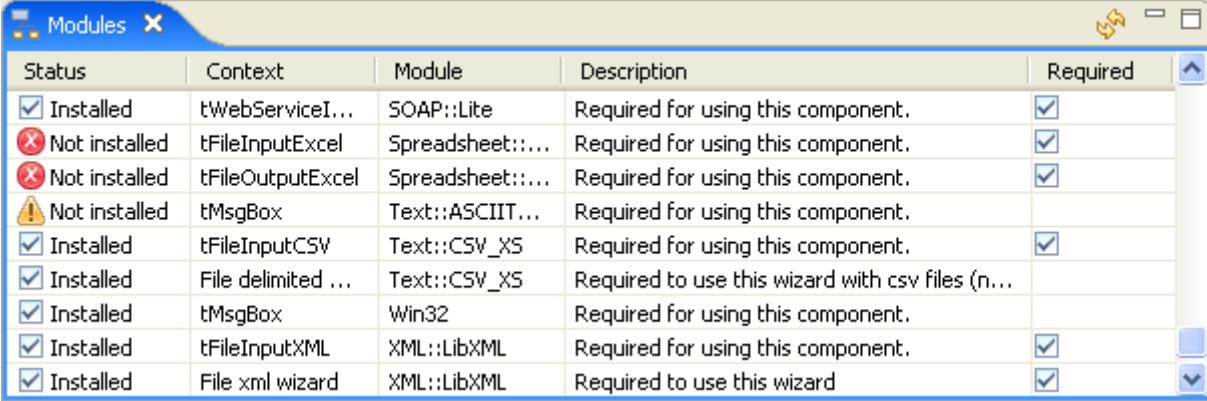
Modules and Scheduler

The **Modules** and **Scheduler** tabs are located in the same tab system as the **Properties**, **Logs** and **Run Job** tabs. Both views are independent from the active or inactive jobs open on the workspace.

Modules view

The use of some components requires specific Perl modules to be installed, check the **Modules** view, what modules you have or should have to run smoothly your jobs.

If the **Modules** tab doesn't show on the tab system of your workspace, go to **Window > Show View... > Talend**, and select **Modules** in the developed Talend node.



Status	Context	Module	Description	Required
<input checked="" type="checkbox"/> Installed	tWebServiceI...	SOAP::Lite	Required for using this component.	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Not installed	tFileInputExcel	Spreadsheet::...	Required for using this component.	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Not installed	tFileOutputExcel	Spreadsheet::...	Required for using this component.	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Not installed	tMsgBox	Text::ASCIIIT...	Required for using this component.	
<input checked="" type="checkbox"/> Installed	tFileInputCSV	Text::CSV_XS	Required for using this component.	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Installed	File delimited ...	Text::CSV_XS	Required to use this wizard with csv files (n...	
<input checked="" type="checkbox"/> Installed	tMsgBox	Win32	Required for using this component.	
<input checked="" type="checkbox"/> Installed	tFileInputXML	XML::LibXML	Required for using this component.	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Installed	File xml wizard	XML::LibXML	Required to use this wizard	<input checked="" type="checkbox"/>

The view shows if a module is necessary and required for the use of a referenced component.

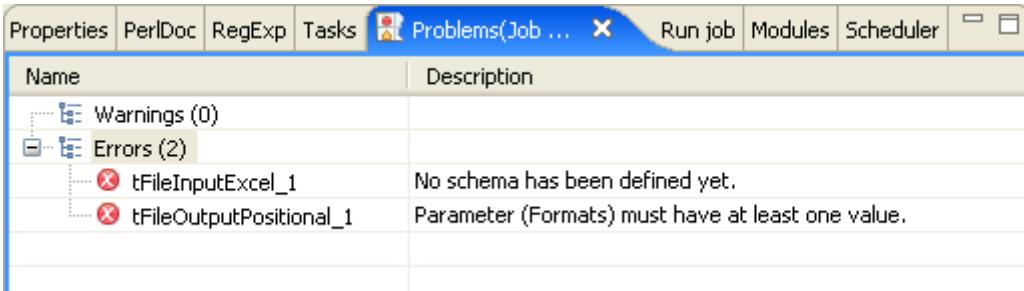
The **Status** column points out if the modules are yet or not yet installed on your system. The warning triangle icon indicates that the module is not necessarily required for this component.

For example, the *DBD::Oracle* module is only required for using tDBSQLRow if the latter is meant to run with Oracle DB. The same way, *DBD::Pg* module is only required if you use PostgreSQL. But all of them can be necessary.

The red crossed circle means the module is absolutely required for the component to run.

If the component field is empty, the module is then required for the general use of **JasperETL**.

When building your job, if a component misses a module that is absolutely required, an error is generated and displays on the **Problems** tab.



Properties		PerlDoc	RegExp	Tasks	Problems(Job ...)	Run job	Modules	Scheduler
Name	Description							
<input checked="" type="checkbox"/> Warnings (0)								
<input checked="" type="checkbox"/> Errors (2)								
<input checked="" type="checkbox"/> tFileInputExcel_1	No schema has been defined yet.							
<input checked="" type="checkbox"/> tFileOutputPositional_1	Parameter (Formats) must have at least one value.							

To install any missing Perl module, refer to the relevant installation manual on <http://talendforge.org/wiki/>

Open Scheduler

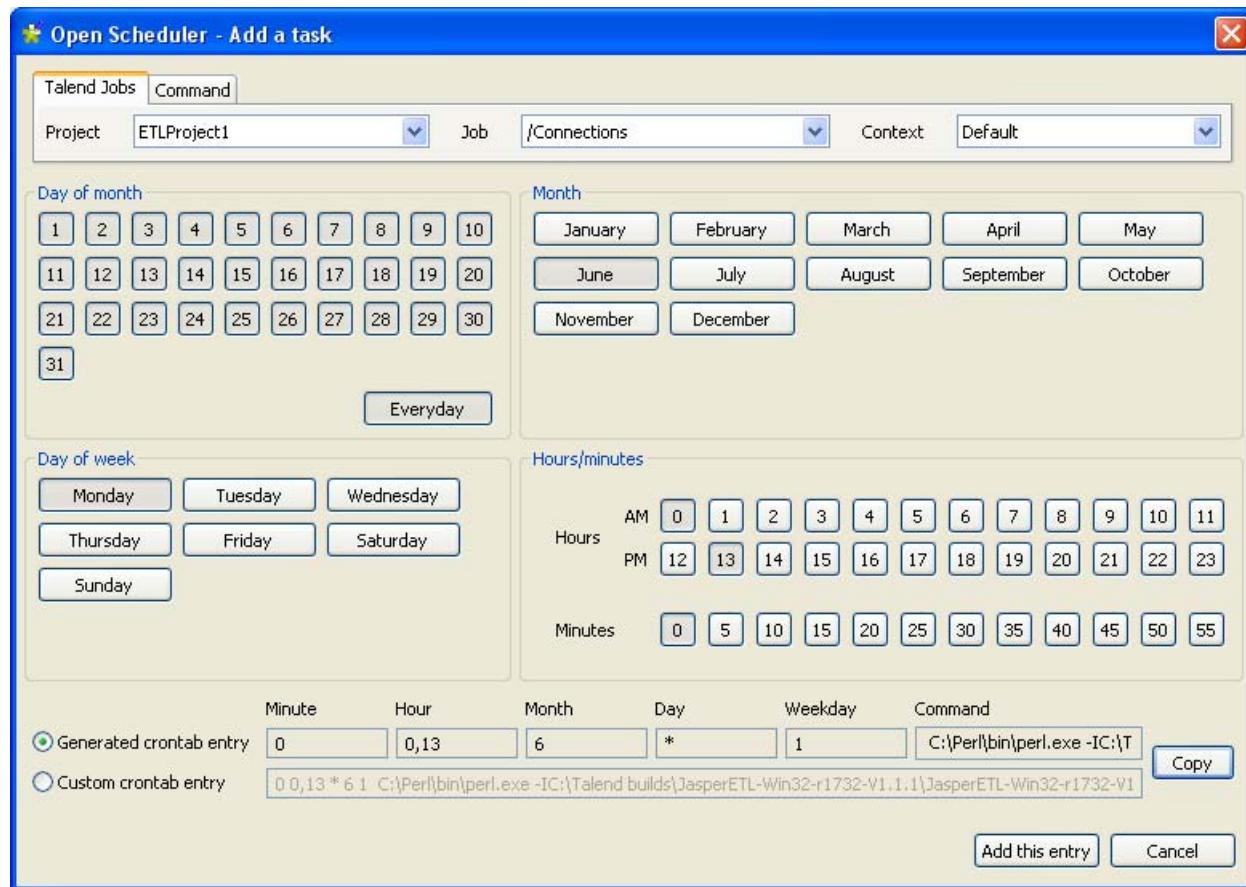
The Open Scheduler is based on the *crontab* command, found in Unix and Unix-like operating systems. This *cron* can be also installed on any Windows system.

Open Scheduler generates cron-compatible entries allowing you to launch periodically a job via the crontab program.

Getting started with JasperETL

Describing the GUI

If the **Scheduler** tab doesn't display on the tab system of your workspace, go to **Window > Show View... > Talend**, and select **Scheduler** in the developed Talend node.



Set the time and comprehensive date details to schedule the task.

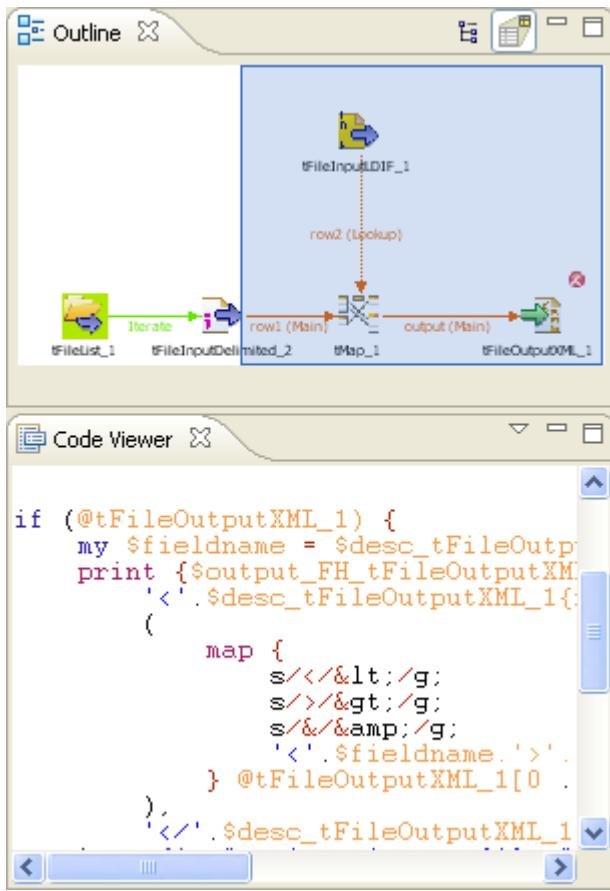
Open Scheduler automatically generates the corresponding task command that will be added to the crontab program.

Outline and Code Summary panel

The Information panel is composed of two tabs, **Outline** and **Code Viewer**, which provide information regarding the displayed diagram (either job design or business model).

Outline

The **Outline** tab offers a quick view of the business model or job design open on the modeling workspace and a tree view of variables. As the workspace, like any other window area can be resized upon your needs. The Outline view is convenient to check out where about on your workspace, you are located.



This graphical representation of the diagram highlights in a blue rectangle the diagram part showing in the workspace.

Click on the blue-highlighted view and hold down the mouse button. Then, move the rectangle over the job.

The view in the workspace moves accordingly.

The **Outline** view can also be displaying a folder tree view of components in use in the current diagram. Expand the node of a component, to show the list of variables available for this component.

To switch from the graphical outline view to the tree view, click on either icon docked at the top right of the panel.

Code viewer

The **Code viewer** tab provides lines of code generated for the selected component, behind the active job design view, as well the run menu including Start, Body and End elements.

Note: Note that this view only concerns the job design code, as no code is generated from business models.

Using a graphical colored code view, the tab shows the code of the component selected in the workspace. This is a partial view of the primary Code tab docked at the bottom of the workspace, which shows the code generated for the whole job.

Toolbar and Menus

At the top of **JasperETL** main window, a tool bar as well as various menus gather Talend commonly features along with some Eclipse functions.

Quick access toolbar

The toolbar allows you to access quickly the most commonly used functions. It slightly differs if you work at a Job or a Business Model.



The toolbar allows a quick access to the following actions:

- **Run Job**: Executes the job currently shown on the design workspace. For more information about job execution, see *Running a job on page 101*.
- **Export project**: Launches the Export project wizard. For more information about project export, see *Exporting projects on page 346*.
- **Import project**: Launches the Import project wizard. For more information about project import, see *Importing projects on page 343*.
- **Undo/Redo**: Allows you to redo or undo the last action you performed.
- **Zoom in/out**: Select the zoom percentage to zoom in or zoom out on your Job.

Menus

JasperETL's menus include :

- some standard functions, such as **Save**, **Print**, **Exit**, which are to be used at the application level.
- some Eclipse native features to be used mainly at the **Job Designer** level.
- as well as specific **JasperETL** functions.

Although standard **Job** or **Business Model** creation and edition are only available through right-click on the relevant view, some **JasperETL** features are offered in **Menus**.

In **Window > Preferences > Talend**, you can set your preferences. For more information about preferences, see *Configuring JasperETL preferences on page 22*.

In **Window > Show views**, you can manage the different views to display at the bottom of **JasperETL**.

Configuring **JasperETL** preferences

JasperETL opens up on a multiple panel window.

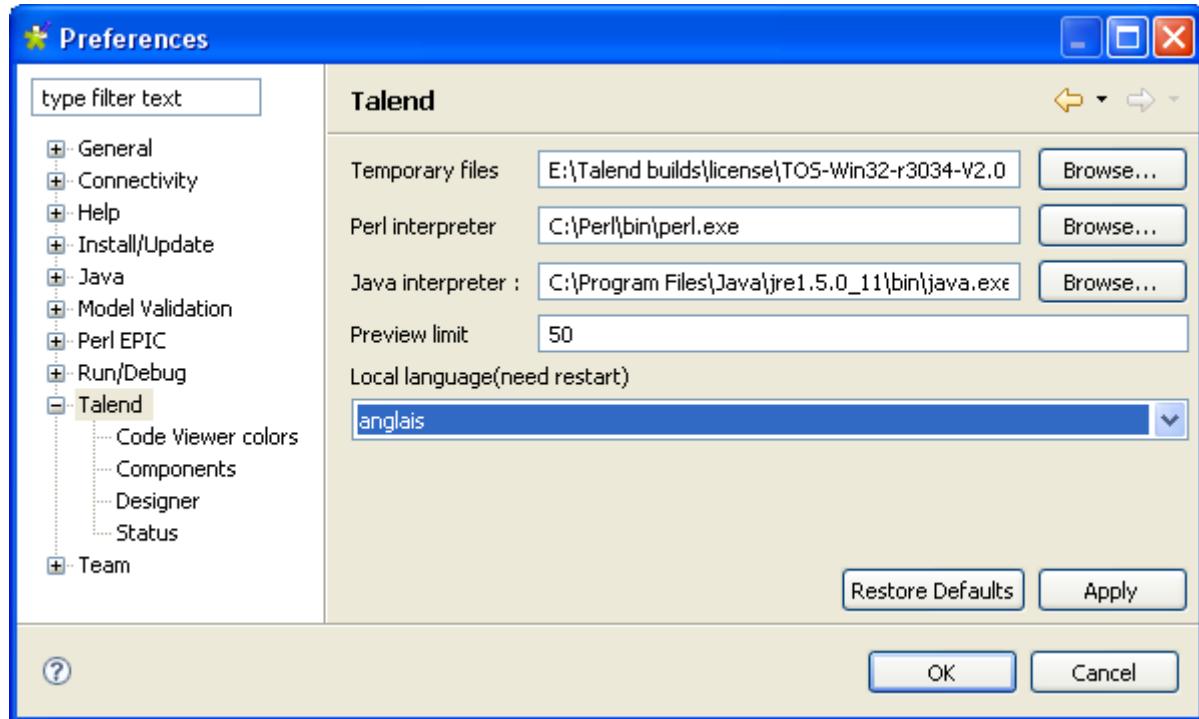
You can define various properties of **JasperETL** main workspace according to your needs and preferences.

First, click on the **Window** menu of your **JasperETL**, then select **Preferences**.

Perl/Java Interpreter path

In the preferences, you might need to let **JasperETL** pointing to the right interpreter path.

- If needed, click on the **Talend** node on the **Preferences** tree (left).
- Enter a path to the Perl/Java interpreter if the default directory does not display the right path.

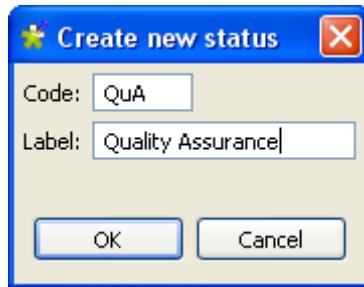


On the same view, you can also change the preview limit and the path to the temporary files or the OS language.

Status

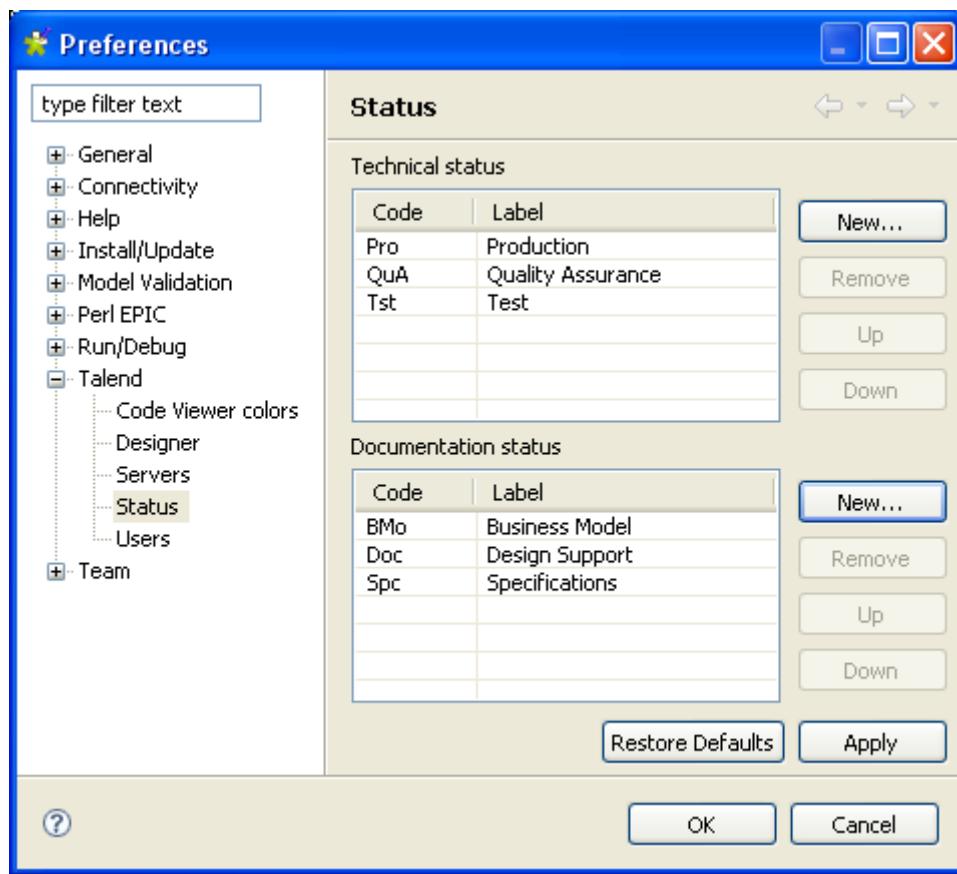
Under the **Talend** node, you can also define the Status.

- Expand the **Talend** node, and click on **Status** to define the main properties of your **Repository** elements.
- The main properties panel of a Repository item gathers information data such as **Name**, **Purpose**, **Description**, **Author**, **Version** and **Status** of the selected item. Most properties are free text fields, but the **Status** field, which is a drop-down list.
- Populate the **Status** list with the most relevant values, according to your needs. Note that the **Code** can not be more than 3-character long and the **Label** is required.



Talend makes a difference between two status types: **Technical status** and **Documentation status**.

The **Technical status** list displays classification codes for elements which are to be running on stations, such as jobs, metadata or routines.



The **Documentation status** list helps classifying the elements of Repository which can be used to document processes (Business Models or documentation).

Once you completed the status setting, click **OK** to save.

The **Status** list will offer the status levels you defined here when defining the main properties of your job designs and business models.

External components

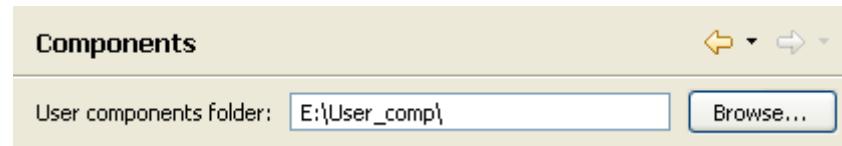
You can create/develop your own components and use them in **JasperETL**.

For more information about the creation and development of user components, refer to our wiki *Component creation tutorial section*.

In the **Preferences** folder tree, expand the **Talend** node, then select **Components**.

Getting started with JasperETL

Configuring JasperETL preferences



- Fill in the **User components folder** path to the components to be added to the **Palette** of **JasperETL**.
- Restart **JasperETL** for the components to show in the Palette.

—Designing a Business Model—

Designing a Business Model

JasperETL offers the best tool to put in place the Top/Down approach allowing high stakeholders to get the grip on analytics of a project from the most general business model to the most precise details in its technical application.

This chapter aims at business managers, decision makers or developers who want to model their flow management needs at a macro level.

Objectives

A Business model is a non technical view of a business workflow need.

Generally, a typical business model will include the strategic systems or process steps already up and running in your company as well as new needs. You can symbolise these systems, steps and needs using multiple shapes and create the connections among them. Likely, all of them can be easily described using repository attributes and formatting tools.

In the **Graphical workspace** of **JasperETL**, you can use multiple tools in order to:

- draw your business needs
- create and assign numerous repository items to your model objects
- define appearance properties of your model objects.

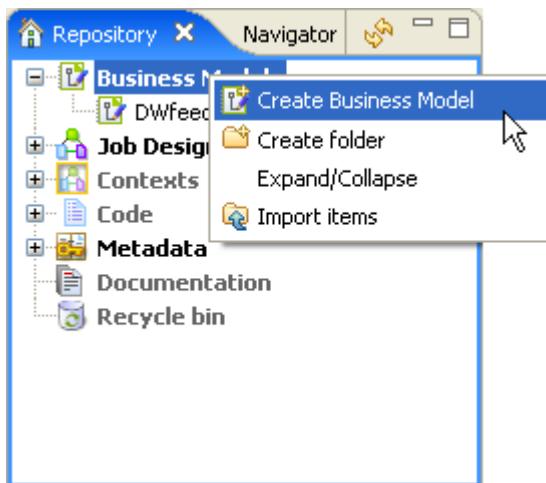
Opening or creating a business model

Open **JasperETL** following the procedure as detailed in the paragraph *Accessing JasperETL on page 9*.

From the main page of **JasperETL**, click on the **Business Models** node of the **Repository** panel to expand the business models tree.

Designing a Business Model

Opening or creating a business model



Select the *Expand/Collapse* option of the right-click menu, to display all existing business models (if any).

Opening a business model

Double-click on the name of the model to be opened.

The workspace opens up on the selected business model view.

Creating a business model

Right-click on the **Business Models** node and select *Create Business Model*.

The Creation wizard guides through the steps to create a

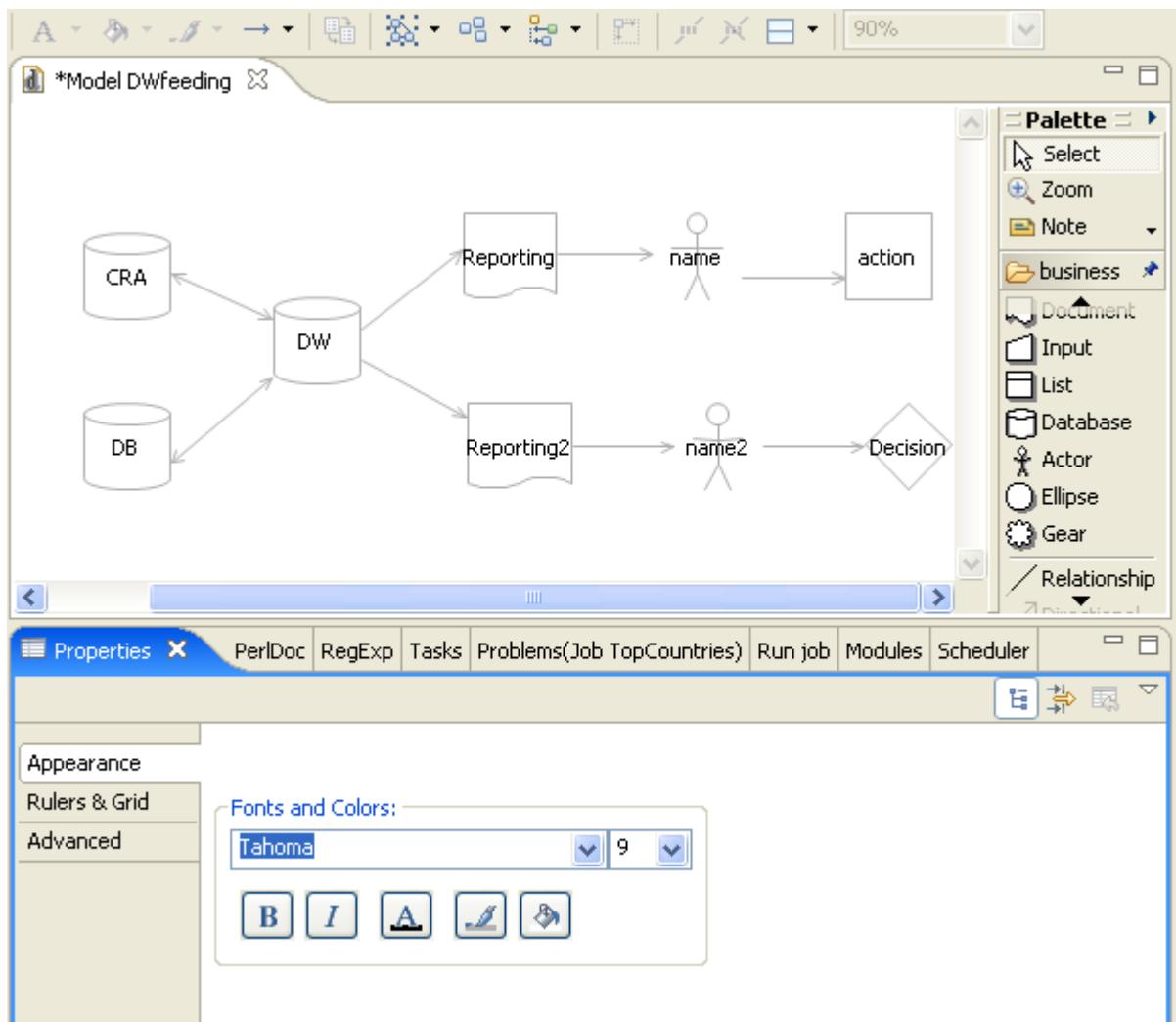
new business model.

Select the **Location** folder where you want the new model to be stored in.

And fill in a **Name** for it. The name you allocate to the file shows as a label on the tab of the model designer.

The **Modeler** opens up on the empty design workspace.

You can create as many models as you want and open them all, they will display in a tab system on your workspace.



The **Modeler** is made of the following panels:

- JasperETL's graphical workspace
- a Palette of shapes and lines specific to the Business modeling
- the Properties panel showing specific information about all or part of the model.

Modeling a business model

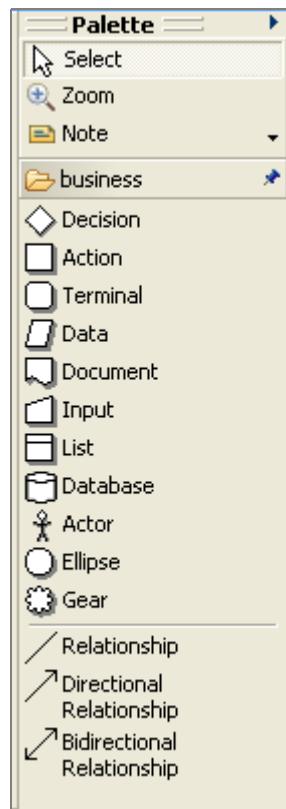
If you have multiple tabs opened on your designer workspace, click on the relevant tab in order to show the appropriate model information.

Note: Properties panel and Menus' items display indeed information relative to the active model.

Use the **Palette** to click and drop the relevant shapes and connect them together with branches and arrange or improve the model visual aspect by zooming in or out.

This **Palette** offers graphical representations for *objects* interacting within a business model.

The *objects* can be of different types, from strategic system to output document or decision step. Each one having a specific role in your business model according to the description, definition and assignment you give to it.



All objects are represented in the **Palette** as *shapes*, and can be included in the model.

Note: If the shapes do not show on the Palette, click on the **business** folder symbol to roll down the library of shapes.

Shapes

Select the shape corresponding to the relevant *object* you want to include in your business model. Double-click on it or click on the shape in the Palette and drop it in the modeling area.

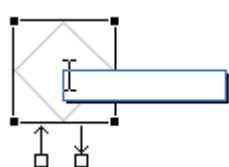
Alternatively, for a quick access to the shape library, keep your cursor still on the modeling area for a couple of seconds to display the quick access toolbar:



For instance, if your business process includes a decision step, select the diamond shape in the Palette to add this decision step to your model.

Note: When you mouse over the quick access toolbar, a tooltip helps you to identify the shapes.

Then a simple click will do to make it show on the modeling area.



The shape is placed in a dotted black frame. Pull the corner dots to resize it at your own convenience.

Also, a blue-edged input box allows you to add a label to the shape. Give an expressive name in order for you to be able to identify at a glance the role of this shape in the model.

Two arrows from and to the added shape allow you to create connections with other shapes. You can hence quickly define sequence order or dependencies between shapes.

Related topic: *Connecting shapes on page 31*.

The available shapes include:

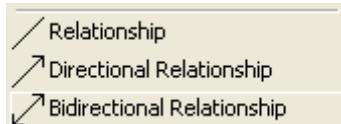
Table 1—

Callout	Details
Decision	The diamond shape generally represents an if condition in the model. Allows to take context-sensitive actions.
Action	The square shape can be used to symbolize actions of any nature, such as transformation, translation or formatting.
Terminal	The rounded corner square can illustrate any type of output terminal
Data	A parallelogram shape symbolize data of any type.
Document	Inserts a Document object which can be any type of document and can be used as input or output for the data processed.
Input	Inserts an input object allowing the user to type in or manually provide data to be processed.
List	forms a list with the extracted data. The list can be defined to hold a certain nature of data
Database	Inserts a database object which can hold the input or output data to be processed.
Actor	This schematic character symbolizes players in the decision-support as well technical processes
Ellipse	Inserts an ellipse shape
Gear	This gearing piece can be used to illustrate pieces of code programmed manually that should be replaced by a JasperSoft job for example.

Connecting shapes

When designing your business model, you want to implement relations between a source shape and a target shape;

There are two possible ways to connect shapes in your workspace:



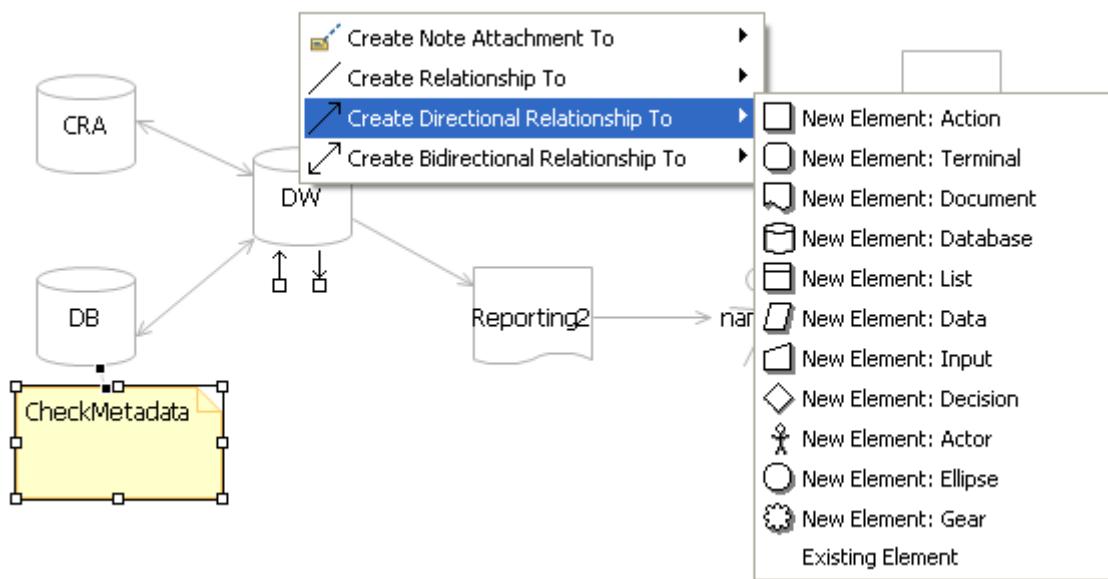
Either select the relevant **Relationship** tool in the **Palette**. Then, in the modeling workspace, pull a link from one shape to the other to draw a connection between them.

Designing a Business Model

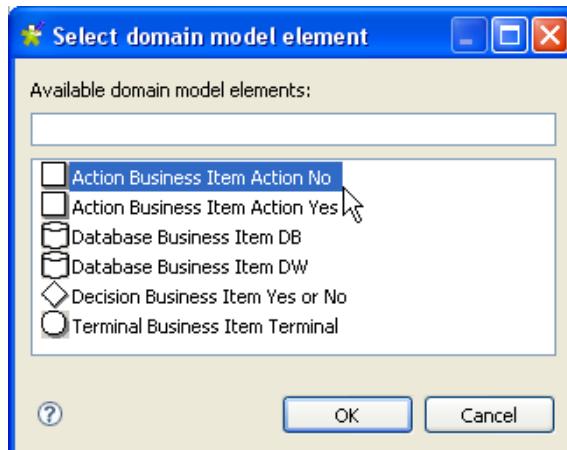
Modeling a business model

Or, you can implement both the relationship and the element to be related to or from, in few clicks.

- Simply mouse over a shape that you already dropped on your design workspace, in order to display the double connection arrows.
- Select the relevant arrow to implement the correct directional connection if need be.
- Drag a link towards an empty area of the design workspace and release to display the connections popup menu.
- Select the appropriate connection among the list of relationships to or from. You can choose among **simple relationship**, **directional relationship** or **bidirectional relationship**.
- Then, select the appropriate element to connect to, among the items listed.



You can create a connection to an existing element of the model. Select **Existing Element** in the popup menu and choose the existing element you want to connect to in the displaying list box.



The connection is automatically created with the selected shape.

The nature of this connection can be defined using **Repository** elements, and can be formatted and labelled in the **Properties** panel, see *Properties on page 34*.

When creating a connection, an input box allows you to add a label to the connection you've created. Choose a meaningful name to help you identify the type of relationship you created.

Note: You can also add note and comment to your model in order to identify elements or connections at a later stage.

Related topic: *Commenting and arranging a model on page 33*

Commenting and arranging a model

The tools of the **Palette** allow you to customize your model:

Table 2—

Callout	Details
Select	Select and move the shapes and lines around in Designer's modeling area.
Zoom	Zoom in to a part of the model. To watch more accurately part of the model. To zoom out, press <i>Shift</i> and click on the modeling area.
Note/Text/Note attachment	Allows comments and notes to be added in order to store any useful information regarding the model or part of it.

Adding a note or free text

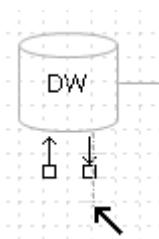
To add a note, select the **Note** tool in the **Palette**, docked at the right of the workspace.

Alternatively right-click on the model or the shape you want to link the note to, and select *Add Note*

A sticky note displays on the modeling area. If the note is linked to a particular shape, a line is automatically drawn to the shape.

Type in the text in the input box or, if the latter doesn't show, type in directly on the sticky note.

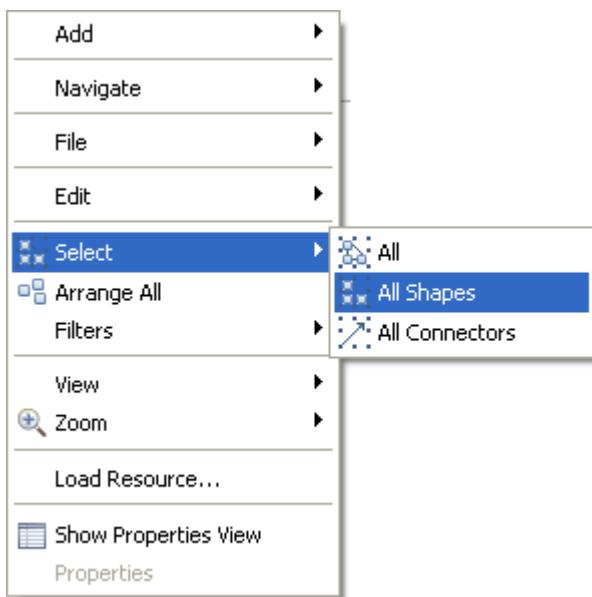
If you want to link your notes and specific shapes of your model, click on the down arrow next to the **Note** tool on the **Palette** and select **Note attachment**. Pull the black arrow towards an empty area of the design workspace, and release. The popup menu offers you to attach a new Note to the selected shape.



You can also select the *Add Text* feature to type in free text directly in the modeling area. You can access this feature in the **Note** drop-down menu of the **Palette** or via a shortcut located next to the *Add Note* feature on the quick access toolbar.

Arranging the model view

You can also rearrange the look and feel of your model via the right-click menu.



Place your cursor in the design area, right-click to display the menu and select *Arrange all*. The shapes automatically move around to give the best possible reading of the model.

Alternatively, you can select manually the whole model or part of it.

To do so, right-click on any part of the modeling area, and click *Select*.

You can select :

- **All** shapes and connectors of the model,
- **All shapes** used in the design workspace,
- **All connectors** branching together the shapes.

From this menu you can also zoom in and out to part of the model and change the view of the model.

Properties

The **Properties** information corresponds to the current selection, if any. This can be the whole model if you selected all shapes of it or more specifically one of the shapes it is made of. If nothing is selected, the **Properties** give general formatting information about the workspace.

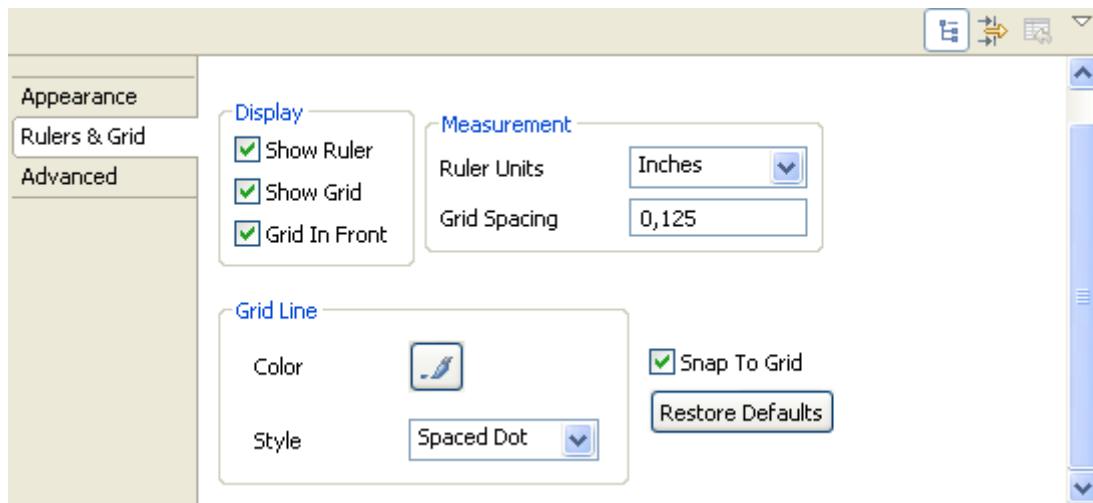
The **Properties** tab contains different type of information regarding:

- Rulers and Grid
- Appearance
- Assignment

Rulers and Grid

To display the **Rulers & Grid** tab, select the **Select** tool on the Palette, then click on any empty area of the design workspace to deselect any current selection.

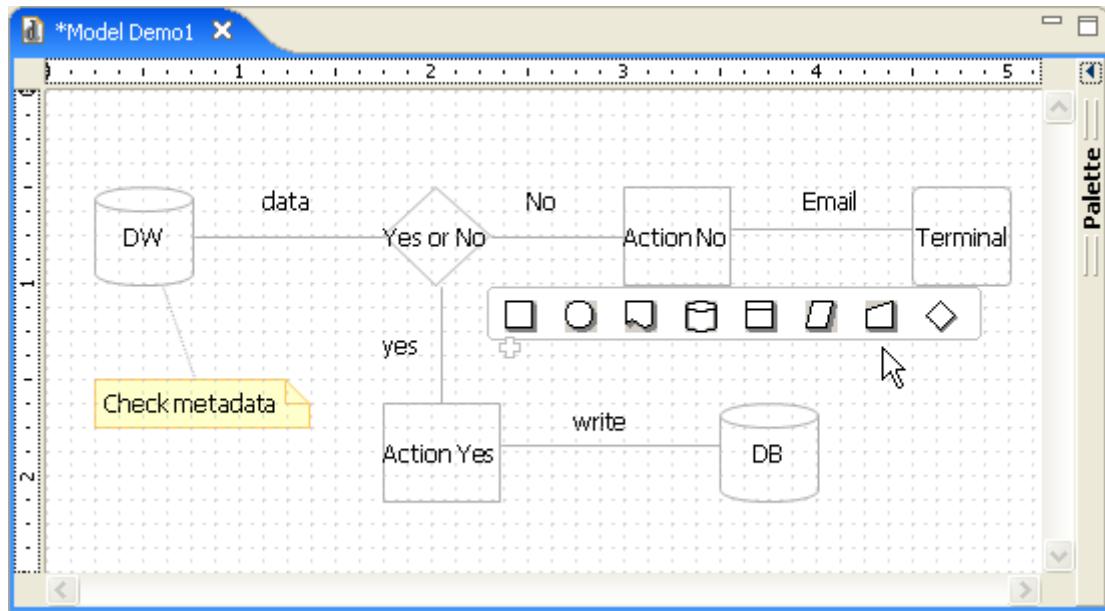
Click on the **Rulers & Grid** tab to access the ruler and grid setting panels.



Check the boxes to show the **Ruler**, the **Grid** or both.

Grid in front sends to the back all shapes of the model. Select the ruling unit among **Centimeters, inches or pixels**.

You can also choose the color as well as the style of the grid lines or restore the default settings.



Appearance

From the **Appearance** tab you can apply filling or border colours, change the appearance of shapes and lines in order to customize your business model or make it easier to read.

Designing a Business Model

Assigning repository elements to a Business Model

The **Properties** tab includes the following formats:

- fill the shape with selected colour.
- color the shape border
- insert text above the shape
- insert gradient colours to the shape
- insert shadow to the shape

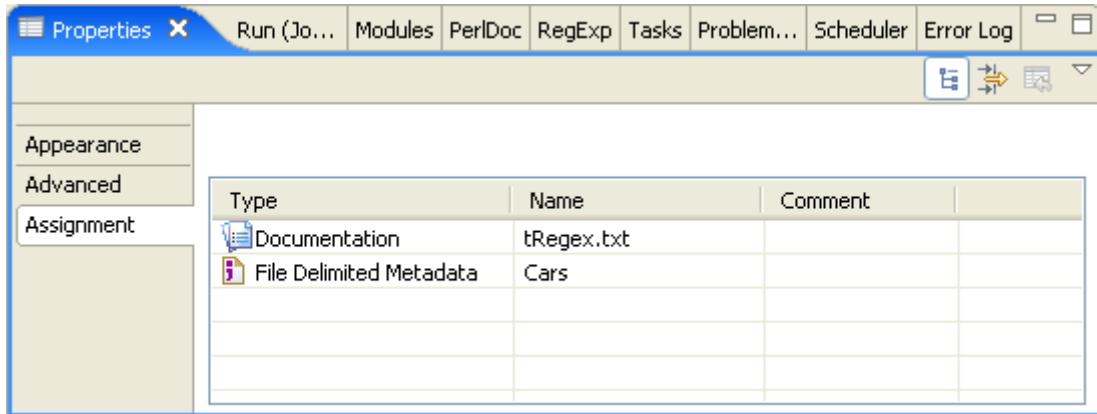
You can also move and manage shapes of your model using the edition tools. Right-click on the relevant shape to access these editing tools.

Assignment

The Assignment table displays details of the Repository attributes you allocated to a shape or a connection.

To display any assignment information in the table, select a shape or a connection in the active model.

You can modify some information. Also, if you update data from the Repository tree, assignment information gets automatically updated.



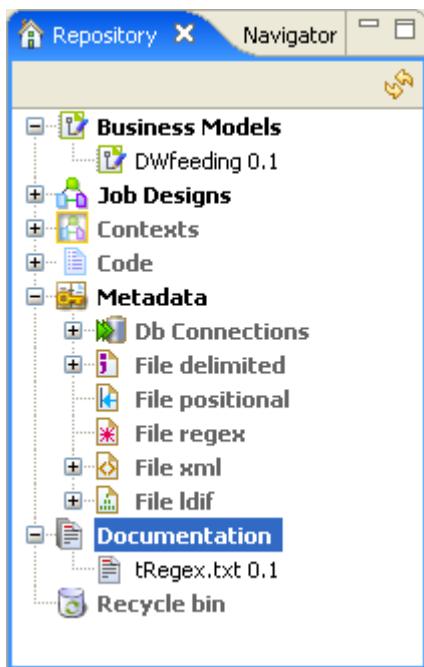
The screenshot shows the JasperETL Properties dialog box. The title bar says "Properties". The menu bar includes "Run (Jo...)", "Modules", "PerlDoc", "RegExp", "Tasks", "Problem...", "Scheduler", and "Error Log". The toolbar has icons for copy, paste, and search. On the left, there's a vertical stack of tabs: "Appearance" (selected), "Advanced", and "Assignment". The main area is a table titled "Assignment". It has columns for "Type", "Name", and "Comment". There are two rows: one for "Documentation" named "tRegex.txt" and another for "File Delimited Metadata" named "Cars".

Type	Name	Comment
Documentation	tRegex.txt	
File Delimited Metadata	Cars	

For further information about how to assign elements to a Business Model, see *Assigning repository elements to a Business Model* on page 36.

Assigning repository elements to a Business Model

The **Assignment table** lists the components from the **Repository** panel of the main window.



You can define or describe a particular object of your Business Model, by associating to it, various types of information .

You can set the nature of the data to be assigned, thus facilitating the job design phase.

The same way as with shapes and connecting lines, simply drag and drop an item from the **Repository** panel to assign it to the relevant shape in the modeling workspace.

Alternatively, you can use the *Un/Assign* button to carry out this operation.

The **Assignment table**, located underneath the workspace gets automatically updated accordingly with the assigned information of the selected object.

You can remove assignments through the *Un/Assign* button.

The **Repository** offers the following types of items that you can assign:

Table 3—

Component	Details
Job designs	If any available job designs developed for other projects in the same repository can be reused in the active business model
Metadata	Any describing data about any of the objects used in the model. It can be connection information to a database for example.
Business Models	If other business model of this repository have been designed, they can be reused in the active model.
Documentation	Any type of documentation in any format. It can be a technical documentation, some guidelines in text format or a simple description of your databases.
Routines (Code)	If some routines have been developed in a previous project, to automate tasks for example, they can be reused in the active model. Routines are stored in the Code folder of the Repository

For more information about the **Repository** Components, see *Designing a Job Design on page 39*

Editing a Business model

Follow the relevant procedure according to your needs:

Renaming a business model

Click on the current business model label on the **Repository** panel, to display the corresponding **Main** properties information.

Then make your edits on the **Name** field. The label is changed automatically on the Repository and will be reflected on the Model tab of the workspace, the next time you open it.

Copying and pasting a business model

In **Repository > Business model**, right-click on the business model name to be copied and select **Copy** in the popup menu, or press **Ctrl+c**.

Then right-click where you want to paste your business model, and select **Paste**.

Moving a business model

To move a business model from a location to another in your business models project folder, select a business model in the **Repository > Business Models** tree.

Then simply drag and drop it to the new location.

Alternatively, right-click on the relevant business model and select **Move** in the popup menu.

Deleting a business model

Right-click on the name of the model to be deleted and select **Delete** in the popup menu.

Alternatively, simply select the relevant business model, then drag and drop it into the **Recycle bin** of the **Repository** panel.

Saving a business model

To save a business model, click on **File > Save** or press **Ctrl+s**. The model is saved under the name you gave during the creation process.

An asterisk displays in front of the business model name tab when changes have been made to the model but not yet saved.



—Designing a Job Design—

Designing a Job Design

This chapter aims at programmers or IT managers who are ready to implement technical aspects of a business model (designed or not in **JasperETL**'s Business Modeler). **JasperETL** helps you to develop the job design that will allow you to put in place an up and running dataflow management.

Objectives

A job design is the runnable layer of a business model. It translates business needs into code, routines and programs, in other words it technically implements your data flow.

The Job design is the graphical and functional view of a technical process.

From **JasperETL**, you can:

- put in place actions in your job design using a library of technical components.
- change the default setting of components or create new components or family of components to match your exact needs.
- set connections and relationships between components in order to define the sequence and the nature of actions
- access code at any time to edit in Perl or document your job components.
- create and add items to the **Repository** for reuse and sharing purposes (in other projects or jobs or with other users).

Opening or Creating a job

Open **JasperETL** following the procedure as detailed in chapter *Accessing JasperETL on page 9*.

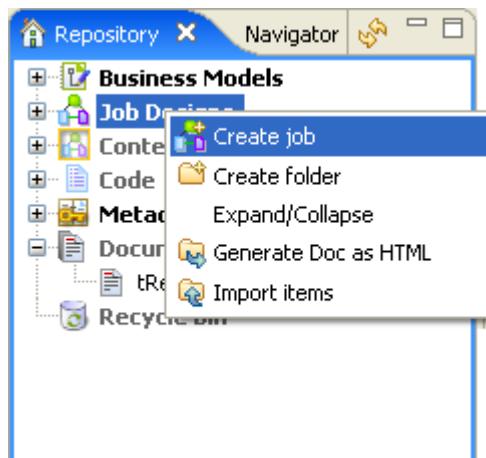
In **JasperETL** Repository panel, click on the **Job Designs** node to expand the technical job tree.

You can create folders via the right-click menu to gather together families of jobs. Right-click on the **Job Designs** node, and choose *Create folder*. Give a name to this folder and click **OK**.

If you have already created jobs that you want to move in this new folder, simply drag and drop them into the folder.

Designing a Job Design

Opening or Creating a job



Opening a job

Double-click on the label of the job you want to open.

The **Designer** opens up on the selected job last view.

Note: You can open as many job designs as you need. They will all display in a tab system on your workspace.

job main properties.

Creating a job

Right-click on the Job Designs node and select *Create job* in the pop-up menu. The Creation wizard helps you to define the new

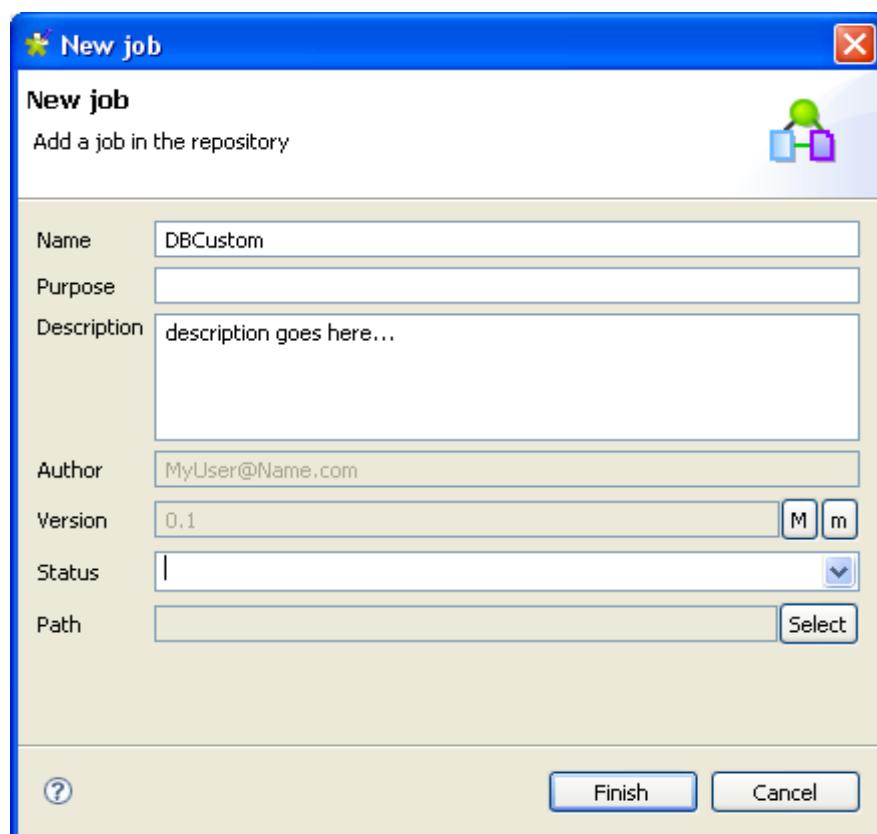


Table 4—

Field	Description
Name	Enter a name for your new job. A message comes up if you enter prohibited characters.
Purpose	Enter the job purpose or any useful information regarding the job use.
Description	Enter a description if need be for the job created.
Author	The Author field is read-only as it shows by default the current user login.
Version	The Version is also read-only. You can manually increment the version using the M and m button
Status	You can define the status of a job in your preferences. By default none is defined. To define them, go to Window > Preferences > Talend >Status.

The **Designer** opens an empty tab, on the workspace, showing only the job name as tab label.

Note: You can create as many job designs as you want and open them all, they will display in a tab system on your workspace.

The **Designer** is made of the following panels:

- **JasperETL's** Graphical workspace
- a **Palette** of components and connections specific to the Job Designer
- a **Properties** panel which can be edited to change or set parameters related to a particular part or component of the model.

Getting started with a Job Design

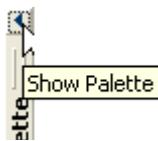
Until a job is created, the design workspace as well as the palette of components are greyed out.

If you're designing a job for the first time, the workspace opens on an empty area. If you're opening an already existing job, it opens on the last view it was saved on.

Showing, hiding and moving the palette

The **Palette** contains all basic elements to create the most complex jobs in the design workspace. These components are gathered in families and sub-families.

By default, the palette is hidden on the right side of your design workspace.



If you want the **Palette** to show permanently, click on the left arrow, at the right top corner of the designer, to make it visible at all time.

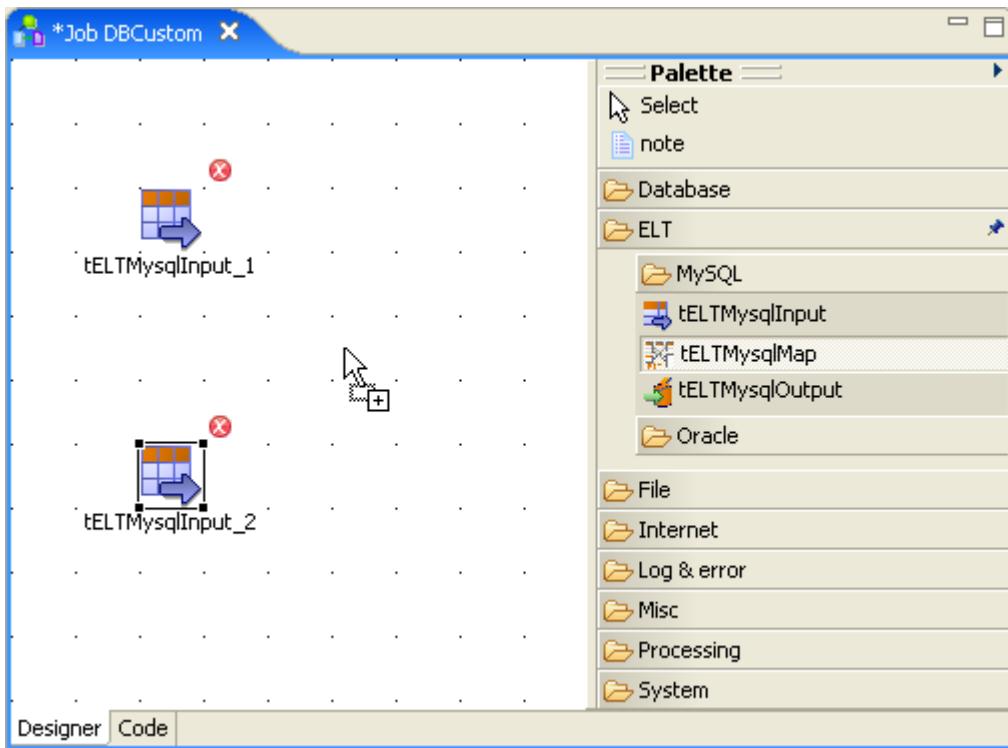
For specific component properties, see chapter *Components on page 115*.

You can also move around the **Palette** outside the workspace within **JasperETL's** window. To enable the standalone **Palette** view, click on **Window** menu > **Show View...** > **General** > **Palette**.

Click and drop elements

Click on a **Component** or a **Note** to start with, on the **Palette**. Then click again to drop it on the workspace and add it to your job design.

If the Palette doesn't show, see *Showing, hiding and moving the palette on page 41*.



Multiple information or warnings may show next to the component. Browse over the component icon to display the information tooltip. This will display until you fully completed your job design and defined each component properties.

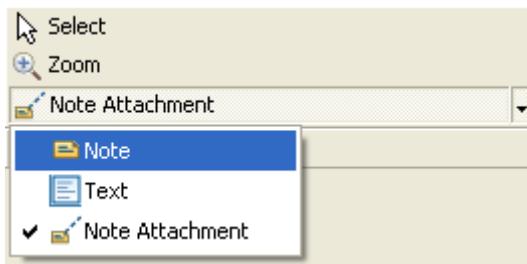
WARNING—you will be required to use the relevant code, i.e. Perl code in perl jobs and java code in Java jobs.

Related topics:

- *Connecting components together on page 45*
- *Warnings and errors on component on page 44*
- *Defining job Properties on page 49*

Adding Notes to a job design

Select the relevant note tool in the list among **Note**, **Text** or **Note attachment**. These various note options are also available through a right-click.



Click and drop the **Note** element onto the workspace to add a note to a particular component or to the whole job.

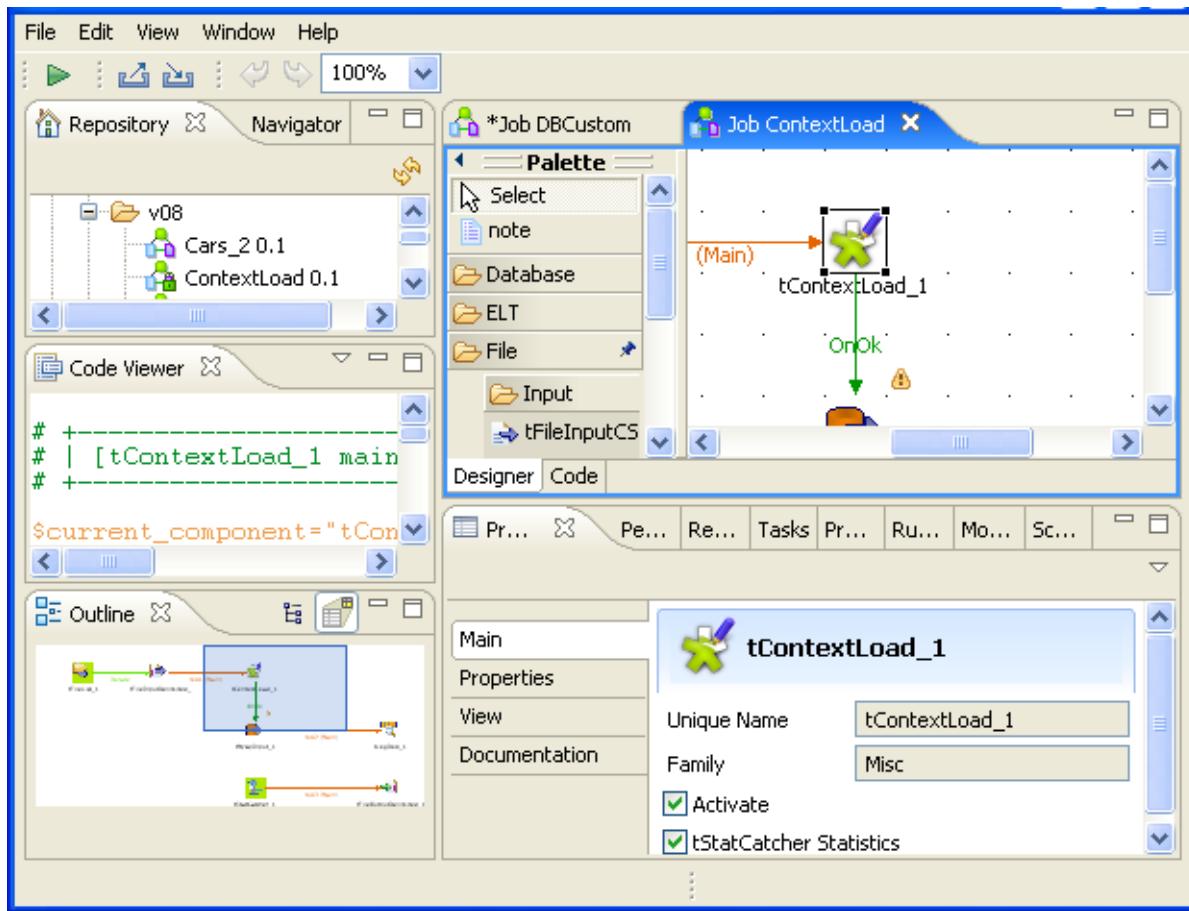
The **Note** shows as a sticky note on the design workspace.

The **Text** note allows you to type in directly onto the workspace.

The **Note attachment** allows you to bind the sticky note to particular element of the workspace.

Changing panels position

All panels can be moved around according to your needs.



Click on the border or on a tab, hold down the mouse button and drag the panel to the target destination. Release to change the panel position.

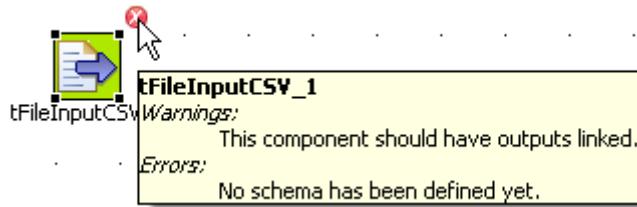
Click on the cross to close one view. To get a view back on display, click **Window > Show View > Talend**, then click on the name of the panel you want to add to your current view or see *Shortcuts and aliases on page 107*.

If the **Palette** doesn't show or if you want to set it apart in a panel, go to **Window > Show view...> General > Palette**. The Palette opens in a separate view that you can move around wherever you like within **JasperETL**'s window.

Warnings and errors on component

When a component is not properly defined or if the link to the next component does not exist yet, a red checked circle or a warning sign is docked at the component icon.

Mouse over the component, to display the tooltip messages or warnings along with the label. This context-sensitive help informs you about any missing data or component status.



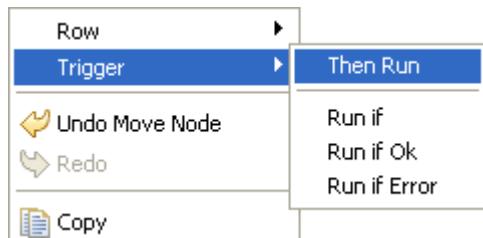
Connecting components together

There are various types of connections which define either the data to be processed, the data output, or else the job logical sequence.

- On your workspace, when dragging the link away from your source component towards the destination component, a graphical plug indicates if the destination component is valid or not. The black crossed circle only disappears once you reached a valid target component.

Connection types

Only the connections authorized for the selected component are listed on the right-click pop-up menu.



The types of connections proposed are different for each component according to its nature and role within the job, if the connection is meant to transfer data (from a defined schema) or if no data is handled.

The types of connections available depend also if the data come from one or multiple input files and get transferred towards one or multiple outputs.

Select a component on the workspace, and right-click to display the pop-up menu. All links available for the selected component display.

Row connection

The **Row** connection handles actual data transfer. The Row links can be **main**, **lookup** or **output** according to the nature of the flow processed.

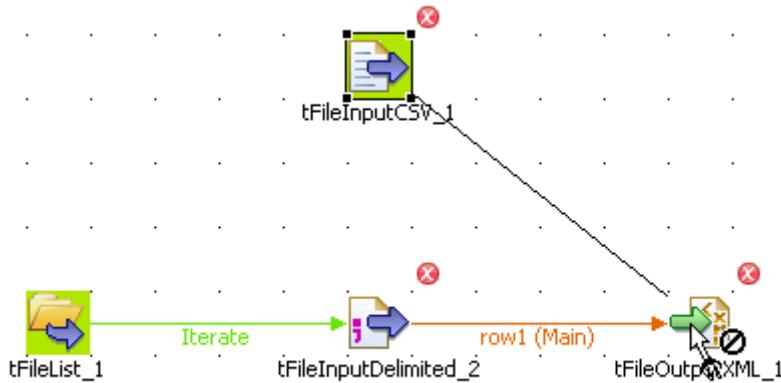
Main row

The **Main** row is the most commonly used connection. It passes on data flows from one component to the other, iterating on each row and reading input data according to the component properties setting (schema).

Data transferred through main rows are characterized by a schema definition which describes the data structure in the input file.

Note: Note that you cannot connect two Input components together using a **main Row** connection.

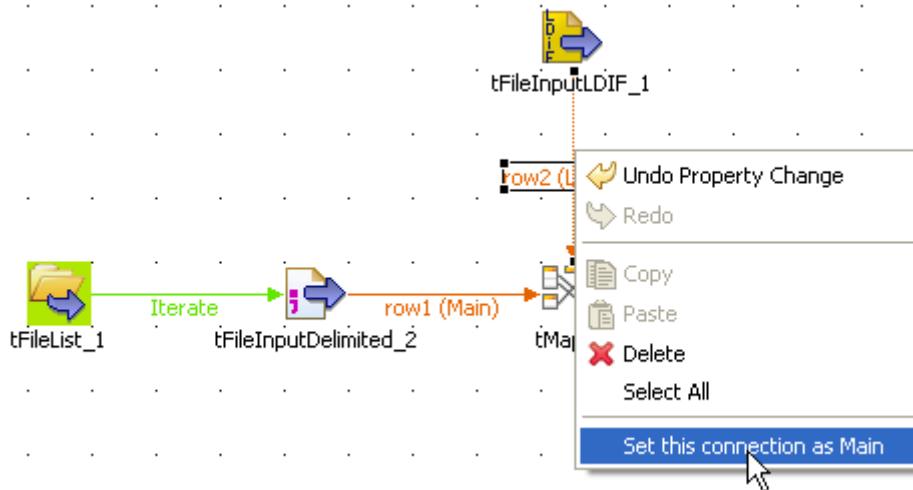
Note: Note also that only *one* incoming Row connection is possible per component. You will not be able to link twice the same target component using a main Row connection.



To be able to use multiple Row connections, see *Multiple Input/Output job on page 48*.

Lookup row

The **Lookup** row is a Row connecting a sub-flow component to a main flow component. This connection is used only in the case of multiple input flows.



A **Lookup** row can be changed into a main row at any time (and in reverse, a main row can be changed to a lookup row). To do so, right-click on the row to be changed, and on the pop-up menu, click on **Set this connection as Main**.

Related topic: *Multiple Input/Output job on page 48*.

Output row

The **Output** row is a Row connecting a component to the final output component. As the job output can be multiple, you get prompted to give a name for each output row created.

Note: Note that the system remembers also deleted output link names (and properties if they were defined) to avoid you to fill in again name and property data in case you want to reuse them.

Related topic: *Multiple Input/Output job on page 48.*

Iterate connection

The **Iterate** connection can be used to loop on files contained in a directory, on rows contained in a file or on DB entries.

A component can be the target of only one **Iterate** link. The **Iterate** link is mainly to be connected to the Start component of a flow (either main or secondary).

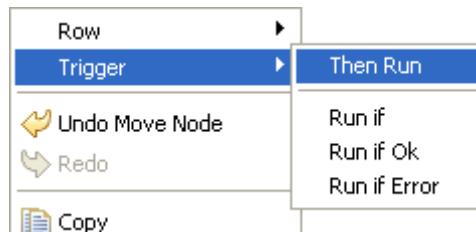
Some components are meant to be connected through an iterate link with the next component, such as tFilelist component.

Note that the **Iterate** link name is read-only unlike the other connections.

Trigger connections

The trigger connections define the processing sequence. No data is handled through trigger connections.

The connection in use will create a dependency between jobs or sub-jobs which therefore will trigger one after the other according to the trigger nature.



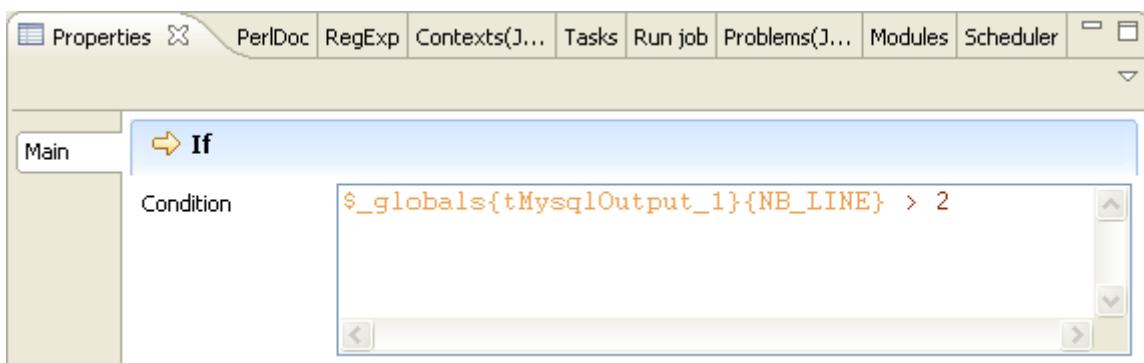
There are two kinds of triggers: chronological trigger and contextual triggers.

ThenRun (previously Run Before and Run after) is a chronological trigger, in the way, that you run the first component and then run the next component. This connection is to be used only with Start components.

Related topic: *Defining the Start component on page 53.*

Run if, Run if OK and Run if Error are contextual triggers. They can be used with any source component but are to be connected to Start component of a main or secondary job flow.

- **Run if OK** will only trigger the target component once the execution of the source component is complete. Its main use could be to trigger notification sub-jobs for example.
- **Run if Error** will trigger the sub-job or component as soon as an error is encountered in the primary job.
- **Run if** triggers a sub-job or component in case the condition defined is met. Click on the connection to display the **If** trigger **Properties** panel and set the condition in Perl or in Java according to the generation language you selected. The **Ctrl+Space bar** allows to access all global and context variables.



Link connection

The **Link** connection can only be used with ELT components. The Links transfer table schema information to the ELT mapper component in order to be used in specific DB query statements.

Related topics: *Components on page 115*

The **Link** connection therefore does not handle actual data but only the metadata regarding the table to be queried on.

When right-clicking on the ELT component to be connected, select **Link > New Output**.

WARNING—Be aware that the name you provide to the link MUST reflect the actual table name.

In fact, the link name will be used in the SQL statement generated through the ETL Mapper, therefore the same name should never be used twice.

Multiple Input/Output job

For the time being, if you need to handle data through multiple input points and/or multiple outputs and integrate a transformation in one flow, you want to use the **tMap** component, which is dedicated to this use.

For further information regarding data mapping , see *Mapping data flows in a job on page 80*.

For properties regarding the tMap component as well as use case scenarios, see *tMap on page 231*.

Defining job Properties

The **Properties** information shows detailed data. The **Main** tab thus recalls information relative to the author and job name as filled in at creation stage as well as other general information. And the other tabs show more specific information about the job or the component selected.

Main

The properties panel shows the **Main** properties of the selected component. The values of information data are filled in automatically by the component itself and will be used in the code. Therefore all fields are read-only, but the **Activate** box.

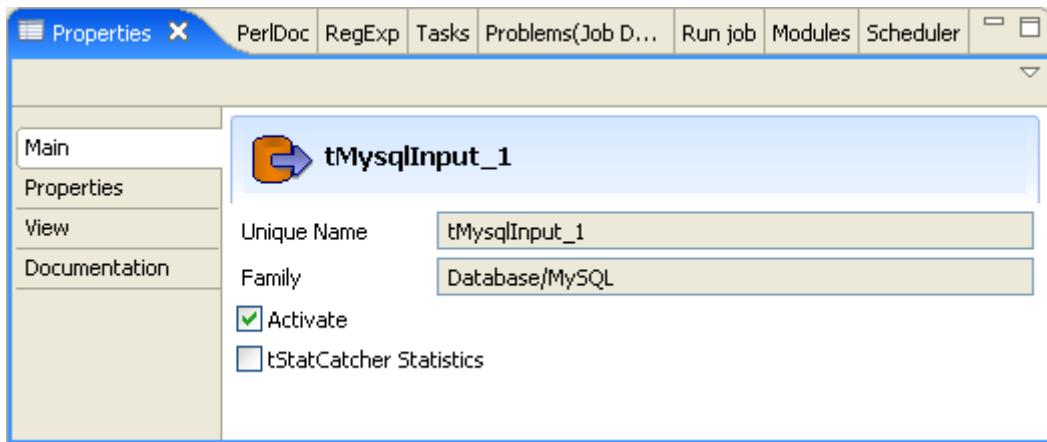


Table 5—

Field	Description
Unique Name	Unique identifier, allocated automatically by the system in order for it to be reused in the code.
Version	Component version, independant from the version of the whole product version.
Family	Group of components relative to the same function. This field is read-write and new family can be created here.
Activate	Check this box to activate the selected component and the directly linked job.
tStatCatcher Statistics	Check this box to allow the tStatCatcher component to aggregate processing data as defined in the properties of <i>tStatCatcher</i> on page 321

The **Activate** box enables the component function in the job or the sub-job it belongs to, hence code related to its properties will be generated.

If the **Activate** box is unchecked, obviously no code will be generated for the component itself but also for all directly related branches in the job.

For further information regarding the enabling/disabling job feature, see *Activating/Disabling a job or sub-job on page 93*.

View

The **View** tab of the **Properties** panel allows you to change the default display format of components on the workspace.

Table 6—

Field	Description
Label format	Free label showing on the workspace. Variables can be set to retrieve and display values from other fields. The field tooltip usually shows the corresponding variable where the field value is stored.
Hint format	Hidden tooltip, showing only when you mouse over the component.
Show hint	Check this box to enable the tooltip feature.

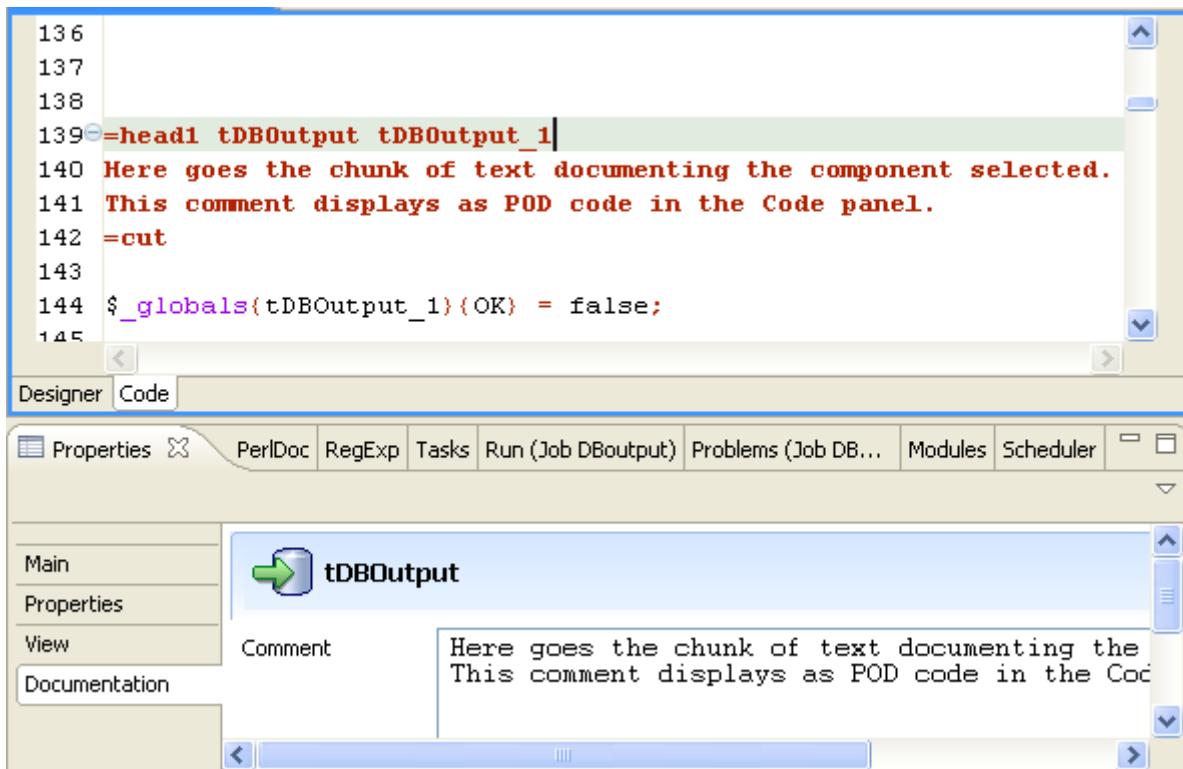
You can graphically highlight both **Label** and **Hint** text with HTML formatting tags:

- Bold: YourLabelOrHint
- Italic: <i> YourLabelOrHint </i>
- Return carriage: YourLabelOrHint
 ContdOnNextLine
- Color: YourLabelOrHint

To change your preferences of this View panel, click Window>Preferences>Talend>Designer.

Documentation

Feel free to add any useful comment or chunk of text or documentation to your component.



The content of this **Comment** field will be formatted using Pod markup and will be integrated in the generated code. You can view your comment in the **Code** panel.

You can show the Documentation in your hint tooltip using the associated variable (`_COMMENT_`)

For advanced use of Documentations, you can use the Repository Documentation area in order to store and reuse any type of documentation.

Properties

Each component has specific properties shown on the **Properties** tab of the **Properties** panel. See *Components on page 115* for details about how to fill in the fields.

- ⚠ Make sure you use the relevant code, i.e. Perl code in perl properties and java code in Java properties.

For all components you can centralize Properties information in metadata files located in the Repository Metadata directory. Select **Repository** as **Property type** and choose the metadata file holding the relevant information. Related topic: *Defining Metadata items on page 54*.

For all **Input**-type components, you can define the schema to follow in order to select data to be processed. Like the Properties data, this schema is either built-in or stored remotely in the Repository in a metadata file that you created.

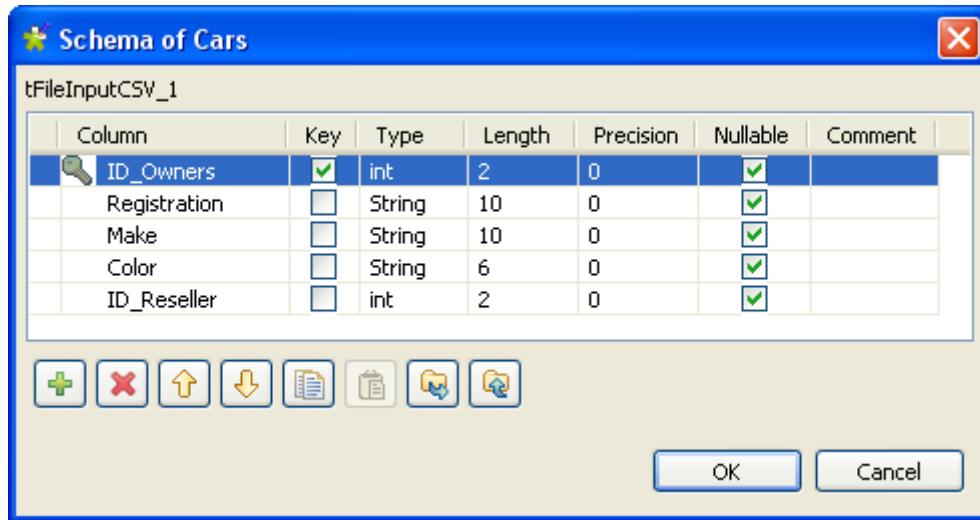
Setting a built-in schema

A schema created as *built in* the job is meant for a single use in a job, hence cannot be reused in another job nor station.

