Scheduling and executing a Talend Job on the Google Kubernetes Engine... 55
Prerequisites to this example................................................................. 55
Publishing your Job as a Docker image............................................... 55
Configuring Google Kubernetes Engine........................................... 59
Scheduling the Job................................................................................. 60
Copyright

Adapted for 7.3.1. Supersedes previous releases.

Copyright © 2020 Talend. All rights reserved.

The content of this document is correct at the time of publication.

However, more recent updates may be available in the online version that can be found on Talend Help Center.

Notices

Talend is a trademark of Talend, Inc.

All brands, product names, company names, trademarks and service marks are the properties of their respective owners.

End User License Agreement

The software described in this documentation is provided under Talend's End User Software and Subscription Agreement ('Agreement') for commercial products. By using the software, you are considered to have fully understood and unconditionally accepted all the terms and conditions of the Agreement.

tMap Job example

To illustrate the way Talend Studio operates, find below a real-life example scenario. In this scenario, we will load a MySQL table with a file, that gets transformed on the fly. Then in a further step, we will select the data to be loaded using a dynamic filter.

Before actually starting the Job, let's inspect the input data and the expected output data.

Input data

Our input file, the data of which will be loaded into the database table, lists clients from all over the State of California.

The file structure usually called Schema in Talend Studio includes the following columns:

- First name
- Last name
- Address
- City

Output data

We want to load into the database, California clients living in a couple of Counties only: Orange and Los Angeles counties.

The table structure is slightly different, therefore the data expected to be loaded into the DB table should have the following structure:

- **Key** (key, Type: Integer)
- **Name** (Type: String, max. length: 40)
- **Address** (Type: String, max. length: 40)
- **County** (Type: String, max. length: 40)

In order to load this table, we will need to use the following mapping process:

The **Key** column is fed with an auto-incremented integer.

The **Name** column is filled out with a concatenation of first and last names.

The **Address** column data comes from the equivalent Address column of the input file, but supports a upper-case transformation before the loading.

The **County** column is fed with the name of the County where the city is located using a reference file which will help filtering Orange and Los Angeles counties' cities.

Reference data

As only Orange and Los Angeles counties data should be loaded into the database, we need to map cities of California with their respective county, in order to filter only Orange and Los Angeles ones.

To do so, we will use a reference file, listing cities that are located in Orange and Los Angeles counties such as:
tMap Job example

<table>
<thead>
<tr>
<th>City</th>
<th>County</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agoura Hills</td>
<td>Los Angeles</td>
</tr>
<tr>
<td>Alhambra</td>
<td>Los Angeles</td>
</tr>
<tr>
<td>Aliso Viejo</td>
<td>Orange</td>
</tr>
<tr>
<td>Anaheim</td>
<td>Orange</td>
</tr>
<tr>
<td>Arcadia</td>
<td>Los Angeles</td>
</tr>
</tbody>
</table>

The reference file in this Job is named LosAngelesandOrangeCounties.txt.

Translating the scenario into a Job

In order to implement this scenario, let's break down the Job into four steps:

1. Creation of the Job, configuration of the input file parameters, and reading of the input file,
2. Mapping of data and transformations,
3. Definition of the reference file parameters, relevant mapping using the tMap component, and selection of inner join mode,
4. Redirection of the output into a MySQL table.

Step 1: Job creation, input definition, file reading

Procedure

1. Launch Talend Studio, and create a local project or import the demo project if you are launching Talend Studio for the first time.
2. To create the Job, right-click Job Designs in the Repository tree view and select Create Job.
3. In the dialog box displaying then, only the first field (Name) is required. Type in California1 and click Finish.

An empty Job then opens on the main window and the Palette of technical components (by default, to the right of the Studio) comes up showing a dozen of component families such as: Databases, Files, Internet, Data Quality and so on, hundreds of components are already available.

4. To read the file California_Clients, let's use the tFileInputDelimited component. This component can be found in the File > Input group of the Palette. Click this component then click to the left of the design workspace to place it on the design area.

5. Let's define now the reading properties for this component: File path, column delimiter, encoding... To do so, let's use the Metadata Manager. This tool offers numerous wizards that will help us to configure parameters and allow us to store these properties for a one-click re-use in all future Jobs we may need.

6. As our input file is a delimited flat file, let's select File Delimited on the right-click list of the Metadata folder in the Repository tree view. Then select Create file delimited.

A wizard dedicated to delimited file thus displays:

- At Step 1, only the Name field is required: simply type in California_clients and go to the next Step.
• At Step 2, select the input file (California_Clients.csv) via the Browse... button. Immediately an extract of the file shows on the Preview, at the bottom of the screen so that you can check its content. Click Next.

• At Step 3, we will define the file parameters: file encoding, line and column delimiters... As our input file is pretty standard, most default values are fine. The first line of our file is a header containing column names. To retrieve automatically these names, click Set heading row as column names then click Refresh Preview. And click Next to the last step.

• At Step 4, each column of the file is to be set. The wizard includes algorithms which guess types and length of the column based on the file first data rows. The suggested data description (called schema in Talend Studio) can be modified at any time. In this particular scenario, they can be used as is.

There you go, the California_clients metadata is complete!

We can now use it in our input component. Select the tFileInputDelimited you had dropped on the design workspace earlier, and select the Component view at the bottom of the window.

7. Select the vertical tab Basic settings. In this tab, you’ll find all technical properties required to let the component work. Rather than setting each one of these properties, let’s use the Metadata entry we just defined.

8. Select Repository as Property type in the list. A new field shows: Repository, click “...” button and select the relevant Metadata entry on the list: California_clients.

You can notice now that all parameters get automatically filled out.

At this stage, we will terminate our flow by simply sending the data read from this input file onto the standard output (StdOut).

9. To do so, add a tLogRow component (from the Logs & Errors group). To link both components, right-click the input component and select Row > Main. Then click the output component: tLogRow.

10. This Job is now ready to be executed. To run it, select the Run tab on the bottom panel.
11. Enable the statistics by selecting the Statistics check box in the Advanced Settings vertical tab of the Run view, then run the Job by clicking Run in the Basic Run tab.

The content of the input file display thus onto the console.

**Step 2: Mapping and transformations**

We will now enrich our Job to include on-the-fly transformations. To implement these transformation, we need to add a tMap component to our Job.

This component is multiple and can handle:

- multiple inputs and outputs
- search for reference (simple, cartesian product, first, last match...)
- join (inner, outer)
- transformations
- rejections
- and more...

**Procedure**

1. Remove the link that binds together the job’s two components via a right-click the link, then Delete option. Then place the tMap of the Processing component group in between before linking the input component to the tMap as we did it previously.
2. Eventually to link the tMap to the standard output, right-click the tMap component, select Row > *New Output* (Main) and click the tLogRow component. Type in `out1` in the dialog box to implement the link. Logically, a message box shows up (for the back-propagation of schemas), ignore it by clicking on No.

3. Double-click the tMap to access its interface.

   To the left, you can see the schema (description) of your input file (row1). To the right, your output is for the time being still empty (out1).

4. Drop the Firstname and Lastname columns to the right, onto the Name column as shown on the screen below. Then drop the other columns Address and City to their respective line.

5. Carry out the following transformations on each column:
   - Change the Expression of the Name column to `row1.Firstname + " " + row1.LastName`. This concatenates the Firstname column with the Lastname column, in order for the columns to display together in one column.
   - Change the Expression of the Address column to `row1.Address.toUpperCase()`, which will thus change the address case to upper case.