Designing a Job Design

Defining job Properties

Select Built-in in the list, and click on **Edit Schema** and create your built-in schema by adding columns and describing their content, according to the input file definition.



In all Output Properties also, you also have to define the schema of the output. To retrieve the schema defined in the Input schema, click on **Sync columns** button.

Note: In Java, some extra information is required. For more information about Date

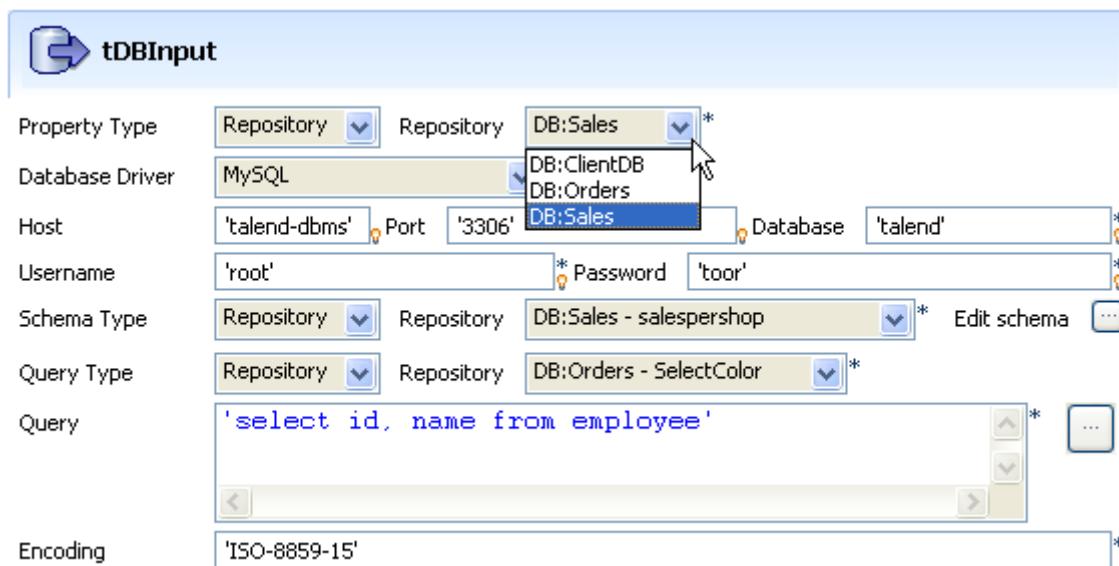
pattern for example, check out:

<http://java.sun.com/j2se/1.5.0/docs/api/index.html>

Setting a repository schema

You can avoid redundancy of schema metadata and hold them together in a central place, by creating metadata files and store them in the Repository Metadata directory.

To recall a metadata file into your current job, select the Schema type **Repository** and select the relevant metadata file in the list. Then click on **Edit Schema** to check the data are appropriate.



You are free to edit a repository schema used for a job. However, note that the schema hence becomes **built-in** to the current job.

You cannot change the remotely stored schema from this window.

Related topics: *Defining Metadata items on page 54*

Defining the Start component

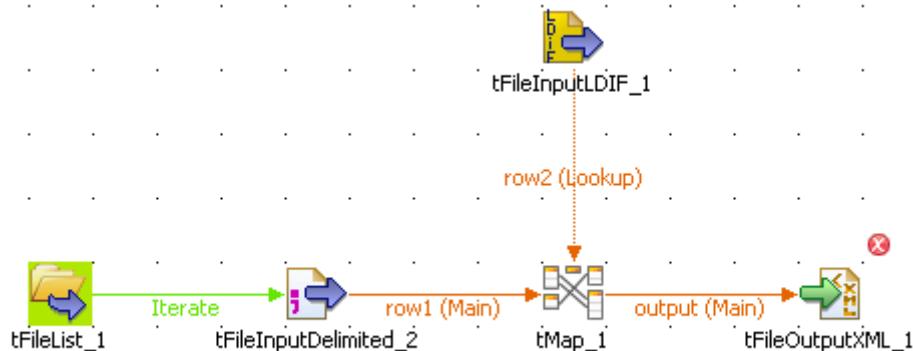
The **Start** component is the trigger of a job. There can be several Start components per job design if there are several flows running in parallel. But for one flow and its *connected* subflows, only one component can be the Start component.

Click and drop a component onto the workspace, all possible start components take a distinctive bright green background colour. Notice that most of the components, can be **Start** components.

Only components which don't make sense to trigger a flow, will not be proposed as Start components, such as tMap component for example.

Designing a Job Design

Defining Metadata items



To distinguish which component is to be the **Start** component of your job, identify the main flow and the secondary flows of your job

- The main flow should be the one connecting a component to the next component using a Row type link. The Start component is then automatically set on the first component of the main flow (icon with green background).
- The secondary flows are also connected using a Row-type link which is then called Lookup row on the workspace to distinguish it from the main flow. This Lookup flow is used to enrich the main flow with more data.

Be aware that you can change the Start component hence the main flow by changing a main Row into a Lookup Row, simply through a right-click on the row to be changed.

Related topics:

- *Connecting components together on page 45*
- *Activating/Disabling a job or sub-job on page 93*

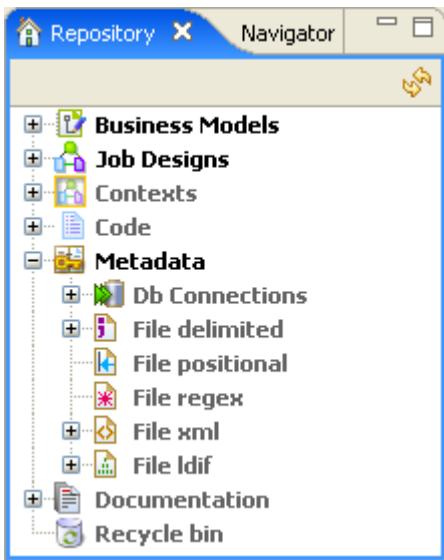
Defining Metadata items

JasperETL is a metadata-driven solution, and can therefore help you ensure the whole job consistency and quality, through a centralized metadata warehouse.

Use the Repository to store, in the Metadata area, the recurrent information on files used to build your job and retrieve them easily from the Properties panel of any component. These metadata generally include: DB connections, File path and schemas.

Follow two main steps to setup schemas either from a DB or a File-type connection.

First step is to setup a connection to the File or to the DB. Then second step is to define the schema based on DB table or File metadata.



This procedure differs slightly depending on the type of connection chosen. Below are the respective procedures to setup various connections and define multiple schemas.

Note that in any case, the connection to a file or DB serves only the purpose of copying the metadata, the schema will be based on, to your Repository (likely on your filesystem).

Click on Metadata in the Repository to expand the folder tree.

Each of the connection nodes will gather the various connections you setup.

Setting up a DB schema

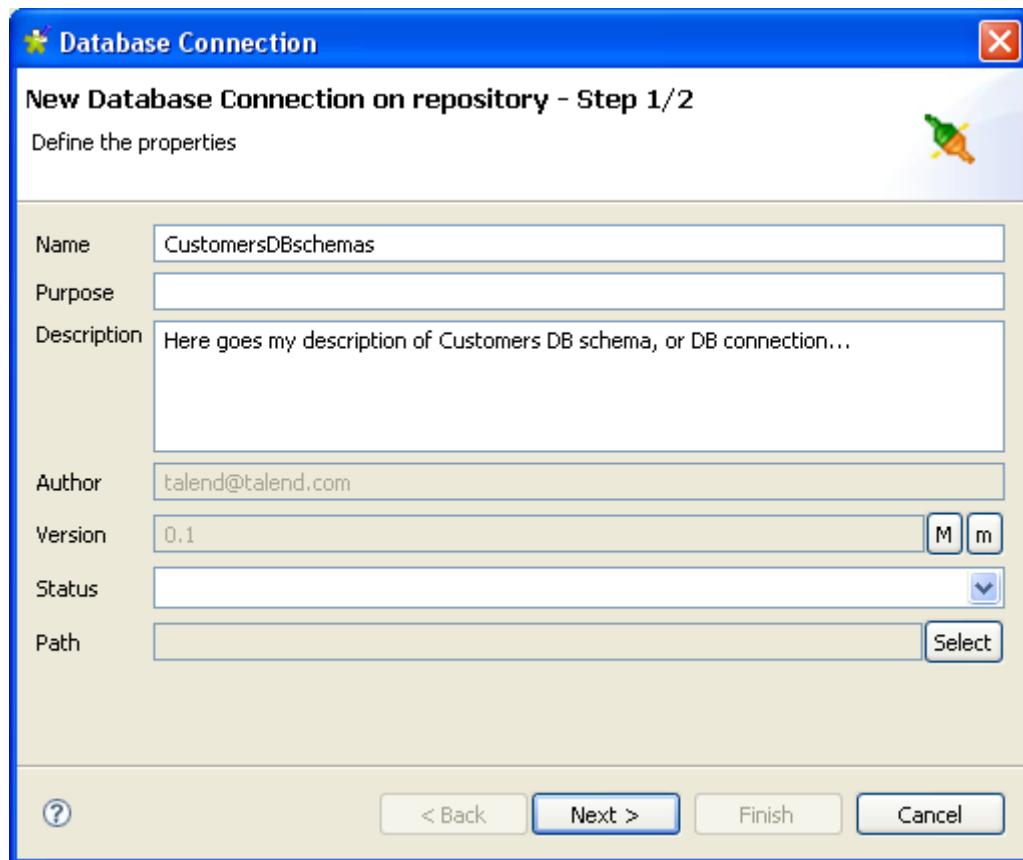
For DB table based schemas, the creation procedure is in two separate but closely related operations. First Right-click on **Db Connections** and select **Create connection** on the pop-up menu.

Step 1: general properties

A connection wizard opens up. Fill in the generic Schema properties such as Schema **Name** and **Description**. The **Status** field is a customized field you can define in *Window > Preferences*.

Designing a Job Design

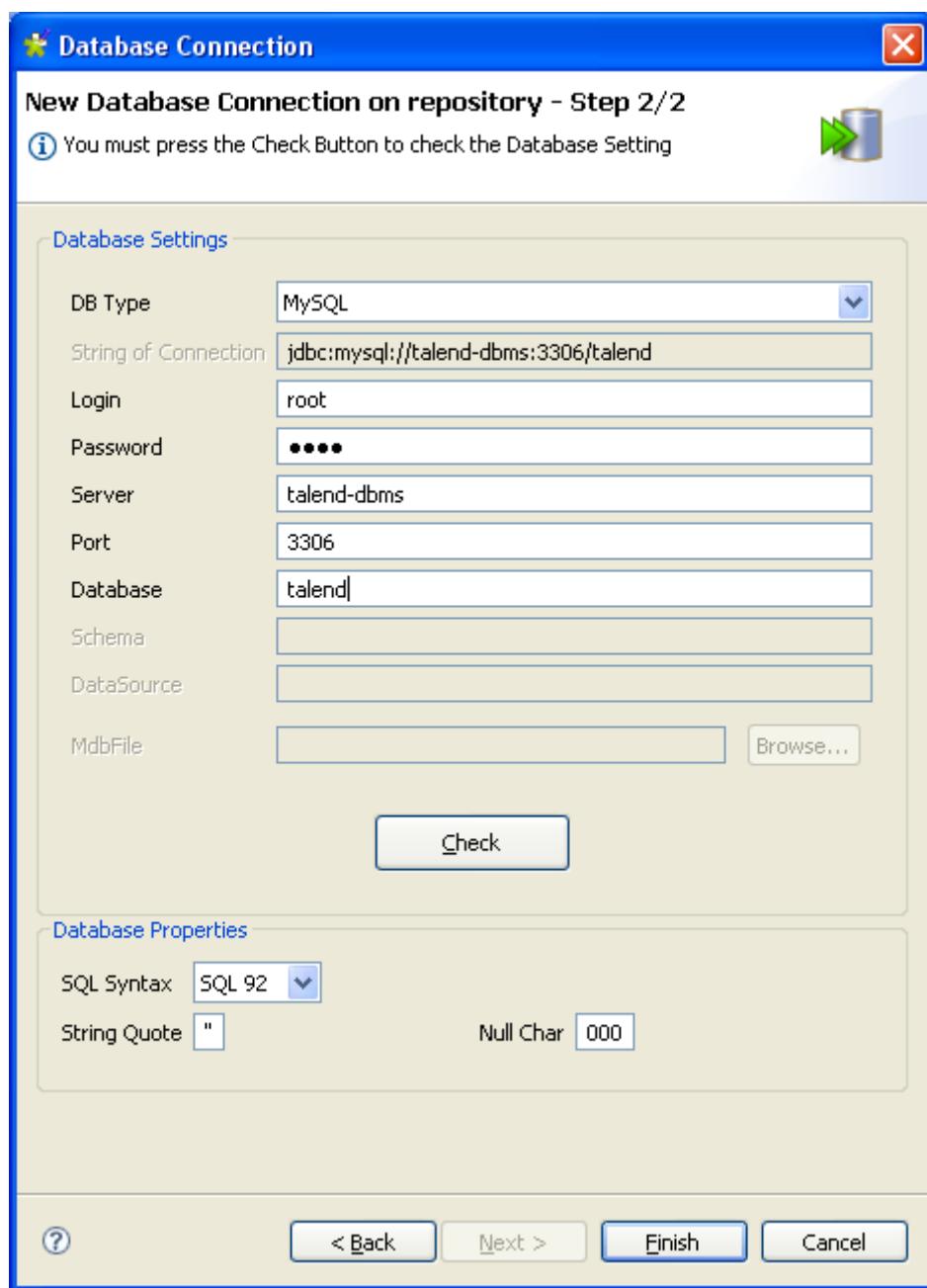
Defining Metadata items



Click **Next** when completed, the second step requires you to fill in DB connection data.

Step 2: connection

Select the type of Database you want to connect to and some fields will be greyed out or enabled according to the DB connection detail requirements.



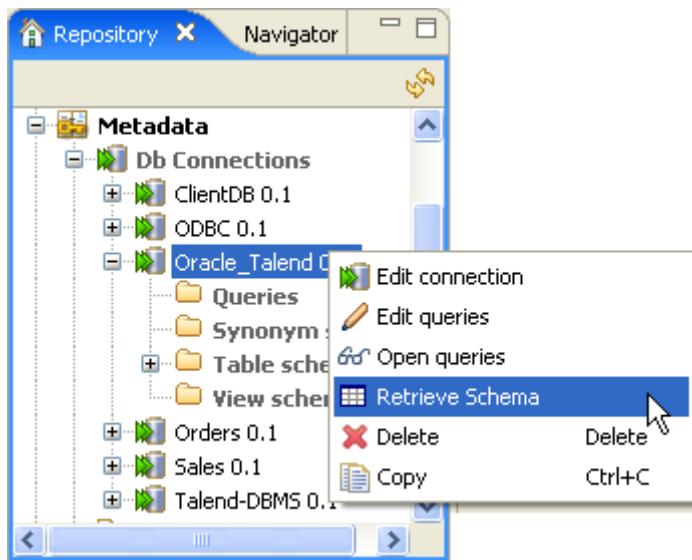
Fill in the connection details and, check your connection by clicking on **Check**.

Fill in if need be, the database properties information. That's all for the first operation on DB connection setup, click **Finish** to validate.

The newly created DB connection is now available in the Repository and displays four folders including **Queries** for SQL queries you save and **Table schemas** that will gather all schema linked to this DB connection.

Designing a Job Design

Defining Metadata items



Now right-click on the newly created connection, and select **Retrieve schema** on the pop-up menu.

Step 3: table upload

A new wizard opens up on the first step window. The List offers all tables present on the DB connection. It can be any type of DBs.

Select one or more tables on the list, to load them on your Repository filesystem. You will base your repository schemas on these tables.

If no schema is visible on the list, click **Check connection**, to verify the DB connection status.

Click **Next**. On the new window, four setting panels help you define the schemas to create.

In Java, make sure the data type is correctly defined. For more information regarding data types, including date pattern, check out <http://java.sun.com/j2se/1.5.0/docs/api/index.html>.

Step 4: schema definition

By default, the schema displayed on the Schema panel is based on the first table selected in the list of schemas loaded (left panel). You can change the name of the schema and according to your needs, you can also customize the schema structure in the schema panel.

Indeed, the tool bar allows you add, remove or move column in your schema. And, you can load an xml schema or export the current schema as xml.

To retrieve a schema based on one of the loaded table schemas, select the DB table schema name in the drop-down list and click on **Retrieve schema**. Note that the retrieved schema then overwrites any current schema and doesn't retain any custom edits.

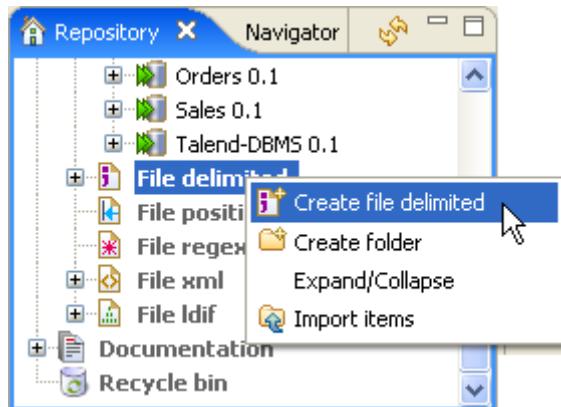
Click **Finish** to complete the DB schemas creation. All created schemas display under the relevant DB connection node.

Setting up a File Delimited schema

File delimited metadata can be used for both InputFileDelimited and InputFileCSV design components as both csv and delimited files are based on the same structure.

WARNING—*The File schema creation is very similar for all types of File connections: Delimited, Positional, Regex, Xml, or Ldif.*

On the Repository, right-click on File Delimited tree node, and select **Create file delimited** on the pop-up menu.



Unlike the DB connection wizard, the Delimited File wizard gathers both file connection and schema definition in a four-step procedure.

Step 1: general properties

On the first step, fill in the schema generic information, such as Schema **Name** and **Description**.

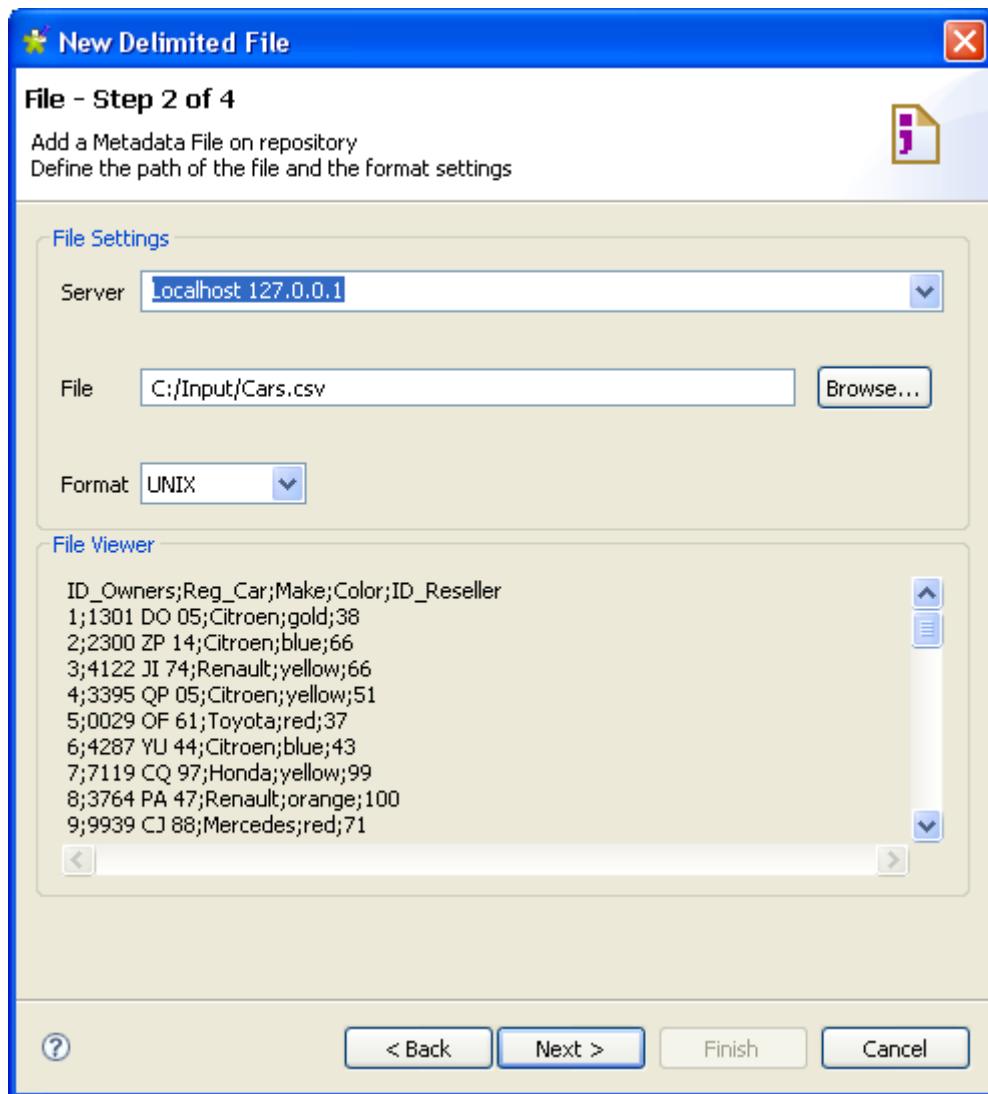
For further information, see *Step 1: general properties on page 55*.

Step 2: file upload

Define the **Server** IP address where the file is stored. And click **Browse...** to set the **File path**.

Designing a Job Design

Defining Metadata items



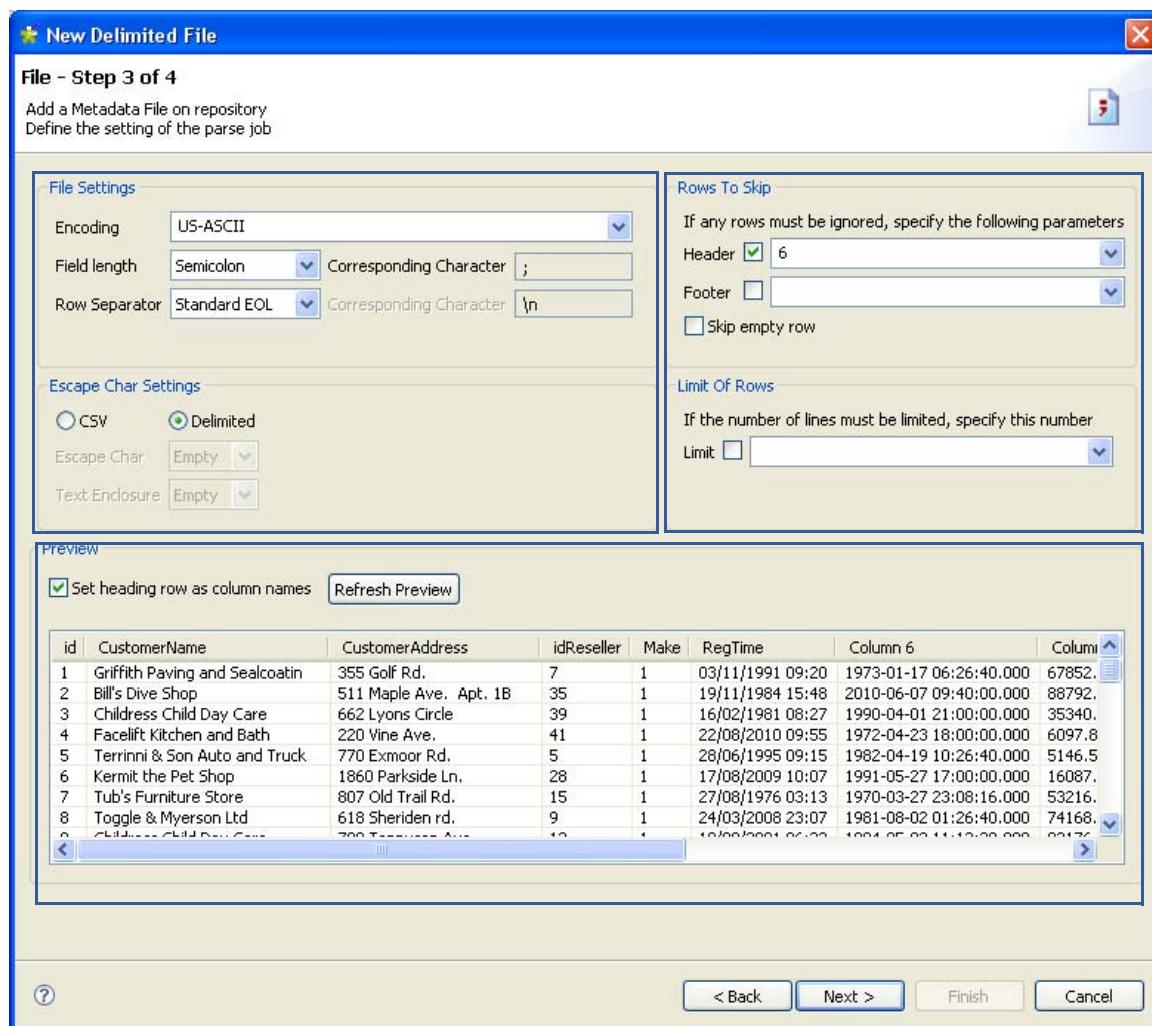
Select the OS **Format** the file was created in. This information is used to prefill subsequent step fields. If the list doesn't include the appropriate format, ignore it.

The **File viewer** gives an instant picture of the file loaded. It allows you to check the file consistency, the presence of header and more generally the file structure.

Click **Next** to Step3.

Step 3: schema definition

In this view, you can refine your data description and file settings. Click on the squares below, to zoom in and get more information.



Set the **Encoding**, as well as Field and Row separators in the Delimited File Settings.

Designing a Job Design

Defining Metadata items

File Settings

Encoding: US-ASCII

Field length: Semicolon Corresponding Character: ;

Row Separator: Standard EOL Corresponding Character: \n

Escape Char Settings

CSV Delimited

Escape Char: Empty

Text Enclosure: Empty

Depending on your file type (csv or delimited), you can also set the Escape and Enclosure characters to be used.

If the file preview shows a header message, you can exclude the header from the parsing. Set the number of header rows to be skipped. Also, if you know that the file contains footer information, set the number of footer lines to be ignored.

Rows To Skip

If any rows must be ignored, specify the following parameters

Header 3

Footer 0

Skip empty row

Limit Of Rows

If the number of lines must be limited, specify this number

Limit 0

The **Limit of rows** allows you to restrict the extend of the file being parsed.

In the File Preview panel, you can view the new settings impact.

Check the **Set heading row as column names** box to transform the first parsed row as labels for schema columns. Note that the number of header rows to be skipped is then incremented of 1.

Preview

Set heading row as column names Refresh Preview

ID	Registration	Make	Color	Reseller ID	Name	Insurance
1	5776 ZQ 94	Volkswagen	gold	7	montmont	KWW2844
3	2580 TT 77	Renault	orange	1	bouhnan	BNU9147
4	1722 WE 11	Citroen	silver	10	kieth	TCU0260

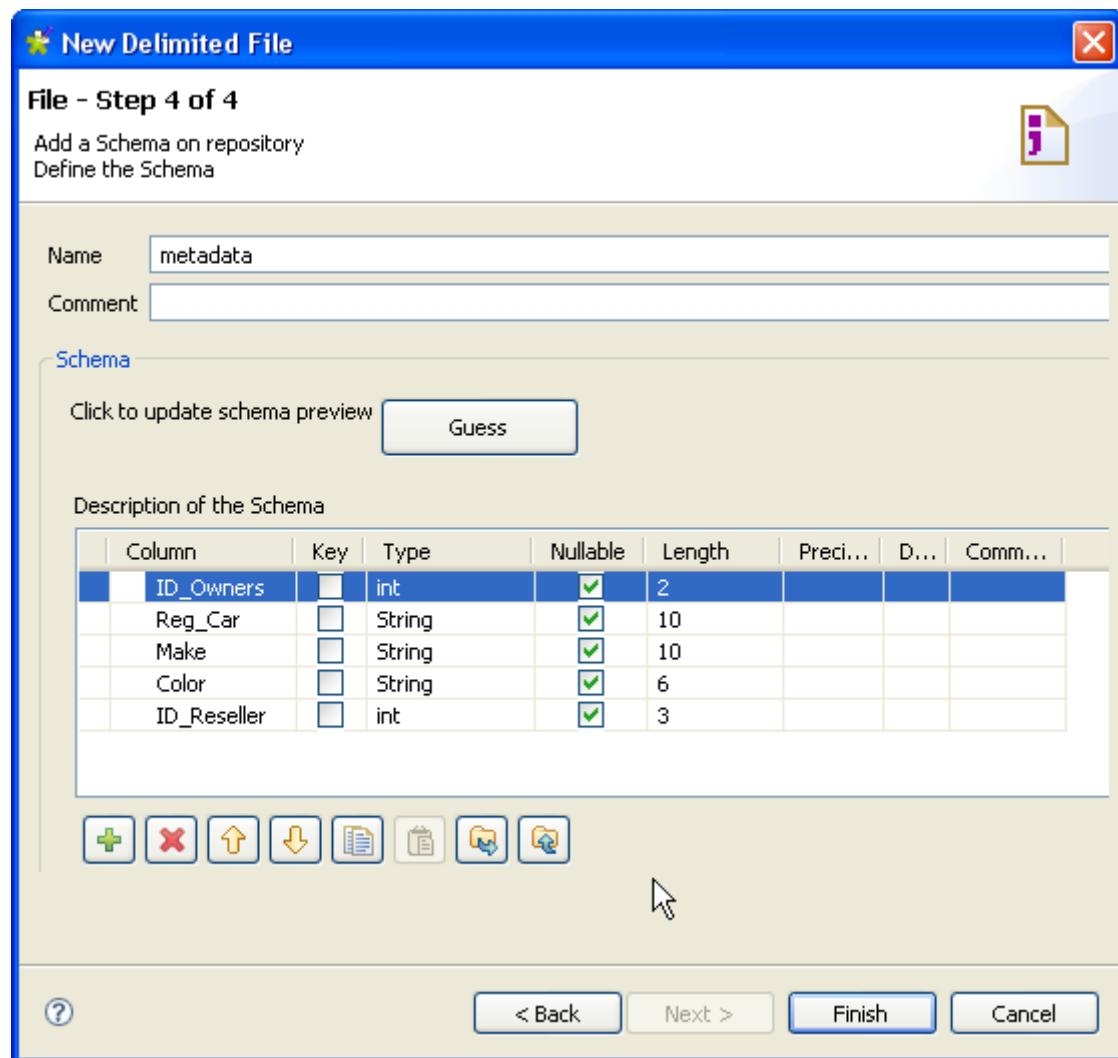
Click **Refresh** on the preview panel for the settings to take effect and view the result on the viewer.

Step 4: final schema

The last step shows the Delimited File schema generated. You can customize the schema using the toolbar underneath the table.

Designing a Job Design

Defining Metadata items

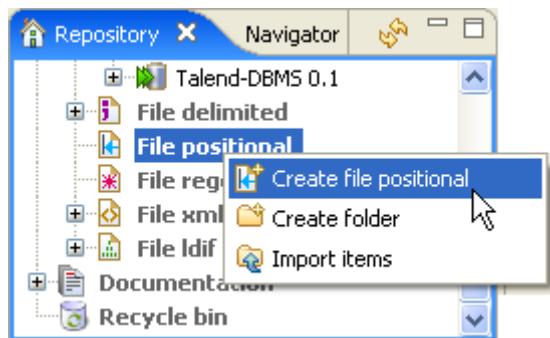


If the Delimited file which the schema is based on is changed, use the **Guess** button to generate again the schema. Note that if you customized the schema, the Guess feature doesn't retain these changes.

Click Finish. The new schema displays on the Repository, under the relevant File Delimited connection node.

Setting up a File Positional schema

On the Repository, right-click on File Delimited tree node, and select **Create file positional** on the pop-up menu.



Proceed the same way as for the file delimited connection. Right-click on Metadata on the Repository and select **Create file positional**.

Step 1: general properties

Fill in the schema generic information, such as Schema **Name** and **Description**.

Step 2: connection and file upload

Then define the positional file connection settings, by filling in the Server IP address and the File path fields.

Like for Delimited File schema creation, the format is requested for prefill purpose of next step fields. If the file creation OS format is not offered in the list, ignore this field.

New Positional File

File - Step 2 of 4

Add a Metadata File on repository
Define the path of the file and the format settings

File Location Settings

Server: Localhost 127.0.0.1

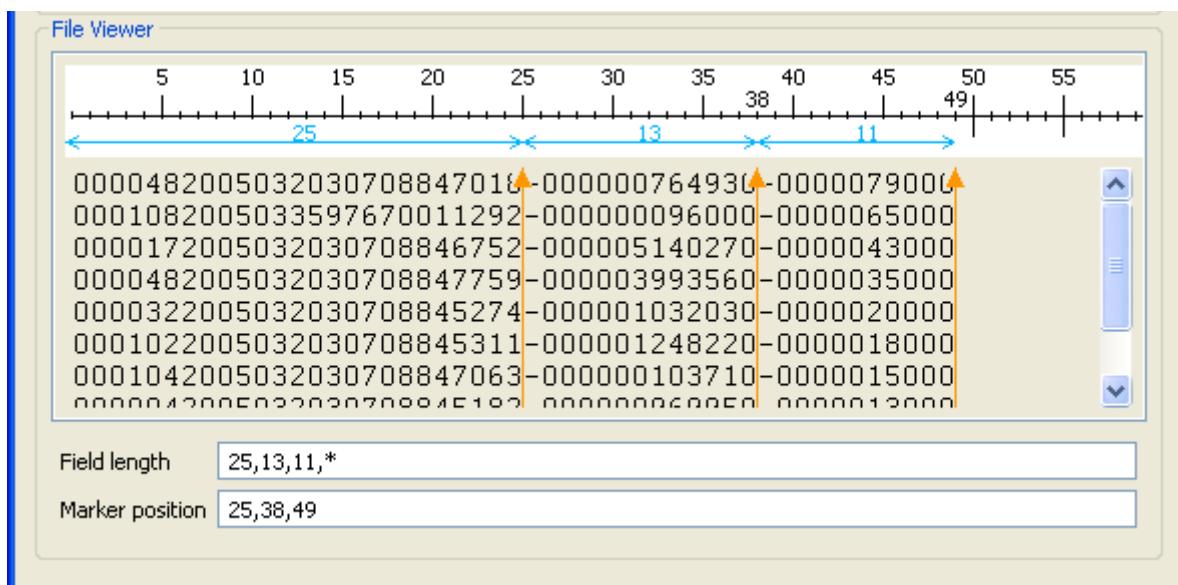
File: C:/Input/raw.head

Format: UNIX

The file viewer shows a file preview and allows you to place your position markers.

Designing a Job Design

Defining Metadata items



Click on the file preview and set the marker against the ruler. The orange arrow helps you refine the position.

The **Field length** lists a series of figures separated by commas, these are the number of characters between markers. The asterisk symbol means all remaining characters on the row, from the preceding marker position.

The **Marker Position** shows the exact position of the marker on the ruler. You can change it to refine the position.

You can add as many markers as needed. To remove a marker, drag it towards the ruler.

Click **Next** to continue.

Step 3: schema refining

The next step opens the schema setting window. As for the Delimited File schema, you can refine the schema definition by precising the field and row separators, the header message number of lines and else...

At this stage, the preview shows the file delimited upon the markers' position. If the file contains column labels, check the box **Set heading row as column names**.

Step 4: final schema

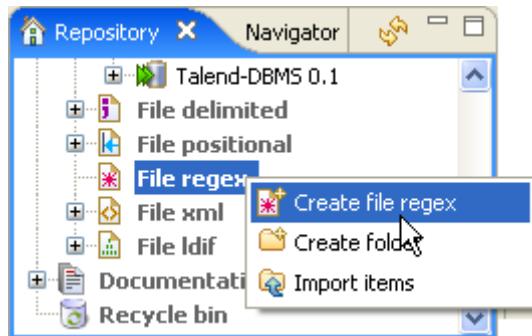
Step 4 shows the final generated schema. Note that any characters which could be misinterpreted by the program are replaced by neutral characters, like underscores replace asterisks.

You can add a customized name (by default, metadata) and make edits to it using the tool bar.

You can also retrieve or update the Positional File schema by clicking on **Guess**. Note however that any edits to the schema might be lost after “guessing” the file-based schema.

Setting up a File Regex schema

Regex file schemas are used for files containing redundant information, such as log files.



Proceed the same way as for the file delimited or positional connection. Right-click on Metadata on the Repository and select **Create file regex**.

Step 1: general properties

Fill in the schema generic information, such as Schema **Name** and **Description**.

Step 2: file upload

Then define the Regex file connection settings, by filling in the Server IP address and the File path fields.

File Settings	
Server	localhost 127.0.0.1
File	C:\Input\Eclipse.log
Format	UNIX

File Viewer	
<pre>***** Test ***** ***** Test ***** !ENTRY org.talend.designer.runprocess 1 0 2006-09-07 10:38:55.050 !ENTRY org.talend.designer.runprocess 1 0 2006-09-07 10:39:25.537 !ENTRY org.talend.designer.runprocess 1 0 2006-09-07 10:39:28.430 !ENTRY org.talend.designer.runprocess 1 0 2006-09-07 10:39:51.377</pre>	

Like for Delimited File schema creation, the format is requested for prefill purpose of next step fields. If the file creation OS format is not offered in the list, ignore this field.

Designing a Job Design

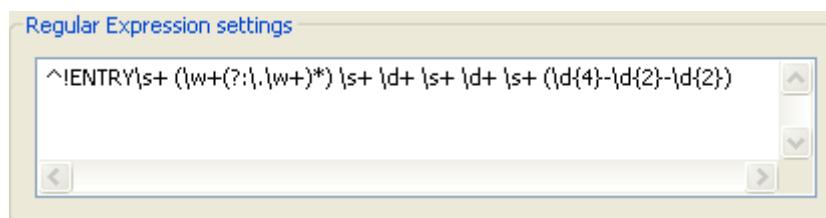
Defining Metadata items

The file viewer gives an instant picture of the loaded file. Click **Next** to define the schema structure.

Step 3: schema definition

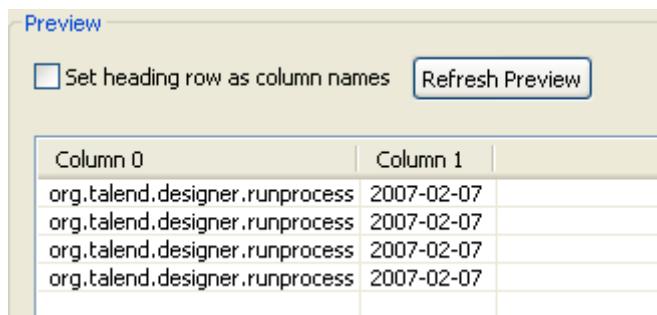
This step opens the schema setting window. As for the other File schemas, you can refine the schema definition by precising the field and row separators, the header message number of lines and else...

In the **Regular Expression settings** panel, enter the regular expression to be used to delimit the file.



In Java: Regular expressions syntax is different in Java/Perl. Make sure you use the relevant syntax according to the generation language used.

Then click **Refresh preview** to take into account the changes. The button changes to **Wait** until the preview is refreshed.



Click next when setting is complete. The last step generates the Regex File schema.

Step 4: final schema

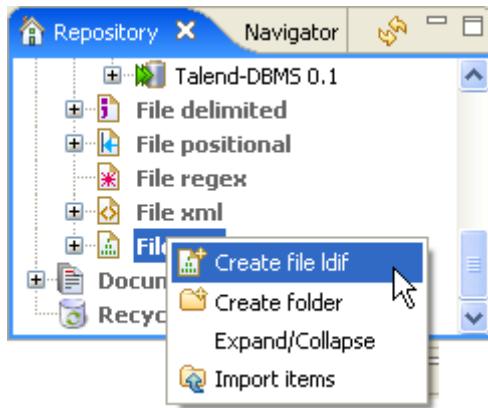
You can add a customized name (by default, metadata) and make edits to it using the tool bar.

You can also retrieve or update the Regex File schema by clicking on **Guess**. Note however that any edits to the schema might be lost after guessing the file based schema.

Click **Finish**. The new schema displays on the Repository, under the relevant File regex connection node.

Setting up a FileLDIF schema

LDIF files are directory files described by attributes. FileLDIF metadata centralize these LDIF type files and their attribute description.



Proceed the same way as for other file connections. Right-click on Metadata on the Repository and select **Create file Ldif**.

Note: Make sure that you installed the relevant Perl module as described in the Installation guide. For more info, check out <http://talendforge.org/wiki/doku.php>

Step 1: general properties

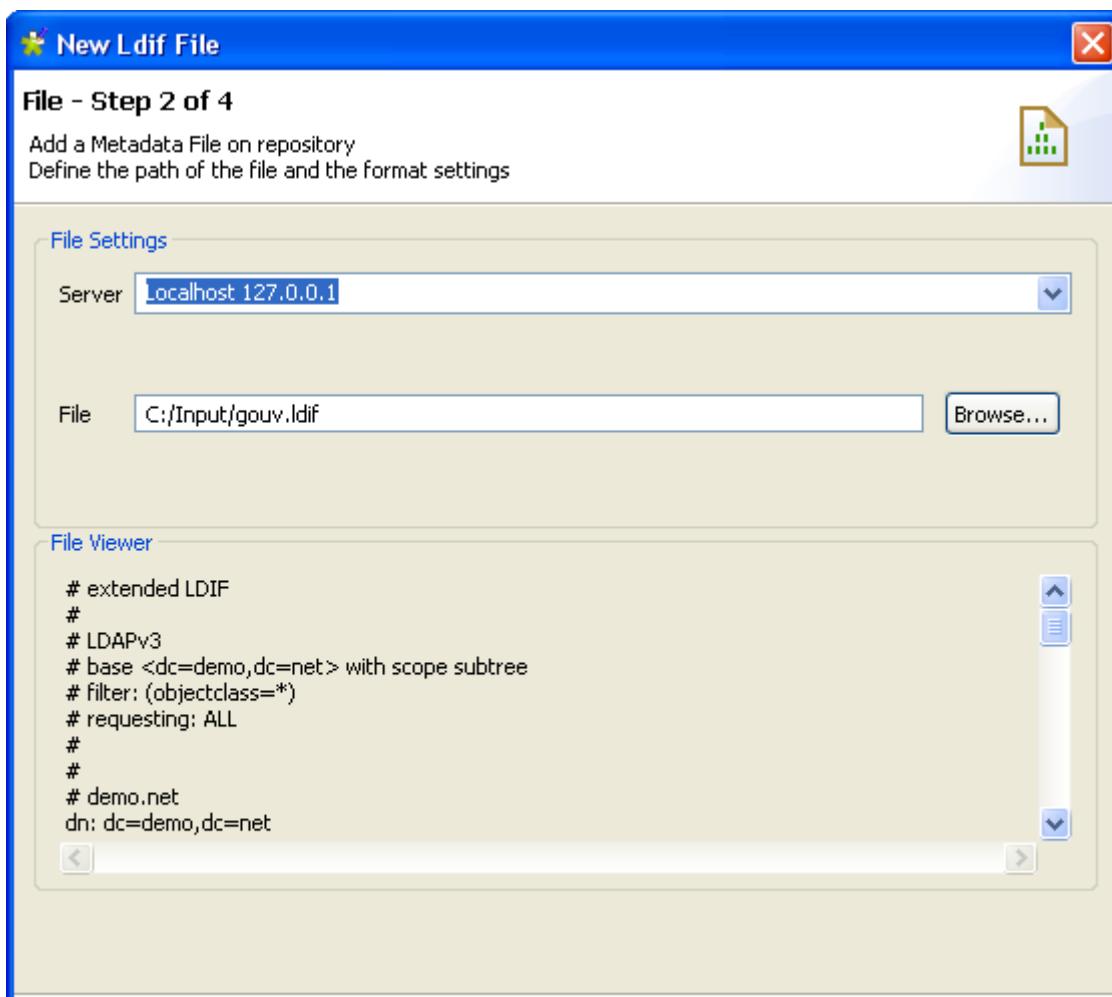
On the first step, fill in the schema generic information, such as Schema name and description.

Step 2: file upload

Then define the Ldif file connection settings, by filling in the **File path** field.

Designing a Job Design

Defining Metadata items

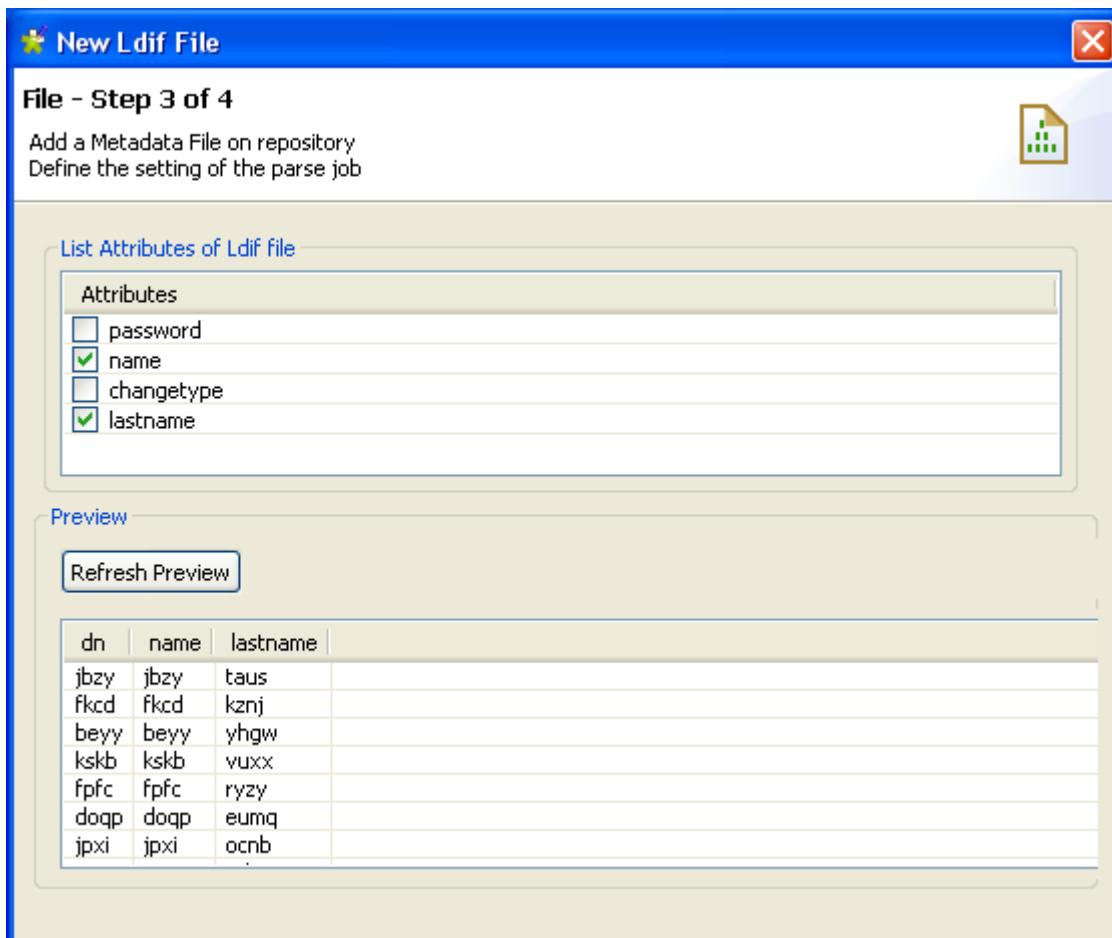


Note: The connection functionality to a remote server is not in operation yet for LDIF file collection.

The File viewer provides a preview of the file's first 50 rows.

Step 3: schema definition

The list of attributes of the description file displays on the top of the panel. Select the attributes you want to extract from the LDIF description file, by checking the relevant boxes.



Click **Refresh Preview** to include the selected attributes into the file preview.

Note: DN is omitted in the list of attributes as this key attribute is automatically included in the file preview hence in the generated schema.

Step 4: final schema

The schema generated shows the columns of the description file. You can customize it upon your needs and reload the original schema using the **Guess** button.

Click **Finish**. The new schema displays on the Repository, under the relevant File LDif connection node.

Setting up a FileXML schema

Centralize your XPath query statements over a defined XML file and gather the values fetched from it.

Proceed the same way as for other file connections. Right-click on Metadata on the Repository and select **Create file XML**.

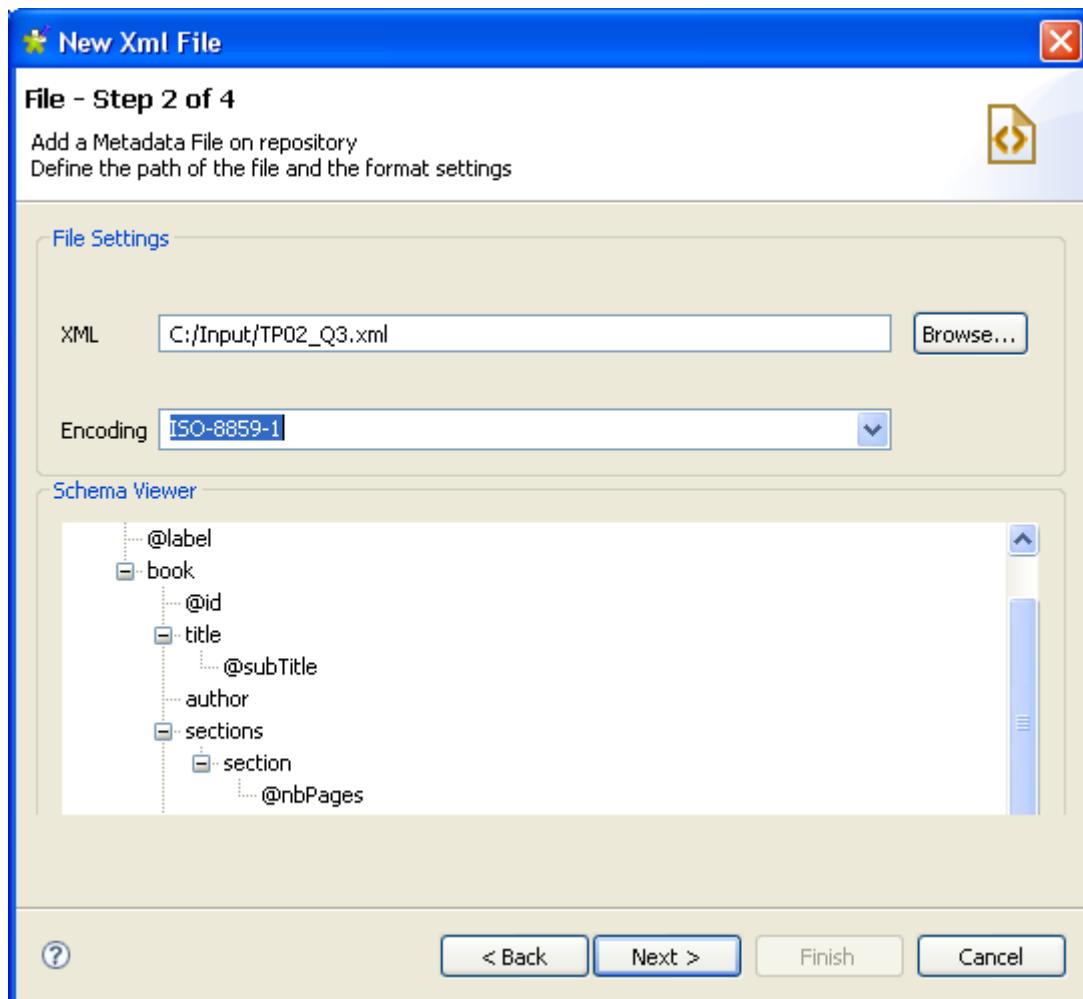
Step 1: general properties

On the first step, fill in the schema generic information, such as Schema name and description. Click **Next** when you're complete.

Step 2: file upload

Browse to the XML File to upload and fill in the **Encoding** if the system didn't detect it automatically.

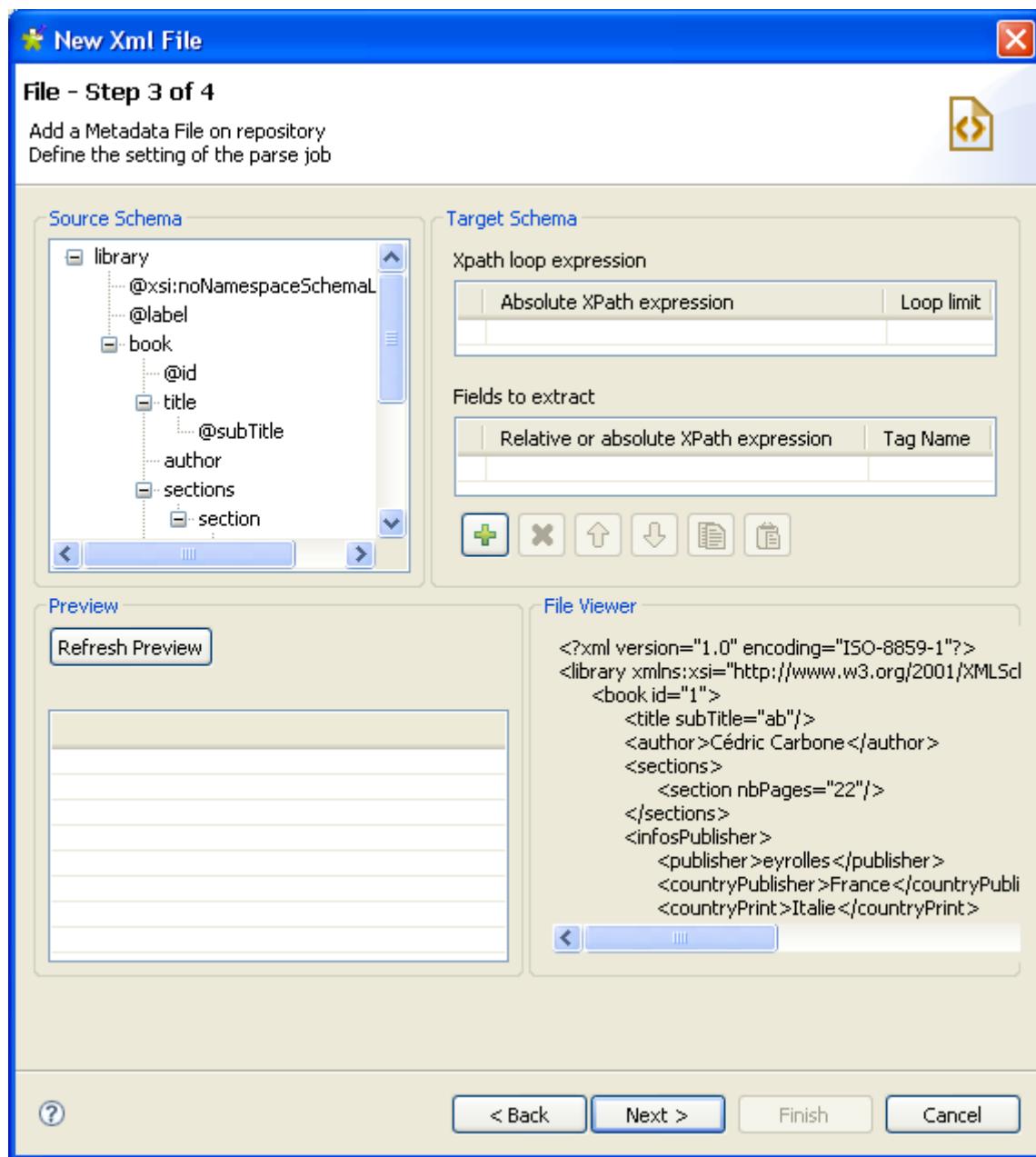
The file preview shows the XML node tree structure.



Click **Next** to the following step.

Step 3: schema definition

Set the parameters to be taken into account for the schema definition.



The schema definition window is divided into four panels:

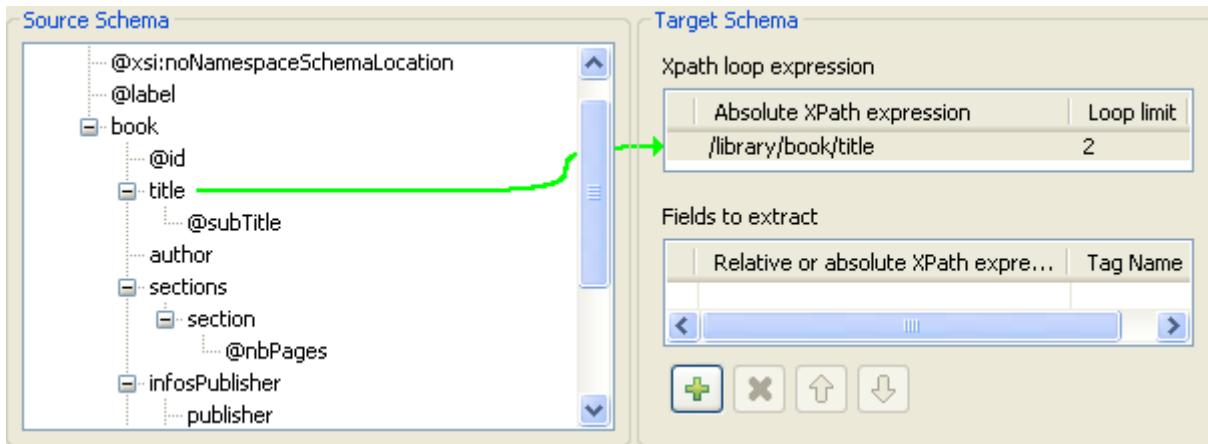
- **Source Schema:** Tree view of the uploaded XML file structure
- **Target Schema:** Extraction and iteration information
- **Preview:** Target schema preview
- **File viewer:** Raw data viewer

In the **Xpath loop expression** field, enter the absolute xpath expression leading to the structure node which the iteration should apply on. You can type in the entire expression or press

Designing a Job Design

Defining Metadata items

Ctrl+Space to get the autocompletion list.

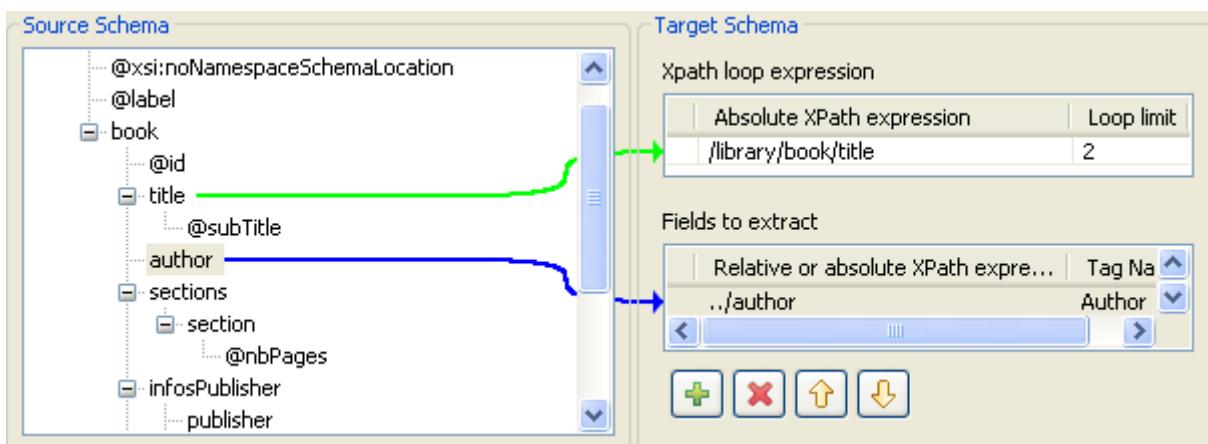


Or else, drag and drop the node from the source structure towards the target schema Xpath field.

Note: The **Xpath loop expression** field is compulsory.

You can also define a **Loop limit** to restrict the iteration to a number of nodes. A green link is then created.

Then define the fields to extract. Simply drag and drop the relevant node to the **Relative or absolute XPath expression** field.



Use the plus sign to add rows to the table and select as many fields to extract, as you need. Press the Ctrl or the Shift keys for multiple selection of grouped or ungrouped nodes, and drag & drop them to the table.

The screenshot shows the SQLBuilder interface with two main panels: 'Source Schema' and 'Target Schema'.

Source Schema: This panel displays the XML schema structure. A 'book' node is selected, indicated by a blue selection bar. Inside the 'book' node, the '@subTitle' attribute is also highlighted with a blue selection bar. Other nodes like 'library', 'author', 'sections', and 'infoPublisher' are shown in grey, indicating they are not currently selected.

Target Schema: This panel contains configuration for extracting fields from the source schema.

- Xpath loop expression:** Shows an 'Absolute XPath expression' of '/library/book' and a 'Loop limit' of 2.
- Fields to extract:** Shows two extracted fields:

Relative or absolute XPath ex...	Tag Name
author	Author
title/@subTitle	Title
- Buttons:** Includes standard icons for adding (+), removing (-), and reordering (up and down arrows).

In the **Tag name** field, give a name the column header that will display on the schema preview (bottom left panel).

The selected link is blue, and all other links are grey. You can prioritize the order of fields to extract using the up and down arrows.

Click **Refresh preview** to display the schema preview. The fields will then be displayed in the schema preview in the order given (top field on the left).

The screenshot shows the 'Preview' and 'File Viewer' panels.

Preview: This panel shows a table with two columns: 'Author' and 'Title'. The data is:

Author	Title
Cédric Carbone	My Life
Chris Antoine	My Career

File Viewer: This panel displays the XML code corresponding to the schema preview:


```
<?xml version="1.0" encoding="ISO-8859-1"?>
<library xmlns:xsi="http://www.w3.org/2001/XMLSchema-instar
<book id="1">
  <title subTitle="ab"/>
  <author>Cédric Carbone</author>
  <sections>
    <section nbPages="22"/>
  </sections>
  <infoPublisher>
    <publisher>eyrolles</publisher>
    <countryPublisher>France</countryPublisher>
  </infoPublisher>
```

Step 4: final schema

The schema generated shows the selected columns of the XML file. You can customize it upon your needs or reload the original schema using the **Guess** button.

Click **Finish**. The new schema displays on the Repository, under the relevant File XML connection node.

Creating queries using SQLBuilder

SQLBuilder helps you build your SQL queries and monitor the changes between DB tables and metadata tables. This editor is available in all DBInput and DBSQLRow components (specific or generic).

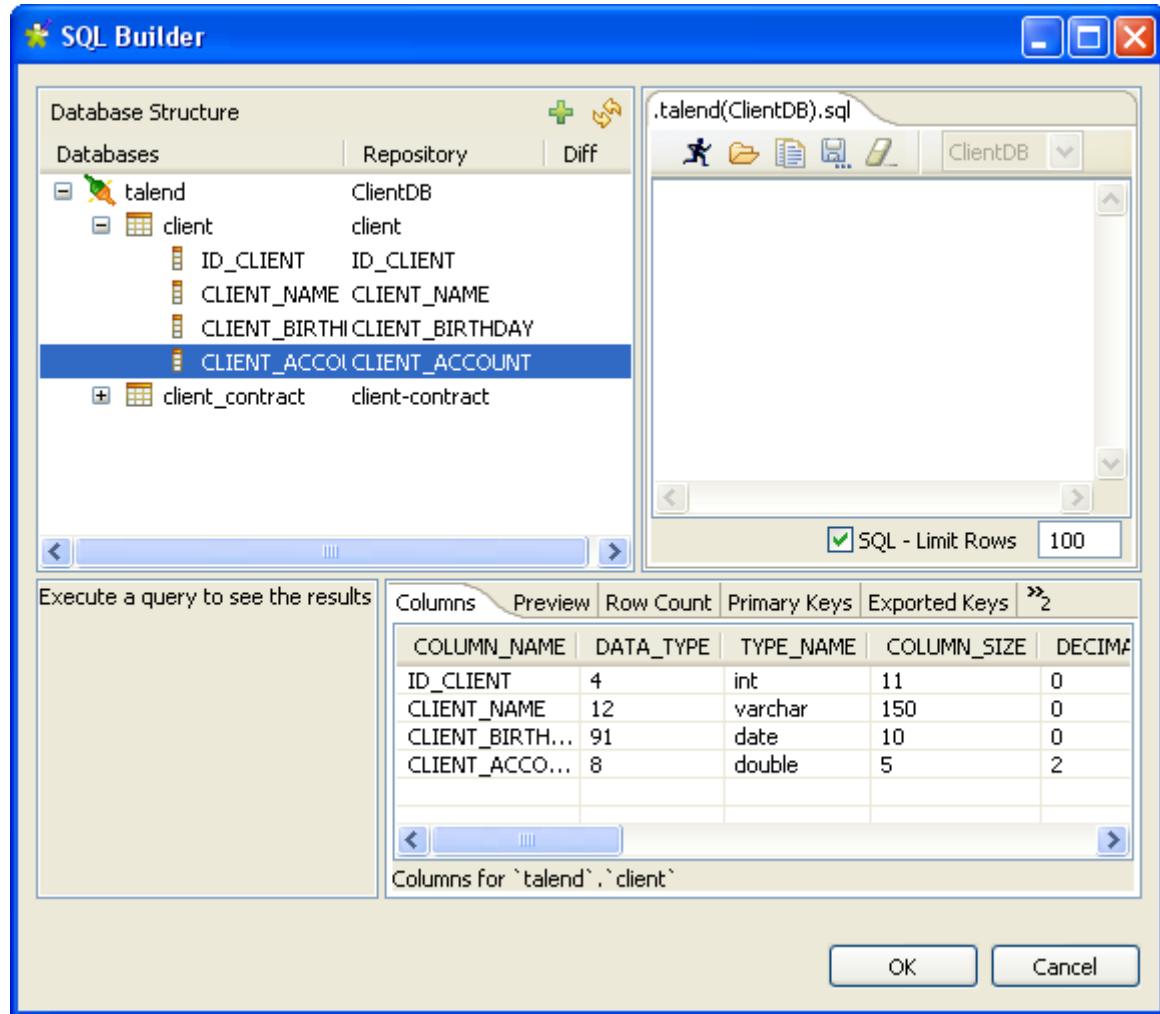
Designing a Job Design

Creating queries using SQLBuilder

You can build a query using SQLbuilder whether your database table schema is stored in the repository or built-in directly in the job component.

Fill in the DB connection details and select the appropriate repository entry if you defined it.

Remove the default query statement in the **Query** field of the component **Properties** panel. Then click on the three-dot button to open the SQL Builder.



The SQL Builder editor is made of the following panels:

- Database structure
- Query editor made of editor and designer tabs
- Query execution view
- Schema view

The Database structure shows the tables for which a schema was defined either in the repository database entry or in your built-in connection.

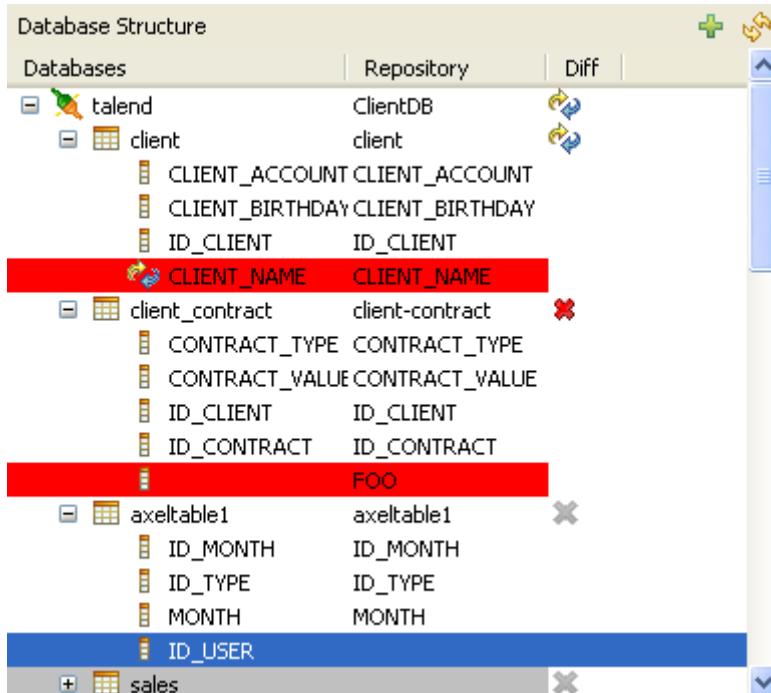
The schema view, in the bottom right corner of the editor, shows the column description.

Database structure comparison

On the database structure panel, are shown all the tables stored in the DB connection metadata entry (Repository) or in case of built-in schema, the tables of the database itself.

Note: the connection to the database, in case of built-in schema or in case of refreshing operation of a repository schema, might take quite some time.

Click the refresh icon to display the differences found between the DB metadata tables and the actual DB tables.



The **Diff** icons point out that the table contains differences or gaps. Develop the table node to show the exact column containing the differences.

The red highlight shows that the content of the column contains differences or that the column is missing from the actual database table.

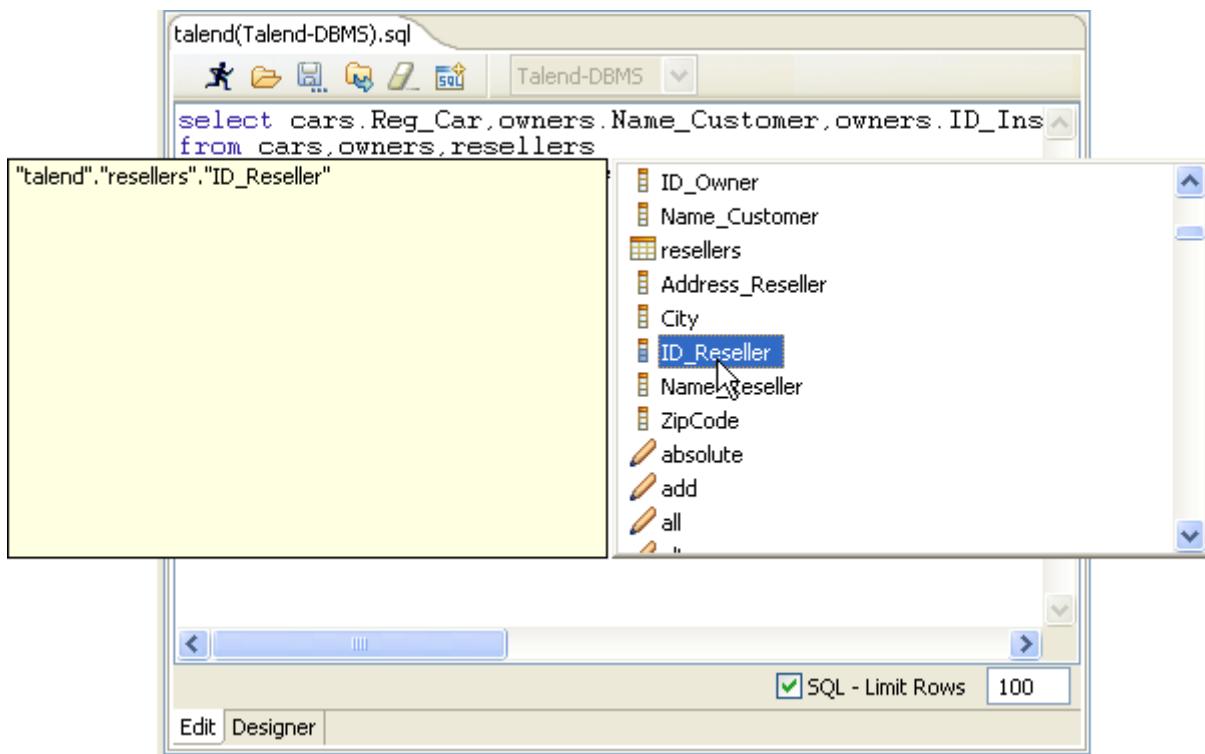
The blue highlight shows that the column is missing from the table stored in the repository metadata.

Building a query

The Query editor is a multiple-tab system allowing you to write or graphically design as many queries as you want.

To create a new query, right-click on the table or on the table column and select **Generate Select Statement** on the pop-up list.

Click on the empty tab showing by default and type in your SQL query or press **Ctrl+Space** to access the autocompletion list. The tooltip bubble shows the whole path to the table or table section you want to search in.



Alternatively the graphical query **designer** allows you to handle tables easily and have real-time generation of the corresponding query in the **edit** tab.

Click on **Designer** tab to switch from manual **Edit** mode to graphical mode.

Note: You may get a message while switching from one view to the other, as some SQL statements cannot be interpreted graphically.

If you selected a table, all columns are selected by default. Uncheck the box facing the relevant columns to exclude them from the selection.

Add more tables in a simple right-click. On the **designer** tab, right-click and select **Add tables** in the pop-up list then select the relevant table to be added.

If joins between these tables already exist, these joins are automatically set up graphically in the editor.

You can also very easily create a join between tables. Right-click on the first table columns to be linked and select **Equal** on the pop-up list, to join it with the relevant field of the second table.

```

select owners.ID_Owner, owners.Name_Customer, owners.ID_Insurance, cars.ID_Owners, cars.Reg_Car, cars.Make, cars.Color, cars.ID_Reseller
from owners, cars, resellers
where owners.ID_Owner=cars.ID_Owners and cars.ID_Reseller=resellers.ID_Reseller
  
```

The SQL statement corresponding to your graphical handlings is also displayed on the viewer part of the editor or click on the **Edit** tab to switch back to manual **Edit** mode.

Note: In **Designer** mode, you cannot include graphically filter criteria. These need to be added in **Edit** mode.

Once your query is complete, execute it by clicking on the running man button.

The toolbar above the query editor allows you to access quickly usual commands such as: execute, open, save and clear.

On the Query results view, are displayed the results of the active tab's query.

The status bar at the bottom of this panel provides information about execution time and number of rows retrieved.

Storing a query in the Repository

To be able to retrieve and reuse queries, we recommend you to store them in the **Repository**.

In the SQL Builder editor, click on **Save** (floppy disk icon on the tool bar) to bind the query with the DB connection and schema in case these are also stored in the **Repository**.

The query can then be accessed from the **Database structure** view, on the left-hand side of the editor.

Designing a Job Design

Mapping data flows in a job

Mapping data flows in a job

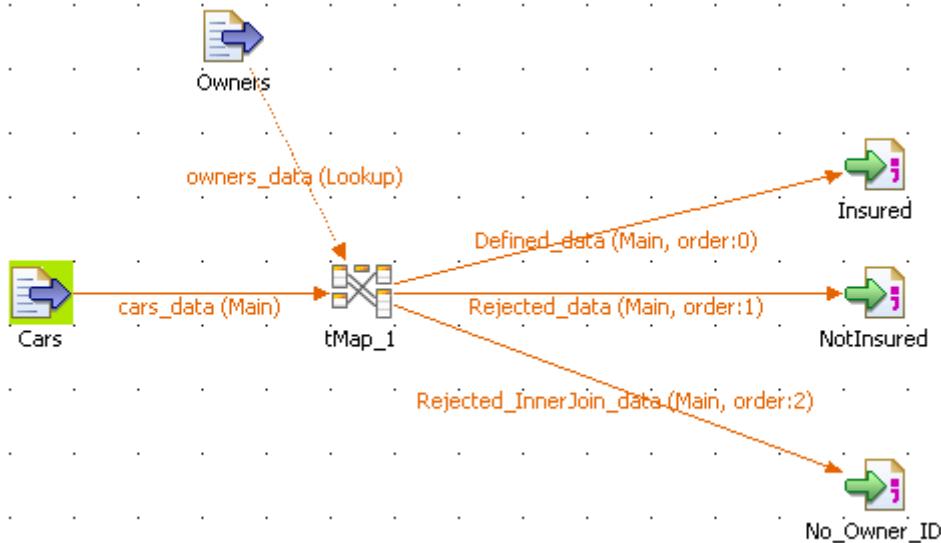
For the time being, the way to handle multiple input and output flows including transformations and data re-routing is to use the tMap component. The following section gives details about the usage principles of this component, for further information or scenarios and use cases, see *tMap on page 231*.

tMap operation overview

tMap allows the following types of operations:

- data multiplexing and demultiplexing
- data transformation on any type of fields
- fields concatenation and interchange
- field filtering using constraints
- data rejecting

As all these operations of transformation and/or routing are carried out by tMap, this component cannot be a Start or End component in the job design.



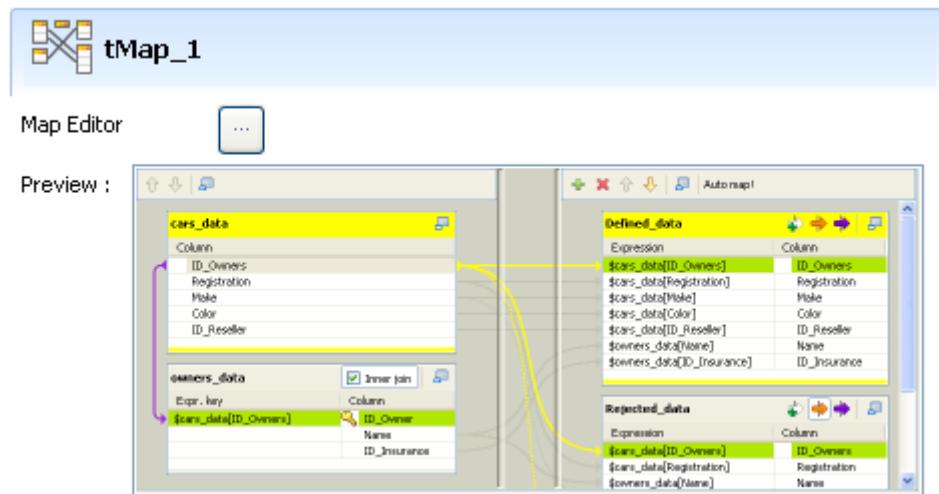
tMap uses incoming connections to pre-fill input schemas with data in the Mapper. Therefore, you cannot create new input schemas straight in the Mapper. Instead, you need to implement as many **Row** connections incoming to tMap component as required, in order to create as many input schemas as needed.

The same way, create as many output row connections as required. However, you can fill in the output with content straight from the Mapper through a convenient graphical editor.

Note that there can be only one **Main** incoming rows. All other incoming rows are of **Lookup** type.
Related topic: *Row connection on page 45*

Lookup rows are incoming connections from secondary (or reference) flows of data. These reference data might depend directly or indirectly on the primary flow. This dependency relationship is translated with an graphical mapping and the creation of an expression key.

Although the mapper requires the connections to be implemented in order to define Input and Output flows, you also need to create the actual mapping in order for the Preview to be available in the Properties panel of the workspace.



Double-click the tMap icon or click on the Map editor three-dot button to open the Mapping editor in a new window.