6. Remove the Lastname column from the out1 table and increase the length of the remaining columns. To do so, go to the Schema Editor located at the bottom of the Map editor and proceed as follows:
   1. Select the column to be removed from the schema, and click the cross icon.
   2. Select the column of which you need increase the length size.
   3. Type in the length size you intend in the length column. In this example, change the length of every remaining column to 40.

   **Note:** As the first name and the last name of a client are concatenated, it is necessary to increase the length of the name columns in order to match the full name size. No transformation is made onto the City column.

7. Click OK to validate the changes and close the Map editor interface.

8. Run your Job.
Results

The addresses are displayed in upper case and the first names and last names are gathered together in the same column.

Step 3: Reference file definition, remapping, inner join mode selection

Procedure

1. Define the Metadata corresponding to the LosAngelesandOrangeCounties.txt file just the way we did it previously for California_clients file, using the wizard.

   At Step1 of the wizard, name this metadata entry: LA_Orange_cities.

2. Then drop this newly created metadata to the top of the design area to create automatically a reading component pointing to this metadata.

3. Then link this component to the tMap component.
4. Double-click again on the **tMap** component to open its interface. Note that the reference input table (**row2**) corresponding to the LA and Orange county file, shows to the left of the window, right under your main input (**row1**).

5. Now let’s define the join between the main flow and the reference flow.

   In this use case, the join is pretty basic to define as the **City** column is present in both files and the data match perfectly. But even though this was not the case, we could have carried out operations directly at this level to establish a link among the data (padding, case change...)

   To implement the join, drop the **City** column from your first input table onto the **City** column of your reference table. A violet link then displays, to materialize this join.

Now, we are able to use the **County** column from the reference table in the output table (**out1**).
6. Eventually, click the OK button to validate your changes, and run the new Job.

The following output should display on the console.

As you can notice, the last column is only filled out for Los Angeles and Orange counties’ cities. For all other lines, this column is empty. The reason for this is that by default, the tMap implements a left outer join mode. If you want to filter your data to only display lines for which a match is found by the tMap, then open again the tMap, click the tMap settings button and select the Inner Join in the Join Model list on the reference table (row2).

**Step 4: Output to a MySQL table**

Our Job works perfectly! To finalize it, let’s direct the output flow to a MySQL table.

**Procedure**

1. To do so, let’s first create the Metadata describing the connection to the MySQL database. Expand the Metadata > MySQL nodes in the Repository and double-click DemoMySQL (on the condition that you imported the Demo project properly) to open the Metadata wizard.

2. On Step2 of the wizard, type in the relevant connection parameters. Check the validity of this connection by clicking on the Check button. Eventually, validate your changes, by clicking on Finish.
3. Drop this metadata to the right of the design workspace, while maintaining the Ctrl key down, in order to create automatically a tMysqlOutput component.

4. Remove the tLogRow component from your Job.

5. Reconnect the out1 output flow from the tMap to the new component tMysqlOutput.

6. On the Basic Settings tab of this component:
   a) Type in LA_Orange_Clients in the Table field, in order to name your target table which will get created on the fly.
   b) Select the Drop table if exists and create option or on the Action on table field.
   c) Click Edit Schema and click the Reset DB type button (DB button on the tool bar) in order to fill out automatically the DB type if need be.

7. Run again the Job.

Results

The target table should be automatically created and filled with data in less a second!

In this scenario, we did use only four different components out of hundreds of components available in the Palette and grouped according to different categories (databases, Web service, FTP and so on)!

And more components, this time created by the community, are also available on the community site (talendforge.org).
Using Dynamic Schema to load data dynamically to database tables

This example describes how to leverage the Dynamic Schema feature to load data dynamically.

Warning: This documentation is relevant only if you are using a subscription-based Talend Studio.

The content of the following three flat files will be loaded to MySQL database tables named customer, employee and delivery respectively:

- **customer_details.csv**:
  ```
  first_name;last_name;order_date
  Reita;Leto;2012-05-15
  Larae;Gudroe;2011-08-07
  Clay;Hoa;2014-04-16
  Irma;Wolfgramm;2012-09-10
  Sylvia;Cousey;2013-02-25
  ```

- **employee_details.csv**:
  ```
  id;name;gender;email;department;order_date
  1;Sage;Male;sage_wieser@cox.net;R&D;2012-05-15
  2;Kris;Female;kris@gmail.com;Test;2011-08-07
  3;Abel;Male;amaclead@gmail.com;HR;2014-04-16
  4;Mattie;Female;mattie@aol.com;Test;2012-09-10
  5;Bernardo;Female;bfigeroa@aol.com;Marketing;2013-02-25
  ```

- **delivery_details.csv**:
  ```
  Name;Datetime
  Reita;2012-02-11 15:30:32
  Larae;2015-08-28 09:30:32
  ```

The example involves two Jobs one calling the other: the parent Job maps the file names to table names in the database, and the child Job reads data from the files and writes the data to the corresponding database tables.

**Prerequisites to this Job example**

Before creating, configuring and executing the Jobs in this example, make sure:

- The target tables have been created in the MySQL database, and the table structures and column names are identical to those of the corresponding source files.

  **Tip:** You can create database tables quickly using the tCreateTable component.

- The source files are available in your file system.

**Creating the parent Job**

**Procedure**

1. Create a standard Job and name it LoadDynamicSchemaParent.
2. Drop the following components onto the design workspace, and label them according to their roles in the Job.
Using Dynamic Schema to load data dynamically to database tables

<table>
<thead>
<tr>
<th>Component</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>tFileList</td>
<td>File_list</td>
</tr>
<tr>
<td>tIterateToFlow</td>
<td>File_name_flow</td>
</tr>
<tr>
<td>tMap</td>
<td>File_table_lookup</td>
</tr>
<tr>
<td>tFixedFlowInput</td>
<td>Mappings</td>
</tr>
<tr>
<td>tJavaRow</td>
<td>Set_context_variables</td>
</tr>
<tr>
<td>tRunJob</td>
<td>Call_child</td>
</tr>
</tbody>
</table>

3. Connect the components:
   a) tFileList to tIterateToFlow using a Row > Iterate connection
   b) tIterateToFlow to tMap using a Row > Main connection
   c) tFixedFlowInput to tMap using a Row > Main connection, as a lookup connection
   d) tMap to tJavaRow using a Row > Main connection (named out in this example)
   e) tJavaRow to tRunJob using a Row > Main connection

Creating the child Job

Procedure

1. Create a standard Job and name it LoadDynamicSchemaChild.
2. Drop the following components onto the design workspace, and label them according to their roles in the Job.

<table>
<thead>
<tr>
<th>Component</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>tFileInputDelimited</td>
<td>Input_file</td>
</tr>
<tr>
<td>tDBOutput</td>
<td>Write_to_DB</td>
</tr>
</tbody>
</table>

3. In the Basic settings view of the tDBOutput component, select MySQL from the Database list and click Apply.
4. Connect the tFileInputDelimited component to the tDBOutput component using a Row > Main connection.
Using Dynamic Schema to load data dynamically to database tables

Configuring the parent Job

Procedure

1. In the Contexts view:
   a) Add the following three variables:
      • filename, type String
      • tablename, type String
      • directory, type Directory
   b) Specify the directory by clicking in the Value field of variable directory. Click the small button that appears and browse to the directory where the source files are stored.

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Comment</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>filename</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>tablename</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>directory</td>
<td>Directory</td>
<td>Dr:Talend/Data/input/SourceFiles/</td>
</tr>
</tbody>
</table>
```

2. In the Basic settings view of the tFileList component, fill the Directory field with context.directory.

   Tip: Click in the field and press Ctrl+Shift to access a list of available variables.

3. Double-click the tIterateToFlow component to open its Basic settings view, and configure the component as follows:
   a) Add a column named filename (type String) in the schema editor.
   b) Fill the Value field of the Mapping table with the CURRENT_FILE global variable generated by the tFileList component, in the format of ((String)globalMap.get("tFileList_1_CURRENT_FILE")).

4. Double-click the tFixedFlowInput component to open its Basic settings view, and configure the component:
   a) Add two columns in the schema: file_name (type String) and table_name (type String).
b) Select the **Use Inline Table** option, and define the following file-to-table mappings in the table:

<table>
<thead>
<tr>
<th>file_name</th>
<th>table_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;customer_details.csv&quot;</td>
<td>&quot;customer&quot;</td>
</tr>
<tr>
<td>&quot;delivery_details.csv&quot;</td>
<td>&quot;delivery&quot;</td>
</tr>
<tr>
<td>&quot;employee_details.csv&quot;</td>
<td>&quot;employee&quot;</td>
</tr>
</tbody>
</table>

5. **Double-click the tMap component to open the map editor, and in the map editor:**

   a) Drag the `filename` column of table **row1** (which is from the `tIterateToFlow` component) and drop it onto the `file_name` column of table **row2** (which is from the `tFixedFlowInput` component), to join the two tables for file name look-up.

   b) Click the spawner button to show the join properties, and set:
      - the **Match Model** to **Unique match**
      - the **Join Model** to **Inner Join**

   c) In the schema editor, add two columns to the **out table**: `tablename` (type String) and `filename` (type String).

   d) Drag the `filename` column of table **row1** and drop it onto the `filename` column of table **out**.

   e) Drag the `table_name` column of table **row2** and drop it onto the `tablename` of table **out**.

   f) Click **OK** to validate your settings and close map editor.
6. Double-click the **tJavaRow** component, and enter the following code in the **Code** field.

```java
context.tablename = out.tablename;
context.filename = out.filename;
```

7. In the **Basic settings** view of the **tRunJob** component:
   a) Click the three-dot button next to the **Job** field and select the child Job you want to call from the **Repository Content** dialog box, **LoadDynamicSchemaChild** in this example.
   b) Select the **Transmit whole context** option to pass the context variables to the child Job.

8. Press **Ctrl + S** to save the Job.

## Configuring the child Job

### Procedure

1. In the **Contexts** view, add the following three variables, leaving their values undefined:
   - filename, type String
   - tablename, type String
   - directory, type String

2. Double-click the **tFileInputDelimited** component (labelled **Input_file**) to open its **Basic settings** view, and configure the component as follows:
   - Open the schema editor and add a column **data**, and set its type to **Dynamic**.
   - Fill the **File name/Stream** field with the context variables defined for file accessing: `context.directory+context.filename`.
     - **Tip**: Click in the field and press **Ctrl+Shift** to access a list of available variables.
   - In the **Header** field, specify the number of rows at the beginning of the files to skip.
     - In this example, the first row of each file is the header row.
   - Leave the other settings as default.
3. Double-click the tDBOutput component (labelled Write_to_DB) to open its Basic settings view, and configure the component as follows:

   - Specify the database connection details, including the host name or IP address, the port number, and database name, and the authentication credentials.
   - Fill the Table field with the context variable defined for the table name, contexttablename in this example.
   - From the Action on table list, select Default.
   - From the Action on data list, select Insert.
   - Click Sync columns to ensure the schema is the same as the input component: a single column named data, type Dynamic.
   - Leave other options as default.

4. Press Ctrl + S to save the Job.

Executing the parent Job and check the data loading result

Procedure

1. Go to the parent Job.
2. Press **F6** to run the Job and check your MySQL database tables.

**Results**

The content of each of the flat files is loaded to the corresponding database table.
Using the output stream feature

The following use case aims to show how to use the output stream feature in a number of components in order to greatly improve the output performance.

In this scenario, a pre-defined csv file containing customer information is loaded in a database table. Then the loaded data is selected using a tMap, and output to a local file and to the console using the output stream feature.


Input data

The input file, the data of which will be loaded into the database table, contains customer information of various aspects.

The file structure usually called Schema in Talend Studio includes the following columns:

- id (Type: Integer)
- CustomerName (Type: String)
- CustomerAge (Type: Integer)
- CustomerAddress (Type: String)
- CustomerCity (Type: String)
- RegisterTime (Type: Date)

Output data

The tMap component is used to select the id, CustomerName and CustomerAge columns from the input data. Then the selected data is output using the output stream feature.

Thus the expected output data should have the following structure:

- id (Type: Integer)
- CustomerName (Type: String)
- CustomerAge (Type: Integer)

All the three columns above come from the respective columns in the input data.

Translating the scenario into a Job

In order to implement this scenario, break down the Job into four steps:

1. Create the Job, define the schema for the input data, and read the input file according to the defined schema.
2. Set the command to enable the output stream feature.
3. Map the data using the tMap component.
4. Output the selected data stream.

A complete Job looks as what it displays in the following image. For the detailed instruction for designing the Job, read the following sections.
Step 1: Reading input data from a local file

We will use the `tFileInputDelimited` component to read the file `customers.csv` for the input data. This component can be found in the File/Input group of the Palette.

Procedure

1. Drop a `tFileInputDelimited` component onto the design workspace, and double-click the to open the Basic settings view to set its properties.

2. Click the three-dot button next to the File name/Stream field to browse to the path of the input data file. You can also type in the path of the input data file manually.

3. Click Edit schema to open a dialog box to configure the file structure of the input file.

4. Click the plus button to add six columns and set the Type and columns names to what we listed in the following:

<table>
<thead>
<tr>
<th>Column</th>
<th>Key</th>
<th>Type</th>
<th>N_</th>
<th>Date Pattern</th>
<th>Length</th>
<th>Pre_</th>
<th>D_</th>
<th>Co_</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td></td>
<td>Integer</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CustomerName</td>
<td></td>
<td>String</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CustomerAge</td>
<td></td>
<td>Integer</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CustomerAddress</td>
<td></td>
<td>String</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CustomerCity</td>
<td></td>
<td>String</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RegisterTime</td>
<td></td>
<td>Date</td>
<td>✔</td>
<td>&quot;dd-MM...&quot;</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5. Click OK to close the dialog box.
**Step 2: Setting the command to enable the output stream feature**

Now we will make use of **tJava** to set the command for creating an output file and a directory that contains the output file.

**Procedure**

1. Drop a **tJava** component onto the design workspace, and double-click it to open the **Basic settings** view to set its properties.

   ![tJava component](image)

   **Code**
   ```java
   new java.io.File("C:/myFolder").mkdirs();
   globalMap.put("out_file", new java.io.FileOutputStream("C:/myFolder/customerselection.txt", false));
   ```

   **Note:**
   The command we typed in this step will create a new directory `C:/myFolder` for saving the output file `customerselection.txt`. You can customize the command in accordance with actual practice.

2. Fill in the **Code** area with the following command:

   ```java
   new java.io.File("C:/myFolder").mkdirs();
   globalMap.put("out_file", new java.io.FileOutputStream("C:/myFolder/customerselection.txt", false));
   ```

3. Connect **tJava** to **tFileInputDelimited** using a **Trigger > On Subjob Ok** connection.
   This will trigger the subjob that starts with **tFileInputDelimited** when **tJava** succeeds in running.

   ![Connection diagram](image)

**Step 3: Mapping the data using the tMap component**

**Procedure**

1. Drop a **tMap** component onto the design workspace, and double-click it to open the **Basic settings** view to set its properties.
2. Click the three-dot button next to **Map Editor** to open a dialog box to set the mapping.

3. Click the plus button on the left to add six columns for the schema of the incoming data, these columns should be the same as the following:

   - id
   - CustomerName
   - CustomerAge
   - CustomerAddress
   - CustomerCity
   - RegisterTime

4. Click the plus button on the right to add a schema of the outgoing data flow.

5. Select **New output** and click **OK** to save the output schema.
For the time being, the output schema is still empty.

6. Click the plus button beneath the **out1** table to add three columns for the output data.

7. Drop the **id**, **CustomerName** and **CustomerAge** columns onto their respective line on the right.

8. Click **OK** to save the settings.

**Step 4: Outputing the selected data stream**

**Procedure**

1. Drop a **tFileOutputDelimited** component onto the design workspace, and double-click it to open the **Basic settings** view to set its component properties.

2. Select the **Use Output Stream** check box to enable the **Output Stream** field and fill the **Output Stream** field with the following command:

   ```java
   (java.io.OutputStream)globalMap.get("out_file")
   ```
Using the output stream feature

**Note:**
You can customize the command in the **Output Stream** field by pressing **Ctrl+Space** to select built-in command from the list or type in the command into the field manually in accordance with actual practice. In this scenario, the command we use in the **Output Stream** field will call the `java.io.OutputStream` class to output the filtered data stream to a local file which is defined in the **Code** area of **tJava** in this scenario.

3. Connect **tFileInputDelimited** to **tMap** using a **Row > Main** connection and connect **tMap** to **tFileOutputDelimited** using a **Row > out1** connection which is defined in the **Map Editor** of **tMap**.

4. Click **Sync columns** to retrieve the schema defined in the preceding component.

5. Drop a **tLogRow** component onto the design workspace, and double-click it to open its **Basic settings** view.

6. Select the **Table** radio button in the **Mode** area.

7. Connect **tFileOutputDelimited** to **tLogRow** using a **Row > Main** connection.

8. Click **Sync columns** to retrieve the schema defined in the preceding component.

This Job is now ready to be executed.

9. Press **Ctrl+S** to save your Job and press **F6** to execute it.

The content of the selected data is displayed on the console.