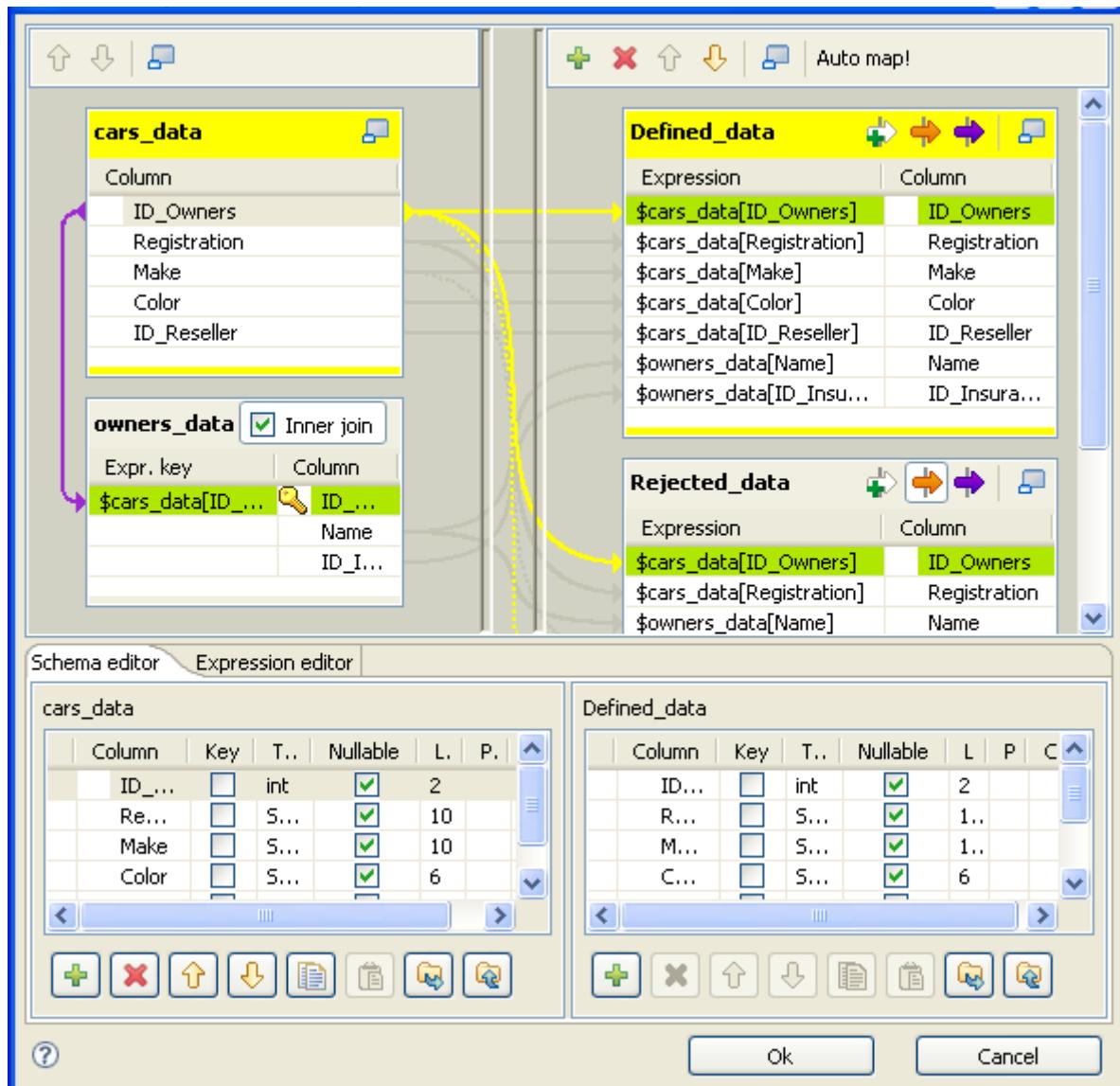
tMap interface

tMap is an advanced component which requires more information than other components. Indeed, the **Mapper** is an “all-in-one” tool allowing you to define all parameters needed to map, transform and route your data flows via a convenient graphical interface.

For all tables and the Mapper window, you can minimize and restore the window using the window icon.

Designing a Job Design

Mapping data flows in a job



The Mapper is made of several panels:

- The **Input panel** is the top left panel on the window. It offers a graphical representation of all (main and lookup) incoming data flows. The data are gathered in various columns of input tables. Note that the table name reflects the main or lookup row from the job design on the workspace.
- The **Variable panel** is the central panel on the Mapper window. It allows the centralization of redundant information through the mapping to variable and allows you to carry out transformations.
- The **Output panel** is the top right panel on the window. It allows mapping data and fields from Input tables and Variables to the appropriate Output rows.
- Both bottom panels are the Input and Output schemas description. The **Schema editor** tab offers a schema view of all columns of input and output tables in selection in their respective

panel.

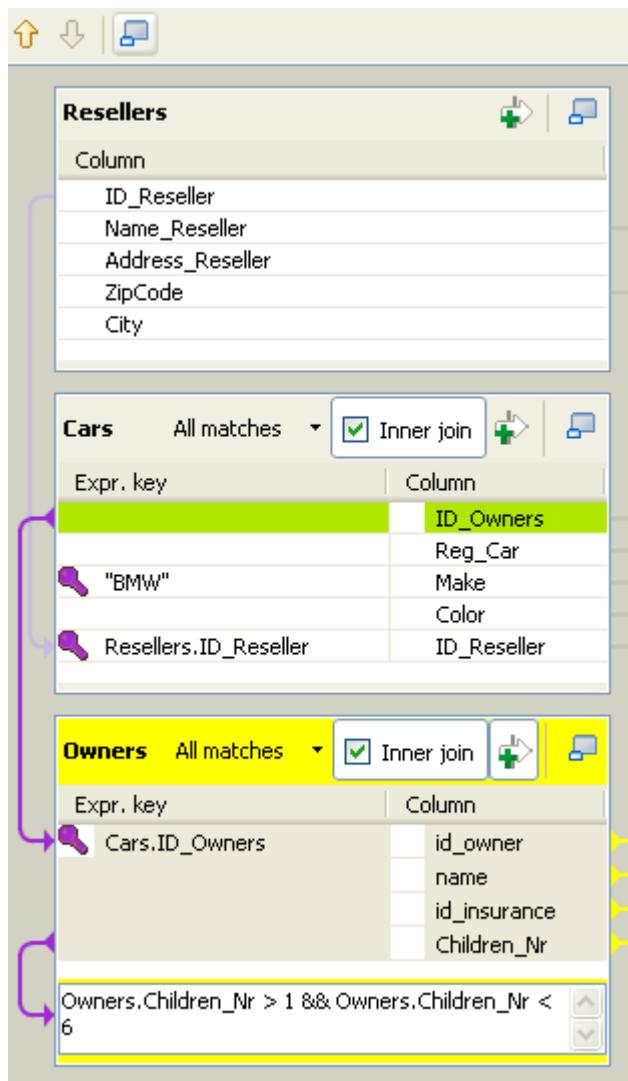
- **Expression editor** is the edition tool for all expression keys of Input/Output data, variable expressions or filtering conditions.

The name of Input/Output tables in the mapping editor reflects the name of the incoming and outgoing flows (row connections).

Setting the input flow in the Mapper

The order of the **Input** tables is essential. The top table reflects the **Main** flow connection, and for this reason, is given priority for reading and processing through the **tMap** component.

For this priority reason, you are not allowed to move up or down the **Main** flow table. This ensures that no Join can be lost.



Although you can use the up and down arrows to interchange **Lookup** tables order, be aware that the **Joins** between two lookup tables may then be lost.

Related topic: *Explicit Join on page 84*.

Filling in Input tables with a schema

To fill in input tables, you need to define first the schemas of all input components connected to the **tMap** component on your Job Designer.

Main and Lookup table content

The order of the **Input** tables is essential.

The **Main Row** connection determines the **Main** flow table content. This input flow is reflected in the first table of the Mapper's Input panel.

The **Lookup** connections' content fills in all other (secondary or subordinate) tables which displays below the **Main** flow table. If you haven't define the schema of an input component yet, the input table displays as empty in the Input area.

The key is also retrieved from the schema defined in the Input component. This **Key** corresponds to the key defined in the input schema where relevant. It has to be distinguished from the hash key that is internally used in the Mapper, which displays in a different color.

Designing a Job Design

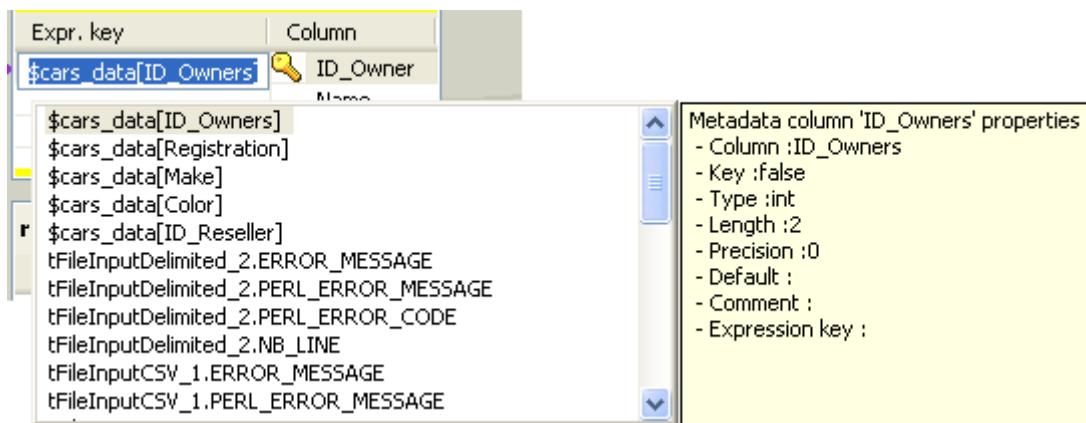
Mapping data flows in a job

Variables

You can use global or context variables or reuse the variable defined in the **Variables** zone.

Press **Ctrl+Space bar** to access the list of variables. This list gathers together global, context and mapping variables.

The list of variables changes according to the context and grows along new variable creation. Only valid mappable variables in the context show on the list.



Docked at the **Variable** list, a metadata tip box display to provide information about the selected column.

Related topic: *Mapping variables on page 88*

Explicit Join

In fact, **Joins** let you select data from a table depending upon the data from another table. In the Mapper context, the data of a **Main** table and of a **Lookup** table can be bound together on **expression keys**. In this case, the order of table does fully make sense.

Simply drag and drop column names from one table to a subordinate one, to create a **Join** relationship between the two tables. This way, you can retrieve and process data from multiple inputs.

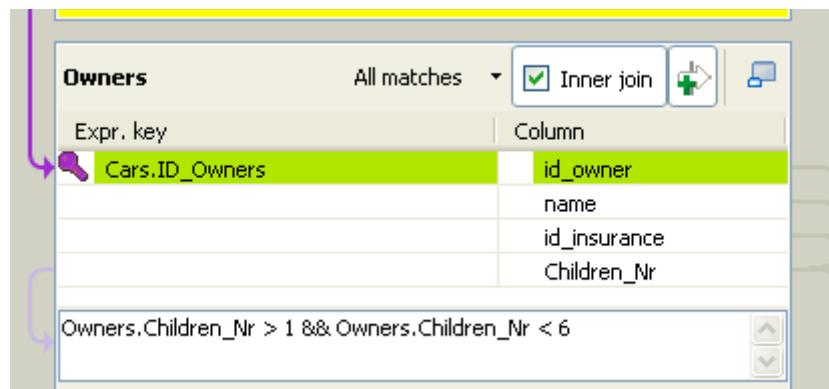
The join displays graphically as a violet link and creates automatically a key that will be used as a hash key to speed up the match search.

You can create direct joins between the main table and lookup tables. But you can also create indirect joins from the main table to a lookup table, via another lookup table. This requires a direct join between one of the **Lookup** table to the **Main** one.

Note: You cannot create a **Join** from a subordinate table towards a superior table in the Input area.

The **Expression key** field which is filled in with the dragged and dropped data is editable in the input schema or in the **Schema editor** panel, whereas the column name can only be changed from the **Schema editor** panel.

You can either insert the dragged data into a new entry or replace the existing entries or else concatenate all selected data into one cell.



For further information about possible types of drag & drops, see *Output setting on page 89*.

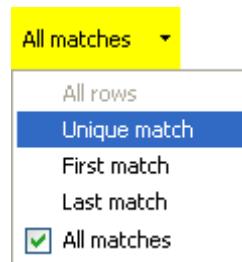
Note: If you have a great number of input tables, note that you can use the minimize/maximize icon to reduce or restore the table size in the **Input** area. The Join binding two tables remain visible even though the table is minimized.

Creating a join automatically assigns a hash key onto the joined field name. The key symbol displays in violet on the input table itself and is removed when the join between the two tables is removed.

Related topics:

- *Schema editor on page 92*
- *Inner join on page 86*

Along with the explicit Join you can select whether you want to filter down to a unique match or if you allow several matches to be taken into account. In this last case, you can choose to only consider the first or the last match or all of them.



Unique Match (java)

This is the default selection when you implement an explicit Join. This means that zero or one match from the Lookup will be taken into account and passed on to the output.

If more matches are available, a warning notification displays.

First or Last Match (java)

This selection implies that several matches can be expected in the lookup. The First or Last Match selection means that in the lookup only the first encountered or the last encountered match will be taken into account and passed onto the main output flow.

The other matches will then be ignored.

All Matches (java)

This selection implies that several matches can be expected in the lookup flow. In this case, all matches are taken into account and passed on to the main output flow.

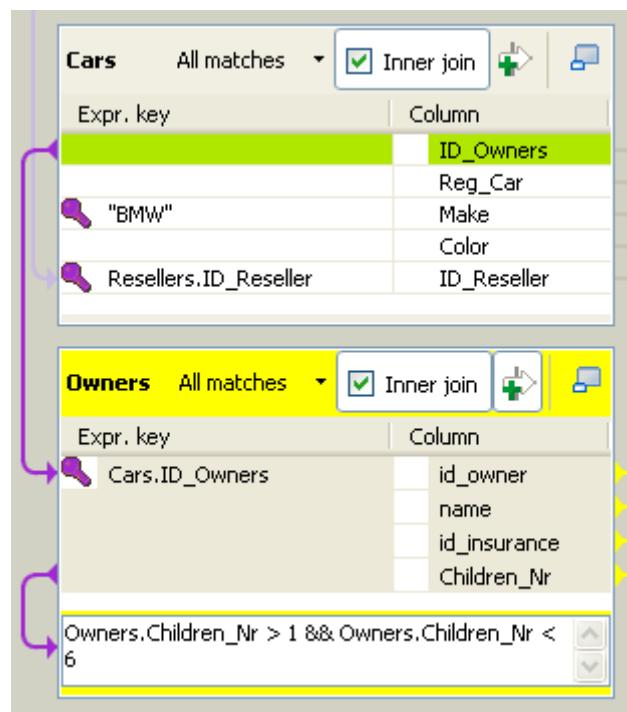
Inner join

The **Inner join** is a particular type of Join that distinguishes itself by the way the rejection is performed.

This option avoids that null values are passed on to the main output flow. It allows also to pass on the rejected data to a specific table called **Inner Join Reject** table.

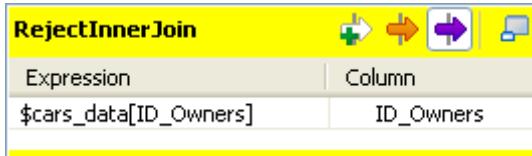
If the data searched cannot be retrieved through the explicit join or the filter (inner) join, in other words, the Inner Join cannot be established for any reason, then the requested data will be rejected to the Output table defined as **Inner Join Reject** table if any.

Basically check the **Inner Join** box located at the top a lookup table, to define this table as **Inner Join** table.



On the **Output** area, click on the **Inner Join Reject** button to define the Inner Join Reject output.

Note: An Inner Join table should always be coupled to an Inner Join Reject table



You can also use the filter button to decrease the number of rows to be searched and improve the performance (in java).

Related topics:

- *Inner Join Rejection on page 91*
- *Filtering an input flow (java) on page 87*

All rows (java)

When you check the **All rows** box , the **Inner Join** feature gets automatically greyed out. This **All rows** option means that all the rows are loaded from the **Lookup** flow and searched against the **Main** flow.

The output corresponds to the Cartesian product of both table (or more tables if need be).

Filtering an input flow (java)

Click the **Filter** button next to the Inner join button to add a **Filter** area.



In the Filter area, type in the condition to be applied. This allows to reduce the number of rows parsed against the main flow, enhancing the performance on long and heterogeneous flows.

You can use the Autocompletion tool via the **Ctrl+Space bar** keystrokes in order to reuse schema columns in the condition statement.



This feature is only available in Java therefore the filter condition needs to be written in Java.

Removing Input entries from table

To remove Input entries, click on the red cross sign on the Schema Editor of the selected table. Press Ctrl or Shift and click on fields for multiple selection to be removed.

Designing a Job Design

Mapping data flows in a job

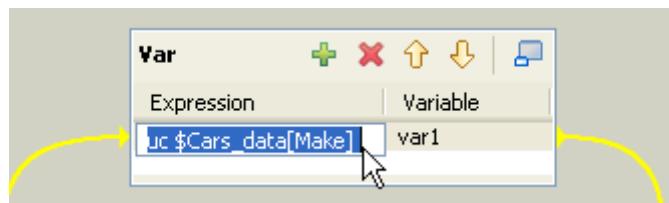
Note: Note that if you remove Input entries from the Mapper schema, this removal also occurs in your component schema definition.

Mapping variables

The Variable table regroups all mapping variables which are used numerous times in various places.

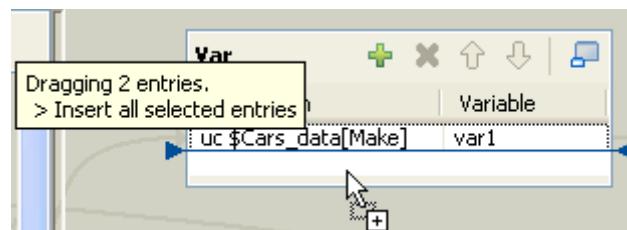
You can also use the Expression field of the Var table to carry out any transformation you want to, using Perl code or Java Code.

Variables help you save processing time and avoid you to retype many times the same data.



There are various possibilities to create variables:

- Type in freely your variables in Perl. Enter the strings between quotes or concatenate functions using a dot as coded in Perl.
- Add new lines using the plus sign and remove lines using the red cross sign. And press Ctrl+Space to retrieve existing global and context variables.
- Drag and drop one or more Input entries to the Var table.



Select an entry on the Input zone or press Shift key to select multiple entries of one Input table.

Press Ctrl to select either non-appended entries in the same input table or entries from various tables. When selecting entries in the second table, notice that the first selection displays in grey. Hold the Ctrl key down to drag all entries together. A tooltip shows you how many entries are in selection.

Then various types of drag and drops are possible depending on the action you want to carry out.

How to...	Associated actions
Insert all selected entries as separated variables.	Simply drag & drop to the Var table. Arrows show you where the new Var entry can be inserted. Each Input is inserted in a separate cell.
Concatenate all selected input entries together with an existing Var entry	Drag & drop onto the Var entry which gets highlighted. All entries get concatenated into one cell. Add the required operators using Perl/Java operations signs. The dot concatenates string variables.
Overwrite a Var entry with selected concatenated Input entries	Drag & drop onto the relevant Var entry which gets highlighted then press Ctrl and release. All selected entries are concatenated and overwrite the highlighted Var.
Concatenate selected input entries with highlighted Var entries and create new Var lines if needed	Drag & drop onto an existing Var then press Shift when browsing over the chosen Var entries. First entries get concatenated with the highlighted Var entries. And if necessary new lines get created to hold remaining entries.

Accessing global or context variables

Press **Ctrl+Space** to access the global and context variable list.

Appended to the Variable list, a metadata list provides information about the selected column.

Removing variables

To remove a selected Var entry, click on the red cross sign. This removes the whole line as well as the link.

Press Ctrl or Shift and click on fields for multiple selection then click the red cross sign.

Output setting

On the workspace, the creation of a Row connection from the tMap component to the output components adds Output schema tables to the Mapper window.

You can also add an Output schema in your Mapper, using the plus sign from the tool bar of the Output zone.

Unlike the Input zone, the order of output schema tables does not make such a difference, as there is no subordination relationship between outputs (of Join type).

Once all connections, hence output schema tables, are created, you can select and organize the output data via drag & drops.

You can drag and drop one or several entries from the Input zone straight to the relevant output table.

Press Ctrl or Shift, and click on entries to carry out multiple selection.

Designing a Job Design

Mapping data flows in a job

Or you can drag expressions from the Var zone and drop them to fill in the output schemas with the appropriate reusable data.

Note that if you make any change to the Input column in the Schema Editor, a dialog prompts you to decide to propagate the changes throughout all Input/Variable/Output table entries, where concerned.

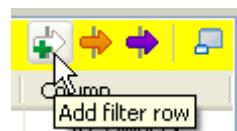
Action	Result
Drag & Drop onto existing expressions	Concatenates the selected expression with the existing expressions.
Drag & Drop to insertion line	Inserts a or several new entries at start or end of table or between two existing lines.
Drag & Drop + Ctrl	Replaces highlighted expression with selected expression.
Drag & Drop + Shift	Adds to all highlighted expressions the selected fields. Inserts new lines if needed.
Drag & Drop + Ctrl + Shift	Replaces all highlighted expressions with selected fields. Inserts new lines if needed.

You can add filters and rejection to customize your outputs.

Filters

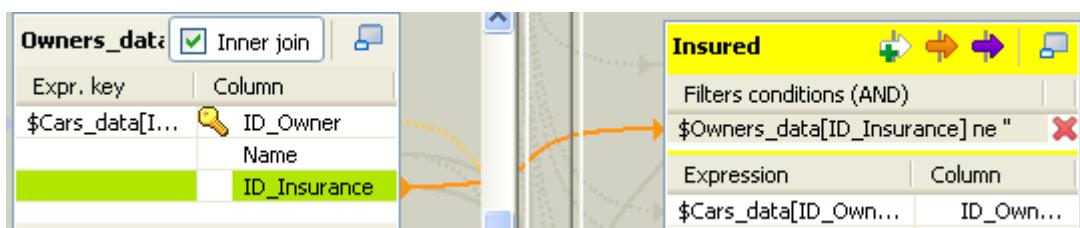
Filters allow you to make a selection among the input fields, and send only the selected fields to various outputs.

Click the plus button at the top of the table to add a filter line.



You can enter freely your filter statements using Perl operators and function.

Drag and drop expressions from the Input zone or from the Var zone to the Filter row entry of the relevant Output table.



An orange link is then created. Add the required Perl/Java operator to finalize your filter formula.

You can create various filters on different lines. The AND operator is the logical conjunction of all stated filters.

Rejections

Reject options define the nature of an output table.

It groups data which do not satisfy one or more filters defined in the regular output tables. Note that as regular output tables, are meant all non-reject tables.

This way, data rejected from other output tables, are gathered in one or more dedicated tables, allowing you to spot any error or unpredicted case.

The Reject principle concatenates all non Reject tables filters and defines them as an ELSE statement.

Create a dedicated table and click the Output reject button to define it as Else part of the regular tables.



You can define several Reject tables, to offer multiple refined outputs. To differentiate various Reject outputs, add filter lines, by clicking on the plus arrow button.

Once a table is defined as Reject, the verification process will be first enforced on regular tables before taking in consideration possible constraints of the Reject tables.

Note that data are not exclusively processed to one output. Although a data satisfied one constraint, hence is routed to the corresponding output, this data still gets checked against the other constraints and can be routed to other outputs.

Inner Join Rejection

The Inner Join is a Lookup Join. The Inner Join Reject table is a particular type of Rejection output. It gathers rejected data from the main row table after an Inner Join could not be established.

To define an Output flow as container for rejected Inner Join data, create a new output component on your job that you connect to the Mapper. Then in the Mapper, click on the Inner Join Reject button to define this particular Output table as Inner Join Reject table.



Removing Output entries

To remove Output entries, click on the cross sign on the Schema Editor of the selected table.

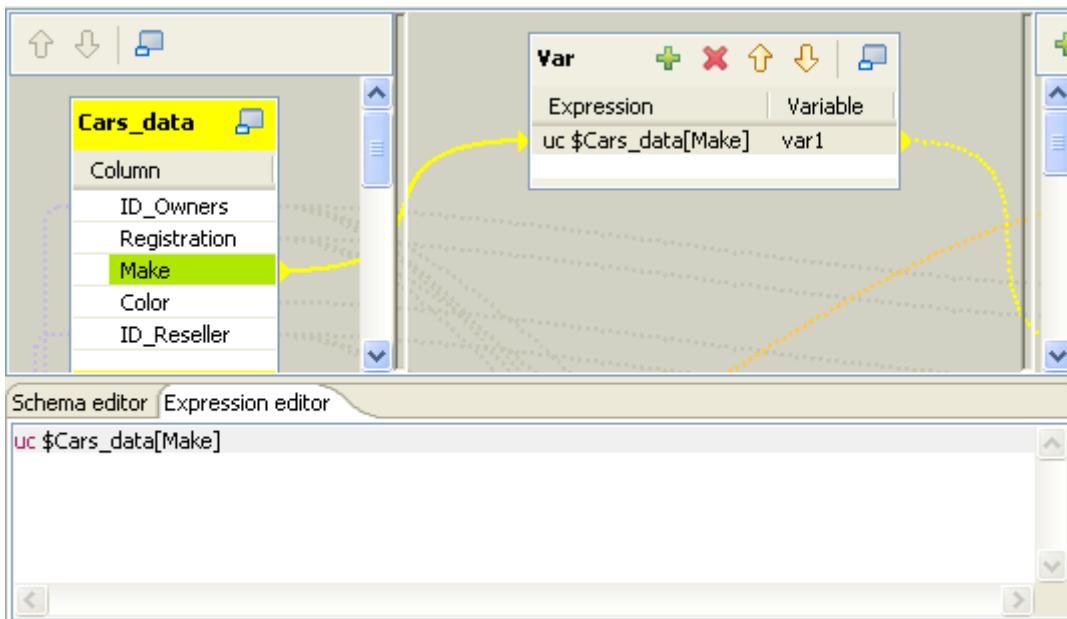
Expression editor

All expressions (Input, Var or Output) and constraint statements can be viewed and edited from the Expression editor. This editor provides visual comfort to write any functions or transformation in a handy dedicated window.

Select the expression to be edited. Click on Expression editor.

Designing a Job Design

Mapping data flows in a job



Enter the Perl code or Java code according to your needs. The corresponding table expression is synchronized.

Note: Refer to the relevant Perl or Java documentation for more information regarding functions and operations.

Schema editor

The Schema Editor details all fields of the selected table.

The screenshot shows the JasperETL Schema editor interface. The title bar has tabs for 'Schema editor' and 'Expression editor', with 'Schema editor' selected. Below the tabs, a table named 'owners_data' is displayed with four columns: ID_Owner, First Name, Name, and ID_Insurance. The 'ID_Owner' column is defined as a primary key (indicated by a key icon) with type int, length 2, precision 0, and nullable checked. The 'First Name' column is currently being edited, with the text 'First N' visible in the input field. The 'Name' and 'ID_Insurance' columns have type String, length 8 and 7 respectively, and are nullable. Below the table is a toolbar with icons for adding, removing, and modifying columns, as well as for loading and saving the schema.

Use the tool bar below the schema table, to add, move or remove columns from the schema.

You can also load a schema from the Repository or export it into a file.

Metadata	Description
Column	Column name as defined on the Mapper schemas and on the Input or Output component schemas
Key	The Key shows if the expression key data should be used to retrieve data through the Join link. If unchecked, the Join relation is disabled.
Type	Type of data. String or Integer. Note: This column should always be defined in Java version.
Length	-1 shows that no length value has been defined in the schema.
Precision	precises the length value if any is defined.
Nullable	Uncheck this box if the field value should not be null
Default	Shows any default value that may be defined for this field.
Comment	Free text field. Enter any useful comment.

Note: Note that Input metadata and Output metadata are independent from each other. You can for instance change the label of a column on the Output side without the column label of the Input schema being changed.

However, any change made to the metadata are immediately reflected in the corresponding schema on the tMap relevant (Input or Output) zone, but also on the schema defined for the component itself on the workspace.

A Red colored background shows that an invalid character has been entered. Most special characters are prohibited in order for the job to be able to interpret and use the text entered in the code. Authorized characters include lower-case, upper-case, figures except as start character.

Browse the mouse over the red field, a tooltip displays the error message.

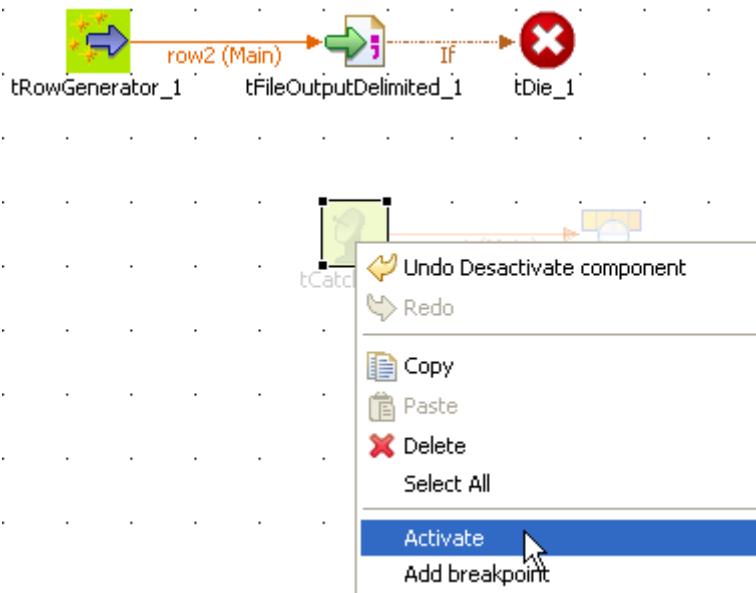
Activating/Disabling a job or sub-job

You can enable or disable the whole job or a sub-job directly connected to the selected component. By default, a component is activated.

In the **Main** properties of the selected component, check or uncheck the **Activate** box.

Designing a Job Design

Defining a job design context and related variables



Alternatively, right-click on the component and select the relevant **Activate/Deactivate** command according to the current component status.

If you disable a component, no code will be generated, you will not be able to add or modify links from the disabled component to active or new components.

Related topic: *Defining the Start component on page 53.*

Disabling a Start component

In the case the component you deactivated is a **Start** component, components of all types and links of all nature connected directly and indirectly to it will get disabled too.

Disabling a non-Start component

When you uncheck the **Activate** box of a regular (non Start) component, are deactivated only the selected component itself along with all direct links.

If a direct link to the disabled component is a main Row connection to a sub-job. All components of this sub-job will also get disabled.

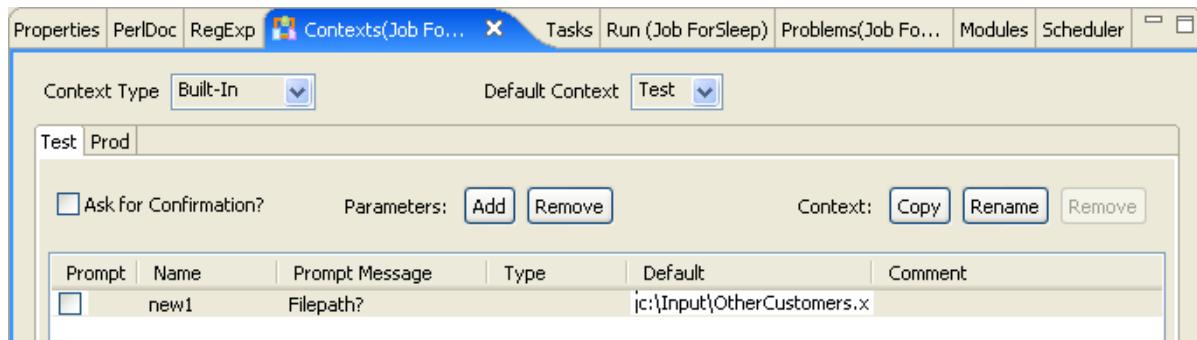
Defining a job design context and related variables

Depending on the circumstances the job design is being used in, you might want to manage it differently for various environment (Prod and Test for example).

For instance, there might be various stages of test, you want to perform and validate before a job design is ready to go-live for production use. Therefore, **JasperETL** offers you to have multiple contexts to avoid having to fill in components properties twice or more for each context of use.

For example, one context can be dedicated to development while another is prepared to production. The only difference between both contexts, being some critical field values and the scale of use. The production context is thus a duplicate of the development context with few changes that make the difference.

Click on the **Contexts** tab located next to the properties tab system, to display the context configuration view.



If the **Contexts** view is not offered in the tab system, go to **Window > Show view > Talend**, and select **Contexts**.

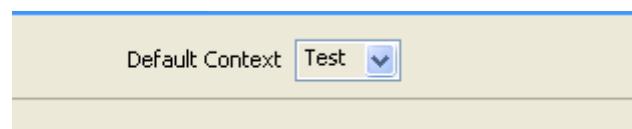
Adding or renaming a context

To change the name of an existing context, click on **Rename** and enter the new Context name in the dialog box showing up.

To add a new context tab, click on **Copy**. The default context is thus duplicated in a new tab. Enter a name for the newly created context on the dialog box.



When you copy a context, the entire default context legacy is copied over to the new context. You hence only need to edit the relevant fields to customize the context according to the targeted use.



The drop-down list **Default Context** shows all the contexts you created.

You can switch Default context by simply selecting the new default context on the list.

Note that the Default context can never be deleted. There should always be a context to run the job. This context being called Default or any other name.

Defining the context variables

In any **Properties** field defining a component, you can use an existing global variable or a context variables.

Press **Ctrl+Space** to display the whole list of global and context variables used in various predefined Perl functions. The context variables are created by the user for a particular context, whereas the global variables are a system variables.

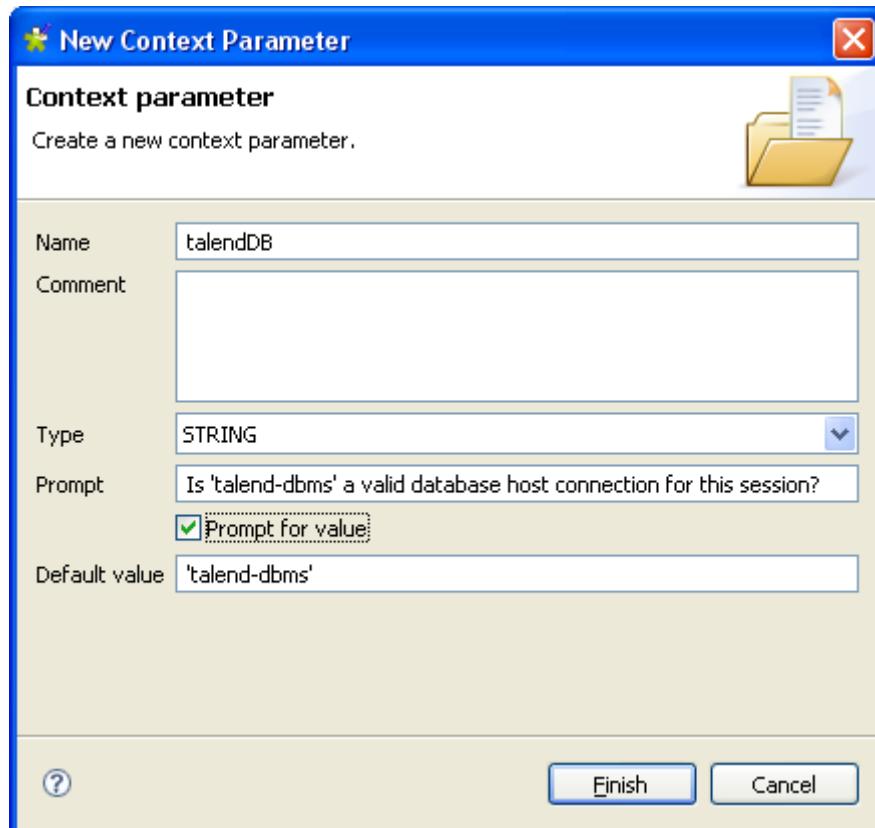
The list grows along with new user (aka context) variable creations.

Related topic: *Defining a job design context and related variables on page 94*

Short creation of context variables

Create quickly your context variables via the **F5** keystroke:

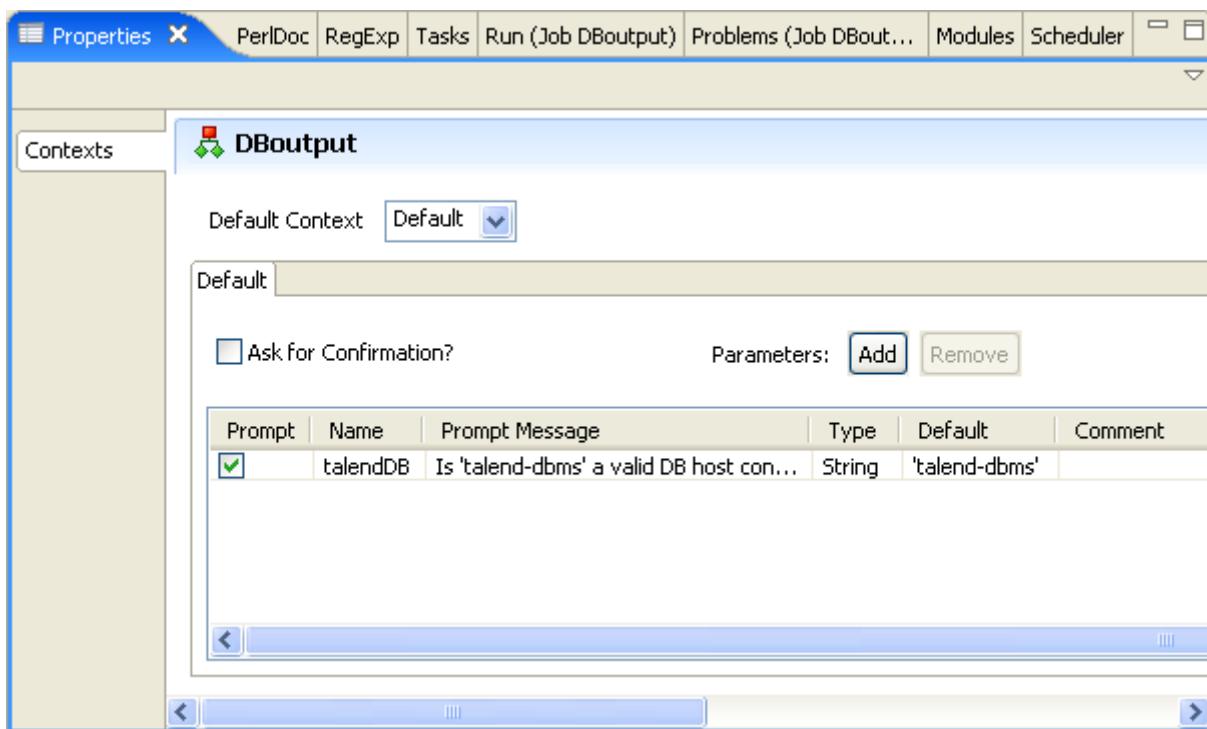
- Place your cursor on the relevant field that you want to parameterize in the current context (possibly the default one).
- Press **F5** to display the context parameter dialog box:



- Give a **Name** to this new variable, fill in the **Comment** zone and choose the **Type**.
- Enter a **Prompt** to be displayed to confirm the use of this variable in the current job execution.
- Check the **Prompt for value** box to display the field as editable value. If you filled in a value already in the corresponding properties field, this value is displayed in the **Default value** field.
- Click **Finish** to validate. Click anywhere on your job's design workspace, and go to the **Properties** tab. Notice that the context parameters lists the newly created variable.

Designing a Job Design

Defining a job design context and related variables



You can modify this variable at any time.

StoreSQLQuery

StoreSQLQuery is a context specific variable, in the sense that it gets set up by the user and is not provided by the system. This variable is dedicated to debugging mainly.

StoreSQLQuery is different from other context variables in the fact that its main purpose is to be used as parameter of a specific global variable, said “Query”. It allows you to dynamically feed the global query variable.

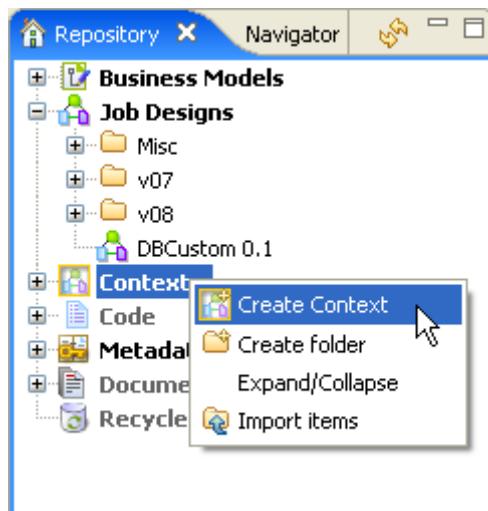
The global variable Query, is available on the proposals list (Ctrl+Space) for couple of DB input components.

For further details on StoreSQLQuery settings, see *Components on page 115*.

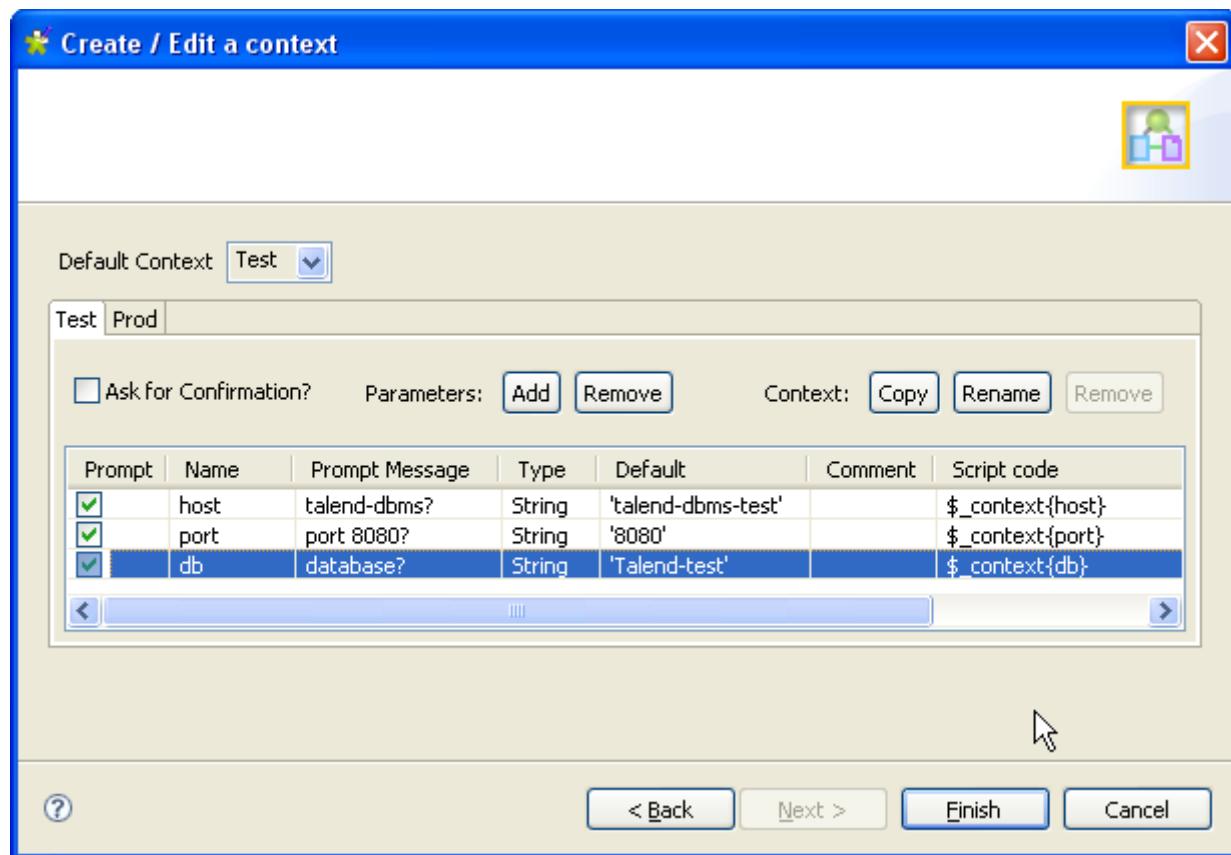
Storing contexts in the Repository

We strongly recommend to store centrally all contexts for ease of use in all your jobs within a project.

Right-click on the **Contexts** entry in the repository and select **Create new Context** in the list.



A wizard helps you to define the context parameters, that you'll be able to select for a Job execution.

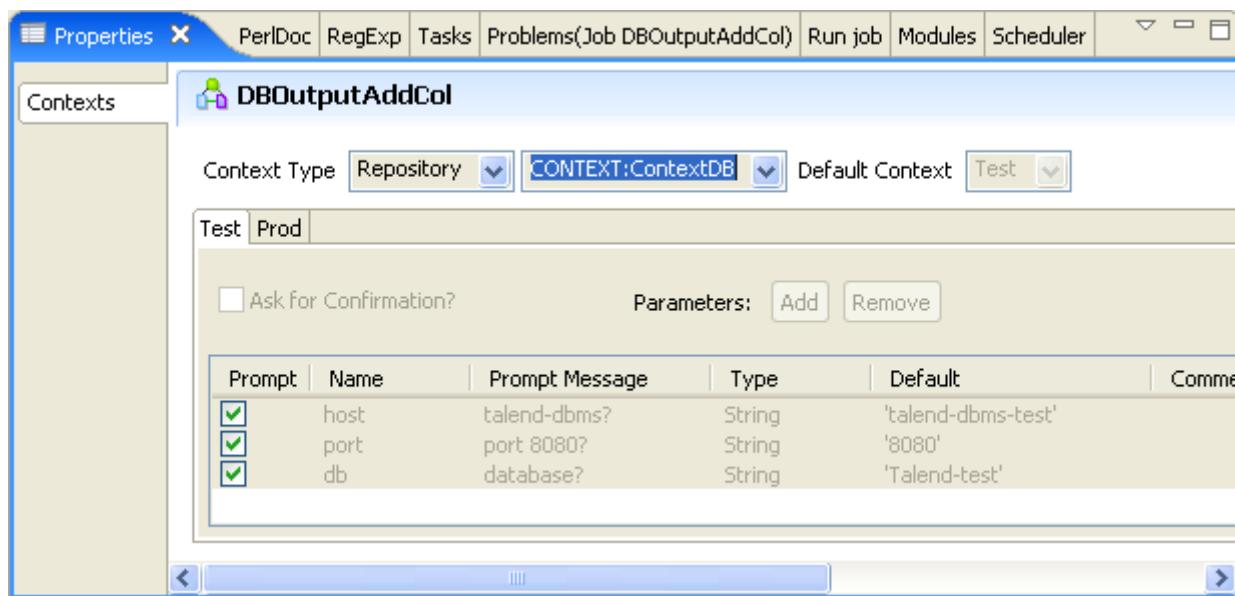


Click **Finish** to validate.

To apply a context to a job, click on any empty area of your relevant job workspace and in the **Context** tab, select **Repository** as **Context type**.

Designing a Job Design

Defining a job design context and related variables



Then select the relevant **Context** from the repository. Click on the relevant tab, the selected context's parameters show as read-only values.

Defining the parameters in the Context tab

A context is characterized by parameters. These parameters are mostly context-sensitive variables which will be added to the list of variables available for reuse in the component-specific properties through the Ctrl+Space keystrokes.

Add a parameter line to the table by clicking on **Add**, and fill the job's common data. You can add as many entries as you need. The **Script code** field points out the corresponding variable created. It follows a script standard which depends on the **Generation language** you selected (Java or Perl) such as in Perl: `$_context{YourParameterName}`

Check the **Ask for confirmation?** box, if you want to be prompted for confirmation before running a job. This can be of use at the go-live stage, when the job is ready to switch from Development to Production context, for instance.

If you also check the prompt checkbox next to the variable entry, you will also be able to modify the variable for a particular job execution.

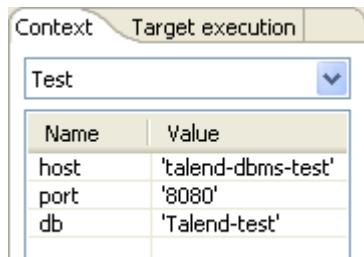
Table 7—

Fields	Description
Prompt	Check this box, if you want the variable to be editable in the Confirmation dialog box when running the job.
Name	Fill in a name for the variable. A context variable is meant to be reused in different component properties, this way it avoids repetition of data typing.
Prompt message	If you asked for a confirmation to popup, fill in this field to define the message in the popup window.
Type	Select in the list the type of data entered.
Default	Enter the default value for this variable. Through this field you can update the value of the variable and hence update all properties calling this variable.
Comment	This is a free text field.
Script code	This field is read-only. A script code is generated according to the parameter name entered. This is the variable format as it shows on the Ctrl+Space bar list.

Note that the variable or parameter name should follow some typing rules and should not contain any forbidden characters, such as space char.

Running a job in selected context

You can select the context you want the job design to be executed in.



Click on **Run Job** tab, and in the **Context** area, select the relevant context among the various ones you created.

If you didn't create any context, only the **Default** context shows on the list.

All the context variables you created for the selected context display, along with their respective value, in a table underneath. If you checked the **Prompt** box next to some variables, you will get a dialog box allowing you to change the variable value for this job execution only.

To make a change permanent in a variable value, you need to change it on the context parameter setup panel. Related topic: *Defining the parameters in the Context tab on page 100*

Running a job

You can execute a job in several ways. This mainly depends on the purpose of your job execution and on your user level.

If you are an advanced Perl/Java user and want to execute your project step by step to check and possibly modify it on the run, see *Running in debug mode on page 103*.

Designing a Job Design

Running a job

If you don't have advanced Perl knowledge and want to execute and monitor your job in normal mode, see *Running in normal mode on page 102*.

Running in normal mode

Make sure you saved your job before running it in order for all properties to be taken into account.

- Click on the **Run Job** tab to access the panel.
- In the **Context** zone, select the right context for the job to be executed in. You can also check the variable values

If you haven't defined any particular execution context, the context parameter table is empty and the context is the default one. Related topic: *Defining a job design context and related variables on page 94*

- Click on **Run** to start the execution.
- On the same panel, the log displays the progress of the execution. The log includes any error message as well as start and end messages. It also shows the job output in case of tLogRow component is used in the job design.

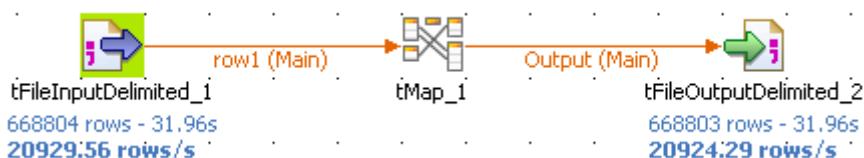
Before running again a job, you might want to remove the log content from the execution panel. Check the **Clear before run** box, for the log to be cleared each time you execute again a job.

If for any reason, you want to stop the job in progress, simply click on **Kill** button. You'll need to click the **Run** button again, to start again the job.

JasperETL offers various informative features, such as statistics and traces, facilitating the job monitoring and debugging work.

Displaying Statistics

The **Statistics** feature displays each component performance rate, underneath the component icon on the design workspace.



It shows the number of rows processed and the processing time in row per second, allowing you to spot straight away any bottleneck in the data processing flow.

Note: Exception is made for external components which cannot offer this feature if their design doesn't include it.

Check the **Statistics** box to activate the stats feature and click again to disable it.

The Stats calculation only starts along with the job execution launchs, and stops at the end of it.

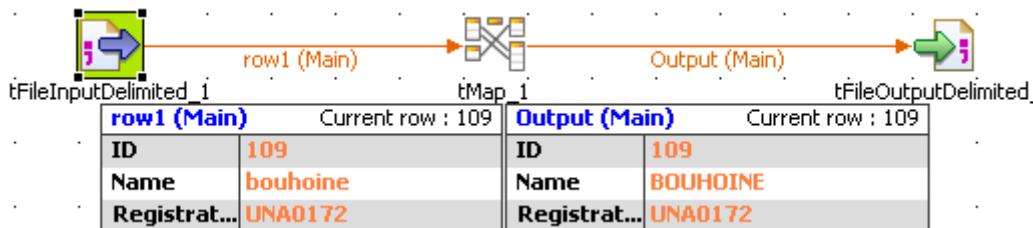
Click **Clear** to remove the calculated stats displayed. Check the Clear before Run box to reset the Stats feature before each execution.

Note: The statistics thread slows down sensibly the time performance of a job execution as the job must send these stats data to the Designer in order to be displayed.

Displaying Traces

The tracking feature is relatively basic in **JasperETL** for the time being. But it should be enhanced in a near future.

It provides a row by row view of the component behaviour and displays the dynamic result next to the row link.



This feature allows you to monitor all components of a job, without switching to Debug mode, hence without requiring advanced Perl/Java knowledge.

The Traces function displays the content of processed rows in a table.

Note: Exception is made for external components which cannot offer this feature if their design doesn't include it.

Click on **Traces** button to activate the tracking feature and click again to disable it.

The trace only launches along with the job launches, and stops at the end of it.

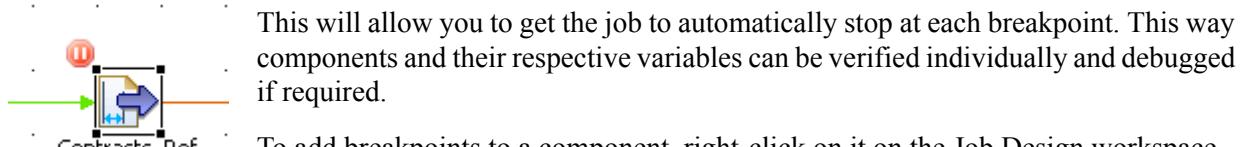
Click on **Clear** to remove the tracking data displayed.

Note: Note that the table is limited horizontally, however mouse over the table to display the whole data table. On the other hand, the table does not have any vertical limitation. This might become an issue for very long data tables.

Running in debug mode

Note that to run a job in Debug mode, you need the EPIC module to be installed.

Before running your job in Debug mode, add breakpoints to the major steps of your job flow.



A pause icon displays next to the component where the break is added.

Designing a Job Design

Saving or exporting your jobs

To switch to debug mode, click on the **Debug** button on the **Run Job** panel. **JasperETL**'s window gets reorganised for debugging.

You can then run the job step by step and check each breakpoint component for the expected behaviour and variable values.

To switch back to Talend Open Studio designer mode, click on **Window**, then **Perspective** and select **Talend Open Studio**.

Saving or exporting your jobs

Saving a job

When closing a job or **JasperETL**, a dialog box prompts you to save the currently open jobs if not already done.

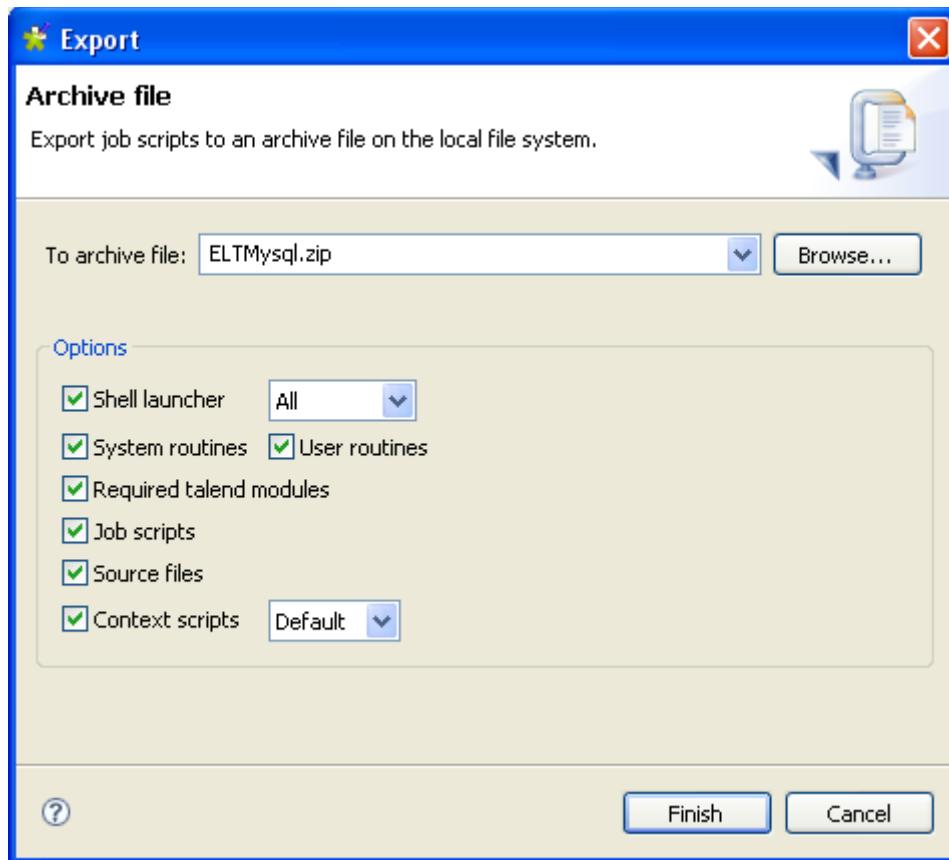
In case several job were unsaved, check the box facing the jobs you want to save. The Job is saved in the relevant project folder of your workspace directory.

Alternately, click **File > Save** or press **Ctrl+S**.

Exporting job scripts

You can store and file your job scripts into an archive using the export functionality.

Right-click on the relevant job in the Repository, and select **Export Job Scripts**.



Select the type of files you want to add to your archive and give a path to the archive file to create.

If you want to reuse the job in another version of **JasperETL**, make sure you checked the **Source files** box.

Click **Finish** when complete.

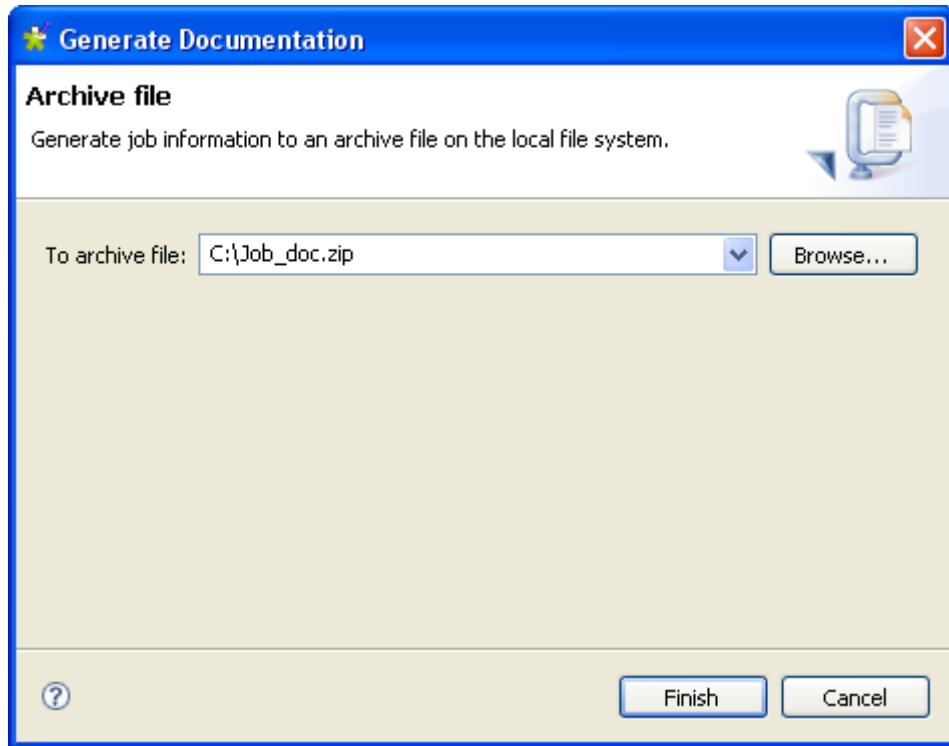
The output file is an archive that you can store or send as you need. It contains the run.bat and run.sh needed to execute the scripts.

Note: The Export Job scripts dialog might differ slightly whether you work on a Java or a Perl version of the product.

Generating HTML documentation

JasperETL allows you to produce detailed documentation in HTML of the jobs selected.

- On the **Repository**, right-click on a **Job** entry or select several **Job Designs** to produce multiple documentations.
- Select **Generate Doc as HTML** on the pop-up menu.



- Browse to the location where the generated documentation archive should be stored.
- On the same field, type in a Name for the archive gathering all generated documents.
- Click **Finish** to validate the generation operation.

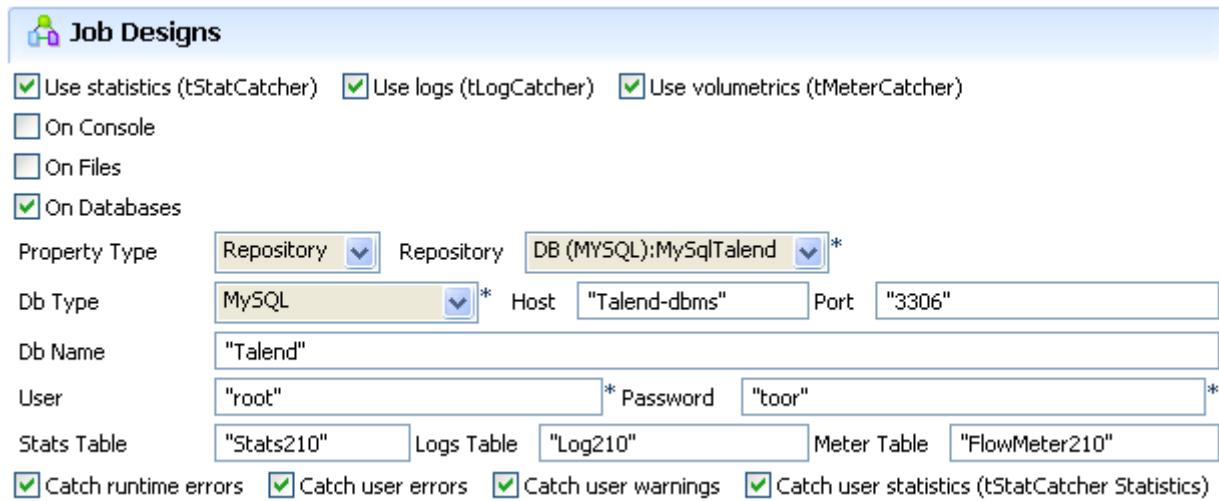
The archive file contains all required files along with the Html output file. Open the html file in your favourite browser.

Automating stats & logs use

If you have a great need of log, statistics and other measurement of your data flows, you are facing the issue of having too many log-related components loading your job designs. You can automate the use of **tFlowMeterCatcher**, **tStatCatcher**, **tLogCatcher** functionalities without using the components in your job thanks to the **Stats & Logs** tab.

The **Stats & Logs** tab is located underneath the Design workspace and prevents overloading your jobs designs by superseding the log-related components with a general log configuration.

- Click anywhere on your Job design but on the component.
- Select the **Stats & Logs** tab to display the configuration view.



- Set the relevant details depending on the output you prefer (console, file or database).
- Check the relevant **Catch** option according to your needs.

Shortcuts and aliases

Below is a table gathering all keystrokes currently in use:

Designing a Job Design

Shortcuts and aliases

Table 8—

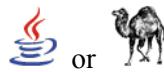
Operation	Context	Shortcut
Show Properties view	Global application	F3
Show Run Job view	Global application	F4
Run current job or Show Run Job view if no job is open.	Global application	F6
Show Module view	Global application	Ctrl + F2
Show Problems view	Global application	Ctrl + F3
Switch to current Job Design view	Global application	Ctrl + H
Show Code tab of current Job	Global application	Ctrl + G
Synchronize components perljet templates and associated java classes	Global application	Ctrl + Shift + F3
Switch to Debug mode	From Run Job view	F7
Create a context variable from any properties field	From any job Properties tab view	F5
Kill current job	From Run Job view	F8
Refresh Modules install status	From Modules view	F5

At any place of your job properties setting, you can use **Ctrl+Space** to set the relevant field value as context variable.

—Components—

Components

This chapter details the main components' properties provided in the Palette of **JasperETL**. Each component has a specific list of properties and parameters, editable through the **Properties** tab of the **Properties** panel.



In the component properties section, an icon or points out whether the component is available in Java and/or in Perl.

Click on one of the following link to jump to the relevant component datasheet:

Families		Components		
Business Connectors	<i>Salesforce</i>	tSalesforceInput	tSalesforceOutput	
	<i>SugarCRM</i>	tSugarCRMInput	tSugarCRMOOutput	
Data quality		tFuzzyMatch	tAddCRCRow	
Databases	<i>DB Generic</i>	tDBInput	tDBOutput	tDBSQLRow
	<i>DB2</i>	tDB2Input	tDB2Output	tDB2Row
	<i>MSSqlServer</i>	tMSSqlInput	tMSSqlOutput	tMSSqlRow
	<i>MySQL</i>	tMySqlInput	tMySqlOutput	tMySqlRow
		tMySqlOutputBulk	tMySqlBulkExec	tMySqlOutputBulk Exec
		tMySqlConnection	tMySqlCommit	
	<i>SQLite</i>	tSQLiteInput	tSQLiteOutput	
ELT		tELTMysqlInput	tELTMysqlMap	tELTMysqlOutput
	<i>Oracle</i>	tOracleInput	tOracleOutput	tOracleRow
		tOracleBulkExec	tELTOraclInput	tELTOraclMap
		tELTOraclOutput		
	<i>PostgresSQL</i>	tPostgresqlInput	tPostgresqlOutput	tPostgresqlRow
	<i>Sybase</i>	tSybaseInput	tSybaseOutput	tSybaseRow
		tSybaseBulkExec		
File	<i>Input</i>	tFileInputDelimited	tFileInputPositional	tFileInputRegex
		tFileInputXML	tFileInputMail	

Components

Families		Components		
	<i>Management</i>	tFileDialog	tFileCompare	tFileUnarchive
		tFileCopy	tFileDelete	
	<i>Output</i>	tFileOutputXML	tFileOutputLDIF	tFileOutputExcel
Internet	FTP	tFTP		
		tSendMail	tWebServiceInput	
Log/Error		tLogRow	tStatCatcher	tLogCatcher
		tWarn	tDie	
Misc		tMsgBox	tRowGenerator	tContextLoad
Processing		tPerl	tMap	tAggregateRow
		tSortRow	tUniqRow	tNormalize
		tDenormalize		
System		tSystem	tRunJob	
XML		tDTDValidator	tXSDValidator	tXSLT

tAggregateRow



tAggregateRow properties

Component family	Processing	
Function	tAggregateRow receives a flow and aggregates it based on one or more columns. For each output line, are provided the aggregation key and the relevant result of set operations (min, max, sum...).	
Purpose	Helps to provide a set of metrics based on values or calculations.	
Properties	Schema type and <i>Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository. Click Edit Schema to make changes to the schema. Note that if you make changes, the schema automatically becomes built-in.
	Built-in: The schema will be created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>	
	Repository: The schema already exists and is stored in the Repository, hence can be reused in various projets and job flowcharts. Related topic: <i>Setting a repository schema on page 52</i>	
	Group by	Define the aggregation sets, the values of which will be used for calculations.
	Output Column: Select the column label in the list offered based on the schema structure you defined. You can add as many output columns as you wish to make more precise aggregations. Ex: Select Country to calculate an average of values for each country of a list or select Country and Region if you want to compare one country's regions with another country' regions.	
	Input Column: Match the input column label with your output columns, in case the output label of the aggregation set needs to be different.	

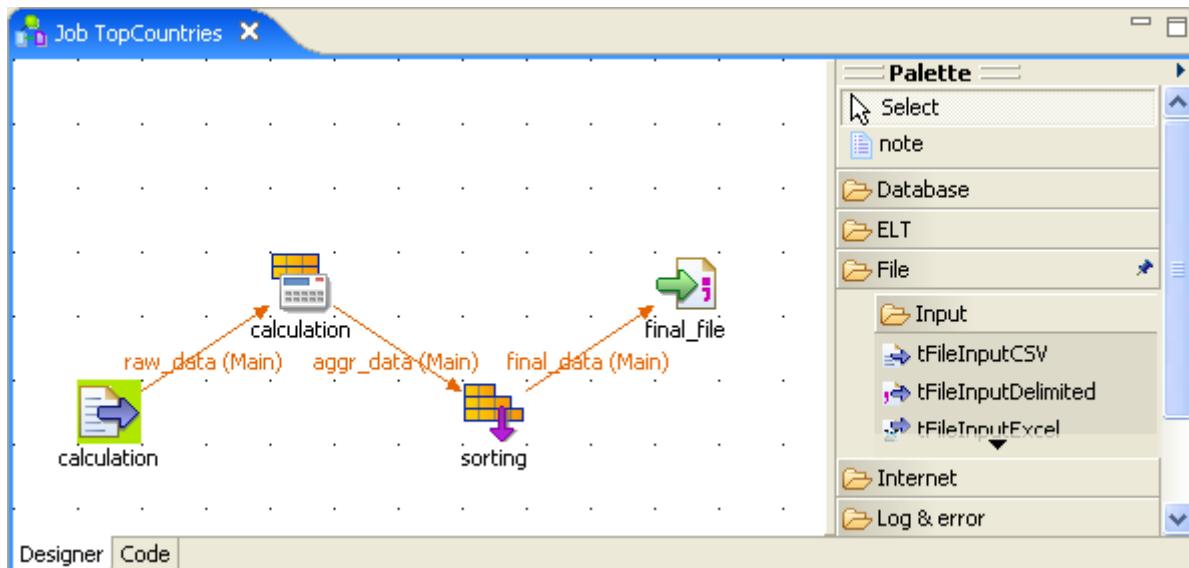
Components

tAggregateRow

	Operations	Select the type of operation along with the value to use for the calculation and the output field.
		Output Column: Select the destination field in the list.
		Function: Select the operator among: count, min, max, avg, first, last.
		Input column: Select the input column from which the values are taken to be aggregated.
Usage		This component handles flow of data therefore it requires input and output, hence is defined as an intermediary step. Usually the use of tAggregateRow is combined with the tSortRow component.
Limitation		n/a

Scenario: Aggregating values and sorting data

The following scenario describes a four-component job. As input component, a CSV file contains countries and notation values to be sorted by best average value. This component is connected to a **tAggregateRow** operator, in charge of the average calculation then to a **tSortRow** component for the ascending sort. The output flow goes to the new csv file.



- From the **File** folder in the Palette, click and drop a **tFileInputCSV** component.
- Click on the label and rename it as *Countries*. Or rename it from the **View** tab panel
- In the **Properties** tab panel of this component, define the filepath and the delimitation criteria. Or select the metadata file in the repository if it exists.

- Click on **Edit schema...** and set the columns: *Countries* and *Points* to match the file structure. If your file description is stored in the Metadata area of the Repository, the schema is automatically uploaded when you click on **Repository** in **Schema type** field.
- Then from the **Processing** folder in the Palette, click and drop a **tAggregateRow** component. Rename it as *Calculation*.
- Connect *Countries* to *Calculation* via a right-click and select **Row > Main**.
- Double-click on *Calculation* (**tAggregateRow** component) to set the properties. Click on **Edit schema** and define the output schema. You can add as many columns as you need to hold the set operations results in the output flow.

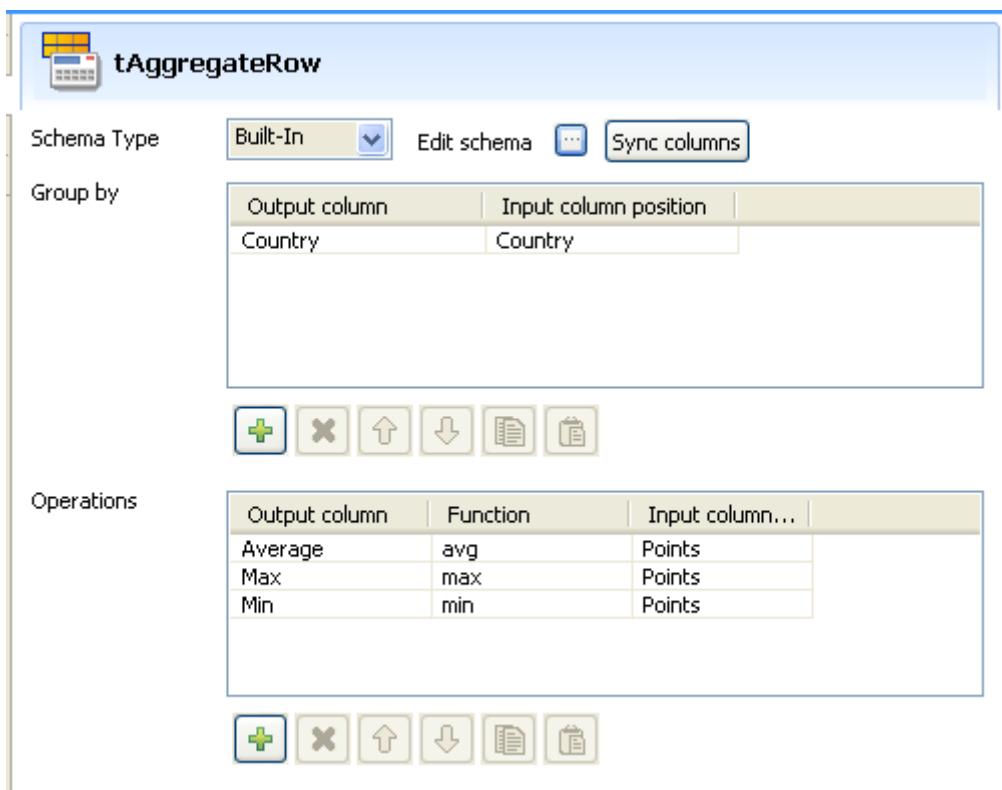
tAggregateRow_1 (Output)						
Column	Key	Type	Length	Precision	Nullable	Com...
Country	<input checked="" type="checkbox"/>		-1	-1	<input type="checkbox"/>	
Average	<input type="checkbox"/>		-1	-1	<input type="checkbox"/>	
Max	<input type="checkbox"/>		-1	-1	<input type="checkbox"/>	
Min	<input type="checkbox"/>		-1	-1	<input type="checkbox"/>	

Toolbox:

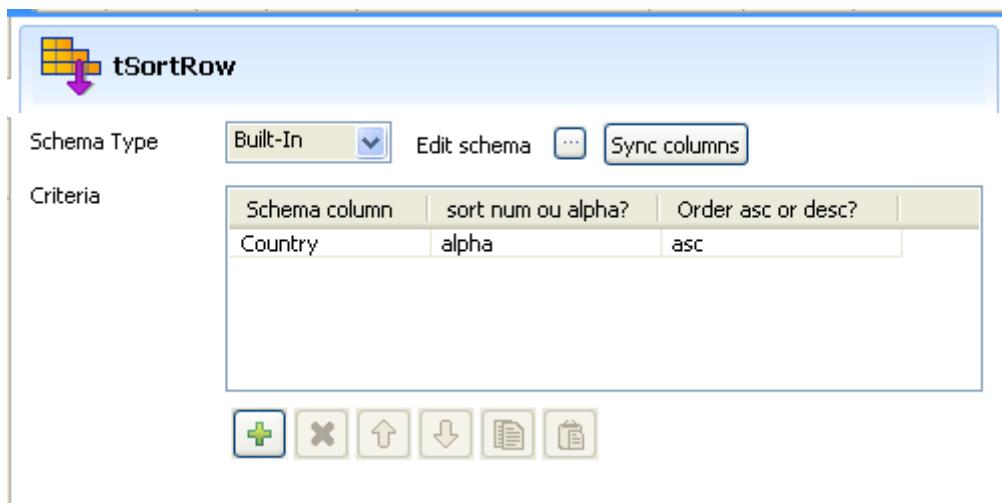
- In this example, we'll calculate the average notation value and we will display the max and the min notation for each country, given that each country holds several notations. Click OK when the schema is complete.
- To carry out the various set operations, back in the **Properties** panel, define the sets holding the operations in the **Group By** area. In this example, select **Country** as group by column. Note that the output column needs to be defined a key field in the schema. The first column mentioned as output column in the Group By table is the main set of calculation. All other output sets will be secondary by order of display.
- Choose the input column which the values will be taken from.
- Then fill in the various operations to be carried out. The functions are average, min, max for this use case. Select the Input columns, where the values are taken from.

Components

tAggregateRow

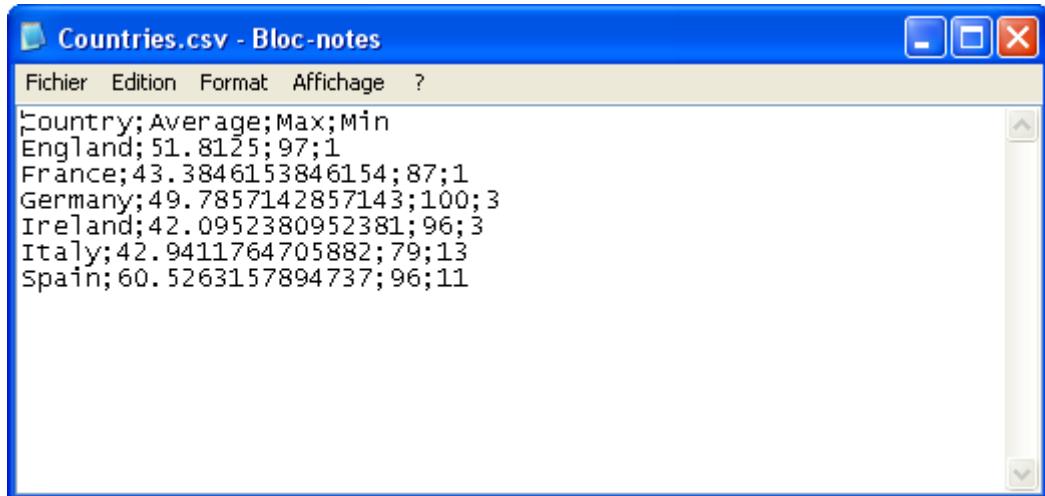


- Click and drop a **tSortRow** component from the Palette onto the modeling workspace. For more information regarding this component , see *tSortRow properties on page 310*.
- Connect the **tAggregateRow** to this new component using a row main link.
- On the Properties tab of the tSortRow component, define the column the sorting is based on, the sorting type and order.



- In this case, the column to be sorted by is Country, the sort type is alphabetical and the order is ascending.

- Add a last component to your job, to set the output flow. Click and drop a tFileOutputDelimited and define it.
- Connect the tSortRow component to this output component.
- In the Properties panel, enter the output filepath. Edit the schema if need be. In this case the delimited file is of csv type. And check the Include Header box to reuse the schema column labels in your output flow.
- Press F6 to execute the job. The csv file thus created contains the aggregating result.





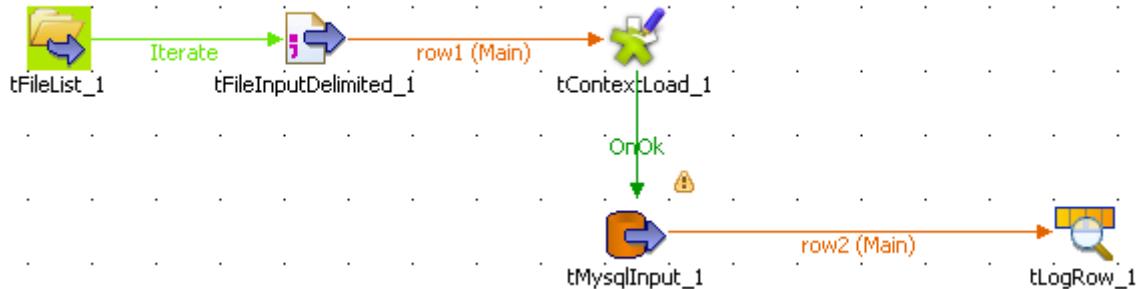
tContextLoad

tContextLoad properties

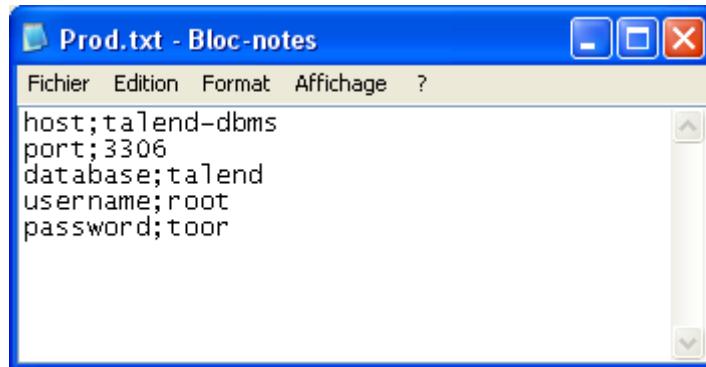
Component family	Misc	
Function	tContextLoad modifies dynamically the values of the active context.	
Purpose	tContextLoad can be used to load a context from a flow. This component performs also two controls. It warns when the parameters defined in the incoming flow are not defined in the context, and the other way around, it also warns when a context value is not initialized in the incoming flow. But note that this does not block the processing.	
Properties	<p><i>Schema type and Edit Schema</i></p> <p>In the tContextLoad use, the schema must be made of two columns, including the parameter name and the parameter value to be loaded.</p> <p>A schema is a row description, i.e., it defines the fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository. Click Edit Schema to make changes to the schema. Note that if you make changes, the schema automatically becomes built-in.</p>	
	<p>Built-in: The schema will be created and stored locally for this component only.</p> <p>Related topic: <i>Setting a built-in schema on page 51</i></p>	
	<p>Repository: The schema already exists and is stored in the Repository, hence can be reused in various projects and job flowcharts.</p> <p>Related topic: <i>Setting a repository schema on page 52</i></p>	
	<i>Print operations</i>	Check this box to display the context parameters set in the Run job view.
Usage	This component relies on the data flow to load the context values to be used, therefore it requires a preceding input component and thus cannot be a start component.	
Limitation	tContextLoad does not create any non-defined variable in the default context.	

Scenario: Dynamic context use in MySQL DB insert

This scenario is made of two subjobs. The first subjob aims at dynamically load the context parameters, and the second subjob uses the loaded context to display the content of a DB table.



- Click and drop a **tFilelist**, **tFileInputDelimited**, **tContextLoad** for the first subjob.
- And click and drop the **tMysqlInput** and a **tLogRow** for the second subjob.
- Connect all the components together.
- Create as many delimited files as there are different contexts and store them in a specific directory, named *Contexts*. In this scenario, *test.txt* contains the local database connection details for testing purpose. And *prod.txt* holds the actual production db details.
- Each file is made of two fields, contain the parameter name and the corresponding value, according to the context.

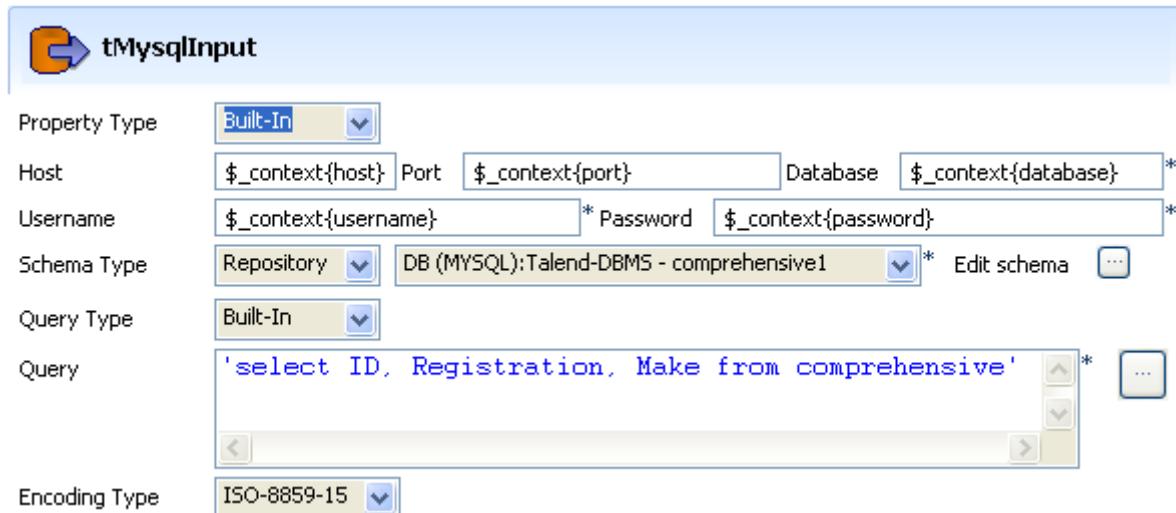


- In the **tFileList** component **Properties** panel, select the directory where both context files, *test* and *prod*, are held.
- In the **tFileInputDelimited** component **Properties** panel, press **Ctrl+Space bar** to access the global variable list. Select **\$_globals{tFileList_1}{CURRENT_FILEPATH}** to loop on the context files' directory.
- Define the schema manually (Built-in). It contains two columns defined as: *Key* and *Value*.
- Accept the defined schema to be propagated to the next component (**tContextLoad**).

Components

tContextLoad

- For this scenario, check the **Print operations** box in order for the context parameters in use to be displayed on the **Run Job** panel.
- Then double-click to open the **tMySQLInput** component **Properties**.
- For each of the field values being stored in a context file, press F5 and define the user-defined context parameter. For example: The **Host** field has for value parameter `$_context{host}`, as the parameter name is *host* in the context file. Its actual value being *talend-dbms*.



- Then fill in the Schema information. If you store the schema in the Repository Metadata, then you can retrieve by selecting **Repository** and the relevant entry in the list.
- And type in the SQL **Query** to be executed on the DB table specified. In this case, a simple select of three columns of the table, which will be displayed on the **Run Job** tab, through the **tLogRow** component.
- Eventually, press **F6** to run the job.

Job ContextLoad

Execution

Context Target execution

Default

Name	Value
host	'talend-dbms'
port	'3306'
database	'talend'
username	'root'
password	'toor'

Debug Run Kill

Clear before run Exec time

Starting job ContextLoad at 17:22 29/03/2007.

```
tContextLoad set key host with value talend-dbms
tContextLoad set key port with value 3306
tContextLoad set key database with value talend
tContextLoad set key username with value root
tContextLoad set key password with value toor
12|4322 DP 76|BMW|||||
15|0142 CB 08|BMW|||||
18|8545 GP 25|Mercedes|||||
24|5382 KC 94|Volkswagen|||||
40|8386 GH 71|Mercedes|||||
```

The context parameters as well as the select values from the DB table are all displayed on the **Run Job** view.

tCreateTable



tCreateTable Properties

Component family	Databases	
Function	tCreateTable creates, drops and creates or clear the specified table.	
Purpose	This Java specific component helps create or drop any database table	
Properties	<i>DB Type</i>	Select the DBMS type in the List offered.
	<i>Special Action</i>	Select the action to be carried out on the database among: Create table : when you know already that the table doesn't exist. Create table when not exists : when you don't know whether the table is already created or not Drop and create table : when you know that the table exists already and needs to be replaced.
	<i>Use existing connection</i>	Check this box in case you use tMysqlConnection or tOracleConnection component.
	<i>Property type</i>	Either Built-in or Repository
		Built-in : No existing property data is to be retrieved
		Repository : Select the Repository file where Properties are stored. The following fields are pre-filled in using fetched data.
	Host	Database server IP address
	Port	Listening port number of DB server.
	Database	Name of the database
	<i>Username and Password</i>	DB user authentication data.
	<i>New table name</i>	Type in between quotes a name for the newly created table.
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields to be processed and passed on to the next component. The schema is either built-in or remotely stored in the Repository. ⚠️ Reset the DB type by clicking the relevant button, to make sure data type is correct
		Built-in : The schema is created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>

		Repository: The schema already exists and is stored in the Repository, hence can be reused. Related topic: <i>Setting a repository schema on page 52</i>
	<i>Mapping</i>	Select the correct mapping according to your Db type. This allows a check of DB type in the schema defined. If the DB type standards do not match, they will display in a different color in the Edit Schema .
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.
Usage	This component offers the flexibility benefit of the DB query and covers all possibilities of SQL queries. More scenarios are available for specific DB Input components	

Scenario: Creating new table in a MySql Database

The job described below aims at creating a table in a database, made of a dummy schema taken from a delimited file schema stored in the Repository. This job is composed of a single component.



- Click and drop a **tCreateTable** component from the **Databases** family in the **Palette**.
- In the **Properties** view, define the **Database type** on *MySQL* for this use case.

tCreateTable_1

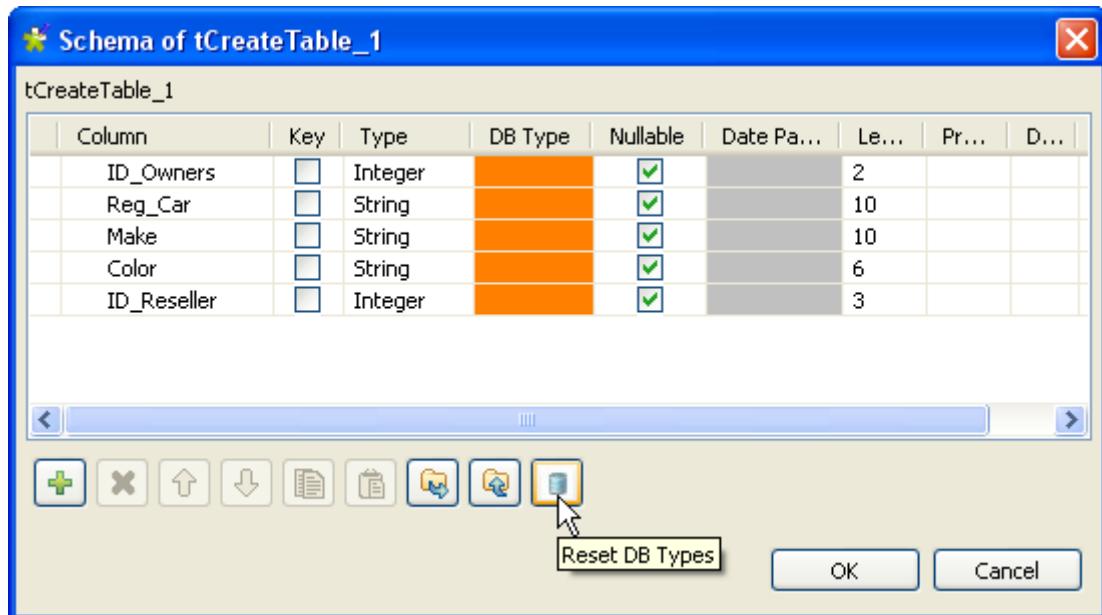
DB Type	MySQL	Special Action	Create table
<input type="checkbox"/> Use Existing Connection			
Property Type	Repository	Repository	DB (MYSQL):MySQL
Host	"talend-dbms"	Port	"3306"
User Name	"root"	* Password	"toor"
New Table Name	"NewCarsTable"		
Schema Type	Built-In	Edit schema	[...]
Mapping	Mapping Mysql	Encoding Type	ISO-8859-15

- In the **Special Action** list, select **Create table**.

Components

tCreateTable

- Check **Use Existing Connection** only in the case, you are using a dedicated connection component, see *tMysqlConnection on page 254*. In this use case, we won't use this option.
- In the **Property type** field, select **Repository** so that all following connection fields are automatically filled in. If you didn't define a **Metadata DB connection** entry for your Db connection, fill in manually the details as **Built-in**.
- In the **New Table Name** field, fill in a name for the table to be created.
- If you want to retrieve the **Schema** from the Metadata (it doesn't need to be a DB connection Schema metadata), select Repository then the relevant entry.
- In any case (Built-in or Repository) click **Edit Schema** to check the Data type mapping.



- Click the **Reset DB Types** button in case the DB type column is empty or shows discrepancies marks (orange colour). This allows to map any data type to the relevant DB data type.
- Click **OK**.
- Then press **F6** to run the job.

The table is created empty but with all columns defined in the Schema.

tAddCRCRow



tAddCRCRow properties

Component family	Data quality	 
Function	Calculates a surrogate key based on one or several columns and adds it to the defined schema	
Purpose	Providing a unique ID helps improving the quality of processed data.	
Properties	<i>Schema type</i> and <i>Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository. In this component, a new CRC column is automatically added.
	Built-in: The schema will be created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>	
	Repository: The schema already exists and is stored in the Repository, hence can be reused in various projects and job designs. Related topic: <i>Setting a repository schema on page 52</i>	
	<i>Implication</i>	Tick the checkbox facing the relevant columns to be used for the surrogate key checksum.
	<i>CRC type</i>	Select the CRC type length. The longer the CRC, the least overlap.
Usage	This component is an intermediary step, and requires an input flow as well as an output.	
Limitation	n/a	

Scenario: Adding a surrogate key to a file

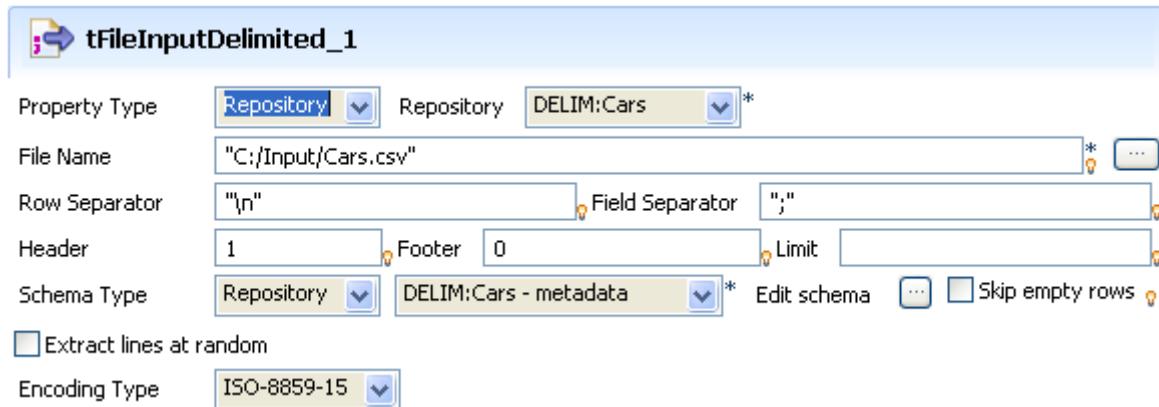
This scenario describes a job adding a surrogate key to a delimited file schema.

Components

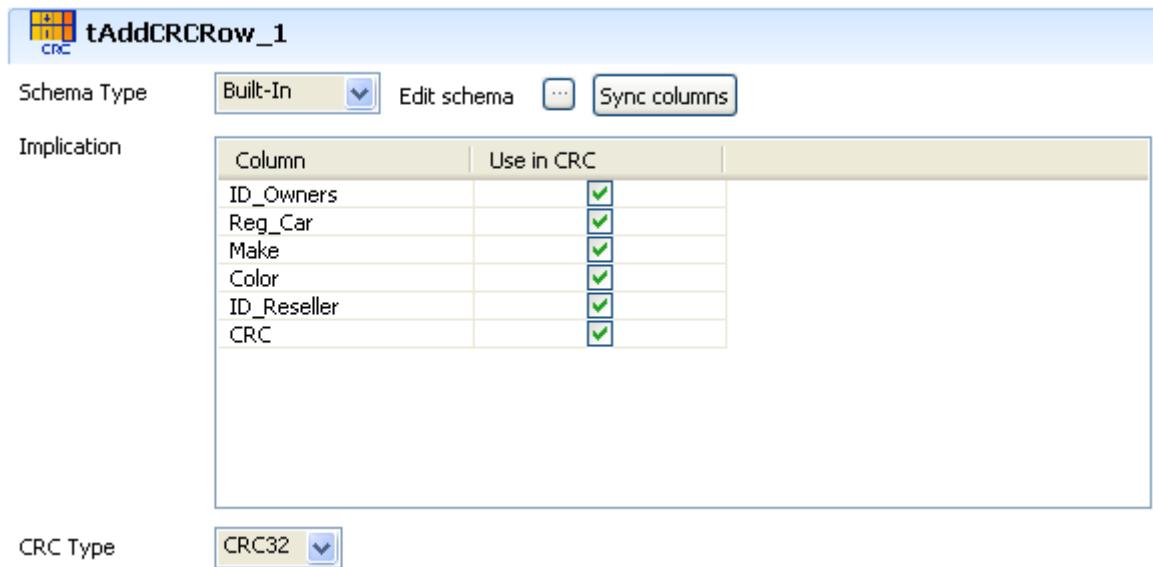
tAddCRCRow



- Click and drop the following components: **tFileInputDelimited**, **tAddCRCRow** and **tLogRow**.
- Connect them using a **Main row** connection.
- In the **tFileInputDelimited Properties** view, set the **File Name** path and all related properties in case these are not stored in the **Repository**.



- Create the schema through the **Edit Schema** button, in case the schema is not stored already in the **Repository**. In Java, mind the data type column and in case of Date pattern to be filled in, check out <http://java.sun.com/j2se/1.5.0/docs/api/index.html>.
- In the **tAddCRCRow Properties** view, check the Input flow columns to be used to calculate the CRC.



- Notice that a CRC column (read-only) has been added at the end of the schema.
- Select **CRC32** as **CRC Type** to get a longer surrogate key.



- In the **tLogRow** Properties view, check the **Print values in cells of a table** option to display the output data in a table on the Console.
- Then save your job and run it.

Components

tAddCRCRow

Starting job addcrc at 11:51 06/07/2007.					
tLogRow_1					
ID_Owners	Reg_Car	Make	Color	ID_Reseller	CRC
1	1301 DO 05	Citroen	gold	38	27510715125
2	2300 ZP 14	Citroen	blue	16	33211434545
3	4122 JI 74	Renault	yellow	36	11525215315
4	3395 QP 05	Citroen	yellow	51	14306204562
5	0029 OF 61	Toyota	red	37	10711350076
6	4287 YU 44	Citroen	blue	43	25561510712
7	7119 CQ 97	Honda	yellow	65	10136571035
8	3764 PA 47	Renault	orange	30	31723253034
9	9939 CJ 88	Mercedes	red	41	27451544441
10	7476 RV 09	Citroen	grey	34	27775721061
11	5287 BP 14	Toyota	green	27	2716661270
12	0750 OG 65	Toyota	green	8	23636130023
13	7577 ZQ 59	Volkswagen	purple	55	37277337005

An additional CRC Column has been added to the schema calculated on all previously selected columns (in this case all columns of the schema).

tDB2Input



tDB2Input properties

The properties of the generic **tDBInput**, apply to the **tDB2Input** component, therefore the properties are not detailed again here.

For detailed information about generic **tDBInput** properties, see *tDBInput properties on page 136*.

Related scenarios

Related topics in generic **tDBInput** scenarios:

- *Scenario 1: Displaying selected data from DB table on page 137*
- *Scenario 2: Using StoreSQLQuery variable on page 138*

Related topic in **tContextLoad Scenario: Dynamic context use in MySQL DB insert on page 123**.

tDB2Output



tDB2Output properties

The properties of the generic component, **tDBOutput**, apply to the **tDB2Output** component, therefore the properties are not detailed again here.

For detailed information about generic **tDBOutput** properties, see *DBOutput properties on page 140*.

Related scenarios

For **tDB2Output** related topics, see

- **tDBOutput Scenario: Displaying DB output on page 142**
- **tMySQLOutput Scenario: Adding new column and altering data on page 261.**

tDB2Row

tDB2Row properties

The properties of the generic component, **tDBSQLRow**, apply to the **tDB2Row** component, therefore the properties are not detailed again here.

For detailed information about generic **tDBSQLRow** properties, see *tDBSQLRow properties on page 144*.

Related scenarios

For **tDB2Row** related topics, see:

- **tDBSQLRow Scenario 1: Resetting a DB auto-increment on page 145**
- **tMySQLRow Scenario: Removing and regenerating a MySQL table index on page 275.**

Components

tDBInput



tDBInput

tDBInput properties

Component family	Databases	
Function	tDBInput reads a database and extracts fields based on a query.	
Purpose	<p>tDBInput executes a DB query with a strictly defined order which must correspond to the schema definition. Then it passes on the field list to the next component via a Main row link. The following DB-specific input components share tDBInput's properties: tMysqlInput, tDB2Input, tOracleInput, tSybaseInput.</p> <p>Note: For performance reasons, specific Input component (if offered) should always be preferred to generic DBInput component.</p>	
Properties	<i>Property type</i>	Either Built-in or Repository
		Built-in: No existing property data is to be retrieved
		Repository: Select the Repository file where Properties are stored. The following fields are pre-filled in using fetched data.
	<i>DB driver</i>	Available DBMS are MySQL, MS SQL and PostgreSQL..
	<i>Host</i>	Database server IP address
	<i>Port</i>	Listening port number of DB server.
	<i>Database</i>	Name of the database
	<i>Username and Password</i>	DB user authentication data.
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields to be processed and passed on to the next component. The schema is either built-in or remotely stored in the Repository.
		Built-in: The schema is created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>
		Repository: The schema already exists and is stored in the Repository, hence can be reused. Related topic: <i>Setting a repository schema on page 52</i>
	<i>Query</i>	Enter your DB query paying particularly attention to properly sequence the fields in order to match the schema definition.

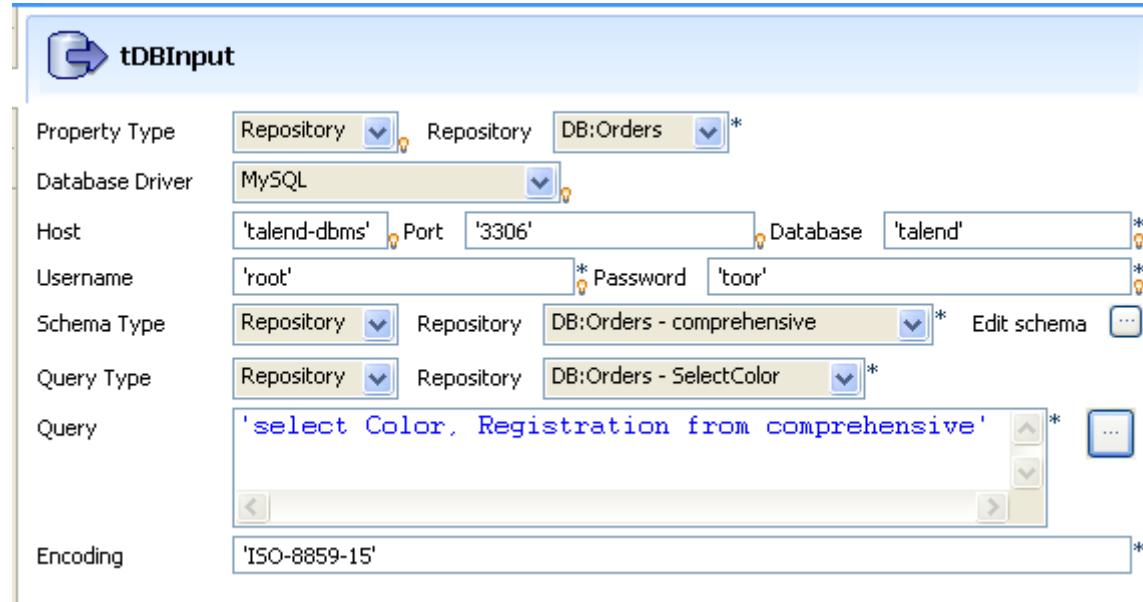
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.
Usage	This component offers the flexibility benefit of the DB query and covers all possibilities of SQL queries. More scenarios are available for specific DB Input components	

Scenario 1: Displaying selected data from DB table

The following scenario creates a two-component job, reading data from a database using a DB query and outputting delimited data into the standard output (console).



- Click and drop a **tDBInput** and **tLogRow** component from the Palette to the design workspace.
- Right-click on the **tDBInput** component and select *Row > Main*. Drag this main row link onto the **tLogRow** component and release when the plug symbol displays.
- Select the **tDBInput** again so the properties tab shows up, and define the properties:



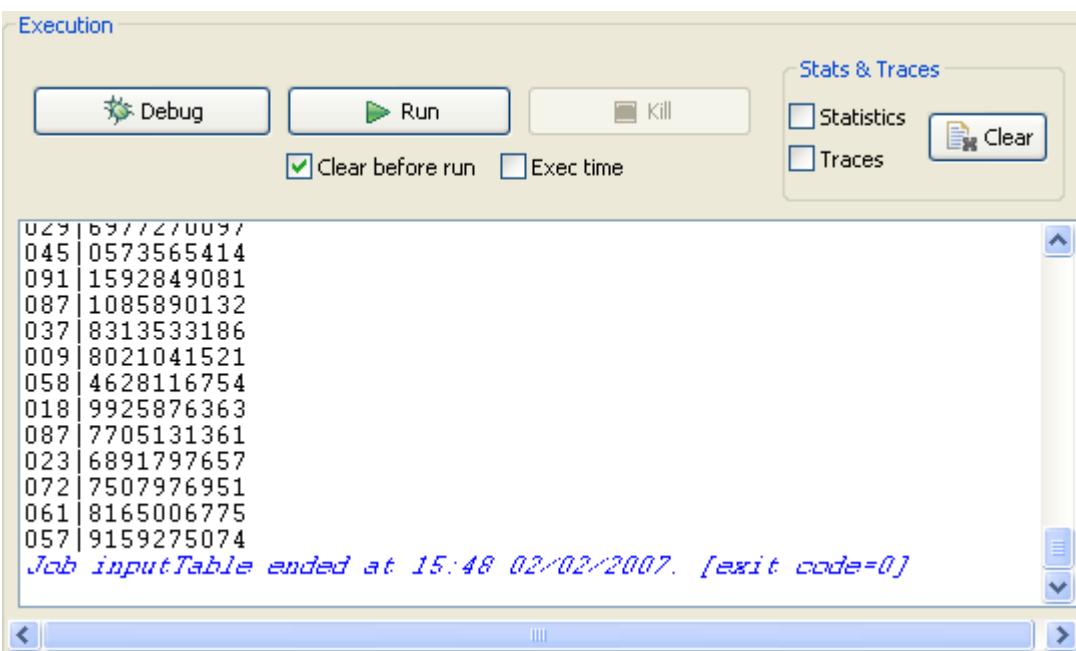
- The component property data are **Built-In** for this scenario.
- Select **Mysql** as database driver.
- Fill in the DB connection data in Host, Port, Database name, User name and password fields.

Components

tDBInput

- The schema is **Built-In**. This means that it is available for this job and on this station only.
- Click on **Edit Schema** and create a 2-column description including shop code and sales
- Type in the query making sure it includes all columns in the same order as defined in the Schema. In this case, as we'll select all columns of the schema, the asterisk symbol makes sense.
- Enter the Encoding for information only. And click on the second component to define it.
- Enter the fields separator. In this case, a pipe separator.
- Now go to the **Run Job** tab, and click on **Run** to execute the job.

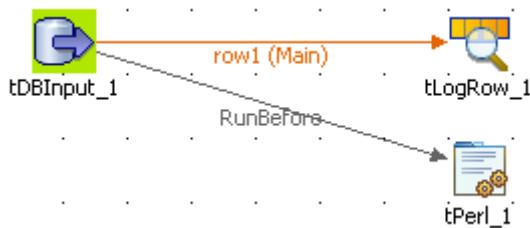
The DB is parsed and queried data is extracted and passed on to the Job log console. You can view the output file straight on the console.



Scenario 2: Using StoreSQLQuery variable

StoreSQLQuery is a variable that can be used to debug a tDBInput scenario which does not operate correctly. It allows you to dynamically feed the SQL query set in your tDBInput component.

- Use the same scenario as scenario 1 above and add a third component, **tPerl**.
- Connect **tDBInput** component to **tPerl** component using a trigger connection of **ThenRun** type. In this case, we want the **tDBInput** to run before the **tPerl** component.



- Set both **tDBInput** and **tLogRow** component as in **tDBInput** scenario 1.
- Click anywhere on the design workspace to display the **Context** property panel.
- Create a new parameter called explicitly **StoreSQLQuery**. Enter a default value of 1. This value of 1 means the **StoreSQLQuery** is “true” for a use in the **QUERY** global variable.
- Click on the **tPerl** component to display the Properties. Enter the command **Print** to display the query content, press **Ctrl+Space bar** to access the variable list and select the global variable **QUERY**.



- Go to your **Run tab** and execute the job.
- The query entered in the **tDBInput** component shows at the end of the job results, on the log:

```

silver|3962 SM 31|||
orange|6398 UJ 08|||
select Color, Registration from comprehensive
Job RegAndColor ended at 18:32 14/02/2007. [exit code=0]

```

Components

tDBOutput

tDBOutput



DBOutput properties

Component family	Databases	
Function	tDBOutput writes, updates, makes changes or suppresses entries in a database.	
Purpose	tDBOutput executes the action defined on the table and/or on the data contained in the table, based on the flow incoming from the preceding component in the job. The following DB-specific output components share tDBOutput's properties: tMysqlInput , tDB2Input , tOracleInput , tSybaseInput . Note: Specific Input component should always be preferred to generic DBInput component.	
Properties	<i>Property type</i>	Either Built-in or Repository.
		Built-in: No existing property data is to be retrieved
		Repository: Select the Repository file where Properties are stored. The following fields are pre-filled in using fetched data.
	<i>Database driver</i>	Available DBMS are MySQL, MS SQL and PostgreSQL.
	<i>Host</i>	Database server IP address
	<i>Port</i>	Listening port number of DB server.
	<i>Database</i>	Name of the database
	<i>Username and Password</i>	DB user authentication data.
	<i>Table</i>	Name of the table to be written. Note that only one table can be written at a time
<i>In Java, use tCreateTable as substitute for this function..</i>	<i>Action on table</i>	On the table defined, you can perform one of the following operations: None: No operation carried out Drop and create the table: The table is removed and created again Create a table: The table doesn't exist and gets created. Clear a table: The table content is deleted

	<i>Action on data</i>	On the data of the table defined, you can perform: Insert: Add new entries to the table. If duplicates are found, job stops. Update: Make changes to existing entries Insert or update: Add entries or update existing ones. Update or insert: Update existing entries or create it if non existing Delete: Remove entries corresponding to the input flow.
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields to be processed and passed on to the next component. The schema is either built-in or remotely stored in the Repository.
		Built-in: The schema is created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>
		Repository: The schema already exists and is stored in the Repository, hence can be reused. Related topic: <i>Setting a repository schema on page 52</i>
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.
	<i>Additional Columns</i>	This option is not offered if you create (with or without drop) the Db table. This option allows you to perform actions on columns, which are not insert, nor update or delete actions or requires a particular preprocessing.
		Name: Type in the name of the schema column to be altered or inserted as new column
		SQL expression: Type in the SQL statement to be executed in order to alter or insert the relevant column data.
		Position: Select Before, Replace or After, following the action to be performed on the reference column.
		Reference column: Type in a column of reference that the tDBOutput can use to place or replace the new or altered column.
	Commit every	Number of rows to be completed before committing batches of rows together into the DB. This option ensures transaction quality (but not rollback) and above all better performance on executions.

Components

tDBOutput

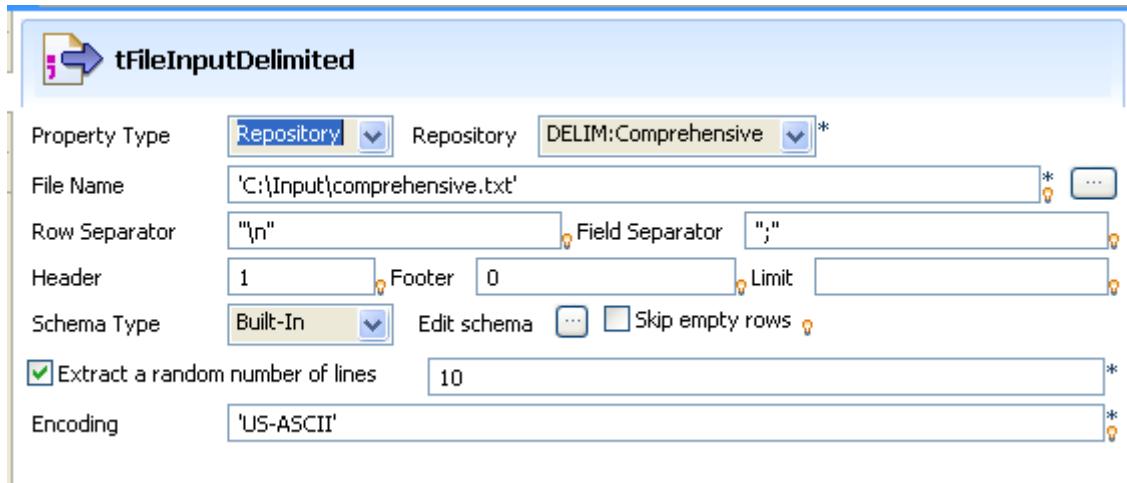
Usage	This component offers the flexibility benefit of the DB query and covers all possibilities of SQL queries.
-------	--

Scenario: Displaying DB output

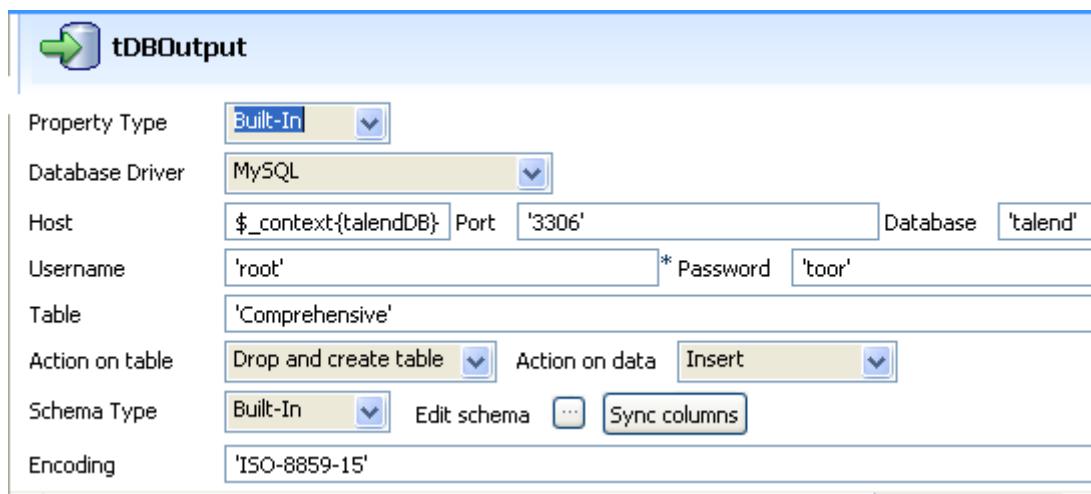
This following scenario is a three-component job aiming at creating a new table in the database defined and filling it with data. The **tFileInputDelimited** passes on the Input flow to the **tDBOutput** component. As the content of a DB is not viewable as such, a **tLogRow** component is used to display the main flow on the **Run Job** console.



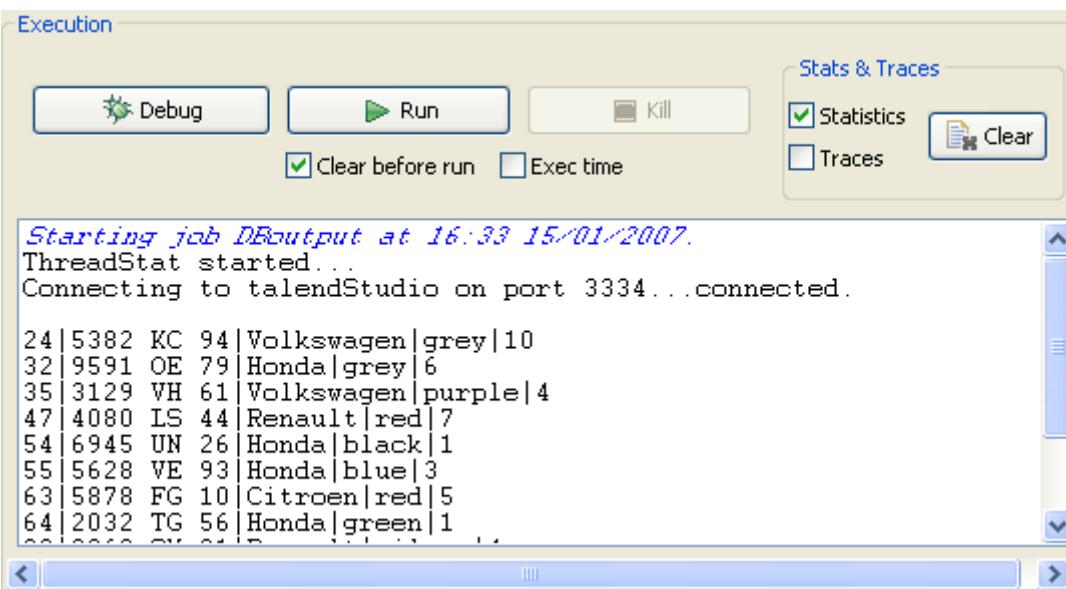
- First click and drop the three components required for this job.
- On the **Properties** tab of **tFileInputDelimited**, define the input flow parameters. In this use case, the file contains cars' owner id, makes, color and registration references organised as follows: semi-colon as field separator, carriage return as row separator. The input file contains a header row to be considered in the schema. If this file is already described in your metadata, you can retrieve the properties by selecting the relevant repository entry list.



- And also, if your schema is already loaded in the Repository, select **Repository** as **Schema type** and choose the relevant metadata entry in the list. If you haven't defined the schema already, define the data structure in the built-in schema you edit.
- Restrict the extraction to 10 lines, for this example.
- Then define the **tDBOutput** component to configure the output flow. Select the database to connect to. Note that you can store all the database connection details in different context variables. For more information about how to create and use context variables , see *Defining a job design context and related variables on page 94*.



- Fill in the table name in the **Table** field. Then select the operations to be performed:
- As **Action on table**, select **Drop and create table** in the list. This allows you to overwrite the possible existing table with the new selected data. Alternately you can insert only extra rows into an existing table, but note that duplicate management is not supported natively. See *tUniqRow properties on page 334* for further information.
- As **Action on data**, select **Insert**. The data flow incoming as input will be thus added to the selected table.
- To view the output flow easily, connect the DBOutput component to an tLogRow component. Define the field separator as a pipe symbol. Press **F6** to execute the job.
- As the processing can take some time to reach the **tLogRow** component, we recommend you to enable the **Statistics** functionality on the **Run Job** console.



Related topic: *tMysqlOutput properties on page 261*

Components

tDBSQLRow

tDBSQLRow



tDBSQLRow properties

Component family	Databases	
Function	tDBSQLRow is the generic component for database query. It executes the SQL query stated onto the specified database. The row suffix means the component implements a flow in the job design although it doesn't provide output. The following DB-specific row components share tDBSQLRow's properties: tMySQLRow , tDB2Row , tOracleRow , tSybaseRow .	
Purpose	Depending on the nature of the query and the database, tDBSQLRow acts on the actual DB structure or on the data (although without handling data). The SQLBuilder tool helps you write easily your SQL statements.	
Properties	<i>Property type</i>	Either Built-in or Repository.
		Built-in: No existing property data is to be retrieved
		Repository: Select the Repository file where Properties are stored. The following fields are pre-filled in using fetched data.
	<i>DB driver</i>	Available DBMS are MySQL, MS SQL and PostgreSQL. or other specific DBMS
	<i>Host</i>	Database server IP address
	<i>Port</i>	Listening port number of DB server.
	<i>Database</i>	Name of the database
	<i>Username and Password</i>	DB user authentication data.
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields to be processed and passed on to the next component. The schema is either built-in or remotely stored in the Repository.
		Built-in: The schema is created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>
		Repository: The schema already exists and is stored in the Repository, hence can be reused. Related topic: <i>Setting a repository schema on page 52</i>
	<i>Query type</i>	Either Built-in or Repository.
		Built-in: Fill in manually the query statement or build it graphically using SQLBuilder

		Repository: Select the relevant query stored in the Repository. The Query field gets accordingly filled in.
	<i>Query</i>	Enter your DB query paying particularly attention to properly sequence the fields in order to match the schema definition.
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.
Usage	This component offers the flexibility benefit of the DB query and covers all possibilities of SQL queries.	
		Use the relevant DBRow component according to the DB type you use. Most of the database have their specific DBRow component.

Scenario 1: Resetting a DB auto-increment

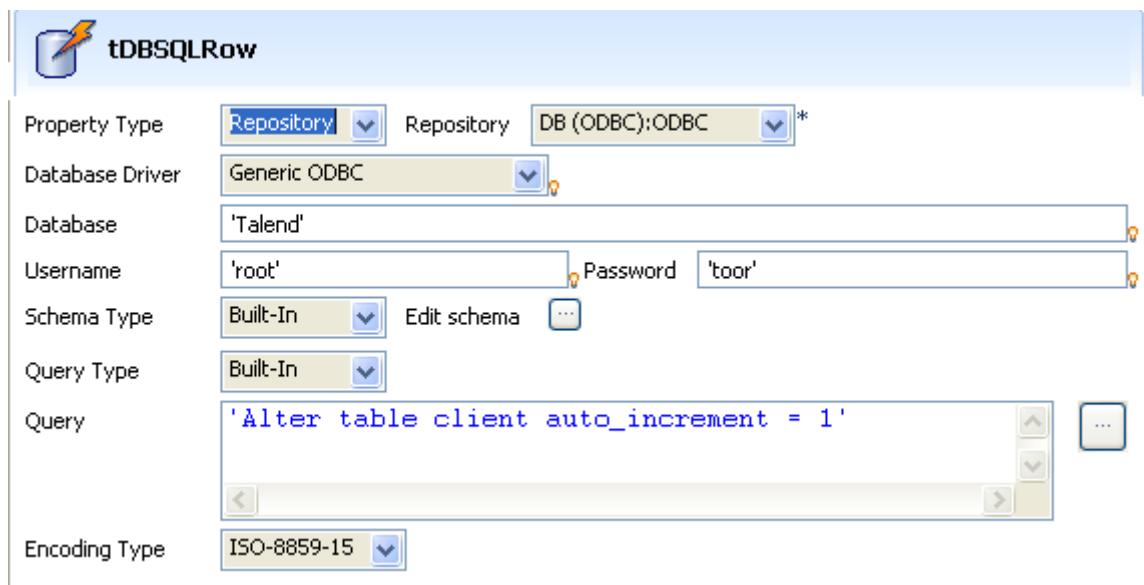
This scenario describes a single component job which aims at reinitializing the DB auto-increment to 1. This job has no output and is generally to be used before running a script.



- Drag and drop a **tDBSQLRow** component from the Palette to the Job designer.
- On the **Properties** panel, fill in the DB connection properties.

Components

tDBSQLRow



- The general connection information to the database is stored in the Repository. The **Database Driver** is a generic ODBC driver.
- The **Schema type** is built-in for this job and describes the Talend database structure. The schema doesn't really matter for this particular instance of job as the action is made on the table auto-increment and not on data.
- The **Query type** is also built-in. Click on the three dot button to launch the SQLbuilder editor, or else type in directly in the Query area:
Alter table <TableName> auto_increment = 1
- Then click **OK** to validate the **Properties**. Then press **F6** to run the job.

The database autoincrement is reset to 1.

Related topics: *tMysqlRow properties on page 275*.



tDenormalize

tDenormalize Properties

Component family	Processing	 
Function	Denormalizes the input flow based on one column.	
Purpose	tDenormalize helps synthesize the input flow.	
Properties	<i>Schema type</i> and <i>Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository. In this component, the schema is read-only.
		Built-in: The schema will be created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>
 Perl feature	<i>Column to denormalize</i>	Select the column from the input flow which the normalization is based on (included in key)
 Java feature	<i>Group by</i>	Select one or several columns to be grouped. We recommend to remove unused columns from the schema before processing.
	<i>Separator</i>	Enter the separator which will delimit data in the denormalized flow.
	<i>Deduplicate items</i>	Removes duplicates when concatenating denormalized values.
Usage	This component can be used as intermediate step in a data flow.	
Limitation	n/a	

Scenario 1: Denormalizing on one column in Perl

This scenario illustrates a Perl job denormalizing one column in a delimited file.



- Click and drop the following components: **tFileInputDelimited**, **tDenormalize**, **tLogRow**.
- Connect the components using **Row main** connections.

Components

tDenormalize

- On the **tFileInputDelimited** properties panel, set the filepath to the file to be denormalized.



- Define the **Header**, **Row Separator** and **Field Separator** parameters.
- The input file schema is made of two columns, *Fathers* and *Children*.

	Fathers&Sons
1	Fathers;Children;
2	Pierrick;Erwann;
3	Fabrice;Martin;
4	Stéphane;Agathe;
5	Pierrick;Tiphaine;
6	Robert;Manon;
7	Stéphane;Clémence;
8	Richard;Roméo;
9	Mickael;Océane;
10	

- In the **Properties** of **tDenormalize**, define the column that contains multiple values to be grouped.
- In this use case, the column to denormalize is *Children*.



- Set the **Item Separator** to separate the grouped values. Beware as only one column can be denormalized.
- Check the **Deduplicate items**, if you know that some values to be grouped are strictly identical.

- Save your job and run it.

```
Starting job Denormalize at 20:39 03/07/2007.
Richard| Roméo
Stéphane| Agathe, Clémence
Mickael| Océane
Pierrick| Erwann, Tiphaine
Robert| Manon
Fabrice| Martin
Job Denormalize ended at 20:39 03/07/2007. [exit]
```

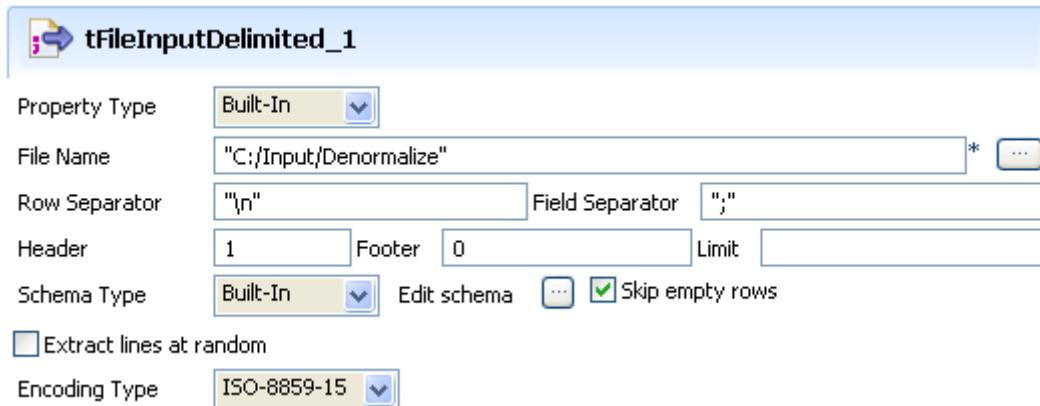
All values from the column *Children* (set as column to denormalize) are grouped by their *Fathers* column. Values are separated by a comma.

Scenario 2: Denormalizing on multiple columns in Java

This scenario illustrates a Java job denormalizing two columns from a delimited file.



- Click and drop the following components: **tFileInputDelimited**, **tDenormalize**, **tLogRow**.
- Connect all components using a **Row main** connection.
- On the **tFileInputDelimited Properties** panel, set the filepath to the file to be denormalized.



- Define the **Row** and **Field** separators, the **Header** and other information if required.
- The file schema is made of four columns including: **Name**, **FirstName**, **HomeTown**, **WorkTown**.

Components

tDenormalize

	Denormalize
1	Name;FirstName;HomeCity;WorkCity
2	Pitt;Brad;Beverly Hills;Los Angeles
3	Pitt;Brad;Paris;London
4	Joli;Angelina;Berlin;Berlin
5	Joli;Angelina;Berlin;Los Angeles
6	Joli;Angelina;Los Angeles;Los Angeles
7	Willis;Bruce;Paris;Los Angeles
8	Willis;Bruce;Paris;Madrid
9	Willis;Bruce;Madrid;Madrid
10	Willis;Bruce;Roma;Dublin
11	Moore;Demi;New York;Paris
12	Moore;Demi;Rio de Janeiro;Los Angeles
13	

- In the **tDenormalize** component **Properties**, select the columns that contain the repetition. These are the column which are meant to occur multiple times in the document. In this use case, *FirstName* and *Name* are the columns against which the denormalization is performed.
- Add as many line to the table as you need using the plus button. Then select the relevant columns in the drop-down list.

Group by

Input column
Name
FirstName

- Define the delimiter for concatenated values. In this case, the comma is used.
- Save your job and run it.

```
Starting job denormalize at 21:03 03/07/2007.  
Pitt|Brad|Beverly Hills,Paris|Los Angeles, London  
Willis|Bruce|Paris,Paris,Madrid,Roma|Los  
Angeles,Madrid,Madrid,Dublin  
Moore|Demi|New York,Rio de Janeiro|Paris,Los Angeles  
Joli|Angelina|Berlin,Berlin,Los Angeles|Berlin,Los Angeles,Los  
Angeles  
Job denormalize ended at 21:03 03/07/2007. [exit code=0]
```

- The result shows the denormalized values concatenated using a comma.

- Back to the tDenormalize components Properties, check the Deduplicate box to remove the duplicate occurrences.
- Save your job again and run it.

```
Starting job denormalize at 21:20 03/07/2007.  
Pitt|Brad|Paris,Beverly Hills|Los Angeles,London  
Willis|Bruce|Madrid,Paris,Roma|Madrid,Los Angeles,Dublin  
Moore|Demi|Rio de Janeiro,New York|Los Angeles,Paris  
Joli|Angelina|Los Angeles,Berlin|Los Angeles,Berlin  
Job denormalize ended at 21:20 03/07/2007. [exit code=0]
```

This time, the console shows the results with no duplicate instances.

tDie

Both **tDie** and **tWarn** components are closely related to the **tLogCatcher** component. They generally make sense when used alongside a **tLogCatcher** in order for the log data collected to be encapsulated and passed on to the output defined.

tDie properties

Component family	Log & Error	 
Function	Kills the current job. Generally used with a tCatch for log purpose.	
Purpose	Triggers the tLogCatcher component for exhaustive log before killing the job.	
	<i>Die message</i>	Enter the message to be displayed before the job is killed.
	<i>Error code</i>	Enter the error code if need be, as an integer
	<i>Priority</i>	Set the level of priority, as an integer
Usage	Cannot be used as a start component.	
Limitation	n/a	

Related scenarios

For uses cases in relation with **tDie**, see **tLogCatcher** scenarios:

- *Scenario 1: warning & log on entries on page 226*
- *Scenario 2: log & kill a job on page 228*

tDTDValidator



tDTDValidator Properties

Component family	XML	
Function	Validates the XML input file against a DTD file and sends the validation log to the defined output.	
Purpose	Helps at controlling data and structure quality of the file to be processed	
Properties	Schema type and <i>Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields to be processed and passed on to the next component. The schema is either built-in or remotely stored in the Repository but in this case, the schema is read-only. It contains standard information regarding the file validation.
	<i>DTD file</i>	Filepath to the reference DTD file.
	<i>XML file</i>	Filepath to the XML file to be validated.
	<i>If XML is valid, display</i> <i>If XML is not valid detected, display</i>	Type in a message to be displayed in the Run Job console based on the result of the comparison.
	<i>Print to console</i>	Check the box to display the validation message
Usage	This component can be used as standalone component but it is usually linked to an output component to gather the log data.	
Limitation	n/a	

Scenario: Validating xml files

This scenario describes a job that validates several files from a folder and outputs the log information for the invalid files into a delimited file.



- Click and drop the following components from the Palette: **tFileList**, **tDTDValidator**, **tMap**, **tFileOutputDelimited**.

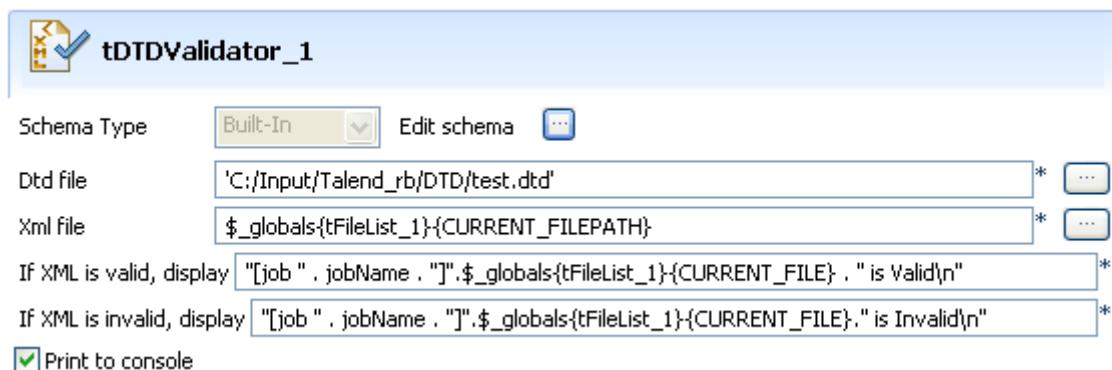
Components

tDTDValidator

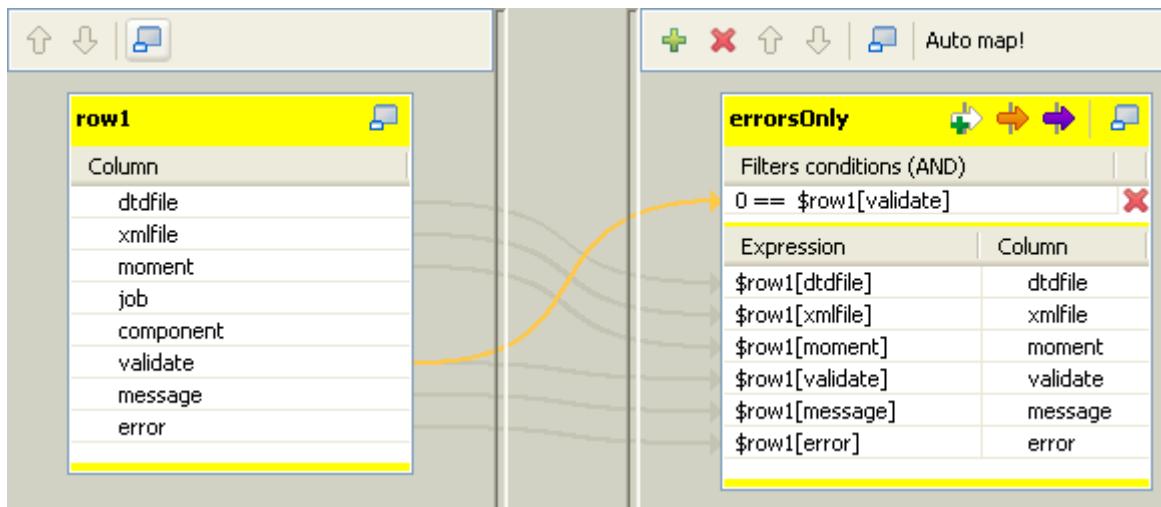
- Connect the **tFileList** to the **tDTDValidator** with an **Iterate** link and the remaining component using a **main** row.
- Set the tFileList component Properties, to fetch an XML file from a folder.



- Change the **Filemask** to `*.xml`. Mind the quotes depending on the Perl or Java version you are using.
- Uncheck the **Case Sensitive** box.
- In the **tDTDValidator** component **Properties**, the schema is read-only as it contains standard log information related to the validation process.
- Set the **DTD file** to be used as reference.



- Press **Ctrl+Space bar** to access the variable list. In the XML file field, select the current filepath global variable : `$_globals{tFileList_1}{CURRENT_FILEPATH}` (in Perl)
- In the various messages to display in the Run Job tab console, use the jobName to recall the job name tag. Recall the filename using the relevant global variable: `$_globals{tFileList_1}{CURRENT_FILE}`. Mind the Perl or Java operators such as the dot or the plus sign to build your message.
- Check the **Print to Console** box.
- In the **tMap** component, drag and drop the information data from the standard schema that you want to pass on to the output file.



- Once the Output schema is defined as required, add a filter condition to only select the log information data when the XML file is invalid.
- Follow the best practice by typing first the wanted value for the variable, then the operator based on the type of data filtered then the variable that should meet the requirement. In this case (in Java and Perl): `0 == $row1[validate]`
- Then connect (if not already done) the **tMap** to the **tFileOutputDelimited** component using a main row. Name it as relevant, in this example: **errorsOnly**.
- In the **tFileOutputDelimited Properties**, Define the destination filepath, the field delimiters and the encoding.
- Save your job and press **F6** to run it.

```

Starting job DTDValidate at 15:29 21/06/2007.
[job jobName]test.xml is Invalid
[job jobName]test2.xml is Invalid
[job jobName]test3.xml is Invalid
[job jobName]test4.xml is Invalid
[job jobName]test5.xml is Invalid
[job jobName]test6.xml is Invalid
Job DTDValidate ended at 15:29 21/06/2007. [exit code=0]

```

On the Run Job console the messages defined display for each of the invalid files. At the same time the output file is filled with log data.

Components

tELTMysqlInput



tELTMysqlInput

All three ELT MySQL components are closely related together in regard to their operating condition. These components should be used to handle MySQL DB schemas to generate Insert statements including clauses, which are to be executed to the DB output table defined.

tELTMysqlInput properties

Component family	ELT	
Function	Provides the table schema to be used for the SQL statement to execute.	
Purpose	Allows to add as many Input tables as required for the most complicated Insert statement.	
Properties	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the nature and number of fields to be processed. The schema is either built-in or remotely stored in the Repository. The Schema defined is then passed on to the ELT Mapper to be included to the Insert SQL statement.
		Built-in: The schema is created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>
		Repository: The schema already exists and is stored in the Repository, hence can be reused. Related topic: <i>Setting a repository schema on page 52</i>
Usage	<p>tELTMysqlInput is to be used along with the tELTMysqlMap. Note that the Output link to be used with these components has to reflect faithfully the name of the table</p> <p>Note: Note that the ELT components do not handle actual data flow but only schema information.</p>	

Related scenarios

For uses cases in relation with **tELTMysqlInput**, see **tELTMysqlMap** scenarios:

- *Scenario 1: Aggregating table columns and filtering on page 160*
- *Scenario 2: ELT using Alias table on page 163*



tELTMySQLMap

All three ELT MySQL components are closely related together in regard to their operating condition. These components should be used to handle MySQL DB schemas to generate Insert statements including clauses, which are to be executed to the DB output table defined.

tELTMysqlMap properties

Component family	ELT	 
Function	Helps to graphically build the SQL statement using the table provided as input.	
Purpose	Uses the tables provided as input, to feed the parameter in the built statement. The statement can include inner or outer joins to be implemented between tables or between one table and its aliases.	
Properties	<i>Property type</i>	Either Built-in or Repository.
		Built-in: No existing property data is to be retrieved
		Repository: Select the Repository file where Properties are stored. The following fields are pre-filled in using fetched data.
	<i>Host</i>	Database server IP address
	<i>Port</i>	Listening port number of DB server.
	<i>Database</i>	Name of the database
	<i>Username and Password</i>	DB user authentication data.
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.
	<i>Preview</i>	The preview is an instant shot of the Mapper data. It becomes available when Mapper properties have been filled in with data. The preview synchronization takes effect only after saving changes.
	<i>Map editor</i>	The ELT Map editor allows you to define the output schema as well as build graphically the SQL statement to be executed.
Usage	tELTMysqlMap is used along with a tELTMysqlInput and tELTMysqlOutput . Note that the Output link to be used with these components has to reflect faithfully the name of the tables. Note: Note that the ELT components do not handle actual data flow but only schema information.	

Connecting ELT components

The ELT components do not handle any data as such but table schema information that will be used to build the SQL query to execute.

Therefore the only connection required to connect these components together is a simple link.

Note: The output name you give to this link when creating it should always be the exact name of the table to be accessed as this parameter will be used in the SQL statement generated.

Related topic: *Link connection on page 48*

Mapping and joining tables

In the ELT Mapper, you can select specific columns from input schemas and include them in the output schema.

- As you would do it in the regular Mapper editor, simply drag & drop the content from the input schema towards the output table defined.
- Use the Ctrl and Shift keys for multiple selection of contiguous or non contiguous table columns.

You can implement explicit joins to retrieve various data from different tables.

- Click on the **Join** drop-down list and select the relevant explicit join.
- Possible joins include: **Inner Join**, **Left Outer Join**, **Right Outer Join** or **Full Outer Join** and **Cross Join**.
- By default the **Inner Join** is selected.

You can also create **Alias** tables to retrieve various data from the same table.

- In the Input area, click on the plus (+) button to create an Alias.
- Define the table to base the alias on.
- Type in a new name for the alias table, preferably not the same as the main table.

Adding where clauses

You can also restrict the Select statement based on a Where clause. Click on the **Add filter row** button at the top of the output table and type in the relevant restriction to be applied.

Make sure that all input components are linked correctly to the ELT Map component to be able to implement all inclusions, joins and clauses.

Generating the SQL statement

The mapping of elements from the input schemas to the output schemas create instantly the corresponding Select statement.

Components

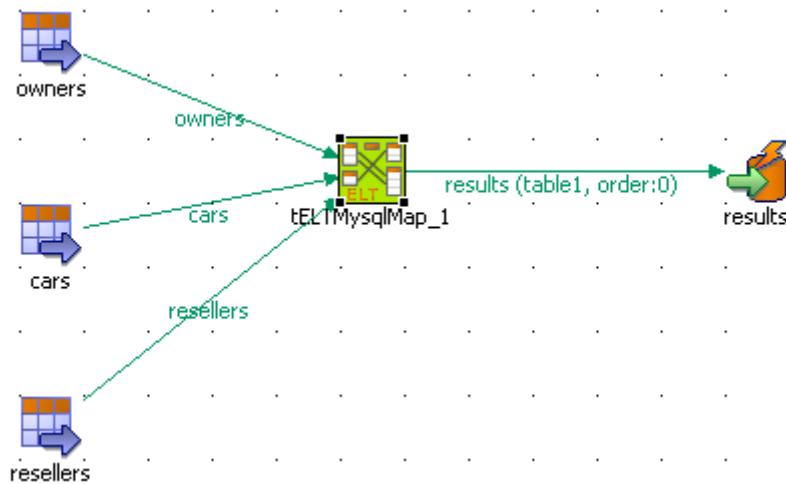
tELTMysqlMap

```
Schema editor Expression editor Generated SQL Select query for 'table1' output
SELECT
owners.ID_Owner, owners.Name_Customer, owners.ID_Insurance, cars.Reg_Car, cars.Make, cars.Color,
resellers.Name_Reseller, resellers.City
FROM
owners INNER JOIN cars ON( cars.ID_Owners = owners.ID_Owner )
INNER JOIN resellers ON( resellers.ID_Reseller = cars.ID_Reseller )
WHERE resellers.City ='West Coast City'
```

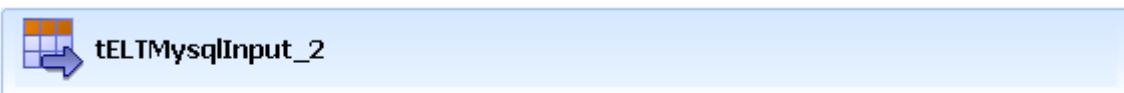
The clause are also included automatically.

Scenario1: Aggregating table columns and filtering

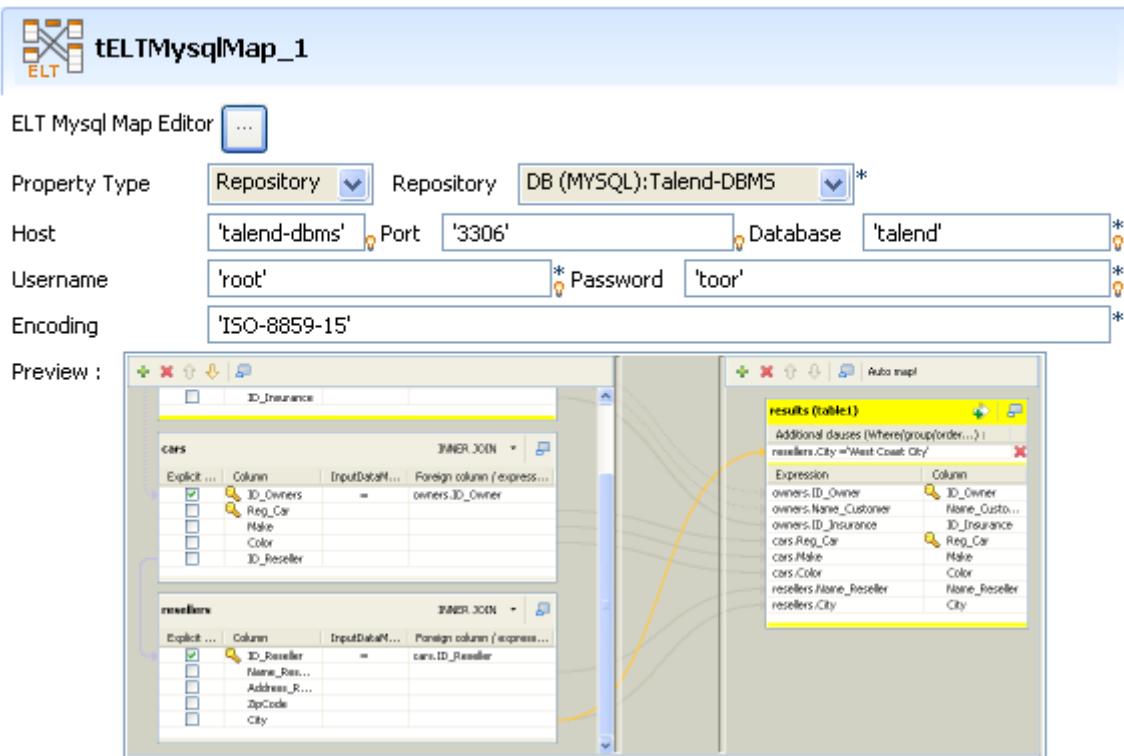
This scenario describes a job gathering together several Input DB table schemas and implementing a clause to filter the resulting output using an SQL statement.



- Click and drop the following components: **tELTMysqlInput**, **tELTMysqlMap**, **tELTMysqlOutput**.
- Three input components are required for this job.
- Connect the three ELT input components to the ELT mapper using links named following strictly the actual DB table names: *owners*, *cars* and *resellers*.
- Then connect the ELT mapper to the ELT Output component using another link that you call *results*.
- All three Input schemas are stored in the **Metadata** area of the **Repository**. They can therefore be easily retrieved.



- Click on the ELT mapper component to define the Database connection details.
- The Database connection details are stored in the Repository again

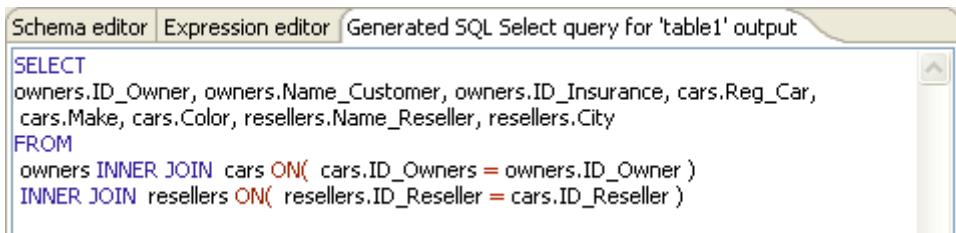


- The default encoding for Mysql database is retained.
- Launch the ELT Map editor to set up the join between Input tables
- Drag & drop the *ID_Owner* column from the *Owners* table to the corresponding column of the *Cars* table.
- Select **INNER JOIN** in the *Cars* table join list, and check **Explicit Join**, in front of the *ID_Owners*.
- Drag the *ID_Resellers* column from the *Cars* table to the *Resellers* table to set up the second join. Select here again **INNER JOIN** in the list of the *Resellers* table and check the **Explicit Join** box of the relevant column.
- Then select the columns to be aggregated into the output.
- Select all columns from the *Cars* and *Owners* table and only the *Reseller_Name* and *City* columns from the *Resellers* table.

Components

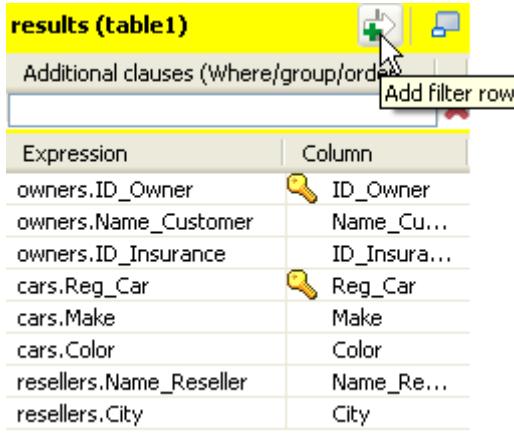
tELTMysqlMap

- Drag & drop them to the *Results* output table.
- The mapping displays in yellow and the joins display in dark violet.
- Click on the **Generated SQL Select query** tab to display the corresponding SQL Statement.



```
Schema editor Expression editor Generated SQL Select query for 'table1' output
SELECT
    owners.ID_Owner, owners.Name_Customer, owners.ID_Insurance, cars.Reg_Car,
    cars.Make, cars.Color, resellers.Name_Reseller, resellers.City
FROM
    owners INNER JOIN cars ON( cars.ID_Owners = owners.ID_Owner )
    INNER JOIN resellers ON( resellers.ID_Reseller = cars.ID_Reseller )
```

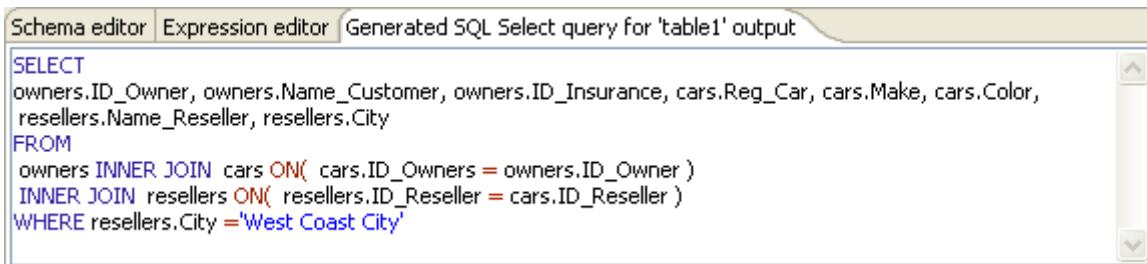
- Then implement a filter on the output table.
- Click on the **Add filter row** button of the output table.



results (table1)

Expression	Column
owners.ID_Owner	ID_Owner
owners.Name_Customer	Name_Cu...
owners.ID_Insurance	ID_Insura...
cars.Reg_Car	Reg_Car
cars.Make	Make
cars.Color	Color
resellers.Name_Reseller	Name_Re...
resellers.City	City

- Restrict the Select using a Where clause such as: resellers.City ='West Coast City'
- See the reflected where clause on the Generated SQL Select query tab.



```
Schema editor Expression editor Generated SQL Select query for 'table1' output
SELECT
    owners.ID_Owner, owners.Name_Customer, owners.ID_Insurance, cars.Reg_Car, cars.Make, cars.Color,
    resellers.Name_Reseller, resellers.City
FROM
    owners INNER JOIN cars ON( cars.ID_Owners = owners.ID_Owner )
    INNER JOIN resellers ON( resellers.ID_Reseller = cars.ID_Reseller )
WHERE resellers.City ='West Coast City'
```

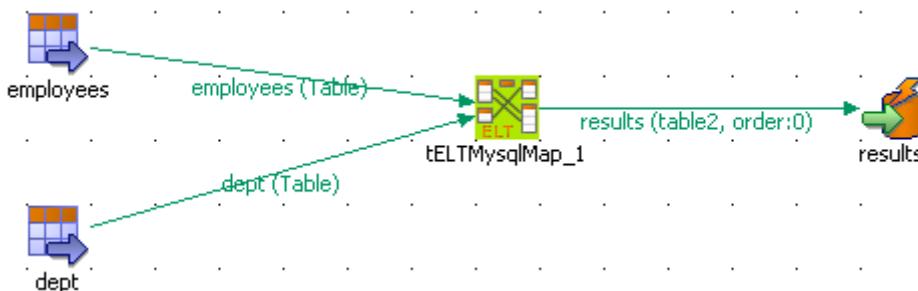
- Click **OK** to save the ELT Map setting.
- Define the ELT Output in the Properties Panel of the tELTMysqlOutput component.
- The **Action on table** is **Drop and create table** for this use case and the only action available on data in MySQL is **Insert**.
- The schema is to be synchronised with the tELTMysqlMap component as it aggregates several source schemas.

ID...	Name_Customer	ID_Insur...	Reg_Car	Make	Color	Name_Reseller	City
4	hirtken	ENX9366	6225 GT 57	Renault	green	Best Cars Shop	West Coast City
16	kennan	RTA8580	0601 SQ 67	Renault	blue	Cars & Pickup Re...	West Coast City
31	antken	QDG0199	5729 DJ 52	Renault	blue	Cars & Pickup Sp...	West Coast City
33	hirtken	MYA1613	0427 LR 72	Toyota	gold	Best Cars Specialist	West Coast City
34	nanant	XBM2459	7355 IB 28	BMW	purple	Best Cars Resale	West Coast City
39	nanneng	CSP8847	8402 JE 03	Volkswa...	blue	Cars & Pickup Re...	West Coast City
62	gallken	LZP1021	3940 ZW 19	BMW	green	All you need Outlet	West Coast City
65	oinele	ANX9956	6523 DY 26	Mercedes	gold	Best Cars Shop	West Coast City
76	nengle	SLG5853	2087 IW 01	Toyota	gold	Best Cars Outlet	West Coast City
77	boot	QFS6844	3795 GN 95	Toyota	blue	All you need Outlet	West Coast City
88	carbo	EHN3338	7752 OB 89	Mercedes	blue	Cars & Pickup Re...	West Coast City
89	bobouh	UMT0348	0462 FV 53	Toyota	purple	Best Cars Outlet	West Coast City

All selected data are inserted in the *results* table as specified in the SQL statement defined respecting the clause.

Scenario 2: ELT using Alias table

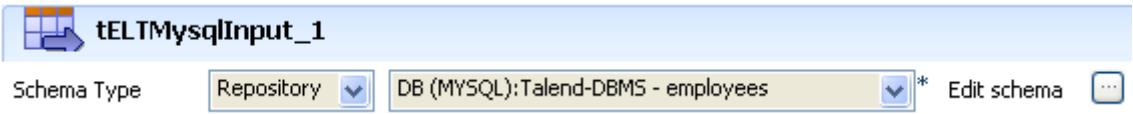
This scenario describes a job that uses an Alias table. The *employees* table contains all employees details as well as the ID of their respective manager, which are also considered as employees and hence included in the *employees* table. The *dept* table contains location and department information about the employees.



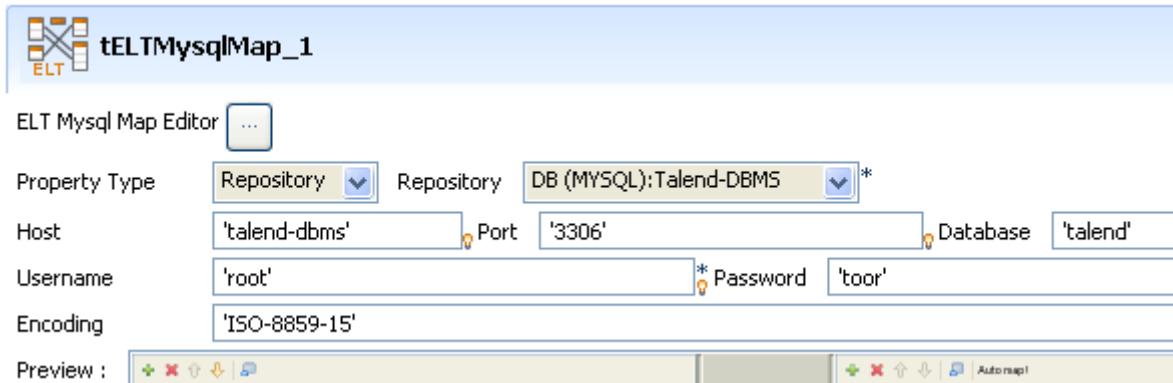
- Drag and drop **tELTMysqlInput** components to retrieve respectively the *employees* and *dept* table schemas.
- In this use case, both schemas are stored in the Repository and can therefore be easily retrieved.

Components

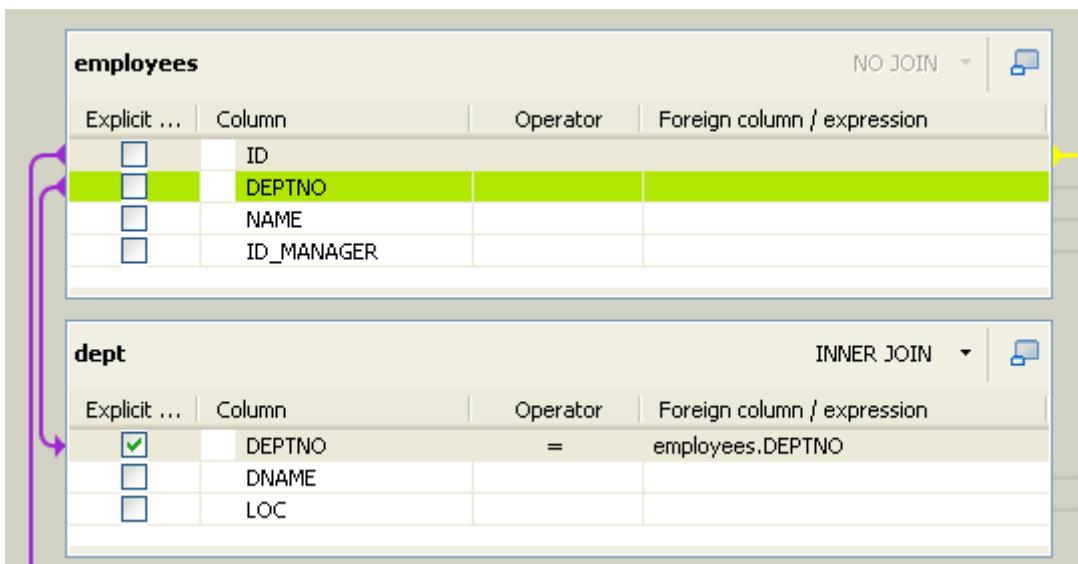
tELTMysqlMap



- Then select the tELTMysqlMap and define the Mysql database connection details.
- Here again the connection information is stored in the Repository's Metadata.



- Click on the button to launch the ELT Map editor.
- First make sure the correct input table is positioned at the top of the Input area, as the Joins are highly dependent on this position.
- In this example, the *employees* table should be on top.



- Drag and drop the *DeptNo* column from the *employees* table to the *dept* table to set up the **join** between both input tables.
- Check the **Explicit Join** box and define the join as an **Inner Join**.
- Then create the **Alias** table based on the *employees* table



- Name it *Managers* and click OK to display it as a new Input table in the ELT mapper.
- Drag & drop the *ID* column from the *employees* table to the *ID_Manager* column of the newly added *Managers* table.
- Check the **Explicit Join** box and define it as **Left Outer Join**, in order for results to be output even though they contain a **Null** value.

employees			
Explicit ...	Column	Operator	Foreign column / expression
<input type="checkbox"/>	ID		
<input type="checkbox"/>	DEPTNO		
<input type="checkbox"/>	NAME		
<input type="checkbox"/>	ID_MANAGER		

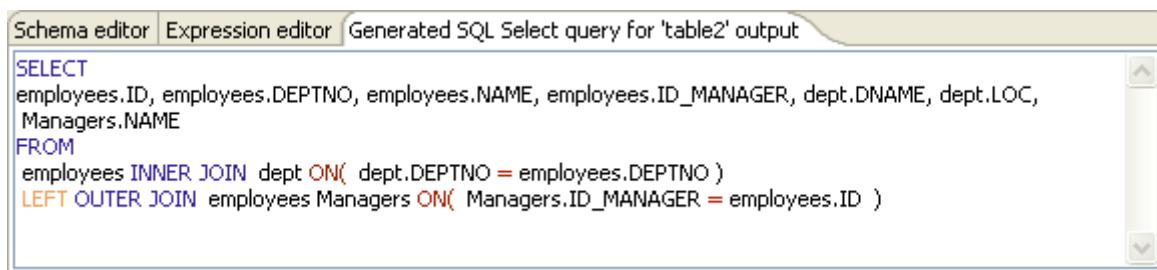
dept			
Explicit ...	Column	Operator	Foreign column / expression
<input checked="" type="checkbox"/>	DEPTNO	=	employees.DEPTNO
<input type="checkbox"/>	DNAME		
<input type="checkbox"/>	LOC		

Managers (alias of table 'employees')			
Explicit ...	Column	Operator	Foreign column / expression
<input type="checkbox"/>	ID		
<input type="checkbox"/>	DEPTNO		
<input type="checkbox"/>	NAME		
<input checked="" type="checkbox"/>	ID_MANAGER	=	employees.ID

- Drag and drop the content of both Input tables, *employees* and *dept*, as well as the *Name* column from the *Manager* table to the Output table.
- Click on the **Generated SQL Select query** tab to display the query to be executed.

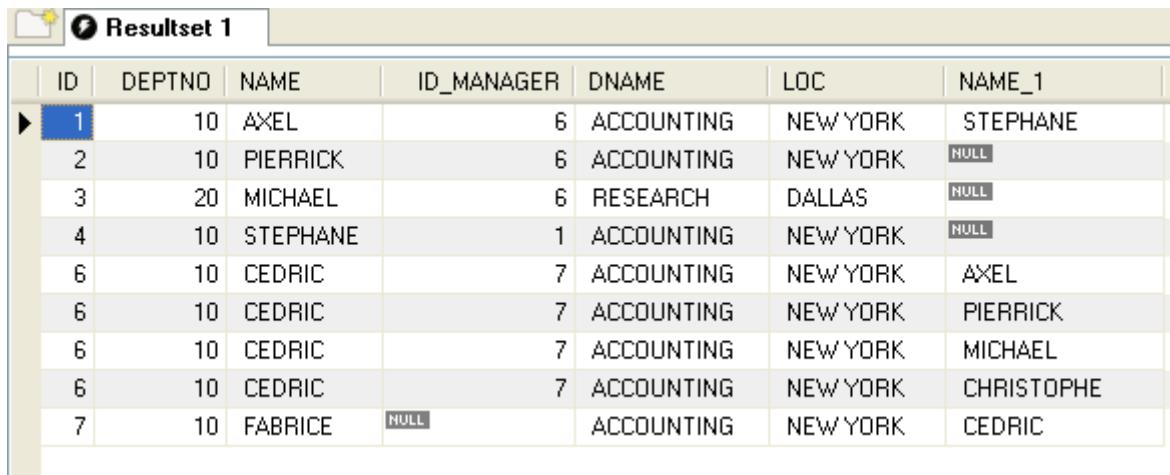
Components

tELTMySQLMap



```
Schema editor Expression editor Generated SQL Select query for 'table2' output
SELECT
employees.ID, employees.DEPTNO, employees.NAME, employees.ID_MANAGER, dept.DNAME, dept.LOC,
Managers.NAME
FROM
employees INNER JOIN dept ON( dept.DEPTNO = employees.DEPTNO )
LEFT OUTER JOIN employees Managers ON( Managers.ID_MANAGER = employees.ID )
```

- Then click on the Output component and define the **Action on data on Insert**.
- Make sure the schema is synchronized with the Output table from the ELT mapper before running the job through **F6** or via the toolbar.