```
Starting job OutputStream at 17:31 19/10/2011.
[statistics] connecting to socket on port 4059
[statistics] connected

<table>
<thead>
<tr>
<th>id</th>
<th>CustomerName</th>
<th>CustomerAge</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>Warren</td>
<td>57</td>
</tr>
<tr>
<td>10002</td>
<td>Woodrow</td>
<td>68</td>
</tr>
<tr>
<td>10003</td>
<td>Grover</td>
<td>77</td>
</tr>
<tr>
<td>10004</td>
<td>Abraham</td>
<td>74</td>
</tr>
<tr>
<td>10005</td>
<td>Chester</td>
<td>78</td>
</tr>
<tr>
<td>10006</td>
<td>Calvin</td>
<td>63</td>
</tr>
<tr>
<td>10007</td>
<td>Zachary</td>
<td>53</td>
</tr>
<tr>
<td>10008</td>
<td>Chester</td>
<td>36</td>
</tr>
<tr>
<td>10009</td>
<td>Chester</td>
<td>60</td>
</tr>
<tr>
<td>10010</td>
<td>Woodrow</td>
<td>57</td>
</tr>
</tbody>
</table>

[statistics] disconnected
Job OutputStream ended at 17:31 19/10/2011. [exit code=0]
```
The selected data is also output to the specified local file `customerselection.txt`.

<table>
<thead>
<tr>
<th>id</th>
<th>CustomerName</th>
<th>CustomerAge</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>Warren</td>
<td>67</td>
</tr>
<tr>
<td>10002</td>
<td>Woodrow</td>
<td>68</td>
</tr>
<tr>
<td>10003</td>
<td>Grover</td>
<td>77</td>
</tr>
<tr>
<td>10004</td>
<td>Abraham</td>
<td>74</td>
</tr>
<tr>
<td>10005</td>
<td>Chester</td>
<td>78</td>
</tr>
<tr>
<td>10006</td>
<td>Calvin</td>
<td>63</td>
</tr>
<tr>
<td>10007</td>
<td>Zachary</td>
<td>53</td>
</tr>
<tr>
<td>10008</td>
<td>Chester</td>
<td>36</td>
</tr>
<tr>
<td>10009</td>
<td>Chester</td>
<td>60</td>
</tr>
<tr>
<td>10010</td>
<td>Woodrow</td>
<td>57</td>
</tr>
</tbody>
</table>
Using the Implicit Context Load feature

Job parameterization based on context variables enables you to orchestrate and execute your Jobs in different contexts or environments. You can define the values of your context variables when creating them, or load your context parameters dynamically, either explicitly or implicitly, when your Jobs are executed.

The use case below describes how to use the Implicit Context Load feature of your Talend Studio to load context parameters dynamically at the time of Job execution.

The Job in this use case is composed of only two components. It will read employees data stored in two MySQL databases, one for testing and the other for production purposes. The connection parameters for accessing these two databases are stored in another MySQL database. When executed, the Job loads these connection parameters dynamically to connect to the two databases.

Creating the Job and defining context variables

Before you begin

Create two tables named db_testing and db_production respectively in a MySQL database named db_connections, to hold the connection parameters for accessing the above mentioned databases, testing and production. Each table should contain only two columns: key and value, both of type VARCHAR. Below is an example of the content of the database tables:

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>localhost</td>
</tr>
<tr>
<td>port</td>
<td>3306</td>
</tr>
<tr>
<td>username</td>
<td>root</td>
</tr>
<tr>
<td>password</td>
<td>talend</td>
</tr>
<tr>
<td>database</td>
<td>testing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>localhost</td>
</tr>
<tr>
<td>port</td>
<td>3306</td>
</tr>
<tr>
<td>username</td>
<td>root</td>
</tr>
<tr>
<td>password</td>
<td>talend</td>
</tr>
<tr>
<td>database</td>
<td>production</td>
</tr>
</tbody>
</table>

You can create these database tables using another Talend Job that contains tFixedFlowInput and tMysqlOutput components.
Using the Implicit Context Load feature

Procedure

1. Create a Job and add a tMysqlInput component and a tLogRow component onto the design workspace, and link them using a Row > Main connection.

2. Select the Contexts view of the Job, and click the [+] button at the bottom of the view to add five rows in the table to define the following variables, all of type String, without defining their values, which will be loaded dynamically at Job execution: host, port, username, password, and database.

3. Now create another variable named db_connection of type List Of Value.

4. Click in the Value field of the newly created variable and click the button that appears to open the Configure Values dialog box, and click New... to open the New Value dialog box. Enter the name of one of the database tables holding the database connection details and click OK.
5. Click **New...** again to define the other table holding the database connection details. When done, click **OK** to close the **Configure Values** dialog box.

Now the variable `db_connection` has a list of values `db_testing` and `db_production`, which are the database tables to load the connection parameters from.

6. Select the **Prompt** check box next to the **Value** field of the `db_connection` variable to show the **Prompt** fields and enter the prompt message to be displayed at the execution time.
Configuring the components

Procedure

1. Then double-click to open the `tMysqlInput` component Basic settings view.
2. Fill the Host, Port, Database, Username, Password, and Table Name fields with the relevant variables defined in the Contexts tab view: context.host, context.port, context.database, context.username, and context.password respectively in this example.

3. Fill the Table Name field with employees, which is the name of the table holding employees information in both databases in our example.

4. Then fill in the Schema information. If you stored the schema in the Repository, then you can retrieve it by selecting Repository and the relevant entry in the list.

In this example, the schema of both the database tables to read is made of six columns: id (INT, 2 characters long), name (VARCHAR, 20 characters long), email (VARCHAR, 25 characters long), sex (VARCHAR, 1 characters long), department (VARCHAR, 10 characters long), and position (VARCHAR, 10 characters long).
5. Click **Guess Query** to retrieve all the table columns, which will be displayed on the **Run** tab, through the **tLogRow** component.

6. In the **Basic settings** view of the **tLogRow** component, select the **Table** option to display data records in the form of a table.

### Configuring the Implicit Context Load feature

You can configure the Implicit Context Load feature either in Project Settings so that it can be used across Jobs within the project, or in the Job view for a particular Job.

The following example shows how to configure the Implicit Context Load feature in the **Job** view for this particular Job. If you want to configure the feature to be reused across different Jobs, select **File > Edit Project properties** from the menu bar to open the **Project Settings** dialog box, go to **Job Settings > Implicit context load**, select the **Implicit tContextLoad** check box, and set the parameters following steps 2 through 6 below. Then in the **Job** view, select the **Use Project Settings** check box to apply the settings to the Job.

**Procedure**

1. From the **Job** view, select the **Extra** vertical tab, and select the **Implicit tContextLoad** check box to enable context loading without using the **tContextLoad** component explicitly in the Job.

2. Select the source to load context parameters from. A context source can be a two-column flat file or a two-column database table. In this use case the database connection details are stored in database tables, so select the **From Database** option.

3. Define the database connection details just like defining the basic settings of a database input component.

   In this example, all the connection parameters are used just for this particular Job, so select **Built-In** from the **Property Type** list and fill in the connection details manually.
Using the Implicit Context Load feature

4. Fill the Table Name field with the context variable named `db_connection` defined in the Contexts view of the Job so that we will be able to choose the database table to load context parameters from dynamically at Job execution.

5. As we will fetch all the connection details from the database tables unconditionally, leave the Query Condition field blank.

6. Select the Print operations check box to list the context parameters loaded at Job execution.

**Executing the Job**

**Procedure**

1. Press Ctrl+S to save the Job, and press F6 to run the Job.

2. A dialog box pops up asking you to select a database. Select a database and click OK.

   The loaded context parameters and the content of the "employees" table of the selected database are displayed on the Run console.

3. Now press F6 to launch the Job again and select the other database when prompted.

   The loaded context parameters and the content of the "employees" table of the other database are displayed on the Run console.
Using the Multi-thread Execution feature to run Jobs in parallel

Based on the previous use case Using the output stream feature on page 21, this use case give an example of how to use the Multi-thread Execution feature to run two Jobs in parallel to display the employees information in both the testing and production environments at the same time. When handling large data volumes, this feature can significantly optimize the Job execution performance of the Talend Studio.


Preparing Jobs to read employees data in different contexts

Procedure

1. In the Repository tree view, right-click the Job created in the use case Using the output stream feature on page 21 and select Duplicate from the context menu. Then, in the Duplicate dialog box enter a new name for the Job, employees_testing in this example, and click OK.

2. Open the new Job, and label the components to better reflect their roles.

3. Create another Job named employees_production by repeating the steps above.
4. In the **Contexts** view of both Jobs, remove the **db_connection** variable.

5. On **Extra** tab of the **Job** view of the Job **employees_testing**, fill the **Table Name** field of database settings with **db_testing**; on the **Extra** tab of the **Job** view of the Job **employees_production**, fill the **Table Name** field with **db_production**.

---

**Set up a parent Job to run the Jobs in parallel**

**Procedure**

1. Create a new Job and add two **tRunJob** components on the design workspace, and label the components to better reflect their roles.

2. In the **Component** view of the first **tRunJob** component, click the **[...]** button next to the **Job** field and specify the Job it will run, **employees_testing** in this example.

3. Configure the other **tRunJob** component to run the other Job, **employees_production**.
Using the Multi-thread Execution feature to run Jobs in parallel

4. On the Extra tab of the Job view, select the Multi thread execution check box to activate the Multi-thread Execution feature.

Executing the Jobs

Procedure

1. Save each Job by pressing Ctrl+S.
2. In the parent Job, press F6 of click Run on the Run view to start execution of the child Jobs.

   The child Jobs are executed in parallel, reading employees data from both databases and displaying the data on the console.
Using the Multi-thread Execution feature to run Jobs in parallel

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>email</th>
<th>sex</th>
<th>department</th>
<th>position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Elisa</td>
<td><a href="mailto:elisa@company.com">elisa@company.com</a></td>
<td>F</td>
<td>R&amp;D</td>
<td>Manager</td>
</tr>
<tr>
<td>2</td>
<td>Nicolas</td>
<td><a href="mailto:nicolas@company.com">nicolas@company.com</a></td>
<td>M</td>
<td>R&amp;D</td>
<td>Developer</td>
</tr>
<tr>
<td>3</td>
<td>Cedric</td>
<td><a href="mailto:cedric@company.com">cedric@company.com</a></td>
<td>M</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>4</td>
<td>Rabbit</td>
<td><a href="mailto:rabbit@company.com">rabbit@company.com</a></td>
<td>M</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>5</td>
<td>Mike</td>
<td><a href="mailto:mike@company.com">mike@company.com</a></td>
<td>M</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>6</td>
<td>Sabrina</td>
<td><a href="mailto:sabrina@company.com">sabrina@company.com</a></td>
<td>F</td>
<td>Community</td>
<td>Developer</td>
</tr>
<tr>
<td>7</td>
<td>Stephanz</td>
<td><a href="mailto:stephanz@company.com">stephanz@company.com</a></td>
<td>M</td>
<td>Sales</td>
<td>Manager</td>
</tr>
<tr>
<td>8</td>
<td>Jim</td>
<td><a href="mailto:jim@company.com">jim@company.com</a></td>
<td>M</td>
<td>Sales</td>
<td>Pre-sales</td>
</tr>
<tr>
<td>9</td>
<td>John</td>
<td><a href="mailto:john@company.com">john@company.com</a></td>
<td>M</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>email</th>
<th>sex</th>
<th>department</th>
<th>position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Herbert Pierce</td>
<td><a href="mailto:hp@tailead.com">hp@tailead.com</a></td>
<td>M</td>
<td>R&amp;D</td>
<td>Manager</td>
</tr>
<tr>
<td>2</td>
<td>John Hoover</td>
<td><a href="mailto:jh@tailead.com">jh@tailead.com</a></td>
<td>M</td>
<td>Finance</td>
<td>Manager</td>
</tr>
<tr>
<td>3</td>
<td>Benjamin Harrison</td>
<td><a href="mailto:bh@tailead.com">bh@tailead.com</a></td>
<td>M</td>
<td>HR</td>
<td>Manager</td>
</tr>
<tr>
<td>4</td>
<td>George Harrison</td>
<td><a href="mailto:gh@tailead.com">gh@tailead.com</a></td>
<td>M</td>
<td>Sales</td>
<td>Manager</td>
</tr>
<tr>
<td>5</td>
<td>Helen Monroe</td>
<td><a href="mailto:hn@tailead.com">hn@tailead.com</a></td>
<td>F</td>
<td>R&amp;D</td>
<td>Developer</td>
</tr>
<tr>
<td>6</td>
<td>Anne Harrison</td>
<td><a href="mailto:ah@tailead.com">ah@tailead.com</a></td>
<td>F</td>
<td>Sales</td>
<td>Pre-sales</td>
</tr>
<tr>
<td>7</td>
<td>Thomas Nixon</td>
<td><a href="mailto:tn@tailead.com">tn@tailead.com</a></td>
<td>M</td>
<td>R&amp;D</td>
<td>Developer</td>
</tr>
<tr>
<td>8</td>
<td>James Lincoln</td>
<td><a href="mailto:ji@tailead.com">ji@tailead.com</a></td>
<td>M</td>
<td>R&amp;D</td>
<td>Developer</td>
</tr>
<tr>
<td>9</td>
<td>Rutherford Fillmore</td>
<td><a href="mailto:rf@tailead.com">rf@tailead.com</a></td>
<td>M</td>
<td>Finance</td>
<td>Accountant</td>
</tr>
<tr>
<td>10</td>
<td>Maria Pierce</td>
<td><a href="mailto:mp@tailead.com">mp@tailead.com</a></td>
<td>F</td>
<td>Finance</td>
<td>Accountant</td>
</tr>
</tbody>
</table>
Calling the MetaServlet REST Web service to execute a task

In the following scenario, a Job created in the Studio invokes the MetaServlet REST Web service to execute a specific task in Talend Administration Center.

**Warning:** This documentation is relevant only if you are using a subscription-based Talend Studio.

**Prerequisites:**
Make sure an existing task is available in the Job Conductor page of Talend Administration Center.

**Dropping and linking the components**

**Procedure**
1. Drop the following components from the Palette onto the design workspace: a `tSetGlobalVar` component, a `tREST` component and a `tLogRow` component.
2. Connect the `tREST` component to the `tLogRow` component using a `Row > Main` connection.
3. Connect the `tSetGlobalVar` component to the `tREST` component using a `Trigger > OnSubjobOK` connection.

**Configuring the context variables**

You have to define the variables that will be used to call the MetaServlet REST Web service.

**Procedure**
1. In the **Contexts** tab view, click the `+` button four times to add four variables.
2. Name these variables `tac_url`, `tac_user`, `tac_pwd` and `task_id`.
3. In the Value field under the Default context, enter the variable values:
• For the tac_url variable, type in the URL of the Talend Administration Center Web application, http://localhost:8080/org.talend.administrator for example.
• For the tac_user variable, type in the administrator user name in Talend Administration Center Web application, admin@company.com for example.
• For the tac_pwd variable, type in the administrator password in Talend Administration Center Web application, admin for example.
• For the task_id variable, type in the ID of the task you want to generate, 1 for example.

4. Press Ctrl+S to save your changes.

Configuring the Routine

You have to define a user routine that encodes the MetaServlet parameters in base64 to be transmitted to the REST API. This routine will be called in the Job.

Procedure

1. In the Repository tree view, expand Code to display the Routines folder.
2. Right-click Routines and select Create routine.
3. The New routine dialog box opens. Enter the information required to create the routine, then click Finish to proceed to the next step.

The newly created routine appears in the Repository tree view, directly below the Routines node. The routine editor opens to reveal a model routine which contains a simple example, by default, comprising descriptive text in blue, followed by the corresponding code.

4. At the beginning, right after the package routines line of code, add the following:

```java
import com.sun.org.apache.xml.internal.security.utils.Base64;
```

To do so, start typing and press Ctrl+Space to open the list of templates, then select com.sun.org.apache.xml.internal.security.utils.*; then replace the * sign with Base64.

5. Modify the model at the end with your own code:

```java
public static String base64Encode(String message) {
    return message==null ? null : String.valueOf(Base64.encode(message.getBytes()).replace("
",""));
}

public static void main(String[] args) {
    String tmp = "{""actionName"":"runTask"","taskId"":"1","mode":"synchronous","
"authPass":"talend","authUser":"talend@talend.com"}");
    System.out.println("Base 64: " + base64Encode(tmp));
}
```

This allows you to encode the MetaServlet action, runTask, in base64.

For more information about the parameters and actions available in the MetaServlet, see the Talend Administration Center User Guide.
Configuring the components

Procedure

1. Double-click the `tSetGlobalVar` component to display its Basic settings view in the Component tab.
2. Click the [+] button to add a line in the Variables table:
   - In the Key field, type in `jsonEncoded`
   - In the Value fields, type in:
     ```
     MetaServlet.base64Encode("{"actionName":"runTask","taskId":"","mode":"synchronous","context":{"Default":"" + ((String)globalMap.get("tMsgBox_1_RESULT")) + "","authPass":"" + context.tac_pwd + ","authUser":"" + context.tac_user + "\"")
     ```
   to call the routine you have previously created.
3. Then double-click the `tREST` component to display its Basic settings view.
4. Fill the URL field with the URL of the Web service you are going to invoke. For this use case, type in:
   ```
   context.tac_url + "/metaServlet?" +globalMap.get("jsonEncoded")
   ```
   to call the service and encode the MetaServlet parameters in a Json format.
5. From the HTTP Method list, select GET to send an HTTP request for generating a task.
   In this way, the MetaSerlet is invoked via the REST API of Talend Administration Center with the relevant parameters.
6. In the Basic settings view of the `tLogRow` component, select the Basic option to display the result in the Run console.
7. Save your Job and press F6 to launch it.

Results

The console shows that the `tREST` component sends an HTTP request to the server end to run the specified task, and that the task has been executed without errors.

In the Job Conductor page of Talend Administration Center, the status of the task is now ready to run.
Calling the MetaServlet REST Web service to execute a task

<table>
<thead>
<tr>
<th>Project: bpm (1 item)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ready to run</td>
</tr>
</tbody>
</table>
Validation rules Job example

**Warning:** This documentation is relevant only if you are using a subscription-based Talend Studio.

If you need to filter your data to only send valid data to your output flows, you can create validation rules that will check your data. These validation rules can be a basic check for correct values or referential integrity check. As they are stored at the metadata level in the Repository, they can be easily reused and modified.

For more information on how to create those rules, see the *Centralizing a Validation Rule* section of the Talend Studio User Guide at https://help.talend.com.

**Using a reference validation rule**

After setting up a validation rule, you can apply it on your Job designs. For example, let’s apply a referential integrity check validation rule when updating a database with new data. But before uploading those new data we want to make sure they match the ones already existing in the database.

This scenario is based on the reference check validation rule created in the *Referential rule* section of the Talend Studio User Guide.

**Setting up the Job environment**

**Procedure**

1. From the Palette, drop these components onto the design workspace: a database input component and a database output component, here `tMysqlInput` and `tMysqlOutput`, to upload the data, a `tLogRow` component to display the rejected data in the console and a `tJava` component to display the number of lines processed in the console.

2. Connect the input and output database components using a Row > Main connection, and connect the `tMysqlInput` to the `tJava` components using a Trigger > OnSubjobOk connection.

**Note:**

You will be able to create the reject link between the `tMysqlOutput` and `tLogRow` components only when you will have applied the validation rule to the `tMysqlOutput` component.
Configuring the components

Procedure

1. Double-click the `tMysqlInput` component to display its Basic settings.

2. Select Repository as Property type and click the three-dot button next to the field to retrieve the connection properties that corresponds to the metadata you want to check.

3. Select Repository from the Schema drop down list and click the three-dot button next to the field to retrieve the schema that corresponds to your database table.

4. Click the three-dot button next to the Table field to select the table to check.

5. Click Guess Query to automatically retrieve the query corresponding to the table schema.
6. Double-click the **tMysqlOutput** component to display its **Basic settings**.

![tMysqlOutput](Image)

<table>
<thead>
<tr>
<th>Basic settings</th>
<th>Advanced settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property Type</td>
<td>Repository - DB (MySQL):talend</td>
</tr>
<tr>
<td>DB Version</td>
<td>Mysql 5 -</td>
</tr>
<tr>
<td>Use an existing connection</td>
<td></td>
</tr>
</tbody>
</table>

7. Select **Repository** as **Property type** and click the three-dot button next to the field to retrieve the connection properties that corresponds to the database table in which you want to load the new data.

8. Click the three-dot button next to the **Table** field to select the table in which you will load the data.

9. In the **Action on table** list, select **Default** and in the **Action on data** list, select the action corresponding to the one(s) defined in the validation rule you apply on the Job. Here, as we selected **On insert** and **On update** in the referential check validation rule we use, so select **Update** or **insert** to trigger the rule.

10. If the schema of the input and output components did not synchronize automatically, click **Sync columns** and the schema of the input flow will automatically be retrieved.

### Applying the validation rule and viewing the Job execution result

**Procedure**

1. Click the **Validation Rules** tab in the **Component** view of **tMysqlOutput**.

   ![tMysqlOutput](Image)

   - Use an existing validation rule
   - Validation Rule Type: Repository - VRestate_validation

2. Select the **Use an existing validation rule** check box to apply the validation rule to the component.

3. In the **Validation Rule Type** list, select **Repository** and click the three-dot button to select the validation rule from the **Repository Content** window.

4. Right-click **tMysqlOutput**, select **Row > Rejects** in the menu and drag to **tLogRow** to create a reject link between the two components.

   If you have enabled the **Reject link** option for this validation rule you can retrieve the rejected data to the reject flow.
5. Double-click the tJava component to display its Basic settings.

6. In the Code field, type in the code that will display the number of updated, inserted and rejected lines processed:

```
System.out.println("Updates: "+((Integer)globalMap.get("tMysqlOutput_1_NB_LINE_UPDATED"))) + "\nInserts: "+((Integer)globalMap.get("tMysqlOutput_1_NB_LINE_INSERTED"))) + "\nRejects: "+((Integer)globalMap.get("tLogRow_1_NB_LINE")));  
```

7. Save your Job and press F6 to execute it.

```

[statistics] connecting to socket on port 3651
[statistics] connected
1|Andrew|Cleveland|South Roosevelt Drive|20/01/2005|55239|XX|
4|Grover|Ford|Cleveland Ave.|27/11/2001|76036|XX|
Updates: 1
Inserts: 1
Rejects: 2
[statistics] disconnected
Job validation_rules_by_reference ended at 18:15 12/04/2011. [exit code=0]
```

**Results**

Valid data is inserted or updated in the database table and the console displays the rows rejected by the validation rule, along with the number of updates, inserts and rejects processed in the Job.

**Using a basic validation rule**

After setting up a validation rule, you can apply it on your Job designs. For example, let’s apply a basic check for correct values validation rule when reading a database table. This scenario is based on the basic check validation rule created in the Basic rule section of Talend Studio User Guide.

**Setting up the Job environment**

**Procedure**

1. From the Palette, drop these components onto the design workspace: a database input component, here tMysqlInput, that you will read and check the values of, two tFileOutputDelimited components to extract the valid data in one file and the rejected ones in another file, and a tJava component to display the number of lines processed in the console.
2. Connect the input database component to the first `tFileOutputDelimited` component using a `Row > Main` connection, and connect the `tMysqlInput` component to the `tJava` component with a `Trigger > OnSubjobOk` connection.

**Note:**
You will be able to create the reject link between the `tMysqlInput` and the second `tFileOutputDelimited` component only when you will have applied the validation rule to the `tMysqlInput` component.

### Configuring tMysqlInput

**Procedure**

1. Double-click the `tMysqlInput` component to display its Basic settings.
2. Select **Repository** as Property type and click the three-dot button next to the field to retrieve the connection properties that corresponds to the metadata you want to check.

3. Select **Repository** from the Schema drop down list and click the three-dot button next to the field to retrieve the schema that corresponds to your database table.

4. Click the three-dot button next to the **Table** field to select the table to check.

5. Click **Guess Query** to automatically retrieve the query corresponding to the table schema.

**Applying the validation rule and creating a reject link**

As the **On select** option was selected for this validation rule, the validation rule must be applied to the input component.

**Procedure**

1. Click the Validation Rules tab of the **tMysqlInput Component** view.

2. Select the **Use an existing validation rule** check box to apply the validation rule to the component.
3. In the Validation Rule Type list, select Repository and click the [...] button to select the validation rule from the Repository Content window.

4. Right-click tMysqlInput, select Row > Reject in the menu and drag to the second tFileOutputDelimited component to create a reject link between the two components.

   If you have enabled the Reject link option for this validation rule you can retrieve the rejected data to a reject flow.

**Configuring the output components and viewing the Job execution result**

**Procedure**

1. Double-click the first tFileOutputDelimited component to display its Basic settings.

   ![tFileOutputDelimited_1](image)

   2. In the File Name field, specify the path and name of the file to write with the valid data.
   3. Define the row and field separators in the corresponding fields.
   4. Select the Include Header check box to include column headers in the output data.
   5. Repeat the steps above on the second tFileOutputDelimited component to configure the output of the rejected data.

   ![tFileOutputDelimited_2](image)

6. Double-click the tJava component to display its Basic settings.
7. In the **Code** field, type in the code that will display the number of updated, inserted and rejected lines processed:

```java
System.out.println("Valid data: "+((Integer)globalMap.get("tFileOutputDelimited_1_NB_LINE")+"\nRejected data: "+((Integer)globalMap.get("tFileOutputDelimited_2_NB_LINE")));)
```

8. Save your Job and press **F6** to execute it.