ID	DEPTNO	NAME	ID_MANAGER	DNAME	LOC	NAME_1
1	10	AXEL	6	ACCOUNTING	NEW YORK	STEPHANE
2	10	PIERRICK	6	ACCOUNTING	NEW YORK	NULL
3	20	MICHAEL	6	RESEARCH	DALLAS	NULL
4	10	STEPHANE	1	ACCOUNTING	NEW YORK	NULL
6	10	CEDRIC	7	ACCOUNTING	NEW YORK	AXEL
6	10	CEDRIC	7	ACCOUNTING	NEW YORK	PIERRICK
6	10	CEDRIC	7	ACCOUNTING	NEW YORK	MICHAEL
6	10	CEDRIC	7	ACCOUNTING	NEW YORK	CHRISTOPHE
7	10	FABRICE	NULL	ACCOUNTING	NEW YORK	CEDRIC

The *Department* information as well as the *Employees* entries are coupled in the output, and the *Manager Name* could be retrieved via the explicit join.



tELTMySQLOutput

All three ELT MySQL components are closely related together in regard to their operating condition. These components should be used to handle MySQL DB schemas to generate Insert statements including clauses, which are to be executed to the DB output table defined.

tELTMysqlOutput properties

Component family	ELT	 
Function	Carries out the action on the table specified and inserts the data according to the output schema defined the ELT Mapper.	
Purpose	Executes the SQL Insert statement to the Mysql database	
Properties	<p>Action on table</p> <p> <i>In Java, use tCreateTable as substitute for this function.</i></p>	<p>On the table defined, you can perform one of the following operations:</p> <p>None: No operation carried out</p> <p>Drop and create the table: The table is removed and created again</p> <p>Create a table: The table doesn't exist and gets created. If the table exists, an error is generated and the job is stopped.</p> <p>Create table if doesn't exist: Create the table if needed and carries out the action on data anyway.</p> <p>Clear a table: The table content is deleted</p>
	<p>Action on data</p>	<p>On the data of the table defined, you can perform the following operation:</p> <p>Insert: Add new entries to the table. If duplicates are found, job stops.</p> <p>Note that in Mysql ELT, only Insert operation is available.</p>
	<p>Schema type and Edit Schema</p>	<p>A schema is a row description, i.e., it defines the number of fields to be processed and passed on to the next component. The schema is either built-in or remotely stored in the Repository.</p>
		<p>Built-in: The schema is created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i></p>
		<p>Repository: The schema already exists and is stored in the Repository, hence can be reused. Related topic: <i>Setting a repository schema on page 52</i></p>
	<p>Encoding</p>	<p>Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.</p>
Usage	<p>tELTMysqlOutput is to be used along with the tELTMysqlMap. Note that the Output link to be used with these components has to reflect faithfully the name of the table.</p> <p>Note: Note that the ELT components do not handle actual data flow but only schema information.</p>	

Related scenarios

For uses cases in relation with **tELTMysqlOutput**, see **tELTMysqlMap** scenarios:

- *Scenario 1: Aggregating table columns and filtering on page 160*
- *Scenario 2: ELT using Alias table on page 163*



tELTOracleInput

All three ELT Oracle components are closely related together in regard to their operating condition. These components should be used to handle Oracle DB schemas to generate Insert, Update or Delete statements including clauses, which are to be executed to the DB output table defined.

tELTOracleInput properties

Component family	ELT	
Function	Provides the table schema to be used for the SQL statement to execute.	
Purpose	Allows to add as many Input tables as required for the most complicated Insert statement.	
Properties	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the nature and number of fields to be processed. The schema is either built-in or remotely stored in the Repository. The Schema defined is then passed on to the ELT Mapper to be included to the Insert SQL statement.
		Built-in: The schema is created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>
		Repository: The schema already exists and is stored in the Repository, hence can be reused. Related topic: <i>Setting a repository schema on page 52</i>
Usage	tELTOracleInput is to be used along with the tELTOracleMap . Note that the Output link to be used with these components has to reflect faithfully the name of the table Note: Note that the ELT components do not handle actual data flow but only schema information.	

Related scenarios

For uses cases in relation with **tELTOracleInput**, see **tELTOracleMap Scenario 1: Updating Oracle DB entries on page 173**.

tELTOracleMap

All three ELT Oracle components are closely related together in regard to their operating condition. These components should be used to handle Oracle DB schemas to generate Insert, Update or Delete statements including clauses, which are to be executed to the DB output table defined.



tELTOracleMap properties

Component family	ELT	
Function	Helps to graphically build the SQL statement using the table provided as input.	
Purpose	Uses the tables provided as input, to feed the parameter in the built statement. The statement can include inner or outer joins to be implemented between tables or between one table and its aliases.	
Properties	<i>Property type</i>	Either Built-in or Repository.
		Built-in: No existing property data is to be retrieved
		Repository: Select the Repository file where Properties are stored. The following fields are pre-filled in using fetched data.
	<i>Host</i>	Database server IP address
	<i>Port</i>	Listening port number of DB server.
	<i>Database</i>	Name of the database
	<i>Username and Password</i>	DB user authentication data.
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.
	<i>Preview</i>	The preview is an instant shot of the Mapper data. It becomes available when Mapper properties have been filled in with data. The preview synchronization takes effect only after saving changes.
	<i>Map editor</i>	The ELT Map editor allows you to define the output schema as well as build graphically the SQL statement to be executed.
Usage	<p>tELTOracleMap is used along with a tELTOracleInput and tELTOracleOutput. Note that the Output link to be used with these components has to reflect faithfully the name of the tables.</p> <p>Note: Note that the ELT components do not handle actual data flow but only schema information.</p>	

Connecting ELT components

For detailed information regarding ELT component connections, see *Connecting ELT components on page 158*.

Related topic: *Link connection on page 48*

Mapping and joining tables

In the ELT Mapper, you can select specific columns from input schemas and include them in the output schema.

For detailed information regarding the table schema mapping and joining, see *Mapping and joining tables on page 173*.

Adding where clauses

For details regarding the clause handling, see *Adding where clauses on page 173*.

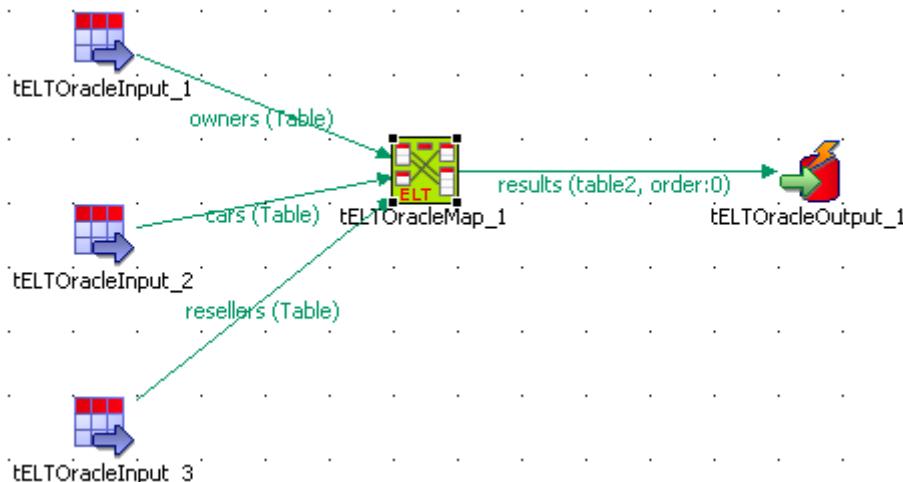
Generating the SQL statement

The mapping of elements from the input schemas to the output schemas create instantly the corresponding Select statement.

The clause defined internally in the ELT Mapper are also included automatically.

Scenario 1: Updating Oracle DB entries

This scenario relies on the job described in ELT MySQL components, *Scenario1: Aggregating table columns and filtering on page 160*. As the update action on the data is available in Oracle DB, this scenario describes a job updating particular entries of the *results* table, adding a *model* to the *make* column of the *cars* table.



- Define all three Input components as described in *Scenario1: Aggregating table columns and filtering on page 160*.
- When connecting the ELT Input components to the ELT mapper, make sure you use the relevant table names as these table names will be used as parameters in the SQL statement generated in the ELT mapper.
- Remove the additional clause used to filter the output columns.

Components

tELTOracleMap

- Add a new filter row to the output table in the ELT mapper to setup a relationship between input and output tables : owners.ID_OWNER=results.ID_OWNER

Expression	Column
cars.MAKE 'C-Class'	MAKE
cars.REG_CAR 'sold by' resellers.NAME_RESELLER	NAME_RESELLER_1

- Remove also all the columns unused for the **Update** action on the output table.
- Then apply the update to the *Make* column adding the mention *C-Class* preceding by a double-pipe.
- And also add the mention *Sold by* in front of the *reseller name* column from the *resellers* table.
- Check the **Generated SQL select query** to be executed.

```
Schema editor | Expression editor | Generated SQL Select query for 'table2' output
SELECT
cars.MAKE || 'C-Class', cars.REG_CAR || 'sold by' || resellers.NAME_RESELLER
FROM
owners , cars , resellers
WHERE
cars.ID_OWNER = owners.ID_OWNER
AND resellers.ID_SELLER = cars.ID_SELLER
AND owners.ID_OWNER=results.ID_OWNER
```

- Click **OK** to validate the changes in the ELT mapper. And make sure the Oracle DB connection details are filled in in the **tELTOracleMap** component properties.
- Select the **tELTOracleOutput** component to define the Action on data to be carried out.

Action on table	Action on data
None	Update

Schema Type: Built-In

Where clauses (for UPDATE and DELETE only): "results.MAKE= 'Mercedes'"

- There is no action on the table, and the **Action on data** is set to **Update**.
- Check that the Schema type corresponds to the output table from the ELT Mapper.
- In the **Where clause** area, add an additional clause: results.MAKE= 'Mercedes' .

- Then press F6 to run the job and check the results table in a DB viewer.

```
Starting job ELTOraclleUpdate at 12:52 11/04/2007.  
Updating with :  
UPDATE results SET (MAKE,NAME_RESELLER) = (SELECT cars.MAKE || '  
C-Class', cars.REG_CAR || ' sold by ' || resellers.NAME_RESELLER  
FROM owners , cars , resellers WHERE cars.ID OWNERS =  
owners.ID OWNER AND resellers.ID RESELLER = cars.ID RESELLER  
AND owners.ID OWNER=results.ID OWNER) WHERE results.MAKE=  
'Mercedes'  
--> 2 rows updated.  
Job ELTOraclleUpdate ended at 12:52 11/04/2007. [exit code=0]
```

The job executes the query generated and updates the relevant rows.

BMW	green	98	All you need Outlet	West Coast City
Mercedes C-Class	gold	3	6523 DY 26 sold by Best Cars Shop	West Coast City
Toyota	gold	84	Best Cars Outlet	West Coast City
Toyota	blue	98	All you need Outlet	West Coast City
Mercedes C-Class	blue	58	7752 OB 89 sold by Cars & Pickup Resale	West Coast City



tELTOracleOutput

All three ELT Oracle components are closely related together in regard to their operating condition. These components should be used to handle Oracle DB schemas to generate Insert, Update or Delete statements including clauses, which are to be executed to the DB output table defined.

tELTOraclerOutput properties

Component family	ELT	 
Function	Carries out the action on the table specified and inserts the data according to the output schema defined the ELT Mapper.	
Purpose	Executes the SQL Insert statement to the MySql database	
Properties	<p>Action on table</p> <p> <i>In Java, use tCreateTable as substitute for this function.</i></p>	<p>On the table defined, you can perform one of the following operations:</p> <p>None: No operation carried out</p> <p>Drop and create the table: The table is removed and created again</p> <p>Create a table: The table doesn't exist and gets created. If the table exists, an error is generated and the job is stopped.</p> <p>Create table if doesn't exist: Create the table if needed and carries out the action on data anyway.</p> <p>Clear a table: The table content is deleted</p>
	<p>Action on data</p>	<p>On the data of the table defined, you can perform the following operation:</p> <p>Insert: Add new entries to the table. If duplicates are found, job stops.</p> <p>Update: updates entries in the table.</p>
	<p>Schema type and Edit Schema</p>	<p>A schema is a row description, i.e., it defines the number of fields to be processed and passed on to the next component. The schema is either built-in or remotely stored in the Repository.</p>
		<p>Built-in: The schema is created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i></p>
		<p>Repository: The schema already exists and is stored in the Repository, hence can be reused. Related topic: <i>Setting a repository schema on page 52</i></p>
	<p>Encoding</p>	<p>Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.</p>
Usage	<p>tELTOraclerOutput is to be used along with the tELTOraclerMap. Note that the Output link to be used with these components has to reflect faithfully the name of the table.</p> <p>Note: Note that the ELT components do not handle actual data flow but only schema information.</p>	

Related scenarios

For uses cases in relation with **tELTOracleOutput**, see **tELTOracleMap Scenario 1: Updating Oracle DB entries on page 173**.



tFileCompare

tFileCompare properties

Component family	File/Management	 
Function	Compares two files and provides comparison data (based on a read-only schema)	
Purpose	Helps at controlling the data quality of files being processed.	
Properties	<i>Schema type</i> and <i>Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields to be processed and passed on to the next component. The schema is either built-in or remotely stored in the Repository but in this case, the schema is read-only.
	<i>File to compare</i>	Filepath to the file to be checked.
	<i>Reference file</i>	Filepath to the file, the comparison is based on.
	<i>If differences are detected, display</i> <i>If no difference detected, display</i>	Type in a message to be displayed in the Run Job console based on the result of the comparison.
	<i>Print to console</i>	Check the box to display the cumessage
Usage	This component can be used as standalone component but it is usually linked to an output component to gather the log data.	
Limitation	n/a	

Scenario: Comparing unzipped files

This scenario describes a job unarchiving a file and comparing it to a reference file to make sure it didn't change. The output of the comparison is stored into a delimited file and a message displays in the console.

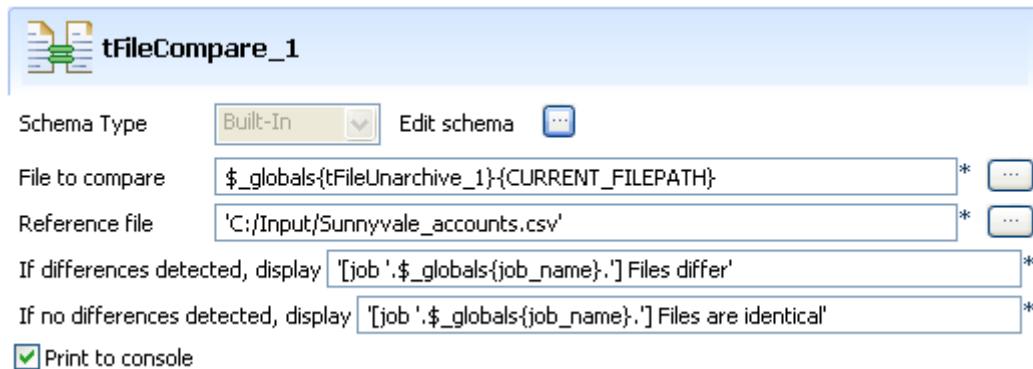


- Drag and drop the following components: **tFileUnarchive**, **tFileCompare**, and **tFileOutputDelimited**.
- Link the **tFileUnarchive** to the **tFileCompare** with **Iterate** connection.

Components

tFileCompare

- Connect the **tFileCompare** to the output component, using a **Main** row link.
- In the **tFileUnarchive** component properties, fill in the path to the archive to unzip.
- In the **Extraction Directory** field, fill in the destination folder for the unarchived file.
- In the **tFileCompare** Properties, set the **File to compare**. Press *Ctrl+Space bar* to display the list of global variables. Select `$_globals{tFileUnarchive_1}{CURRENT_FILEPATH}` or `"(String)globalMap.get("tFileUnarchive_1_CURRENT_FILEPATH")"` according to the language you work with, to fetch the file path from the **tFileUnarchive** component.



- And set the **Reference file** to base the comparison on it.
- In the messages fields, set the messages you want to see in case the files differ or in case the files are identical, for example: '[job '.\$_globals{job_name}.'] Files differ' if you work with Perl or "[job " + jobName + "] Files differ" if you work in Java.
- Check the **Print to Console** box, for the message defined to display at the end of the execution.
- The schema is read-only and contains standard information data. Click **Edit schema** to have a look to it.

Schema of tFileCompare_1							
Column	Key	Type	Nullable	Length	Precision	Comment	
file		String	✓	255			
file_ref		String	✓	255			
moment		Day	✓				
job		String	✓	50			
component		String	✓	255			
differ		int	✓	1			
message		String	✓	255			

- Then set the output component as usual with semi-colon as data separators.
- Save your job and press **F6** to run it.

```
Starting job CompareFiles at 14:11 19/06/2007.  
[job CompareFiles] Files differ  
Job CompareFiles ended at 14:11 19/06/2007. [exit code=0]
```

The message set is displayed to the console and the output shows the schema information data.

```
|file;file_ref;moment;job;component;differ;message  
C:\Input\Accounts\Sunnyvale_accounts_new.xls;C:/Input/sunnyvale_accounts.csv;  
2007-06-19 14:11:59;CompareFiles;tFileCompare_1;1;[job CompareFiles] Files  
differ
```



tFileCopy

tFileCopy Properties

Component family	File/Management	
Function	Copies a source file into a target directory and can remove the source file if so defined.	
Purpose	Helps to streamline processes by automating recurrent and tedious tasks such as copy.	
Properties	<i>File Name</i>	Path to the file to be copied or moved
	<i>Destination</i>	Path to the directory where the file is copied/moved to.
	<i>Remove source file</i>	Check this box to move the file to the destination.
	<i>Replace existing file</i>	Check this box to overwrite any existing file with the newly copied file.
Usage	This component can be used as standalone component .	
Limitation	n/a	

Scenario: Restoring files from bin

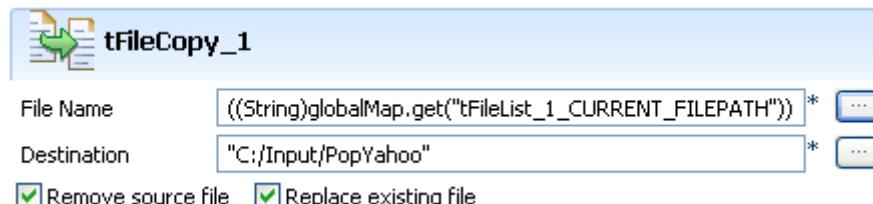
This scenario describes a job that iterates on a list of files, copies each file from the defined source directory to a target directory. It then removes the copied files from the source directory.



- Click and drop a **tFileList** and a **tFileCopy**.
- Link both components using an **Iterate** link.
- In the **tFileList Properties**, set the directory for the iteration loop.



- Set the **Filemask** to “*.txt” to catch all files with this extension. For this use case, the case is not sensitive.
- Then select the **tFileCopy** to set its **Properties**.



- In the File Name field, press Ctrl+Space bar to access the list of variables.
- Select the global variable `((String)globalMap.get("tFileList_1_CURRENT_FILEPATH"))` if you work in Java, or `$_globals{tFileList_1}{CURRENT_FILEPATH}` if you work in Perl. This way, all files from the source directory can be processed.
- Check the **Remove Source file** box to get rid of the file that have been copied.
- Check the **Replace existing file** to overwrite any file possibly present in the destination directory.
- Save your job and press **F6**.

The files are copied onto the destination folder and are removed from the source folder.

Components

tFileDelete

tFileDelete



tFileDelete Properties

Component family	File/Management	
Function	Suppresses a file from a defined directory.	
Purpose	Helps to streamline processes by automating recurrent and tedious tasks such as delete..	
Properties	File Name	Path to the file to be copied or moved
Usage	This component can be used as standalone component.	
Limitation	n/a	

Scenario: Deleting files

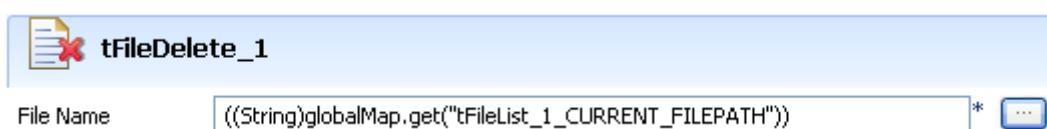
This very simple scenario describes a job deleting files from a given directory.



- Click and drop the following components: tFileList, tFileDelete, tJava.
- In the **tFileList Properties**, set the directory to loop on in the **Directory** field.



- The filenmask is “*.txt” and no case check is to carry out.
- In the **tFileDelete Properties** panel, set the **File Name** field in order for the current file in selection in the **tFileList** component be deleted. This allows to delete all files contained in the directory defined earlier on.



- press **Ctrl+Space bar** to access the list of global variables. In Java, the relevant variable to collect the current file is:
`((String)globalMap.get("tFileList_1_CURRENT_FILEPATH")).`
- Then in the **tJava** component, define the message to be displayed in the standard output (Run Job console). In this Java use case, type in the Code field, the following script:
`System.out.println(
((String)globalMap.get("tFileList_1_CURRENT_FILE"))
+ " has been deleted!");`
- Then save your job and press **F6** to run it.

```
Starting job FileDel at 18:29 20/06/2007.  
16.txt has been deleted!  
15.txt has been deleted!  
14.txt has been deleted!  
13.txt has been deleted!  
12.txt has been deleted!  
11.txt has been deleted!  
10.txt has been deleted!  
09.txt has been deleted!  
08.txt has been deleted!  
07.txt has been deleted!  
06.txt has been deleted!  
05.txt has been deleted!  
04.txt has been deleted!  
03.txt has been deleted!  
02.txt has been deleted!  
01.txt has been deleted!  
Job FileDel ended at 18:29 20/06/2007. [exit code=0]
```

The message set in the **tJava** component displays in the log, for each file that has been deleted through the **tFileDelete** component.

Components

tFileFetch



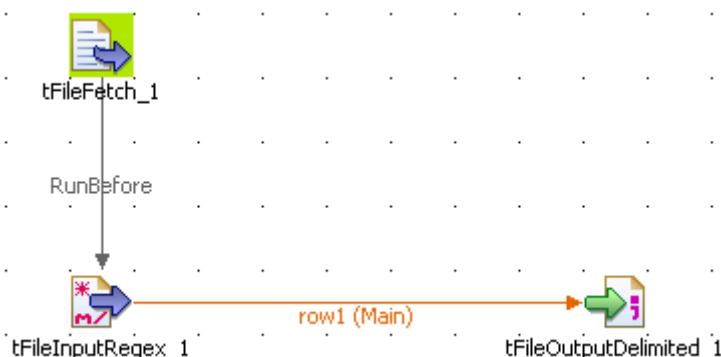
tFileFetch

tFileFetch properties

Component family	Internet	
Function	tFileFetch retrieves a file from HTTP	
Purpose	tFileFetch allows to fetch data contained in a file through HTTP protocol.	
Properties	<i>URI</i>	Type in the URI of the HTTP site where the file is to be fetched from.
	<i>Destination Directory</i>	Browse to the destination folder where the file fetched will be placed.
	<i>Destination Filename</i>	Type in a new name for the file fetched, if need be.
Usage	This component is generally used as a start component to feed the input flow of a job and is often connected to the job through a ThenRun link.	
Limitation	n/a	

Scenario: Fetching data through HTTP

This scenario describes a three-component job which retrieves data from an HTTP website and select data that will be stored into a delimited file.



- Click and drop a **tFileFetch**, a **tFileInputRegex** and a **tFileOutputDelimited** onto your workspace.
- In the **tFileFetch Properties** panel, type in the URI where the file to be fetched can retrieved from.
- In the **Destination directory** field, browse to the folder where the fetched file is to be stored.

- In the **Filename** field, type in a new name for the file if you want it to be changed. In this example, *filefetch.txt*.
- Select the **tFileInputRegex**, set the **File name** so that it corresponds to the file fetched earlier.
- Using a regular expression, in the **Regex** field, select the relevant data from the fetched file. In this example: <td(?: class="leftalign")?> \s* (t\w+) \s* </td>

 Take care to use the correct Regex syntax according to the generation language in use as the syntax is different in Java/Perl.

- Define the **header**, **footer** and **limit** if need be. In this case, we'll ignore these fields.
- Define also the schema describing the flow to be passed on to the final output.
- The schema should be automatically propagated to the final output, but to be sure, check the schema in the **Properties** panel of the tFileOutputDelimited component.
- Then press **F6** to run the job.

Components

tFileInputDelimited



tFileInputDelimited

tFileInputDelimited properties

Component family	File/Input	
Function	tFileInputDelimited reads a given file row by row with simple separated fields.	
Purpose	Opens a file and reads it row by row to split them up into fields then sends fields as defined in the Schema to the next job component, via a Row link.	
Properties	<i>Property type</i>	Either Built-in or Repository.
		Built-in: No existing property data is to be retrieved
		Repository: Select the Repository file where Properties are stored. The following fields are pre-filled in using fetched data.
	<i>File Name</i>	Name of the file to be processed. Related topic: <i>Defining the context variables on page 96</i>
	<i>Field separator</i>	Character, string or regular expression to separate fields.
	<i>Row separator</i>	String (ex: "\n"on Unix) to distinguish rows.
	<i>Header</i>	Number of rows to be skipped in the beginning of file
	<i>Footer</i>	Number of rows to be skipped at the end of the file.
	<i>Limit</i>	Maximum number of rows to be processed. If Limit = 0, no row is read or processed.
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository. Click Edit Schema to make changes to the schema. Note that if you make changes, the schema automatically becomes built-in. Click Sync columns to retrieve the schema from the previous component connected in the job.
		Built-in: The schema will be created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>
		Repository: The schema already exists and is stored in the Repository, hence can be reused in various projects and job flowcharts. Related topic: <i>Setting a repository schema on page 52</i>
	<i>Skip empty rows</i>	Check this box to skip empty rows.

	Extract lines at random/ Number of lines	Check this box to set the number of lines to be extracted randomly.
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.
Usage	Use this component to read a file and separate fields contained in this file using a defined separator.	

Scenario: Delimited file content display

The following scenario creates a two-component job, which aims at reading each row of a file, selecting delimited data and displaying the output in the **Run Job** log console.



- Click and drop a **tFileInputDelimited** component from the Palette to the design workspace.
- Click and drop a **tLogRow** component the same way.
- Right-click on the **tFileInputDelimited** component and select *Row > Main*. Then drag it onto the **tLogRow** component and release when the plug symbol shows up.
- Select the **tFileInputDelimited** component again, and define its properties:

tFileInputDelimited_1

Property Type	Repository	Repository	DELIM:Owners	*
File Name	'C:/Input/Owners.csv'			
Row Separator	"\n"	Field Separator	''	
Header	1	Footer	0	Limit
Schema Type	Repository	DELIM:Owners - metadata	*	Edit schema
<input type="checkbox"/> Extract lines at random				
Encoding Type	ISO-8859-15			

- Fill in a path to the file in the **File Name** field. This field is mandatory.
- Define the **Row separator** allowing to identify the end of a row. Then define the **Field separator** used to delimit fields in a row.
- In this scenario, the header and footer limits are not set. And the **Limit** number of processed rows is set on 50.

Components

tFileInputDelimited

- Select either a local (**Built-in**) or a remotely managed (**Repository**) **Schema type** to define the data to pass on to the **tLogRow** component.
- You can load and/or edit the schema via the **Edit Schema** function.

Related topics: *Setting a built-in schema* and *Setting a repository schema on page 52*

- As selected, the empty rows will be ignored.
- Enter the encoding standard the input file is encoded in. This setting is meant to ensure encoding consistency throughout all input and output files.
- Select the **tLogRow** and define the **Field separator** to use for the output display. Related topic: *tLogRow properties on page 230*.
- Check the **Print schema column name in front of each value** box to retrieve the column labels in the output displayed.
- Go to **Run Job** tab, and click on **Run** to execute the job.

The file is read row by row and the extracted fields are displayed on the Run Job log as defined in both components **Properties**.

```
Starting job FileDel at 10:45 12/04/2007.
ID_Owner: 2|Name: lebouh|ID_Insurance: PKI2906
ID_Owner: 3|Name: bouhnan|ID_Insurance: BNU9147
ID_Owner: 4|Name: hirtle|ID_Insurance: TEV8360
ID_Owner: 5|Name: bobouh|ID_Insurance: WPM3151
ID_Owner: 6|Name: carbone|ID_Insurance: IFY9885
```

The Log sums up all parameters in a header followed by the result of the job.



tFileInputMail

tFileInputMail properties

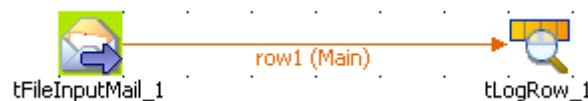
Component family	File/Input	
Function	reads the header and content parts of an email file defined	
Purpose	helps to extract standard key data from emails	
Properties	<i>File name</i>	Browse to the source email file
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository . Click Edit Schema to make changes to the schema. Note that if you make changes, the schema automatically becomes built-in. Click Sync columns to retrieve the schema from the previous component connected in the job.
		Built-in: The schema will be created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>
		Repository: The schema already exists and is stored in the Repository, hence can be reused in various projects and job flowcharts. Related topic: <i>Setting a repository schema on page 52</i>
	<i>Mail parts</i>	Column: This field is automatically populated with the columns defined in the schema that you propagated.
		Mail part: Type in the label of the header part or body to be displayed on the output.
Usage	This component handles flow of data therefore it requires input and output, hence is defined as an intermediary step.	
Limitation	n/a	

Scenario: Extracting key fields from email

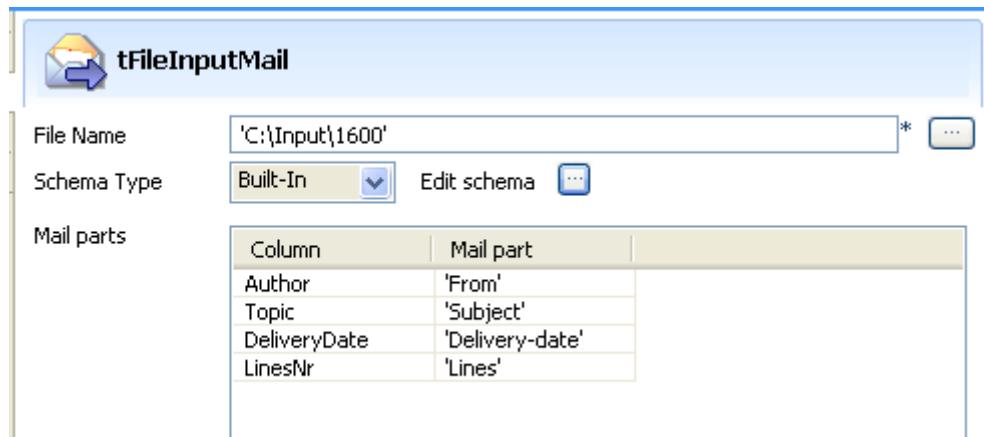
This two-component scenario is aimed at extracting some key standard fields and displaying the values on the **Run Job** console.

Components

tFileInputMail



- Click and drop a **tFileInputMail** and a **tLogRow** component
- On the **Properties** tab, define the email parameters:



- Browse to the mail File to be processed. Define the schema including all columns you want to retrieve on your output.
- Once the schema is defined, click **OK** to propagate it into the **Mail parts** table
- On the **Mail part** column of the table, type in the actual header or body standard keys that will be used to retrieve the values to be displayed.
- Define the tLogRow in order for the values to be separated by a carriage return. On Windows OS, type in `\n` between double quotes.
- Press F6 to run the job and display the output flow on the execution console.

```
Starting job FileInputMail at 11:42 18/01/2007.  
"Brice L" <Brice.L@technologies.fr>  
Re: Gestion de transactions  
Tue, 16 Jan 2007 16:30:02 +0100  
111  
Job FileInputMail ended at 11:42 18/01/2007. [exit code=0]
```

The header key values are extracted as defined in the **Mail parts** table. Indeed, the author, topic, delivery date and number of lines are part of the output displayed.

tFileInputPositional



tFileInputPositional properties

Component family	File/Input	 
Function	tFileInputPositional reads a given file row by row and extracts fields based on a pattern.	
Purpose	Opens a file and reads it row by row to split them up into fields then sends fields as defined in the Schema to the next job component, via a Row link.	
Properties	<p><i>Property type</i></p> <p>Built-in: No existing property data is to be retrieved</p> <p>Repository: Select the Repository file where Properties are stored. The following fields are pre-filled in using fetched data.</p>	Either Built-in or Repository.
<i>File Name</i>		Name of the file to be processed. Related topic: <i>Defining the context variables on page 96</i>
<i>Field separator</i>		Character, string or regular expression to separate fields.
<i>Row separator</i>		String (ex: "\n" on Unix) to distinguish rows.
<i>Header</i>		Number of rows to be skipped in the beginning of file
<i>Footer</i>		Number of rows to be skipped at the end of the file.
<i>Limit</i>		Maximum number of rows to be processed. If Limit = 0, no row is read or processed.
<i>Schema type and Edit Schema</i>		A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository.
		Built-in: The schema will be created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>
		Repository: The schema already exists and is stored in the Repository, hence can be reused in various projects and job flowcharts. Related topic: <i>Setting a repository schema on page 52</i>
<i>Skip empty rows</i>		Check this box to skip empty rows.
<i>Pattern</i>		Length values separated by commas, interpreted as a string between quotes. Make sure the values entered in this fields are consistent with the schema defined.
<i>Encoding</i>		Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.

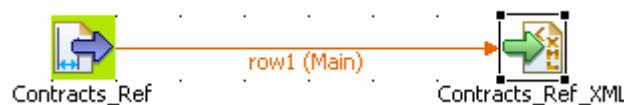
Components

tFileInputPositional

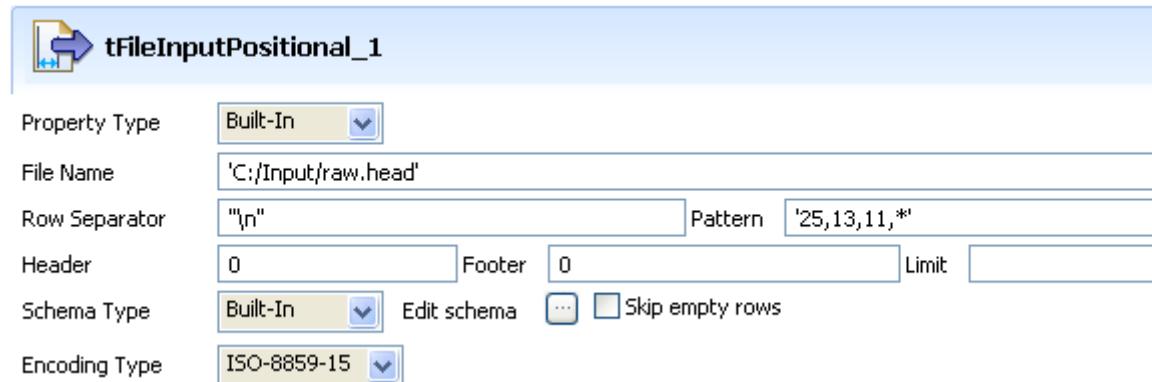
Usage	Use this component to read a file and separate fields using a position separator value.
--------------	---

Scenario: From Positional to XML file

The following scenario creates a two-component job, which aims at reading data of an Input file and outputting selected data (according to the data position) into an XML file.



- Click and drop a **tFileInputPositional** component from the Palette to the design workspace. The file contains raw data, in this case, contract nr, customer references and insurance numbers.
- Click and drop a **tFileOutputXML** component. This file is meant to receive the references in a structured way.
- Right-click on the **tFileInputPositional** component and select *Row > Main*. Then drag it onto the **tFileOutputXML** component and release when the plug symbol shows up.
- Select the **tFileInputPositional** component again, and define its properties.
- The job properties are built-in for this scenario. As opposed to the Repository, this means that the **Property type** is set for this station only.

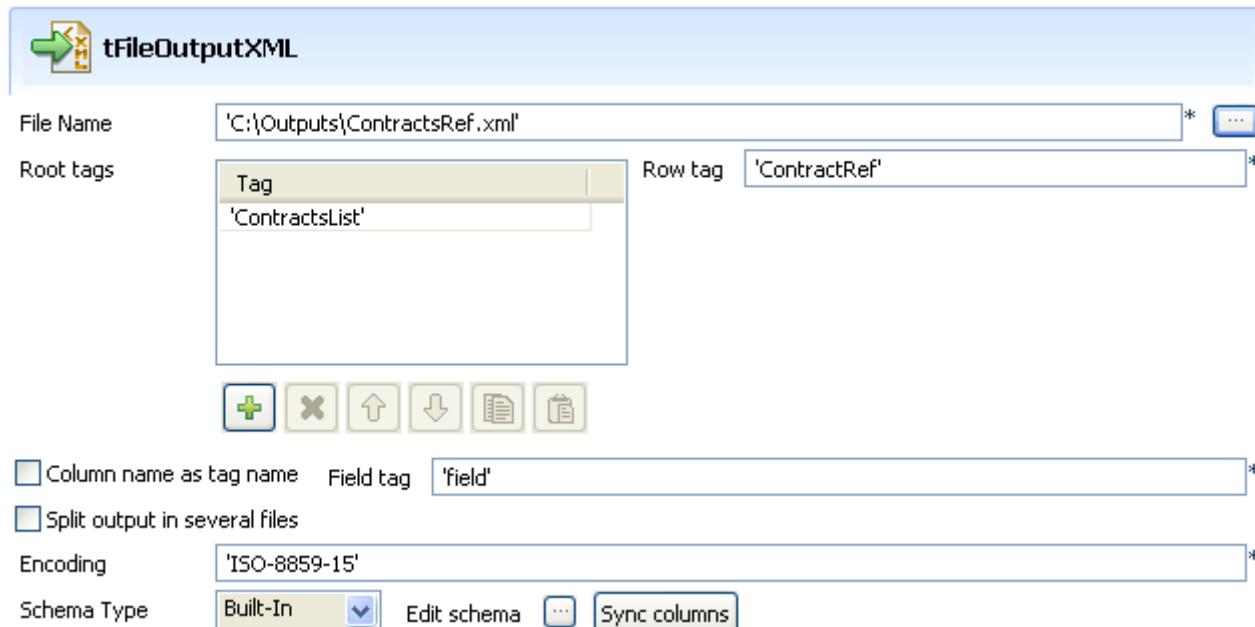


- Fill in a path to the file in the **File Name** field. This field is mandatory.
- Define the **Row separator** identifying the end of a row, by default, a carriage return.
- Then define the **Pattern** to delimit fields in a row. The pattern is a series of length values corresponding to the values of your input files. The values should be entered between quotes, and separated by a comma. Make sure the values you enter match the schema defined.
- In this scenario, the header, footer and limit fields are not set. But depending on the input file structure, you may need to define them.

- Select a **Schema type** to define the data to pass on to the **tFileOutputXML** component.
- You can load and/or edit the schema via the **Edit Schema** function. For this schema, define three columns, respectively *Contracts*, *CustomerRef* and *InsuranceNr* matching the three value lengths defined.



- Then define the second component properties:
- Enter the XML output file path.



- Enter a root tag (or more), to wrap the XML structure output, in this case ‘ContractsList’.
- Define the row tag that will wrap each line data, in this case ‘ContractRef’.
- Check the box **Column name as tag name** to reuse the column label from the input schema as tag label. By default, ‘field’ is used for each column value data.

Components

tFileInputPositional

- Enter the **Encoding** standard, the input file is encoded in. Note that, for the time being, the encoding consistency verification is not supported.
- Select the **Schema type**. If the row connection is already implemented, the schema is automatically synchronized with the Input file schema. Else, click on **Sync columns**.
- Go to the **Run Job** tab, and click on **Run** to execute the job.

The file is read row by row and split up into fields based on the length values defined in the **Pattern** field. You can open it using any standard XML editor.

```
- <ContractsList>
  - <ContractRef>
    <Contract>00004</Contract>
    <CustomerRef>8200</CustomerRef>
    <InsuranceNr>50320</InsuranceNr>
  </ContractRef>
  - <ContractRef>
    <Contract>00010</Contract>
    <CustomerRef>8200</CustomerRef>
    <InsuranceNr>50335</InsuranceNr>
  </ContractRef>
  - <ContractRef>
    <Contract>00001</Contract>
    <CustomerRef>7200</CustomerRef>
    <InsuranceNr>50320</InsuranceNr>
  </ContractRef>
```

tFileInputRegex



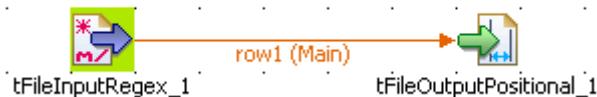
tFileInputRegex properties

Component family	File/Input	
Function	Powerful feature which can replace number of other components of the File family. Requires some advanced knowledge on regular expression syntax	
Purpose	Opens a file and reads it row by row to split them up into fields using regular expressions. Then sends fields as defined in the Schema to the next job component, via a Row link.	
Properties	<i>Property type</i>	Either Built-in or Repository.
		Built-in: No existing property data is to be retrieved
		Repository: Select the Repository file where Properties are stored. The following fields are pre-filled in using fetched data.
	<i>File Name</i>	Name of the file to be processed. Related topic: <i>Defining the context variables on page 96</i>
	<i>Row separator</i>	String (ex: "\n" on Unix) to distinguish rows.
	<i>Regex</i>	This field is Perl or Java compatible and can contain multiple lines. Type in your regular expressions including the subpattern matching the fields to be extracted. Note: In Java, antislashes need to be doubled in regexp Regexp syntax is different if in Java/Perl
	<i>Header</i>	Number of rows to be skipped in the beginning of file
	<i>Footer</i>	Number of rows to be skipped at the end of the file.
	<i>Limit</i>	Maximum number of rows to be processed. If Limit = 0, no row is read or processed.
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository.
		Built-in: The schema will be created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>

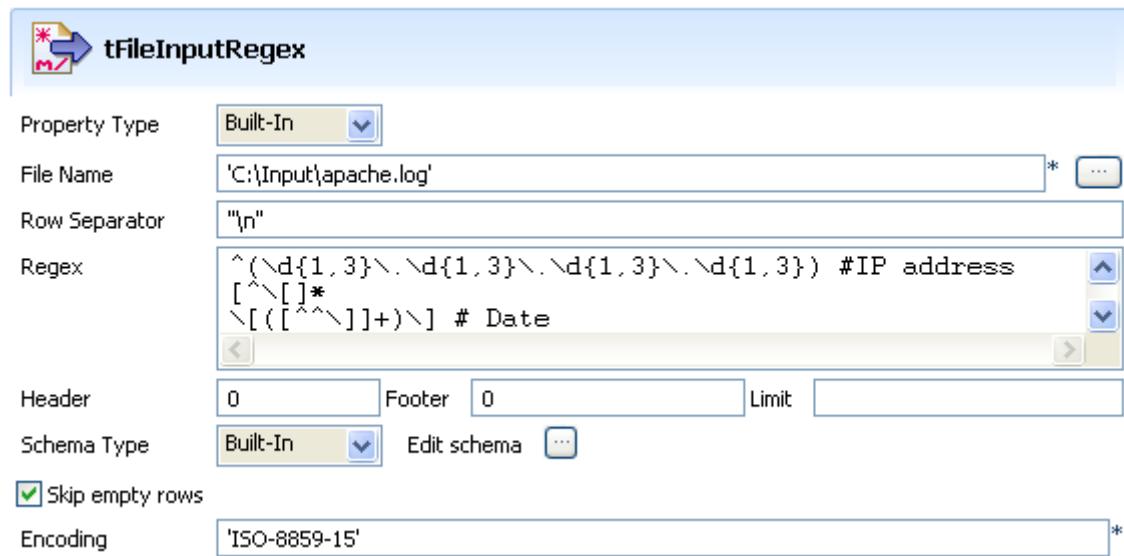
		Repository: The schema already exists and is stored in the Repository, hence can be reused in various projects and job flowcharts. Related topic: <i>Setting a repository schema on page 52</i>
	<i>Skip empty rows</i>	Check this box to skip empty rows.
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.
Usage	Use this component to read a file and separate fields contained in this file according to the defined Regex.	
Limitation	n/a	

Scenario: Regex to Positional file

The following scenario creates a two-component job, reading data from an Input file using regular expression and outputting delimited data into an XML file.



- Click and drop a **tFileInputRegex** component from the Palette to the design workspace.
- Click and drop a **tFileOutputPositional** component the same way.
- Right-click on the **tFileInputRegex** component and select *Row > Main*. Drag this main row link onto the **tFileOutputPositional** component and release when the plug symbol displays.
- Select the **tFileInputRegex** again so the properties tab shows up, and define the properties:



- The job is built-in for this scenario. Hence, the Properties are set for this station only.
- Fill in a path to the file in **File Name** field. This field is mandatory.
- Define the **Row separator** identifying the end of a row.
- Then define the **Regular expression** in order to delimit fields of a row, which are to be passed on to the next component. You can type in a regular expression using Perl code, and on multiple lines if needed.

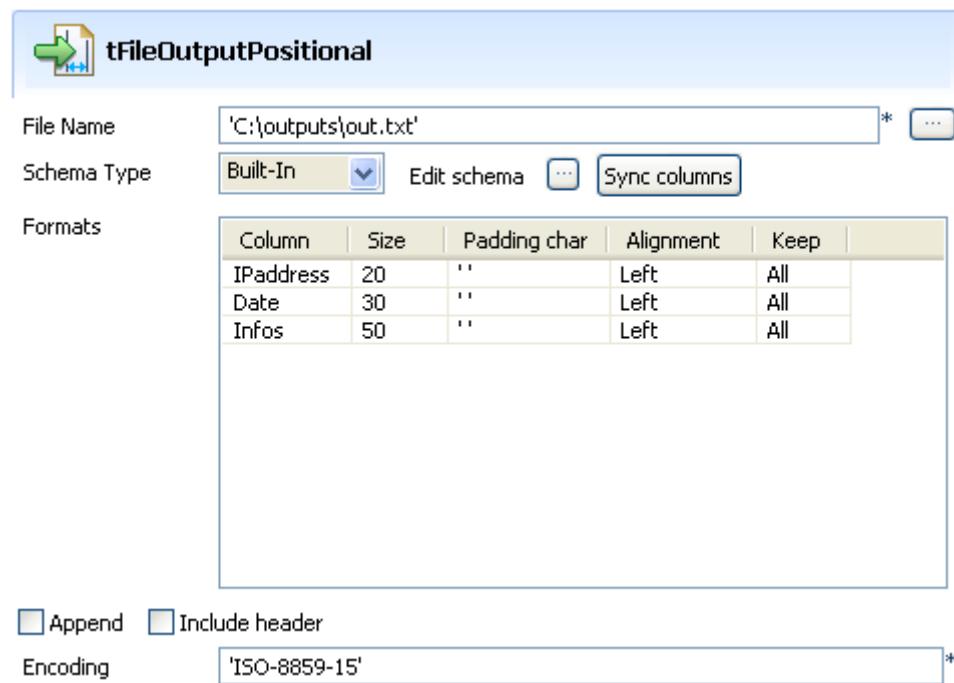


Take care to use the correct Regex syntax according to the generation language in use as the syntax is different in Java/Perl.

- In this expression, make sure you include all subpatterns matching the fields to be extracted.
- In this scenario, ignore the header, footer and limit fields.
- Select a local (**Built-in**) **Schema type** to define the data to pass on to the **tFileOutputPositional** component.
- You can load or create the schema through the **Edit Schema** function.
- Then define the second component properties:
-

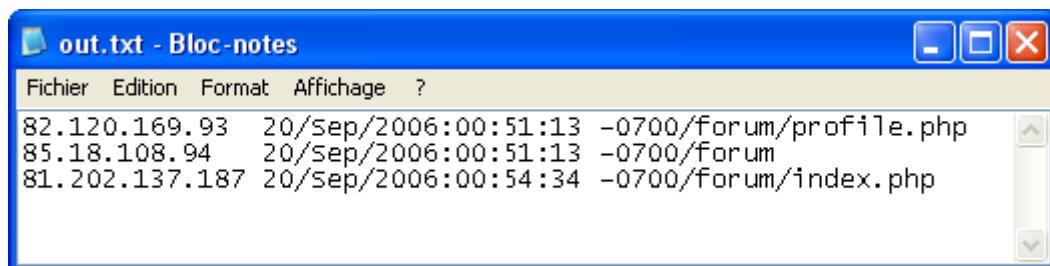
Components

tFileInputRegex



- Enter the Positional file output path.
- Enter the **Encoding** standard, the output file is encoded in. Note that, for the time being, the encoding consistency verification is not supported.
- Select the **Schema type**. Click on **Sync columns** to automatically synchronize the schema with the Input file schema.
- Now go to the **Run Job** tab, and click on **Run** to execute the job.

The file is read row by row and split up into fields based on the **Regular Expression** definition. You can open it using any standard file editor.



tFileInputXML



tFileInputXML Properties

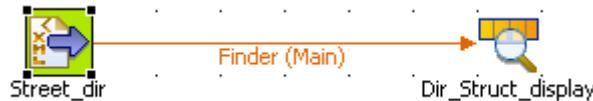
Component family	File/Input	
Function	tFileInputXML reads an XML structured file and extracts data row by row.	
Purpose	Opens an XML structured file and reads it row by row to split them up into fields then sends fields as defined in the Schema to the next component, via a Row link.	
Properties	<i>Property type</i>	Either Built-in or Repository. Built-in: No existing property data is to be retrieved Repository: Select the Repository file where Properties are stored. The following fields are pre-filled in using fetched data.
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository.
		Built-in: The schema will be created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>
		Repository: The schema already exists and is stored in the Repository, hence can be reused in various projets and job flowcharts. Related topic: <i>Setting a repository schema on page 52</i>
	<i>File Name</i>	Name of the file to be processed. Related topic: <i>Defining the context variables on page 96</i>
	<i>Loop XPath query</i>	Node of the tree, which the loop is based on
	<i>Mapping column/XPath Query</i>	Column reflects the schema as defined by the Schema type field XPath Query: Enter the fields to be extracted from the structured input.
	<i>Limit</i>	Maximum number of rows to be processed. If Limit = 0, no row is read nor processed.
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.
Limitation	n/a	

Components

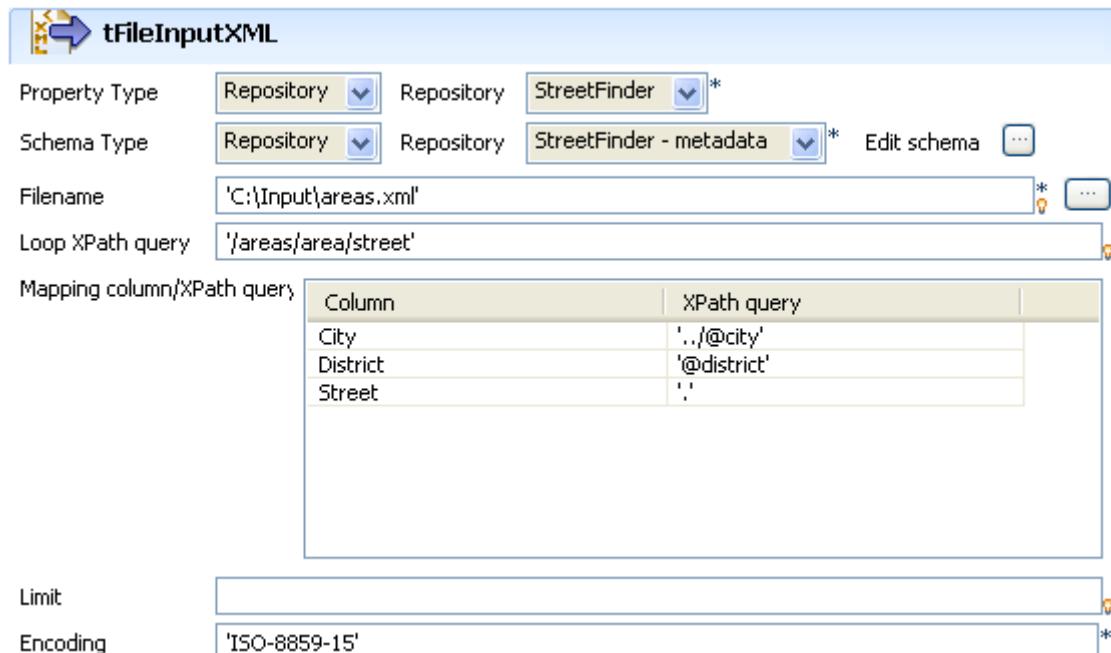
tFileInputXML

Scenario: XML street finder

This very basic scenario is made of two components. A **tFileInputXML** component extracts from the defined street directory file and the output is displayed on the **Run Job** console via a **tLogRow** component.



- Select a **tFileInputXML** file from the **File** folder in the **Palette**. Click and drop also a **tLogRow** component and connect both components.
- On the **Properties** panel of the tFileInputXML, define the properties:



- As the street dir file used as input file has been previously defined in the Metadata area, select **Repository** as **Property type**. This way, the properties are automatically leveraged and the rest of the properties fields are filled in (apart from Schema). For more information regarding the metadata creation wizards, see *Defining Metadata items on page 54*.
- Select the same way the relevant schema in the Repository metadata list. **Edit schema** if you want to make any change to the schema loaded.
- The **Filename** shows the structured file to be used as input
- In **Loop XPath query**, change if needed the node of the structure where the loop is based.
- On the **Mapping** table, fill the fields to be extracted and displayed in the output.
- If the file size is consequent, fill in a **Limit** of rows to be read.

- Enter the encoding if needed then double-click on tLogRow to define the separator character.
- At last, press **F6** or go to **Run Job** and click **Run** to execute the job. On the console, the fields defined in the input properties are extracted from the XML structured and displayed.

```
Starting job XMLStreetFinder at 12:42 05/01/2007.  
Paris|2eme arrondissement|Rue de la Paix  
Paris|8eme arrondissement|Champs Elysees  
New York City|Manhattan|Madison avenue  
New York City|Brooklyn|Washington heights  
Job XMLStreetFinder ended at 12:42 05/01/2007. [exit code
```

Components

tFileDialog

tFileDialog

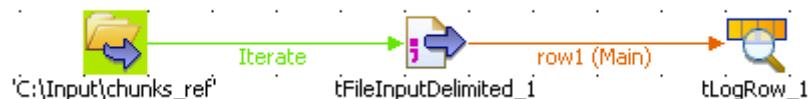


tFileDialog properties

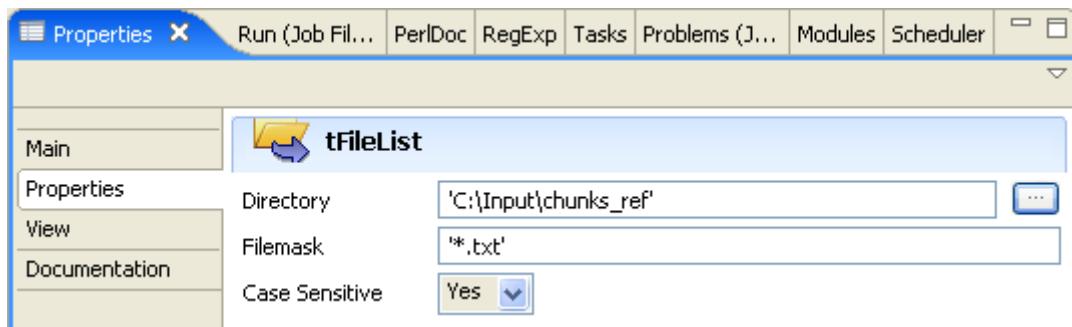
Component family	File/Management	
Function	iterates on files of a set directory.	
Purpose	tFileDialog takes out a set of files based on a filmask pattern and iterates on each file.	
Properties	Directory	Path to the directory where files are stored
	Filmask	Filename or filmask using wildcharacter (*).
	Case sensitive	Create case sensitive filter on filenames.
Usage	tFileDialog provides a list of files from a defined directory on which to iterate	

Scenario: Iterating on a file directory

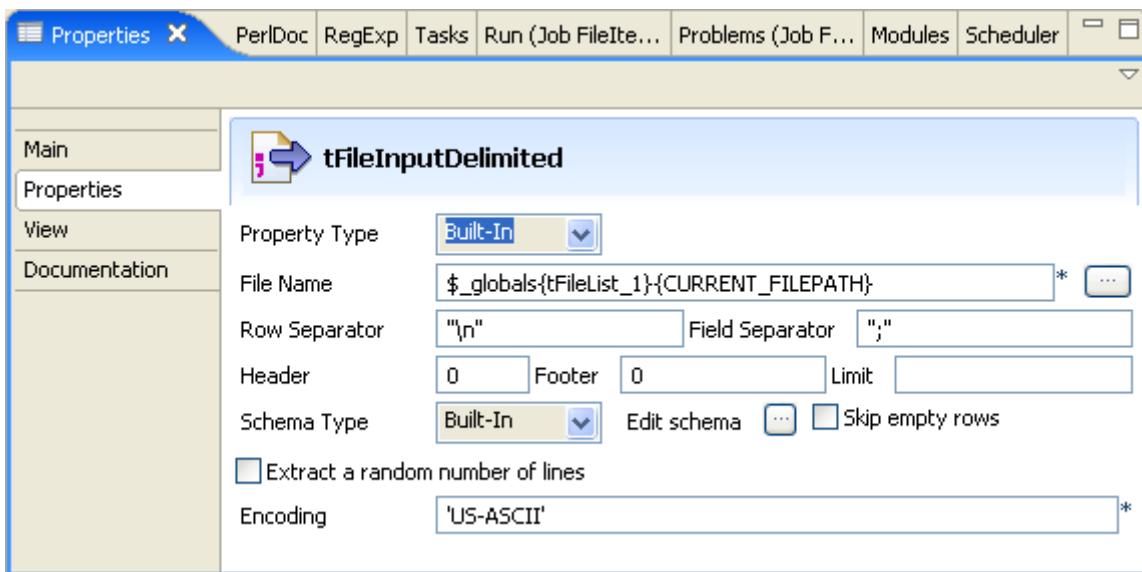
The following scenario creates a three-component job, which aims at listing files from a defined directory, reading each file by iteration, selecting delimited data and displaying the output in the **Run Job** log console.



- Click and drop a **tFileDialog**, a **tFileInputDelimited** and a **tLogRow** component into the Design workspace.
- Right-click on the **tFileDialog** component, and pull an **Iterate** connection to the **tFileInputDelimited** component. Then pull a **Main** row from the **tFileInputDelimited** to the **tLogRow** component.
- Now define the properties of all three components.
- First select the **tFileDialog** component, and click on **Properties** tab:



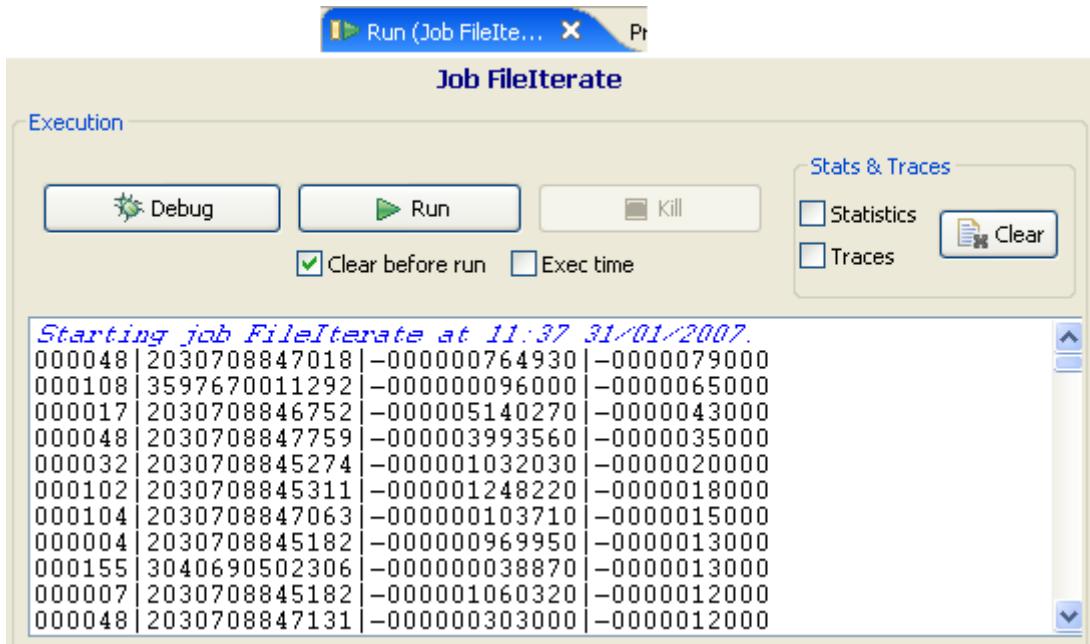
- Browse to the **Directory** of the files to process. To display the path on the job itself, use the hint label (`__DIRECTORY__`) that shows up when you browse over the Directory field. Type in this reference in the **Label Format** field of the **View** tab.
- Enter a **Filmask** using wildcards if need be.
- The case is sensitive.
- Click on the **tFileInputDelimited** component and set the properties:



- Enter the **File Name** field using a variable containing the current filename path, as you filled in in the properties of **tFileDialog**. Press **Ctrl+Space bar** to access the autocomplete list of variables.
- Fill in all other fields as detailed in the **tFileInputDelimited** section. Related topic: *tFileInputDelimited properties on page 188*
- Select the last component, **tLogRow** and fill in the separator to be used to distinguish field content displayed on the Log. Related topic: *tLogRow properties on page 230*.

Components

tFileDialog



The job iterates on the directory defined, and reads each file contained. Then delimited data is passed on to the last component which displays it on the Log.

For other scenarios using tFileList, see *tFileCopy on page 182*.

tFileOutputExcel tFileOutputExcel Properties

Component family	File/Output	 
Function	tFileOutputExcel outputs data to an MS Excel type of file.	
Purpose	tFileOutputExcel writes an MS Excel file with separated data value according to a defined schema.	
Properties	<i>File name</i>	Name or path to the output file. Related topic: <i>Defining the context variables on page 96</i>
	<i>Sheet name</i>	Name of the sheet
	<i>Include header</i>	Check the box to include header row to the output file
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository.
		Built-in: The schema will be created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>
		Repository: The schema already exists and is stored in the Repository, hence can be reused in various projects and job designs. Related topic: <i>Setting a repository schema on page 52</i>
	<i>Sync columns</i>	Click to synchronize the output file schema with the input file schema. The Sync function only displays once the Row connection is linked with the Output component.
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.
Usage	Use this component to write an XML file with data passed on from other components using a Row link.	
Limitation	n/a	

Related scenario

For **tFileOutputExcel** related scenario, see *tSugarCRMInput on page 324*.

tFileOutputLDIF

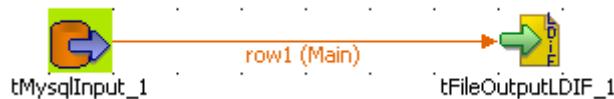


tFileOutputLDIF Properties

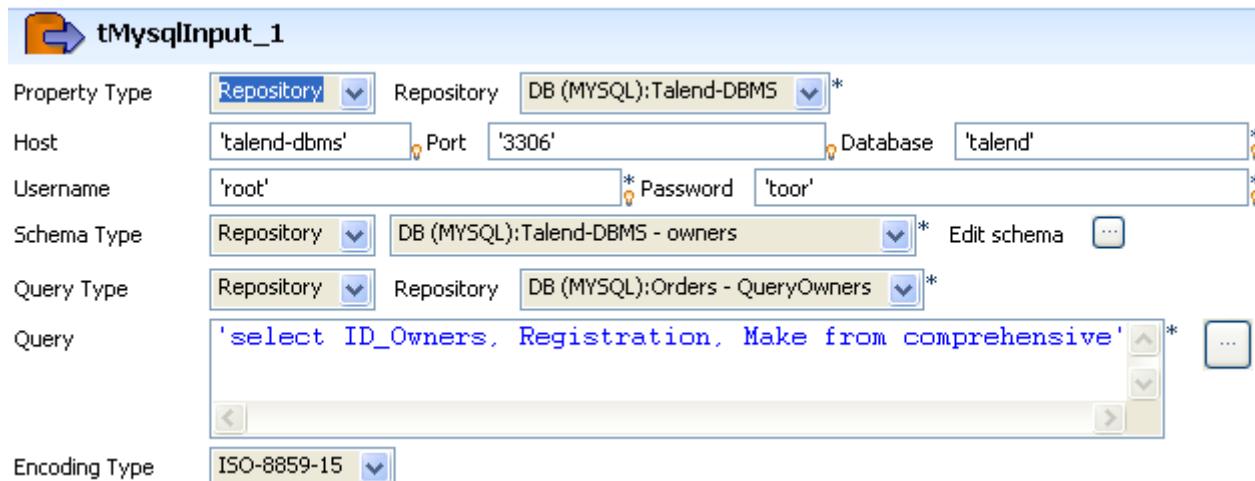
Component family	File/Output	
Function	tFileOutputLDIF outputs data to an LDIF type of file which can then be loaded into a LDAP directory.	
Purpose	tFileOutputLDIF writes or modifies a LDIF file with data separated in respective entries based on the schema defined.,or else deletes content from an LDIF file.	
Properties	<i>File name</i>	Name or path to the output file. Related topic: <i>Defining the context variables on page 96</i>
	<i>Wrap</i>	Wraps the file content, every defined number of characters.
	<i>Change type</i>	Select Add , Modify or Delete to respectively create an LDIF file, modify or remove an existing LDIF file. In case of modification, set the type of attribute changes to be made.
	<i>Change on attributes</i>	Select Add , Modify or Delete to respectively add a new attribute to the file, replace the attributes with new ones or suppress attributes from the file defined.
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository.
	Built-in: The schema will be created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>	
	Repository: The schema already exists and is stored in the Repository, hence can be reused in various projects and job designs. Related topic: <i>Setting a repository schema on page 52</i>	
	<i>Sync columns</i>	Click to synchronize the output file schema with the input file schema. The Sync function only displays once the Row connection is linked with the Output component.
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.
Usage	Use this component to write an XML file with data passed on from other components using a Row link.	
Limitation	n/a	

Scenario: Writing DB data into an LDIF-type file

This scenario describes a two component job which aims at extracting data from a database table and writing this data into a new output LDIF file.



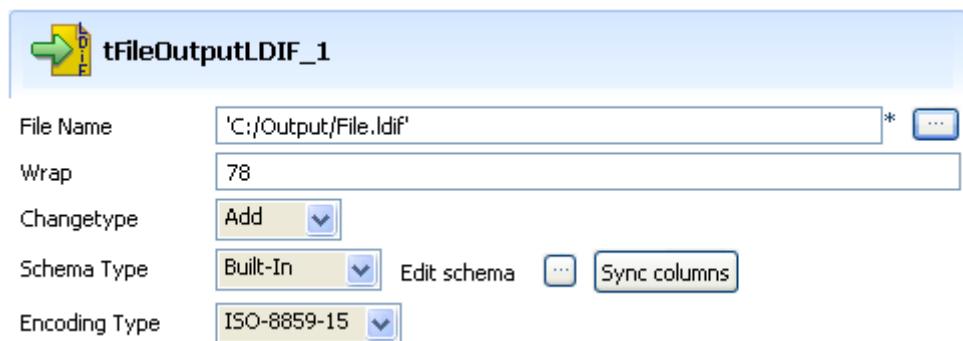
- Click and drop a **tDBInput** and a **tFileOutputLDIF** component from the Palette to the design area. Bind them together using a **Row > Main** link.
- Select the **tDBInput** component, and go to the **Properties** panel then select the **Properties** tab.
- If you stored the DB connection details in a **Metadata** entry in the Repository, set the **Property type** as well as the **Schema type** on **Repository** and select the relevant metadata entry. All other fields are filled in automatically, and retrieve the metadata-stored parameters.



- Alternatively select **Built-in** as **Property type** and **Schema type** and fill in the DB connection and schema fields manually.
- Then double-click on **tFileOutputLDIF** and define the **Properties**.
- Browse to the folder where you store the Output file. In this use case, a new LDIF file is to be created. Thus type in the name of this new file.
- In the **Wrap** field, enter the number of characters held on one line. The text coming afterwards will get wrapped onto the next line.

Components

tFileOutputLDIF



- Select **Add** as **Change Type** as the newly created file is by definition empty. In case of modification type of Change, you'll need to define the nature of the modification you want to make to the file.
- As **Schema Type**, select **Built-in** and use the **Sync Columns** button to retrieve the input schema definition.
- Press F6 to short run the job.

The screenshot shows the Lister application window displaying LDIF data. The data consists of three entries:

```
dn: 24
changetype: add
id_owners: 24
registration: 5382 KC 94
make: Volkswagen

dn: 32
changetype: add
id_owners: 32
registration: 9591 OE 79
make: Honda

dn: 35
changetype: add
id_owners: 35
registration: 3129 VH 61
make: Volkswagen
```

The LDIF file created contains the data from the DB table and the type of change made to the file, in this use case, addition.

tFileOutputXML



tFileOutputXML properties

Component family	File/Output	
Function	tFileOutputXML outputs data to an XML type of file.	
Purpose	tFileOutputXML writes an XML file with separated data value according to a defined schema.	
Properties	<i>File name</i>	Name or path to the output file. Related topic: <i>Defining the context variables on page 96</i>
	<i>Root tag</i>	Wraps the whole output file structure and data.
	<i>Row tag</i>	Wraps data and structure per row
	<i>Column name as tag name</i>	Check the box to leverage the column labels from the input schema, as data wrapping tag.
	<i>Split output in files</i>	If the XML file output is big , you can split the file every certain number of rows.
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository.
		Built-in: The schema will be created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>
		Repository: The schema already exists and is stored in the Repository, hence can be reused in various projects and job designs. Related topic: <i>Setting a repository schema on page 52</i>
	<i>Sync columns</i>	Click to synchronize the output file schema with the input file schema. The Sync function only displays once the Row connection is linked with the Output component.
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.
Usage	Use this component to write an XML file with data passed on from other components using a Row link.	
Limitation	n/a	

Scenario: From Positional to XML file

Find a scenario using tFileOutputXML component at section: *Scenario: From Positional to XML file on page 194.*



tFileUnarchive

tFileUnarchive Properties

Component family	File/Management	
Function	Decompresses the archive file provided as parameter and put it in the extraction directory.	
Purpose	Unarchives a file of any format (zip, rar...) that is mostlikely to be processed.	
Properties	<i>Archive file</i>	File path to the archive
	<i>Extract Directory</i>	Folder where the unarchived file is put
Java only features	<i>Use archive name as root directory / Extract file paths</i>	Check the box to reproduce the whole path to the file or if none exists create a new folder
Perl only feature	<i>Use Command line tools</i>	Check this box to use another unarchiving tool than the one provided by default in the Perl package.
Usage	This component can be used as a standalone component but it can also be used within a job as a Start component using an Iterate link.	
Limitation	n/a	

Related scenario

For **tFileUnarchive** related scenario, see *tFileCompare on page 179*.

Components

tFlowMeter

tFlowMeter



tFlowMeter Properties

Component family	Log/Error	
Function	Counts the number of rows processed in the defined flow.	
Purpose	The number of rows is then meant to be caught by the tFlowMeterCatcher for logging purpose.	
Properties	<i>Use input connection name as label</i>	Check the box to reuse the name given to the input main row flow as label in the logged data.
	<i>Mode</i>	Select the type of values for the data measured: Absolute : the actual number of rows is logged Relative : a ratio (%) of the number of rows is logged. When selecting this option, the reference
	<i>Thresholds</i>	Adds a threshold to watch proportions in volumes measured. you can decide that the normal flow has to be between low and top end of a row number range, and if the flow is under this low end, there is a bottleneck.
Usage	Cannot be used as a start component as it requires an input flow to operate.	
Limitation	n/a	

If you have a need of log, statistics and other measurement of your data flows, see *Automating stats & logs use on page 106*.

Related scenario

tFor



tFor Properties

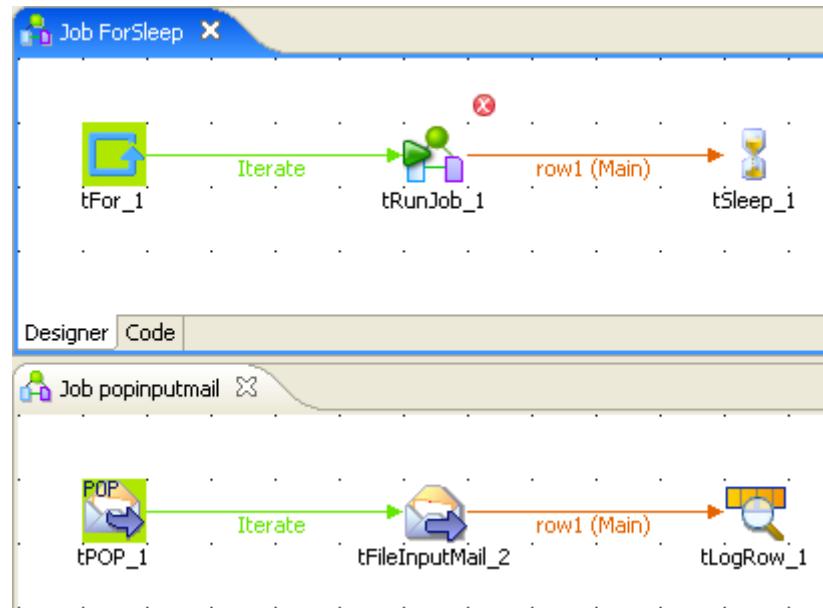
Component family	Misc	
Function	tFor iterates on a task execution.	
Purpose	tFor allows to automatically execute a task or a job based on a loop	
Properties	<i>From</i>	Type in the first instance number which the loop should start from. A start instance number of 2 with a step of 2 means the loop takes on every even number instance.
	<i>To</i>	Type in the last instance number which the loop should finish with.
	<i>Step</i>	Type in the step the loop should be incremented of. A step of 2 means every second instance.
Usage	tFor is to be used as a starting component and can only be used with an iterate connection to the next component.	
Limitation	n/a	

Scenario: Job execution in a loop

This scenario describes a job composed of a parent job and a child job. The parent job implements a loop which executes n times a child job, with a pause between each execution.

Components

tFor



- In the parent job, click and drop a **tFor**, a **tRunJob** and a **tSleep** component onto the workspace.
- Connect the **tFor** to the **tRunJob** using an **Iterate** connection.
- Then connect the **tRunJob** to a **tSleep** component using a **Row** connection.
- On the child job, click and drop the following components: **tPOP**, **tFileInputMail** and **tLogRow**.
- On the **Properties** panel of the **tFor** component, type in the instance number to start from (1), the instance number to finish with (5) and the step (1)
- On the **Properties** panel of the **tRunJob** component, select the child job in the list of stored jobs offered. In this example: *popinputmail*
- Select the context if relevant. In this use case, the context is *default* with no variables stored.
- In the **tSleep Properties** panel, type in the time-off value in second. In this example: *3 seconds*
- Then in the child job, define the connection parameters to the pop server, on the **Properties** panel.
- In the **tFileInputMail Properties** panel, select a global variable as **File Name**, to collect the current file in the directory defined in the **tPOP** component. Press **Ctrl+Space bar** to access the variable list. In this example, the variable to be used is:
\$_globals{tPOP_1}{CURRENT_FILEPATH}
- Define the **Schema**, for it to include the mail element to be processed, such as *author*, *topic*, *delivery date* and *number of lines*.
- In the **Mail Parts** table, type in the corresponding **Mail part** for each column defined in the schema. ex: *author* comes from the *From* part of the email file.

- Then connect the **tFileInputMail** to a **tLogRow** to check out the execution result on the **Run Job** view.
- Press **F6** to run the job.



tFTP

tFTP properties

Component family	Internet/FTP	
Function	This component transfers defined files via an FTP connection.	
Purpose	tFTP purposes vary according to the action selected. It can be used to get a file, put a file, remove a file or replace it on the FTP server defined.	
Properties	<i>Host</i>	FTP IP address
	<i>Port</i>	Listening port number of the FTP site.
	<i>Username and Password</i>	FTP user authentication data.
	<i>Local directory</i>	File Path. Use depends on action taken.
	<i>Remote directory</i>	File Path. Use depends on action taken.
	<i>Action</i>	List of available actions to transfer files. Related links: tFTP put , tFTP get , tFTP rename , tFTP delete on page 219.
	<i>Files</i>	Filemask of the file and New Name in case of Rename action. Wildcard character (*) can be used to transfer a set of files. Or right-click to add lines to the table.
Usage	This component is typically used as a single-component sub-job but can also be used as output or end object.	
Limitation	tFTP cannot handle both a Get and a Put action at the same time. In order to carry out both actions in parallel, duplicate the tFTP component in the job and set them differently for both actions.	

tFTP put

Purpose	tFTP copies selected files from a defined local directory to a destination remote FTP directory.
Local directory	Path to source location of the file(s).
Remote directory	Path to destination directory of the file(s).
Filemask	File names or path to the files to be transferred.

Note: If you enter a file path in the Filemask field, you don't need to fill in the local directory field.

tFTP get

Purpose	tFTP retrieves selected files from a defined remote FTP directory and copy them into a local directory .
Local directory	Path to destination location of the file.
Remote directory	Path to source directory where the files can be fetched.
Filemask	File name or path to the files to be transferred.

Note: If you enter a file path in the Filemask field, you don't need to fill in the local directory field.

tFTP rename

Purpose	tFTP remotely renames or moves files in a filesystem.
Local directory	unused in this action.
Remote directory	Source directory where the files to be renamed or moved can be fetched.
Filemask	File name or path to the files to be renamed.
New name	Enter the new name for the file.

Note: If you enter a file path in the Filemask field, you don't need to fill in the local directory field.

tFTP delete

Purpose	tFTP remotely deletes files in a filesystem.
Local directory	unused in this action.
Remote directory	Source directory where the files to be deleted are located.
Filemask	File name or path to the files to be deleted.

Note: If you enter a file path in the Filemask field, you don't need to fill in the local directory field.

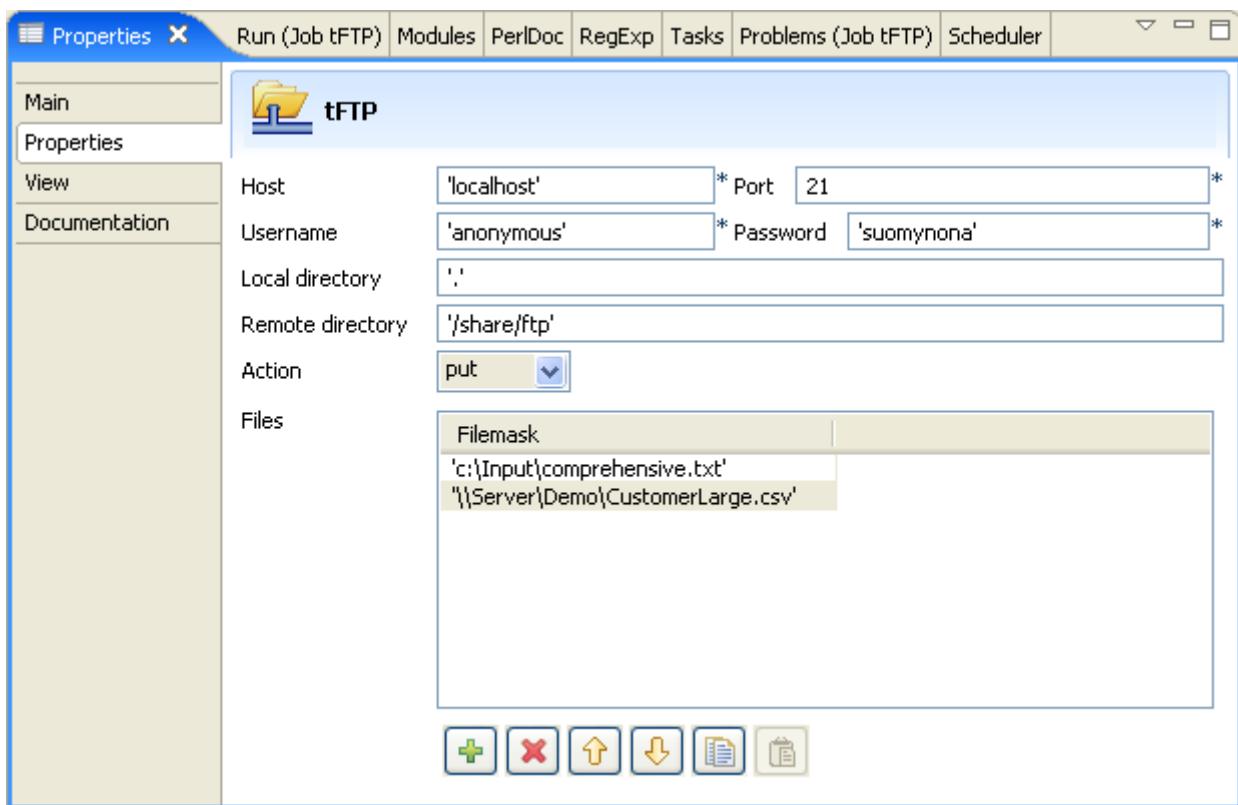
Scenario: Putting files on a remote FTP server

This scenario creates a single-component job which puts the files defined on a remote server.

- Click and drop a **tFTP** component onto the design workspace.
- Click on **Properties** tab, to define the tFTP component parameters:

Components

tFTP



- Fill in the **Host** IP address, the listening **Port** number, as well as the connection details.
- Fill in the local directory details unless you fill it directly in the different filemasks.
- Fill the details of the remote server directory.
- Select the action to be carried out, in this usecase, we'll perform a **Put** action.
- Right-click in the **Files** area, to add new lines and fill in the filemasks of all files to be copied onto the remote directory.
- Click on **Run Job** tab and execute the job.

Files defined in the Filemask are copied on the remote server.



tFuzzyMatch

tFuzzyMatch properties

Component family	Data quality	
Function	C.compares a column from the main flow with a reference column from the lookup flow and outputs the main flow data displaying the distance	
Purpose	Helps ensuring the data quality of any source data against a reference data source.	
Properties	<p><i>Schema type</i> and <i>Edit Schema</i></p> <p>A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository.</p> <p>Two read-only columns, Value and Match are added to the output schema automatically. These</p>	<p>Built-in: The schema will be created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i></p> <p>Repository: The schema already exists and is stored in the Repository, hence can be reused in various projects and job designs. Related topic: <i>Setting a repository schema on page 52</i></p>
	<i>Matching type</i>	<p>Select the relevant matching algorythm among:</p> <p>Levenshtein: Based on the edit distance theory. This calculates the number of insertion, deletion or substitution required to match the reference</p> <p>Metaphone: Based on the phonetics. It first loads the phonetics of all entries of the lookup reference and checks all entries of the main flow against it.</p> <p>Double Metaphone: If disambiguation is required in Metaphone, use this option..</p>
	<i>Min Distance</i>	(Levenshtein only) Set the minimum number of changes allowed to match the reference. If set to 0, only perfect matchs are returned.
	<i>Max Distance</i>	(Levenshtein only) Set the maximum number of changes allowed to match the reference.

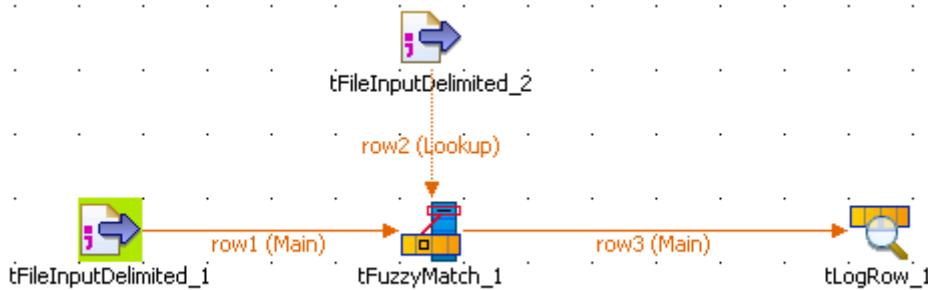
Components

tFuzzyMatch

	<i>Matching Column</i>	Select the column of the main flow that needs to be checked against the reference (lookup) key column
	<i>Unique Matching</i>	Check this box if you want to get the best match possible, in case several matches are available.
	<i>Matching item separator</i>	In case several matches are available, all of them are displayed unless the unique match box is checked. Define the delimiter between all matchs.
Usage	This component is not startable (green background) and it requires two input components and an output component.	
Limitation/prerequisite	Perl users: Make sure the relevant packages are installed. Check the Module view for modules to be installed	

Scenario 1: Levenshtein distance of 0 in first names

This scenario describes a four-component job aiming at checking the edit distance between the First Name column of an input file with the data of the reference input file. The output of this Levenshtein type check is displayed along with the content of the main flow on a



- Drag and drop the following components from the Palette to the workspace:
tFileInputDelimited (x2), **tFuzzyMatch**, **tFileOutputDelimited**.
- Define the first **tFileInputDelimited** properties. Browse the system to the input file to be analysed and most importantly set the schema to be used for the flow to be checked.
- In the schema, set the **Type** of data in the Java version, especially if you are in **Built-in** mode.
- Link the defined input to the **tFuzzyMatch** using a **Main** row link.
- Define the second **tFileInputDelimited** component the same way.

WARNING—Make sure the reference column is set as key column in the schema of the lookup flow.

Schema of Reference							
Column	Key	Type	Nullable	Length	Precision	Comment	
01_firstname	<input checked="" type="checkbox"/>	String	<input checked="" type="checkbox"/>	13			
02_gender	<input type="checkbox"/>	String	<input checked="" type="checkbox"/>	3			

- Then connect the second input component to the **tFuzzyMatch** using a main row (which displays as a **Lookup** row on the workspace).
- Select the **tFuzzyMatch** properties.
- The **Schema** should match the **Main** input flow schema in order for the main flow to be checked against the reference.

tFuzzyMatch_1 (Output)							
	Column	Key	Type	Nullable	Length	Precis...	Com...
	01_firstname	<input type="checkbox"/>	String	<input checked="" type="checkbox"/>	13		
	VALUE	<input type="checkbox"/>	String	<input checked="" type="checkbox"/>	255		
	MATCHING	<input type="checkbox"/>	String	<input checked="" type="checkbox"/>	255		

- Note that two columns, **Value** and **Matching**, are added to the output schema. These are standard matching information and are read-only.
- Select the method to be used to check the incoming data. In this scenario, *Levenshtein* is the **Matching type** to be used.
- Then set the distance. In this method, the distance is the number of char changes (insertion, deletion or substitution) that needs to be carried out in order for the entry to fully match the reference.

tFuzzyMatch_1

Schema Type	Built-In	Edit schema	Sync columns
Matching type	Levenshtein	* Case sensitive	<input type="checkbox"/>
Min. distance	0	* Max. distance	0
Matching column	01_firstname	<input type="checkbox"/>	
<input type="checkbox"/> Unique matching	Matching item separator	::	

- In this use case, we want the distance be of 0 for the min. or for the max. This means only the exact matches will be output.
- Also, uncheck the **Case sensitive** box.
- And select the column of the main flow schema that will be checked. In this example, the first name.

Components

tFuzzyMatch

- No need to check the **Unique matching** nor hence the separator.
- Link the **tFuzzyMatch** to the standard output **tLogRow**. No other parameters than the display delimiter is ot be set for this scenario.
- Save the job and press **F6** to execute the job.

```
audra {2}||  
audrea||  
audrey||  
august (1)||  
august (2)||  
augusta||  
auguste|0|auguste  
augustijn||  
augustin|0|augustin  
augustine||  
augusto||  
augsts||  
augustus||  
augustini||
```

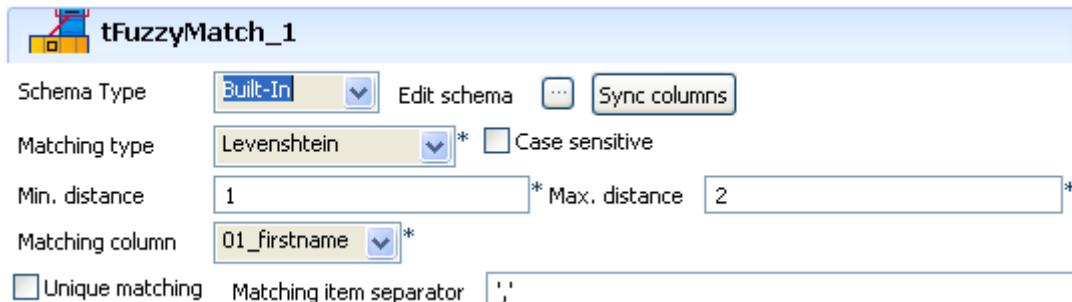
As the edit distance has been set to 0 (min and max), the output shows the result of a regular join between the main flow and the lookup (reference) flow, hence only full matchs with Value of 0 are displayed.

A more obvious example is with a minimum distance of 1 and a max. distance of 2, see *Scenario 2: Levenshtein distance of 1 or 2 in first names on page 224*.

Scenario 2: Levenshtein distance of 1 or 2 in first names

This scenario is based on the scenario 1 described above. Only the min and max distance settings in **tFuzzyMatch** component get modified, which will change the output displayed.

- In the Properties panel of the **tFuzzyMatch**, change the min distance from 0 to 1. This excludes straight away the exact matchs (which would show a distance of 0).
- Change also the max distance to 2 as the max distance cannot be lower than the min distance. The output will provide all matching entries showing a discrepancy of 2 characters at most.



- No other change of the setting is required.
- Make sure the Matching item separator is defined, as several references might be matching the main flow entry.

- Save the new job and press F6 to run it.

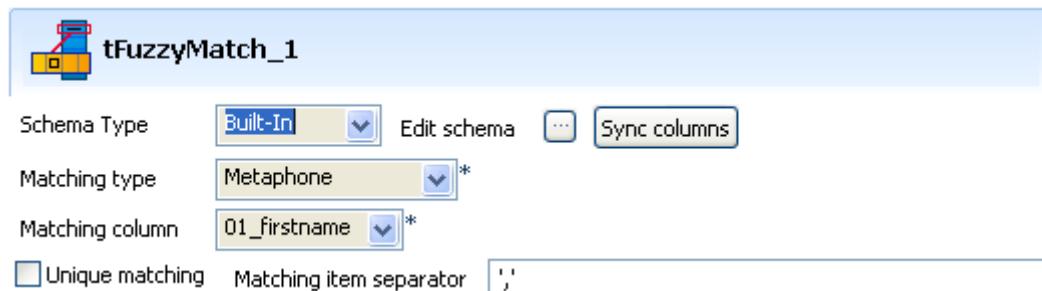
```
audrea|2|aude
audrey|2|aude
august (1) ||
august (2) ||
augusta|1|auguste
auguste|2|augustin
augustijn|1|augustin
augustin|2|auguste
augustine|1|augustin
augusto|1|auguste
augsts|1|auguste
augustus|2|auguste,augustin
aukusti|2|auguste,augustin
aulay ||
aulus|2|jules
```

As the edit distance has been set to 2, some entries of the main flow match several reference entries.

You can also use another method, the metaphone, to assess the distance between the main flow and the reference,

Scenario 3: Metaphonic distance in first name

This scenario is based on the scenario 1 described above.



- Change the **Matching type** to **Metaphone**. There is no min nor max distance to set as the matching method is based on the discrepancies with the phonetics of the reference.
- Save the job and press **F6**. The phonetics value is displayed along with the possible matchs.

```
audrey ||
august (1)|AKST|auguste
august (2)|AKST|auguste
augusta|AKST|auguste
auguste|AKST|auguste
augustijn ||
augustin|AKSTN|augustin
augustine|AKSTN|augustin
augusto|AKST|auguste
augsts ||
augustus ||
aukusti|AKST|auguste
aulay ||
aulus ||
aune ||
```

tLogCatcher

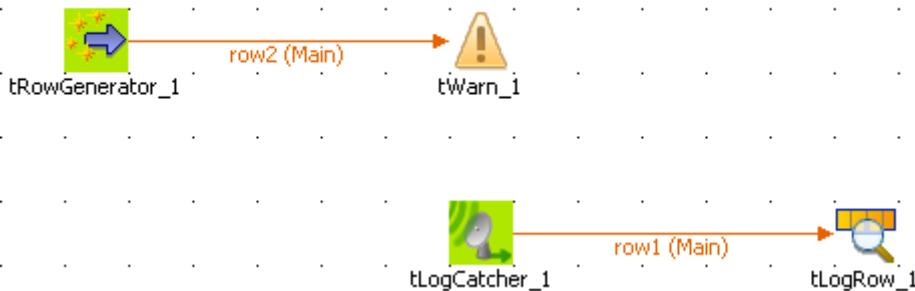
 Both **tDie** and **tWarn** components are closely related to the **tLogCatcher** component. They generally make sense when used alongside a tLogCatcher in order for the log data collected to be encapsulated and passed on to the output defined.

tLogCatcher properties

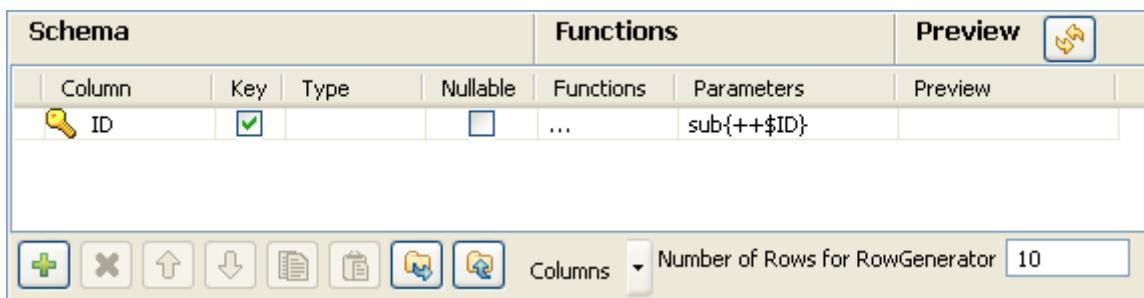
Component family	Log & Error	 
Function	Fetches set fields and messages from PerlDie, tDie and/or tWarn and passes them on to the next component.	
Purpose	Operates as a log function triggered by one of the three: PerlDie, tDie or tWarn, to collect and transfer log data.	
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository.
		Built-in: The schema will be created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>
		Repository: The schema already exists and is stored in the Repository, hence can be reused in various projets and job flowcharts. Related topic: <i>Setting a repository schema on page 52</i>
	<i>Catch PerlDie</i>	Check this box to trigger the tCatch function when a PerlDie occurs in the job
	<i>Catch tDie</i>	Check this box to trigger the tCatch function when a tDie is called in a job
	<i>Catch tWarn</i>	Check this box to trigger the tCatch function when a tWarn is called in a job
Usage	This component is the start component of a secondary job which automatically triggers at the end of the main job	
Limitation	n/a	

Scenario1: warning & log on entries

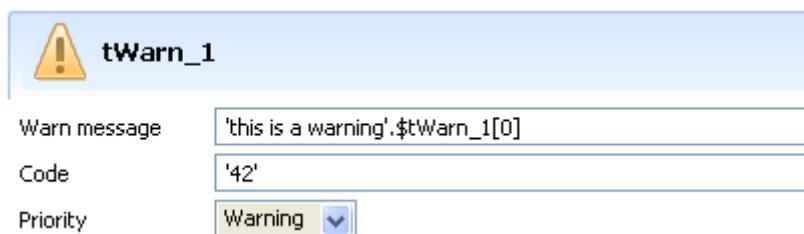
In this basic scenario made of three components, a **tRowGenerator** creates random entries (id to be incremented). The input hits a **tWarn** component which triggers the **tLogCatcher** subjob. This subjob fetches the warning message as well as standard predefined information and passes them on to the **tLogRow** for a quick display of the log data.



- Click and drop a **tRowGenerator**, a **tWarn**, a **tLogCatcher** and a **tLogRow** from the Palette, on your workspace
- Connect the **tRowGenerator** to the **tWarn** component.
- Connect separately the **tLogCatcher** to the **tLogRow**.
- On the **tRowGenerator** editor, set the random entries creation using a basic Perl function:



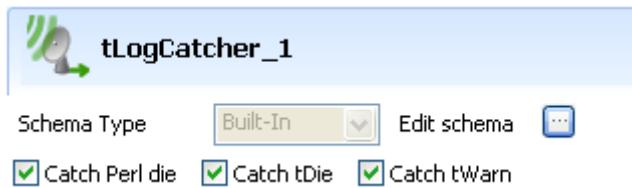
- On the **tWarn** Properties panel, set your warning message, the code the priority level. In this case, the message is “this is a warning”.
- For this scenario, we will concatenate a Perl function to the message above, in order to collect the first value from the input table.



- On the **tLogCatcher** properties panel, check the **tWarn** box in order for the message from the latter to be collected by the subjob.
- Click **Edit Schema** to view the schema used as log output. Notice that the log is comprehensive.

Components

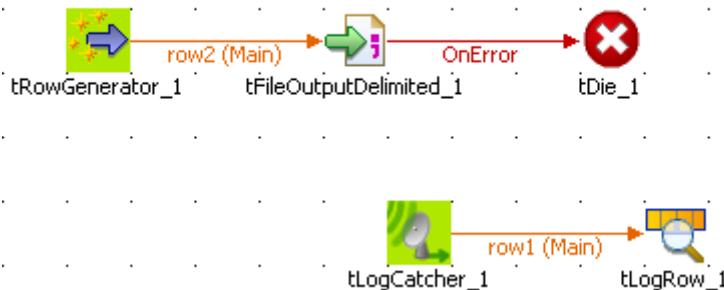
tLogCatcher



Press **F6** to execute the job. Notice that the Log produced is exhaustive.

Scenario 2: log & kill a job

This scenario uses a **tLogCatcher** and a **tDie** component. A **tRowGenerator** is connected to a **tFileOutputDelimited** using a Row link. On error, the **tDie** triggers the catcher subjob which displays the log data content on the **Run Job** console.



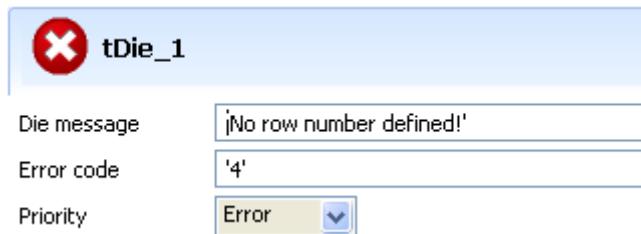
- Click and drop all required components from various folders of the **Palette**: **tRowGenerator**, **tFileOutputDelimited**, **tDie**, **tLogCatcher**, **tLogRow**.
- On the tRowGenerator properties panel, define the setting of the input entries to be handled.

Schema		Functions			Preview	
Column	Key	Type	Nullable	Func...	Para...	Preview
id	<input checked="" type="checkbox"/>	int	<input checked="" type="checkbox"/>	...	sub{\$...	
name	<input type="checkbox"/>	String	<input type="checkbox"/>	...	sub{'l...	
quantity	<input type="checkbox"/>	int	<input type="checkbox"/>	...	1..1000	
flag	<input type="checkbox"/>	int	<input type="checkbox"/>	...	0,1	
creation	<input type="checkbox"/>	Day	<input type="checkbox"/>	getDate	forma...	

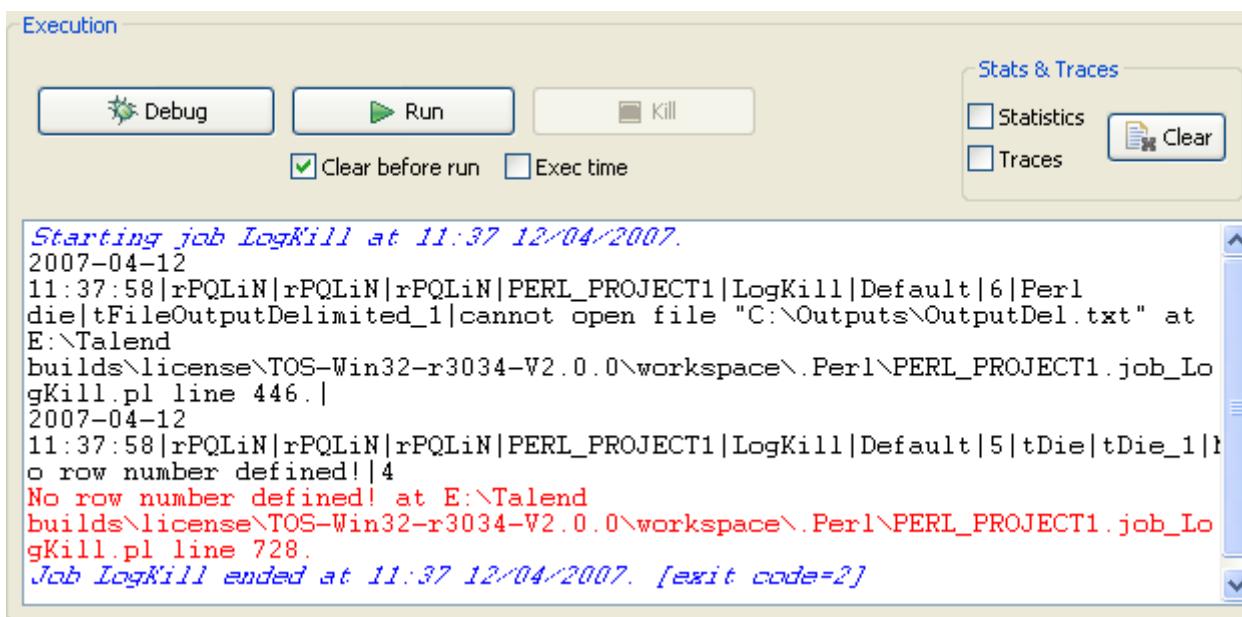
Buttons at the bottom: +, -, up, down, file, save, preview, columns, number of rows for RowGenerator (0).

- Edit the schema and define the following columns as random input examples: *id*, *name*, *quantity*, *flag* and *creation*.
- Set the **Number of rows** onto 0. This will constitute the error which the Die operation is based on.
- On the **Values** table, define the **Perl array** functions to feed the input flow.

- Define the **tFileOutputDelimited** to hold the possible output data. The row connection from the tRowGenerator feeds automatically the output schema. The separator is a simple semi-colon.
- Connect this output component to the tDie using a **Trigger > If** connection. Double-click on the newly created connection to define the if:
 $\$_globals\{tRowGenerator_1\}\{NB_LINE\} \leq 0$
- Then double-click to select and define the **Properties** of the **tDie** component.



- Enter your **Die** message to be transmitted to the **tLogCatcher** before the actual kill job operation happens.
- Next to the job but not physically connected to it, click and drop a **tLogCatcher** and connect it to a **tLogRow** component.
- Define the tLogCatcher properties. Make sure the **tDie** box is checked in order to add the Die message to the Log information transmitted to the final component.



- Press **F6** to run the job and notice that the log contains a black message and a red one.
- The black log data come from the tDie and are transmitted by the tLogCatcher. In addition the normal PerlDie message in red displays as a job abnormally died.

Components

tLogRow

tLogRow



tLogRow properties

Component family	Log & Error	
Function	Displays data or results in the Run Job console	
Purpose	tLogRow helps monitoring data processed.	
Properties	<i>Print values in table cells</i>	
	Separator	Enter the separator which will delimit data on the Log display
	Print component unique name in front of each output row	Check this box in case several LogRow components are used. Allows to differentiate outputs
	Print schema column name in front of each value	Check this box to retrieve column labels from output schema.
	Use fixed length for values	Check this box to set a fixed width for the value display.
Usage	This component can be used as intermediate step in a data flow or as a n end object in the job flowchart.	
Limitation	n/a	

Scenario: Delimited file content display

Related topics using a **tLogRow** component:

- **tFileInputDelimited** *Scenario: Delimited file content display on page 189.*
- **tContextLoad** *Scenario: Dynamic context use in MySQL DB insert on page 123*
- **tWarn, tDie, tLogCatcher** *Scenario1: warning & log on entries on page 226 and Scenario 2: log & kill a job on page 228*



tMap

tMap properties

Component family	Processing	 
Function	tMap is an advanced use component, which integrates itself as plugin to JasperETL .	
Purpose	tMap transforms and routes data from single or multiple sources to single or multiple destinations.	
Properties	Preview	The preview is an instant shot of the Mapper data. It becomes available when Mapper properties have been filled in with data. The preview synchronization takes effect only after saving changes.
	Mapping links display as	Auto : the default setting is curves links Curves : the mapping display as curves Lines : the mapping displays as straight lines. This last option allows to slightly enhance performance.
	Map editor	Mapper is the tMap editor. It allows you to define the tMap routing and transformation properties.
Usage	Possible uses are from a simple reorganisation of fields to the most complex jobs of data multiplexing or demultiplexing transformation, concatenation, inversion, filtering and more...	
Limitation	The use of tMap supposes minimum Perl or Java knowledge in order to fully exploit its functionalities. This component is a junction step, and for this reason cannot be a start nor end component in the job	

Note: For further information, see *Mapping data flows in a job on page 80*.

Scenario 1: Mapping with filter and simple explicit join (Perl)

The job described below aims at reading data from a csv file stored in the Repository, looking up at a reference file also stored remotely, then extracting data from these two files based on defined filters to an output file or a reject file.

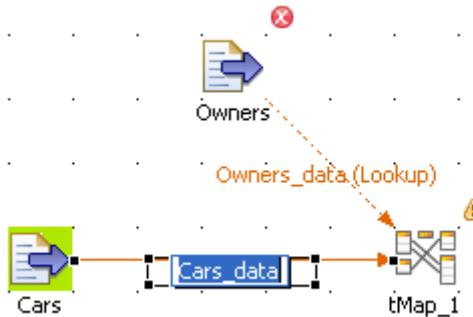
This job is presented in Perl but could also be carried out in Java.

- Click on **File** in the Palette of components, select **tFileInputCSV** and drop it on the design area. Rename the Label to *Cars*, either by double-clicking on the label in the workspace or via the **View** tab of the Properties panel.
- Repeat this operation, and rename this second input component: *Owners*.

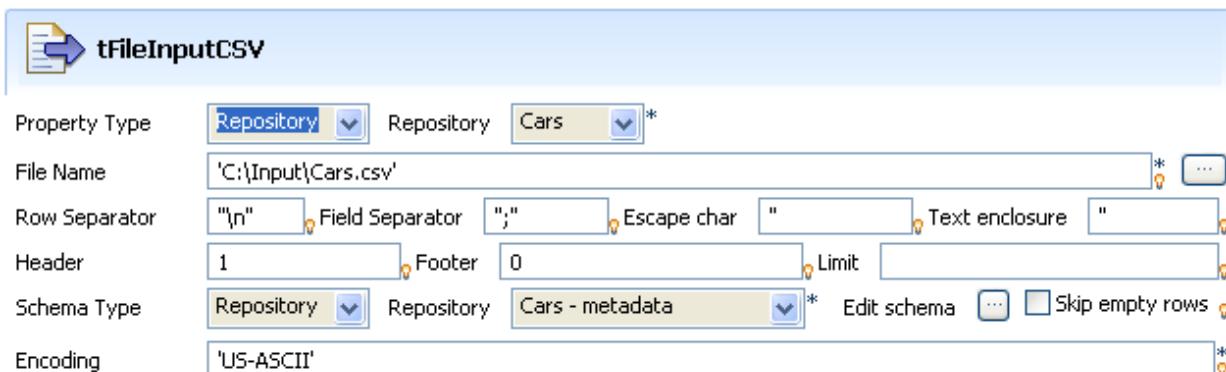
Components

tMap

- Click on **Processing** in the Palette of components, select **tMap** and drop it on the design area.
- Connect the two Input components to the mapping component and customize the row connection labels.



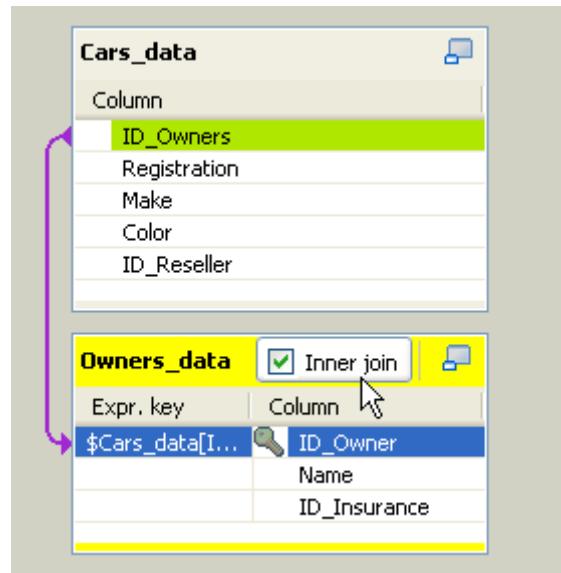
- The Cars and Owners delimited files metadata are defined in the Metadata area of the repository. You can hence use their Repository reference in the Properties panel.
- Double-click on *Cars*, to set the Properties panel.



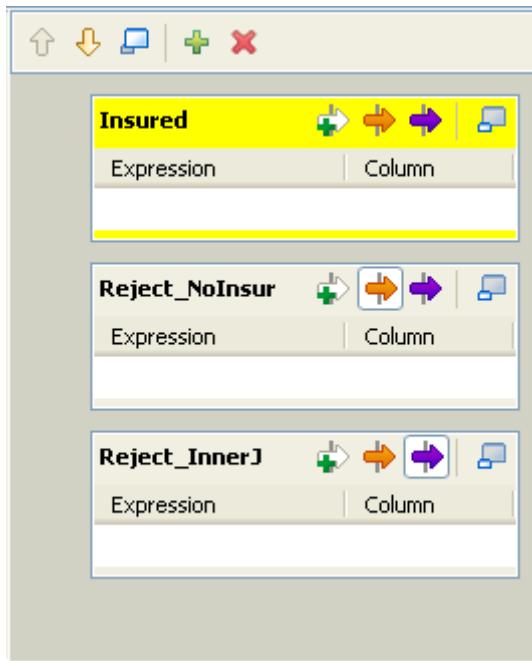
- Select **Repository** to retrieve **Property type** as well as **Schema type**. The rest of the fields gets automatically filled in appropriately when you select the relevant Metadata entry on the list.
- Double-click on the Owners component and repeat the setting operation. Select the corresponding Metadata entry.

For further information regarding Metadata creation in the Repository, see *Defining Metadata items* on page 54.

- Then double-click on the **tMap** component to open the Mapper. Notice that the Input area is already filled with the Input component metadata tables and that the top table is the Main flow table.
- Notice also that the respective row connection labels display on the top bar of the tables.
- Create a Join between the two tables on the ID_Owner field by simply dragging the top table ID_Owner field to the lookup input table.



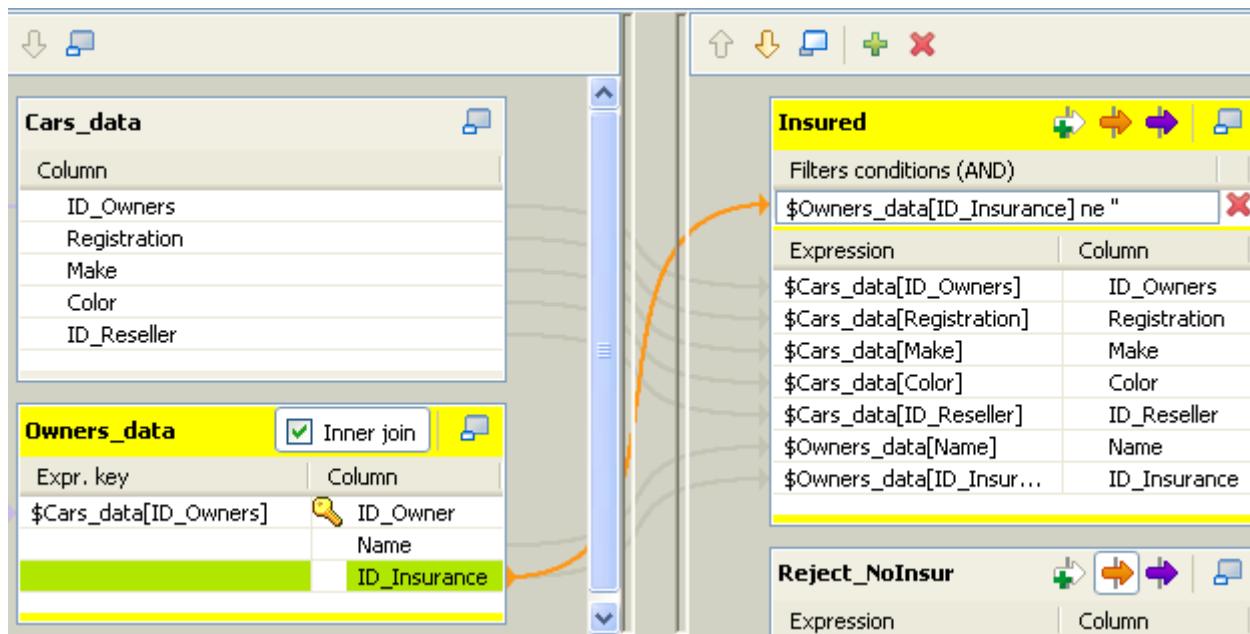
- Define this link as an Inner Join by checking the box.
- Click on the **Plus** button on the Output area of the Mapper to add three Output tables



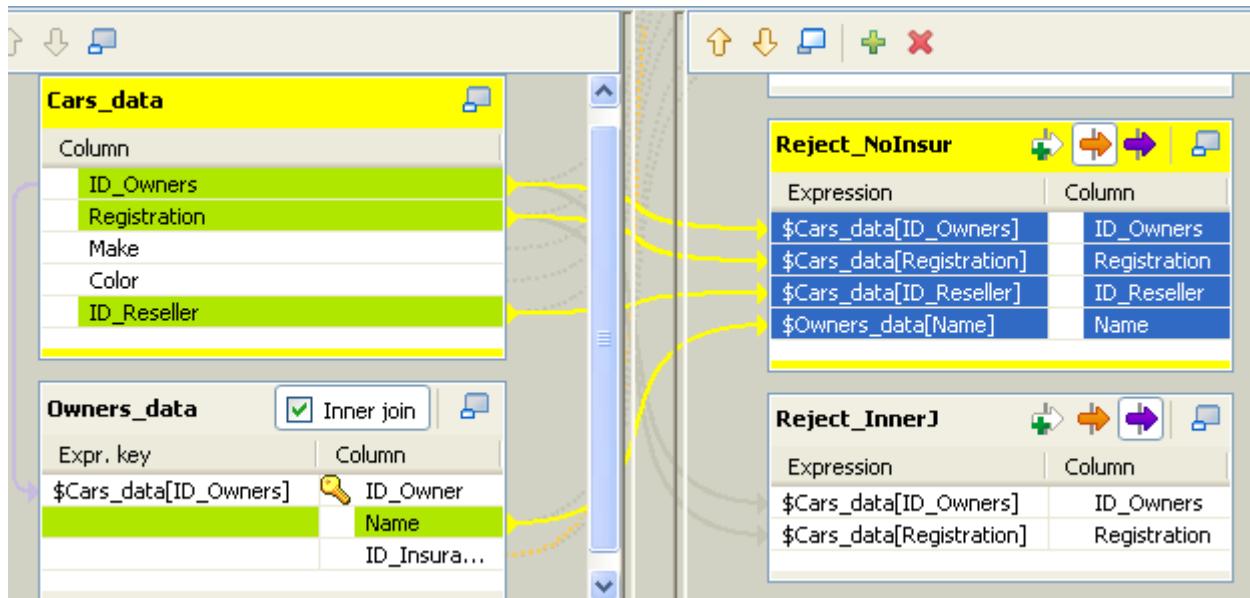
- Drag and drop Input content to fill in the first output schema. For more information regarding data mapping, see *Mapping data flows in a job on page 80*.
- Click on the plus arrow button to add a filter row. In the Insured table, will be gathered cars and owners data which include an Insurance ID.

Components

tMap

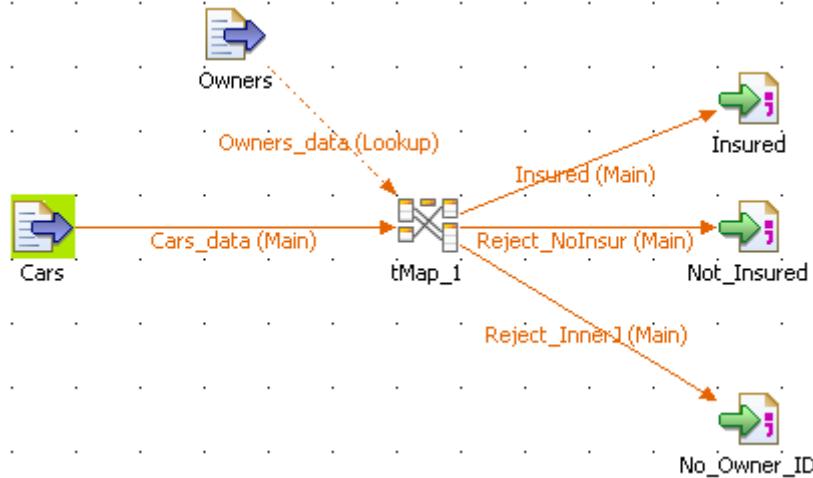


- Therefore drag the ID_Insurance field to the Filter condition area and enter the formula used meaning ‘not undefined’: `$Owners_data[ID_Insurance] ne ''`
- The Reject_NoInsur table is a standard reject output flow containing all data that do not satisfy the required filter condition. Click the orange arrow to set the table as Reject Output.

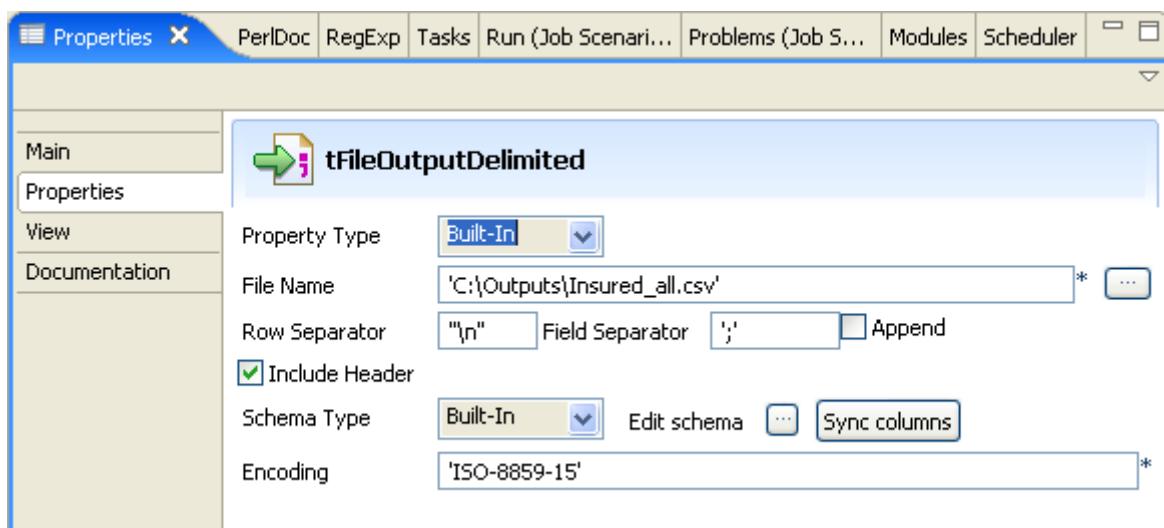


- The third and last table gathers the schema entries whose Inner Join could not be established. One Owners_ID from the Car database does not match any Owner_ID from the Owner file.
- Click the violet arrow button to set the last table as the Inner Join Reject output flow.
- Click OK to validate and come back to the design area.

- Add three **tFileOutputDelimited** components to the workspace and right-click on the tMap to connect the Mapper with all three output components, using the relevant Row connection.
- Relabel the three output components accordingly.



- Then double-click on each of them, one after the other, in order to define their respective output filepath. If you want a new file to be created, browse to the destination output folder, and type in a file name including the extension.
- Check the **Include header** box to reuse the column labels from the schema as header row in the output file.



- Run the Job using the F6 keystroke or via the **Run Job** panel.
- Output files have been created if need be.

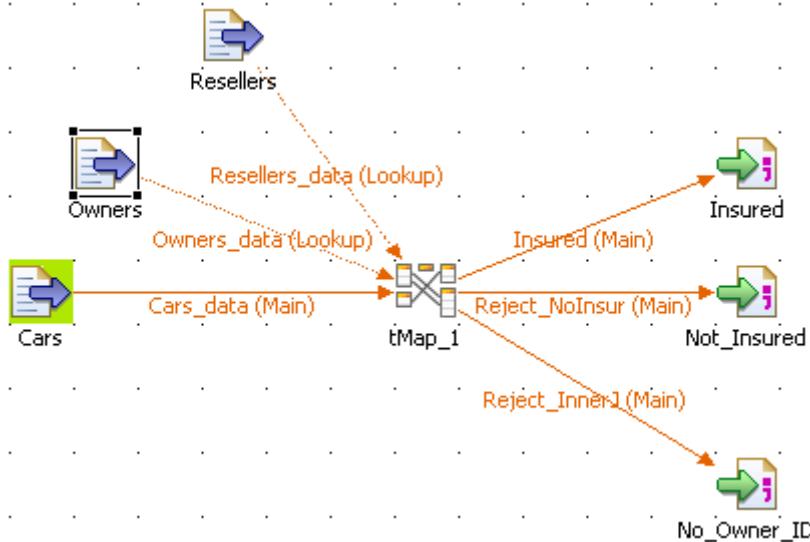
The screenshot shows a window titled "Not_Insured.csv - Bloc-notes". The menu bar includes "Fichier", "Edition", "Format", "Affichage", and "?". The content area displays a list of rows from a CSV file:

ID_Owners	Registration	ID_Reseller	Name
10;7040	AS 24;;vaesmont		
21;5177	GC 89;1;carbo		
25;7163	NT 90;2;sabmau		
28;1335	AP 27;;bouhnau		
30;2573	ID 63;1;mauneng		
40;8386	GH 71;8;carmau		
47;4080	LS 44;7;sabneng		
53;4597	NL 94;7;hirtvaes		
58;1120	WH 42;;kensab		
63;5878	FG 10;5;nengken		
76;7526	XP 68;5;lebone		
81;0247	HG 17;;sabmont		
82;3962	SM 31;4;otmont		
83;6518	HH 86;;bogall		
96;3799	DB 43;2;oinemau		

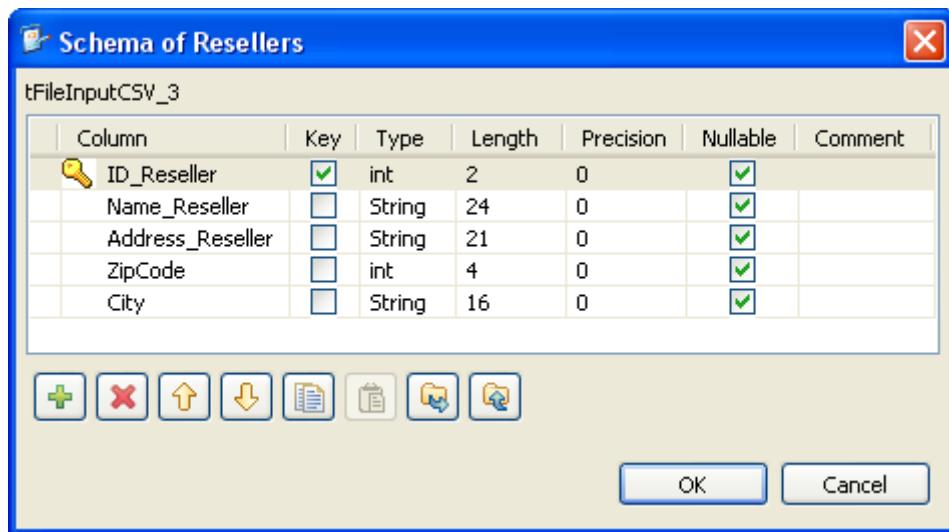
Scenario 2: Mapping with Inner join rejection (Perl)

This scenario, based on scenario 1, adds one input file containing Resellers details and extra fields in the main Output table. Two filters on Inner Joins are added to gather specific rejections.

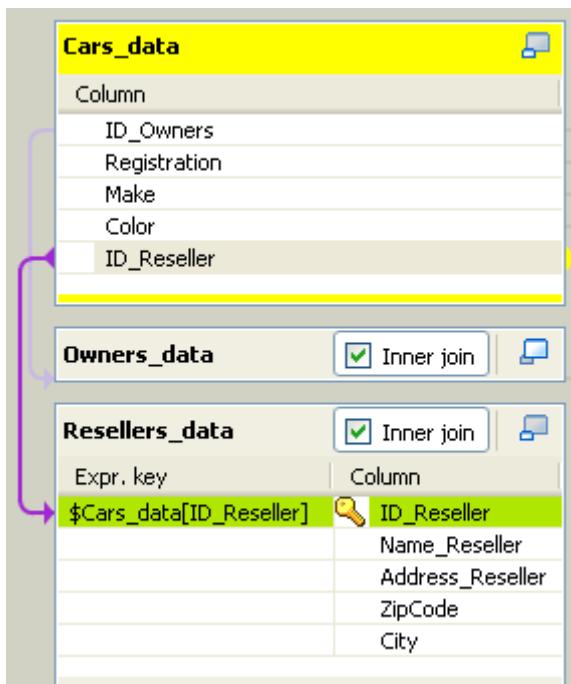
- Click on **File** in the Palette of Components, and drop a **tFileInputCSV** component on the workspace.
- Connect it to the Mapper and put a label on the connection.



- Double-click on the Resellers component, to define the Reseller input properties.
- Browse to the Resellers.csv file. Edit the schema and add the columns as needed to match the file structure.



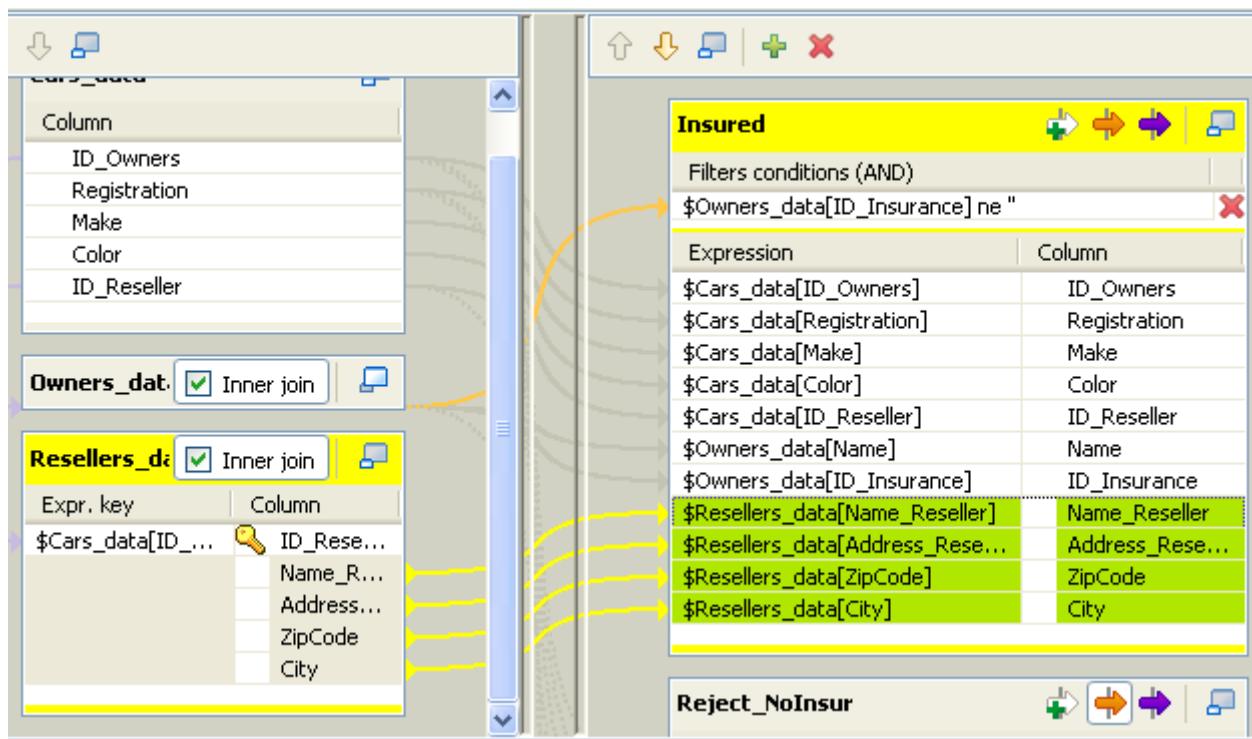
- You could also create a metadata entry for this file description and select Repository as properties and schema type. For further information, see *Setting up a File Delimited schema on page 59*.
- Double-click on the tMap component and notice that the schema is added on the Input area.



- Create a join between the main input flow and the resellers input. Check the Inner Join box to define that an Inner Join Reject output is to be created.
- Drag & drop the fields from the Resellers table to the main Output table.

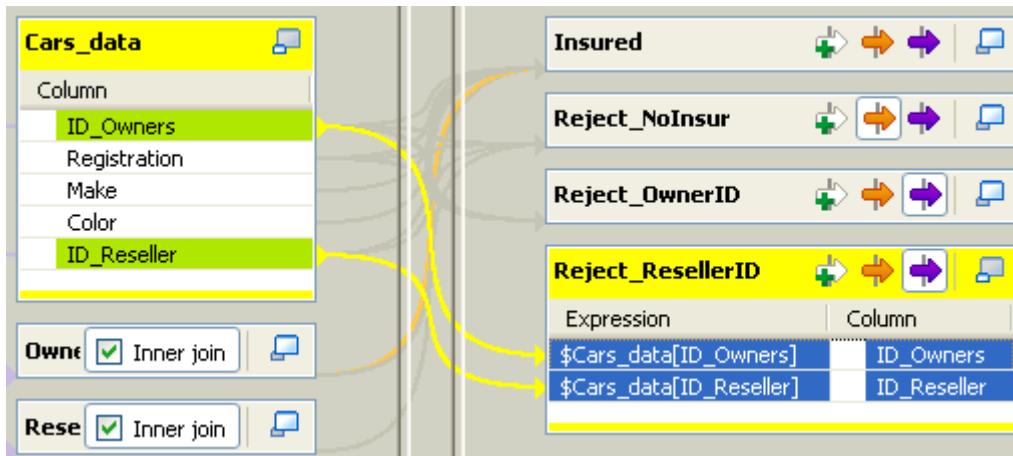
Components

tMap

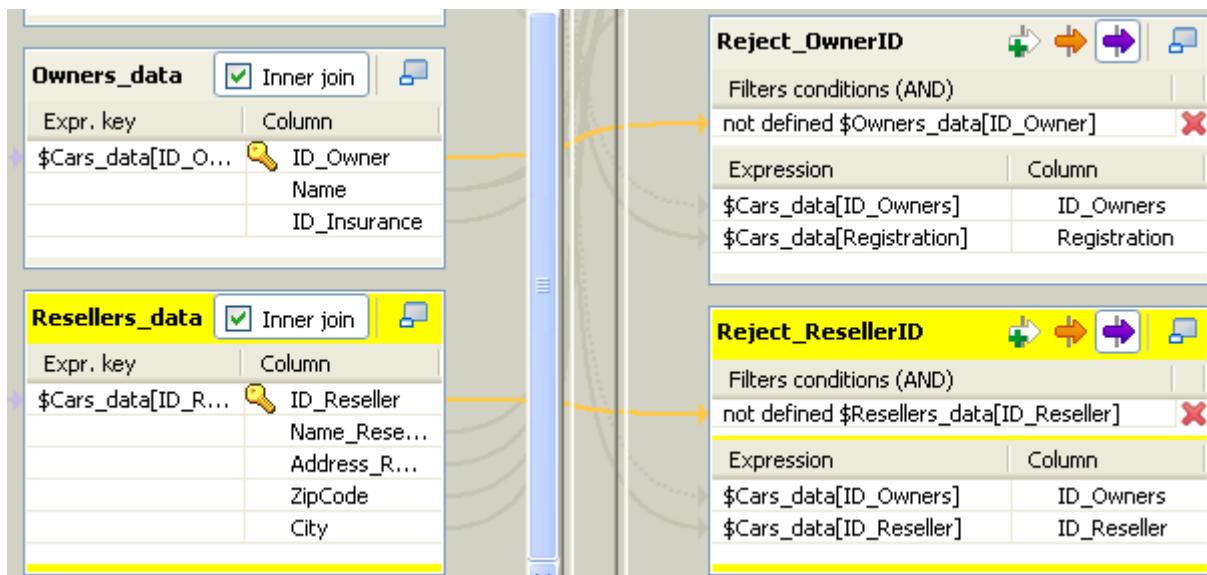


Note: When two inner joins are defined, you either need to define two different inner join reject tables to differentiate both rejections or if there is only one Inner Join reject output, both Inner Join rejections will be stored in the same output.

- On the output area, click on the plus button to add a new output table.
- Give a name to this new Output connection: Reject_ResellerID
- Click the Inner Join reject button to define this new output table as Inner Join Reject output.
- Drag & drop two fields from the main input flow (*Cars*) to the new reject output table. In this case, if the Inner Join cannot be established, these data (*ID_Cars* & *ID_resellers*) will be gathered in the output file and will help identify the bottleneck.



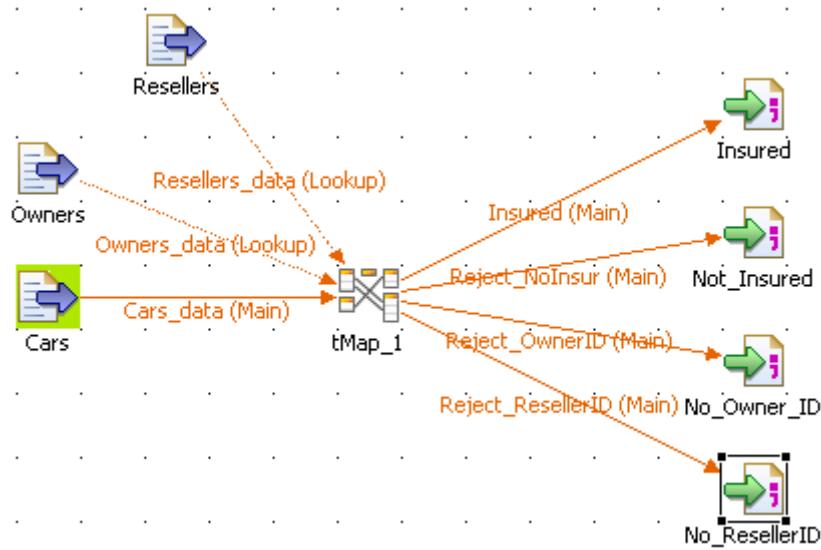
- Now apply filters on the two Inner Join reject outputs, in order for to distinguish both types of rejection.
- In the first Inner Join output table (*Reject_OwnerID*), click the plus arrow button to add a filter line and fill it in with the following formula to gather only OwnerID-related rejection: *not defined \$Owners_data[ID_Owner]*
- In the second Inner Join output table (*Reject_ResellerID*), repeat the same operation using the following formula: *not defined \$Resellers_data[ID_Reseller]*



- Click OK to validate and close the Mapper editor.
- Right-click on the **tMap** component, click on **Row** and select *Reject_ResellerID* in the list.
- Connect the main row from the Mapper to the Reseller Inner Rejection output component

Components

tMap



- For this scenario, remove from the Resellers.csv the rows corresponding to Reseller ID 5 and 8.
- Then run the job through a F6 key stroke or from the Run Job panel.

The image shows three separate windows, each titled "Bloc-notes" and displaying CSV file contents:

- Cars.csv - Bloc-notes:** Contains a list of car details separated by semicolons. The data includes:
 - 1;5776 ZQ 94;volkswagen;gold;7
 - 2;9983 TW 80;Honda;purple;7
 - 3;2580 TT 77;Renault;orange;1
 - 4;1723 YF 11;Citroen;silver;10
 - 5;4178 EL 94;Citroen;blue;4
 - 6;2777 UI 68;Citroen;yellow;8
 - 7;1225 GO 26;Toyota;purple;4
 - 8;9873 DF 59;Citroen;yellow;4
 - 9;0921 OM 37;Volkswagen;green;8
 - 10;7040 AS 24;Honda;orange;3
 - 11;8630 KS 58;Mercedes;yellow;2
 - 12;4322 DP 76;BMW;purple;1
 - 13;2373 BC 30;BMW;black;9
- Reseller.csv - Bloc-notes:** Contains a list of reseller details separated by semicolons. The data includes:
 - 1;Cars & Pickup Shop;38 Boot Avenue;5113;North Coast City
 - 2;Cars & Pickup Specialist;15 Rubber Drive;5952;West Coast City
 - 3;Quality Car Resale;29 Wipers Road;7794;Atlantic City
 - 4;All you need Outlet;45 Rubber Gate;5987;Caribbean City
 - 6;All Cycle Resale;20 Tyre Road;6593;Caribbean City
 - 7;Cars & Pickup Resale;35 Tyre Road;2486;Caribbean City
 - 9;All Cycle Specialist;30 Windshield Street;9219;North Coast City
 - 10;All Cycle outlet;45 Rubber Avenue;5529;Pacific City
- No_ResellerID.csv - Bloc-notes:** Contains a list of ID_Owners and ID_Reseller pairs separated by semicolons. The data includes:
 - 6;8
 - 9;8
 - 28;5
 - 29;8
 - 40;8
 - 48;8
 - 57;5
 - 60;5
 - 63;5
 - 69;8
 - 72;8

- The four output files are all created in the defined folder (Outputs).
- Notice in the Inner Join reject output file, *NoResellerID.csv*, that the ID_Owners field values matching the Reseller ID 5 and 8 were rejected from the cars file to this file.

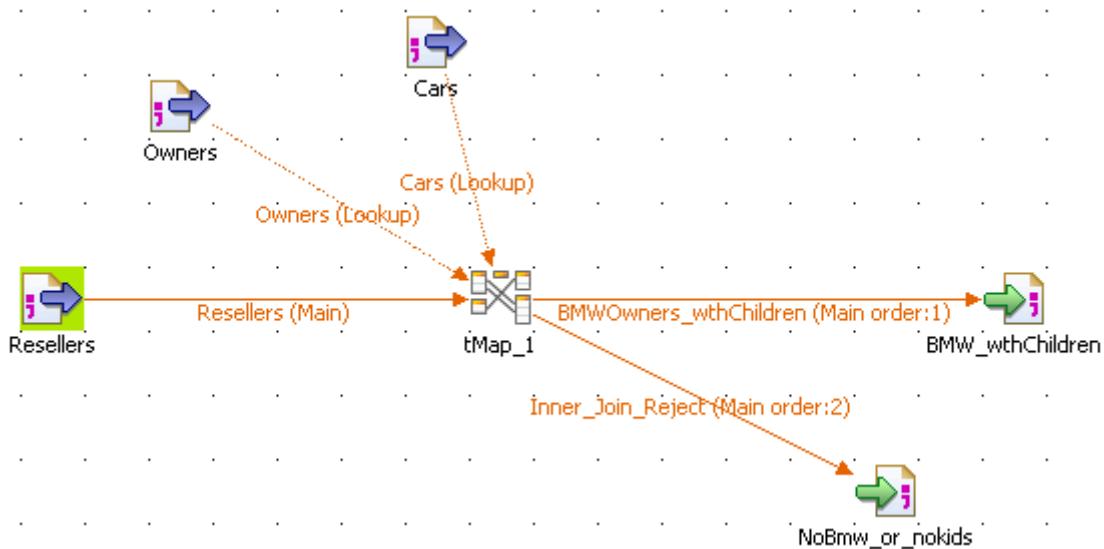
Scenario 3: Cascading join mapping

As third advanced use scenario, based on the scenario 2, add a new Input table containing Insurance details for example.

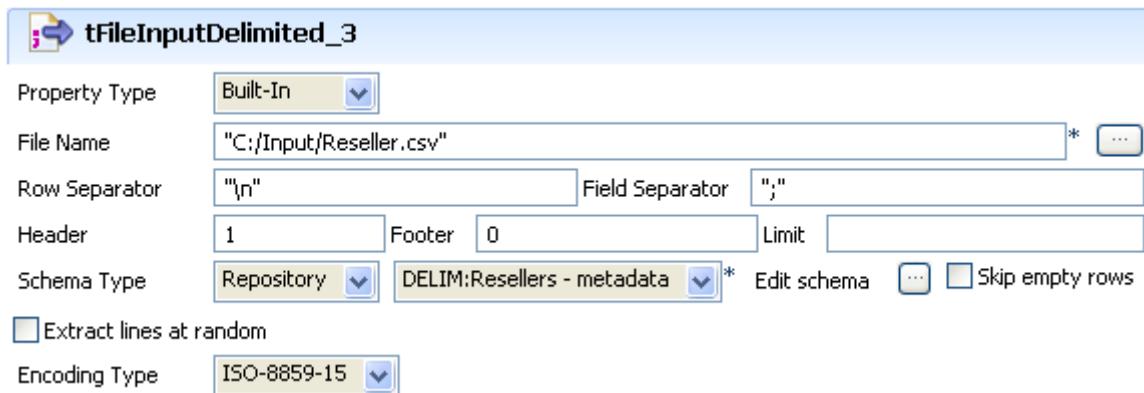
Set up an Inner Join between two lookup input tables (Owners and Insurance) in the Mapper to create a cascade lookup and hence retrieve Insurance details via the Owners table data.

Scenario 4: Advanced mapping with filters, explicit joins and Inner join rejection (Java)

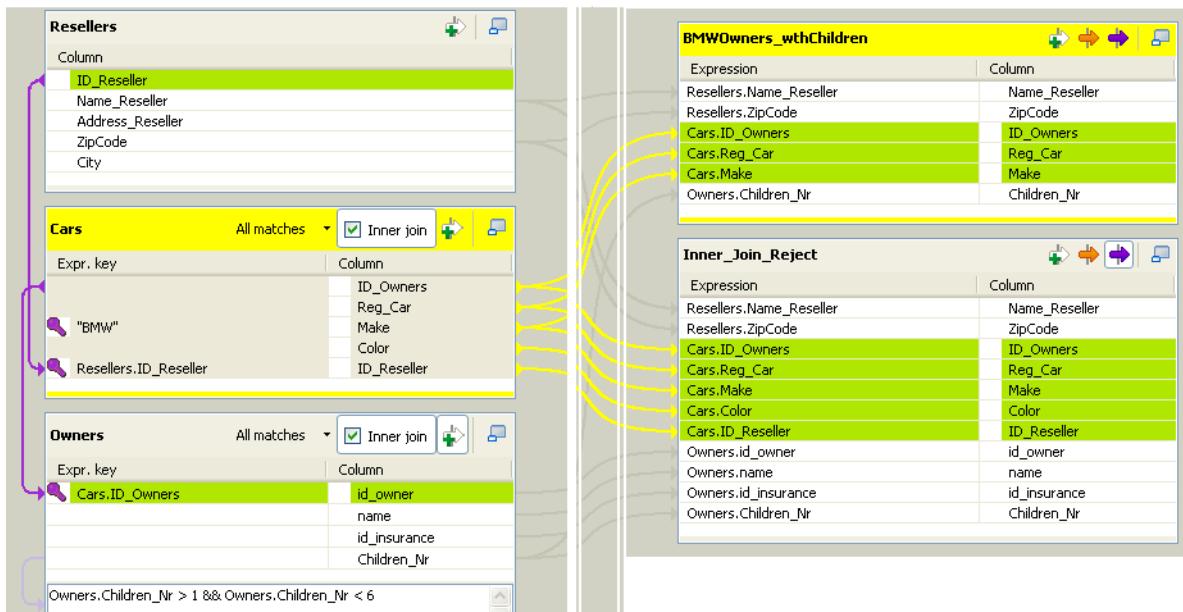
This scenario introduces a job in Java tMap which allows to find the reseller's customer leads who are owners of a defined make, and have between 2 and 6 children (inclusive), for upsale purpose for example.



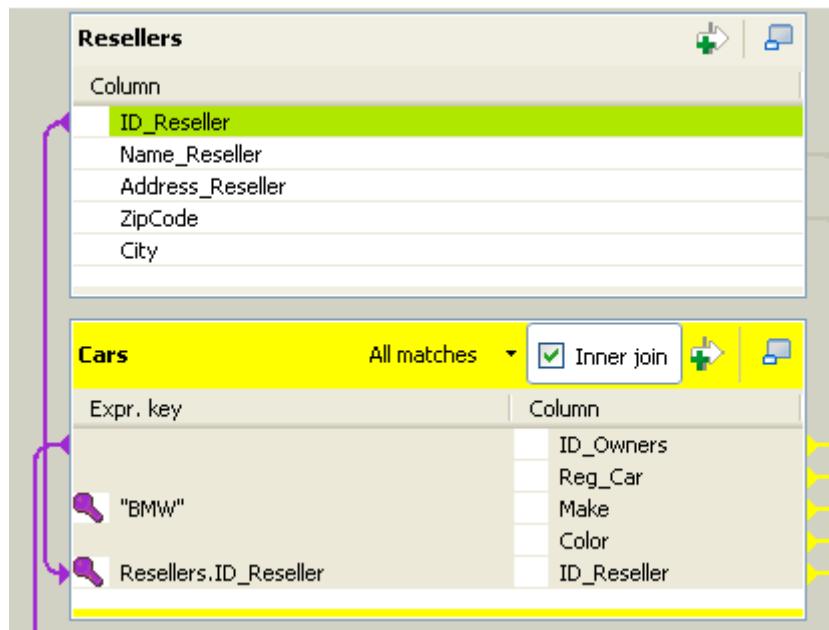
- Drag and drop the following components from the Palette: **tFileInputDelimited** (x3), **tMap**, **tFileOutputDelimited** (x2)
- Connect the input flow components to the tMap using a **Main row** connection. Pay attention to the file you connect first as it will automatically be set as **Main** flow. And all other connections will thus become **Lookup** flows.
- Define the **Properties** of each of the Input components. For example, define the *Resellers* file path used as **Main** flow in the job.



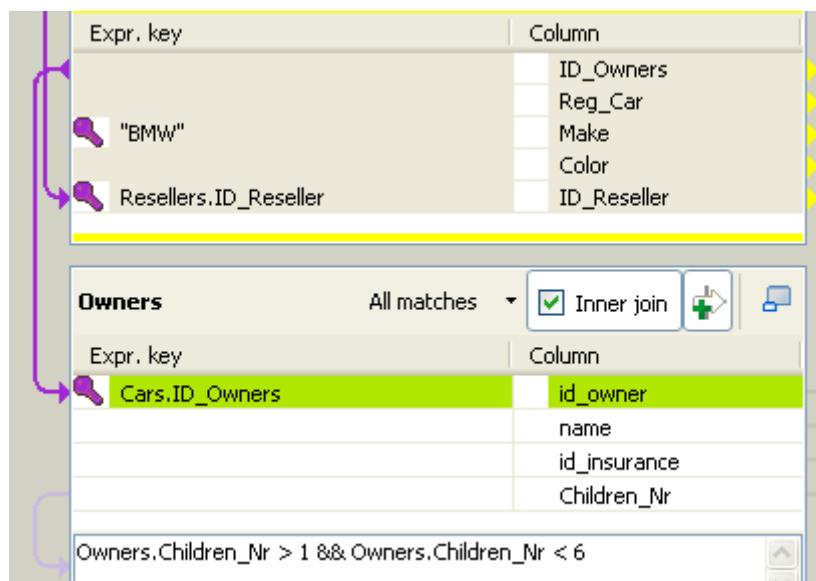
- Define the delimited file to be used, the Row and the Field Separator, the Header and Footer rows if any.
- Edit the Schema** if it hasn't been stored in the **Repository**. You will retrieve this schema in the **Main** table at the top of the **Input** area of the mapper.
- Carry out these previous steps for the other Input components: *Cars* and *Owners*. These two **Lookup** flows will fill in secondary (lookup) tables in the Input area of the Mapper.
- Then double-click on the **tMap** component to launch the Mapper and define the mapping and filters.



- First set the explicit joins between the **Main** flow and the **Lookup** flows.
- Simply drag & drop the **ID_Resellers** column towards the corresponding column and this way fill in the **Expression key** field of the **Lookup** table.

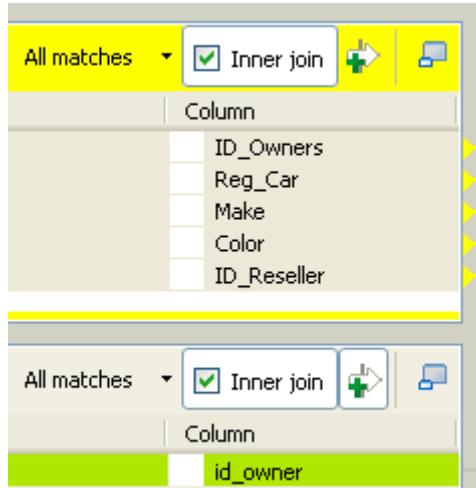


- The explicit join displays in color along with a hash key.
- Then in the **Expr. Key** of the *Make* column, type in (in Java) the filter. In this use case, simply type in “BMW” as the search is focused on the Owners of this particular Make.
- Implement a cascading join between the two lookup tables *Cars* and *Owners*, in order to retrieve owners information regarding the number of children they have.

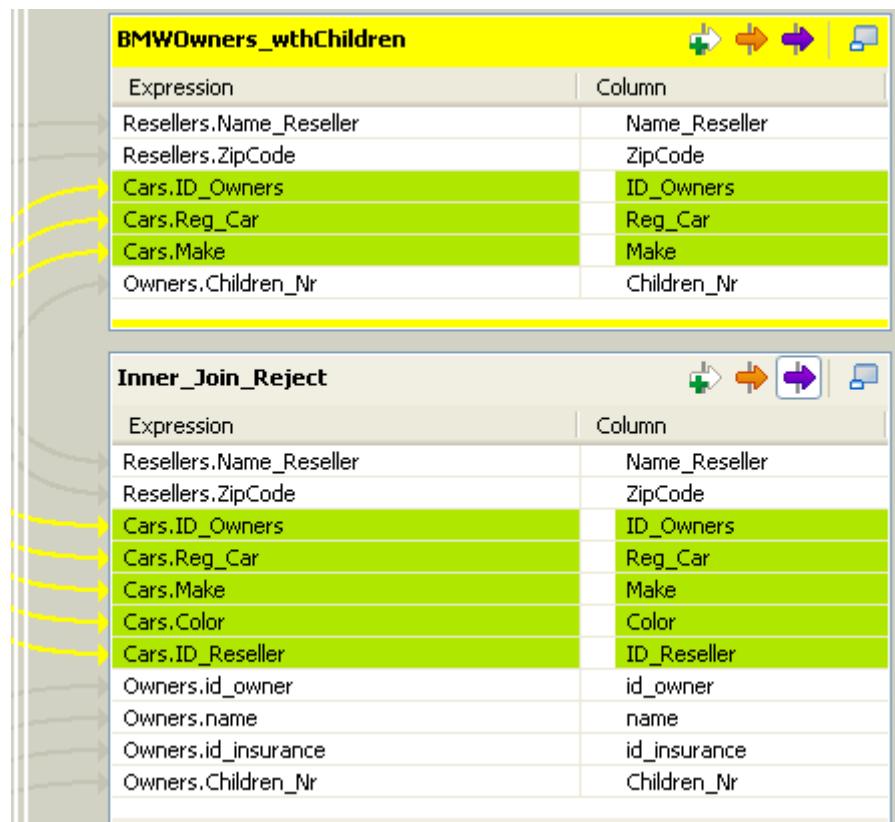


- Simply drag and drop the *ID_Owners* column from the *Cars* table towards the **Expr. Key** field of the *id_owner* column from the *Owners* table.
- Click the **Filter** button next to the **Inner Join** button to display the **Filter** expression area.

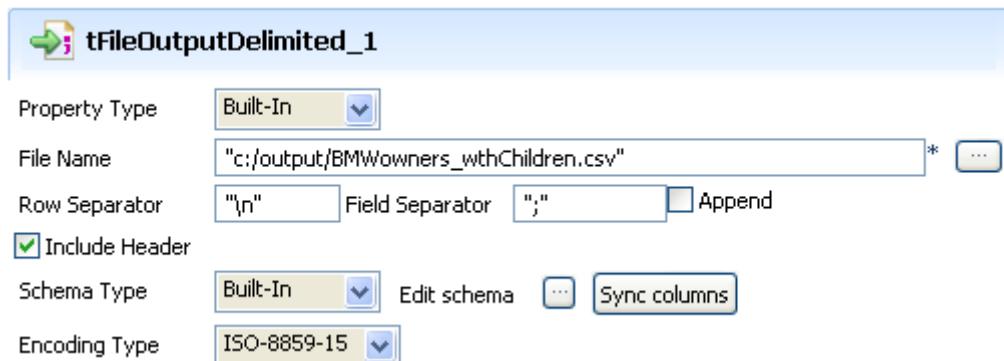
- Type in the **Filter** statement to narrow down the number of rows loaded in the **Lookup** flows. In this use case, the statement reads: `Owners.Children_Nr > 1 && Owners.Children_Nr < 6`
- Then, as you want to reject the null values into a separate table and exclude them from the standard output, check the **Inner Join** box for each of the filtered **Lookup** tables.



- In the Inner join, you can then choose to include only a **Unique match, the First or Last match or All Matches**. In this use case, the **All matches** option is selected. Thus if several matches are found in the Inner Join, i.e. rows matching the explicit join as well as the filter, all of them will be added to the output flow (either in Rejection or the regular output).
- Then on the **Output** area of the **Mapper**, add two tables, one for the full matches and one for the rejections.
- Click the plus button to add the tables and give a name to the outputs.

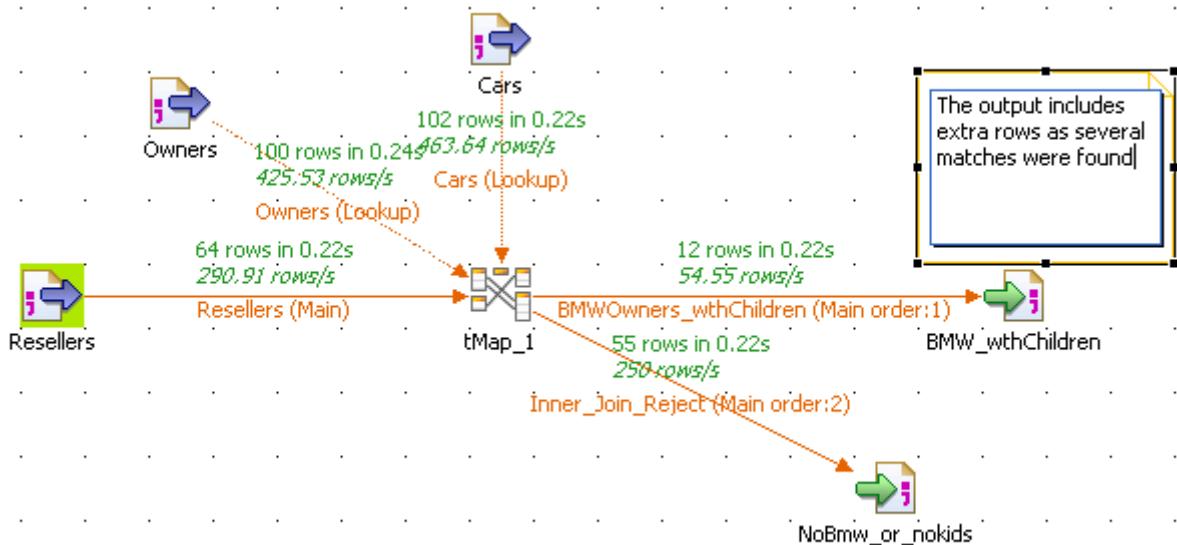


- Drag and drop data from the **Main** and **Lookup** tables in the Input area, towards the respective output tables, following the type of information you want to fetch.
- In the rejection table used to direct the non-matches from the external join or the filter, click the **Inner Join Reject** button (violet arrow) to activate it.
- On the **Designer** space, right-click on the **tMap** and pull the respective output link to the relevant components.
- Define the **Properties** of the **Output** components.



- Define the filepath, the expected Row and Field separator. And for this use case, check the Include Header box.

- The **Schema** should be automatically propagated from the **Mapper**.
- Save your job, then go to the **Run Job** tab and check the **Statistics** box to follow the processing thread.



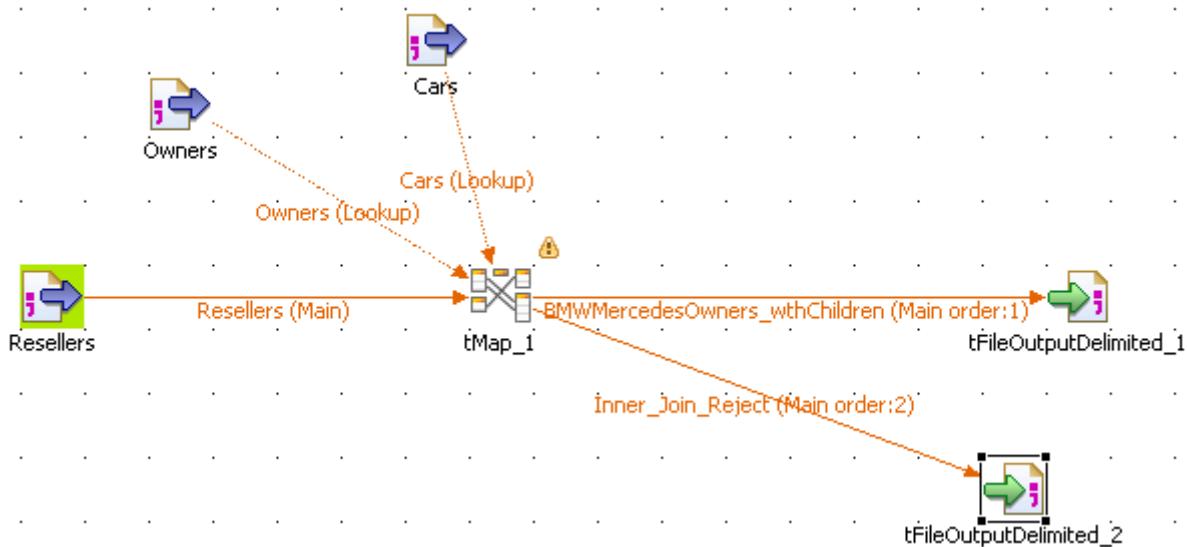
The statistics show that several matches were found and therefore the sum of the output rows (Main + rejected) exceeds the **Main** flow input rows.

Scenario 5: Advanced mapping with filters and a check of all rows

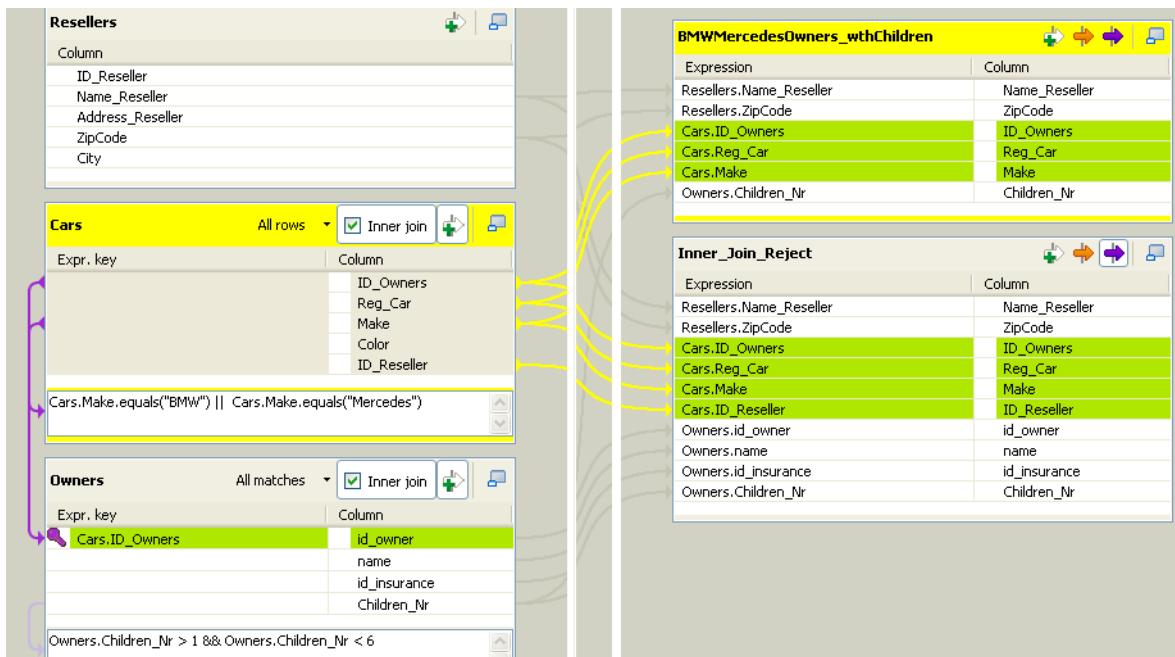
This scenario is a modified version of the preceding scenario. It describes a job that applies filters and then checks each row of loaded lookup rows.