```
Starting job basic_phone_validation at 16:00 26/04/2011.
[statistics] connecting to socket on port 3510
[statistics] connected
Valid data: 510
Rejected data: 482
[statistics] disconnected
Job basic_phone_validation ended at 16:00 26/04/2011.
[exit code=0]
```

**Results**

Valid data is outputted in the first delimited file and rejects to the second, and the console displays the number of valid lines and the number of rejects processed in the Job.
Using the JDBC connector to connect to Amazon Athena

This example shows how to use the Java JDBC driver to connect to the Amazon Athena service and leverage Amazon Athena service in Talend Studio.

Talend 7.1.1 is used for the implementation in this article. However, any version of Talend that supports JDBC can be used to connect to Amazon Athena. Amazon Athena is currently available only in selected AWS regions. It is recommended to verify the availability of the service from AWS global infrastructure region table.

The query results from Amazon Athena needs to be saved to Amazon S3. So, we have created an S3 output folder for the same.

For more information on Amazon Athena, Amazon S3, and JDBC, see:
- Amazon Athena
- Amazon S3
- Java JDBC

Setting up a JDBC connection to connect to Amazon Athena

Before you begin
You have the required additional library AthenaJDBC42-2.0.7.jar ready. If not, download the Athena JDBC driver here.

About this task
Follow the procedure below to set up a JDBC connection to connect the Talend Studio Job to Amazon Athena.

Procedure
1. In the Repository tree view of your Talend Studio, expand the Metadata node, right-click Db Connections and select Create connection.
   The Database connection wizard opens.
2. Provide the connection name and purpose and click Next.
3. From the **DB Type** list, select JDBC.
4. In the **JDBC URL** field, provide the JDBC URL details, including your AWS user ID and password.

   jdbc:awsathena://AwsRegion=<AWS Region>;User=<AWS Access Key>;Password=<AWS Secret Key>;S3OutputLocation=<S3 folder>

5. Install the required addition library.
   a) In the **Drivers** table, click the [+ ] button to add a line and click the [...] button to open the **Module** dialog box.
   b) Select the **Install a new module** option, click the [...] button to browse to the downloaded .jar.
   c) Click the **Detect the module install status** button, and then click **OK** to close the dialog box.
6. Back in the **Database connection** wizard, click the **Select class name** button and select the main class of the driver allowing to communicate with Amazon Athena.

7. Fill in your database user authentication data in the **User Id** and **Password** fields.

8. Click **Test connection** and then **Finish** to close the wizard.

9. In the **Repository** tree view, right-click the JDBC connection you just created and select **Retrieve Schema** to retrieve table schema of the connected Amazon Athena service.

**Setting up a Job to connect to Amazon Athena**

**Procedure**

1. Create a Talend standard Job, drop two components to its design workspace: a **tDBInput** and a **tLogRow**, and label them based on your needs.

2. Double-click the **tDBInput** component, select **JDBC** from the **Database** list, and click **Apply**.

3. Connect the **tDBInput** and **tLogRow** components using a **Row > Main** connection.
4. In the Repository tree view, expand the JDBC connection you just created, select the table schema of interest, and drop it onto the tDBInput component. The connection information, table name, and schema are automatically set.

5. Click Guess Query to generate the query clause, and edit it based on your needs.

6. In the Basic settings view of the tLogRow component, click Sync columns to make sure it has the identical schema with the input components, and select the Table (each is key/value list) option for better readability of the console display.

7. Press F6 to run your Job.

Results
The Run console displays the data read from the Amazon Athena service.
Scheduling and executing a Talend Job on the Google Kubernetes Engine

To enable server-less execution, Talend allows developers to publish Jobs as Docker images. The containerized Jobs can then be executed in a Kubernetes engine.

This example shows how to:

- publish a Talend Job as a docker image to a Docker Registry
- schedule the Docker image to execute the Job on a Google Kubernetes Engine (GKE) cluster.

Prerequisites to this example

To publish a Job as an Docker image and execute it on the Google Kubernetes Engine, you need to have:

- A Talend Platform Studio or Talend Data Fabric Studio, version 7.1.1 or newer, installed on the client/local machine
- A Docker registry account
- A licensed Google Cloud Platform (GCP) account.

Publishing your Job as a Docker image

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Follow the steps below to publish a Job as a Docker image.

Procedure

1. Open Talend Studio and create a simple Standard Job like below, and make sure it runs successfully.

2. Right-click the Job in the Repository tree view, and select Publish from the contextual menu.
3. From the Export Type, select Docker Image and click Next.
4. Select **Local** as the Docker host, and provide the following details together with your docker registry credentials.

- **Image name**: A name for the docker image. A repository will be created with the same name in the Docker registry.
- **Image tag**: A value to tag your Docker image. By default, the Job version will be the tag value.
- **Registry**: Provide the Docker registry URL with your Docker ID. For Docker Hub, it will be hub.docker.com/dockerId.
- **Username** and **Password**: The credentials you use to connect to Docker registry.
5. Log in to your Docker registry account and under the Repositories tab. You can see the Docker image containing the Job created.
Configuring Google Kubernetes Engine

Google Kubernetes Engine (GKE) is a management and orchestration system for Docker container and container clusters that run within the Google’s public cloud services.

Procedure

1. Log in to your GCP environment.
2. Select the Kubernetes Engine from the Google Services list.
3. Click the Create Cluster button and provide the required configuration for your Kubernetes Engine.

For more information on the configuration details, see https://cloud.google.com/kubernetes-engine/docs/how-to/creating-a-cluster.

Once the cluster is created, the page looks like below:

4. Click the Connect button on the right to connect to this cluster.
   A window like below pops up.
Connect to the cluster

You can connect to your cluster via command-line or using a dashboard.

**Command-line access**

Configure `kubectl` command line access by running the following command:

```
$ gcloud container clusters get-credentials demo-cluster --zone us-central1-a --project customer-success-166522
```

**Run in Cloud Shell**

**Cloud Console dashboard**

You can view the workloads running in your cluster in the Cloud Console Workloads dashboard.

**Open Workloads dashboard**

5. Click **Run in Cloud Shell**.

Google Cloud Shell opens with a command. Press **Enter** to execute the command.

You have connected to the Cluster now.

**Scheduling the Job**

**Procedure**

1. Click the **Workloads** tab on the left side of the **Kubernetes Engine** page, and click the **Activate Cloud Shell** on the top.
2. Create the Secret with the name `regcred` by executing the following command.

```bash
kubectl create secret docker-registry regcred --docker-server=<docker-registry-server> --docker-username=<dockerId> --docker-password=<docker-password> --docker-email=<your-email>
```

Kubernetes cluster uses the Secret feature of docker-registry type to authenticate and pull a private image. This Secret can be created in the Cloud Shell. For more information about Secret, see [https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/](https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/).

3. Create a `config.yml` file on your local workstation with the following details. Make sure you do not change the alignment for the parameters.

```yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: talend-standard-job
spec:
  schedule: "*/3 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: kubernetes-job
              image: dockerId/standard_job:latest
              restartPolicy: Never
              imagePullSecrets:
                - name: regcred
              backoffLimit: 4
```

The schedule parameter defines when and how often a Job runs in crontab. It follows Unix standards and all the CronJob times are in UTC. There are 5 fields in the schedule parameter. To know how to set up a schedule for the CronJob, see [https://cloud.google.com/kubernetes-engine/docs/how-to/cronjobs](https://cloud.google.com/kubernetes-engine/docs/how-to/cronjobs).

4. Click the **Settings** button for the shell window as shown below and click **Upload file** to choose and upload the created `config.yml` file.
5. After uploading the file, run the following command in the cloud shell.

   ```bash
   kubectl create -f config.yml
   ```

   This creates a Cron Job named `talend-standard-job` like below.

6. Once the Cron Job is created, click it to see the following details.
Scheduling and executing a Talend Job on the Google Kubernetes Engine

This Job is scheduled to run every 3 minutes.

To see the Job execution logs, click the Container logs on the Cron Job details page.

To delete the Cron Job, click it and on the Cron Job details page, click KUBECTL > Delete. This will open a Cloud Shell with a command in it. Press Enter to delete the Job.
Scheduling and executing a Talend Job on the Google Kubernetes Engine

### Google Cloud Platform

#### Google Kubernetes Engine

### One Job details

- **Name**: Standard
- **Labels**: None

### CPU Usage

<table>
<thead>
<tr>
<th>Time</th>
<th>CPU Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan 6, 2019 2:54 PM</td>
<td>1.5%</td>
</tr>
<tr>
<td>Jan 6, 2019 3:25 PM</td>
<td>3.2%</td>
</tr>
</tbody>
</table>

### Memory Usage

<table>
<thead>
<tr>
<th>Time</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan 6, 2019 2:54 PM</td>
<td>16.0 MB</td>
</tr>
<tr>
<td>Jan 6, 2019 3:25 PM</td>
<td>32.0 MB</td>
</tr>
</tbody>
</table>

### Disk Usage

<table>
<thead>
<tr>
<th>Time</th>
<th>Disk Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan 6, 2019 2:54 PM</td>
<td>No data</td>
</tr>
<tr>
<td>Jan 6, 2019 3:25 PM</td>
<td>No data</td>
</tr>
</tbody>
</table>

### Schedule

- **Schedule Type**: Daily
- **Last Schedule**: Jan 6, 2019 2:00 PM
- **Duration**: 1 hour

### Workload

- **Workload Type**: None
- **Labels**: None