Components

tMap

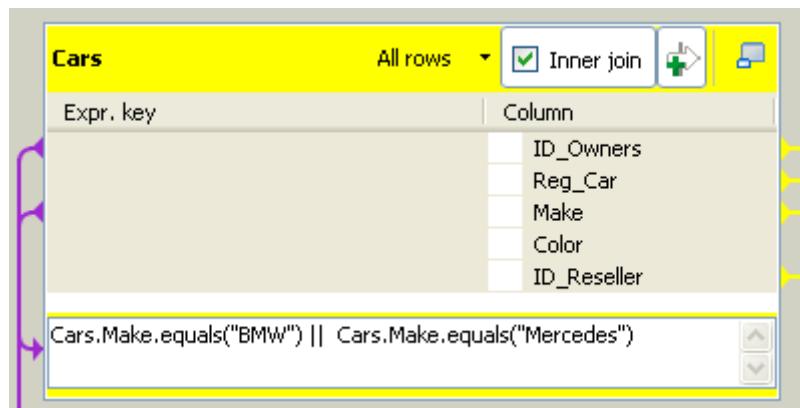


- Take the same job as in *Scenario 4: Advanced mapping with filters, explicit joins and Inner join rejection (Java)* on page 242.
- No changes are required in the Input delimited files.
- Launch the Mapper to change the mapping and the filters.

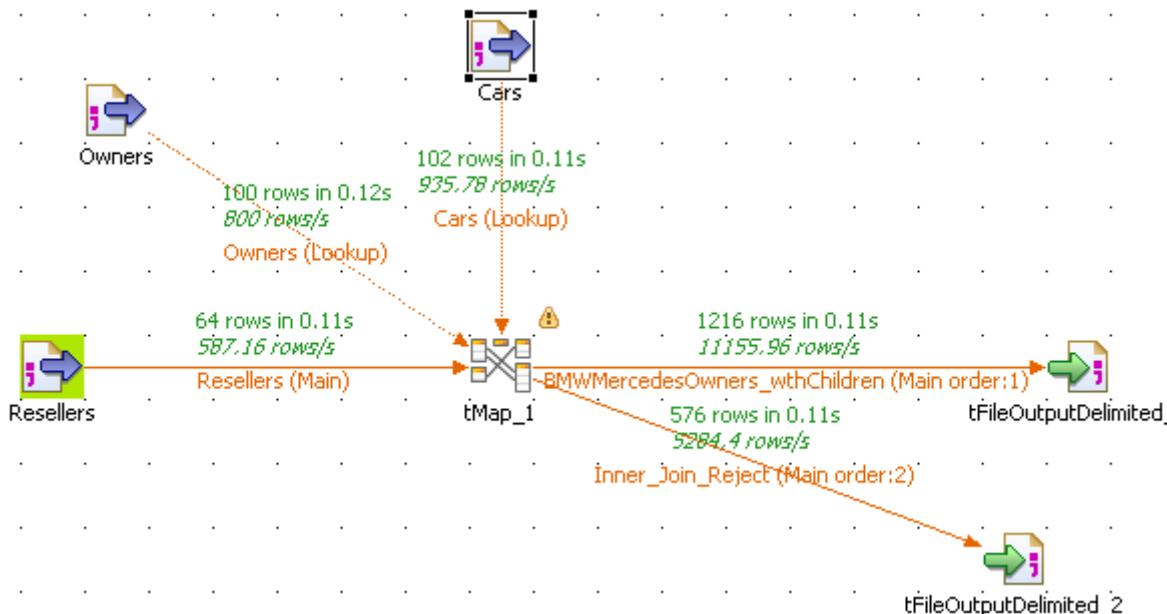


- Remove all explicit joins between the **Main** table and the *Cars Lookup* table.
- Notice that the **All Matches** setting changes automatically to **All Rows**. In fact, as no explicit join is declared (no hash keys), all lookup rows need to be loaded and checked against all main flow rows.

- Remove the Expr. key filter ("BMW") from the Cars table.



- And click the **Filters** button to display the **Filter** area. Then type in the new filter to narrow down the search to *BMW* or *Mercedes* car makes. The statement reads as follows:
Cars.Make.equals ("BMW") || Cars.Make.equals ("Mercedes")
- The filter on the **Owners Lookup** table doesn't change from the previous scenario.
- Define new file paths for the respective outputs.
- Save the job and enable the **Statistics** on the Run Job tab before executing the job.



The Statistics show that a cartesian product has been carried out between the Main flow rows with the filtered Lookup rows.

Components

tMap

	BMWMercedes_wthChildren.csv
34	Cars & Pickup Specialist;5952;76;2251 JG 82;Mercedes;5
35	Cars & Pickup Specialist;5952;79;2930 CP 77;Mercedes;5
36	Cars & Pickup Specialist;5952;96;8506 ZQ 08;BMW;4
37	Cars & Pickup Specialist;5952;97;7757 KQ 65;BMW;5
38	Cars & Pickup Specialist;5952;99;9162 MC 60;Mercedes;4
39	Cars & Pickup Specialist;5952;100;0146 DA 20;BMW;5
40	Quality Car Resale;7794;9;9939 CJ 88;Mercedes;3
41	Quality Car Resale;7794;15;4563 ZB 33;BMW;3
42	Quality Car Resale;7794;20;3408 EW 35;Mercedes;2
43	Quality Car Resale;7794;27;5792 QT 18;BMW;3
44	Quality Car Resale;7794;33;8253 DP 32;BMW;3
45	Quality Car Resale;7794;44;3748 NN 21;BMW;2
46	Quality Car Resale;7794;45;4065 EA 69;Mercedes;3

The content of the main output flow shows that the filtered rows have correctly been passed on.

	MWMercedes_wthChildren.csv	BMWMercedes_wthchildren_InnerReject.csv
1	Name_Reseller;ZipCode;ID_Owners;Reg_Car;Make;ID_Reseller;id_owner;name; id_insurance;Children_Nr	
2	Cars & Pickup Shop;5113;16;6709 YE 10;BMW;27;16;bobouh;QYW1412;1	
3	Cars & Pickup Shop;5113;37;5898 EB 09;BMW;54;37;hirtgall;MFR4898;1	
4	Cars & Pickup Shop;5113;65;0439 XF 39;BMW;32;65;mauvaes;FBG6516;6	
5	Cars & Pickup Shop;5113;68;8147 RS 83;Mercedes;33;68;bouhle;NGT4401;6	
6	Cars & Pickup Shop;5113;80;1359 DY 17;Mercedes;8;80;galigall;GAC9240;6	
7	Cars & Pickup Shop;5113;86;0094 SH 41;BMW;26;86;otbo;OFU7978;6	
8	Cars & Pickup Shop;5113;92;6544 LF 76;BMW;50;92;otmau;XNH2512;6	

Whereas, the Reject result clearly shows the rows that didn't match one of the filter.

tMSSqlInput



tMSSqlInput properties

The properties of the generic component, **tDBInput**, apply to the **tMSSqlInput** component, therefore the properties are not detailed again here.

For detailed information about generic **tDBInput** properties, see *tDBInput properties on page 136*.

Related scenarios

Related topics in **tDBInput** scenarios:

- *Scenario 1: Displaying selected data from DB table on page 137*
- *Scenario 2: Using StoreSQLQuery variable on page 138*

Related topic in **tContextLoad** Scenario: *Dynamic context use in MySQL DB insert on page 123*.

tMSSqlOutput



tMSSqlOutput properties

The properties of the generic component, **tDBOutput**, apply to the **tMSSqlOutput** component, therefore the properties are not detailed again here.

For detailed information about generic **tDBOutput** properties, see *DBOutput properties on page 140*.

Related scenarios

For **tMSSqlOutput** related topics, see:

- **tDBOutput Scenario: Displaying DB output on page 142**
- **tMySQLOutput Scenario: Adding new column and altering data on page 261.**

tMSqlRow

tMSqlRow properties

The properties of the generic component, **tDBSQLRow**, apply to the **tMSqlRow** component, therefore the properties are not detailed again here.

For detailed information about generic **tDBSQLRow** properties, see *tDBSQLRow properties on page 144*.

Related scenarios

For **tMSqlRow** related topics, see:

- **tDBSQLRow Scenario 1: Resetting a DB auto-increment on page 145**
- **tMySQLRow Scenario: Removing and regenerating a MySQL table index on page 275.**



tMysqlConnection

This component is closely related to **tMysqlCommit** and **tMysqlRollback**. It usually doesn't make much sense to use one of the latters without using a **tMysqlConnection** component to open a connection for the current transaction.

tMysqlConnection Properties

Component family	Databases	
Function	Opens a connection to the database for a current transaction.	
Purpose	Allows to commit a whole job data in one go to the output database as one transaction when validated.	
Properties	<i>Property type</i>	Either Built-in or Repository.
		Built-in: No existing property data is to be retrieved
		Repository: Select the Repository file where Properties are stored. The following fields are pre-filled in with fetched data.
	<i>Host</i>	Database server IP address
	<i>Port</i>	Listening port number of DB server.
	<i>Database</i>	Name of the database
	<i>Username and Password</i>	DB user authentication data.
	<i>Encoding type</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.
Usage	This component is to be used along with Mysql components, especially with tMysqlCommit and tMysqlRollback components.	
Limitation	n/a	

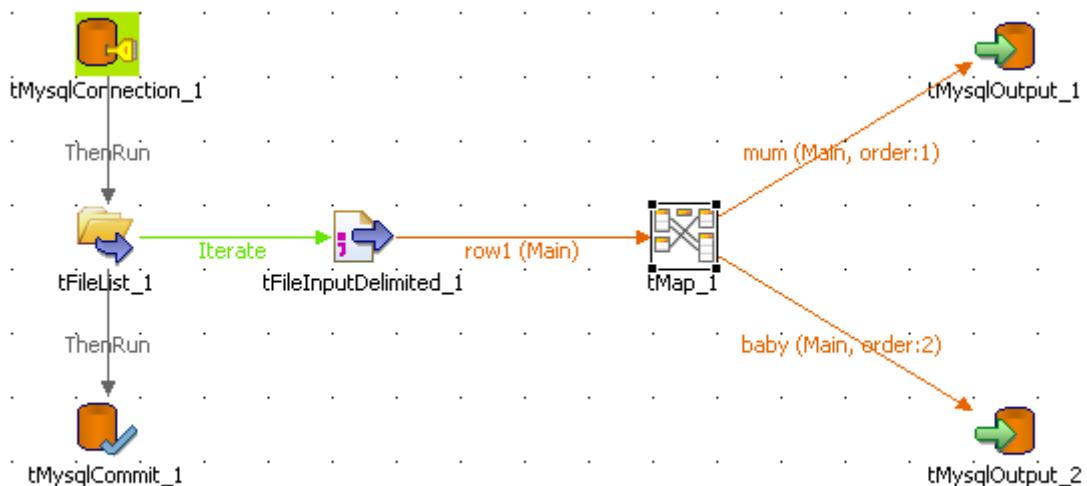
Scenario: Inserting data in mother/daughter tables

The following job is dedicated to advanced database users, who want to carry out multiple table insertions using a parent table id to feed a child table. As a prerequisite to this job, follow the steps described below to create the relevant tables using an engine such as *innodb*.

- In a command line editor, connect to your Mysql server.
- Once connected to the relevant database, type in the following command to create the parent table:
`create table f1090_mum(id int not null auto_increment, name varchar(10), primary key(id)) engine=innodb;`

- Then create the second table: `create table baby (id_baby int not null, years int) engine=innodb;`

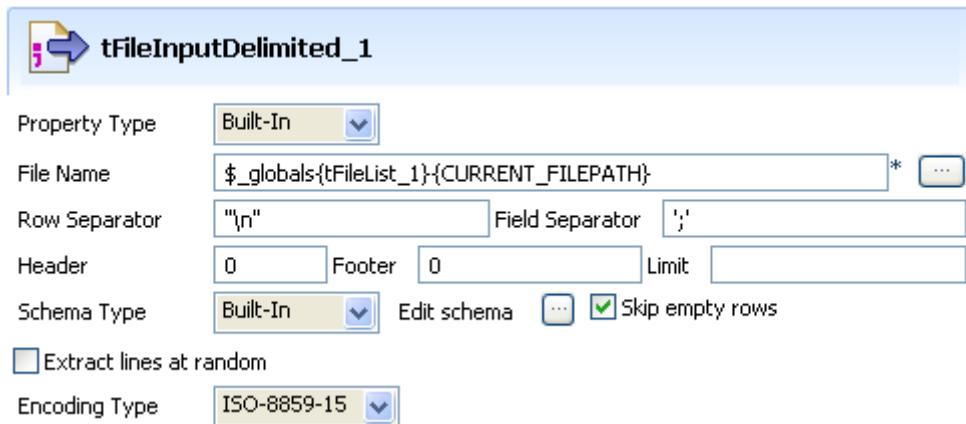
Back into **JasperETL**, the job requires seven components including **tMysqlConnection** and **tMysqlCommit**.



- Drag and drop the following components from the Palette: **tFileList**, **tFileInputDelimited**, **tMap**, **tMysqlOutput** (x2).
- Connect the **tFileList** component to the input file component using an **Iterate** link as the name of the file to be processed will be dynamically filled in from the **tFileList** directory using a global variable.
- Connect the **tFileInputDelimited** component to the **tMap** and dispatch the flow between the two output Mysql DB components. Use a **Row** link for each for these connections representing the main data flow.
- Set the **tFileList** component properties, such as the directory. name where files will be fetched from.
- Add a **tMysqlConnection** component and connect it to the starter component of this job, in this example, the **tFileList** component using a **ThenRun** link to define the execution order.
- In the **tMysqlConnection** Properties panel, set the connection details manually or fetch them from the Repository if you centrally stored them as a Metadata DB connection entry. For more information about Metadata, see *Defining Metadata items on page 54*.
- On the **tFileInputDelimited** component's **Properties** panel, press Ctrl+Space bar to access the variable list. Set the **File Name** field to the global variable:
`$_globals{tFileList_1}{CURRENT_FILEPATH}`

Components

tMysqlConnection

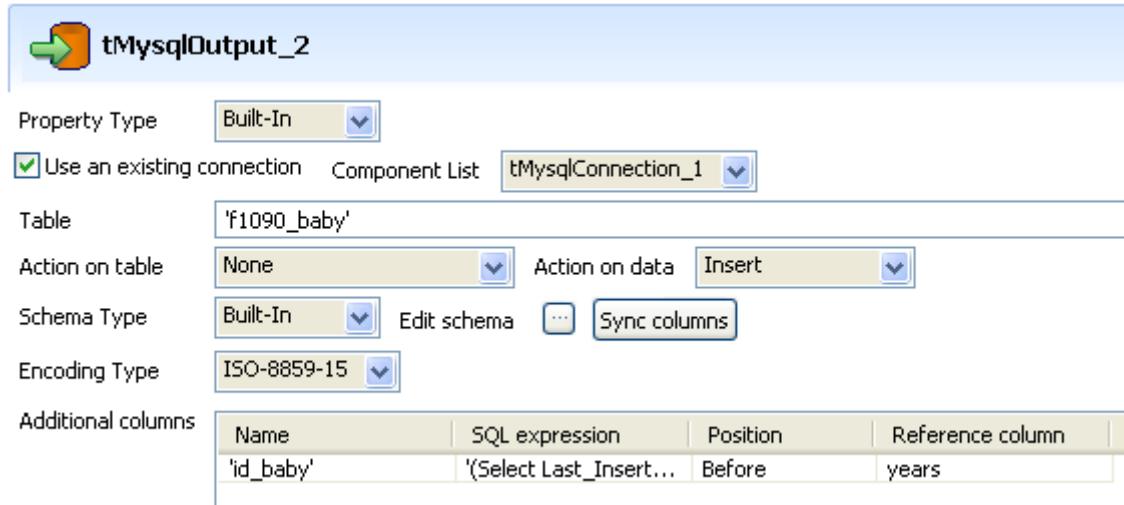


- Set the rest of the fields as usual, defining the row and field separators according to your file structure.
- Then set the schema manually through the **Edit schema** feature or select the schema from the Repository. In Java version, make sure the data type is correctly set, in accordance with the nature of the data processed.
- Change the encoding if different from the default one.
- In the **tMap** Output area, add two output tables, one called mum for the parent table, the second called baby, for the child table.
- Drag the *Name* column from the **Input** area, and drop it to the mum table.
- Drag the *Years* column from the **Input** area and drop it to the baby table.



- Make sure the mum table is on the top of the baby table as the order is determining for the flow sequence hence the DB insert to perform correctly.
- Then connect the output row link to distribute correctly the flow to the relevant DB output component.

- In each of the **tMysqlOutput** components' **Properties** panel, check the **Use an existing connection** box to retrieve the **tMysqlConnection** details.
- Notice (in Perl version) that the **Commit every** field doesn't show anymore as you are supposed to use the **tMysqlCommit** instead to manage the global transaction commit. In Java version, ignore the field as this command will get overridden by the **tMysqlCommit**.



- Set the **Table** name making sure it corresponds to the correct table, in this example either *f1090_mum* or *f1090_baby*.
- There is no action on the table as they are already created.
- Select **Insert** as **Action on data** for both output components.
- Click on Sync columns to retrieve the schema set in the tMap.
- Change the encoding type if need be.
- In the **Additional columns** area of the DB output component corresponding to the child table (*f1090_baby*), set the *id_baby* column so that it reuses the *id* from the parent table.
- In the **SQL expression** field type in: '`(Select Last_Insert_id())`'
- The position is *Before* and the **Reference column** is *years*.
- Add the **tMysqlCommit** component to the job workspace and connect it from the **tFileList** component using a **ThenRun** connection in order for the job to terminate with the transaction commit.
- On the **tMysqlCommit** component Properties panel, select in the list the connection to be used.

Save your job and press **F6** to run it.

Components

tMysqlConnection

```
mysql> select * from f1090_mum  
-> ;  
+-----+  
| id | names |  
+-----+  
| 6  | john  |  
| 7  | bruce |  
| 8  | beth  |  
| 9  | andrew|  
| 10 | donald|  
| 11 | betty |  
| 12 | john  |  
| 13 | bruce |  
| 14 | beth  |  
| 15 | andrew|  
| 16 | donald|  
| 17 | betty |  
+-----+  
12 rows in set (0.00 sec)
```

```
mysql> select * from f1090_baby  
-> ;  
+-----+-----+  
| id_baby | years |  
+-----+-----+  
| 6        | 10    |  
| 7        | 23    |  
| 8        | 34    |  
| 9        | 10    |  
| 10       | 23    |  
| 11       | 34    |  
| 12       | 10    |  
| 13       | 23    |  
| 14       | 34    |  
| 15       | 10    |  
| 16       | 23    |  
| 17       | 34    |  
+-----+-----+  
12 rows in set (0.00 sec)
```

The parent table *id* has been reused to feed the *id_baby* column.

tMysqlCommit



This component is closely related to **tMysqlCommit** and **tMysqlRollback**. It usually doesn't make much sense to use these components independently in a transaction..

tMysqlCommit Properties

Component family	Databases	
Function	Validates the data processed through the job into the connected DB	
Purpose	Using a unique connection, commits in one go a global transaction instead of every row or every batch. Provides a gain in performance	
Properties	<i>Component list</i>	Select the tMysqlConnection component in the list if more than one connection are planned for the current job.
Usage	This component is to be used along with Mysql components, especially with tMysqlConnection and tMysqlRollback components.	
Limitation	n/a	

Related scenario

This component is closely related to **tMysqlCommit** and **tMysqlRollback**. It usually doesn't make much sense to use one of the latters without using a **tMysqlConnection** component to open a connection for the current transaction.

For **tMysqlCommit** related scenario, see *tMysqlConnection on page 254*.

tMysqlInput



tMysqlInput properties

The properties of the generic component, **tDBInput**, apply to the **tMysqlInput** component, therefore the properties are not detailed again here.

For detailed information about generic **tDBInput** properties, see *tDBInput properties on page 136*.

Related scenarios

Related topic in **tDBInput** scenarios:

- *Scenario 1: Displaying selected data from DB table on page 137*
- *Scenario 2: Using StoreSQLQuery variable on page 138*

Related topic in **tContextLoad** Scenario: *Dynamic context use in MySQL DB insert on page 123*.

tMysqlOutput



tMysqlOutput properties

The properties of the generic component, **tDBOutput**, apply to the **tMysqlOutput** component, therefore the properties are not detailed again here.

For detailed information about generic **tDBOutput** properties, see *DBOutput properties on page 140*.

Scenario: Adding new column and altering data

This scenario is a three-component job aiming at creating random data using a **tRowGenerator**, duplicating a column to be altered using the **tMap** component, eventually altering the data to be inserted based on a SQL expression, as well as inserting a new column in the DB, using the **tDBOutput** component.



- Drag and drop the **tRowGenerator**, **tMap** and **tMySQLOutput** components onto the designer.
- Link the **tRowGenerator** to the **tMap**.
- Set the **tRowGenerator** component properties. Create a two-column schema: **Name** and **Random_date**
 - The **Name** column does pick up randomly names from a list specified. In this use case, the list includes *FabriceB*, *PierrickL*, *GabrielM* and *ElisaS*.
 - Then, double-click on the **tMap** component to duplicate the **random_date** column and adapt the schema in order to alter the data in the output component.
 - In the Mapper, create an output link to the **tMySQLOutput** component. Add one more column (based on the input schema) and name it *random_date_I* to distinguish it from the other *random_date* column.
 - Drag and drop the *random_date* content from the input area to the output area.
 - Then double-click on the **tMySQLOutput** component to set its parameters.
 - First fill in the DB connection details, either through the Repository or manually in case of Built-in information.
 - Select the table to be altered, in this example: *Feature516*.
 - No **Action on table** is to be carried out, the **Action on data** is *Insert*.

- In the **Additional Columns** area, set the alteration to be performed on columns and the specific insertion of a new *moment* column onto the database.
- The *One_month_later* column is a replacement column for the *random_date_1* column. Also, the data it-self gets altered using an SQL expression, which adds one month to the randomly picked-up date of the *random_date_1* column. ex: 2007-08-12 becomes 2007-09-12
- Therefore, in **Name** field goes the new column label (*One_Month_Later*) and in **SQL expression** field, type in the relevant addition script to be performed: 'adddate(?, interval 1 month)' then as **Position**, select **Replace**, and the **Reference column** is *Random_date_1*.
- Note that for this job we duplicated the *random_date_1* column in the DB table before replacing one instance of it with the *One_Month_Later* column. The aim of this workaround was to be able to view upfront the modification performed.
- The second entry is the new column, *moment*, to be inserted into the database table. As **SQL expression**, type in the moment function: *now()* and in the **Position** field, select **Before**, the **Reference column** is name in this example.
- Once the Output setting is complete, press **F6** to run the job.

Two new columns were added or altered onto the DB table: *One_Month_Later* and *Moment*.

Related topic: *DBOutput properties on page 140*



tMysqlOutputBulk

tMysqlOutputBulk and tMysqlBulkExec components are used together to first output the file that will be then used as parameter to execute the SQL query stated. These two steps compose the tMysqlOutputBulkExec component, detailed in a separate section. The interest in having two separate elements lies in the fact that it allows transformations to be carried out before the data loading.

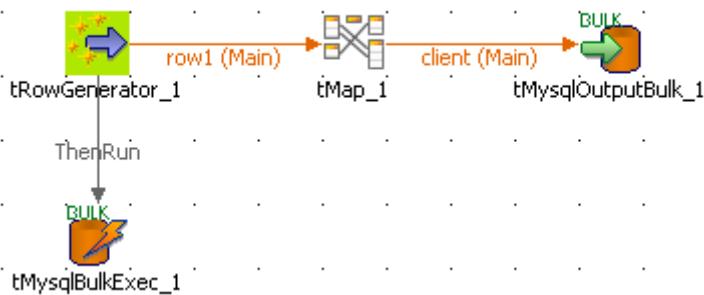
tMysqlOutputBulk properties

Component family	Databases	 
Function	Writes a file with columns based on the defined delimiter and the MySQL standards	
Purpose	Prepares the file to be used as parameter in the INSERT query to feed the MySQL database.	
Properties	<i>Property type</i>	Either Built-in or Repository.
		Built-in: No existing property data is to be retrieved
		Repository: Select the Repository file where Properties are stored. The following fields are pre-filled in using fetched data.
	<i>File Name</i>	Name of the file to be processed. Related topic: <i>Defining the context variables on page 96</i>
	<i>Field separator</i>	Character, string or regular expression to separate fields.
	<i>Row separator</i>	String (ex: “\n”on Unix) to distinguish rows.
	<i>Append</i>	Check this option box to add the new rows at the end of the file
	<i>Include header</i>	Check this box to include the column header to the file.
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository.
		Built-in: The schema will be created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>
		Repository: The schema already exists and is stored in the Repository, hence can be reused in various projects and job designs. Related topic: <i>Setting a repository schema on page 52</i>
	<i>Sync columns</i>	Click to synchronize the output file schema with the input file schema. The Sync function only displays once the Row connection is linked with the Output component.
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.

Usage	This component is to be used along with tMySQIBulkExec component. Used together they offer gains in performance while feeding a MySQL database.
--------------	--

Scenario: Inserting transformed data in MySQL database

This scenario describes a four-component job which aims at fueling a database with data contained in a file, including transformed data. Two steps are required in this job, first step is to create the file, that will then be used in the second step. The first step includes a transformation phase of the data included in the file.



- Drag and drop a **tRowGenerator**, a **tMap**, a **tMysqlOutputBulk** as well as a **tMysqlBulkExec** component.
- Connect the main flow using **row main** links.
- And connect the start component (**tRowgenerator** in this example) to the **tMysqlBulkExec** using a **trigger** connection, of type **ThenRun**.
- A **tRowGenerator** is used to generate random data. Double-click on the **tRowGenerator** component to launch the editor.
- Define the schema of the rows to be generated and the nature of data to generate. In this example, the *clients* file to be produced will contain the following columns: *ID*, *First Name*, *Last Name*, *Address*, *City* which all are defined as string data but the *ID* that is of integer type.

Components

tMysqlOutputBulk

The screenshot shows the Talend Open Studio interface with the title bar "Talend Open Studio - tRowGenerator - tRowGenerator2_1". Below the title bar is a toolbar with various icons for schema management. The main area is divided into three sections: "Schema", "Functions", and "Preview".
Schema Section: A table showing column definitions. Columns include ID (int), FirstName (String), LastName (String), Address (String), and City (String). Functions listed are sequence, firstName, lastName, street, and city.
Functions Section: Shows the mapping of columns to functions: ID to sequence, FirstName to firstName, LastName to lastName, Address to street, and City to city.
Preview Section: A table showing generated rows. The first row is highlighted with a yellow background. The columns are ID, FirstName, LastName, Address, and City. The data for the first row is 1, Abraham, Garfield, Apalachee Parkway, and Salt Lake City respectively.
At the bottom of the window, there is a toolbar with buttons for adding, deleting, and modifying columns, along with a "Columns" button and a "Number of Rows for RowGenerator" input field set to 100.

- Some schema information don't necessarily need to be displayed. To hide them away, click on **Columns** list button next to the toolbar, and uncheck the relevant entries, such as **Precision** or **Parameters**.
- Use the plus button to add as many columns to your schema definition.
- Click the Refresh button to preview the first generated row of your output.
- Then select the **tMap** component to set the transformation.
- Drag and drop all columns from the input table to the output table.

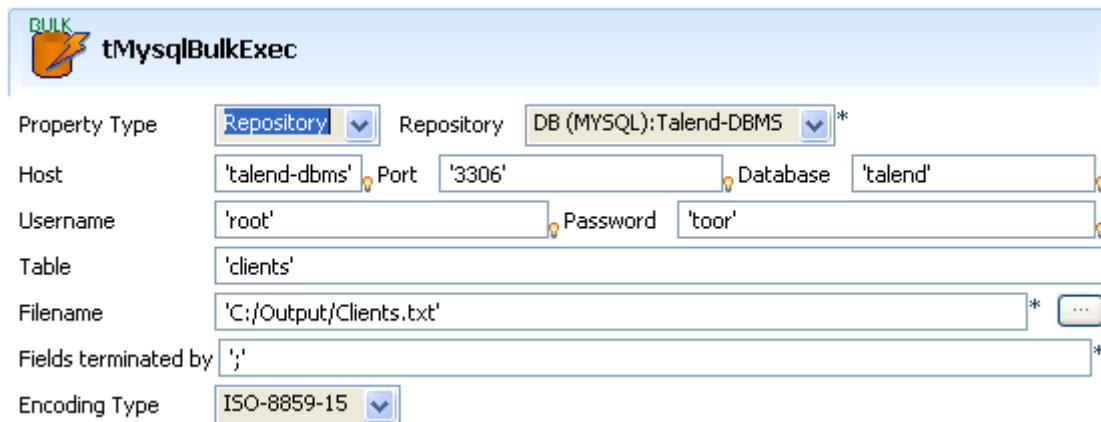
The screenshot shows the tMap component configuration. The top section is titled "client" with a toolbar. Below it is a table with two columns: "Expression" and "Column".
The table rows are:

- \$row1[ID] | ID
- \$row1[FirstName] | FirstName
- uc|\$row1[LastName]** | LastName
- \$row1[Address] | Address
- \$row1[City] | City

A yellow arrow points to the "uc|\$row1[LastName]" row, highlighting the transformation step.

- Apply the transformation on the *LastName* column by adding uc in front of it.
- Click **OK** to validate the transformation.
- Then double-click on the **tMysqlOutputBulk** component.
- Define the name of the file to be produced in **File Name** field. If the delimited file information is stored in the **Repository**, select it in **Property type** field, to retrieve relevant data. In this use case the file name is *clients.txt*.

- The schema is propagated from the **tMap** component, if you accepted it when prompted.
- In this example, don't include the header information as the table should already contain it.
- The encoding is the default one for this use case.
- Click OK to validate the output.
- Then double-click on the **tMysqlBulkExec** to set the INSERT query to be executed.
- Define the database connection details. We recommend you to store this type of information in the **Repository**, so that you can retrieve them at any time for any job.



- Set the table to be filled in with the collected data, in the **Table** field.
- Fill in the column delimiters in the **Field terminated by** area.
- Make sure the encoding corresponds to the data encoding.
- Then press **F6** to run the job.

Resultset 1					
	ID	First Name	Last Name	Address	City
▶	1	Martin	REAGAN	Hutchinson Rd	Dover
	2	Herbert	REAGAN	Bailard Avenue	Frankfort
	3	Franklin	WASHINGTON	Bayshore Freeway	Denver
	4	Franklin	CARTER	Burnett Road	Frankfort
	5	Woodrow	MCKINLEY	San Ysidro Blvd	Des Moines
	6	Bill	QUINCY	Fairview Avenue	Topeka
	7	Bill	BUREN	Calle Real	Jefferson City
	8	Andrew	ARTHUR	Santa Ana Freeway	Indianapolis
	9	Woodrow	FORD	N Harrison St	Augusta
	10	Calvin	COOLIDGE	Harbor Dr	Frankfort

The *clients* database table is filled with data from the file including upper-case *last name* as transformed in the job.

Components

tMysqlOutputBulk

For simple Insert operations that don't include any transformation, the use of **tMysqlOutputBulkExec** allows to spare a step in the process hence to gain some performance.

Related topic: *tMysqlOutputBulkExec properties on page 272*



tMysqlBulkExec

tMysqlOutputBulk and **tMysqlBulkExec** components are used together to first output the file that will be then used as parameter to execute the SQL query stated. These two steps compose the **tMysqlOutput-BulkExec** component, detailed in a separate section. The interest in having two separate elements lies in the

Components

tMysqlBulkExec

fact that it allows transformations to be carried out before the data loading in the database.

tMysqlBulkExec properties

Component family	Databases	 
Function	Executes the Insert action on the data provided.	
Purpose	As a dedicated component, tMysqlBulkExec offers gains in performance while carrying out the Insert operations to a MySQL database	
Properties	<i>Property type</i>	Either Built-in or Repository.
		Built-in: No existing property data is to be retrieved
		Repository: Select the Repository file where Properties are stored. The following fields are pre-filled in using fetched data.
	<i>Host</i>	Database server IP address
	<i>Port</i>	Listening port number of DB server.
	<i>Database</i>	Name of the database
	<i>Username and Password</i>	DB user authentication data.
	<i>Table</i>	Name of the table to be written. Note that only one table can be written at a time and that the table must exist for the insert operation to succeed.
	<i>File Name</i>	Name of the file to be processed. Related topic: <i>Defining the context variables on page 96</i>
	<i>Fields terminated by</i>	Character, string or regular expression to separate fields.
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.
	<i>Commit every</i>	Number of rows to be completed before committing batches of rows together into the DB. This option ensures transaction quality (but not rollback) and above all better performance on executions.
Usage	This component is to be used along with tMySQLOutputBulk component. Used together, they can offer gains in performance while feeding a MySQL database.	
Limitation	n/a	

Related scenarios

For uses cases in relation with **tMysqlBulkExec**, see the following scenarios:

- **tMysqlOutputBulk Scenario: Inserting transformed data in MySQL database on page 265**
- **tMysqlOutputBulkExec Scenario: Inserting data in MySQL database on page 273**

Components

tMysqlOutputBulkExec



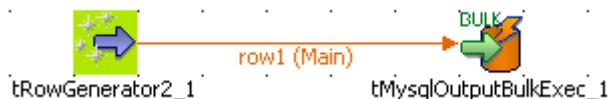
tMysqlOutputBulkExec

tMysqlOutputBulkExec properties

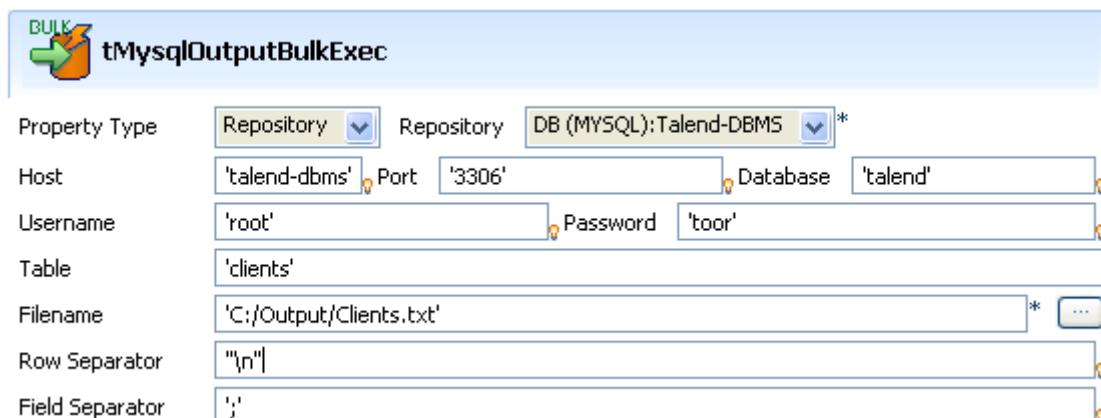
Component family	Databases	
Function	Executes the Insert action on the data provided.	
Purpose	As a dedicated component, it allows gains in performance during Insert operations to a MySQL database.	
Properties	<i>Property type</i>	Either Built-in or Repository.
		Built-in: No existing property data is to be retrieved
		Repository: Select the Repository file where Properties are stored. The following fields are pre-filled in using fetched data.
	<i>Host</i>	Database server IP address
	<i>Port</i>	Listening port number of DB server.
	<i>Database</i>	Name of the database
	<i>Username and Password</i>	DB user authentication data.
	<i>Table</i>	Name of the table to be written. Note that only one table can be written at a time and that the table must exist for the insert operation to succeed.
	<i>File Name</i>	Name of the file to be processed. Related topic: <i>Defining the context variables on page 96</i>
	<i>Field separator</i>	Character, string or regular expression to separate fields.
	<i>Row separator</i>	String (ex: "\n" on Unix) to distinguish rows.
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.
	<i>Commit every</i>	Number of rows to be completed before committing batches of rows together into the DB. This option ensures transaction quality (but not rollback) and above all better performance on executions.
Usage	This component is mainly used when no particular transformation is required on the data to be loaded onto the database.	
Limitation	n/a	

Scenario: Inserting data in MySQL database

This scenario describes a two-component job which carries out the same operation as the one described for *tMysqlOutputBulk properties on page 264* and *tMysqlBulkExec properties on page 270*, although no transformation of data is performed.



- Click and drop a **tRowGenerator** and a **tMysqlOutputBulkExec** component.
- The **tRowGenerator** is to be set the same way as in the *Scenario: Inserting transformed data in MySQL database on page 265*. The schema is made of four columns including: *ID*, *First Name*, *Last Name*, *Address* and *City*.
- Then set the DB connection if needed, the best practices being to store the connection details in the Metadata repository.
- Then fill in the table to be filled in with the generated data in the **Table** field.
- And the name of the file to be loaded in **File Name** field.



Then press F6 to execute the job.

The result should be pretty much the same as in *Scenario: Inserting transformed data in MySQL database on page 265*, but the data might differ as these are regenerated randomly everytime the job is run.

Components

tMysqlRollback



tMysqlRollback

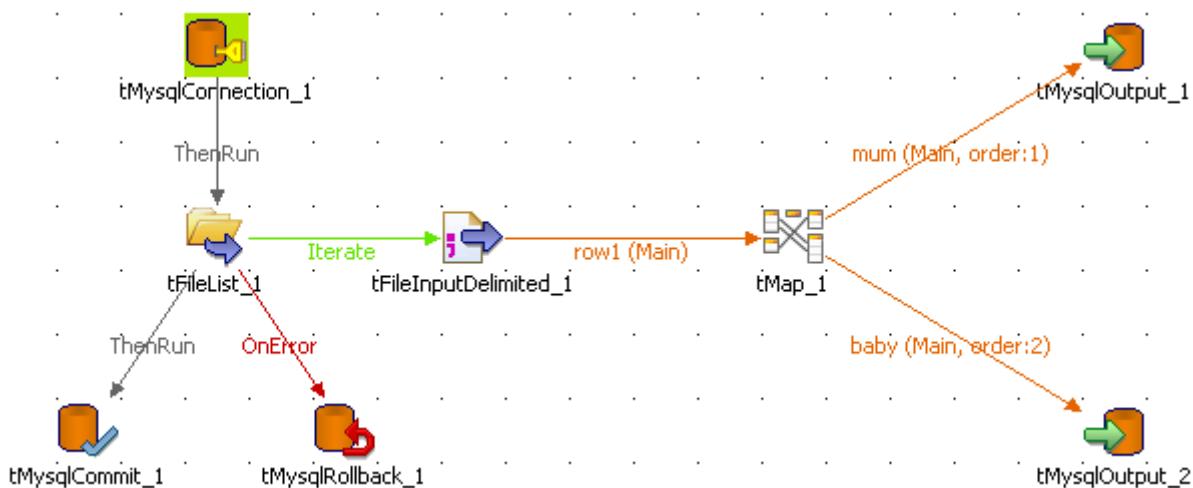
This component is closely related to **tMysqlCommit** and **tMysqlConnection**. It usually doesn't make much sense to use these components independently in a transaction..

tMysqlRollback properties

Component family	Databases	
Function	Cancel the transaction commit in the connected DB.	
Purpose	Avois to commit part of a transaction unvolontarily.	
Properties	Component list	Select the tMysqlConnection component in the list if more than one connection are planned for the current job.
Usage	This component is to be used along with Mysql components, especially with tMysqlConnection and tMysqlCommit components.	
Limitation	n/a	

Scenario: Rollback from inserting data in mother/daughter tables

Based on the tMysqlConnection Scenario: *Inserting data in mother/daughter tables on page 254*, insert a rollback function in order to prevent unwanted commit.



- Drag and drop a **tMysqlRollback** to the workspace and connect it to the Start component.
- Set the Rollback unique field on the relevant DB connection.

This complementary element to the job ensures that the transaction won't be partly committed.

tMysqlRow



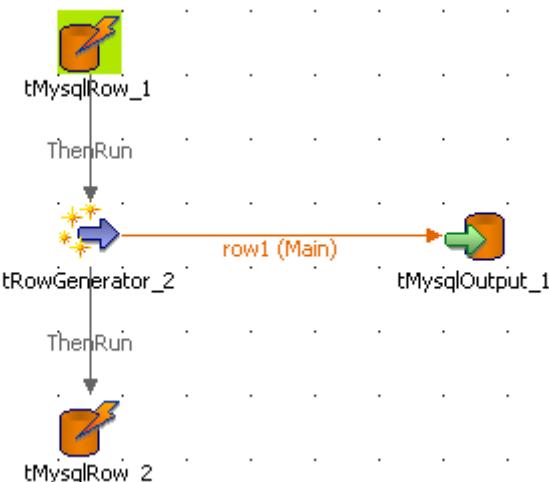
tMysqlRow properties

The properties of the generic component, **tDBSQLRow**, apply to the **tMysqlRow** component, therefore the properties are not detailed again here.

For detailed information about generic **tDBSQLRow** properties, see *tDBSQLRow properties on page 144*.

Scenario: Removing and regenerating a MySQL table index

This scenario describes a four-component job which wants to remove a table index, make a select insert action onto a table then regenerate the index.



- Select and drop the following components onto the graphical workspace: **tMysqlRow** (x2), **tRowGenerator**, **tMysqlOutput**.
- Connect **tMysqlInput** to the **tRowGenerator**.
- Then using a **ThenRun** connection, link the first **tMysqlRow** to the **tMysqlInput**,
- Then connect **tRowGenerator** to the second **tMysqlRow** using a **ThenRun** link again.
- Select the **tMysqlRow** to fill in the **DB Properties**.
- In **Property type** as well in **Schema type**, select the relevant DB entry in the list.
- The DB connection details and the table schema are accordingly filled in.
- Propagate the properties and schema details onto the other components of the job.
- The query being stored in the **Metadata** area of the Repository, you can also select **Repository** in the **Query type** field and the relevant query entry.

- If you didn't store your query in the **Repository**, type in the following SQL statement to alter the database entries: drop index <index_name> on <table_name>
- Then select the second **tMysqlRow** component, check the DB properties and schema.
- Then type in the SQL statement to recreate an index on the table using the following statement: create index <index_name> on <table_name> (<column_name>);
- The **tRowGenerator** component is used to generate automatically the columns to be added to the DB output table defined.
- Select the **tMysqlOutput** component and fill in the DB connection properties either from the Repository or manually the DB connection details are specific for this use only. The table to be fed is named: *comprehensive*.
- The schema should be automatically inherited from the data flow coming from the **tLogRow**. Edit the schema to check its structure and check that it corresponds to the schema expected on the DB table specified.
- The **Action on table** is *None* and the **Action on data** is *Insert*.
- No additional Columns is required for this job.
- Press **F6** to run the job.

If you manage to watch the action on DB data, you can notice that the index is dropped at the start of the job and recreated at the end of the insert action.

Related topics: *tDBSQLRow properties on page 144*.

tMsgBox



tMsgBox properties

Component family	Misc	
Function	Opens a dialog box with an OK button requiring action from the user.	
Purpose	tMsgBox is a graphical break in the job execution progress.	
Properties	<i>Title</i>	Text entered shows on the title bar of the dialog box created.
	<i>Buttons</i>	Listbox of buttons you want to include in the dialog box. The button combinations are restricted and cannot be changed.
	<i>Icon</i>	Icon shows on the title bar of the dialog box.
	<i>Message</i>	Free text to display as message on the dialog box. Text can be dynamic (for example: retrieve and show a file name).
Usage	This component can be used as intermediate step in a data flow or as a start or end object in the job flowchart. It can be connected to the next/previous component using either a Row or Iterate link.	
Limitation	For Perl users: Make sure the relevant package is installed.	

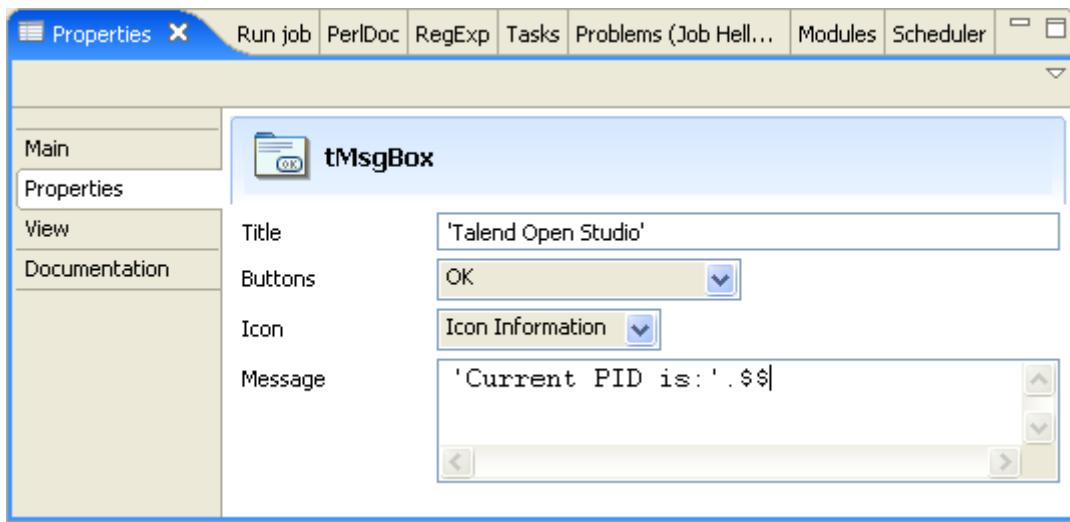
Scenario: ‘Hello world!’ type test

The following scenario creates a single-component job, where tMsgBox is used to display the pid (process id) in place of the traditional “Hello World!” message.

- Click and drop a **tMsgBox** component into the workspace.
- Define the dialog box display properties:

Components

tMsgBox



- ‘My Title’ is the message box title, it can be any variable.
- In the Message field comes the message text in quotes concatenated with the Perl scalar variable (\$\$) containing the “pid” for this example.
- Switch to the **Run job** tab to execute the job defined.

The Message box displays the message and requires the user to click OK to go to the next component or end the job.



After the user clicked on OK button, the **Run Job** log is updated accordingly.

Related topic: *Running a job on page 101*



tNormalize

tNormalize Properties

Component family	Processing	
Function	Normalizes the input flow following SQL standard.	
Purpose	tNormalize helps improve data quality and thus eases the data update.	
Properties	<i>Schema type</i> and <i>Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository. In this component, the schema is read-only.
	Built-in: The schema will be created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>	
	<i>Column to normalize</i>	Select the column from the input flow which the normalization is based on
	<i>Separator</i>	Enter the separator which will delimits data in the input flow.
Usage	This component can be used as intermediate step in a data flow.	
Limitation	n/a	

Scenario: Normalizing data

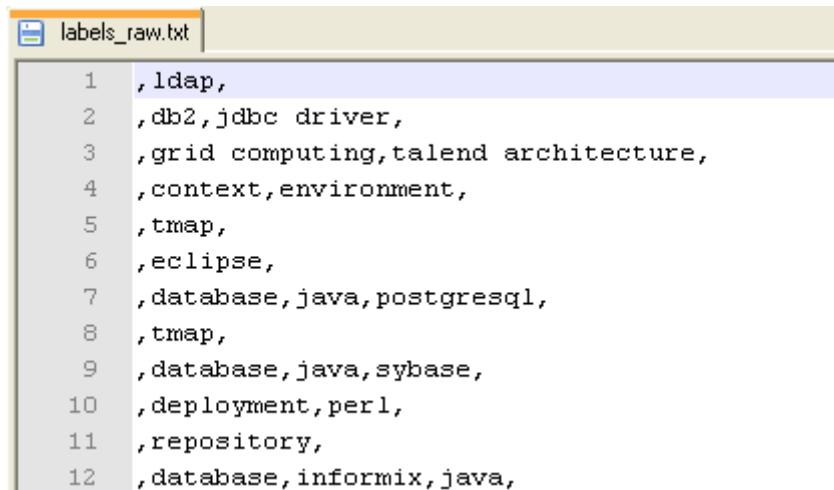
This simple scenario illustrates a job that normalizes a list of tags for Web forum topics and outputs them into a table in the standard output console (Run Job tab).



- Click and drop the following components onto the designing workspace: **tFileInputDelimited**, **tNormalize**, **tLogRow**.
- In the **tFileInputDelimited** properties, set the input file to be normalized.

Components

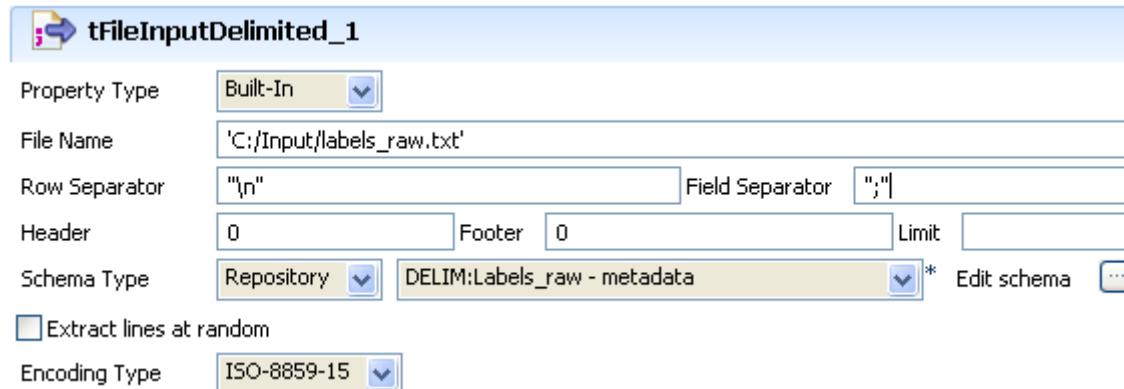
tNormalize



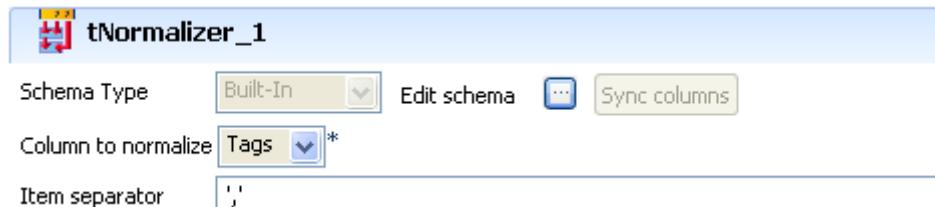
The screenshot shows a text editor window titled "labels_raw.txt". The content of the file is a list of tags, each preceded by a line number from 1 to 12. The tags listed are: ldap, db2, jdbc driver, grid computing, talend architecture, context, environment, tmap, eclipse, database, java, postgresql, tmap, database, java, sybase, deployment, perl, repository, database, informix, and java.

```
1 ,ldap,
2 ,db2,jdbc driver,
3 ,grid computing,talend architecture,
4 ,context,environment,
5 ,tmap,
6 ,eclipse,
7 ,database,java,postgresql,
8 ,tmap,
9 ,database,java,sybase,
10 ,deployment,perl,
11 ,repository,
12 ,database,informix,java,
```

- The file schema is stored in the repository for ease of use. It is made of one column, called *Tags*, containing rows with one or more keywords.
- Set the **Row Separator** and the **Field Separator**.



- On the **tNormalize Properties** panel, define the column the normalization operation is based on.
- In this use case, the column to normalize is *Tags*.



- The **Item separator** is the comma, surrounded here by single quotes as the job is done in Perl.
- In the **tLogRow** component, check the **Print values in the cells of table** box.

- Save the Job and run it.

Starting job tNormalize at 17:54 03/07/2007.	
	tLogRow_1
+	Tags
	ldap
	db2
	jdbc driver
	grid computing
	talend architecture
	context
	environment
	tmap
	eclipse
	database
	java
	postgresql
	tmap
	database

The values are normalized and displayed in a table cell on the console.

tOracleInput



tOracleInput properties

The properties of the generic component, **tDBInput**, apply to the **tOracleInput** component, therefore the properties are not detailed again here.

For detailed information about generic **tDBInput** properties, see *tDBInput properties on page 136*.

Related scenarios

Related topics in **tDBInput** scenarios:

- *Scenario 1: Displaying selected data from DB table on page 137*
- *Scenario 2: Using StoreSQLQuery variable on page 138*

Related topic in **tContextLoad** Scenario: *Dynamic context use in MySQL DB insert on page 123*.

tOracleBulkExec



tOracleBulkExec properties

Component family	Databases	 
Function	tOracleBulkExec inserts, appends, replaces or truncate data in an Oracle database.	
Purpose	As a dedicated component, it allows gains in performance during operations performed on data of an Oracle database.	
Properties	<i>Property type</i>	Either Built-in or Repository.
		Built-in: No existing property data is to be retrieved
		Repository: Select the Repository file where Properties are stored. The following fields are pre-filled in using fetched data.
	Service Name	Oracle Service Name or SID in Oracle database. Note: In Java projects, the full database connection details are required.
	<i>Username and Password</i>	DB user authentication data.
	<i>Table</i>	Name of the table to be written. Note that only one table can be written at a time
	<i>Action on data</i>	On the data of the table defined, you can perform: Insert: Inserts rows to an empty table. If duplicates are found, job stops. Append: Add rows to the existing data of the table Replace: Overwrites some rows of the table Truncate: Drops table entries and inserts new input flow data.
	Data File Name	Name of the file to be processed. Related topic: <i>Defining the context variables on page 96</i>
	Fields terminated by	Character, string or regular expression to separate fields.
	Fields optionnally enclosed by	Data enclosure characters.
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.

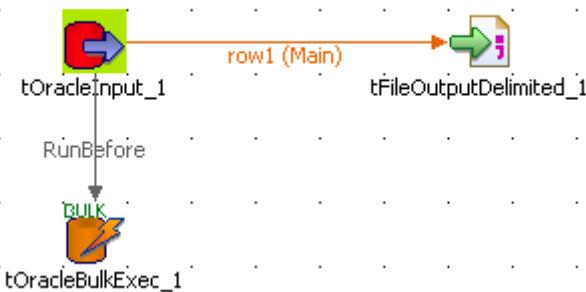
Components

tOracleBulkExec

	Output to	Console: Loading information Global variable: returned values from ctl, bad or log files.
Usage	This dedicated component offers performance and flexibility of Oracle DB query handling.	

Scenario: Truncating and inserting file data into Oracle DB

This scenario describes how to truncate the content of an Oracle DB and load an input file content. The related job is composed of three components that respectively creates the content, output this content into a file to be loaded onto the Oracle database after the DB table has been truncated.



- Click and drop the following components: **tOracleInput**, **tFileOutputDelimited**, **tOracleBulkExec**
- Connect the **tOracleInput** with the **tFileOutputDelimited** using a **row main** link.
- And connect the **tOracleInput** to the **tOracleBulkExec** using a **ThenRun** trigger link.
- Define the Oracle connection details. We recommend you to store the DB connection details in the Metadata repository in order to retrieve them easily at any time in any job.

tOracleInput_1

Property Type	Repository	Repository	DB (ORACLE):Oracle_Talend	*			
Host	'talend-dbms'	Port	'1521'	Database	'TALEND'	* Schema	'ROO'
Username	'root'			Password	'toor'		
Schema Type	Repository	DB (ORACLE):Oracle_Talend - CLIENT			*	Edit schema	[...]
Query Type	Built-In	Guess Query					
Query	'SELECT ID_CONTRACT, ID_CLIENT, CONTRACT_TYPE, CONTRACT_VALUE FROM CLIENT_CONTRACT'						
Encoding Type	CUSTOM	'AL32UTF8'					

- Define the schema, if it isn't stored either in the **Repository**. In this example, the schema is as follows: *ID_Contract, ID_Client, Contract_type, Contract_Value*.
- Change the default encoding to *AL32UTF8* encoding type.
- Define the **tFileOutputDelimited** component parameters, including output **File Name**, **Row separator** and **Fields delimiter**.
- Set also the **encoding** to the Oracle encoding type as above.
- Then double-click on the **tOracleBulkExec** to define the DB feeding properties.

tOracleBulkExec_1

Property Type	Repository	Repository	DB (ORACLE):Oracle_Talend	*
Service name	'TALEND'			
Username	'root'	Password		'toor'
Table	'emp'	Action on data		Insert
Data file name	'C:/talend_files/emp_bulk.txt'			
Fields terminated by	','	Fields optionally enclosed by		'''
Encoding Type	CUSTOM	'AL32UTF8'		* Output to console

- Fill in the DB connection details if they are not available from the Repository.
- Fill in the name of the **Table** to be fed and the Action on data to be carried out, in this use case: **insert**.
- Define the **encoding** as in preceding steps.
- For this scenario, the log **output** is to be displayed in the **console**.

Components

tOracleBulkExec

Press **F6** to run the job. The log output displays in the **Run Job** tab and the table is fed with the parameter file data.

Related topic: *Scenario: Inserting data in MySQL database on page 273*

tOracleOutput



tOracleOutput properties

The properties of the generic component, **tDBOutput**, apply to the **tOracleOutput** component, therefore the properties are not detailed again here.

For detailed information about generic **tDBOutput** properties, see *DBOutput properties on page 140*.

Related scenarios

For **tOracleOutput** related topics, see:

- **tDBOutput Scenario: Displaying DB output on page 142**
- **tMySQLOutput Scenario: Adding new column and altering data on page 261.**

tOracleRow



tOracleRow properties

The properties of the generic component, **tDBSQLRow**, apply to the **tOracleRow** component, therefore the properties are not detailed again here.

For detailed information about generic **tDBSQLRow** properties, see *tDBSQLRow properties on page 144*.

Related scenarios

For **tOracleRow** related topics, see:

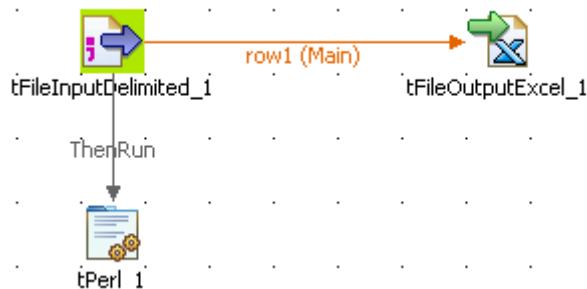
- **tDBSQLRow Scenario 1: Resetting a DB auto-increment on page 145**
- **tMySQLRow Scenario: Removing and regenerating a MySQL table index on page 275.**

tPerl**tPerl properties**

Component family	Processing	
Function	tPerl transforms any data entered as argument of Perl commands.	
Purpose	tPerl is an (Perl) editor that is a very flexible tool within a job.	
Properties	Code	Type in the Perl code based on the command and task you need to perform. For further information about Perl functions syntax, see JasperETL online Help (under Talend Open Studio User Guide > Perl)
Usage	Typically used for debugging but can also be used to display a variable content.	
Limitation	This component requires an advanced Perl user level and is not meant to be used with a Row connection as is meant for single use.	

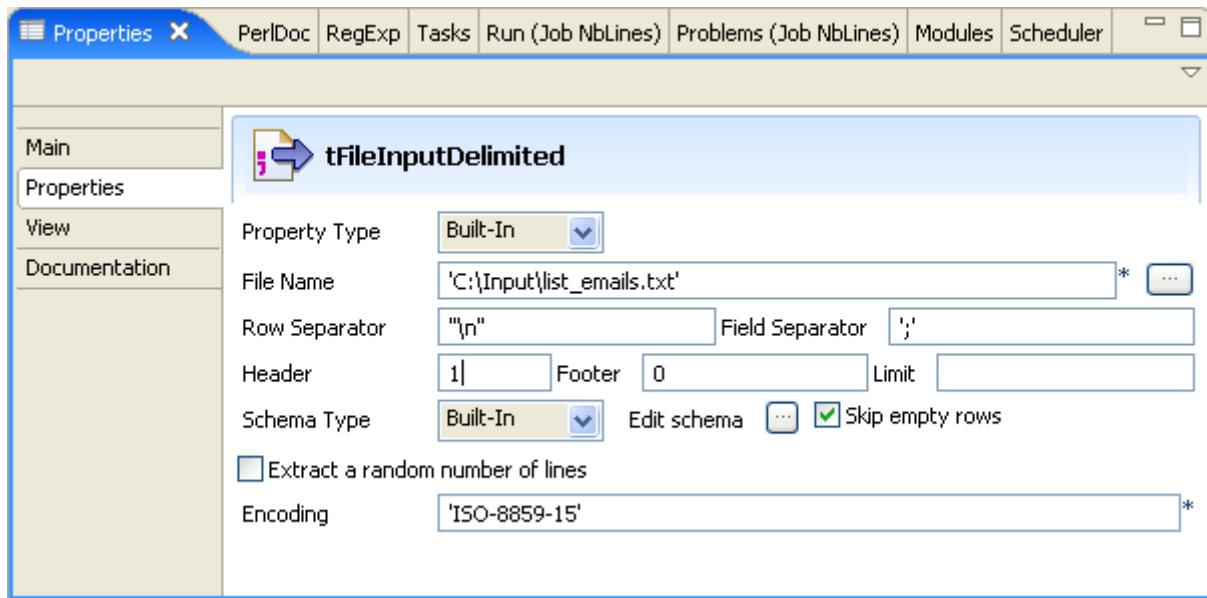
Scenario: Displaying number of processed lines

This scenario is a three-component job showing in the Log the number of rows being processed and output in an XML file.

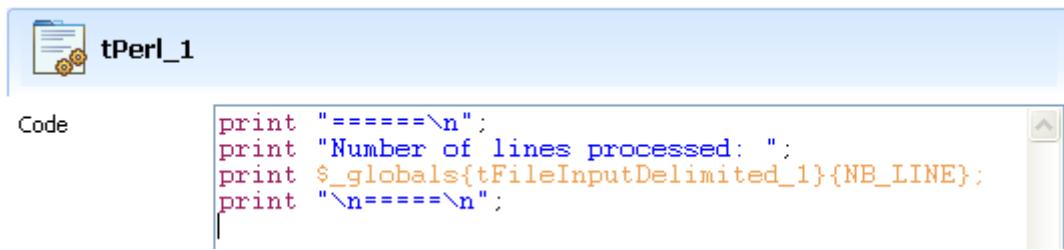


- Click and drop three components from the Palette to the workspace: **tFileInputDelimited**, **tFileOutputExcel**, **tPerl**
- Right-click on the **tFileInputDelimited** object and connect it to the **tFileOutputExcel** component using a main Row.
- Right-click again on **tFileInputDelimited** and link it with the **tPerl** component using a **Trigger > ThenRun** link. This link means that, following the arrow direction, the first component (**tFileDelimited**) will run before the second component (**tPerl**).

- Click once on **tFileInputDelimited** and select **Properties** tab to define the component properties.



- The **Properties** are not reused from or for another job stored in the repository, but instead are used for this job only. Therefore select **Built-In** in the drop-down list.
- Enter a path or browse to the file containing the data to be processed. In this example, the text file gathers a list of names facing the relevant email addresses.
- Define the **Row** and **Field** separators. In this scenario, there is one name and the matching email per row. And the fields are separated by a semi-colon.
- The first row of the file contains the labels of the columns, therefore it should be ignored in the job. Therefore the Header field value is 1.
- There is no footer nor limit value to be defined for this scenario.
- The **Schema type** is also built-in in this case. Click on **Edit Schema** and describe the content of the input file. In this scenario, there are two columns labelled Name and Emails, of type String and with no length defined. Key field being Email.
- Select the **tFileOutputExcel** component and define it accordingly.
- Select the output file path, Sheet and synchronize the schema.
- Then define the **tPerl** sub-job in order to get the number of rows transferred to the XML Output file.

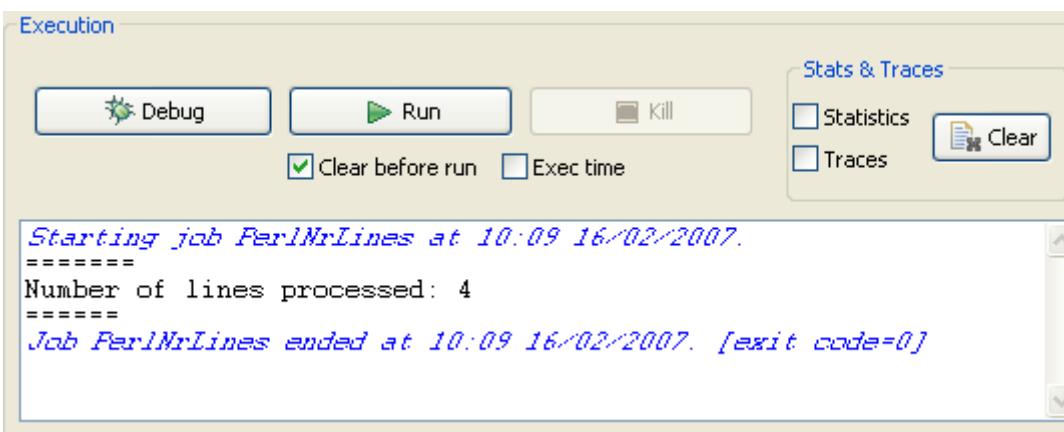


The screenshot shows the configuration window for the tPerl component. The title bar is labeled "tPerl_1". On the left, there is a "Code" section containing the following Perl script:

```
print "=====\\n";
print "Number of lines processed: ";
print ${_globals{tFileInputDelimited_1}}{NB_LINE};
print "\\n=====\\n";
```

- Enter the Perl command `print` to get the variable containing the number of rows read in the `tFileInputDelimited`. To access the list of available variables, press **Ctrl+Space** then select the relevant variable in the list.
- For a better readability in the **Run Job** log, add equal signs before and after the commands. Note also that commands, strings and variables are coloured differently.
- Then switch to the **Run Job** tab and execute the job.

The job runs smoothly and creates an output XML file following the two-field schema defined: Name and Email.



The screenshot shows the "Execution" tab of the job log. It includes buttons for "Debug", "Run" (which is highlighted), and "Kill". There are checkboxes for "Clear before run" (checked) and "Exec time" (unchecked). A "Stats & Traces" panel on the right has checkboxes for "Statistics" and "Traces", with a "Clear" button. The main log area displays the following text:

```
Starting job PerlNrLines at 10:09 16/02/2007.
=====
Number of lines processed: 4
=====
Job PerlNrLines ended at 10:09 16/02/2007. [exit code=0]
```

The Perl command result is shown in the job log.

tPostgresqlInput



tPostgresqlInput properties

The properties of the generic component, **tDBInput**, apply to the **tPostgresqlInput** component, therefore the properties are not detailed again here.

For detailed information about generic **tDBInput** properties, see *tDBInput properties on page 136*.

Related scenarios

Related topics in **tDBInput** scenarios:

- *Scenario 1: Displaying selected data from DB table on page 137*
- *Scenario 2: Using StoreSQLQuery variable on page 138*

Related topic in **tContextLoad** Scenario: *Dynamic context use in MySQL DB insert on page 123*.

tPostgresqlOutput



tPostgresqlOutput properties

The properties of the generic component, **tDBOutput**, apply to the **tPostgresqlOutput** component, therefore the properties are not detailed again here.

For detailed information about generic **tDBOutput** properties, see *DBOutput properties on page 140*.

Related scenarios

For **tPostgresqlOutput** related topics, see:

- **tDBOutput Scenario: Displaying DB output on page 142**
- **tMySQLOutput Scenario: Adding new column and altering data on page 261.**

tPostgresqlRow



tPostgresqlRow properties

The properties of the generic component, **tDBSQLRow**, apply to the **tPostgresqlRow** component, therefore the properties are not detailed again here.

For detailed information about generic **tDBSQLRow** properties, see *tDBSQLRow properties on page 144*.

Related scenarios

For **tPostgresqlRow** related topics, see:

- **tDBSQLRow Scenario 1: Resetting a DB auto-increment on page 145**
- **tMySQLRow Scenario: Removing and regenerating a MySQL table index on page 275.**



tRowGenerator

tRowGenerator properties

Component family	Misc	
Function	tRowGenerator generates as many rows and fields as needed using random values taken in a list.	
Purpose	Can be used to create an input flow in a job for testing purpose in particular for boundary test sets	
Properties	Row generation editor	The editor allows you to define precisely the columns and nature of data to be generated. You can use predefined routines or type in yourself the function to be used to generate the data specified
Usage	The tRowGenerator Editor's ease of use allows users without any Perl or Java knowledge to generate random data for test purpose.	
Limitation	n/a	

The **tRowGenerator** Editor opens up on a separate window made of two parts:

- a **Schema** definition panel at the top of the window
- and a **Function** definition and preview panel at the bottom.

Defining the schema

First you need to define the structure of data to be generated.

- Add as many columns to your schema as needed, using the **plus (+)** button.
- Type in the names of the columns to be created in the **Columns** area and check the **Key** box if required
- Make sure you define then the nature of the data contained in the column, by selecting the **Type** in the list. According to the type you select, the list of **Functions** offered will differ. This information is therefore compulsory.

Schema				Functions	Preview
Column	Key	Type	Nullable	Functions	Preview
ID_employees	<input checked="" type="checkbox"/>	int	<input type="checkbox"/>	sequence	2
First_Name	<input type="checkbox"/>	String	<input type="checkbox"/>	...	Phoebe H
Last_Name	<input type="checkbox"/>	String	<input type="checkbox"/>	lastName	Eisenhower
Hire_Date	<input type="checkbox"/>	Day	<input type="checkbox"/>	getRandomDate	2008-08-31

Toolbar:

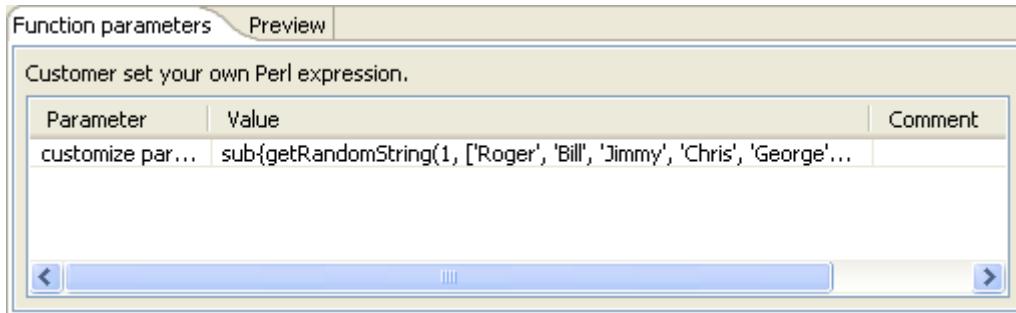
- Some extra information, although not required, might be useful such as **Length**, **Precision** or **Comment**. You can also hide these columns, by clicking on the **Columns** drop-down button next to the toolbar, and unchecking the relevant entries on the list.
- In the **Function** area, you can select the predefined routine/function if one of them corresponds to your needs. You can also add to this list any routine you stored in the **Routine** area of the **Repository**. Or you can type in the function you want to use in the **Function** definition panel. Related topic: *Defining the function on page 297*
- Click **Refresh** to have a preview of the data generated.
- Type in a number of rows to be generated. The more rows to be generated, the longer it'll take to carry out the generation operation.

Note: Note that the functions list differs from Perl to Java.

Defining the function

You selected the three dots [...] as **Function** in the Schema definition panel, as you want to customize the function parameters.

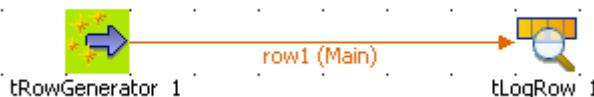
- Select the **Function parameters** tab
- The **Parameter** area displays **Customized parameter** as function name (read-only)



- In the **Value** area, type in the Perl or Java function to be used to generate the data specified.
- Click on the **Preview** tab and click **Preview** to check out a sample of the data generated.

Scenario: Generating random java data

The following scenario creates a two-component job made in Java, generating 50 rows structured as follows: a randomly picked-up ID in a 1-to-3 range, a random ascii First Name and Last Name generation and a random date taken in a defined range.



- Click and drop a **tRowGenerator** and a **tLogRow** component from the Palette to the workspace.
- Right-click on the **tRowGenerator** component and select *Row > Main*. Drag this main row link onto the **tLogRow** component and release when the plug symbol displays.
- Double-click on the **tRowGenerator** component to open the Editor.
- Define the fields to be generated.

Components

tRowGenerator

Schema				Functions		Preview
Column	Key	Type	Nullable	Func...	Parameters	Preview
Random_ID	<input type="checkbox"/>	int	<input type="checkbox"/>	...	1,2,3	3
First_Name	<input type="checkbox"/>	String	<input type="checkbox"/>	getAs...	length=>6 ;	bF1lpb
Last_Name	<input type="checkbox"/>	String	<input type="checkbox"/>	getAs...	length=>6 ;	2lT4mM
Date	<input type="checkbox"/>	Date	<input type="checkbox"/>	getRa...	min =>"2004-01-...	Sun Jun 29 11:19:3...

Columns 50

- The random ID column is of integer type, the First and Last names are of string type and the Date is of date type.
- In the **Function** list, select the relevant function or set on the three dots for custom function.
- On the **Function parameters** tab, define the Values to be randomly picked up.

Parameter	Value	Comment
customize parameter	1,2,3	

- First_Name* and *Last_Name* columns are to be generated using the `getAsciiRandomString` function that is predefined in the system routines. By default the length defined is 6 character-long. But you can change it if need be.
- The *Date* column calls the also predefined `getRandomDate` function. You can edit the parameter values in the **Function parameters** tab.
- Set the **Number of Rows** to be generated to 50.
- Click OK to validate the setting.
- Double-click on the tLogRow component to view the properties. The default setting is retained for this job.
- Press **F6** to run the job.

```
Starting job JavaRowGenerate at 11:44 10/04/2007.
Running process with context: Default
1|iH6mtj|y3hSXH|Wed Apr 16 11:44:15 CEST 2008
3|cityeQ|uvnKkO|Sun Jan 06 11:44:16 CET 2008
2|X2O0DP|1SVzKT|Wed Oct 26 11:44:16 CEST 2005
1|DSLuE1|S8u15i|Sat Mar 04 11:44:16 CET 2006
3|cc4znX|yuc9cf|Thu Jan 20 11:44:16 CET 2005
3|NwK3PN|lnNyDU|Mon Jun 06 11:44:16 CEST 2005
3|CtO6Ba|pCQgwp|Sat Aug 07 11:44:16 CEST 2004
3|XMt7KN|SUIzFn|Sun May 09 11:44:16 CEST 2004
1|jiPor7|145rp7|Wed Aug 25 11:44:16 CEST 2004
1|OBN4TX|hywNdP|Sat Feb 09 11:44:16 CET 2008 [ ]
2|kbVUuE|s21FA0|Tue Jan 27 11:44:16 CET 2004
2|zQLteU|VNhb5w|Sun Oct 05 11:44:16 CEST 2008
3|4nrniu|5Tbnxd|Fri Nov 14 11:44:16 CET 2008
2|7A0qqu|s3jEzJ|Tue Feb 13 11:44:16 CET 2007
3|FgVoTg|uYhJmF|Thu Mar 01 11:44:16 CET 2007
1|Sp4e9P|zXcTK5|Thu Apr 08 11:44:16 CEST 2004
1|E3A4Q3|vKNocj|Wed Jan 11 11:44:16 CET 2006
1|pOMGpG|koklW2|Thu Apr 22 11:44:16 CEST 2004
3|9XOgm8|CCbFya|Sat Feb 18 11:44:16 CET 2006
2|itZXWx|LhQBx1|Tue Apr 18 11:44:16 CEST 2006
1|On1dwv|E7Yaqn|Fri Dec 10 11:44:16 CET 2004
1|UPTZ9M|8s902c|Fri Nov 11 11:44:16 CET 2005
1|WhIT0u|KWn8hO|Mon May 07 11:44:16 CEST 2007
1|LbbeFv|Evn1lc|Thu Mar 16 11:44:16 CET 2006
```

The 50 rows are generated following the setting defined in the **tRowGenerator** editor and the output is displayed in the **Run Job** console.

Components

tRunJob

tRunJob

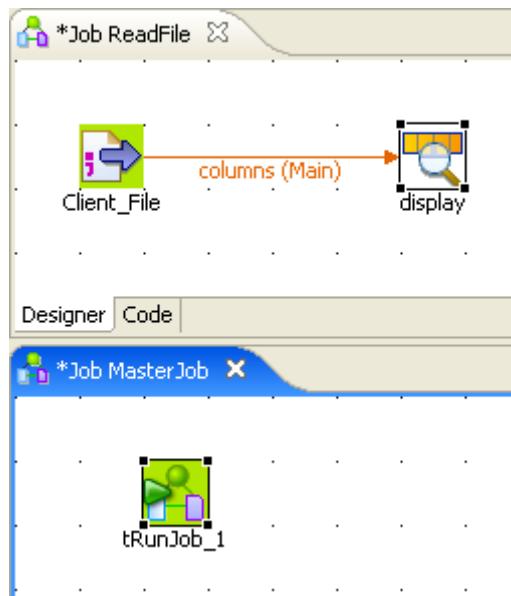


tRunJob Properties

Component family	System	
Function	Executes the job called in the component's Properties, in the frame of the context defined.	
Purpose	tRunJob helps mastering complex job systems which need to execute one job after another.	
Properties	<i>Process</i>	Select the job to be called in and processed. Make sure you already executed once the job called, beforehand, in order to ensure a smooth run through the tRunJob.
	<i>Context</i>	If you defined contexts and variables for the job to be run by the tRunJob , select the applicable context entry on the list.
	<i>Context parameter</i>	You can change the selected context parameters. Click the plus button to add the parameters as defined in the Context of the child job
	<i>Generate Code</i>	Click the button to validate the context selection in the tRunJob and generate the relevant code.
Usage	This component can be used as a standalone job or can help clarifying complex job by avoiding having too many sub-jobs all together in one job.	
Limitation	n/a	

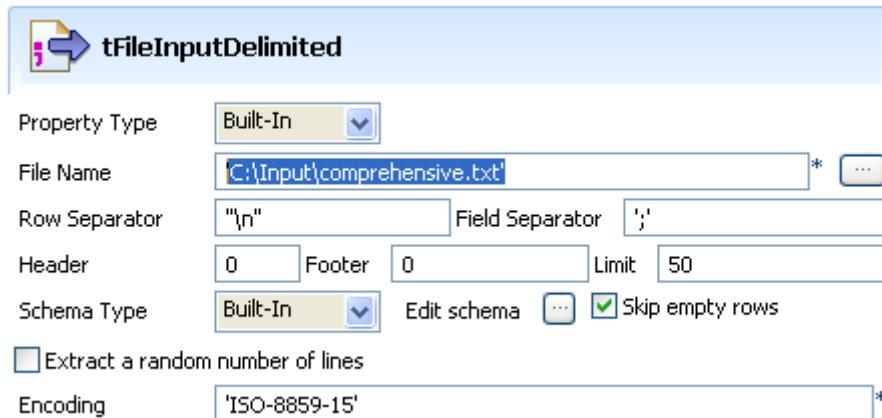
Scenario: Executing a remote job

This particular scenario describes a single-component job calling in and executing another job. The job to be executed reads a basic delimited file and simply displays its content on the Run Job log console. The particularity of this job lies in the fact that this latter job is executed from a separate job and uses a context variable to prompt for the input file to be processed.

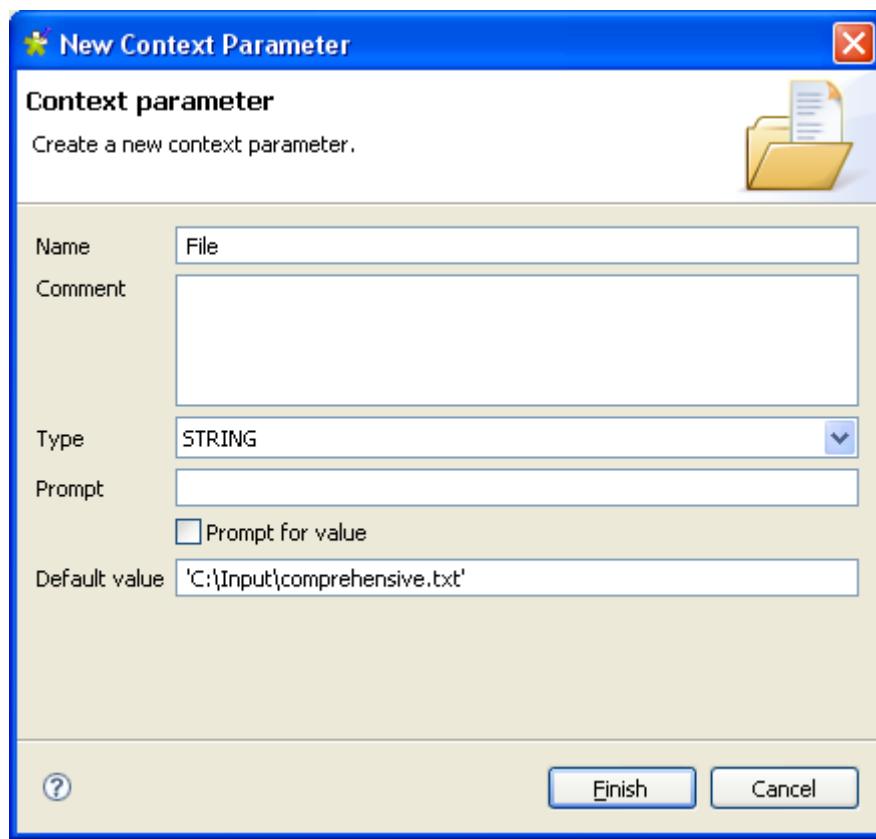


Create the first job reading the delimited file.

- Click and drop a **tFileInputDelimited** and a **tLogRow** onto the Designer.
- Set the input component properties in the **Properties** panel.
- Set the **Property type** on **Built-In** for this job.



- In **File Name**, browse to the input file. In this example, the file is a txt file called Comprehensive.
- Select the path to this Input file and press **F5** to open the Variable configuration window.
- Give a name to the new context variable, in this scenario, it is called File.



- No need to check the **Prompt for value** box nor set a prompt message for this use case, as the default parameter value is ok to be used.
- Click **Finish** to validate and press again **Enter** to make sure the new context variable is stored the **File Name** field.
- Back on the **Properties** view, type in the field and row separators used in the input file.
- In this case, no header nor footer are to be set. But set a limited number of rows to be processed. In the **Limit** field, type in 50.
- The **Schema type** is **Built-in** for this scenario. Click the three-dot button to configure manually the schema.
- Add two columns and name them following the first and second column name of your input file. In this example: ID and Registration.
- If you stored your schema in the repository, you only need to select the relevant metadata entry corresponding to your input file structure.
- Then link the Input component to your output component, **tLogRow**.

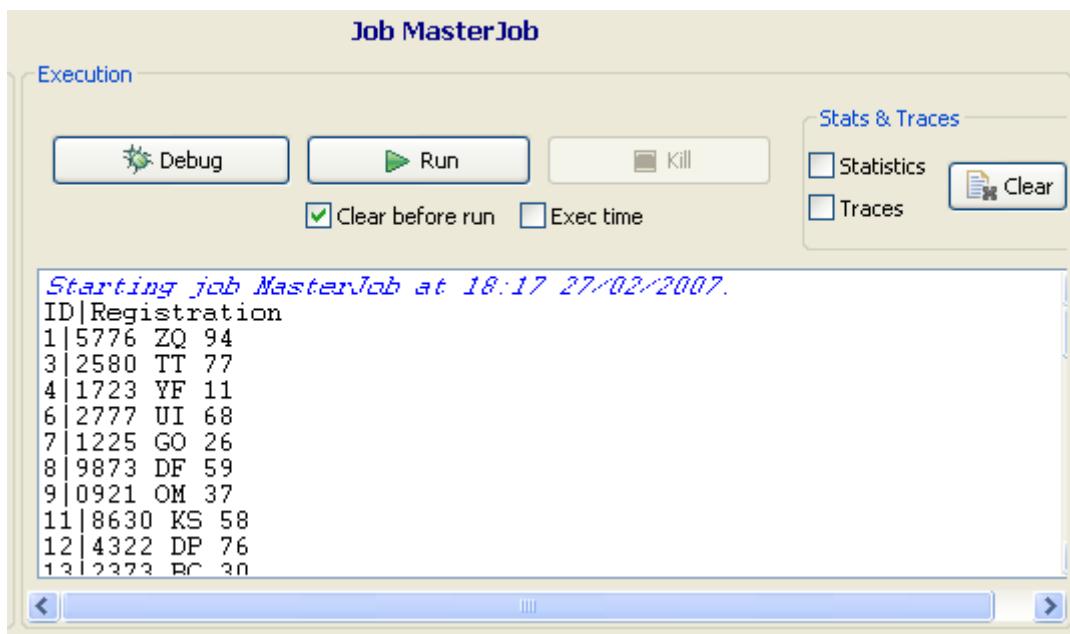
Create the second job, to play the role of master job.

- In the Properties panel of the tRunJob component, select the job to be executed.

- We recommend you to run once the called-in job before executing it through the **tRunJob** component in order to make sure it runs smoothly.



- In the **Context** field, select the relevant context. In this case, only the **Default** context is available and holds the context variable created earlier.
- Click **Generate Code** to validate the context selection and generate the related code.
- Save the master job and press **F6** to run it.



The called-in job reads the data contained in the input file, as defined by the input schema, and the result of this job is displayed directly in the **Run Job** console.

Related topic: *Scenario: Job execution in a loop on page 215*

Components

tSalesforceInput



tSalesforceInput

tSalesforceInput Properties

Component family	Business	
Function	Connects to a module of a Salesforce database via the relevant webservice.	
Purpose	Allows to extract data from a Salesforce DB based on a query.	
Properties	<i>Salesforce Webservice URL</i>	Type in the webservice URL to connect to the Salesforce DB.
	<i>Username and Password</i>	Type in the Webservice user authentication data.
	<i>Module</i>	Select the relevant module in the list
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository. Click Edit Schema to make changes to the schema. Note that if you make changes, the schema automatically becomes built-in. In this component the schema is related to the Module selected.
	<i>Query condition</i>	Type in the query to select the data to be extracted. Example: account_name= 'Talend'
Usage	Usually used as a Start component. An output component is required.	
Limitation	n/a	

Related scenario

The operation is similar to the connection to SugarCRM, therefore see scenario of *tSugarCRMInput* on page 324 for more information.

tSalesforceOutput



tSalesforceOutput Properties

Component family	Business	
Function	Writes in a module of a Salesforce database via the relevant webservice.	
Purpose	Allows to write data into a Salesforce DB.	
Properties	<i>Salesforce Webservice URL</i>	Type in the webservice URL to connect to the Salesforce DB.
	<i>Username and Password</i>	Type in the Webservice user authentication data.
	<i>Action</i>	Insert or Update the data in the Salesforce module.
	<i>Module</i>	Select the relevant module in the list
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository. Click Edit Schema to make changes to the schema. Note that if you make changes, the schema automatically becomes built-in. Click Sync columns to retrieve the schema from the previous component connected in the job.
Usage	Used as an output component. An Input component is required.	
Limitation	n/a	

Related scenario

No scenario is available yet for this component.

Components

tSendMail



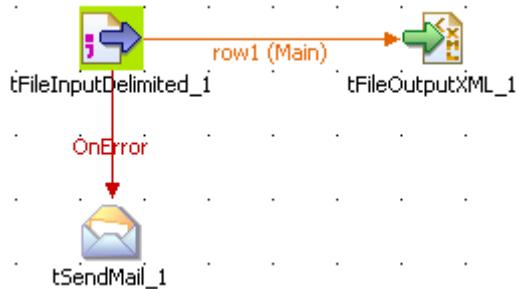
tSendMail

tSendMail properties

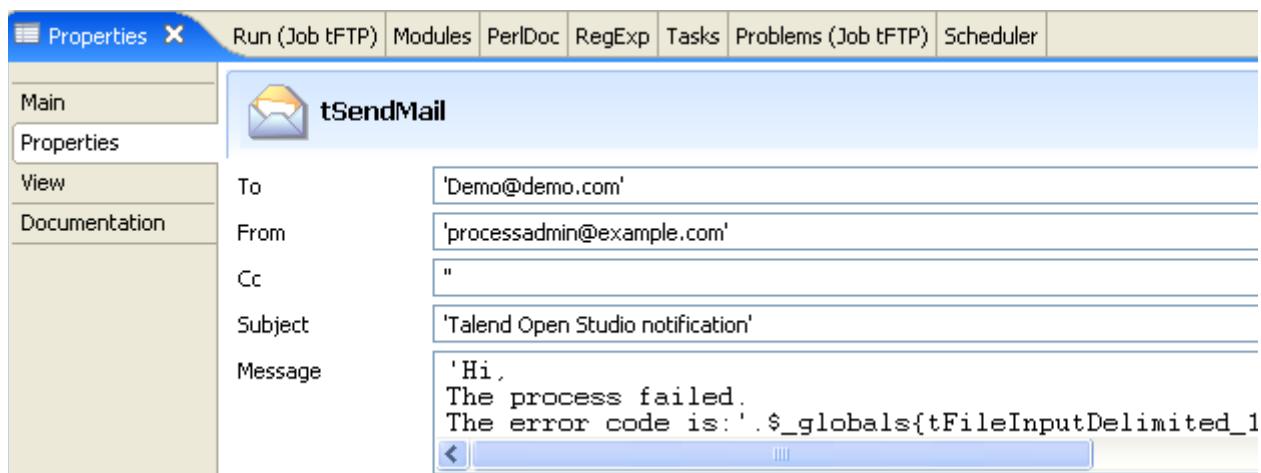
Component family	Internet	
Function	tSendMail sends emails and any attachments to defined recipients.	
Purpose	tSendMail purpose is to notify recipients about a particular state of the job or possible errors .	
Properties	<i>To</i>	Main recipient email address
	<i>From</i>	Sending server's email address
	<i>Cc</i>	Carbon copy recipient email
	<i>Subject</i>	Heading of the mail
	<i>Message</i>	Body message of the email. Press Ctrl+Space to display the list of available variables
	<i>Attachment</i>	Filmask or path to the file to be sent along with the mail, if any.
	<i>Other Headers</i>	Type in the Key and corresponding value of any header information that does not belong to the standard header.
	<i>SMTP Host and Port</i>	IP address of SMTP server used to send emails.
	Usage	This component is typically used as one sub-job but can also be used as output or end object. It can be connected to other components with either Row or Iterate links.
Limitation	Note that email sendings with or without attachment require two different perl module	

Scenario: Email on error

This scenario creates a three-component job which sends an email to defined recipients when an error occurs.



- Click and drop the following components from your palette to the workspace:
tFileInputDelimited, **tFileOutputXML**, **tSendMail**.
 - Define **tFileInputDelimited** properties. Related topic: *tFileInputDelimited properties on page 188*.
 - Right-click on the **tFileInputDelimited** component and select *Row > Main*. Then drag it onto the **tFileOutputXML** component and release when the plug symbol shows up.
 - Define **tFileOutputXML** properties.
 - Drag a **Run on Error** link from **tFileDelimited** to **tSendMail** component.
 - Define the **tSendMail** component properties:



- Enter the recipient and sender email addresses, as well as the email subject.
 - Enter a message containing the error code produced using the corresponding global variable. Access the list of variables by pressing **Ctrl+Space**.
 - Add attachments and extra header information if any. Type in the SMTP information.

Components

tSendMail

Attachments

File	'c:\Input\error.log'
------	----------------------

Other headers

Key	Value
-----	-------

SMTP host * SMTP port *

In this scenario, the file containing data to be transferred to XML output cannot be found. **tSendmail** runs on this error and sends an notification email the defined recipient.



tSleep

tSleep Properties

Component family	Misc	
Function	tSleep implements a time off in a job execution.	
Purpose	Allows to identify possible bottlenecks using a time break in the job for testing or tracking purpose. In production, it can be used for any needed pause in the job to feed input flow for example.	
Properties	<i>Pause (in second)</i>	Time in second the job execution is stopped for.
Usage	tSleep component is generally used as a middle component to make a break/pause in the job, before resuming the job.	
Limitation	n/a	

Related scenarios

For use cases in relation with tSleep, see [tFor Scenario: Job execution in a loop on page 215](#).



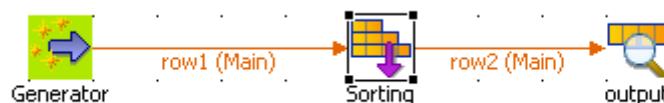
tSortRow

tSortRow properties

Component family	Processing	
Function	Sorts input data based on one or several columns, by sort type and order	
Purpose	Helps creating metrics and classification table.	
Properties	<p>Schema type and Edit Schema</p> <p>A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository.</p> <p>Click Edit Schema to make changes to the schema. Note that if you make changes, the schema automatically becomes built-in.</p> <p>Click Sync columns to retrieve the schema from the previous component connected in the job.</p> <p>Built-in: The schema will be created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i></p> <p>Repository: The schema already exists and is stored in the Repository, hence can be reused in various projects and job flowcharts. Related topic: <i>Setting a repository schema on page 52</i></p> <p>Criteria</p> <p>Click + to add as many lines as required for the sort to be complete. By default the first column defined in your schema is selected.</p> <p>Schema column: Select the column label from your schema, which the sort will be based on. Note that the order is essential as it determines the sorting priority.</p> <p>Sort type: Numerical and Alphabetical order are proposed. More sorting types to come.</p> <p>Order: Ascending or descending order.</p>	
Usage	This component handles flow of data therefore it requires input and output, hence is defined as an intermediary step.	
Limitation	n/a	

Scenario: Sorting entries

This scenario describes a three-component job. A tRowGenerator is used to create random entries which are directly sent to a tSortRow to be ordered following a defined value entry. In this scenario, we suppose the input flow contains names of salespersons along with their respective sales and their years of presence in the company. The result of the sorting operation is displayed on the Run job console.



- Click and drop the three components required for this use case: **tRowGenerator**, **tSortRow** and **tLogRow**.
- Connect them together using **Row main** links.
- On the **tRowGenerator** editor, define the values to be randomly used in the Sort component. For more information regarding the use of this particular component, see *tRowGenerator properties on page 295*.

Schema			Functions			Preview
Column	Key	Type	Nullable	Functions	Parameters	Preview
ID	<input checked="" type="checkbox"/>	int	<input type="checkbox"/>	...	sub(++\$id)	
YearsInComp	<input type="checkbox"/>	int	<input type="checkbox"/>	...	1..4	
Name	<input type="checkbox"/>	String	<input type="checkbox"/>	...	qw /Pierrick Mickael Steffie Fabrice Bertr...	
Sales	<input type="checkbox"/>	int	<input type="checkbox"/>	...	1..100	

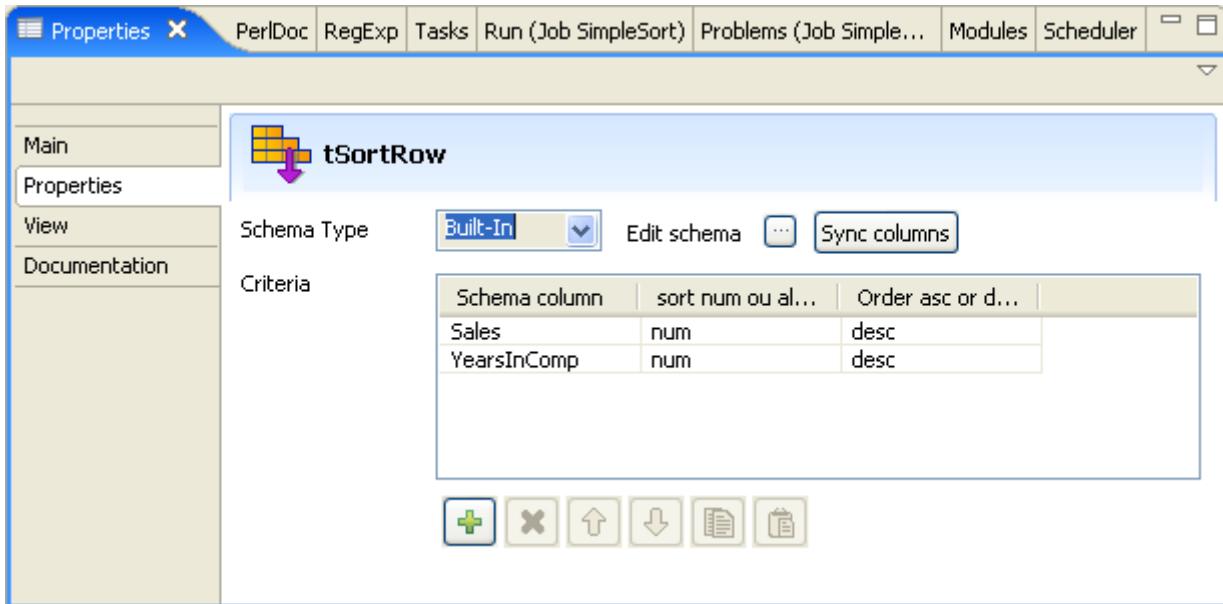
< > Columns Number of Rows for RowGenerator 10

+ X ↑ ↓ ↶ ↷

- In this scenario, we want to rank each salesperson according to its Sales value and to its number of years in the company.
- Double-click on tSortRow to display the **Properties** tab panel. Set the sort priority on the Sales value and as secondary criteria, set the number of years in the company.

Components

tSortRow



- Use the plus button to add the number of rows required. Set the type of sorting, in this case, both criteria being integer, the sort is numerical. At last, given that the output wanted is a rank classification, set the order as descending.
- Make sure you connected this flow to the output component, tLogRow, to display the result in the Job console.
- Press F6 to run the Job or go to the **Run Job** panel and click **Run**. The ranking is based first on the Sales value and second on the number of years of experience.



tSQLiteInput



tSQLiteInput Properties

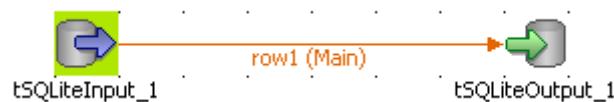
Component family	Databases	
Function	tSQLiteInput reads a database file and extracts fields based on an SQL query. As it embeds the SQLite engine, no need of connecting to any database server.	
Purpose	tSQLiteInput executes a DB query with a defined command which must correspond to the schema definition. Then it passes on rows to the next component via a Main row link.	
Properties	Database	Filepath to the SQLite database file.
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields to be processed and passed on to the next component. The schema is either built-in or remotely stored in the Repository.
		Built-in: The schema is created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>
		Repository: The schema already exists and is stored in the Repository, hence can be reused. Related topic: <i>Setting a repository schema on page 52</i>
	<i>Query type</i>	The query can be built-in for a particular job or for commonly used query, it can be stored in the repository to ease the query reuse.
	<i>Query</i>	If your query is not stored in the Repository, type in your DB query paying particularly attention to properly sequence the fields in order to match the schema definition.
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.
Usage	This component is standalone as it includes the SQLite engine. This is a startable component that can initiate a data flow processing.	

Scenario: Filtering SQLite data

This scenario describes a rather simple job which uses a select statement based on a filter to extract rows from a source SQLite Database and feed an output SQLite table.

Components

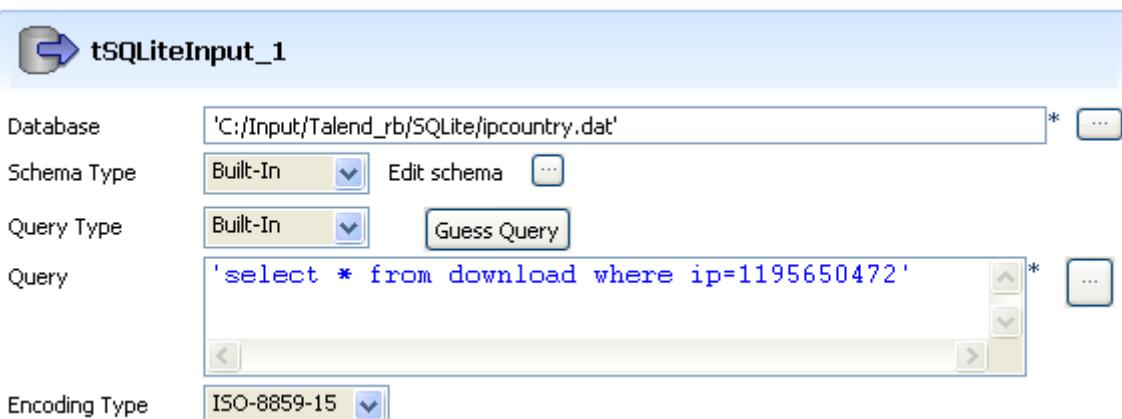
tSQLiteInput



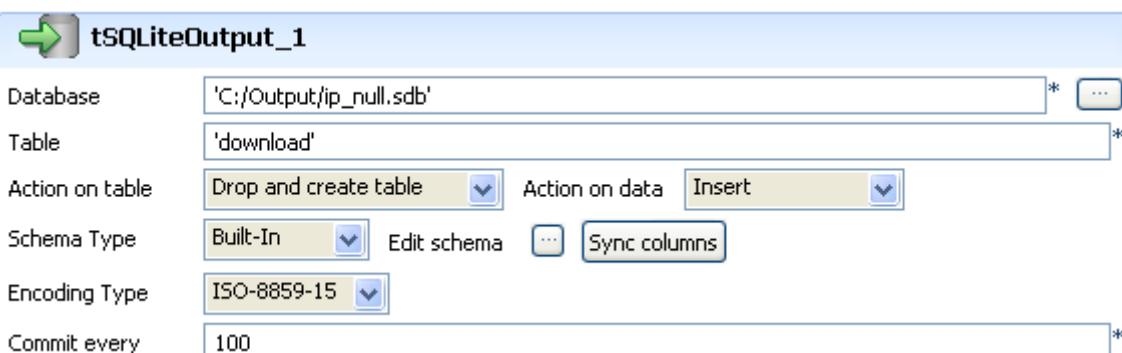
- Click and drop from the **Palette**, a **tSQLiteInput** and a **tSQLiteOutput** component.
- Connect the input to the output using a row main link.
- On the **tSQLiteInput** properties, type in or browse to the SQLite Database input file.

	id	version	download_date	ip	type	type_os	
1	20027	TOS-Win32-200610	13/11/2006	1191947907	1	101	
2	20028	TOS-Win32-200610	13/11/2006	1195650472	1	102	
3	20030	TOS-Win32-200610	13/11/2006	3375565745	1	103	
4	20031	TOS-Win32-200610	13/11/2006	1195650472	1	104	
5	20032	TOS-Win32-200610	13/11/2006	1195650472	1	105	
6	20033	TOS-Win32-200610	13/11/2006	1104872453	1	106	
7	20034	TOS-Win32-200610	13/11/2006	1104872453	1	107	
8	20036	TOS-Win32-200610	13/11/2006	1190898057	1	108	
9	20037	TOS-Win32-200610	13/11/2006	1190898057	1	109	
10	20038	TOS-Win32-200610	13/11/2006	1348977142	1	110	
11	20040	TOS-Win32-200610	13/11/2006	3581349521	1	11	
12	20041	TOS-Win32-200610	13/11/2006	1190898057	1	12	
13	20043	TOS-Win32-200610	13/11/2006	1196485544	1	13	
14	20044	TOS-Win32-200610	13/11/2006	1066743463	1	14	
15	20045	TOS-Win32-200610	13/11/2006	1196485544	1	15	
16	20046	TOS-Win32-200610	13/11/2006	2024217704	1	16	

- The file contains hundreds of lines and includes an **ip** column which the select statement will based on
- On the **tSQLite** Properties, edit the schema for it to match the table structure.



- In the **Query** field, type in your select statement based on the *ip* column.
- Select the right encoding parameter.
- On the **tSQLiteOutput** component **Properties** panel, select the **Database** filepath.



- Type in the **Table** to be fed with the selected data.
- Select the **Action on table** and **Action on Data**. In this use case, the action on table is *Drop and create* and the action on data is *Insert*.
- The schema should be synchronized with the input schema.
- Select the encoding and define the threshold to commit.
- Save the job and run it.

	id	version	download_date	ip	type	type_os
1	20028	TOS-Win32-200610	13/11/2006	1195650472	1	102
2	20031	TOS-Win32-200610	13/11/2006	1195650472	1	104
3	20032	TOS-Win32-200610	13/11/2006	1195650472	1	105

The queried data are returned in the defined SQLite file.

tSQLiteOutput



tSQLiteOutput Properties

Component family	Databases	
Function	tSQLiteOutput writes, updates, makes changes or suppresses entries in an SQLite database. As it embeds the SQLite engine, no need of connecting to any database server.	
Purpose	tSQLiteOutput executes the action defined on the table and/or on the data contained in the table, based on the flow incoming from the preceding component in the job.	
Properties	<i>Property type</i>	Either Built-in or Repository.
		Built-in: No existing property data is to be retrieved
		Repository: Select the Repository file where Properties are stored. The following fields are pre-filled in using fetched data.
	<i>Database</i>	Filepath to the Database file
	<i>Table</i>	Name of the table to be written. Note that only one table can be written at a time
<i>In Java, use tCreateTable as substitute for this function..</i>	<i>Action on table</i>	On the table defined, you can perform one of the following operations: None: No operation carried out Drop and create the table: The table is removed and created again Create a table: The table doesn't exist and gets created. Clear a table: The table content is deleted
	<i>Action on data</i>	On the data of the table defined, you can perform: Insert: Add new entries to the table. If duplicates are found, job stops. Update: Make changes to existing entries Insert or update: Add entries or update existing ones. Update or insert: Update existing entries or create it if non existing Delete: Remove entries corresponding to the input flow.
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields to be processed and passed on to the next component. The schema is either built-in or remotely stored in the Repository.

		Built-in: The schema is created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>
		Repository: The schema already exists and is stored in the Repository, hence can be reused. Related topic: <i>Setting a repository schema on page 52</i>
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.
 Perl components do not include this feature yet.	<i>Additional Columns</i>	This option is not offered if you create (with or without drop) the Db table. This option allows you to perform actions on columns, which are not insert, nor update or delete actions or requires a particular preprocessing.
		Name: Type in the name of the schema column to be altered or inserted as new column
		SQL expression: Type in the SQL statement to be executed in order to alter or insert the relevant column data.
		Position: Select Before, Replace or After, following the action to be performed on the reference column.
		Reference column: Type in a column of reference that the tDBOutput can use to place or replace the new or altered column.
	Commit every	Number of rows to be completed before committing batches of rows together into the DB. This option ensures transaction quality (but not rollback) and above all better performance on executions.
Usage	This component is required to be connected to an Input component.	

Related Scenario

For scenarios related to **tSQLiteOutput**, see *tSQLiteInput on page 313*.

tSQLiteRow



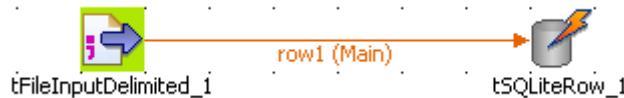
tSQLiteRow Properties

Component family	Databases	
Function	tSQLiteRow executes the defined query onto the specified database and uses the parameters bound with the column .	
Purpose	A prepared statement uses the input flow to replace the placeholders with the values for each parameters defined. This component can be very useful for updates.	
Properties	<i>Property type</i>	Either Built-in or Repository.
		Built-in: No existing property data is to be retrieved
		Repository: Select the Repository file where Properties are stored. The following fields are pre-filled in using fetched data.
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields to be processed and passed on to the next component. The schema is either built-in or remotely stored in the Repository.
		Built-in: The schema is created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>
		Repository: The schema already exists and is stored in the Repository, hence can be reused. Related topic: <i>Setting a repository schema on page 52</i>
	<i>Query type</i>	Either Built-in or Repository.
		Built-in: Fill in manually the query statement or build it graphically using SQLBuilder
		Repository: Select the relevant query stored in the Repository. The Query field gets accordingly filled in.
	<i>Query</i>	Enter your DB query paying particularly attention to properly sequence the fields in order to match the schema definition.
	<i>Prepared statement and Input parameters</i>	Check the Prepared statement box, to display the Input parameters table. In the table, click the plus button to add a row for each parameters invoked in the query.
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.

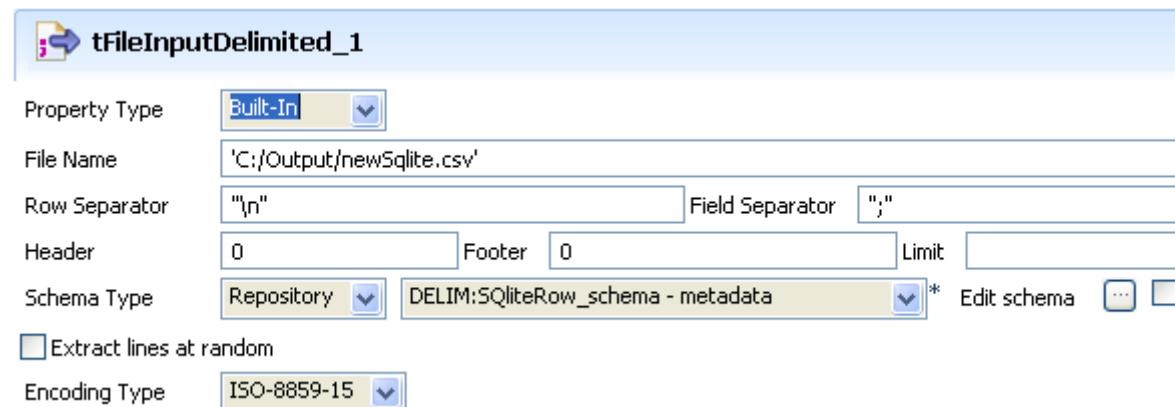
	<i>Commit every</i>	Number of rows before committing
--	---------------------	----------------------------------

Scenario: Updating SQLite rows

This scenario describes a job which updates an SQLite database file based on a prepared statement and using a delimited file.



- Click and drop a **tFileInputDelimited** and a **tSQLiteRow** component.
- On the **tFileInputDelimited Properties** panel, browse to the input file that will be used to update rows in the database.



- There is no header nor footer. The Row separator is a carriage return and the field separator is a semi-colon.
- Edit the schema in case it is not stored in the Repository.

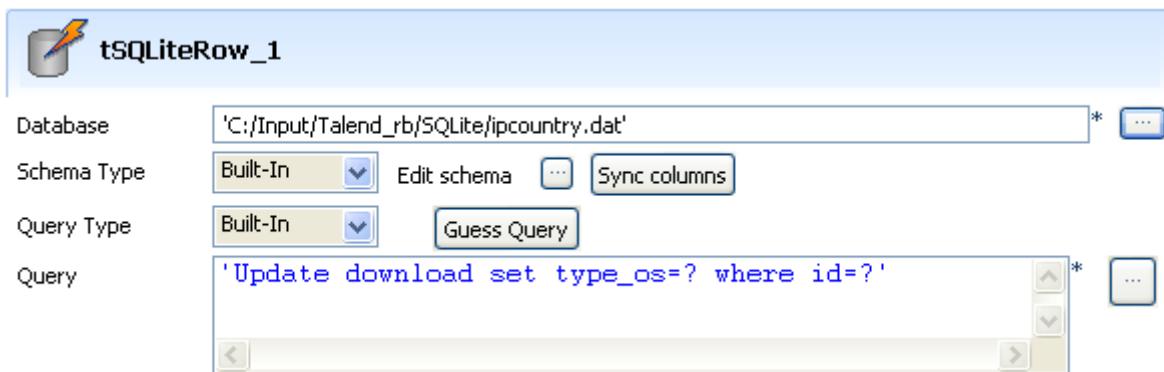
tFileInputDelimited_1							
Column	Key	Type	Nullable	Length	Precision	Comment	
id	<input type="checkbox"/>	int	<input checked="" type="checkbox"/>	6			
version	<input type="checkbox"/>	String	<input checked="" type="checkbox"/>	40			
download_date	<input type="checkbox"/>	String	<input checked="" type="checkbox"/>	20			
ip	<input type="checkbox"/>	String	<input checked="" type="checkbox"/>	20			
type	<input type="checkbox"/>	int	<input checked="" type="checkbox"/>	1			
type_os	<input type="checkbox"/>	int	<input checked="" type="checkbox"/>	3			

- Make sure the length and type are respectively correct and large enough to define the columns.

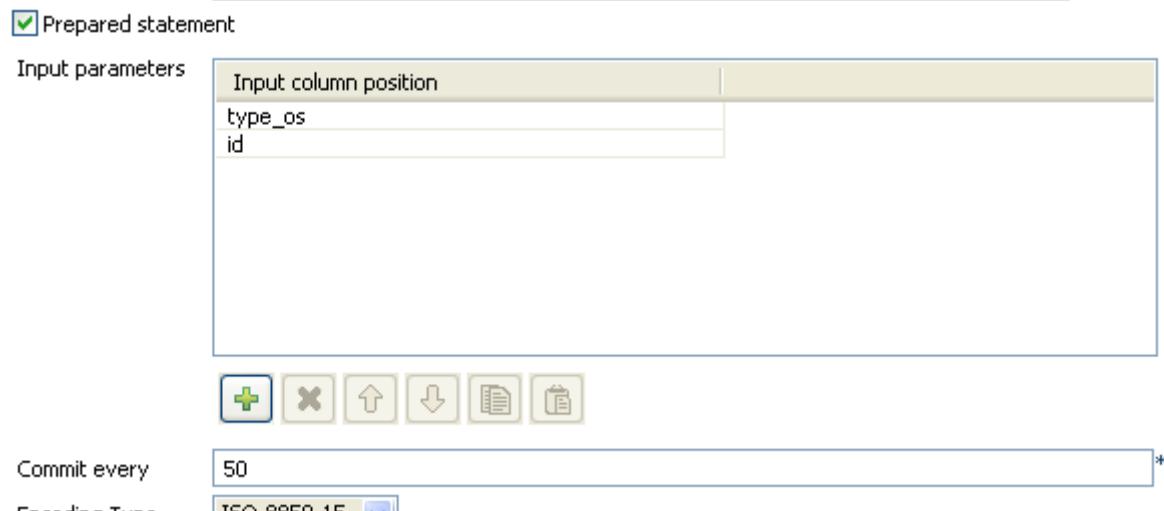
Components

tSQLiteRow

- Then in the **tSQLiteRow Properties** panel, set the **Database** filepath to the file to be updated.



- The schema is read-only as it is required to match the input schema.
- Type in the query or retrieve it from the Repository. In this use case, we updated the *type_os* for the *id* defined in the Input flow. The statement is as follows: 'Update download set type_os=? where id=?'
- Then check the **Prepared statement** box to display the placeholders' parameter table.



- In the Input parameters table, add as many lines as necessary to cover all placeholders. In this scenario, *type_os* and *id* are to be defined.
- Set the **Commit every** field and select the **Encoding type** in the list.
- Save the job and press **F6** to run it.

The *dowload* table from the SQLite database is thus updated with new *type_os* code according to the delimited input file.

tStatCatcher



tStatCatcher properties

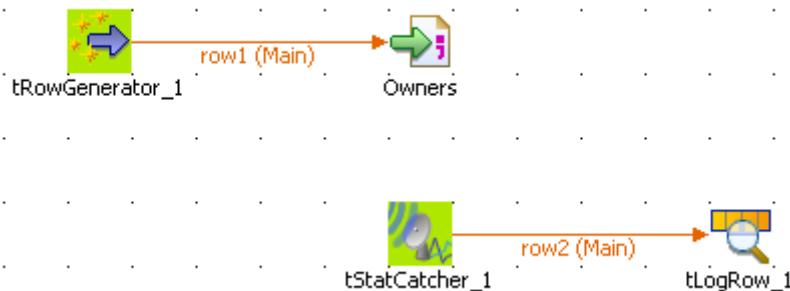
Component family	Log & Error	 
Function	Based on a defined schema, gathers the job processing metadata at a job level as well as at each component level.	
Purpose	Operates as a log function triggered by the StatsCatcher Statistics checkbox of individual components, and collects and transfers this log data to the output defined.	
	Schema type	A schema is a row description, i.e., it defines the fields to be processed and passed on to the next component. In this particular case, the schema is read-only, as this component gathers standard log information including:
		Moment: Processing time and date
		Pid: Process ID
		Father_pid: Process ID of the father job if applicable. If not applicable, Pid is duplicated.
		Root-pid: Process ID of the root job if applicable. If not applicable, pid of current job is duplicated.
		Project: Project name, the job belongs to.
		Job: Name of the current job
		Context: Name of the current context
		Origin: Name of the component if any
		Message: Begin or End.
Usage	This component is the start component of a secondary job which triggers automatically at the end of the main job. The processing time is also displayed at the end of the log.	
Limitation	n/a	

Scenario: Displaying job stats log

This scenario describes a four-component job, aiming at displaying on the Run Job console the statistics log fetched from the file generation through the tStatCatcher component.

Components

tStatCatcher



- Click and drop the required components: **tRowGenerator**, **tFileOutputDelimited**, **tStatCatcher** and **tLogRow**
- In the **Properties** panel of **tRowGenerator**, define the data to be generated. For this job, the schema is composed of three columns: *ID_Owner*, *Name_Customer* and *ID_Insurance*, generated using Perl script.

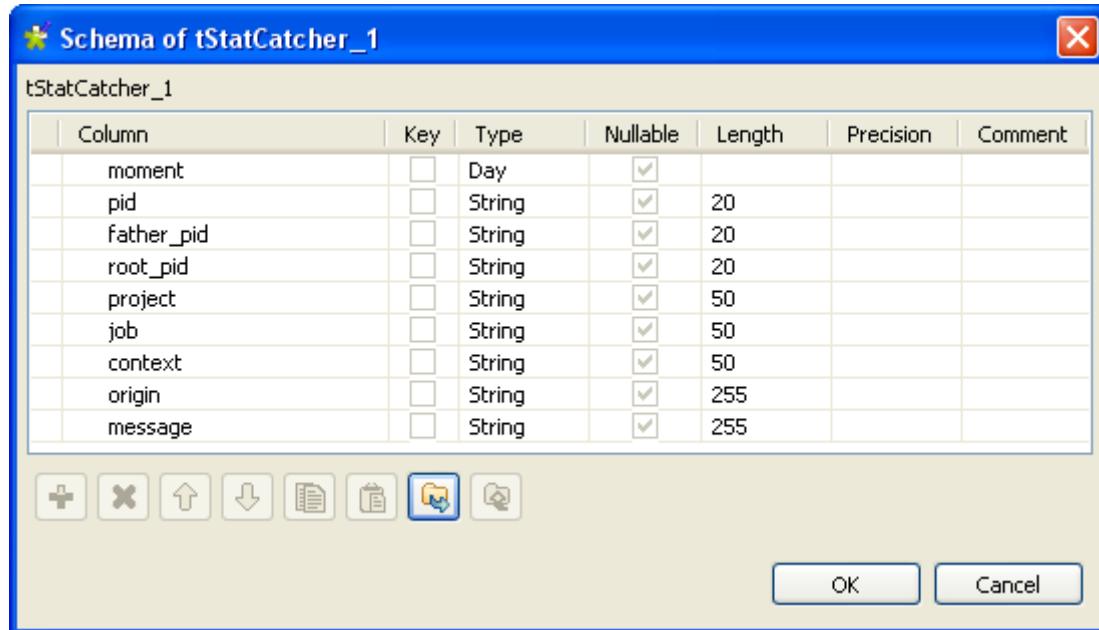
The screenshot shows the 'Schema' and 'Functions' tabs of the properties view for 'tRowGenerator_1'. The 'Schema' tab displays a table with three columns: ID_Owner (Key, int), Name_Customer (String), and ID_Insurance (String). The 'Functions' tab is empty. Below the tabs, there are icons for adding, deleting, and modifying rows, along with a 'Columns' button and a 'Number of Rows for RowGenerator' input field set to 100.

- The number of rows can be restricted to 100.
- Click on the **Main** tab of the Properties view.

The screenshot shows the 'Main' tab of the properties view for 'tRowGenerator_1'. It includes fields for Unique Name (set to 'tRowGenerator_1'), Family (set to 'Misc'), and checkboxes for 'Activate' and 'tStatCatcher Statistics' (both checked).

- And check the **tStatCatcher Statistics** box to enable the statistics fetching operation.
- Then, define the output component's properties. In the **tFileOutputDelimited** Properties panel, browse to the output file or enter a name for the output file to be created. Define the delimiters, such as semi-colon, and the encoding.

- Click on **Edit schema** and make sure the schema is recollected from the input schema. If need be, click on **Sync Columns**.
- Then click on the **Main** tab of the **Properties** view, and check here as well the **tStatCatcher Statistics** box to enable the processing data gathering.
- In the secondary job, double-click on the **tStatCatcher** component. Note that the Properties are provided for information only as the schema representing the processing data to be gathered and aggregated in statistics, is defined and read-only.



- Define then the **tLogRow** to set the delimiter to be displayed on the console.
- Eventually, press **F6** to run the job and display the job result.

```

Starting job StatsCatch at 15:10 23/02/2007.
2007-02-23 15:10:30|3656|StatsCatch|Default||begin
2007-02-23
15:10:30|3656|StatsCatch|Default|tFileOutputDelimited_1||begin
2007-02-23 15:10:30|3656|StatsCatch|Default|tRowGenerator_1||begin
2007-02-23 15:10:30|3656|StatsCatch|Default|tRowGenerator_1||end
2007-02-23 15:10:30|3656|StatsCatch|Default|tRowGenerator_1||0.0 seconds
2007-02-23 15:10:30|3656|StatsCatch|Default|tFileOutputDelimited_1||end
2007-02-23 15:10:30|3656|StatsCatch|Default|tFileOutputDelimited_1||0.0
seconds
2007-02-23 15:10:30|3656|StatsCatch|Default||end
2007-02-23 15:10:30|3656|StatsCatch|Default||0.0 seconds
Job StatsCatch ended at 15:10 23/02/2007. [exit code=0]

```

The log shows the Begin and End information for the job itself and for each of the component used in the job.

Components

tSugarCRMInput

tSugarCRMInput

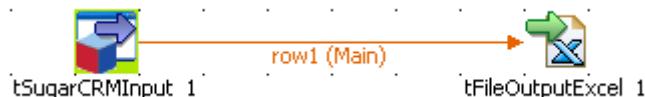


tSugarCRMInput Properties

Component family	Business	
Function	Connects to a module of a Sugar CRM database via the relevant webservice.	
Purpose	Allows to extract data from a SugarCRM DB based on a query.	
Properties	<i>SugarCRM Webservice URL</i>	Type in the webservice URL to connect to the SugarCRM DB.
	<i>Module</i>	Select the relevant module in the list
	<i>Username and Password</i>	Type in the Webservice user authentication data.
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository. Click Edit Schema to make changes to the schema. Note that if you make changes, the schema automatically becomes built-in. In this component the schema is related to the Module selected.
	<i>Query condition</i>	Type in the query to select the data to be extracted. Example: account_name= 'Talend'
Usage	Usually used as a Start component. An output component is required.	
Limitation	n/a	

Scenario: Extracting account data from SugarCRM

This scenario describes a two-component job which aims at extracting account information from a SugarCRM database to an Excel output file.



- Click and drop a **tSugarCRMInput** and a **tFileOutputExcel** component.
- Connect the input component to the output component using a main row link.
- On the **tSugarCRMInput** Properties panel, fill in the connection information in the SugarCRM Web Service URL as well as the **Username** and **Password** fields

- Then select the **Module** in the list of modules offered. In this example, *Accounts* is selected.

The screenshot shows the configuration window for the tSugarCRMInput_1 component. It includes fields for SugarCRM WebService URL (set to "http://localhost/sugar/soap.php"), Module (set to "Accounts"), Username ("admin"), Password ("root"), Schema Type (set to "Built-In"), and Query Condition ("billing_address_city='Sunnyvale'").

- The **Schema** is then automatically set according to the module selected. But you can change it and remove the columns that you don't require in the output.
- In the **Query Condition** field, type in the query you want to extract from the CRM. In this example: "billing_address_city=' Sunnyvale'"
- Then select the **tFileOutputExcel** component, .

The screenshot shows the configuration window for the tFileOutputExcel_1 component. It includes fields for File Name ("C:/Output/billing_city.xls"), Sheet name ("accounts"), Include header (checked), Schema Type (set to "Built-In"), Sync columns, and Encoding Type (set to "ISO-8859-15").

- Set the destination file name as well as the **Sheet** name and check the **Include header** box.
- Save the job and press F6 to run it.

The screenshot shows an Excel spreadsheet titled "accounts" containing data from the SugarCRM database. The columns are labeled A through F. The data includes various company names, their industries, and addresses, such as "T-Cat Media Group Inc", "Environmental", "777 West", and "Sunnyvale". The spreadsheet has 11 rows, with rows 8 through 10 being empty.

The filtered data is output in the defined spreadsheet of the specified Excel type file.

Components

tSugarCRMOutput



tSugarCRMOutput

tSugarCRMOutput properties

Component family	Business	
Function	Writes in a module of a Sugar CRM database via the relevant webservice.	
Purpose	Allows to write data into a SugarCRM DB.	
Properties	<i>SugarCRM Webservice URL</i>	Type in the webservice URL to connect to the SugarCRM DB.
	<i>Module</i>	Select the relevant module in the list
	<i>Username and Password</i>	Type in the Webservice user authentication data.
	<i>Action</i>	Insert or Update the data in the SugarCRM module.
	<i>Schema type and Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository. Click Edit Schema to make changes to the schema. Note that if you make changes, the schema automatically becomes built-in. Click Sync columns to retrieve the schema from the previous component connected in the job.
Usage	Used as an output component. An Input component is required.	
Limitation	n/a	

Related Scenario

No scenario is available yet for this component.

tSybaseInput



tSybaseInput properties

The properties of the generic component, **tDBInput**, apply to the **tSybaseInput** component, therefore the properties are not detailed again here.

For detailed information about generic **tDBInput** properties, see *tDBInput properties on page 136*.

Related scenarios

Related topic in **tDBInput** scenarios:

- *Scenario 1: Displaying selected data from DB table on page 137*
- *Scenario 2: Using StoreSQLQuery variable on page 138*

Related topic in **tContextLoad** Scenario: *Dynamic context use in MySQL DB insert on page 123*.

tSybaseOutput



tSybaseOutput properties

The properties of the generic component, **tDBOutput**, apply to the **tSybaseOutput** component, therefore the properties are not detailed again here.

For detailed information about generic **tDBOutput** properties, see *DBOutput properties on page 140*.

Related scenarios

For use cases in relation with **tSybaseOutput**, see:

- **tDBOutput Scenario: Displaying DB output on page 142**
- **tMySQLOutput Scenario: Adding new column and altering data on page 261.**



tSybaseBulkExec

tSybaseBulkExec properties

Component family	Databases	
Function	Executes the Insert action on the data provided.	
Purpose	As a dedicated component, it allows gains in performance during Insert operations to a Sybase database.	
Properties	<i>Property type</i>	Either Built-in or Repository.
		Built-in: No existing property data is to be retrieved
		Repository: Select the Repository file where Properties are stored. The following fields are pre-filled in using fetched data.
	<i>Server</i>	Database server IP address
	<i>Database</i>	Name of the database
	<i>Username and Password</i>	DB user authentication data.
	<i>Table</i>	Name of the table to be written. Note that only one table can be written at a time and that the table must exist for the insert operation to succeed.
	<i>File Name</i>	Name of the file to be processed. Related topic: <i>Defining the context variables on page 96</i>
	Fields terminated by	Character, string or regular expression to separate fields.
	<i>Encoding</i>	Select the encoding from the list or select Custom and define it manually. This field is compulsory for DB data handling.
	<i>Output to</i>	Console: Loading information Global variable: Returned values from log files.
Usage	This component is mainly used when no particular transformation is required on the data to be loaded onto the database.	
Limitation	As opposed to the Oracle dedicated bulk component, no action on data is possible using this Sybase dedicated component	

Related scenarios

For tSybaseBulkExec topic, see:

- **tMysqlOutputBulkExec** *Scenario: Inserting transformed data in MySQL database on page 265*
- **tOracleBulkExec** *Scenario: Truncating and inserting file data into Oracle DB on page 284.*

tSybaseRow



tSybaseRow properties

The properties of the generic component, **tDBSQLRow**, apply to the **tSybaseRow** component, therefore the properties are not detailed again here.

For detailed information about generic **tDBSQLRow** properties, see *tDBSQLRow properties on page 144*.

Related scenarios

For **tSybaseRow** related topics, see:

- **tDBSQLRow Scenario 1: Resetting a DB auto-increment on page 145**
- **tMySQLRow Scenario: Removing and regenerating a MySQL table index on page 275.**

Components

tSystem

tSystem



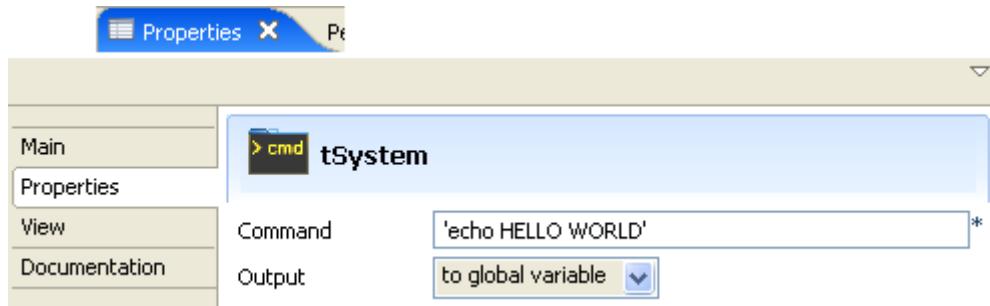
tSystem properties

Component family	System	
Function	tSystem executes one or more system commands.	
Purpose	tSystem can call other processing commands, already up and running in a larger job.	
Properties	Command	Enter the system command. Note that the syntax is not checked.
	Output	Select the type of output for the processed data to be passed onto.
		to console: standard output passes on data to be viewed in the Log view.
		to global variable: data is put in output variable linked to tsystem component.
Usage	This component can typically be used for companies which have already implemented other applications that they want to integrate into their processing flow through Talend.	
Limitation	n/a	

Scenario: Echo ‘Hello World!’

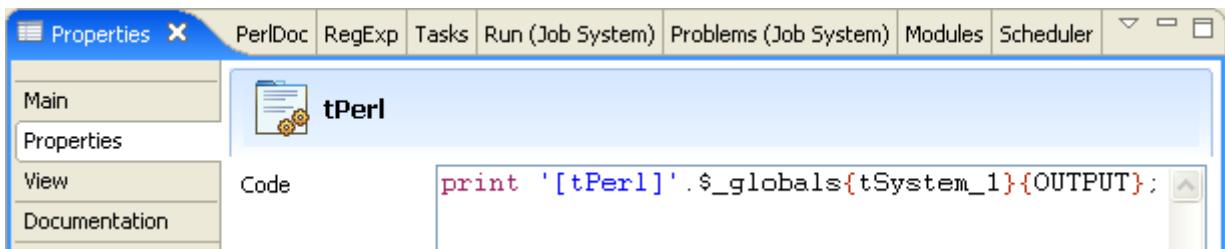
This scenario is a two-component job showing a message in the Log.

- Click and drop a **tSystem** and a **tPerl** component onto the workspace.
- Right-click on **tSystem**, and pull a **ThenRun** link between the two components. When executing the job, the first component will trigger before the second one.
- Click on the **tSystem** and select the **Properties** tab:

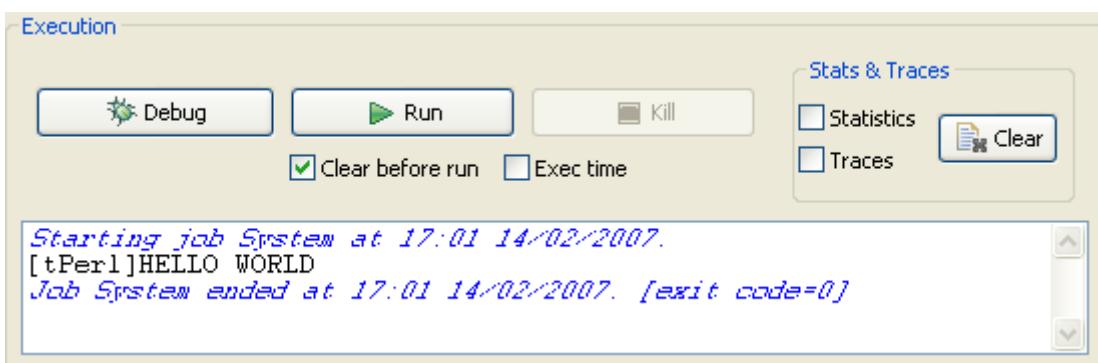


- Enter the echo command and string “Hello World!” to be displayed

- Select **To a global variable** option as **Output** to include the command output value into
- Then select the **tPerl** component



- Enter a Perl command to display the **tSystem** output variable in the console.
- Go to the **Run Job** tab and execute the job.



The job executes an echo command and shows the output in the Log using a Print command in the tPerl component.

tUniqRow

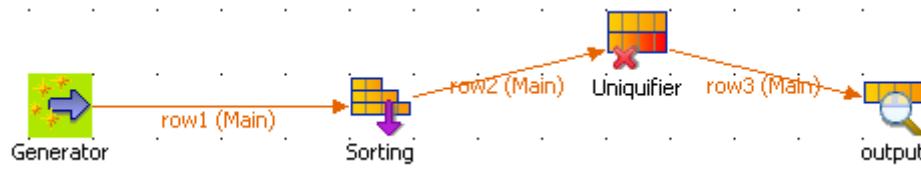


tUniqRow properties

Component family	Data Quality	
Function	Compares entries and removes the first encountered duplicate from the input flow.	
Purpose	Ensures data quality of input or output flow in a job.	
Properties	Schema type and <i>Edit Schema</i>	<p>A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository.</p> <p>Click Edit Schema to make changes to the schema. Note that if you make changes, the schema automatically becomes built-in.</p> <p>Click Sync columns to retrieve the schema from the previous component connected in the job.</p> <p> If you want the deduplication to be carried out on particular columns, define them on the schema.</p>
	Built-in: The schema will be created and stored locally for this component only. Related topic: <i>Setting a built-in schema on page 51</i>	
	Repository: The schema already exists and is stored in the Repository, hence can be reused in various projects and job flowcharts. Related topic: <i>Setting a repository schema on page 52</i>	
	Case sensitive	Check the box to consider the lower or upper case.
Usage	This component handles flow of data therefore it requires input and output, hence is defined as an intermediary step.	
Limitation	n/a	

Scenario: Unduplicating entries

Based on the **tSortRow** job, the **tUniqRow** component is added to the job in order to unify the entries in the output flow. In fact, as the input data is randomly created, duplication cannot be avoided.



- On the **Properties** tab panel of the **tUniqRow** component, click **Edit Schema...** to set the **Key on Names** field to uniquify the output flow on this criteria.
- Check the **Case Sensitive** box to differentiate lower case and upper case.
- Press **F6** to run the job again. The console displays the sorted and unique results

```

Starting job UnduplicateJob at 11:02 03/01/2007.
5|3|Mickael|93
2|1|Pierrick|92
8|1|Steffie|89
4|3|Fabrice|75
3|4|Bertrand|67
7|1|Matthew|28
Job UnduplicateJob ended at 11:02 03/01/2007. [exit code=0]

```

tWarn

Both **tDie** and **tWarn** components are closely related to the **tLogCatcher** component. They generally make sense when used alongside a **tLogCatcher** in order for the log data collected to be encapsulated and passed on to the output defined.

tWarn properties

Component family	Log/Error	
Function	Provides a priority-rated message to the next component	
Purpose	Triggers a warning often caught by the tLogCatcher component for exhaustive log.	
	<i>Warn message</i>	Type in your warning message
	<i>Code</i>	Define the code level
	<i>Priority</i>	Enter the priority level as an integer
Usage	Cannot be used as a start component. If an output component is connected to it, an input component should be preceding it.	
Limitation	n/a	

Related scenarios

For uses cases in relation with **tWarn**, see **tLogCatcher** scenarios:

- *Scenario 1: warning & log on entries on page 226*
- *Scenario 2: log & kill a job on page 228*



tWebServiceInput

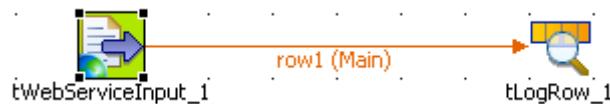
tWebServiceInput properties

Component family	Internet	
Function	Calls the defined method from the invoked webservice, and returns the class as defined, based on the given parameters.	
Purpose	Invokes a Method through a webservice and for the described purpose	
Properties	<i>Schema type</i> and <i>Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields that will be processed and passed on to the next component. The schema is either built-in or remote in the Repository. Click Edit Schema to make changes to the schema. Note that if you make changes, the schema automatically becomes built-in. Click Sync columns to retrieve the schema from the previous component connected in the job.
	<i>End Point URI</i>	Resource identifier of the web service
	<i>WSDL</i>	Description of Web service bindings and configuration
Java only field	<i>SOAPAction URI</i>	SOAP standard end point if required
	<i>Method Name</i>	Enter the exact name of the Method to be invoked. The Method name MUST match the corresponding method described in the Web Service. The Method name is also case-sensitive.
Java only field	<i>Return class</i>	Select the type of data to be returned by the method. Make sure it fully matches the one defined in the method. Note: For .Net services, use the returned class: <i>org.apache.axis.types.Schema.class</i>
	<i>Parameters</i>	Enter the parameters expected and the sought values to be returned. Make sure that the parameters entered fully match the names and the case of the parameters described in the method.
Usage	This component is generally used as a Start component . It requires to be linked to an output component.	
Limitation	n/a	

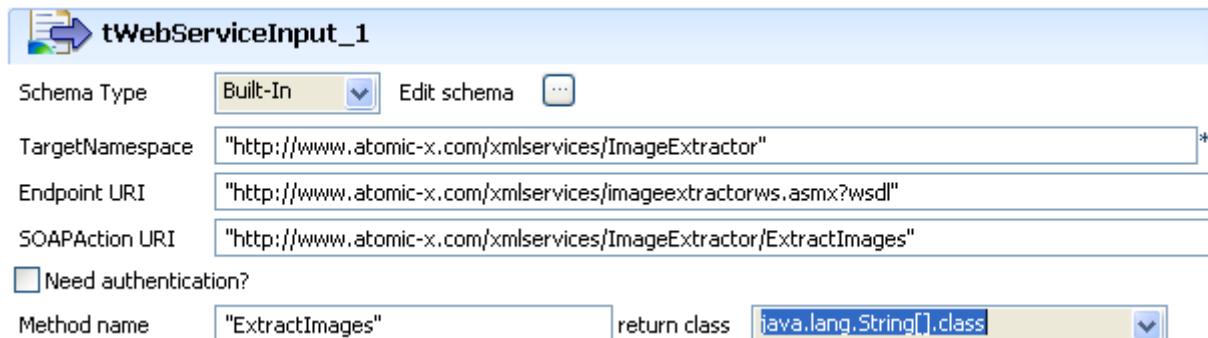
Scenario: Extracting images through a Webservice

This scenario describes a two-component job aiming at using a Webservice method and display the output on the standard view.

The method takes a full url as an input string and returns a string array of images from a given web page.



- Click and drop a **tWebServiceInput** component and a **tLogRow** component.
- On the **Properties** view of the **tWebServiceInput** component, define the WSDL specifications, such as **End Point URI**, **WSDL** and **SOAPAction URI** where required.
- If the Web service you invoked requires authentication details, check the box and provide the relevant authentication information.



- In the **Method Name** field, type in the method name as defined in the Web Service description. The name and the case of the method entered must match exactly the corresponding Web service method.
- Then select the **return class** corresponding to the expected value type.
- In the Parameters area, click the plus (+) button to add a line to the table.
- Then type in the exact parameters' name as expected by the method.

Parameters

name	value	class
"aUrl"	"http://www.yahoo.com"	java.lang.String.class



- In the **Value** column, type in the URL of the Website, the images are to be extracted from.
- The **Class** column is automatically filled in with the return class type selected earlier.
- Link the tWebServiceInput component to the standard output component, **tLogRow**.
- Then press **F6** to run the job.

```

Starting job WebService at 16:30 06/06/2007.
</img>
</img>
</img>
</img>
</img>
</img>
</img>
</img>
</img>
Job WebService ended at 16:30 06/06/2007. [exit code=0]

```

All images extracted from the given website are returned as a list of URLs on the **Run Job** view.

tXSDValidator



tDTDValidator Properties

Component family	XML	
Function	Validates the XML input file against a XSD file and sends the validation log to the defined output.	
Purpose	Helps at controlling data and structure quality of the file to be processed	
Properties	<i>Schema type</i> and <i>Edit Schema</i>	A schema is a row description, i.e., it defines the number of fields to be processed and passed on to the next component. The schema is either built-in or remotely stored in the Repository but in this case, the schema is read-only. It contains standard information regarding the file validation.
	<i>XSD file</i>	Filepath to the reference DTD file.
	<i>XML file</i>	Filepath to the XML file to be validated.
	<i>If XML is valid, display</i> <i>If XML is not valid detected, display</i>	Type in a message to be displayed in the Run Job console based on the result of the comparison.
	<i>Print to console</i>	Check the box to display the validation message
Usage	This component can be used as standalone component but it is usually linked to an output component to gather the log data.	
Limitation	n/a	

Related scenario

For related tXSDValidator use cases, see *Scenario: Validating xml files on page 153*.

tXSLT



tXSLT

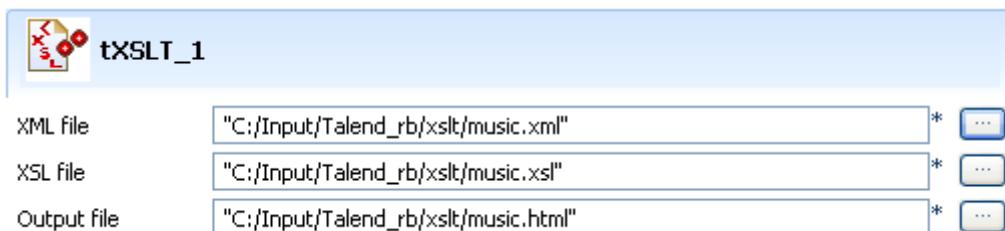
Component family	XML	
Function	Refers to an XSL stylesheet, to transform an XML source file into a defined output file.	
Purpose	Helps to transform data structure to another structure.	
Properties	XML file	Filepath to the XML file to be validated.
	XSL file	Filepath to the reference XSL transformation file.
	Output file	Filepath to the output file. If the file doesn't exist, it will be created. The output file can be any structured or unstructured file such as html, xml, txt or also pdf or edifact depending on your xsl.
Usage	This component can be used as standalone component.	
Limitation	n/a	

Scenario: Transforming XML to html using an XSL stylesheet

This scenario describes a job applying an xsl stylesheet on an xml file and outputs an html file.



- Drag and drop the **tXSLT** component.
- On the **Properties** view of the component, set the XML file to be transformed. In this use case, a list of MP3 titles and their corresponding artist



- Then set the filepath to the relevant XSL file in order to apply the wanted transformation.

- In this use case, we want to add an image and apply an stylesheet to create a table in HTML.

```

1  <?xml version='1.0' encoding="ISO-8859-1" ?>
2  <xsl:stylesheet version="1.0" xmlns:xsl=
3      "http://www.w3.org/1999/XSL/Transform">
4  <xsl:template match="/">
5  <html>
6  <body>
7      <IMG SRC="http://www.talend.com/img/logo-talend-fast.jpg"
8          ALT="Talend image" TITLE="Talend Open studio">
9      </IMG>
10     <table border="1" cellspacing="0" cellpadding="3">
11         <tr bgcolor="#b5dc10">
12             <td>Title</td>
13             <td>Artist</td>
14         </tr>
15         <xsl:for-each select="compilation/mp3">
16             <tr>
17                 <td><xsl:value-of select="Title"/></td>
18                 <td><xsl:value-of select="Artist"/></td>
19             </tr>
20         </xsl:for-each>
21     </table>
22     </body>
23     </html>
24     </xsl:template>
25     </xsl:stylesheet>

```

- Eventually set the output filepath to the HTML file.
- Save the job and press F6 to run it. Open the Html file in a browser to check the output.



Title	Artist
Thriller	Michael Jackson
OK Computer	Radiohead
What a wonderful world	Louis Armstrong
Jo le taxi	Vanessa Paradis
Tears in heaven	Eric Clapton

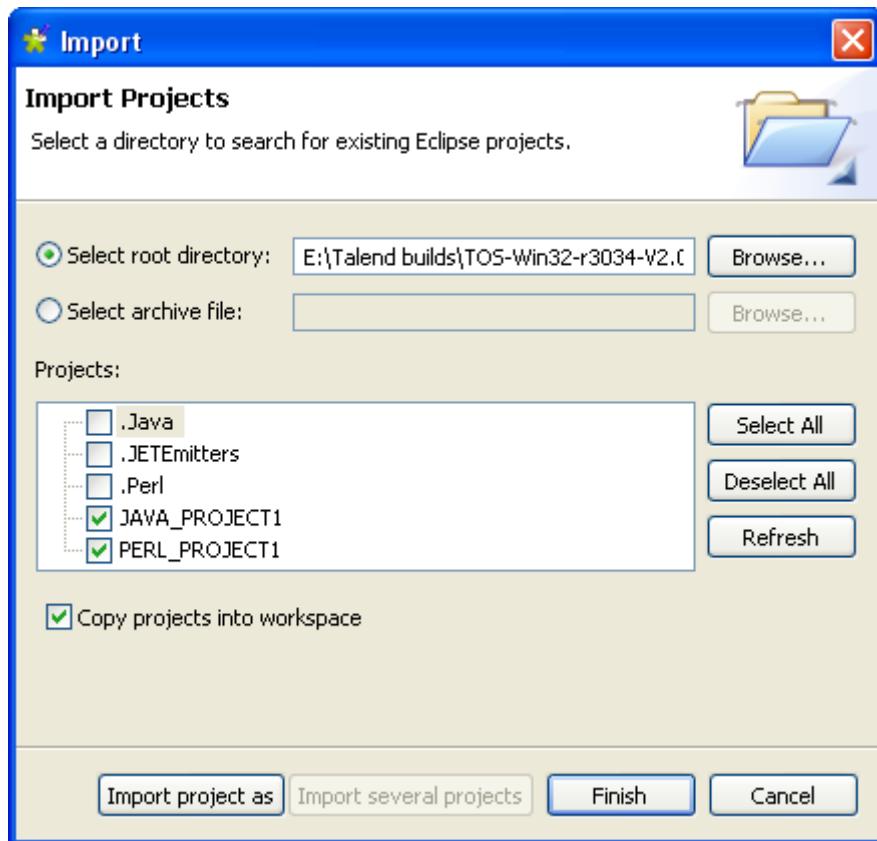
—Managing projects—

Managing projects

Importing projects

On the login window, click **Import projects** to open the Import wizard. Click on **Open several projects** if you intend to import more than one project at once.

Or in **JasperETL** main window, click on the **Import projects** button on the toolbar.



Click **Browse...** to select the **Workspace** directory or the specific project folder. By default, the workspace in selection is the current release's one. Browse up to reach the previous release workspace directory containing the projects to import.

Check the **Copy projects into workspace** option box to make a copy of the project instead of moving them. If you want to remove the original project directories from the previous **JasperETL** release workspace directory, uncheck this box. But we strongly recommend you to keep it selected for backup purpose.

Managing projects

Importing Job samples (Demos)

Select in the list the projects to import and click **Finish** to validate.

In the login window, the projects imported now display in the **Project** list, select it from the list

Or from **JasperETL** workspace, click **File > Switch projects...** to get back to the login window.



Click **OK** to launch **JasperETL**.

Note: A generation initialization window might come up when launching the application. Wait until the initialization is complete.

If, instead of importing the whole project, you'd rather select individual items from your projects,

Importing Job samples (Demos)

As for the import of projects from previous releases of **JasperETL**, you can import in your workspace the Demos project folder, that includes numerous samples of job.

On the Login window of **JasperETL**, click on the Demos button.



Select your preferred language between Perl and Java.

The Job samples covering all needs are automatically imported into your workspace and made available in the Repository.

A message displays to confirm the import operation successfully.

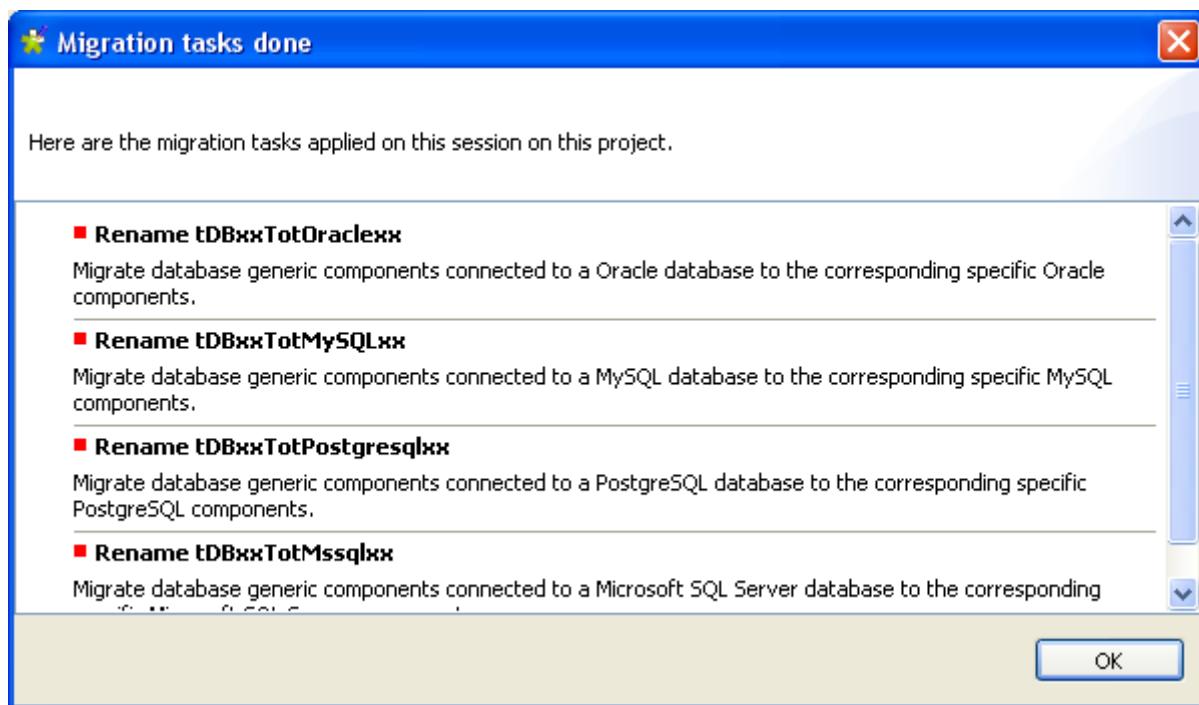
The Demos project displays in the list of **existing Projects** in the Login window.

You can use these samples to get started with your own job design.

Migration tasks

Migration tasks are performed to ensure the compatibility of the projects you created with a previous version of **JasperETL** with the current release.

As some changes might become visible to the user, we thought we'd share these update tasks with you through an information window.



Some changes that affect the usage of **JasperETL** include, for example:

- tDBInput used with a MySQL database becomes a specific tDBMysqlInput component the aspect of which is automatically changed in the job where it is used.
- tUniqRow used to be based on the Input schema keys, whereas the current tUniqRow allows the user to select the column to base the unicity on.

This information window pops up when you launch the project you imported (created) in a previous version of **JasperETL**.

It lists and provides a short description of the tasks which were successfully performed so that you can smoothly roll your projects.

Importing Repository items

You can now import items from previous versions of **JasperETL** or a different project of your current version.

The items you can possibly import are multiple:

- Business Models
- Jobs Designs
- Routines
- Documentation
- Metadata

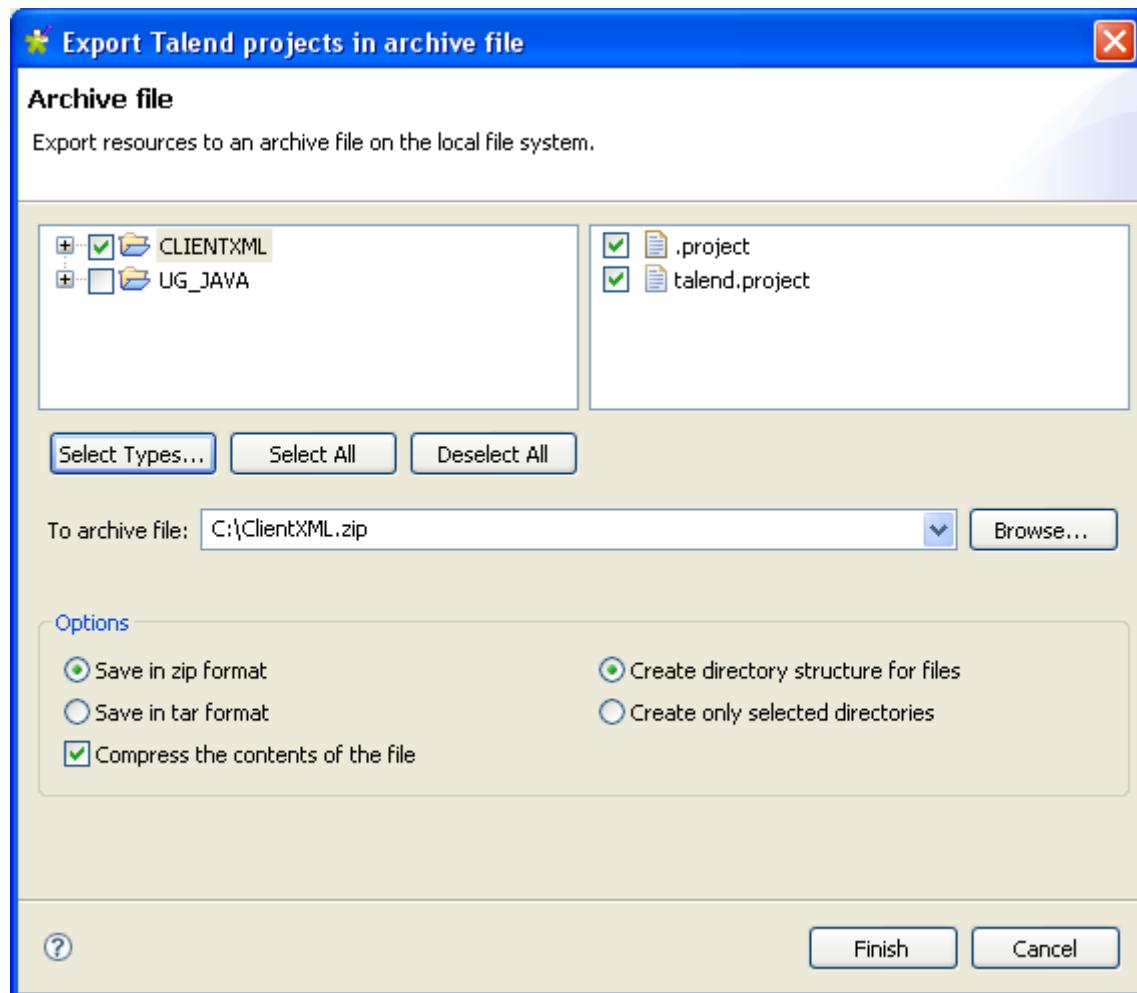
Right-click on the node of the Repository, you want to import an item to. Then select **Import items** function on the pop-up menu. Follow the wizard to complete the import process.

Note: Note that if the item is already present in the current workspace, the item will not be suggested.

Exporting projects

On **JasperETL** workspace, the toolbar allows you to export the current project.

Click on the **Export project** button of the toolbar, at the top of **JasperETL** main window.



The **Export** window opens up on the workspace directory showing the projects you can export to an archive file.

You can select multiple projects or only select parts of the project through the **Select Types** link, if need be (for advanced users).

Type in the name of the archive or browse to the archive file if it already exists.

In the **Option** area, select the compression format and the structure type you prefer.

Click **Finish** to validate.

A	
Activate/Deactivate	94
Alias	159
Appearance	35
Assignment table	37
B	
Breakpoint	103
Business Model	15, 27, 28
Creating	28
Opening	28
Business Modeler	29
Business Models	27
C	
Code	15
Code Viewer	20, 21
Component	42, 115
Activate	49
External	25
Start	53
Connection	
Iterate	47
Link	48
Lookup	45
Main	45
Output	45
Row	45
Context	100, 102, 300
D	
Data quality	
tAddCRCRow	129
tFuzzyMatch	221
Database	
tDB2Input	133
tDB2Output	134
tDBInput	136
tDBOutput	140
tDBSQLRow	144
tMSSqlInput	251
tMSSqlOutput	252
tMSSqlRow	253
tMysqlBulkExec	269
tMysqlCommit	259

tMysqlConnection	254
tMysqlInput	260
tMysqlOutput	261
tMysqlOutputBulk	263
tMysqlOutputBulkExec	272
tMysqlRollback	274
tMysqlRow	275
Debug mode	104
Delimited	59
Documentation	16
 E	
Edit Schema	52
ELT	
tELTMysqlInput	156
tELTMysqlMap	157
tELTMysqlOutput	167
tELTOraacleInput	170
tELTOracleMap	171
tELTOraacleOutput	176
Explicit Join	159
Exporting	
Projects	346
 F	
File	
tFileCompare	179
tFileCopy	182
tFileDelete	184
tFileInputDelimited	188
tFileInputMail	191
tFileInputPositional	193
tFileInputRegex	197
tFileInputXML	201
tFileList	204
tFileOutputExcel	207
tFileOutputLDIF	208
tFileOutputXML	211
tFileUnarchive	213
File XML	
Loop limit	74
FilePositional	64
FileRegex	67

G	
Generation language	13
Graphical workspace	17, 27
Grid	34
H	
Hash key	83
I	
Importing	
Items	346
Inner join	86
Inner Join Reject	86
Internet	
tFileFetch	186
tFTP	218
tSendMail	306
Item	
Importing	346
Iterate	47
J	
Job	
Creating	41
Opening/Creating	39
Running	101, 102
Job Designer	41
Panels	43
Job Designs	15, 39, 40
Job script	
Exporting	104
Join	
Explicit	159
K	
Key	83
L	
LDIFfile	68
Left Outer Join	165
Link	48
Log&Error	
tDie	152
tLogCatcher	226
Log/error	

tLogRow	230
tStatCatcher	321
Logs	18
Lookup	46
M	
Main properties	38
Main row	45
Mapper	48
Metadata	16, 54
DB Connection schema	55
FileDelimited schema	59
FileLDIF Schema	68
FilePositional schema	64
FileRegex schema	67
FileXML schema	71
Misc	
tContextLoad	122
tFor	215
tMsgBox	277
tRowGenerator	295
Model	
Arranging	33, 34
Assigning	37
Commenting	33
Copying	38
Deleting	38
Moving	38
Saving	38
Modeler	29
Multiple Input/Output	48
O	
Object	30
Outline	20, 21
Output	46
P	
Palette	17, 29, 30, 33, 41, 42
Components	115
Note	33
Note attachment	33
Select	33
Zoom	33
Primary Key	83

Processing	
tAggregateRow	117
tDenormalize	147
tMap	231
tNormalize	279
tPerl	289
tSortRow	310
tUniqRow	334
Project	
Exporting	346
Properties	17, 18, 29, 34
Comment	51
Main	38
Properties	51
Rulers & Grid	34
View	50
Q	
Query	
SQLBuilder	75
R	
Recycle bin	16, 38
Refresh	12
Regular Expressions	68
Relationship	31
bidirectional	32
directional	32
Simple	32
Repository	12, 14, 27, 39
Routine	15
Row	45
Main	45
Rulers	34
Run Job	18, 101, 102
S	
Scheduler	19
Schema	
Built-in	53
Shape	30
SQLBuilder	75
Start	53
Statistics	102
StoreSQLQuery	98, 138

Sync columns	52
System	
tRunJob	300
tSystem	332

T

Table	
Alias	159
Technical name	13
tFlowMeterCatcher	106
tLogCatcher	106
tMap	48
Traces	103
Trigger	
Run After	47
Run Before	47
Run if	48
Run if Error	48
Run if OK	48
ThenRun	47
tStatCatcher	106

V

Variable	96, 300
StoreSQLQuery	98, 138
Views	
Moving	43

X

XML	
tDTDValidator	153
XMLFile	71
Xpath	73